pylinac Documentation

Release 3.18.0

James

Dec 19, 2023

INTRODUCTION

Contents

Pylinac provides TG-142 quality assurance (QA) tools to Python programmers in the field of therapy and diagnostic medical physics.

Pylinac contains high-level modules for automatically analyzing images and data generated by linear accelerators, CT simulators, and other radiation oncology equipment. Most scripts can be utilized with less than 10 lines of code.

The library also contains lower-level modules & tools for creating your own image analysis algorithms.

The major attributes of the package are:

- Simple, concise image analysis API
- Automatic analysis of imaging and performance metrics like MTF, Contrast, ROIs, etc.
- PDF report generation for solid documentation
- Automatic phantom registration even if you don't set up your phantom perfect
- Image loading from file, ZIP archives, or URLs

CHAPTER

ONE

INSTALLATION

Install via `pip`:

```
$ pip install pylinac
```

See the Installation page for further details.

CHAPTER

TWO

CITATION

You may cite the pylinac library in publications; see the paper in the Journal of Open Source Software. The citation string is:

Kerns, J. R., (2023). Pylinac: Image analysis for routine quality assurance in radiotherapy. Journal of Open Source Software, 8(92), 6001, https://doi.org/10.21105/joss.06001

And the BibTeX entry:

```
@article{Kerns2023, doi = {10.21105/joss.06001}, url = {https://doi.org/10.21105/joss.
↪06001}, year = {2023}, publisher = {The Open Journal}, volume = {8}, number = {92},␣
↪pages = {6001}, author = {James R. Kerns}, title = {Pylinac: Image analysis for␣
↪routine quality assurance in radiotherapy}, journal = {Journal of Open Source Software}
↪ }
```

CHAPTER

THREE

DOCUMENTATION

To get started, install the package, run the demos, view the API docs, and learn the module design, visit the Full Documentation on Read The Docs.

CHAPTER

FOUR

FEATURES

4.1 Low-level Tooling

4.1.1 DICOM, XIM & Image Loading

Load DICOM files, XIM, and TIFF images:

```
from pylinac import image
from pylinac.core.image import XIM

my_dcm = image.load("path/to/my/image.dcm")
my_dcm.metadata.GantryAngle  # the GantryAngle tag of the DICOM file
# these won't have the metadata property as they aren't DICOM
my_tiff = image.load("path/to/my/image.tiff")
my_jpg = image.load("path/to/my/image.jpg")

my_xim_file = r"C:\TDS\H12345\QA\image.xim"
xim_img = XIM(my_xim_file)

# plot the image
xim_img.plot()

# see the XIM properties
print(xim_img.properties)
```

Read more about DICOM and pixel loading: Image Loading. Read more about XIM images: XIM Images.

4.1.2 Image Manipulation

Images can be manipulated in a variety of ways. This is helpful when combined with the loading utilities above:

```
from pylinac import image

# load an image
my_img = image.load("path/to/my/image.dcm")

# rotate the image
my_img.rotate(90)

# flip the image
```

(continues on next page)

(continued from previous page)

```
my_img.flipud()

# crop the image
my_img.crop(pixels=50, edges=("left", "top"))

# invert the image
my_img.bit_invert()

# normalize the array (max value = 1)
my_img.normalize()

# plot the image
my_img.plot()

# save the image back out to DICOM
my_img.save("path/to/new.dcm")
```

Convert TIFF to DICOM:

```
from pylinac import image

# load the TIFF image
new_dicom = image.tiff_to_dicom(
    "path/to/my/image.tiff", sid=1000, gantry=90, coll=0, couch=0, dpi=400
)

# save out the FILE to DICOM
new_dicom.save("path/to/new.dcm")
```

4.1.3 Compute Gamma

Compute gamma between two arrays:

```
from pylinac import image

# load the images
img1 = image.load("path/to/image1.dcm")
img2 = image.load("path/to/image2.dcm")

# compute gamma
gamma = image.gamma_2d(
    reference=img1.array,
    evaluation=img2.array,
    dose_to_agreement=1,
    distance_to_agreement=1,
    gamma_cap_value=2,
    global_dose=True,
    dose_threshold=5,
)

# plot the gamma map
```

(continues on next page)

(continued from previous page)

```
plt.imshow(gamma)
```

Compute gamma for 1D profiles:

```
from pylinac import profile

# load the images and profiles
img1 = image.load("path/to/image1.dcm")
img2 = image.load("path/to/image2.dcm")
mid_img1_profile = img1.array[img1.shape[0] // 2, :]
mid_img2_profile = img2.array[img2.shape[0] // 2, :]

# compute gamma
gamma = profile.gamma_1d(
    reference=mid_img1_profile,
    evaluation=mid_img2_profile,
    dose_to_agreement=1,
    distance_to_agreement=1,
    gamma_cap_value=2,
    global_dose=True,
    dose_threshold=5,
)

# plot the gamma map
plt.plot(gamma)
```

4.1.4 Compute Custom Metrics on Profiles

Pylinac comes with several built-in metrics that can be computed on 1D profiles, each of which can be configured.

Writing new metrics is also easy.

- Left Penumbra using `LeftPenumbralMetric`
- Right Penumbra using `RightPenumbralMetric`
- FFF "Top" using `TopPosition`
- Flatness (Difference) using `FlatnessDifferenceMetric`
- Flatness (Ratio) using `FlatnessRatioMetric`
- Symmetry (Point Difference) using `SymmetryPointDifferenceMetric`
- Symmetry (Point Difference Quotient) using `SymmetryPointDifferenceQuotientMetric`

Calculate the penumbra of a profile using the built-in `LeftPenumbraMetric`:

```
from pylinac import profile
from pylinac.metrics.profile import LeftPenumbraMetric

# load the image and profile
img = image.load("path/to/image.dcm")
mid_profile = FWXMProfile(img.array[img.shape[0] // 2, :])
```

(continues on next page)

(continued from previous page)

```
# compute the penumbra
left_penumbra = mid_profile.compute(metrics=[LeftPenumbraMetric(upper=80, lower=20)])

print(left_penumbra)  # prints the penumbra value
```

Read more about 1D metrics: Profiles & 1D Metrics.

4.1.5 Convert Gantry, Collimator, Couch Coordinate Systems

Convert gantry, collimator, and couch coordinates to and from each other:

```
from pylinac.core.scale import convert, MachineScale

gantry = 0
coll = 90
couch = 45

new_gantry, new_coll, new_couch = convert(
    input_scale=MachineScale.Varian,
    output_scale=MachineScale.IEC61217,
    gantry=gantry,
    collimator=coll,
    rotation=couch,
)
```

Read more: Coordinate Systems.

4.1.6 Generate Synthetic Images

Want to generate images to test out your image analysis algorithms? Pylinac can do that.

Generate an AS1000 50x50mm, centered open field image at gantry 45:

```
from matplotlib import pyplot as plt

from pylinac.core.image_generator import AS1000Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1000 = AS1000Image()  # this will set the pixel size and shape automatically
as1000.add_layer(
    FilteredFieldLayer(field_size_mm=(50, 50))
)  # create a 50x50mm square field
as1000.add_layer(
    GaussianFilterLayer(sigma_mm=2)
)  # add an image-wide gaussian to simulate penumbra/scatter
as1000.generate_dicom(
    file_out_name="my_AS1000.dcm", gantry_angle=45
)  # create a DICOM file with the simulated image
# plot the generated image
plt.imshow(as1000.image)
```

4.2 TG-51

TG-51 & TRS-398 Absolute Dose Calibration - Input the raw data and pylinac can calculate either individual values (kQ, PDDx, Pion, etc) or use the provided classes to input all measurement data and have it calculate all factors and dose values automatically.

Example script:

```
from pylinac import tg51, trs398

ENERGY = 6
TEMP = 22.1
PRESS = tg51.mmHg2kPa(755.0)
CHAMBER = "30013"  # PTW
P_ELEC = 1.000
ND_w = 5.443  # Gy/nC
MU = 200
CLINICAL_PDD = 66.5

tg51_6x = tg51.TG51Photon(
    unit="TrueBeam1",
    chamber=CHAMBER,
    temp=TEMP,
    press=PRESS,
    n_dw=ND_w,
    p_elec=P_ELEC,
    measured_pdd10=66.4,
    lead_foil=None,
    clinical_pdd10=66.5,
    energy=ENERGY,
    voltage_reference=-300,
    voltage_reduced=-150,
    m_reference=(25.65, 25.66, 25.65),
    m_opposite=(25.64, 25.65, 25.65),
    m_reduced=(25.64, 25.63, 25.63),
    mu=MU,
    tissue_correction=1.0,
)

# Done!
print(tg51_6x.dose_mu_dmax)

# examine other parameters
print(tg51_6x.pddx)
print(tg51_6x.kq)
print(tg51_6x.p_ion)

# change readings if you adjust output
tg51_6x.m_reference_adjusted = (25.44, 25.44, 25.43)
# print new dose value
print(tg51_6x.dose_mu_dmax_adjusted)

# generate a PDF for record-keeping
```

(continues on next page)

(continued from previous page)

```
tg51_6x.publish_pdf(
    "TB1 6MV TG-51.pdf",
    notes=["My notes", "I used Pylinac to do this; so easy!"],
    open_file=False,
)

# TRS-398 is very similar and just as easy!
```

4.3 Planar Phantom Analysis

Planar Phantom Analysis (Leeds TOR, StandardImaging QC-3 & QC-kV, Las Vegas, Doselab MC2 (kV & MV), SNC kV & MV, PTW EPID QC) - Features:

- **Automatic phantom localization** - Set up your phantom any way you like; automatic positioning, angle, and inversion correction mean you can set up how you like, nor will setup variations give you headache.
- **High and low contrast determination** - Analyze both low and high contrast ROIs. Set thresholds as you see fit.

Example script:

```
from pylinac import LeedsTOR, StandardImagingQC3, LasVegas, DoselabMC2kV, DoselabMC2MV

leeds = LeedsTOR("my_leeds.dcm")
leeds.analyze()
leeds.plot_analyzed_image()
leeds.publish_pdf()

qc3 = StandardImagingQC3("my_qc3.dcm")
qc3.analyze()
qc3.plot_analyzed_image()
qc3.publish_pdf("qc3.pdf")

lv = LasVegas("my_lv.dcm")
lv.analyze()
lv.plot_analyzed_image()
lv.publish_pdf("lv.pdf", open_file=True)  # open the PDF after publishing

...
```

4.4 Winston-Lutz Analysis

Winston-Lutz Analysis - The Winston-Lutz module analyzes EPID images taken of a small radiation field and BB to determine the 2D distance from BB to field CAX. Additionally, the isocenter size of the gantry, collimator, and couch can all be determined *without the BB being at isocenter*. Analysis is based on Winkler et al , Du et al, and Low et al.

Features:

- **Couch shift instructions** - After running a WL test, get immediate feedback on how to shift the couch. Couch values can also be passed in and the new couch values will be presented so you don't have to do that pesky conversion. "Do I subtract that number or add it?"

- **Automatic field & BB positioning** - When an image or directory is loaded, the field CAX and the BB are automatically found, along with the vector and scalar distance between them.
- **Isocenter size determination** - Using backprojections of the EPID images, the 3D gantry isocenter size and position can be determined *independent of the BB position*. Additionally, the 2D planar isocenter size of the collimator and couch can also be determined.
- **Image plotting** - WL images can be plotted separately or together, each of which shows the field CAX, BB and scalar distance from BB to CAX.
- **Axis deviation plots** - Plot the variation of the gantry, collimator, couch, and EPID in each plane as well as RMS variation.
- **File name interpretation** - Rename DICOM filenames to include axis information for linacs that don't include such information in the DICOM tags. E.g. "myWL_gantry45_coll0_couch315.dcm".

Example script:

```
from pylinac import WinstonLutz

wl = WinstonLutz("wl/image/directory")  # images are analyzed upon loading
wl.plot_summary()
print(wl.results())
wl.publish_pdf("my_wl.pdf")
```

4.5 Starshot Analysis

Starshot Analysis - The Starshot module analyses a starshot image made of radiation spokes, whether gantry, collimator, MLC or couch. It is based on ideas from Depuydt et al and Gonzalez et al.

Features:

- **Analyze scanned film images, single EPID images, or a set of EPID images** - Any image that you can load in can be analyzed, including 1 or a set of EPID DICOM images and films that have been digitally scanned.
- **Any image size** - Have machines with different EPIDs? Scanned your film at different resolutions? No problem.
- **Dose/OD can be inverted** - Whether your device/image views dose as an increase in value or a decrease, pylinac will detect it and invert if necessary.
- **Automatic noise detection & correction** - Sometimes there's dirt on the scanned film; sometimes there's a dead pixel on the EPID. Pylinac will detect these spurious noise signals and can avoid or account for them.
- **Accurate, FWHM star line detection** - Pylinac uses not simply the maximum value to find the center of a star line, but analyzes the entire star profile to determine the center of the FWHM, ensuring small noise or maximum value bias is avoided.
- **Adaptive searching** - If you passed pylinac a set of parameters and a good result wasn't found, pylinac can recover and do an adaptive search by adjusting parameters to find a "reasonable" wobble.

Example script:

```
from pylinac import Starshot

star = Starshot("mystarshot.tif")
star.analyze(radius=0.75, tolerance=1.0, fwhm=True)
print(star.results())  # prints out wobble information
```

(continues on next page)

(continued from previous page)

```
star.plot_analyzed_image()  # shows a matplotlib figure
star.publish_pdf()  # publish a PDF report
```

4.6 VMAT Analysis

VMAT QA - The VMAT module consists of two classes: `DRGS` and `DRMLC`, which are capable of loading an EPID DICOM Open field image and MLC field image and analyzing the images according to the Varian RapidArc QA tests and procedures, specifically the Dose-Rate & Gantry-Speed (DRGS) and MLC speed (MLCS) tests.

Features:

- **Do both tests** - Pylinac can handle either DRGS or DRMLC tests.
- **Adjust for offsets** - Older VMAT patterns were off-center. Pylinac will find the field regardless.

Example script:

```
from pylinac import DRGS, DRMLC

drgs = DRGS(image_paths=["path/to/DRGSopen.dcm", "path/to/DRGSdmlc.dcm"])
drgs.analyze(tolerance=1.5)
print(drgs.results())  # prints out ROI information
drgs.plot_analyzed_image()  # shows a matplotlib figure
drgs.publish_pdf("mydrgs.pdf")  # generate a PDF report
```

4.7 CatPhan, Quart, ACR, Cheese Phantom Analysis

CatPhan, Quart, ACR phantom QA - The CBCT module automatically analyzes DICOM images of a CatPhan 504, 503, 600, 604, Quart DVT, and ACR CT/MR acquired when doing CT, CBCT, or MR quality assurance. It can load a folder or zip file that the images are in and automatically correct for phantom setup in 6 axes. CatPhans analyze the HU regions and image scaling (CTP404), the high-contrast line pairs (CTP528) to calculate the modulation transfer function (MTF), and the HU uniformity (CTP486) on the corresponding slice. Quart and ACR analyze similar metrics where possible.

Features:

- **Automatic phantom registration** - Your phantom can be tilted, rotated, or translated–pylinac will register the phantom.
- **Automatic testing of all major modules** - Major modules are automatically registered and analyzed.
- **Any scan protocol** - Scan your CatPhan with any protocol; or even scan it in a regular CT scanner. Any field size or field extent is allowed.
- **Customize modules** - You can easily override settings in the event you have a custom scenario such as a partial scan.

Example script:

```
from pylinac import (
    CatPhan504,
    CatPhan503,
    CatPhan600,
```

(continues on next page)

(continued from previous page)

```
    CatPhan604,
    QuartDVT,
    ACRCT,
    ACRMRILarge,
)

# for this example, we'll use the CatPhan504
cbct = CatPhan504("my/cbct_image_folder")
cbct.analyze(
    hu_tolerance=40,
    scaling_tolerance=1,
    thickness_tolerance=0.2,
    low_contrast_threshold=1,
)
print(cbct.results())
cbct.plot_analyzed_image()
cbct.publish_pdf("mycbct.pdf")
```

4.8 Trajectory & Dynalog Analysis

Log Analysis - The log analyzer module reads and parses Varian linear accelerator machine logs, both Dynalogs and Trajectory logs. The module also calculates actual and expected fluences as well as performing gamma evaluations. Data is structured to be easily accessible and easily plottable.

Unlike most other modules of pylinac, the log analyzer module has no end goal. Data is parsed from the logs, but what is done with that info, and which info is analyzed is up to the user.

Features:

- **Analyze Dynalogs or Trajectory logs** - Either platform is supported. Tlog versions 2.1 and 3.0 supported.
- **Save Trajectory log data to CSV** - The Trajectory log binary data format does not allow for easy export of data. Pylinac lets you do that so you can use Excel or other software that you use with Dynalogs.
- **Plot or analyze any axis** - Every data axis can be plotted: the actual, expected, and even the difference.
- **View actual or expected fluences & calculate gamma** - View fluences and gamma maps for any log.
- **Anonymization** - Anonymize your logs so you can share them with others.

Example script:

```
from pylinac import load_log

tlog = load_log("tlog.bin")
# after loading, explore any Axis of the Varian structure
tlog.axis_data.gantry.plot_actual()  # plot the gantry position throughout treatment
tlog.fluence.gamma.calc_map(doseTA=1, distTA=1, threshold=10, resolution=0.1)
tlog.fluence.gamma.plot_map()  # show the gamma map as a matplotlib figure
tlog.publish_pdf()  # publish a PDF report

dlog = load_log("dynalog.dlg")
...
```

4.9 Picket Fence MLC Analysis

Picket Fence MLC Analysis - The picket fence module is meant for analyzing EPID images where a "picket fence" MLC pattern has been made. Physicists regularly check MLC positioning through this test. This test can be done using film and one can "eyeball" it, but this is the 21st century and we have numerous ways of quantifying such data. This module attains to be one of them. It will load in an EPID dicom image and determine the MLC peaks, error of each MLC pair to the picket, and give a few visual indicators for passing/warning/failing.

Features:

- **Preset & customizable MLC configurations** - Standard configurations are built-in and you can create your own configuration of leaves if needed.
- **Easy-to-read pass/warn/fail overlay** - Analysis gives you easy-to-read tools for determining the status of an MLC pair.
- **Any Source-to-Image distance** - Whatever your clinic uses as the SID for picket fence, pylinac can account for it.
- **Account for panel translation** - Have an off-CAX setup? No problem. Translate your EPID and pylinac knows.
- **Account for panel sag** - If your EPID sags at certain angles, just tell pylinac and the results will be shifted.

Example script:

```
from pylinac import PicketFence

pf = PicketFence("mypf.dcm")
pf.analyze(tolerance=0.5, action_tolerance=0.25)
print(pf.results())
pf.plot_analyzed_image()
pf.publish_pdf()
```

4.10 Open Field Analysis

Open Field Analysis - Field analysis from a digital image such as EPID DICOM or 2D device array can easily be analyzed. The module contains built-in flatness and symmetry equation definitions but is extensible to quickly create custom F&S equations.

Features: * **EPID or device data** - Any EPID image or the SNC Profiler. * **Built-in F&S equations** - The common Elekta, Varian, and Siemens definitions are included * **Extensible equations** - Adding custom equations for image metrics are easy

Example script:

```
from pylinac import FieldAnalysis, DeviceFieldAnalysis, Protocol

fa = FieldAnalysis(path="myFS.dcm")  # equivalently, DeviceFieldAnalysis
fa.analyze(protocol=Protocol.VARIAN)
# print results
print(fa.results())
# get results as a dict
fa.results_data()
# plot results
fa.plot_analyzed_image()
```

(continues on next page)

(continued from previous page)

```
# publish a PDF file
fa.publish_pdf(filename="my field analysis.pdf")
```

CHAPTER

FIVE

DISCUSSION

Have questions? Ask them on the pylinac discourse server.

CHAPTER

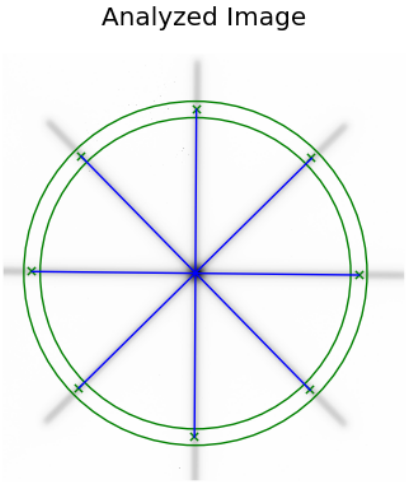
SIX

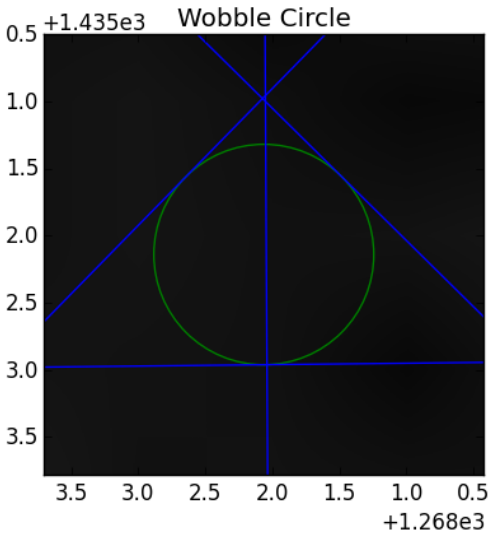
CONTRIBUTING

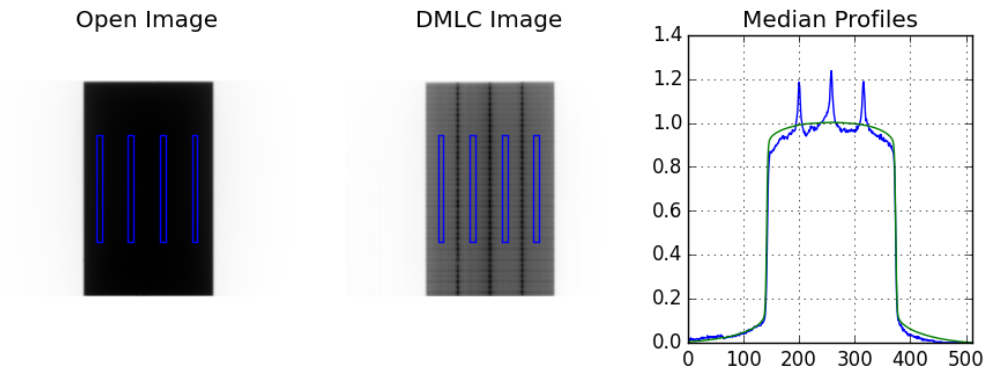
Contributions to pylinac can be many. The most useful things a non-programmer can contribute are images to analyze and bug reports. If you have VMAT images, starshot images, machine log files, CBCT DICOM files, or anything else you want analyzed, upload them privately here.

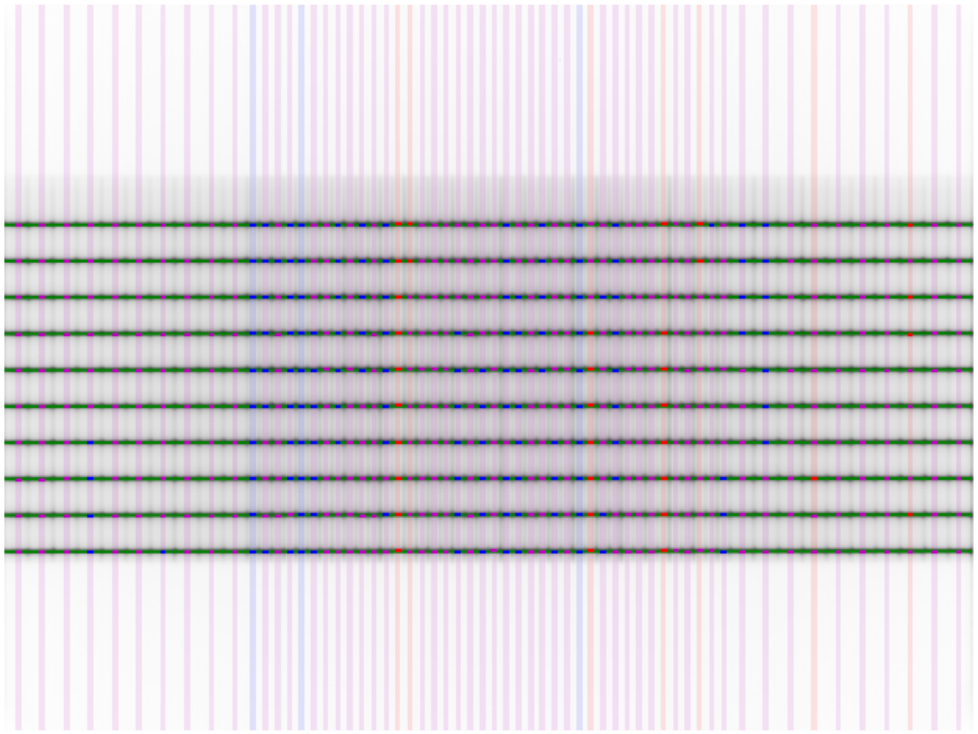
See the full Contributing page for more details.

6.1 Pylinac General Overview









6.1.1 What is pylinac?

Pylinac (pr. "pie-linac") is a Python library to analyze the images and datasets commonly used by therapy medical physicists in the course of their routine QA. These data usually follow tests outlined by AAPM TG-142 and similar TGs. An example would be the "picket fence" test of MLCs done weekly or monthly.

Pylinac will take in the image/data and can output numeric results, plots showing the analysis, or a PDF including both the numerical and plot data.

- Pylinac consumes raw data/images to compute meaningful output related to common physics tests
- Pylinac consumes raw data/images to present meaningful data types for further exploration

6.1.2 What is pylinac NOT?

- Pylinac is not a database. Data outputs should be placed onto your clinic's data repository. You should use a high-quality specialty application like QAtrack+, an open source application taylor-made (bah-dum-ch) for routine physics QA.
- Pylinac is not liable for incorrect outputs. Either by inputting incorrect data, or the algorithm being incorrect, you should **always** validate pylinac against a known methodology (automatic or manual), just as you should do for any software used in your clinic.
- Pylinac is not commercial software. It is open-source with a very lenient MIT license which means it can be used publicly, privately, or even used in commercial applications royalty-free. In fact, it's the image analysis engine for RadMachine, where both the pylinac and QATrack+ authors currently work. Join us!

6.1.3 Intended Audience

Pylinac is intended to be used by physicists who know at least a bit of programming, although just the basics should be enough for most applications.

6.1.4 Philosophy

Pylinac runs on a few philosophical principles:

- A given module should only address 1 overarching task.
- Using pylinac should require a minimal amount of code.
- The user should have to supply as little information as necessary to run an analysis.
- The underlying code of pylinac should be easy to understand.
- Although evaluation against a reference or baseline may occur, it is not the end-goal. As stated above, there are better places for containing reference information. Pylinac's goal moving forward is to provide top-tier analysis and results. How those results are evaluated are more suited for other applications such as QATrack+ and RadMachine.

6.1.5 Algorithm Design Overview

Generally speaking, the design of algorithms should all follow the same guidelines and appear as similar as possible. Each module will outline its own specific algorithm in its documentation.

- Descriptions of algorithms are sorted into steps of the following:
  - **Allowances** – These describe what the pylinac algorithm *can* account for.
  - **Restrictions** – These are the things pylinac *cannot* do and must be addressed before the module can be properly used.
  - **Pre-Analysis** – Algorithm steps that prepare for the main algorithm sequence.
  - **Analysis** – The steps pylinac takes to analyze the image or data.
  - **Post-Analysis** – What pylinac does or can do after analysis, like showing the data or checking against tolerances.
- Algorithm steps should be expressible in a word or short phrase.
- Algorithm method names should be as similar as possible from module to module.

> The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code – not in reams of trivial code that bores the reader to death.
>
> —Guido van Rossum

6.1.6 Module Design

Pylinac has a handful of modules, but most of them work somewhat the same, so here we describe the general patterns you'll see when using pylinac.

- **Each module has its own demonstration method(s)** – If you don't yet have an image or data and want to see how a module works you can run and inspect the code of the demo to get an idea. Most demo methods have a name like or starts with `.run_demo()`.
- **Each module has its own demo image/dataset(s)** – Want to test the analysis but are having trouble with your image? Use the provided demo images. All major classes have a demo image or dataset and are usually similar to `.from_demo_image()`.
- **Each module has similar load, analyze, and show methods and behavior** – The normal flow of a pylinac module use is to 1) Load the data in, 2) Analyze the data, and 3) Show the results.
- **Most modules can be fully utilized in a few lines** – The whole point of pylinac is to automate and simplify the process of analyzing routine QA data. Thus, most routines can be written in a few lines.

6.1.7 Relation to RadMachine

Pylinac is the image analysis engine for RadMachine. The primary author (James Kerns) is currently employed at Radformation. Pylinac has and will continue to be open-source, even as it's used in RadMachine. This is a win-win for the community because the algorithms are visible to all, both RadMachine customers and the community, and the overall community benefits from advances Radformation makes, such as new phantom analyses that RadMachine customers request. There is no intention to close-source pylinac now or in the future. That being said, pylinac remains focused on image analysis and being a utility library. It is unlikely that, for example, pylinac will expand to include a GUI, add database functionality, or data monitoring since that overlaps with QATrack+/RadMachine.

6.2 Installation

Installing pylinac is easy no matter your skill! Determine where you're at and then read the relevant section:

6.2.1 I know Python already

Great! To get started install via pip:

```
$ pip install pylinac
```

Note: Installing from source (`setup.py install`) is possible but not recommended as downloading the source includes numerous sizable test files.

6.2.2 Dependencies

Pylinac, as a scientific package, has fairly standard scientific dependencies (>= means at least that version or newer). Installing the package via `pip` will install these for you:

```
argue
matplotlib >= 2.0
numpy >= 1.16
Pillow >= 4.0
py-linq
pydicom >= 2.0
reportlab >= 3.3
scikit-image >= 0.17
scipy >= 1.1
tabulate~=0.9.0
tqdm >= 3.8
```

6.2.3 I'm new to Python

That's okay! If you're not a programmer at all you'll have a few things to do to get up and running, but never fear. Using pylinac requires not just the base language Python, but a few dependencies as well. Since most physicists don't program, or if they do it's in MATLAB, this section will help jumpstart your use of not just pylinac but Python in general and all its wonderful goodness! Getting started with Python takes some work to get set up and running, but it's well worth the effort.

Get a Distribution Stack

Scientific computing with Python requires some specialized packages which require some specialized computing libraries. While it's possible you have those libraries (for some odd reason), it's not likely. Thus, it's often best to install the libraries *pre-compiled*. There are several options out there; I'll list just a few. Be sure to download the 3.x version, preferably the newest:

- Anaconda - Continuum Analytics provides this one-stop-shop for tons of scientific libraries in an easy to install format. Just download and run the installer. If you don't want to install all 200+ packages, a slimmer option exists: Miniconda, which only installs `conda` and python installation tools. You can then use conda to install packages individually. Here's the Anaconda quick start guide.

  Note: Unlike the other options, individual packages can be upgraded on demand using the `conda` tool.

- WinPython - (Windows only) This grassroots project functions similarly to Anaconda, where all packages are precompiled and run out of the box. There are no corporate sponsors for this project, so support is not guaranteed.

See Scipy's Installation Options for more options.

Warning: Python(x,y) is not yet available for Python 3, so don't choose this to try running pylinac.

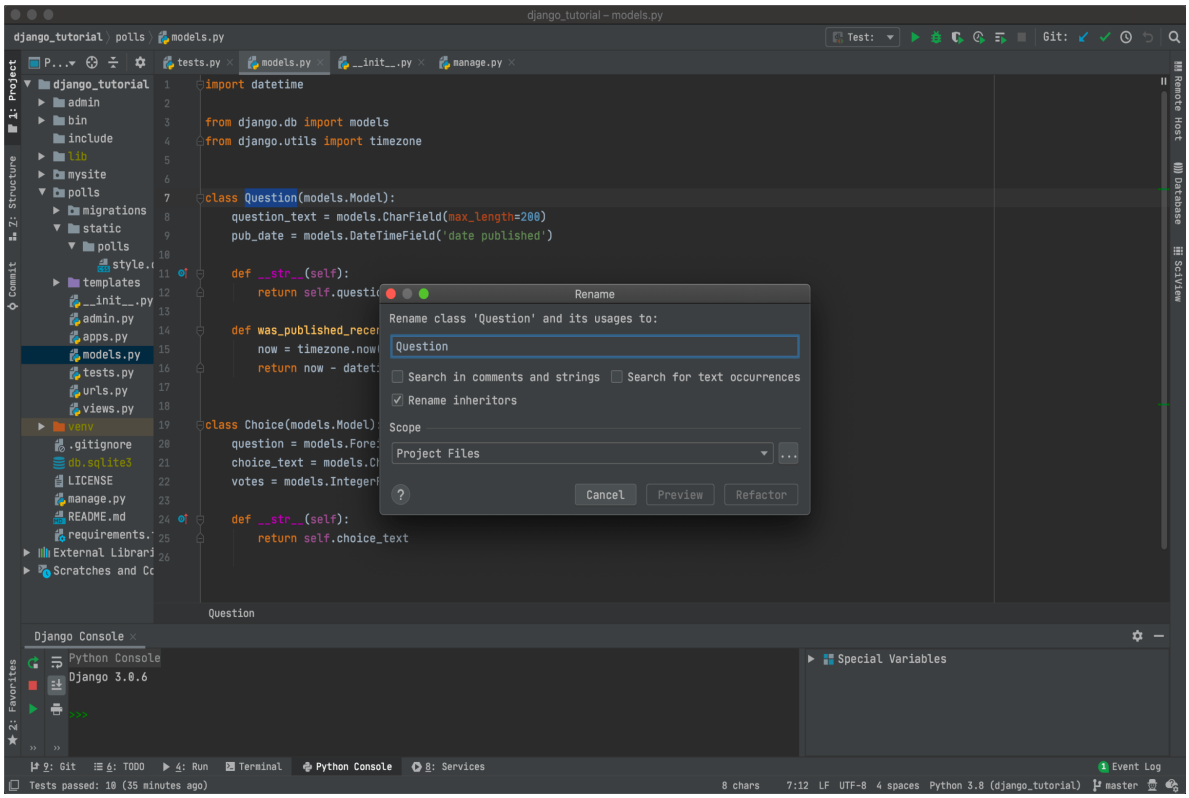
Note: If this is the first/only Python distribution you'll be using it'd be a good idea to activate it when the installer prompts you.

Note: You can install multiple Python stacks/versions, but only one is "active" at any given time.

Get an IDE (optional)

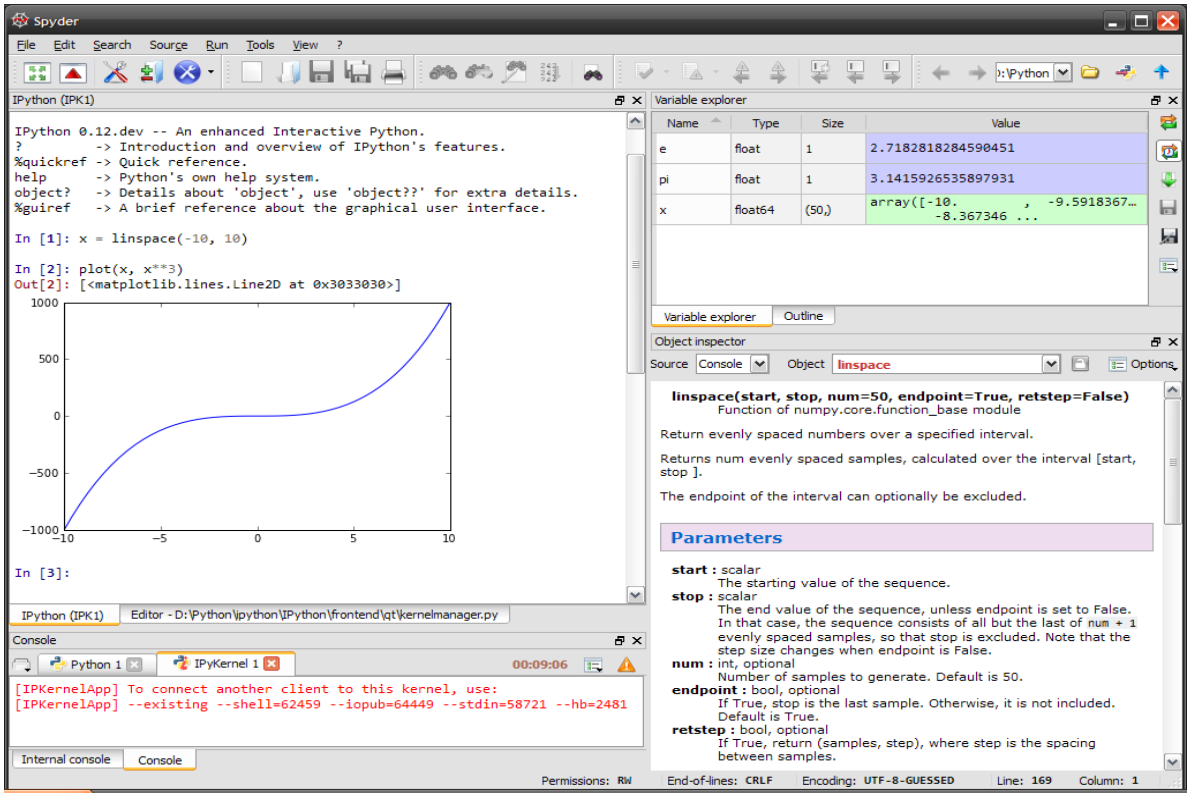
If you come from MATLAB, it's helpful to realize that MATLAB is both a language and an Integrated Development Environment (IDE). Most languages don't have an official IDE, and some people may tell you IDEs are a crutch. If being a cyborg with superpowers is a crutch, then call me a cripple because I find them extremely useful. As with all power, it must be wielded carefully though. The option of getting an IDE is completely up to you. If you want one, here are some options:

- PyCharm - A fully-featured, rich IDE. It's arguably king of the heavyweights and *free*. At least try it. Here's the PyCharm quick start guide.



- Spyder - A MATLAB-like IDE with similar layout, preferred by many working in the scientific realm. Here are the Spyder docs.

Note: Spyder is part of the Anaconda distribution.



6.3 General Tips

Using pylinac is easy! Once installed, you can write your own script in a matter of minutes. Each module of pylinac addresses the topic of its name (e.g. the *Starshot* class, surprisingly, performs starshot analysis). Furthermore, each module is designed as similarly as possible to one another. So once you start using one module, it's easy to use another (see *Module Design*). Each module also has its own demonstration method and data to show off what it can do.

6.3.1 Running a Demo

Let's get started by running a demo of the `Starshot` module. First, import the Starshot class:

```
from pylinac import Starshot
```

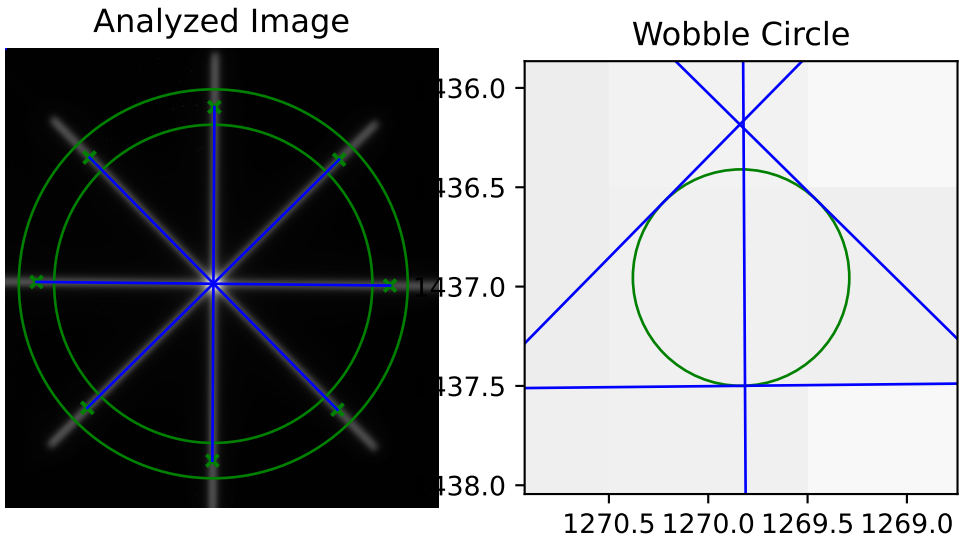
This class has all the capabilities of loading and analyzing a Starshot image. Let's 1) create an instance of that class and then 2) run its demonstration method:

```
from pylinac import Starshot

Starshot.run_demo()
```

Running this should result in a printing of information to the console and an image showing the analyzed image, like so:

```
Result: PASS
```

(continues on next page)


Analyzed Image
Wobble Circle
436.0
436.5
437.0
437.5
438.0
1270.5
1270.0
1269.5
1269.0

(continued from previous page)

```
The minimum circle that touches all the star lines has a diameter of 0.434 mm.

The center of the minimum circle is at 1270.1, 1437.1
```

Congratulations! In 3 lines you've successfully used a pylinac module. Of course there's more to it than that; you'll want to analyze your own images. For further documentation on starshots, see *Starshot*.

6.3.2 Loading in Images/Data

All modules have multiple ways of loading in your data. The best way to use a given module's main class is instantiating with the image/data file name. If you have something else (e.g. a URL or set of multiple images) you can use the class-based constructors that always start with `from_`. Let's use the `log_analyzer` module to demonstrate:

```
from pylinac import TrajectoryLog
```

We can pass the path to the log, and this would be the standard way of constructing:

```
log = TrajectoryLog(r"C:/John/QA/log.dlg")
```

Perhaps the data is stored online somewhere. You can load in the data from a URL:

```
log = TrajectoryLog.from_url("https://myserver.com/logs/log23.bin")
```

If for any reason you don't have data and want to experiment, you can easily load in demo data:

```
tlog = TrajectoryLog.from_demo()
```

You can find out more about logs in the *Log Analyzer*. All modules are similar however; the main class can be instantiated directly, through class-based constructors, from a URL, and all main classes have a demo dataset and demo method.

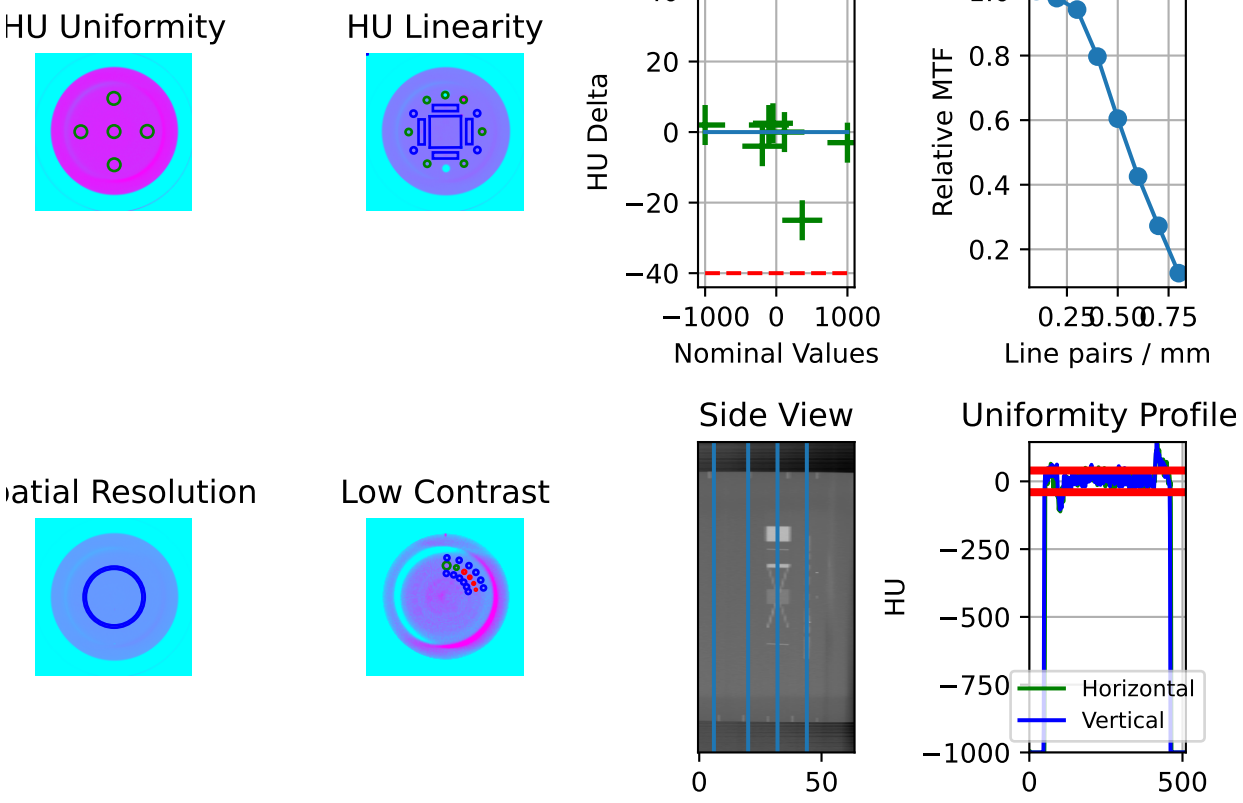
6.3.3 Changing Colormaps

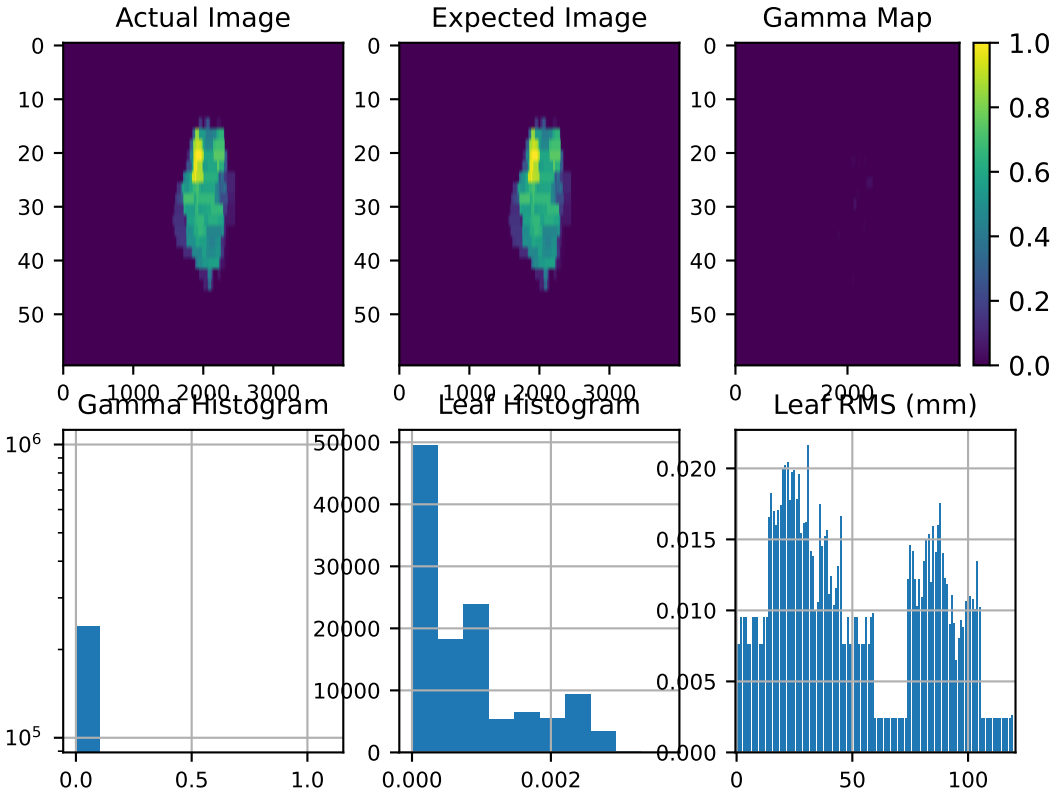
The colormaps in pylinac are pretty standard. By default, DICOM images are shown in grayscale, while most other arrays are shown in jet. Changing these is easy though. All that's required is to pass a valid matplotlib colormap (see options here). Let's set the DICOM plots to be 'cool':

```
import pylinac
# change the colormap setting
pylinac.settings.DICOM_COLORMAP = 'cool'
pylinac.CatPhan504.run_demo()
```

We can also change other arrays, for example the arrays in the `log_analyzer` module. Let's change it to the newer, better matplotlib default colormap, viridis:

```
import matplotlib.pyplot as plt
import pylinac

pylinac.settings.ARRAY_COLORMAP = plt.cm.viridis
pylinac.TrajectoryLog.run_demo()
```


HU Uniformity
HU Linearity
HU linearity
HU Delta
40
20
0
−20
−40
−1000
0
1000
Nominal Values
RMTF
Relative MTF
1.0
0.8
0.6
0.4
0.2
0.25
0.50
0.75
Line pairs / mm
atial Resolution
Low Contrast
Side View
0
50
Uniformity Profile
HU
0
−250
−500
−750
−1000
Horizontal
Vertical
0
500


Actual Image
Expected Image
Gamma Map
Gamma Histogram
Leaf Histogram
Leaf RMS (mm)

6.4 Calibration (TG-51/TRS-398)

6.4.1 Overview

The calibration module actually consists of two submodules: `tg51` and `trs398`, each addressing their respective protocol. Both modules contain functions and classes for calculation the protocol dose. The modules have some overlap, especially with basic functions as well as helper functions. The modules have tried to use the vocabulary of the respective protocol, but occasionally there are differences when we felt that using the same name was clearer. See the vocabulary section for full definitions.

Note: Besides the typical calculations one would expect, the modules also include helper functions, such as a PDD to TPR converter so that a TG-51 calculation can determine kQ from TPR and avoid the tedious PDDx. Additionally, pressure unit converters exist to go from the various units of pressure to kPa which is what pylinac uses.

6.4.2 Vocabulary that may be different than the protocol

- `voltage_reference`: Used in both TG-51 and TRS-398 for the voltage used when taking a reference reading; commonly -300V.
- `voltage_reduced`: Use in both TG-51 and TRS-398 for the lower voltage used to determine ks/Pion; commonly -150V.
- `m_reference`: The ion chamber reading at the reference voltage.
- `m_reduced`: The ion chamber reading at the reduced voltage.
- `m_opposite`: The ion chamber reading at the opposite polarity of the reference voltage: commonly +300V.

Vocabulary not listed here should be the same as the respective protocol.

6.4.3 Changing a bound

Bounds are placed in the module to prevent catastrophic errors from passing in the wrong values; e.g. the wrong units. If you live in a place that has extreme temperatures or pressures or just otherwise want to change the default bounds, you can change the default range of acceptable values. E.g. to change the minimum allowable temperature that can be passed:

```
from pylinac import tg51

tg51.p_tp(temp=5, press=100)  # will raise bounds error

# override
tg51.MIN_TEMP = 0

tg51.p_tp(temp=5, press=100)  # no bounds error will be raised
```

You can override the min/max of temp, pressure, p ion, p elec, p tp, p pol. These bounds are the same for TRS-398. I.e. setting these in either module will set them for both modules.

6.4.4 TG-51

Equation Definitions

Equation definitions are as follows:

- Ptp (Temp/Pressure correction) - TG-51 Eqn. 10:

$$\frac{273.2+T}{273.2+22} * \frac{101.33}{P}$$

> **Warning:** Temperature is in Celsius and pressure is in kPa. Use the helper functions *fahrenheit2celsius()*, *mmHg2kPa()*, and *mbar2kPa()* as needed.

- Ppol (Polarity correction) - Rather than using TG-51 Eqn. 9, we opt instead for TRS-398 Eqn xx, which uses the absolute values of the positive and negative voltages. This is the same results as Eqn. 9 but without worrying about signs.:

$$\frac{|M_{raw}^{+}| + |M_{raw}^{-}|}{2 * M_{raw}}$$

- Pion (Ion collection correction; only for pulsed beams) - TG-51 Eqn. 12:

$$\frac{1 - \frac{V_H}{V_L}}{\frac{M_{raw}^H}{M_{raw}^L} - \frac{V_H}{V_L}}$$

- Dref (Reference electron depth; cm) - TG-51 Eqn. 18:

$$0.6 * R_{50} - 0.1$$

- R50 (Beam quality specifier; 50% dose depth; cm) - TG-51 Eqn. 16 & 17:

$$\begin{cases} 1.029 * I_{50} - 0.06(cm) & 2 \leq I_{50} \leq 10 \\ 1.059 * I_{50} - 0.37(cm) & I_{50}10 \end{cases}$$

- k'R50 (k'R50 for cylindrical chambers) - TG-51 Eqn. 19:

$$0.9905 + 0.0710e^{\frac{-R_{50}}{3.67}}$$

- PQ_gr (PQ gradient correction for cylindrical chambers) - TG-51 Eqn. 21:

$$\frac{M_{raw}(d_{ref} + 0.5 * r_{cav})}{M_{raw} * d_{ref}}$$

- PDDx (PDD from photons only) - TG-51 Eqns. 13, 14 & 15:

$$\begin{cases} PDD(10) & energy < 10 \\ PDD(10) & nolead, energy >= 10, PDD(10) < 75 \\ 1.267 * PDD(10) - 20.0 & nolead, 75 \leq PDD(10) \leq 89 \\ PDD(10)_{Pb} & lead@50cm, PDD(10)_{Pb} < 73 \\ (0.8905 + 0.00150 * PDD(10)_{Pb}) * PDD(10)_{Pb} & lead@50cm, PDD(10)_{Pb} \geq 73 \\ PDD(10)_{Pb} & lead@30cm, PDD(10)_{Pb} < 71 \\ (0.8116 + 0.00264 * PDD(10)_{Pb}) * PDD(10)_{Pb} & lead@30cm, PDD(10)_{Pb} \geq 71 \end{cases}$$

- M-corrected (corrected chamber reading) - TG-51 Eqn. 8:

$$P_{ion} * P_{TP} * P_{elec} * P_{pol} * M_{raw}$$

- kQ for Photons (cylindrical chamber-specific quality conversion factor) - TG-51 Addendum Eqn 1 & Table I:

$$\Big\{A + B * 10^-3 * PDD(10)x + C * 10^-5 * (PDD(10)x)^2 \quad 63 < PDD(10)x < 86$$

  Where A, B, and C are chamber-specific fitting values as given in Table I. Pylinac automatically retrieves values based on the chamber model passed to the function.

- kQ for Electrons (cylindrical chamber-specific quality conversion factor) - Muir & Rogers 2014

  The study of Muir & Rogers was to find kecal values that could be determined soley from R50. Through Monte Carlo experiments, the optimal Pgradient was determined as well as fitting parameters for numerous common ion chambers. That study eliminates the need for Pgradient measurements. These kecal values will very likely be incorporated into the next TG-51 addendum (as has their kQ values for photons in the first addendum). From the paper, we can start with the known relationship given in Eqn. 9:

$$k_Q = k_{Q,ecal} * k'_Q$$

  where Eqn. 11 states:

$$k'_Q = a + b * R_{50}^{-c}$$

  Where a, b, and c are chamber-specific fitting values as given in Table VII and where $k_{Q,ecal}$ is given in Table VI.

- D_w^Q photon (Dose to water at 10cm from a photon beam of quality Q - TG-51 Eqn. 3:

$$M * k_Q * N_{D,w}^{60Co}(Gy)$$

- D_w^Q electron (Dose to water at 10cm from an electron beam of quality Q - TG-51 Eqn. 6:

$$M * P_{gr}^Q * k'_{R_{50}} * k_{ecal} * N_{D,w}^{60Co}(Gy)$$

Function-based Use

Using the TG-51 module can be complementary to your existing workflow, or completely replace it. For example, you could use the kQ function to calculate kQ and then calculate the other corrections and values yourself. If you want something a little more complete, you can use the `TG51Photon`, `TG51ElectronLegacy` and `TG51ElectronModern` classes which will calculate all necessary corrections and values.

Note: The Photon class uses kQ values from the TG-51 addendum. The Legacy Electron class will make the user specify a kecal value and measure Pgradient. The Modern Electron class will calculate kQ completely from R50 and the chamber from Muir & Rogers 2014 paper, no kecal or Pgradient needed.

```
"""A script to calculate TG-51 dose using pylinac functions and following the TG-51␣
↪photon form"""
from pylinac.calibration import tg51

ENERGY = 6
```

(continues on next page)

(continued from previous page)

```
TEMP = 22.1
PRESS = tg51.mmHg2kPa(755.0)
CHAMBER = "30013"  # PTW
P_ELEC = 1.000
ND_w = 5.443  # Gy/nC
MU = 200
CLINICAL_PDD = 66.5

# Section 4 (beam quality)
# since energy is 6MV, PDDx == PDD, but we'll run it through anyway just for show
pdd10x = tg51.pddx(pdd=66.4, energy=ENERGY)

# Section 5 (kQ)
kq = tg51.kq_photon_pddx(chamber=CHAMBER, pddx=pdd10x)
# Alternatively, get kQ from TPR (way quicker to measure, without needing to measure TPR!
↪)
tpr = tg51.tpr2010_from_pdd2010(pdd2010=(38.0 / 66.4))
kq = tg51.kq_photon_tpr(chamber=CHAMBER, tpr=tpr)

# Section 6 (Temp/Press)
p_tp = tg51.p_tp(temp=TEMP, press=PRESS)

# Section 7 (polarity)
m_reference = (25.66, 25.67, 25.66)
m_opposite = (25.67, 25.67, 25.68)
p_pol = tg51.p_pol(m_reference=m_reference, m_opposite=m_opposite)

# Section 8 (ionization)
m_reduced = (25.61, 25.62)
p_ion = tg51.p_ion(
    voltage_reference=300,
    voltage_reduced=150,
    m_reference=m_reference,
    m_reduced=m_reduced,
)

# Section 9 (M corrected)
m_corr = tg51.m_corrected(
    p_ion=p_ion, p_tp=p_tp, p_elec=P_ELEC, p_pol=p_pol, m_reference=m_reference
)

# Section 10 (dose to water @ 10cm)
dose_10 = m_corr * kq * ND_w
dose_10_per_mu = dose_10 / MU

# Section 11 (dose/MU to water @ dmax)
dose_ddmax = dose_10_per_mu / CLINICAL_PDD

# Done!
print(dose_ddmax)
```

Class-based Use

```
"""A script to calculate TG-51 dose using pylinac classes and following the TG-51 photon␣
↪form"""
from pylinac.calibration import tg51

ENERGY = 6
TEMP = 22.1
PRESS = tg51.mmHg2kPa(755.0)
CHAMBER = "30013"  # PTW
P_ELEC = 1.000
ND_w = 5.443  # Gy/nC
MU = 200
CLINICAL_PDD = 66.5

tg51_6x = tg51.TG51Photon(
    unit="TrueBeam1",
    chamber=CHAMBER,
    temp=TEMP,
    press=PRESS,
    n_dw=ND_w,
    p_elec=P_ELEC,
    measured_pdd10=66.4,
    lead_foil=None,
    clinical_pdd10=66.5,
    energy=ENERGY,
    voltage_reference=-300,
    voltage_reduced=-150,
    m_reference=(25.65, 25.66, 25.65),
    m_opposite=(25.64, 25.65, 25.65),
    m_reduced=(25.64, 25.63, 25.63),
    mu=MU,
    tissue_correction=1.0,
)

# Done!
print(tg51_6x.dose_mu_dmax)

# examine other parameters
print(tg51_6x.pddx)
print(tg51_6x.kq)
print(tg51_6x.p_ion)

# change readings if you adjust output
tg51_6x.m_reference_adjusted = (25.44, 25.44, 25.43)
# print new dose value
print(tg51_6x.dose_mu_dmax_adjusted)

# generate a PDF for record-keeping
tg51_6x.publish_pdf(
    "TB1 6MV TG-51.pdf",
    notes=["My notes", "I used Pylinac to do this; so easy!"],
    open_file=False,
```

(continues on next page)

(continued from previous page)

```
)
```

6.4.5 TRS-398

Warning: Pylinac does not calculate electron dose in any other conditions than water; i.e. no solid water.

Equation Definitions

- Ktp (Temp/Pressure correction):

$$\frac{273.2+T}{273.2+22} * \frac{101.33}{P}$$

Warning: Temperature is in Celsius and pressure is in kPa. Use the helper functions fahrenheit2celsius, mmHg2kPa, and mbar2kPa as needed.

- Kpol (Polarity correction):

$$\frac{|M_{raw}^{+}| + |M_{raw}^{-}|}{2 * M_{raw}}$$

- Ks (Ion collection correction; only for pulsed beams):

$$a_0 + a_1 * (\frac{M_1}{M_2}) + a_2 * (\frac{M_1}{M_2})^2$$

- Zref (Reference electron depth; cm) - TRS-398 7.2:

$$0.6 * R_{50} - 0.1$$

- R50 (Beam quality specifier; 50% dose depth; cm) - TRS-398 7.1:

$$\begin{cases} 1.029 * I_{50} - 0.06(cm) & 2 \leq I_{50} \leq 10 \\ 1.059 * I_{50} - 0.37(cm) & I_{50}10 \end{cases}$$

- D_w^Q photon (Dose to water at Zref from a photon or electron beam of quality Q - TRS-398 7.3:

$$D_{w,Q} = M_Q * N_{D,w,Qo} * k_{Q,Qo}(Gy)$$

- M-corrected (corrected chamber reading):

$$M_Q = k_s * k_{TP} * K_{elec} * K_{pol} * M_1$$

- kQ,Qo for Photons (cylindrical chamber-specific quality conversion factor): TRS-398 Table 6.III
- kQ for Electrons (cylindrical chamber-specific quality conversion factor; calibrated in Co-60): TRS-398 Table 7.III

Function-based Use

```
"""A script to calculate TRS-398 dose using pylinac functions and following the TRS-398␣
↪photon form"""
from pylinac.calibration import trs398

TEMP = 22.1
PRESS = trs398.mmHg2kPa(755.0)
CHAMBER = "30013"  # PTW
K_ELEC = 1.000
ND_w = 5.443  # Gy/nC
MU = 200


# Section 3 (dosimeter corrections)
k_tp = trs398.k_tp(temp=TEMP, press=PRESS)
k_pol = trs398.k_pol(
    m_reference=(25.66, 25.67, 25.66), m_opposite=(25.65, 25.66, 25.66)
)
k_s = trs398.k_s(
    voltage_reference=300,
    voltage_reduced=150,
    m_reference=(25.66, 25.67, 25.66),
    m_reduced=(25.63, 25.65, 25.64),
)
m_corrected = (
    trs398.m_corrected(
        m_reference=(25.66, 25.67, 25.66),
        k_tp=k_tp,
        k_elec=K_ELEC,
        k_pol=k_pol,
        k_s=k_s,
    )
    / MU
)

# Section 4 (kQ + dose at zref)
kq = trs398.kq_photon(chamber=CHAMBER, tpr=(39.2 / 68.1))
dose_mu_zref = m_corrected * ND_w * kq

# Section 5 (Dose at zmax)
# SSD setup
CLINICAL_PDD = 66.5
dose_mu_zmax = dose_mu_zref * 100 / CLINICAL_PDD

# SAD setup
CLINICAL_TMR = 0.666
dose_mu_zmax = dose_mu_zref / CLINICAL_TMR

# Done!
print(dose_mu_zmax)
```

Class-based Use

```
"""A script to calculate TRS-398 dose using pylinac classes and following the TRS-398␣
↪photon form"""
from pylinac.calibration import trs398

ENERGY = 6
TEMP = 22.1
PRESS = trs398.mmHg2kPa(755.0)
CHAMBER = "30013"  # PTW
K_ELEC = 1.000
ND_w = 5.443  # Gy/nC
MU = 200
CLINICAL_PDD = 66.5

trs398_6x = trs398.TRS398Photon(
    unit="TrueBeam1",
    setup="SSD",
    chamber=CHAMBER,
    temp=TEMP,
    press=PRESS,
    n_dw=ND_w,
    clinical_pdd_zref=CLINICAL_PDD,
    tpr2010=(38.2 / 66.6),
    energy=ENERGY,
    fff=False,
    k_elec=K_ELEC,
    voltage_reference=-300,
    voltage_reduced=-150,
    m_reference=(25.65, 25.66, 25.65),
    m_opposite=(25.64, 25.65, 25.65),
    m_reduced=(25.64, 25.63, 25.63),
    mu=MU,
    tissue_correction=1.0,
)

# Done!
print(trs398_6x.dose_mu_zmax)

# examine other parameters
print(trs398_6x.kq)
print(trs398_6x.k_s)
print(trs398_6x.k_tp)

# change readings if you adjust output
trs398_6x.m_reference_adjusted = (25.44, 25.44, 25.43)
# print new dose value
print(trs398_6x.dose_mu_zmax_adjusted)

# generate a PDF for record-keeping
trs398_6x.publish_pdf(
    "TB1 6MV TRS-398.pdf",
    notes=["My notes", "I used Pylinac to do this; so easy!"],
```

(continues on next page)

(continued from previous page)

```
    open_file=False,
)
```

6.4.6 TG-51 API Documentation

pylinac.calibration.tg51.**mmHg2kPa**(*mmHg: float*) → float

Utility function to convert from mmHg to kPa.

pylinac.calibration.tg51.**mbar2kPa**(*mbar: float*) → float

Utility function to convert from millibars to kPa.

pylinac.calibration.tg51.**fahrenheit2celsius**(*f: float*) → float

Utility function to convert from Fahrenheit to Celsius.

pylinac.calibration.tg51.**tpr2010_from_pdd2010**(*, *pdd2010: float*) → float

Calculate TPR20,10 from PDD20,10. From TRS-398 footnote 25, section 6.3.1, p.68 (https://www-pub.iaea.org/MTCD/Publications/PDF/TRS398_scr.pdf), and Followill et al 1998 eqn 1.

pylinac.calibration.tg51.**p_tp**(*, *temp: float*, *press: float*) → float

Calculate the temperature & pressure correction.

Parameters

temp
: [float (17-27)] The temperature in degrees Celsius.

press
: [float (91-111)] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

pylinac.calibration.tg51.**p_pol**(*, *m_reference: float | list | tuple | ndarray*, *m_opposite: float | list | tuple | ndarray*) → float

Calculate the polarity correction.

Parameters

m_reference
: [number, array] The readings of the ion chamber at the reference polarity and voltage.

m_opposite
: [number, array] The readings of the ion chamber at the polarity opposite the reference. The sign does not make a difference.

Raises

BoundsError if calculated Ppol is >1% from 1.0.

pylinac.calibration.tg51.**p_ion**(*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: float | list | tuple | ndarray*, *m_reduced: float | list | tuple | ndarray*) → float

Calculate the ion chamber collection correction.

Parameters

voltage_reference
[int] The "high" voltage; same as the TG51 measurement voltage.

voltage_reduced
[int] The "low" voltage; usually half of the high voltage.

m_reference
[float, iterable] The readings of the ion chamber at the "high" voltage.

m_reduced
[float, iterable] The readings of the ion chamber at the "low" voltage.

Raises

BoundsError if calculated Pion is outside the range 1.00-1.05.

pylinac.calibration.tg51.**d_ref**(*, *i_50: float*) → float

Calculate the dref of an electron beam based on the I50 depth.

Parameters

i_50
[float] The value of I50 in cm.

pylinac.calibration.tg51.**r_50**(*, *i_50: float*) → float

Calculate the R50 depth of an electron beam based on the I50 depth.

Parameters

i_50
[float] The value of I50 in cm.

pylinac.calibration.tg51.**kp_r50**(*, *r_50: float*) → float

Calculate k'R50 for Farmer-like chambers.

Parameters

r_50
[float (2-9)] The R50 value in cm.

pylinac.calibration.tg51.**pq_gr**(*, *m_dref_plus: float | list | tuple | ndarray*, *m_dref: float | list | tuple | ndarray*) → float

Calculate PQ_gradient for a cylindrical chamber.

Parameters

m_dref_plus
[float, iterable] The readings of the ion chamber at dref + 0.5rcav.

m_dref
[float, iterable] The readings of the ion chamber at dref.

pylinac.calibration.tg51.**m_corrected**(*, *p_ion: float*, *p_tp: float*, *p_elec: float*, *p_pol: float*, *m_reference: float | list | tuple | ndarray*) → float

Calculate M_corrected, the ion chamber reading with all corrections applied.

Parameters

p_ion
[float (1.00-1.05)] The ion collection correction.

p_tp
[float (0.92-1.08)] The temperature & pressure correction.

p_elec
[float (0.98-1.02)] The electrometer correction.

p_pol
[float (0.98-1.02)] The polarity correction.

m_reference
[float, iterable] The raw ion chamber reading(s).

Returns

float

pylinac.calibration.tg51.**pddx**(*, *pdd: float*, *energy: int*, *lead_foil: str | None = None*) → float

Calculate PDDx based on the PDD.

Parameters

pdd
[{>62.7, <89.0}] The measured PDD. If lead foil was used, this assumes the pdd as measured with the lead in place.

energy
[int] The nominal energy in MV.

lead_foil
[{None, '30cm', '50cm'}] Applicable only for energies >10MV. Whether a lead foil was used to acquire the pdd. Use `None` if no lead foil was used and the interim equation should be used. This is the default Use `50cm` if the lead foil was set to 50cm from the phantom surface. Use `30cm` if the lead foil was set to 30cm from the phantom surface.

pylinac.calibration.tg51.**kq_photon_pddx**(*, *chamber: str*, *pddx: float*) → float

Calculate kQ based on the chamber and clinical measurements of PDD(10)x. This will calculate kQ for photons for *CYLINDRICAL* chambers only.

Parameters

chamber
[str] The chamber of the chamber. Valid values are those listed in Table III of Muir and Rogers and Table I of the TG-51 Addendum.

pddx
[{>63.0, <86.0}] The **PHOTON-ONLY** PDD measurement at 10cm depth for a 10x10cm2 field.

Note: Use the *pddx()* function to convert PDD to PDDx as needed.

Note: Muir and Rogers state limits of 0.627 - 0.861. The TG-51 addendum states them as 0.63 and 0.86. The TG-51 addendum limits are used here.

---

pylinac.calibration.tg51.**kq_photon_tpr**(*, *chamber: str*, *tpr: float*) → float

Calculate kQ based on the chamber and clinical measurements of TPR20,10. This will calculate kQ for photons for *CYLINDRICAL* chambers only.

Parameters

chamber
[str] The chamber of the chamber. Valid values are those listed in Table III of Muir and Rogers and Table I of the TG-51 Addendum.

tpr
[{>0.630, <0.860}] The TPR(20,10) value.

Note: Use the *tpr2010_from_pdd2010()* function to convert from PDD without needing to take TPR measurements.

---

pylinac.calibration.tg51.**kq_electron**(*, *chamber: str*, *r_50: float*) → float

Calculate kQ based on the chamber and clinical measurements. This will calculate kQ for electrons for *CYLINDRICAL* chambers only according to Muir & Rogers.

Parameters

chamber
[str] The chamber of the chamber. Valid values are those listed in Tables VI and VII of Muir and Rogers 2014.

r_50
[float] The R50 value in cm of an electron beam.

class pylinac.calibration.tg51.**TG51Photon**(*, *institution: str = ''*, *physicist: str = ''*, *unit: str*, *measurement_date: str = ''*, *temp: float*, *press: float*, *chamber: str*, *n_dw: float*, *p_elec: float*, *electrometer: str = ''*, *measured_pdd10: float | None = None*, *lead_foil: str | None = None*, *clinical_pdd10: float*, *energy: int*, *fff: bool = False*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: float | list | tuple | ndarray*, *m_opposite: float | list | tuple | ndarray*, *m_reduced: float | list | tuple | ndarray*, *mu: int*, *tissue_correction: float = 1.0*, *m_reference_adjusted: float | list | tuple | ndarray | None = None*)

Bases: TG51Base

Class for calculating absolute dose to water using a cylindrical chamber in a photon beam.

Parameters

institution
[str] Institution name.

physicist
[str] Physicist performing calibration.

unit
[str] Unit name; e.g. TrueBeam1.

measurement_date
[str] Date of measurement. E.g. 10/22/2018.

temp
[float] The temperature in Celsius. Use *fahrenheit2celsius()* to convert if necessary.

press
[float] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

energy
[float] Nominal energy of the beam in MV.

chamber
[str] Chamber model. Must be one of the listed chambers in TG-51 Addendum.

n_dw
[float] NDW value in Gy/nC.

p_elec
[float] Electrometer correction factor; given by the calibration laboratory.

measured_pdd10
[float] The measured value of PDD(10); will be converted to PDDx(10) and used for calculating kq.

lead_foil
[{None, '50cm', '30cm'}] Whether a lead foil was used to acquire PDD(10)x and where its position was. Used to calculate kq.

clinical_pdd10
[float] The PDD used to correct the dose at 10cm back to dmax. Usually the TPS PDD(10) value.

voltage_reference
[int] Reference voltage; i.e. voltage when taking the calibration measurement.

voltage_reduced
[int] Reduced voltage; usually half of the reference voltage.

m_reference
[float, tuple] Ion chamber reading(s) at the reference voltage.

m_opposite
[float, tuple] Ion chamber reading(s) at the opposite voltage of reference.

m_reduced
[float, tuple] Ion chamber reading(s) at the reduced voltage.

mu
[int] The MU delivered to measure the reference reading. E.g. 200.

fff
[bool] Whether the beam is FFF or flat.

tissue_correction
[float] Correction value to calibration to, e.g., muscle. A value of 1.0 means no correction (i.e. water).

`property pddx: float`
The photon-only PDD(10) value.

`property kq: float`
The chamber-specific beam quality correction factor.

`property dose_mu_10: float`
cGy/MU at a depth of 10cm.

`property dose_mu_dmax: float`
cGy/MU at a depth of dmax.

`property dose_mu_10_adjusted: float`
The dose/mu at 10cm depth after adjustment.

`property dose_mu_dmax_adjusted: float`
The dose/mu at dmax depth after adjustment.

`publish_pdf`(*filename: str*, *notes: list | None = None*, *open_file: bool = False*, *metadata: dict | None = None*)
Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[str, file-like object] The file to write the results to.

notes
[str, list] Any notes to be added to the report. If a string, adds everything as one line. If a list, must be a list of strings; each string item will be a new line.

open_file
[bool] Whether to open the file after creation. Will use the default PDF program.

metadata
[dict] Any data that should be appended to every page of the report. This differs from notes in that metadata is at the top of every page while notes is at the bottom of the report.

class pylinac.calibration.tg51.**TG51ElectronLegacy**(*, *institution: str = ''*, *physicist: str = ''*, *unit: str = ''*, *measurement_date: str = ''*, *energy: int*, *temp: float*, *press: float*, *chamber: str*, *k_ecal: float*, *n_dw: float*, *electrometer: str = ''*, *p_elec: float*, *clinical_pdd: float*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: float | list | tuple | ndarray*, *m_opposite: float | list | tuple | ndarray*, *m_reduced: float | list | tuple | ndarray*, *m_gradient: float | list | tuple | ndarray*, *cone: str*, *mu: int*, *i_50: float*, *tissue_correction: float = 1.0*, *m_reference_adjusted=None*)

Bases: `TG51Base`

Class for calculating absolute dose to water using a cylindrical chamber in an electron beam.

Parameters

institution
[str] Institution name.

physicist
[str] Physicist performing calibration.

unit
[str] Unit name; e.g. TrueBeam1.

measurement_date
[str] Date of measurement. E.g. 10/22/2018.

temp
[float (17-27)] The temperature in degrees Celsius.

press
[float (91-111)] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

chamber
[str] Chamber model; only for bookkeeping.

n_dw
[float] NDW value in Gy/nC. Given by the calibration laboratory.

k_ecal
[float] Kecal value which is chamber specific. This value is the major difference between the legacy class and modern class where no kecal is needed.

p_elec
[float] Electrometer correction factor; given by the calibration laboratory.

clinical_pdd
[float] The PDD used to correct the dose back to dref.

voltage_reference
[float] Reference voltage; i.e. voltage when taking the calibration measurement.

voltage_reduced
[float] Reduced voltage; usually half of the reference voltage.

m_reference
[float, tuple] Ion chamber reading(s) at the reference voltage.

m_opposite
[float, tuple] Ion chamber reading(s) at the opposite voltage of reference.

m_reduced
[float, tuple] Ion chamber reading(s) at the reduced voltage.

mu
[int] The MU delivered to measure the reference reading. E.g. 200.

i_50
[float] Depth of 50% ionization.

tissue_correction
[float] Correction value to calibration to, e.g., muscle. A value of 1.0 means no correction (i.e. water).

`property r_50: float`
Depth of the 50% dose value.

`property dref: float`
Depth of the reference point.

`property pq_gr`
Gradient factor

`property kq: float`
The kQ value using classic TG-51

`property dose_mu_dref: float`
cGy/MU at the depth of Dref.

`property dose_mu_dmax: float`
cGy/MU at the depth of dmax.

`property dose_mu_dref_adjusted: float`
cGy/MU at the depth of Dref.

`property dose_mu_dmax_adjusted: float`
cGy/MU at the depth of dmax.

`publish_pdf`(*filename: str*, *notes: list | None = None*, *open_file: bool = False*, *metadata: dict | None = None*)
Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[str, file-like object] The file to write the results to.

notes
[str, list] Any notes to be added to the report. If a string, adds everything as one line. If a list, must be a list of strings; each string item will be a new line.

open_file
[bool] Whether to open the file after creation. Will use the default PDF program.

metadata
[dict] Any data that should be appended to every page of the report. This differs from notes in that metadata is at the top of every page while notes is at the bottom of the report.

class pylinac.calibration.tg51.**TG51ElectronModern**(*, *institution: str = ''*, *physicist: str = ''*, *unit: str = ''*, *measurement_date: str = ''*, *energy: int*, *temp: float*, *press: float*, *chamber: str*, *n_dw: float*, *electrometer: str = ''*, *p_elec: float*, *clinical_pdd: float*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: float | list | tuple | ndarray*, *m_opposite: float | list | tuple | ndarray*, *m_reduced: float | list | tuple | ndarray*, *cone: str*, *mu: int*, *i_50: float*, *tissue_correction: float*, *m_reference_adjusted=None*)

Bases: `TG51Base`

Class for calculating absolute dose to water using a cylindrical chamber in an electron beam.

Warning: This class uses the values of Muir & Rogers. These values are likely to be included in the new TG-51 addendum, but are not official. The results can be up to 1% different. Physicists should use their own judgement when deciding which class to use. To use a manual kecal value, Pgradient and the classic TG-51 equations use the *TG51ElectronLegacy* class.

Parameters

institution
[str] Institution name.

physicist
[str] Physicist performing calibration.

unit
[str] Unit name; e.g. TrueBeam1.

measurement_date
[str] Date of measurement. E.g. 10/22/2018.

press
[float] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

temp
[float] The temperature in Celsius.

voltage_reference
[int] The reference voltage; i.e. the voltage for the calibration reading (e.g. 300V).

voltage_reduced
[int] The reduced voltage, usually a fraction of the reference voltage (e.g. 150V).

m_reference
[array, float] The reading(s) of the chamber at reference voltage.

m_reduced
[array, float] The reading(s) of the chamber at the reduced voltage.

m_opposite
[array, float] The reading(s) of the chamber at the opposite voltage from reference. Sign of the reading does not matter.

chamber
[str] Ion chamber model.

n_dw
[float] NDW value in Gy/nC

p_elec
[float] Electrometer correction given by the calibration laboratory.

clinical_pdd
[float] The PDD used to correct the dose back to dref.

mu
[int] MU delivered.

i_50
[float] Depth of 50% ionization

tissue_correction
[float] Correction value to calibration to, e.g., muscle. A value of 1.0 means no correction (i.e. water).

`property r_50: float`
Depth of the 50% dose value.

`property dref: float`
Depth of the reference point.

`property kq: float`
The kQ value using the updated Muir & Rogers values from their 2014 paper, equation 11, or classically if kecal is passed.

`property dose_mu_dref: float`
cGy/MU at the depth of Dref.

`property dose_mu_dmax: float`
cGy/MU at the depth of dmax.

`property dose_mu_dref_adjusted: float`
cGy/MU at the depth of Dref.

`property dose_mu_dmax_adjusted: float`
cGy/MU at the depth of dmax.

publish_pdf(*filename: str*, *notes: list | None = None*, *open_file: bool = False*, *metadata: dict | None = None*)

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[str, file-like object] The file to write the results to.

notes
[str, list] Any notes to be added to the report. If a string, adds everything as one line. If a list, must be a list of strings; each string item will be a new line.

open_file
[bool] Whether to open the file after creation. Will use the default PDF program.

metadata
[dict] Any data that should be appended to every page of the report. This differs from notes in that metadata is at the top of every page while notes is at the bottom of the report.

6.4.7 TRS-398 API Documentation

pylinac.calibration.trs398.**k_s**(*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: float | list | tuple | ndarray*, *m_reduced: float | list | tuple | ndarray*) → float

Calculate the ion recombination effect using readings at two voltages. The voltages should have a ratio of 2, 2.5, 3, 3.5, 4, or 5.

Parameters

voltage_reference
[int] The voltage at which calibration will be performed (e.g. 300V)

voltage_reduced
[int] The voltage which is lower than reference (e.g. 150V)

m_reference
[array, float] The reading(s) at the reference voltage.

m_reduced
[array, float] The reading(s) at the reduced voltage.

Returns

k_s
[float] The ion recombination factor.

Raises

ValueError
If the voltage ratio is not valid.

ValueError
If the calculated ks value is outside the range (1.0, 1.05).

pylinac.calibration.trs398.**kq_photon**(*, *chamber: str*, *tpr: float*) → float

Calculate the kQ factor for a photon beam given the chamber model and TPR20/10 using Table 6.III. Linear interpolation is used between given TPR ratios.

Parameters

chamber
[str] Allowable chambers are those listed in Table 6.III that are also Farmer-type (e.g. Exradin A14 Farmer).

tpr
[float] The ratio of measured TPR(20cm) / TPR(10cm). Note that this can also be calculated from PDD. See *tpr2010_from_pdd2010()*.

Returns

kQ
[float] The calculated kQ given table Table 6.III

Raises

KeyError
If the passed chamber is not within the acceptable list.

ValueError
If the TPR is not within the range defined by Table 6.III

pylinac.calibration.trs398.**kq_electron**(*, *chamber: str*, *r_50: float*) → float

Calculate the kQ factor for an electron beam given the chamber model and R50 using Table 7.III. Linear interpolation is used between given R50 values.

Parameters

chamber
[str] The Farmer-type chambers listed in Table 7.III (e.g. PTW 30004/30012).

r_50
[float] The depth of R50 in cm in water.

Returns

kQ
[float] The calculated kQ from Table 7.III

Raises

KeyError
If the passed chamber is not within the acceptable list.

ValueError
If the R50 is not within the range defined by Table 7.III

pylinac.calibration.trs398.**m_corrected**(*, *m_reference*, *k_tp*, *k_elec*, *k_pol*, *k_s*) → float

The fully corrected chamber reading.

Parameters

m_reference
[array, float] The chamber reading(s) at the calibration position.

k_tp
[float] Temperature/Pressure correction. See *p_tp()*.

k_elec
[float] Electrometer correction; given by the calibration laboratory.

k_pol
[float] Polarity correction. See *p_pol()*.

k_s
[float] Ion recombination correction. See *k_s()*.

Returns

m
[float] The fully corrected chamber reading.

class pylinac.calibration.trs398.**TRS398Photon**(*, *institution: str = ''*, *physicist: str = ''*, *unit: str = ''*, *measurement_date: str = ''*, *electrometer: str = ''*, *setup: str*, *chamber: str*, *n_dw: float*, *mu: int*, *tpr2010: float*, *energy: int*, *fff: bool*, *press: float*, *temp: float*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: tuple | float*, *m_reduced: tuple | float*, *m_opposite: tuple | float*, *k_elec: float*, *clinical_pdd_zref: float | None = None*, *clinical_tmr_zref: float | None = None*, *tissue_correction: float = 1.0*)

Bases: TRS398Base

Calculation of dose to water at zmax and zref from a high energy photon beam. Setup can be SSD or SAD.

Parameters

setup
[{'SSD', 'SAD'}] The physical setup of the calibration.

institution
[str] Institution name.

physicist
[str] Physicist performing calibration.

unit
[str] Unit name; e.g. TrueBeam1.

measurement_date
[str] Date of measurement. E.g. 10/22/2018.

chamber
[str] Farmer-type chamber model from Table 6.III.

n_dw
[float] NDw of the chamber given by the calibration laboratory.

mu
[float, int] The number of MU given per reading

energy
[int] Nominal energy of the linac in MV; e.g. 6. Bookkeeping only.

fff
[bool] Whether the beam is FFF or flat. Bookkeeping only.

tpr2010
[float] The value of TPR(20)/TPR(10). Can be derived from PDD; see *tpr2010_from_pdd2010()*.

press
[float] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

temp
[float] The temperature in Celsius.

voltage_reference
[int] The reference voltage; i.e. the voltage for the calibration reading (e.g. 300V).

voltage_reduced
[int] The reduced voltage, usually a fraction of the reference voltage (e.g. 150V).

m_reference
[array, float] The reading(s) of the chamber at reference voltage.

m_reduced
[array, float] The reading(s) of the chamber at the reduced voltage.

m_opposite
[array, float] The reading(s) of the chamber at the opposite voltage from reference. Sign of the reading does not matter.

k_elec
[float] The electrometer correction value given by the calibration laboratory.

clinical_pdd_zref
[optional, float] The PDD at the depth of calibration. Use the actual percentage (e.g. 66.7 not 0.667). If not supplied the clinical_tmr_zref value must be supplied.

clinical_tmr_zref
[optional, float] The TMR at the depth of calibration. If not supplied the clinical_pdd_zref value must be supplied.

tissue_correction
[float] The correction of calibration to a medium other than water. Default value is 1 which is water. E.g. use 0.99 if calibrating to muscle.

`property kq`
kQ of the chamber and TPR.

`property dose_mu_zmax`
cGy/MU at a depth of zmax.

`property dose_mu_zmax_adjusted`
The dose/mu at dmax depth after adjustment.

`publish_pdf`(*filename: str*, *notes: list | None = None*, *open_file: bool = False*, *metadata: dict | None = None*)
Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[str, file-like object] The file to write the results to.

notes
[str, list] Any notes to be added to the report. If a string, adds everything as one line. If a list, must be a list of strings; each string item will be a new line.

open_file
[bool] Whether to open the file after creation. Will use the default PDF program.

metadata
[dict] Any data that should be appended to every page of the report. This differs from notes in that metadata is at the top of every page while notes is at the bottom of the report.

`class pylinac.calibration.trs398.TRS398Electron`(*, *institution: str = ''*, *physicist: str = ''*, *unit: str = ''*, *measurement_date: str = ''*, *electrometer: str = ''*, *energy: str*, *cone: str*, *chamber: str*, *n_dw: float*, *mu: int*, *i_50: float*, *press: float*, *temp: float*, *voltage_reference: int*, *voltage_reduced: int*, *m_reference: tuple*, *m_reduced: tuple*, *m_opposite: tuple*, *k_elec: float*, *clinical_pdd_zref: float*, *tissue_correction: float = 1.0*)
Bases: `TRS398Base`

Calculation of dose to water at zmax and zref from a high energy electron beam.

Parameters

institution
[str] Institution name.

physicist
[str] Physicist performing calibration.

unit
[str] Unit name; e.g. TrueBeam1.

measurement_date
[str] Date of measurement. E.g. 10/22/2018.

chamber
[str] Farmer-type chamber model from Table 6.III.

n_dw
[float] NDw of the chamber given by the calibration laboratory.

mu
[float, int] The number of MU given per reading.

i_50
[float] The depth of ionization 50% in cm.

press
[float] The value of pressure in kPa. Can be converted from mmHg and mbar; see *mmHg2kPa()* and *mbar2kPa()*.

temp
[float] The temperature in Celsius.

voltage_reference
[int] The reference voltage; i.e. the voltage for the calibration reading (e.g. 300V).

voltage_reduced
[int] The reduced voltage, usually a fraction of the reference voltage (e.g. 150V).

m_reference
[array, float] The reading(s) of the chamber at reference voltage.

m_reduced
[array, float] The reading(s) of the chamber at the reduced voltage.

m_opposite
[array, float] The reading(s) of the chamber at the opposite voltage from reference. Sign of the reading does not matter.

k_elec
[float] The electrometer correction value given by the calibration laboratory.

pdd_zref
[optional, float] The PDD at the depth of calibration. Use the actual percentage (e.g. 66.7 not 0.667). If not supplied the tmr_zref value should be supplied.

tissue_correction
[float] The correction of calibration to a medium other than water. Default value is 1 which is water. E.g. use 0.99 if calibrating to muscle.

property r_50: float

The depth of R50 in cm, derived from I50.

property zref: float

Depth of the reference point.

property kq

kQ given the chamber and R50.

property dose_mu_zmax

cGy/MU at a depth of zmax.

property dose_mu_zmax_adjusted

The dose/mu at dmax depth after adjustment.

publish_pdf(*filename: str*, *notes: list | None = None*, *open_file: bool = False*, *metadata: dict | None = None*)

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[str, file-like object] The file to write the results to.

notes
[str, list] Any notes to be added to the report. If a string, adds everything as one line. If a list, must be a list of strings; each string item will be a new line.

open_file
[bool] Whether to open the file after creation. Will use the default PDF program.

metadata
[dict] Any data that should be appended to every page of the report. This differs from notes in that metadata is at the top of every page while notes is at the bottom of the report.

6.5 Starshot

6.5.1 Overview

The Starshot module analyses a starshot image made of radiation spokes, whether gantry, collimator, MLC or couch. It is based on ideas from Depuydt et al and Gonzalez et al.

Features:

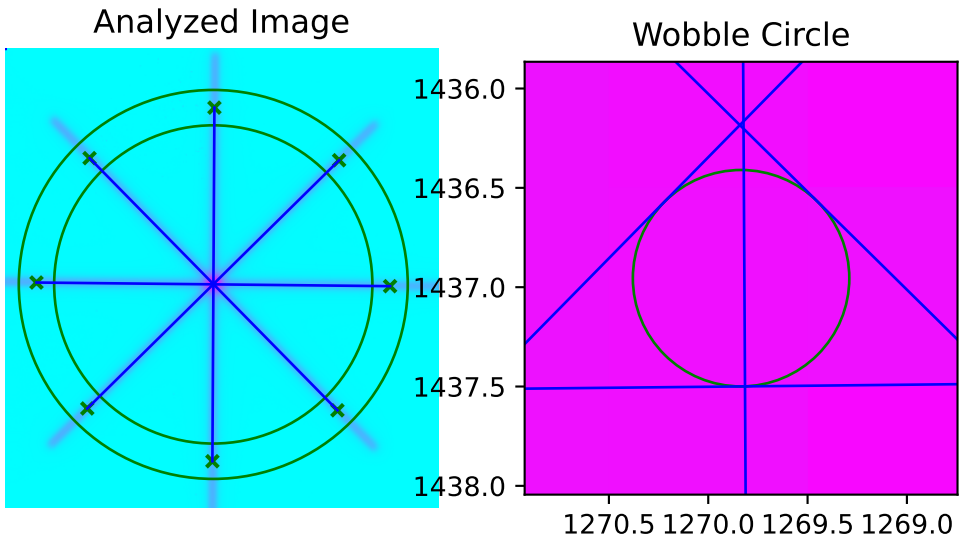
- **Analyze scanned film images, single EPID images, or a set of EPID images** - Any image that you can load in can be analyzed, including 1 or a set of EPID DICOM images and films that have been digitally scanned.
- **Any image size** - Have machines with different EPIDs? Scanned your film at different resolutions? No problem.
- **Dose/OD can be inverted** - Whether your device/image views dose as an increase in value or a decrease, pylinac will detect it and invert if necessary.
- **Automatic noise detection & correction** - Sometimes there's dirt on the scanned film; sometimes there's a dead pixel on the EPID. Pylinac will detect these spurious noise signals and can avoid or account for them.

- **Accurate, FWHM star line detection** - Pylinac uses not simply the maximum value to find the center of a star line, but analyzes the entire star profile to determine the center of the FWHM, ensuring small noise or maximum value bias is avoided.
- **Adaptive searching** - If you passed pylinac a set of parameters and a good result wasn't found, pylinac can recover and do an adaptive search by adjusting parameters to find a "reasonable" wobble.

6.5.2 Running the Demo

To run the Starshot demo, create a script or start an interpreter and input:

```
from pylinac import Starshot

Starshot.run_demo()
```



Results will be printed to the console and a matplotlib figure showing the analyzed starshot image will pop up:

```
Result: PASS

The minimum circle that touches all the star lines has a diameter of 0.381 mm.

The center of the minimum circle is at 1270.0, 1437.2
```

6.5.3 Image Acquisition

To capture starshot images, film is often used, but a sequence of EPID images can also work for collimator measurements. Pylinac can automatically superimpose the images. See the literature mentioned in the *Overview* for more info on acquisition.

6.5.4 Typical Use

The Starshot analysis can be run first by importing the Starshot class:

```
from pylinac import Starshot
```

A typical analysis sequence looks like so:

- **Load image(s)** – Loading film or superimposed EPID DICOM images can be done by passing the file path or by using a UI to find and get the file. The code might look like any of the following:

  ```
  star_img = "C:/QA Folder/gantry_starshot.tif"
  mystar = Starshot(star_img)
  ```

  Multiple images can be easily superimposed and used; e.g. collimator shots at various angles:

  ```
  star_imgs = ["path/star0.tif", "path/star45.tif", "path/star90.tif"]
  mystar = Starshot.from_multiple_images(star_imgs)
  ```

- **Analyze the image** – After loading the image, all that needs to be done is analyze the image. You may optionally pass in some settings:

  ```
  mystar.analyze(radius=0.5, tolerance=0.8)  # see API docs for more parameter info
  ```

- **View the results** – Starshot can print out the summary of results to the console as well as draw a matplotlib image to show the detected radiation lines and wobble circle:

  ```
  # print results to the console
  print(mystar.results())
  # view analyzed image
  mystar.plot_analyzed_image()
  ```

  Additionally, the data can be accessed through a convenient *StarshotResults* class which comes in useful when using pylinac through an API or for passing data to other scripts/routines.

  ```
  # return a dataclass with introspection
  data = mystar.results_data()
  data.tolerance_mm
  data.passed
  ...

  # return as a dict
  data_dict = mystart.results_data(as_dict=True)
  data_dict["passed"]
  ...
  ```

  Each subplot can be plotted independently as well:

```
# just the wobble plot
mystar.plot_analyzed_subimage("wobble")
# just the zoomed-out plot
mystar.plot_analyzed_subimage("whole")
```

Saving the images is also just as easy:

```
mystar.save_analyzed_image("mystar.png")
```

You may also save to PDF:

```
mystar.publish_pdf("mystar.pdf")
```

6.5.5 Algorithm

Allowances

- The image can be either inversion (radiation is darker or brighter).
- The image can be any size.
- The image can be DICOM (from an EPID) or most image formats (scanned film).
- If multiple images are used, they must all be the same size.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The image must have at least 6 spokes (3 angles).
- The center of the "star" must be in the central 1/3 of the image.
- The radiation spokes must extend to both sides of the center. I.e. the spokes must not end at the center of the circle.

Pre-Analysis

- **Check for image noise** – The image is checked for unreasonable noise by comparing the min and max to the 1/99th percentile pixel values respectively. If there is a large difference then there is likely an artifact and a median filter is applied until the min/max and 1/99th percentiles are similar.
- **Check image inversion** – The image is checked for proper inversion using histogram analysis.
- **Set algorithm starting point** – Unless the user has manually set the pixel location of the start point, it is automatically found by summing the image along each axis and finding the center of the full-width, 80%-max of each sum. The maximum value point is also located. Of the two points, the one closest to the center of the image is chosen as the starting point.

Analysis

- **Extract circle profile** – A circular profile is extracted from the image centered around the starting point and at the radius given.
- **Find spokes** – The circle profile is analyzed for peaks. Optionally, the profile is reanalyzed to find the center of the FWHM. An even number of spokes must be found (1 for each side; e.g. 3 collimator angles should produce 6 spokes, one for each side of the CAX).

- **Match peaks** – Peaks are matched to their counterparts opposite the CAX to compose a line using a simple peak number offset.
- **Find wobble** – Starting at the initial starting point, a Nelder-Mead gradient method is utilized to find the point of minimum distance to all lines. If recursive is set to True and a "reasonable" wobble (<2mm) is not found using the passes settings, the peak height and radius are iterated until a reasonable wobble is found.

Post-Analysis

- **Check if passed** – Once the wobble is calculated, it is tested against the tolerance given, and passes if below the tolerance. If the image carried a pixel/mm conversion ratio, the tolerance and result are in mm, otherwise they will be in pixels.

6.5.6 Troubleshooting

First, check the general *Troubleshooting* section, especially if an image won't load. Specific to the starshot analysis, there are a few things you can do.

- **Set recursive to True** - This easy step in *analyze()* allows pylinac to search for a reasonable wobble even if the conditions you passed don't for some reason give one.
- **Make sure the center of the star is in the central 1/3 of the image** - Otherwise, pylinac won't find it.
- **Make sure there aren't egregious artifacts** - Pin pricks can cause wild pixel values; crop them out if possible.
- **Set ``invert`` to True** - While right most of the time, it's possible the inversion checker got it wrong. This would look like peak locations in the "valley" regions of the image. If so, pass `invert=True` to the `analyze` method.

6.5.7 Benchmarking the Algorithm

With the image generator module we can create test images to test the starshot algorithm on known results. This is useful to isolate what is or isn't working if the algorithm doesn't work on a given image and when commissioning pylinac.

Perfect shot

Note: Due to the rounding of pixel positions of the star lines an absolutely perfect (0.0000mm wobble) is not achievable. The uncertainty of the algorithm is ~0.05mm.

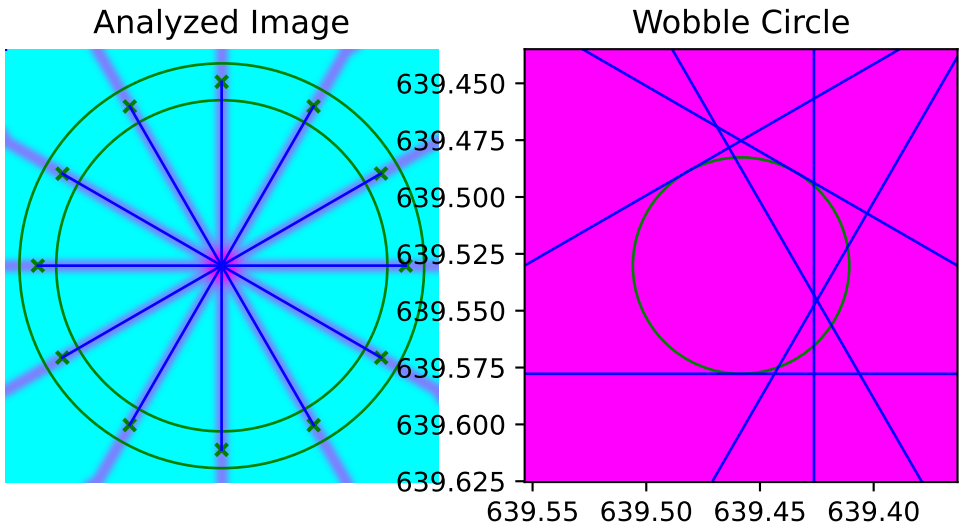
Let's create a perfect irradiation of a starshot pattern:

```
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,␣
↪AS1200Image, RandomNoiseLayer


star_path = 'perfect_starshot.dcm'
as1200 = AS1200Image()
for _ in range(6):
    as1200.add_layer(FilteredFieldLayer((270, 5), alpha=0.5))
    as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
```

(continues on next page)

(continued from previous page)

```
as1200.add_layer(GaussianFilterLayer(sigma_mm=3))
as1200.generate_dicom(file_out_name=star_path)

# analyze it
star = pylinac.Starshot(star_path)
star.analyze()
print(star.results())
star.plot_analyzed_image()
```



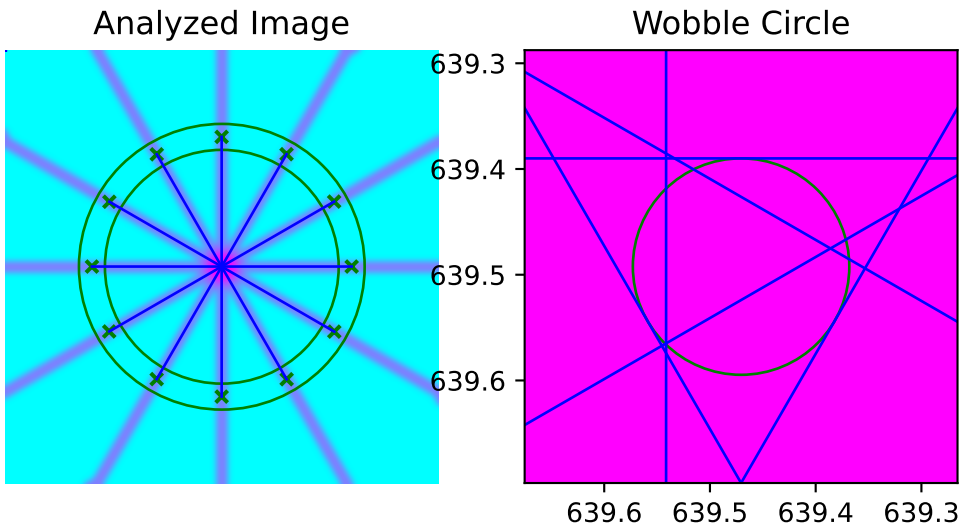
with an output of:

```
Result: PASS

The minimum circle that touches all the star lines has a diameter of 0.045 mm.

The center of the minimum circle is at 639.5, 639.5
```

Note that there is still an identified wobble of ~0.045mm due to pixel position rounding of the generated image star lines. The center of the star is dead on at 639.5 (AS1200 image of shape 1278 and going to the middle of the pixel).

We can also evaluate the effect of changing the radius:

```
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,␣
↪AS1200Image, RandomNoiseLayer


star_path = 'perfect_starshot.dcm'
as1200 = AS1200Image()
for _ in range(6):
    as1200.add_layer(FilteredFieldLayer((270, 5), alpha=0.5))
    as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(GaussianFilterLayer(sigma_mm=3))
as1200.generate_dicom(file_out_name=star_path)

# analyze it
star = pylinac.Starshot(star_path)
star.analyze(radius=0.6)  # radius changed
print(star.results())
star.plot_analyzed_image()
```



which results in:

```
Result: PASS

The minimum circle that touches all the star lines has a diameter of 0.036 mm.

The center of the minimum circle is at 639.5, 639.5
```

The center hasn't moved but we do have a diameter of ~0.03mm now. Again, this is a limitation of both the algorithm and image generation.

Offset

We can also generate an offset starshot:

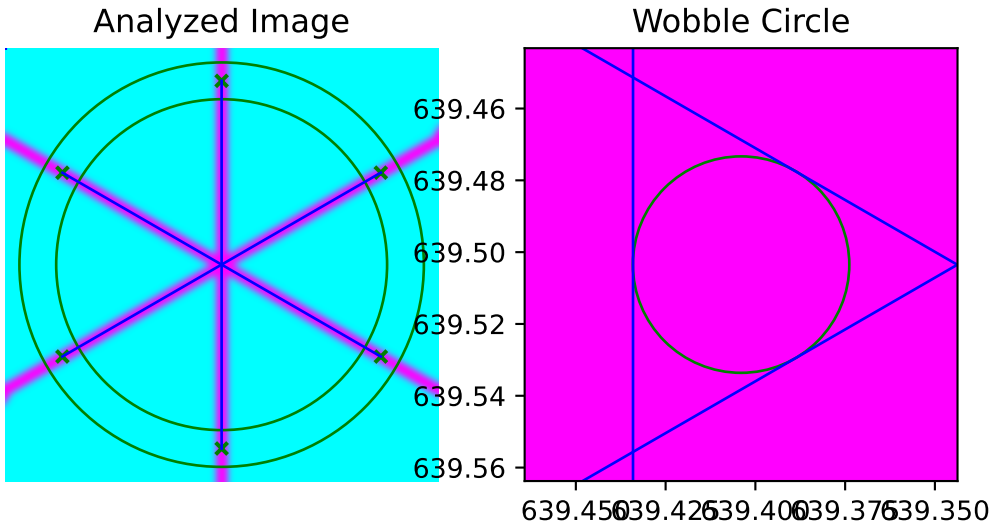
Note: This image is completely generated and depending on the angle and number of spokes, this result may change due to the fragility of rotating the image.

```
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,␣
↪AS1200Image, RandomNoiseLayer


star_path = 'offset_starshot.dcm'
as1200 = AS1200Image()
for _ in range(6):
    as1200.add_layer(FilteredFieldLayer((270, 5), alpha=0.5, cax_offset_mm=(1, 1)))
    as1200.image = ndimage.rotate(as1200.image, 60, reshape=False, mode='nearest')
as1200.add_layer(GaussianFilterLayer(sigma_mm=3))
as1200.generate_dicom(file_out_name=star_path)

# analyze it
star = pylinac.Starshot(star_path)
star.analyze()
print(star.results())
star.plot_analyzed_image()
```

with an output of:

```
Result: FAIL

The minimum circle that touches all the star lines has a diameter of 1.035 mm.

The center of the minimum circle is at 637.8, 633.3
```

Note that we still have the 0.035mm error from the algorithm uncertainty but that we have caught the 1mm offset appropriately.


Analyzed Image
Wobble Circle
639.46
639.48
639.50
639.52
639.54
639.56
639.450
639.425
639.400
639.375
639.350

6.5.8 API Documentation

class pylinac.starshot.**Starshot**(*filepath: str | BinaryIO*, ***kwargs*)

Bases: `object`

Class that can determine the wobble in a "starshot" image, be it gantry, collimator, couch or MLC. The image can be a scanned film (TIF, JPG, etc) or a sequence of EPID DICOM images.

Attributes

image : `Image` circle_profile : *StarProfile* lines : *LineManager* wobble : *Wobble* tolerance : `Tolerance`

Examples

Run the demo:

```
>>> Starshot.run_demo()
```

Typical session:

```
>>> img_path = r"C:/QA/Starshots/Coll.jpeg"
>>> mystar = Starshot(img_path, dpi=105, sid=1000)
>>> mystar.analyze()
>>> print(mystar.results())
>>> mystar.plot_analyzed_image()
```

Parameters

filepath
The path to the image file.

kwargs
Passed to *load()*.

classmethod from_url(*url: str*, ***kwargs*)

Instantiate from a URL.

Parameters

url
[str] URL of the raw file.

kwargs
Passed to *load()*.

classmethod from_demo_image()

Construct a Starshot instance and load the demo image.

classmethod from_multiple_images(*filepath_list: list*, *stretch_each: bool = True*, *method: str = 'sum'*, ***kwargs*)

Construct a Starshot instance and load in and combine multiple images.

Parameters

filepath_list
[iterable] An iterable of file paths to starshot images that are to be superimposed.

stretch_each
[bool] Whether to stretch each image individually before combining. See `load_multiples`.

method
[{'sum', 'mean'}] The method to combine the images. See `load_multiples`.

kwargs
Passed to *load_multiples()*.

classmethod from_zip(*zip_file: str*, ***kwargs*)

Construct a Starshot instance from a ZIP archive.

Parameters

zip_file
[str] Points to the ZIP archive. Can contain a single or multiple images. If multiple images the images are combined and thus should be from the same test sequence.

kwargs
Passed to *load_multiples()*.

analyze(*radius: float = 0.85*, *min_peak_height: float = 0.25*, *tolerance: float = 1.0*, *start_point:* Point | *tuple | None = None*, *fwhm: bool = True*, *recursive: bool = True*, *invert: bool = False*)

Analyze the starshot image.

Analyze finds the minimum radius and center of a circle that touches all the lines (i.e. the wobble circle diameter and wobble center).

Parameters

radius
[float, optional] Distance in % between starting point and closest image edge; used to build the circular profile which finds the radiation lines. Must be between 0.05 and 0.95.

min_peak_height
[float, optional] The percentage minimum height a peak must be to be considered a valid peak. A lower value catches radiation peaks that vary in magnitude (e.g. different MU delivered or gantry shot), but could also pick up noise. If necessary, lower value for gantry shots and increase for noisy images.

tolerance
[int, float, optional] The tolerance in mm to test against for a pass/fail result.

start_point
[2-element iterable, optional] The point where the algorithm should center the circle profile, given as (x-value, y-value). If None (default), will search for a reasonable maximum point nearest the center of the image.

fwhm
[bool] If True (default), the center of the FWHM of the spokes will be determined. If False, the peak value location is used as the spoke center.

Note: In practice, this ends up being a very small difference. Set to false if peak locations are offset or unexpected.

recursive
[bool] If True (default), will recursively search for a "reasonable" wobble, meaning the wobble radius is <3mm. If the wobble found was unreasonable, the minimum peak height is iteratively adjusted from low to high at the passed radius. If for all peak heights at the given radius the wobble is still unreasonable, the radius is then iterated over from most distant inward, iterating over minimum peak heights at each radius. If False, will simply return the first determined value or raise error if a reasonable wobble could not be determined.

Warning: It is strongly recommended to leave this setting at True.

invert
[bool] Whether to force invert the image values. This should be set to True if the automatically-determined pylinac inversion is incorrect.

Raises

RuntimeError
If a reasonable wobble value was not found.

property passed: bool

Boolean specifying whether the determined wobble was within tolerance.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *StarshotResults* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs: dict*)

Draw the star lines, profile circle, and wobble circle on a matplotlib figure.

Parameters

show
[bool] Whether to actually show the image.

plt_kwargs
[dict] Keyword args passed to the plt.subplots() method. Allows one to set things like figure size.

`plot_analyzed_subimage`(*subimage: str = 'wobble'*, *ax: plt.Axes | None = None*, *show: bool = True*, ***plt_kwargs: dict*)

Plot a subimage of the starshot analysis. Current options are the zoomed out image and the zoomed in image.

Parameters

subimage
[str] If 'wobble', will show a zoomed in plot of the wobble circle. Any other string will show the zoomed out plot.

ax
[None, matplotlib Axes] If None (default), will create a new figure to plot on, otherwise plot to the passed axes.

show
[bool] Whether to actually show the image.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size. Only used if ax is not passed.

`save_analyzed_image`(*filename: str*, ***kwargs*)

Save the analyzed image plot to a file.

Parameters

filename
[str, IO stream] The filename to save as. Format is deduced from string extention, if there is one. E.g. 'mystar.png' will produce a PNG image.

kwargs
All other kwargs are passed to plt.savefig().

`save_analyzed_subimage`(*filename: str*, *subimage: str = 'wobble'*, ***kwargs*)

Save the analyzed subimage to a file.

Parameters

filename
[str, file-object] Where to save the file to.

subimage
[str] If 'wobble', will show a zoomed in plot of the wobble circle. Any other string will show the zoomed out plot.

kwargs
Passed to matplotlib.

publish_pdf(*filename: str | BinaryIO*, *notes: str | list[str] | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

static run_demo()

Demonstrate the Starshot module using the demo image.

class pylinac.starshot.**StarshotResults**(*tolerance_mm: float*, *circle_diameter_mm: float*, *circle_radius_mm: float*, *passed: bool*, *circle_center_x_y: tuple[float, float]*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

tolerance_mm: float

circle_diameter_mm: float

circle_radius_mm: float

passed: bool

circle_center_x_y: tuple[float, float]

class pylinac.starshot.**StarProfile**(*image*, *start_point*, *radius*, *min_peak_height*, *fwhm*)

Bases: *CollapsedCircleProfile*

Class that holds and analyzes the circular profile which finds the radiation lines.

Parameters

width_ratio
[float] The "thickness" of the band to sample. The ratio is relative to the radius. E.g. if the radius is 20 and the width_ratio is 0.2, the "thickness" will be 4 pixels.

num_profiles
[int] The number of profiles to sample in the band. Profiles are distributed evenly within the band.

See Also

CircleProfile : Further parameter info.

get_peaks(*min_peak_height*, *min_peak_distance=0.02*, *fwhm=True*)

Determine the peaks of the profile.

class pylinac.starshot.**Wobble**(*center_point=None*, *radius=None*)

Bases: *Circle*

A class that holds the wobble information of the Starshot analysis.

Attributes

radius_mm : The radius of the Circle in **mm**.

Parameters

center_point
[Point, optional] Center point of the wobble circle.

radius
[float, optional] Radius of the wobble circle.

property diameter_mm: float

Diameter of the wobble in mm.

class pylinac.starshot.**LineManager**(*points: list[*Point*]*)

Bases: object

Manages the radiation lines found.

Parameters

points :
The peak points found by the StarProfile

`construct_rad_lines`(*points: list[*Point*]*)

Find and match the positions of peaks in the circle profile (radiation lines)
and map their positions to the starshot image.

Radiation lines are found by finding the FWHM of the radiation spokes, then matching them to form lines.

Returns

lines
[list] A list of Lines (radiation lines) found.

See Also

Starshot.analyze() : min_peak_height parameter info core.profile.CircleProfile.find_FWXM_peaks : min_peak_distance parameter info. geometry.Line : returning object

`match_points`(*points: list[*Point*]*)

Match the peaks found to the same radiation lines.

Peaks are matched by connecting the existing peaks based on an offset of peaks. E.g. if there are 12 peaks, there must be 6 radiation lines. Furthermore, assuming star lines go all the way across the CAX, the 7th peak will be the opposite peak of the 1st peak, forming a line. This method is robust to starting points far away from the real center.

`plot`(*axis: Axes*)

Plot the lines to the axis.

6.6 VMAT

6.6.1 Overview

The VMAT module consists of the class VMAT, which is capable of loading an EPID DICOM Open field image and MLC field image and analyzing the images according to the Varian RapidArc QA tests and procedures, specifically the Dose-Rate & Gantry-Speed (DRGS) and Dose-Rate & MLC speed (DRMLC) tests.

Features:

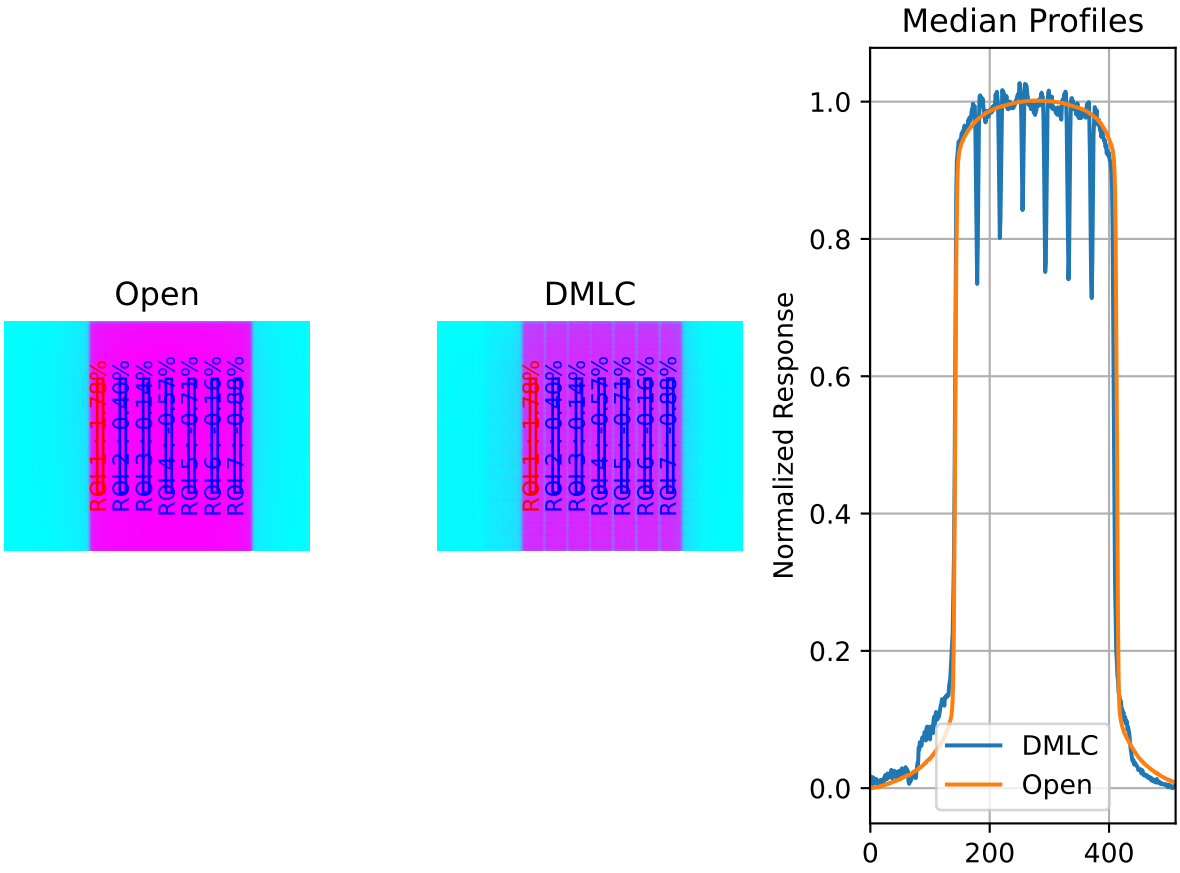
- **Do both tests** - Pylinac can handle either DRGS or DRMLC tests.
- **Automatic offset correction** - Older VMAT tests had the ROIs offset, newer ones are centered. No worries, pylinac finds the ROIs automatically.
- **Automatic open/DMLC identification** - Pass in both images–don't worry about naming. Pylinac will automatically identify the right images.

Note: There are two classes in the VMAT module: `DRGS` and `DRMLC`. Each have the exact same methods. Anytime one class is used here as an example, the other class can be used the same way.

6.6.2 Running the Demos

For this example we will use the DRGS class:

```
from pylinac import DRGS
DRGS.run_demo()
```



Results will be printed to the console and a figure showing both the Open field and MLC field image will pop up:

```
Dose Rate & Gantry Speed
Test Results (Tol. +/-1.5%): PASS
Max Deviation: 1.01%
Absolute Mean Deviation: 0.459%
```

6.6.3 Image Acquisition

If you want to perform these specific QA tests, you'll need DICOM plan files that control the linac precisely to deliver the test fields. These can be downloaded from my.varian.com. Once logged in, search for RapidArc and you should see two items called "RapidArc QA Test Procedures and Files for TrueBeam"; there will be a corresponding one for C-Series. Use the RT Plan files and follow the instructions, not including the assessment procedure, which is the point of this module. Save & move the VMAT images to a place you can use pylinac.

6.6.4 Typical Use

The VMAT QA analysis follows what is specified in the Varian RapidArc QA tests and assumes your tests will run the exact same way. Import the appropriate class:

```
from pylinac import DRGS, DRMLC
```

The minimum needed to get going is to:

- **Load images** – Loading the EPID DICOM images into your VMAT class object can be done by passing the file paths, passing a ZIP archive, or passing a URL:

  ```
  open_img = "C:/QA Folder/VMAT/open_field.dcm"
  dmlc_img = "C:/QA Folder/VMAT/dmlc_field.dcm"
  mydrgs = DRGS(
      image_paths=(open_img, dmlc_img)
  )  # use the DRMLC class the exact same way

  # from zip
  mydrmlc = DRMLC.from_zip(r"C:/path/to/zip.zip")

  # from a URL
  mydrgs = DRGS.from_url("http://myserver.org/vmat.zip")
  ```

  Finally, if you don't have any images, you can use the demo ones provided:

  ```
  mydrgs = DRGS.from_demo_images()
  mydrmlc = DRMLC.from_demo_images()
  ```

- **Analyze the images** – Once the images are loaded, tell the class to analyze the images. See the Algorithm section for details on how this is done. Tolerance can also be passed and has a default value of 1.5%:

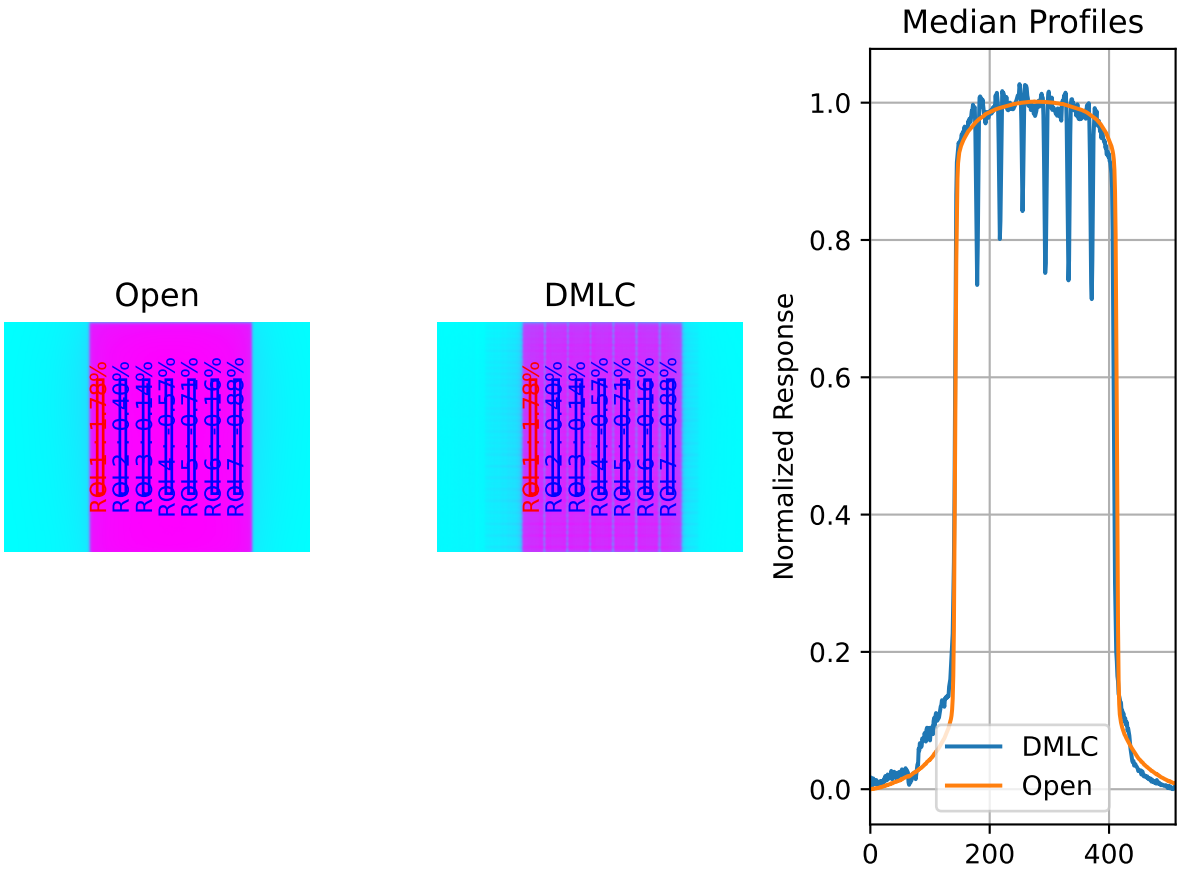
  ```
  mydrgs.analyze(tolerance=1.5)
  ```

- **View/Save the results** – The VMAT module can print out the summary of results to the console as well as draw a matplotlib image to show where the segments were placed and their values:

  ```
  # print results to the console
  print(mydrgs.results())
  # view analyzed images
  mydrgs.plot_analyzed_image()
  ```

  PDF reports can also be generated:

  ```
  myvmat.publish_pdf("drgs.pdf")
  ```


Open
DMLC
Median Profiles
Normalized Response
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
DMLC
Open

6.6.5 Customizing the analysis

You can alter both the segment size and segment positions as desired.

To change the segment size:

```
drgs = DRGS.from_demo_image()
drgs.analyze(
    ..., segment_size_mm=(10, 150)
)  # ROI segments will now be 10mm wide by 150mm tall
# same story for DRMLC
```

To change the x-positions of the ROI segments or change the number of ROI, use a custom ROI config dictionary and pass it to the `analyze` method.

```
from pylinac import DRGS, DRMLC

# note the keys are the names of the ROIs and can be anything you like
custom_roi_config = {
    "200 MU/min": {"offset_mm": -100},
    "300 MU/min": {"offset_mm": -80},
}  # add more as needed

my_drgs = DRGS(...)  # works the same way for DRMLC
my_drgs.analyze(..., roi_config=custom_roi_config)
```

6.6.6 Accessing Data

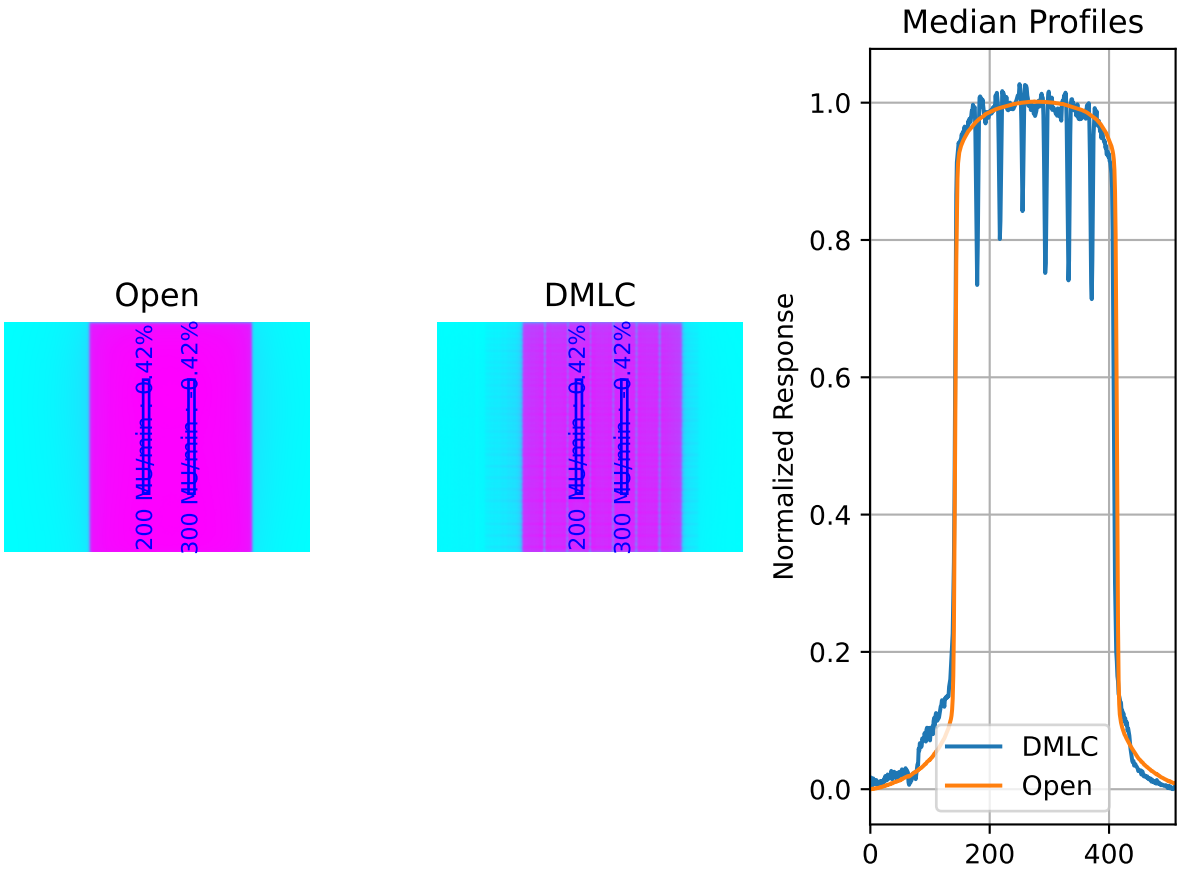
Changed in version 3.0.

Using the VMAT module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *results_data()* method which returns a *VMATResult* instance.

Note: While the pylinac tooling may change under the hood, this object should remain largely the same and/or expand. Thus, using this is more stable than accessing attrs directly.

Continuing from above:

```
data = my_drmlc.results_data()
data.test_type
data.passed
# and more

# return as a dict
data_dict = my_drmlc.results_data(as_dict=True)
data_dict["test_type"]
...
```


Open
DMLC
Median Profiles
Normalized Response
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
DMLC
Open

6.6.7 Algorithm

The VMAT analysis algorithm is based on the Varian RapidArc QA Test Procedures for C-Series and Truebeam. Two tests (besides Picket Fence, which has its own module) are specified. Each test takes 10x0.5cm samples, each corresponding to a distinct section of radiation. A corrected reading of each segment is made, defined as: $M_{corr}(x) = \frac{M_{DRGS}(x)}{M_{open}(x)} * 100$. The reading deviation of each segment is calculated as: $M_{deviation}(x) = \frac{M_{corr}(x)}{\bar{M_{corr}}} * 100 - 100$, where $\bar{M_{corr}}$ is the average of all segments.

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID (aS500, aS1000, aS1200).

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The tests must be delivered using the DICOM RT plan files provided by Varian which follow the test layout of Ling et al.
- The images must be acquired with the EPID.

Pre-Analysis

- **Determine image scaling** – Segment determination is based on offsets from the center pixel of the image. However, some physicists use 150 cm SID and others use 100 cm, and others can use a clinical setting that may be different than either of those. To account for this, the SID is determined and then scaling factors are determined to be able to perform properly-sized segment analysis.

Analysis

Note: Calculations tend to be lazy, computed only on demand. This represents a nominal analysis where all calculations are performed.

- **Calculate sample boundaries** – The Segment x-positions are based on offsets from the center of the FWHM of the detected field. This allows for old and new style tests that have an x-offset from each other. These values are then scaled with the image scaling factor determined above.
- **Calculate the corrected reading** – For each segment, the mean pixel value is determined for both the open and DMLC image. These values are used to determine the corrected reading: M_{corr}.
- **Calculate sample and segment ratios** – The sample values of the DMLC field are divided by their corresponding open field values.
- **Calculate segment deviations** – Segment deviation is then calculated once all the corrected readings are determined. The average absolute deviation is also calculated.

Post-Analysis

- **Test if segments pass tolerance** – Each segment is checked to see if it was within the specified tolerance. If any samples fail, the whole test is considered failing.

6.6.8 Benchmarking the Algorithm

With the image generator module we can create test images to test the VMAT algorithm on known results. This is useful to isolate what is or isn't working if the algorithm doesn't work on a given image and when commissioning pylinac.

Note: The below examples are for the DRMLC test but can equally be applied to the DRGS tests as well.

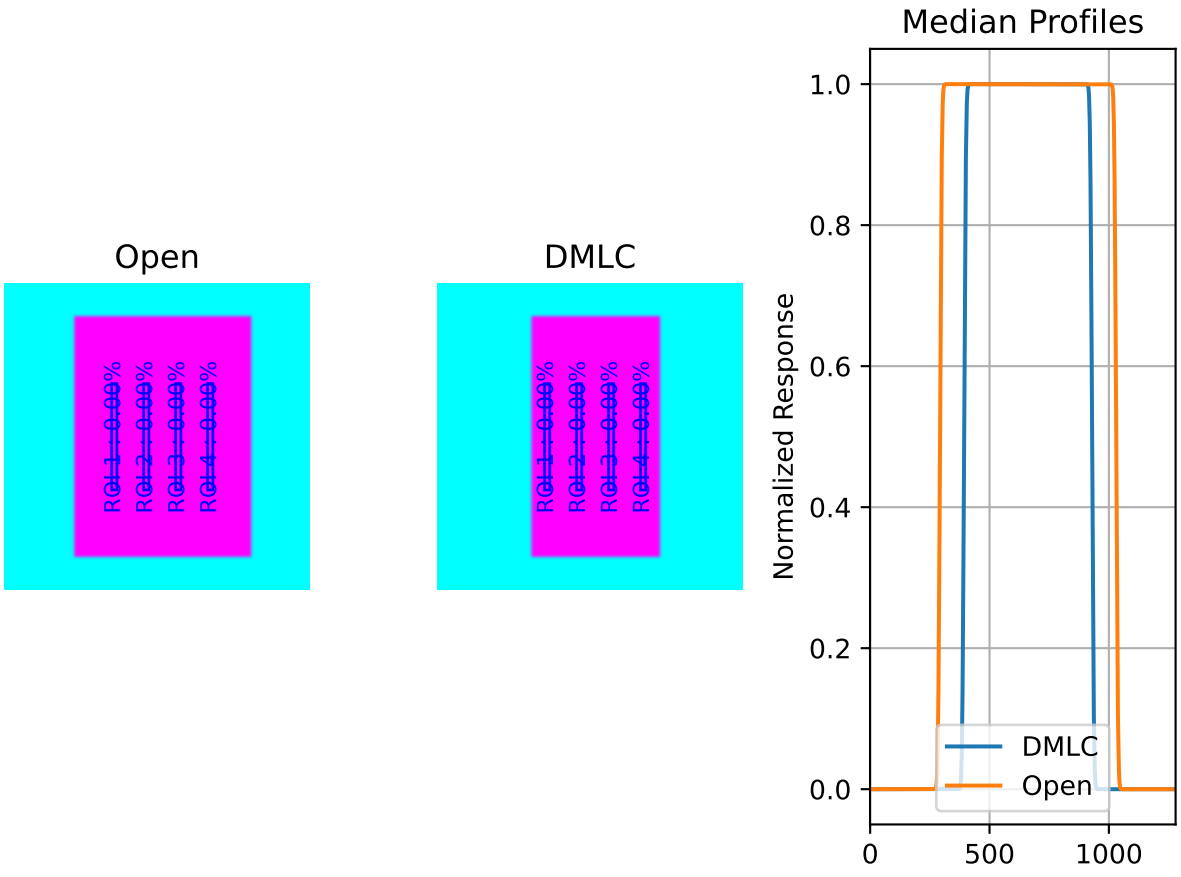
Perfect Fields

In this example, we generate a perfectly flat set of images and analyze them.

```
import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, PerfectFieldLayer,␣
↪AS1200Image

# open image
open_path = 'perfect_open_drmlc.dcm'
as1200 = AS1200Image()
as1200.add_layer(PerfectFieldLayer(field_size_mm=(150, 110), cax_offset_mm=(0, 5)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.generate_dicom(file_out_name=open_path)

# DMLC image
dmlc_path = 'perfect_dmlc_drmlc.dcm'
as1200 = AS1200Image()
for offset in (-40, -10, 20, 50):
    as1200.add_layer(PerfectFieldLayer((150, 20), cax_offset_mm=(0, offset)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.generate_dicom(file_out_name=dmlc_path)

# analyze it
vmat = pylinac.DRMLC(image_paths=(open_path, dmlc_path))
vmat.analyze()
print(vmat.results())
vmat.plot_analyzed_image()
```

with output:

```
Dose Rate & MLC Speed
Test Results (Tol. +/-1.5%): PASS
Max Deviation: 0.0%
Absolute Mean Deviation: 0.0%
```


Open
DMLC
Median Profiles
Normalized Response
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
DMLC
Open

Noisy, Realistic

We now add a horn effect and random noise to the data:

```
import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,␣
↪AS1200Image, RandomNoiseLayer

# open image
open_path = 'noisy_open_drmlc.dcm'
as1200 = AS1200Image()
as1200.add_layer(FilteredFieldLayer(field_size_mm=(150, 110), cax_offset_mm=(0, 5)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=0.03))
as1200.generate_dicom(file_out_name=open_path)

# DMLC image
dmlc_path = 'noisy_dmlc_drmlc.dcm'
as1200 = AS1200Image()
for offset in (-40, -10, 20, 50):
    as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, offset)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=0.03))
as1200.generate_dicom(file_out_name=dmlc_path)

# analyze it
vmat = pylinac.DRMLC(image_paths=(open_path, dmlc_path))
vmat.analyze()
print(vmat.results())
vmat.plot_analyzed_image()
```

with output:

```
Dose Rate & MLC Speed
Test Results (Tol. +/-1.5%): PASS
Max Deviation: 0.0332%
Absolute Mean Deviation: 0.0257%
```

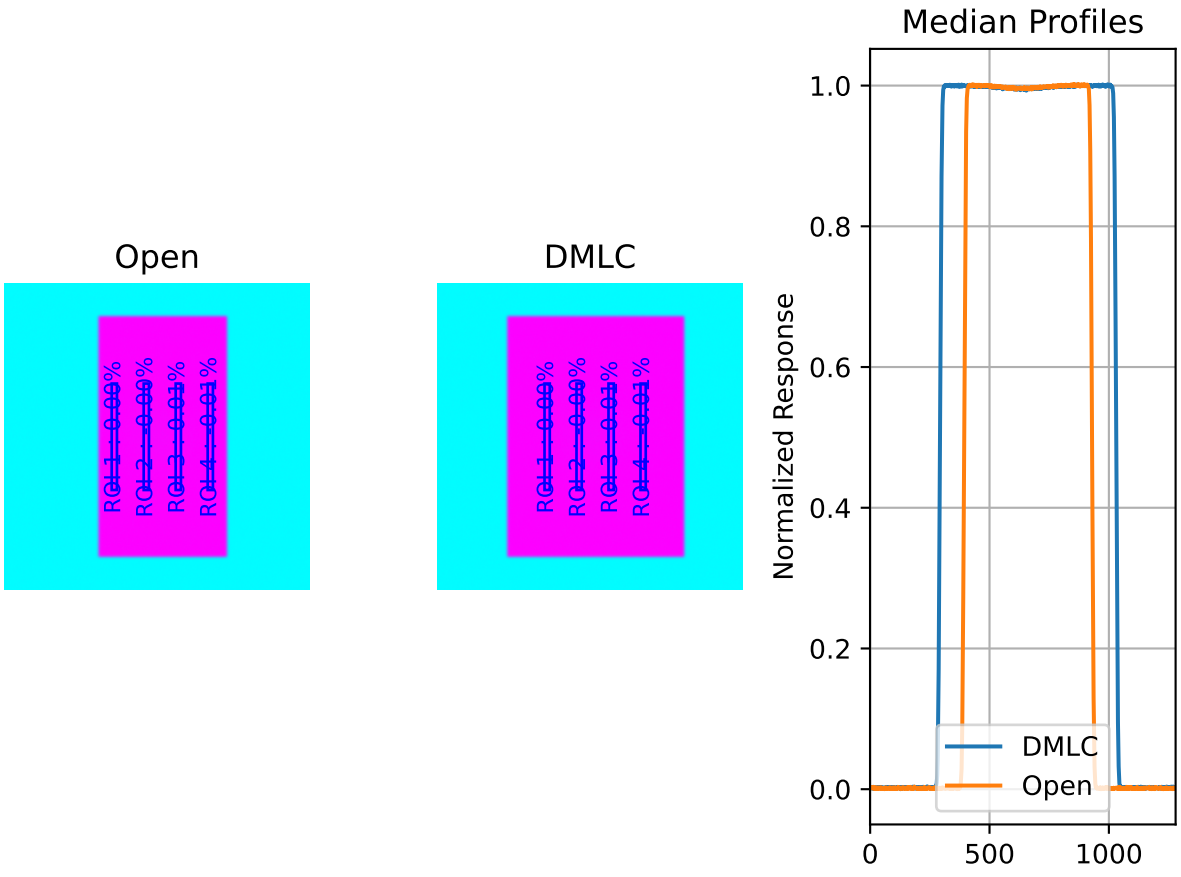
Erroneous data

Let's now get devious and randomly adjust the height of each ROI (effectively changing the apparent MLC speed):

Note: Due to the purposely random nature shown below, this exact result is likely not reproducible, nor was it intended to be. To get reproducible behavior, use numpy with a seed value.

```
import random

import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,␣
↪AS1200Image, RandomNoiseLayer
```

(continues on next page)


Open
RG1 : 0.09%
RG2 : 0.00%
RG3 : 0.01%
RG4 : 0.01%
DMLC
RG1 : 0.09%
RG2 : 0.00%
RG3 : 0.01%
RG4 : 0.01%
Median Profiles
Normalized Response
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
DMLC
Open

(continued from previous page)

```
# open image
open_path = 'noisy_open_drmlc.dcm'
as1200 = AS1200Image()
as1200.add_layer(FilteredFieldLayer(field_size_mm=(150, 110), cax_offset_mm=(0, 5)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=0.03))
as1200.generate_dicom(file_out_name=open_path)

# DMLC image
dmlc_path = 'noisy_dmlc_drmlc.dcm'
as1200 = AS1200Image()
for offset in (-40, -10, 20, 50):
    as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, offset),␣
↪alpha=random.uniform(0.93, 1)))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=0.03))
as1200.generate_dicom(file_out_name=dmlc_path)

# analyze it
vmat = pylinac.DRMLC(image_paths=(open_path, dmlc_path))
vmat.analyze()
print(vmat.results())
vmat.plot_analyzed_image()
```

with an output of:

```
Dose Rate & MLC Speed
Test Results (Tol. +/-1.5%): FAIL
Max Deviation: 2.12%
Absolute Mean Deviation: 1.13%
```

Comparing to other programs

Note: The DRGS pattern is used as an example but the same concepts applies to both DRGS and DRMLC.

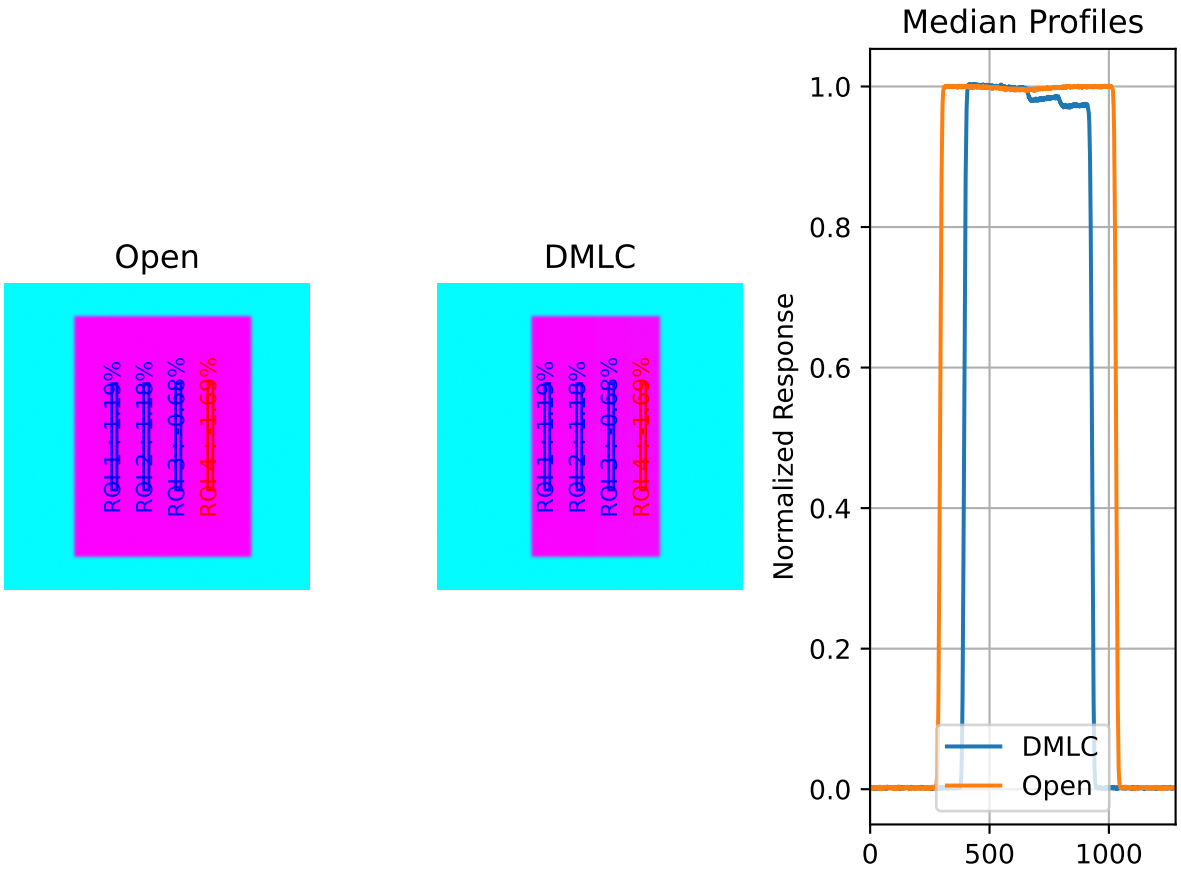
A common question is how results should be compared to other programs. While the answer will depend on several factors, we can make some general observations here.

- **Ensure the ROI sizes are similar** - Different programs may have different defaults for the ROI size. Varian suggests a 5x100mm rectangular ROI, although this seems arbitrarily small in our opinion. In any event, to match Varian's suggestion, the pylinac default segment ROI size is 5x100mm
- **DICOM images may have different reconstruction algorithms** - Pylinac's DICOM image loading algorithm can be read here: *Pixel Data & Inversion*. It tries to use as many tags as it can to reconstruct the correct pixel values. However, this behavior does not appear consistent across all programs. E.g. if tags are not considered when loading images, the resulting pixels (and thus ratio) may not match an image that did use tags.

  Take for instance the below example comparing "raw" pixel values to using the Tag-corrected version:

  ```
  from matplotlib import pyplot as plt
  import pydicom
  ```

(continues on next page)


Open
DMLC
Median Profiles
Normalized Response
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
DMLC
Open

(continued from previous page)

```
from pylinac import image
from pylinac.core.io import retrieve_demo_file, TemporaryZipDirectory

demo_zip = retrieve_demo_file('drgs.zip')
with TemporaryZipDirectory(demo_zip) as tmpzip:
    image_files = image.retrieve_image_files(tmpzip)

    # read the values "raw"
    dmlc_raw = pydicom.read_file(image_files[0])
    open_raw = pydicom.read_file(image_files[1])
    raw = dmlc_raw.pixel_array / open_raw.pixel_array

    # Tag-correct the values
    img_dmlc = image.load(image_files[0])
    img_open = image.load(image_files[1])
    corrected = img_dmlc.array / img_open.array

plt.plot(raw[200, :], label="Raw DICOM pixels")
plt.plot(corrected[200, :], label="Using Rescale + Intercept Tags")
plt.legend()
plt.grid(True)
plt.show()
```

We can also scale the tag-corrected value for the purpose of comparing relative responses:

The point of the second plot is to show what the ratio of each ROI looks like between the normalizations. Inspecting the left-most ROI, we see that the raw pixel normalization is lower than the average ROI response, whereas with the tag-corrected implementation, it's actually higher. When evaluating the ROI results of pylinac vs other programs this explains why the left-most ROI (which is used simply as an example) has a positive deviation whereas other programs may have a negative deviation.

This behavior can change depending on the tags available in the DICOM file. Newer DICOMs also have a "sign" tag to correct for inversion of pixel data. Why this difference can be problematic is that the ratio of the open to DMLC image depends on the initial pixel value.

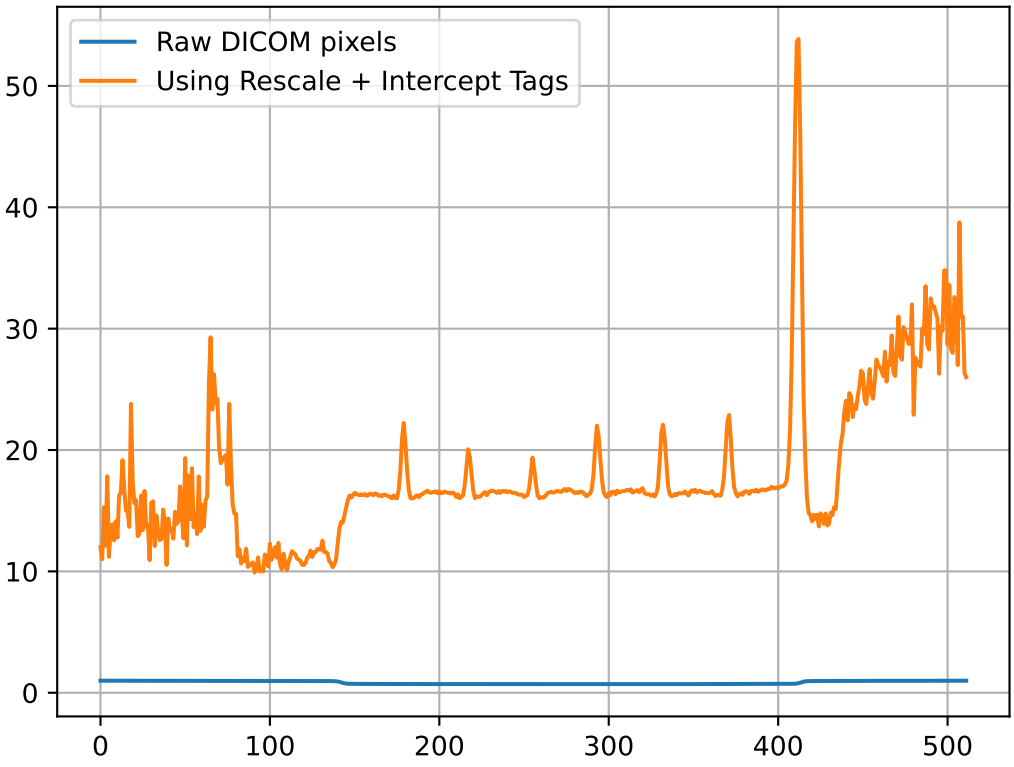
Currently, this is a philosophical difference between programs that don't use DICOM tags and those that do, like pylinac. If the goal is to switch from another program to pylinac, the standard approach of measuring with both algorithms to establish a baseline of differences is recommended, just as you might when switching from a water tank measurement to an array-based measurement scheme.

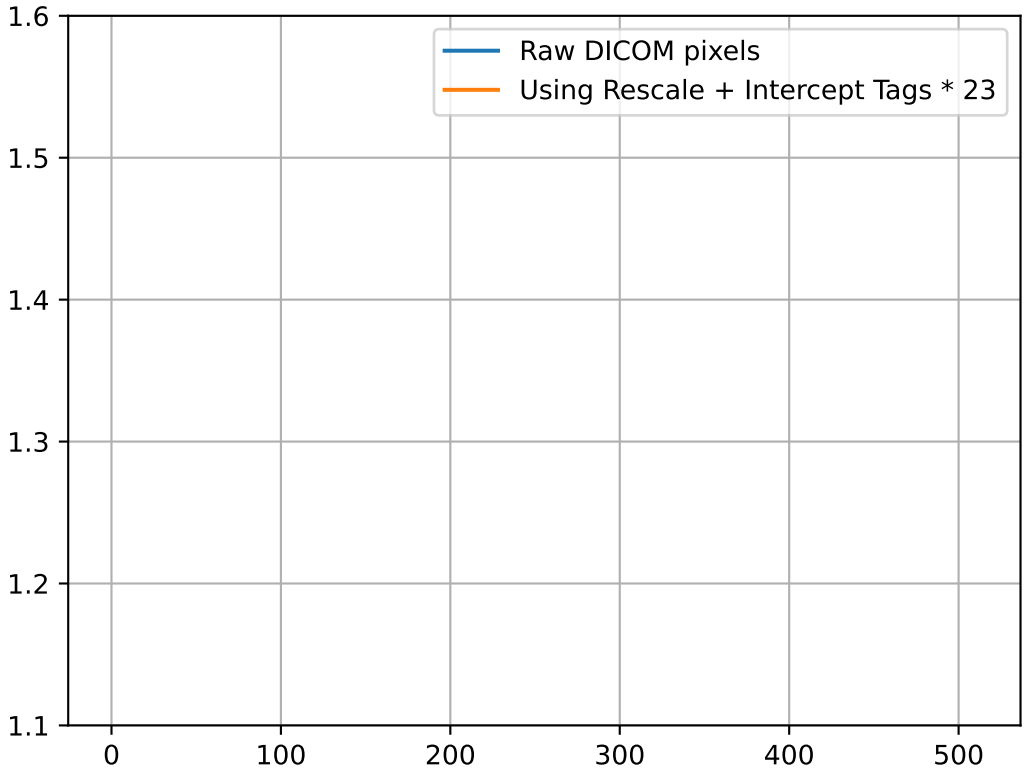
Comparison to Doselab

New in version 3.13.

All that being said, if the goal is to match another program (specifically, Doselab, although this might apply to others) use the following:

```
from pylinac import DRMLC

drmlc = DRMLC(..., raw_pixels=True, ground=False, check_inversion=False)
...
```


Raw DICOM pixels
Using Rescale + Intercept Tags
50
40
30
20
10
0
0
100
200
300
400
500


Raw DICOM pixels
Using Rescale + Intercept Tags * 23
1.6
1.5
1.4
1.3
1.2
1.1
0
100
200
300
400
500

This will skip the checking of DICOM tags for correcting the pixel values as well as other manipulations normally applied.

Here's a table comparing the results of the DRMLC demo dataset with different variations:

	Max R_dev	ROI 1 R_dev	ROI 2 R_dev	ROI 3 R_dev	ROI 4 R_dev
Doselab (normalized)		0.995	1.005	1.006	0.994
Doselab (as % from unity)	0.60%	-0.50%	0.50%	0.60%	-0.60%
Pylinac (raw=True, ground=False, inversion=False)	0.56%	-0.54%	0.53%	0.56%	-0.55%
Pylinac (default)	0.89%	-0.68%	0.89%	-0.10%	-0.11%
Pylinac (raw=False, ground=False, inversion=False)	0.90%	-0.68%	0.90%	-0.08%	-0.12%

The Doselab and pylinac results are very similar when the raw pixels are used. The default settings and the analysis without any extra manipulations are also extremely similar.

Note: For historical continuity, the manipulations are set to `True`. If you are just starting to use Pylinac, it is recommended to use the settings of the last row. However, it is unlikely to make a significant difference.

6.6.9 API Documentation

Main classes

These are the classes a typical user may interface with.

class pylinac.vmat.**DRGS**(*image_paths: Sequence[str | BinaryIO | Path]*, *ground=True*, *check_inversion=True*, ***kwargs*)

Bases: *VMATBase*

Class representing a Dose-Rate, Gantry-speed VMAT test. Will accept, analyze, and return the results.

Parameters

image_paths
[iterable (list, tuple, etc)] A sequence of paths to the image files.

kwargs
Passed to the image loading function. See *load()*.

static run_demo()
Run the demo for the Dose Rate & Gantry Speed test.

analyze(*tolerance: float | int = 1.5*, *segment_size_mm: tuple = (5, 100)*, *roi_config: dict | None = None*)
Analyze the open and DMLC field VMAT images, according to 1 of 2 possible tests.

Parameters

tolerance
[float, int, optional] The tolerance of the sample deviations in percent. Default is 1.5. Must be between 0 and 8.

segment_size_mm
[tuple(int, int)] The (width, height) of the ROI segments in mm.

roi_config
[dict] A dict of the ROI settings. The keys are the names of the ROIs and each value is a dict containing the offset in mm 'offset_mm'.

property avg_abs_r_deviation: float

Return the average of the absolute R_deviation values.

property avg_r_deviation: float

Return the average of the R_deviation values, including the sign.

classmethod from_demo_images(***kwargs*)

Construct a VMAT instance using the demo images.

classmethod from_url(*url: str*)

Load a ZIP archive from a URL. Must follow the naming convention.

Parameters

url
[str] Must point to a valid URL that is a ZIP archive of two VMAT images.

classmethod from_zip(*path: str | Path*, ***kwargs*)

Load VMAT images from a ZIP file that contains both images. Must follow the naming convention.

Parameters

path
[str] Path to the ZIP archive which holds the VMAT image files.

kwargs
Passed to the constructor.

property max_r_deviation: float

Return the value of the maximum R_deviation segment.

plot_analyzed_image(*show: bool = True*, *show_text: bool = True*, ***plt_kwargs: dict*)

Plot the analyzed images. Shows the open and dmlc images with the segments drawn; also plots the median profiles of the two images for visual comparison.

Parameters

show
[bool] Whether to actually show the image.

show_text
[bool] Whether to show the ROI names on the image.

plt_kwargs
[dict] Keyword args passed to the plt.subplots() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

property r_devs: ndarray

Return the deviations of all segments as an array.

results() → str

A string of the summary of the analysis results.

Returns

str
The results string showing the overall result and deviation statistics by segment.

results_data(*as_dict=False*) → *VMATResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

class pylinac.vmat.**DRMLC**(*image_paths: Sequence[str | BinaryIO | Path]*, *ground=True*, *check_inversion=True*, ***kwargs*)

Bases: *VMATBase*

Class representing a Dose-Rate, MLC speed VMAT test. Will accept, analyze, and return the results.

Parameters

image_paths
[iterable (list, tuple, etc)] A sequence of paths to the image files.

kwargs
Passed to the image loading function. See *load()*.

analyze(*tolerance: float | int = 1.5*, *segment_size_mm: tuple = (5, 100)*, *roi_config: dict | None = None*)
Analyze the open and DMLC field VMAT images, according to 1 of 2 possible tests.

Parameters

tolerance
[float, int, optional] The tolerance of the sample deviations in percent. Default is 1.5. Must be between 0 and 8.

segment_size_mm
[tuple(int, int)] The (width, height) of the ROI segments in mm.

roi_config
[dict] A dict of the ROI settings. The keys are the names of the ROIs and each value is a dict containing the offset in mm 'offset_mm'.

property avg_abs_r_deviation: float
Return the average of the absolute R_deviation values.

property avg_r_deviation: float
Return the average of the R_deviation values, including the sign.

classmethod from_demo_images(***kwargs*)
Construct a VMAT instance using the demo images.

classmethod from_url(*url: str*)
Load a ZIP archive from a URL. Must follow the naming convention.

Parameters

url
[str] Must point to a valid URL that is a ZIP archive of two VMAT images.

classmethod from_zip(*path: str | Path*, ***kwargs*)
Load VMAT images from a ZIP file that contains both images. Must follow the naming convention.

Parameters

path
[str] Path to the ZIP archive which holds the VMAT image files.

kwargs
Passed to the constructor.

property max_r_deviation: float

Return the value of the maximum R_deviation segment.

plot_analyzed_image(*show: bool = True*, *show_text: bool = True*, ***plt_kwargs: dict*)

Plot the analyzed images. Shows the open and dmlc images with the segments drawn; also plots the median profiles of the two images for visual comparison.

Parameters

show
[bool] Whether to actually show the image.

show_text
[bool] Whether to show the ROI names on the image.

plt_kwargs
[dict] Keyword args passed to the plt.subplots() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

property r_devs: ndarray

Return the deviations of all segments as an array.

results() → str

A string of the summary of the analysis results.

Returns

str
The results string showing the overall result and deviation statistics by segment.

results_data(*as_dict=False*) → *VMATResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

static run_demo()

Run the demo for the MLC leaf speed test.

class pylinac.vmat.**VMATResult**(*test_type: str*, *tolerance_percent: float*, *max_deviation_percent: float*, *abs_mean_deviation: float*, *passed: bool*, *segment_data: Iterable[SegmentResult]*, *named_segment_data: dict[str, SegmentResult]*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

test_type: str

tolerance_percent: float

max_deviation_percent: float

abs_mean_deviation: float

passed: bool

segment_data: Iterable[*SegmentResult*]

named_segment_data: dict[str, *SegmentResult*]

class pylinac.vmat.**SegmentResult**(*passed: bool*, *x_position_mm: float*, *r_corr: float*, *r_dev: float*, *center_x_y: float*, *stdev: float*)

Bases: `object`

An individual segment/ROI result

passed: bool

x_position_mm: float

r_corr: float

r_dev: float

center_x_y: float

stdev: float

Supporting Classes

You generally won't have to interface with these unless you're doing advanced behavior.

class pylinac.vmat.**VMATBase**(*image_paths: Sequence[str | BinaryIO | Path]*, *ground=True*, *check_inversion=True*, ***kwargs*)

Bases: `object`

Parameters

image_paths
[iterable (list, tuple, etc)] A sequence of paths to the image files.

kwargs
Passed to the image loading function. See *load()*.

classmethod from_url(*url: str*)
Load a ZIP archive from a URL. Must follow the naming convention.

Parameters

url
[str] Must point to a valid URL that is a ZIP archive of two VMAT images.

classmethod from_zip(*path: str | Path*, ***kwargs*)
Load VMAT images from a ZIP file that contains both images. Must follow the naming convention.

Parameters

path
[str] Path to the ZIP archive which holds the VMAT image files.

kwargs
Passed to the constructor.

classmethod from_demo_images(***kwargs*)
Construct a VMAT instance using the demo images.

analyze(*tolerance: float | int = 1.5*, *segment_size_mm: tuple = (5, 100)*, *roi_config: dict | None = None*)
Analyze the open and DMLC field VMAT images, according to 1 of 2 possible tests.

Parameters

tolerance
[float, int, optional] The tolerance of the sample deviations in percent. Default is 1.5. Must be between 0 and 8.

segment_size_mm
[tuple(int, int)] The (width, height) of the ROI segments in mm.

roi_config
[dict] A dict of the ROI settings. The keys are the names of the ROIs and each value is a dict containing the offset in mm 'offset_mm'.

results() → str

A string of the summary of the analysis results.

Returns

str
The results string showing the overall result and deviation statistics by segment.

results_data(*as_dict=False*) → *VMATResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

property r_devs: ndarray

Return the deviations of all segments as an array.

property avg_abs_r_deviation: float

Return the average of the absolute R_deviation values.

property avg_r_deviation: float

Return the average of the R_deviation values, including the sign.

property max_r_deviation: float

Return the value of the maximum R_deviation segment.

plot_analyzed_image(*show: bool = True*, *show_text: bool = True*, ***plt_kwargs: dict*)

Plot the analyzed images. Shows the open and dmlc images with the segments drawn; also plots the median profiles of the two images for visual comparison.

Parameters

show
[bool] Whether to actually show the image.

show_text
[bool] Whether to show the ROI names on the image.

plt_kwargs
[dict] Keyword args passed to the plt.subplots() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

class pylinac.vmat.**Segment**(*center_point:* Point, *open_image:* image.DicomImage, *dmlc_image:* image.DicomImage, *tolerance: float | int*)

Bases: *Rectangle*

A class for holding and analyzing segment data of VMAT tests.

For VMAT tests, there are either 4 or 7 'segments', which represents a section of the image that received radiation under the same conditions.

Attributes

r_dev
[float] The reading deviation (R_dev) from the average readings of all the segments. See documentation for equation info.

Parameters

width
[number] Width of the rectangle. Must be positive

height
[number] Height of the rectangle. Must be positive.

center
[Point, iterable, optional] Center point of rectangle.

as_int
[bool] If False (default), inputs are left as-is. If True, all inputs are converted to integers.

property r_corr: float
Return the ratio of the mean pixel values of DMLC/OPEN images.

property stdev: float
Return the standard deviation of the segment.

property passed: bool

Return whether the segment passed or failed.

get_bg_color() → str

Get the background color of the segment when plotted, based on the pass/fail status.

property bl_corner: *Point*

The location of the bottom left corner.

property br_corner: *Point*

The location of the bottom right corner.

plot2axes(*axes: Axes*, *edgecolor: str = 'black'*, *angle: float = 0.0*, *fill: bool = False*, *alpha: float = 1*, *facecolor: str = 'g'*, *label=None*, *text: str = ''*, *fontsize: str = 'medium'*, *text_rotation: float = 0*)

Plot the Rectangle to the axes.

Parameters

axes
[matplotlib.axes.Axes] An MPL axes to plot to.

edgecolor
[str] The color of the circle.

angle
[float] Angle of the rectangle.

fill
[bool] Whether to fill the rectangle with color or leave hollow.

text: str
If provided, plots the given text at the center. Useful for identifying ROIs on a plotted image apart.

fontsize: str
The size of the text, if provided. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html for options.

text_rotation: float
The rotation of the text in degrees.

property tl_corner: *Point*

The location of the top left corner.

property tr_corner: *Point*

The location of the top right corner.

6.7 CatPhan

6.7.1 Overview

The CT module automatically analyzes DICOM images of a CatPhan 504, 503, 600, Quart DVT, or ACR phantoms acquired when doing CBCT or CT quality assurance. It can load a folder or zip file that the images are in and automatically correct for translational and rotational errors. It can analyze the HU regions and image scaling (CTP404), the high-contrast line pairs (CTP528) to calculate the modulation transfer function (MTF), the HU uniformity (CTP486), and Low Contrast (CTP515) on the corresponding slices.

For ACR and Quart phantoms, the equivalent sections are analyzed where applicable even though each module does not have an explicit name. Where intuitive similarities between the phantoms exist, the library usage is the same.

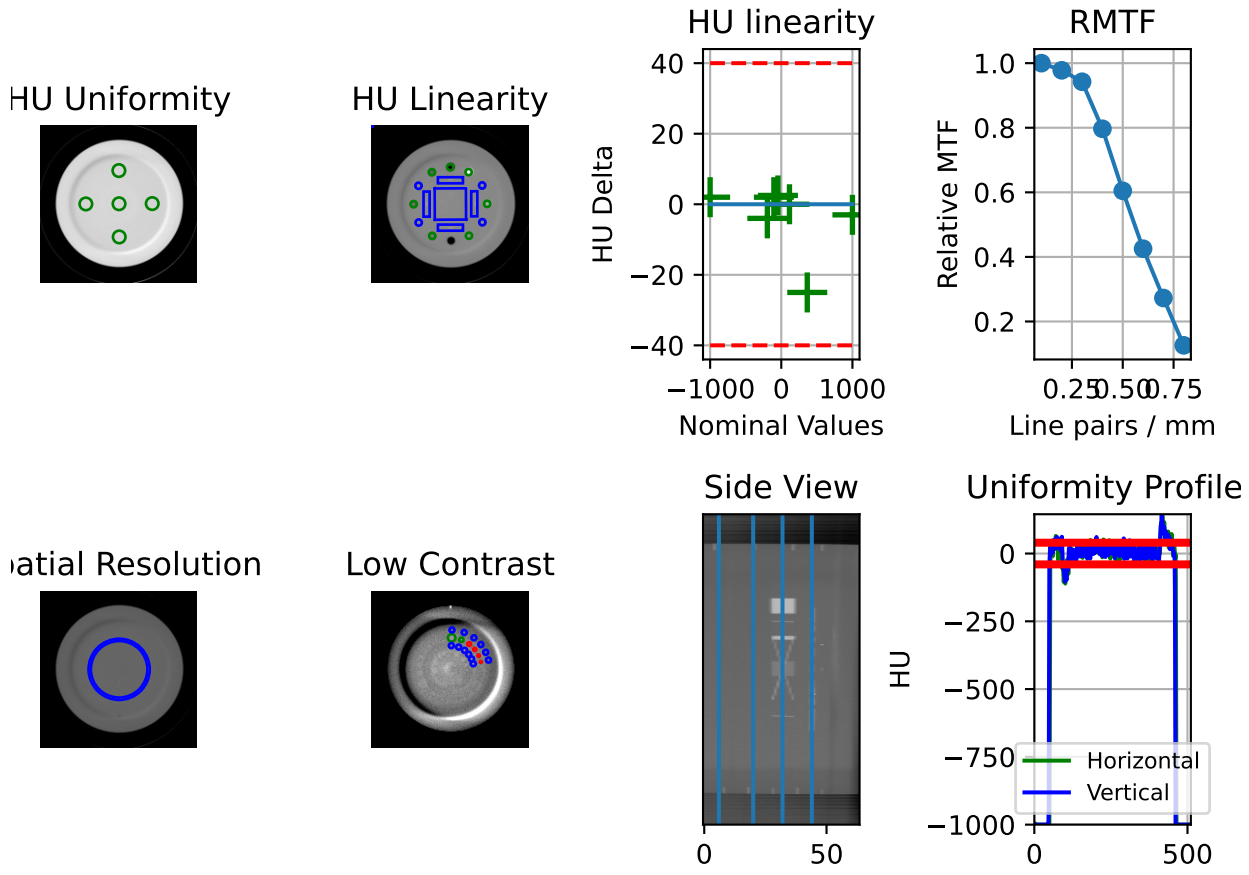
Features:

- **Automatic phantom registration** - Your phantom can be tilted, rotated, or translated–pylinac will automatically register the phantom.
- **Automatic testing of all major modules** - Major modules are automatically registered and analyzed.
- **Any scan protocol** - Scan your CatPhan with any protocol; even scan it in a regular CT scanner. Any field size or field extent is allowed.

6.7.2 Running the Demo

To run one of the CatPhan demos, create a script or start an interpreter and input:

```
from pylinac import CatPhan504
cbct = CatPhan504.run_demo()  # the demo is a Varian high quality head scan
```



Results will be also be printed to the console:

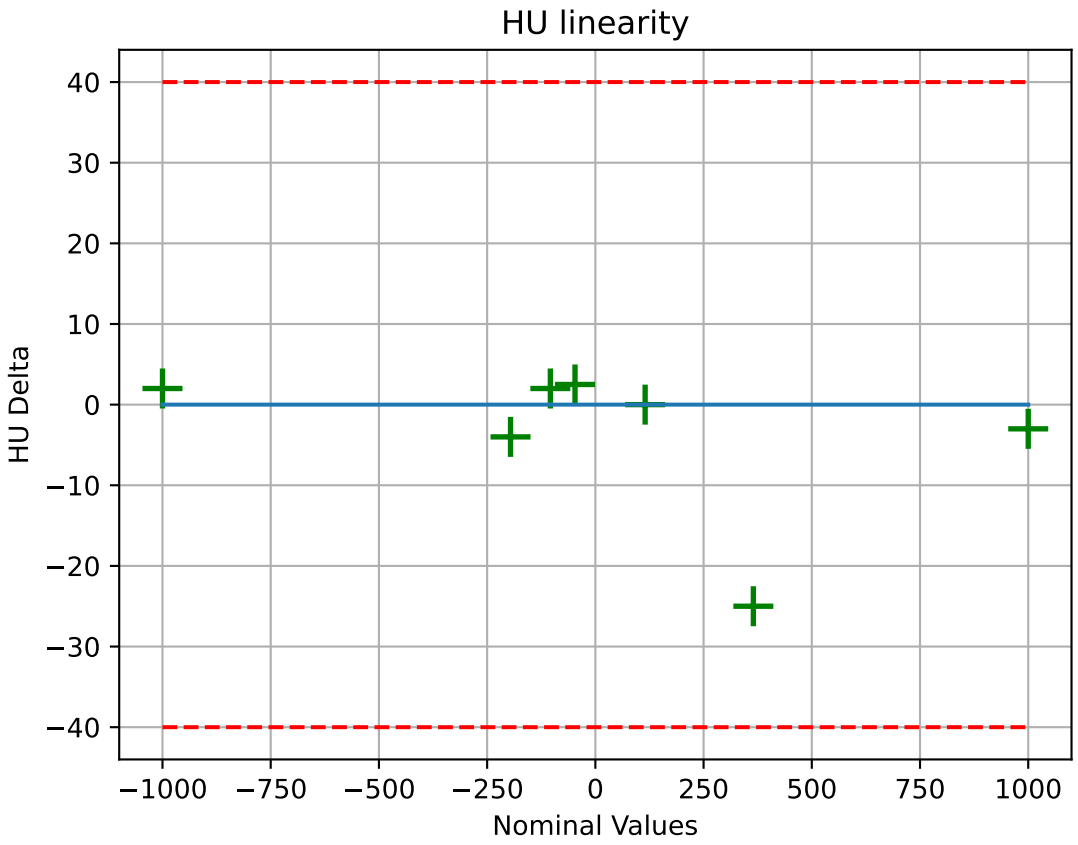
```
- CatPhan 504 QA Test -
HU Linearity ROIs: Air: -998.0, PMP: -200.0, LDPE: -102.0, Poly: -45.0, Acrylic: 115.0,␣
```

(continues on next page)

(continued from previous page)

```
↪Delrin: 340.0, Teflon: 997.0
HU Passed?: True
Low contrast visibility: 3.46
Geometric Line Average (mm): 49.95
Geometry Passed?: True
Measured Slice Thickness (mm): 2.499
Slice Thickness Passed? True
Uniformity ROIs: Top: 6.0, Right: -1.0, Bottom: 5.0, Left: 10.0, Center: 14.0
Uniformity index: -1.479
Integral non-uniformity: 0.0075
Uniformity Passed?: True
MTF 50% (lp/mm): 0.56
Low contrast ROIs "seen": 3
```

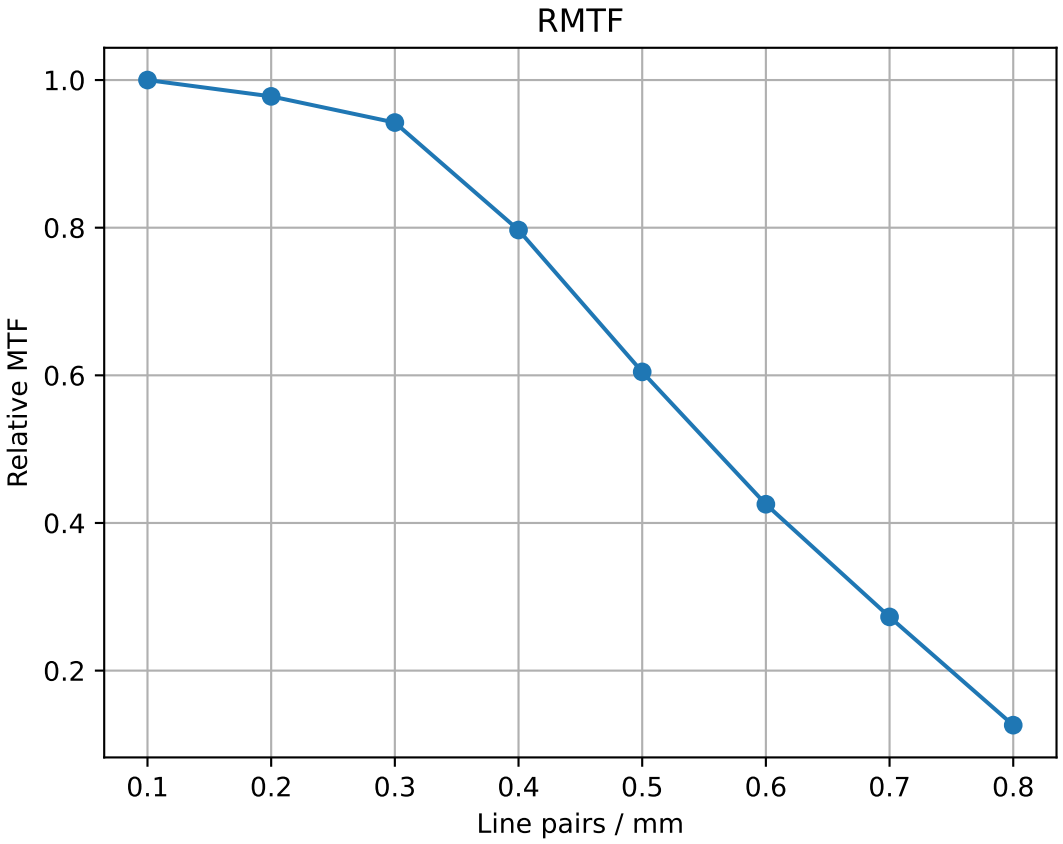
As well, you can plot and save individual pieces of the analysis such as linearity:



Or the rMTF:

```
cbct.plot_analyzed_subimage("rmtf")
```

Or generate a PDF report:


RMTF
Relative MTF
1.0
0.8
0.6
0.4
0.2
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Line pairs / mm

```
cbct.publish_pdf("mycbct.pdf")
```

6.7.3 Image Acquisition

Acquiring a scan of a CatPhan has a few simple requirements:

1. The field of view must be larger than the phantom diameter + a few cm for clearance.
2. The phantom should not be touching any edge of the FOV.
3. The phantom shouldn't touch the couch or other high-HU objects. This may cause localization issues finding the phantom. If the phantom doesn't have an associated cradle, setting it on foam or something similar is recommended.
4. All modules must be visible.

> **Warning:** This can cause strange results if not all modules are scanned. Furthermore, aligning axially to the white dots on the side of the catphan will not catch the inferior modules on a typical CBCT scan. We suggest aligning to the center of the HU module (the module inferior to the white dots) when acquiring via CBCT.

6.7.4 Typical Use

CatPhan analysis as done by this module closely follows what is specified in the CatPhan manuals, replacing the need for manual measurements. There are 4 CatPhan models that pylinac can analyze: *CatPhan504*, *CatPhan503*, & *CatPhan600*, & *CatPhan604*, each with their own class in pylinac. Let's assume you have the CatPhan504 for this example. Using the other models/classes is exactly the same except the class name.

```
from pylinac import CatPhan504  # or import the CatPhan503 or CatPhan600
```

The minimum needed to get going is to:

- **Load images** – Loading the DICOM images into your CatPhan object is done by passing the images in during construction. The most direct way is to pass in the directory where the images are:

  ```
  cbct_folder = r"C:/QA Folder/CBCT/June monthly"
  mycbct = CatPhan504(cbct_folder)
  ```

  or load a zip file of the images:

  ```
  zip_file = r"C:/QA Folder/CBCT/June monthly.zip"
  mycbct = CatPhan504.from_zip(zip_file)
  ```

  You can also use the demo images provided:

  ```
  mycbct = CatPhan504.from_demo_images()
  ```

- **Analyze the images** – Once the folder/images are loaded, tell pylinac to start analyzing the images. See the Algorithm section for details and `analyze()` for analysis options:

  ```
  mycbct.analyze()
  ```

- **View the results** – The CatPhan module can print out the summary of results to the console as well as draw a matplotlib image to show where the samples were taken and their values:

```
# print results to the console
print(mycbct.results())
# view analyzed images
mycbct.plot_analyzed_image()
# save the image
mycbct.save_analyzed_image("mycatphan504.png")
# generate PDF
mycbct.publish_pdf(
    "mycatphan.pdf", open_file=True
)  # open the PDF after saving as well.
```

6.7.5 Custom HU values

New in version 3.16.

By default, expected HU values are drawn from the values stated in the *manual*. It's possible to override one or more of the HU values of the CatPhan modules however. This is useful if you have a custom CatPhan or know the exact HU values of your phantom. To do so, pass in a dictionary of the HU values to the `expected_hu_values` parameter. The keys should be the name of the material and the values should be the HU value.

```
from pylinac import CatPhan504

# overrides the HU values of the Air and PMP regions
mycbct = CatPhan504(...)
mycbct.analyze(..., expected_hu_values={"Air": -999, "PMP": -203})
mycbct.plot_analyzed_image()
```

Note: Not all materials need to be overridden. Only the ones you want to override need to be passed in.

Keys

The keys to override for CatPhan504 and CatPhan503 are listed below along with the default value:

- `Air: -1000`
- `PMP: -196`
- `LDPE: -104`
- `Poly: -47`
- `Acrylic: 115`
- `Delrin: 365`
- `Teflon: 1000`

The CatPhan600 has the above keys as well as:

- `Vial: 0`

The CatPhan604 has the original keys as well as:

- `50% Bone: 725`
- `20% Bone: 237`

6.7.6 Slice Thickness

New in version 3.12.

When measuring slice thickness in pylinac, slices are sometimes combined depending on the slice thickness. Scans with thin slices and low mAs can have very noisy wire ramp measurements. To compensate for this, pylinac will average over 3 slices (+/-1 from CTP404) if the slice thickness is <3.5mm. This will generally improve the statistics of the measurement. This is the only part of the algorithm that may use more than one slice.

If you'd like to override this, you can do so by setting the padding (aka straddle) explicitly. The straddle is the number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The default is `auto`; set to an integer to explicitly use a certain amount of straddle slices. Typical values are 0, 1, and 2. So, a value of 1 averages over 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

Note: This technique can be especially useful when your slices overlap.

```
from pylinac import CatPhan504  # applies to all catphans

ct = CatPhan504(...)
ct.analyze(..., thickness_slice_straddle=0)
...
```

6.7.7 Advanced Use

Using `results_data`

Changed in version 3.0.

Using the catphan module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the `results_data()` method which returns a `CatPhanResult` instance.

Note: While the pylinac tooling may change under the hood, this object should remain largely the same and/or expand. Thus, using this is more stable than accessing attrs directly.

Continuing from above:

```
data = mycbct.results_data()
data.catphan_model
data.ctp404.measured_slice_thickness_mm
# and more

# return as a dict
data_dict = mycbct.results_data(as_dict=True)
data_dict["ctp404"]["measured_slice_thickness_mm"]
...
```

Partial scans

While the default behavior of pylinac is to analyze all modules in the scan (in fact it will error out if they aren't), the behavior can be customized. Pylinac **always** has to be aware of the CTP404 module as that's the reference slice for everything else. Thus, if the 404 is not in the scan you're SOL. However, if one of the other modules is not present you can remove or adjust its offset by subclassing and overloading the `modules` attr:

```
from pylinac import CatPhan504  # works for any of the other phantoms too
from pylinac.ct import CTP515, CTP486


class PartialCatPhan504(CatPhan504):
    modules = {
        CTP486: {"offset": -65},
        CTP515: {"offset": -30},
        # the CTP528 was omitted
    }


ct = PartialCatPhan504.from_zip(...)  # use like normal
```

Examining rMTF

The rMTF can be calculated ad hoc like so. Note that CTP528 must be present (see above):

```
ct = ...  # load a dataset like normal
ct.analyze()
ct.ctp528.mtf.relative_resolution(x=40)  # get the rMTF (lp/mm) at 40% resolution
```

Customizing module locations

Similar to partial scans, to modify the module location(s), overload the `modules` attr and edit the `offset` value. The value is in mm:

```
from pylinac import CatPhan504  # works for any of the other phantoms too
from pylinac.ct import CTP515, CTP486, CTP528


# create custom catphan with module locations
class OffsetCatPhan504(CatPhan504):
    modules = {
        CTP486: {"offset": -60},  # normally -65
        CTP528: {"offset": 30},
        CTP515: {"offset": -25},  # normally -30
    }


ct = OffsetCatPhan504.from_zip(...)  # use like normal
```

Customizing Modules

You can also customize modules themselves in v2.4+. Customization should always be done by subclassing an existing module and overloading the attributes. Then, pass in the new custom module into the parent CatPhan class. The easiest way to get started is copy the relevant attributes from the existing code.

As an example, let's override the angles of the ROIs for CTP404.

```
from pylinac.ct import CatPhan504, CTP404CP504


# first, customize the module
class CustomCTP404(CTP404CP504):
    roi_dist_mm = 58.7  # this is the default value; we repeat here because it's easy to␣
↪copy from source
    roi_radius_mm = 5  # ditto
    roi_settings = {
        "Air": {
            "value": -1000,
            "angle": -93,  # changed 'angle' from -90 to -93
            "distance": roi_dist_mm,
            "radius": roi_radius_mm,
        },
        "PMP": {
            "value": -196,
            "angle": -122,  # changed 'angle' from -120 to -122
            "distance": roi_dist_mm,
            "radius": roi_radius_mm,
        },
        # add other ROIs as appropriate
    }


# then, pass to the CatPhan model
class CustomCP504(CatPhan504):
    modules = {
        CustomCTP404: {"offset": 0}
        # add other modules here as appropriate
    }


# use like normal
ct = CustomCP504(...)
```

Warning: If you overload the `roi_settings` or `modules` attributes, you are responsible for filling it out completely. I.e. when you overload it's not partial. In the above example if you want other CTP modules you **must** populate them.

6.7.8 Algorithm

The CatPhan module is based on the tests and values given in the respective CatPhan manual. The algorithm works like such:

Allowances

- The images can be any size.
- The phantom can have significant translation in all 3 directions.
- The phantom can have significant roll and moderate yaw and pitch.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- All of the modules defined in the `modules` attribute must be within the scan extent.
- Scan slices are not expected to overlap.

  Warning: Overlapping slices are not generally a problem other than the slice thickness measurement. See the *Slice Thickness* section for how to override this to get a valid slice thickness in such a situation.

Pre-Analysis

- **Determine image properties** – Upon load, the image set is analyzed for its DICOM properties to determine mm/pixel spacing, rescale intercept and slope, manufacturer, etc.
- **Convert to HU** – The entire image set is converted from its raw values to HU by applying the rescale intercept and slope which is contained in the DICOM properties.
- **Find the phantom z-location** – Upon loading, all the images are scanned to determine where the HU linearity module (CTP404) is located. This is accomplished by examining each image slice and looking for 2 things:
  - **If the CatPhan is in the image.** At the edges of the scan this may not be true.
  - **If a circular profile has characteristics like the CTP404 module.** If the CatPhan is in the image, a circular profile is taken at the location where the HU linearity regions of interest are located. If the profile contains low, high, and lots of medium values then it is very likely the HU linearity module. All such slices are found and the median slice is set as the HU linearity module location. All other modules are located relative to this position.

Analysis

- **Determine phantom roll** – Precise knowledge of the ROIs to analyze is important, and small changes in rotation could invalidate automatic results. The roll of the phantom is determined by examining the HU module and converting to a binary image. The air holes are then located and the angle of the two holes determines the phantom roll.

Note: For each step below, the "module" analyzed is actually the mean, median, or maximum of 3 slices (+/- 1 slice around and including the nominal slice) to ensure robust measurements. Also, for each step/phantom module, the phantom center is determined, which corrects for the phantom pitch and yaw.

Additionally, values tend to be lazy (computed only when asked for), thus the calculations listed may sometimes be performed only when asked for.

- **Determine HU linearity** – The HU module (CTP404) contains several materials with different HU values. Using hardcoded angles (corrected for roll) and radius from the center of the phantom, circular ROIs are sampled which correspond to the HU material regions. The median pixel value of the ROI is the stated HU value. Nominal HU values are taken as the mean of the range given in the manual(s):

Note: As of v3.16, these can be overriden: *Custom HU values*.

Nominal material formulation and specific gravity

Material	Formula	Z_{eff}[1]	Specific Gravity[2]	HU range[3]
Air	.78N, .21O, .01Ar	8.00	0.00	-1046 : -986
PMP	$[C_6H_{12}(CH_2)]$	5.44	0.83	-220 : -172
LDPE	$[C_2H_4]$	5.44	0.92	-121 : -87
Polystyrene	$[C_8H_8]$	5.70	1.03	-65 : -29
Acrylic	$[C_5H_8O_2]$	6.47	1.18	92 : 137
Bone 20%	.51C, .06Ca, .06H, .06N, .30O, .03P	9.09	1.14	211 : 263
Delrin®	Proprietary	6.95	1.42	344 : 387
Bone 50%	.35C, .14Ca, .04H, .06N, .34O, .06P	11.46	1.40	667 : 783
Teflon®	$[CF_2]$	8.43	2.16	941 : 1060

- **Determine HU uniformity** – HU uniformity (CTP486) is calculated in a similar manner to HU linearity, but within the CTP486 module/slice.
- **Calculate Geometry/Scaling** – The HU module (CTP404), besides HU materials, also contains several "nodes" which have an accurate spacing (50 mm apart). Again, using hardcoded but corrected angles, the area around the 4 nodes are sampled and then a threshold is applied which identifies the node within the ROI sample. The center of mass of the node is determined and then the space between nodes is calculated.
- **Calculate Spatial Resolution/MTF** – The Spatial Resolution module (CTP528) contains 21 pairs of aluminum bars having varying thickness, which also corresponds to the thickness between the bars. One unique advantage of these bars is that they are all focused on and equally distant to the phantom center. This is taken advantage of by extracting a `CollapsedCircleProfile` about the line pairs. The peaks and valleys of the profile are located; peaks and valleys of each line pair are used to calculated the MTF. The relative MTF (i.e. normalized to the first line pair) is then calculated from these values. See *Modulation Transfer Function*.
- **Calculate Low Contrast Resolution** – Low contrast is inherently difficult to determine since detectability of humans is not simply contrast based. Pylinac's analysis uses both the contrast value of the ROI as well as the ROI size to compute a "detectability" score. ROIs above the score are said to be "seen", while those below are

not seen. Only the 1.0% supra-slice ROIs are examined. Two background ROIs are sampled on either side of the ROI contrast set. See *Visibility* for equation details.

- **Calculate Slice Thickness** – Slice thickness is measured by determining the FWHM of the wire ramps in the CTP404 module. A profile of the area around each wire ramp is taken, and the FWHM is determined from the profile. The profiles are averaged and the value is converted from pixels to mm and multiplied by 0.42 (Catphan manual "Scan Slice Geometry" section). Also see *Slice Thickness*.

Post-Analysis

- **Test if values are within tolerance** – For each module, the determined values are compared with the nominal values. If the difference between the two is below the specified tolerance then the module passes.

6.7.9 Troubleshooting

First, check the general *Troubleshooting* section. Most problems in this module revolve around getting the data loaded.

- If you're having trouble getting your dataset in, make sure you're loading the whole dataset. Also make sure you've scanned the whole phantom.
- Make sure there are no external markers on the CatPhan (e.g. BBs), otherwise the localization algorithm will not be able to properly locate the phantom within the image.
- Ensure that the FOV is large enough to encompass the entire phantom. If the scan is cutting off the phantom in any way it will not identify it.
- The phantom should never touch the edge of an image, see above point.
- Make sure you're loading the right CatPhan class. I.e. using a CatPhan600 class on a CatPhan504 scan may result in errors or erroneous results.

6.7.10 API Documentation

Main classes

These are the classes a typical user may interface with.

class pylinac.ct.**CatPhan504**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A class for loading and analyzing CT DICOM files of a CatPhan 504. Can be from a CBCT or CT scanner Analyzes: Uniformity (CTP486), High-Contrast Spatial Resolution (CTP528), Image Scaling & HU Linearity (CTP404), and Low contrast (CTP515).

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`static run_demo`(*show: bool = True*)

Run the CBCT demo using high-quality head protocol images.

`analyze`(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *low_contrast_tolerance: int | float = 1*, *cnr_threshold: int | float = 15*, *zip_after: bool = False*, *contrast_method: str = 'Michelson'*, *visibility_threshold: float = 0.15*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, int | float] | None = None*)

Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto', which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_origin_slice()` → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`find_phantom_roll`(*func: Callable | None = None*) → float

Determine the "roll" of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

Parameters

func
A callable to sort the air ROIs.

Returns

float : the angle of the phantom in **degrees**.

`classmethod from_demo_images`()

Construct a CBCT object from the demo images.

`classmethod from_url`(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

`classmethod from_zip`(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage(*subimage: str = 'hu'*, *delta: bool = True*, *show: bool = True*) → plt.Figure | None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[list[str]]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *CatphanResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

save_analyzed_subimage(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`class` pylinac.ct.**CatPhan503**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A class for loading and analyzing CT DICOM files of a CatPhan 503. Analyzes: Uniformity (CTP486), High-Contrast Spatial Resolution (CTP528), Image Scaling & HU Linearity (CTP404).

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`static` **run_demo**(*show: bool = True*)

Run the CBCT demo using high-quality head protocol images.

analyze(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *low_contrast_tolerance: int | float = 1*, *cnr_threshold: int | float = 15*, *zip_after: bool = False*, *contrast_method: str = 'Michelson'*, *visibility_threshold: float = 0.15*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, int | float] | None = None*)

Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto', which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

find_origin_slice() → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

find_phantom_axis()

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

find_phantom_roll(*func: Callable | None = None*) → float

Determine the "roll" of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

Parameters

func
A callable to sort the air ROIs.

Returns

float : the angle of the phantom in **degrees**.

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage(*subimage: str = 'hu'*, *delta: bool = True*, *show: bool = True*) → plt.Figure | None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[list[str]]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *CatphanResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

save_analyzed_subimage(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

class pylinac.ct.**CatPhan600**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A class for loading and analyzing CT DICOM files of a CatPhan 600. Analyzes: Uniformity (CTP486), High-Contrast Spatial Resolution (CTP528), Image Scaling & HU Linearity (CTP404), and Low contrast (CTP515).

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`static run_demo`(*show: bool = True*)

Run the CatPhan 600 demo.

`find_phantom_roll`(*func: Callable | None = None*) → float

With the CatPhan 600, we have to consider that the top air ROI has a water vial in it (see pg 12 of the manual). If so, the top air ROI won't be detected. Rather, the default algorithm will find the bottom air ROI and teflon to the left. It may also find the top air ROI if the water vial isn't there. We use the below lambda to select the bottom air and teflon ROIs consistently. These two ROIs are at 75 degrees from cardinal. We thus offset the default outcome by 75.

`analyze`(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *low_contrast_tolerance: int | float = 1*, *cnr_threshold: int | float = 15*, *zip_after: bool = False*, *contrast_method: str = 'Michelson'*, *visibility_threshold: float = 0.15*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, int | float] | None = None*)

Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto', which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_origin_slice()` → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`classmethod from_demo_images()`

Construct a CBCT object from the demo images.

`classmethod from_url(`*url: str*, *check_uid: bool = True*`)`

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage(*subimage: str = 'hu'*, *delta: bool = True*, *show: bool = True*) → plt.Figure | None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[list[str]]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *CatphanResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

save_analyzed_subimage(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

class pylinac.ct.**CatPhan604**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A class for loading and analyzing CT DICOM files of a CatPhan 604. Can be from a CBCT or CT scanner Analyzes: Uniformity (CTP486), High-Contrast Spatial Resolution (CTP528), Image Scaling & HU Linearity (CTP404), and Low contrast (CTP515).

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`static run_demo`(*show: bool = True*)

Run the CBCT demo using high-quality head protocol images.

`analyze`(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *low_contrast_tolerance: int | float = 1*, *cnr_threshold: int | float = 15*, *zip_after: bool = False*, *contrast_method: str = 'Michelson'*, *visibility_threshold: float = 0.15*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, int | float] | None = None*)

Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto',

which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_origin_slice()` → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`find_phantom_roll(`*func: Callable | None = None*`)` → float

Determine the "roll" of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

Parameters

func
A callable to sort the air ROIs.

Returns

float : the angle of the phantom in **degrees**.

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage(*subimage: str = 'hu'*, *delta: bool = True*, *show: bool = True*) → plt.Figure | None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[list[str]]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *CatphanResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

save_analyzed_subimage(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

class pylinac.ct.**CatphanResult**(*catphan_model: str*, *catphan_roll_deg: float*, *origin_slice: int*, *num_images: int*, *ctp404:* CTP404Result, *ctp486:* CTP486Result | *None* = *None*, *ctp528:* CTP528Result | *None* = *None*, *ctp515:* CTP515Result | *None* = *None*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

catphan_model: str

catphan_roll_deg: float

origin_slice: int

num_images: int

ctp404: *CTP404Result*

ctp486: *CTP486Result* **| None = None**

ctp528: *CTP528Result* **| None = None**

ctp515: *CTP515Result* **| None = None**

class pylinac.ct.**CTP404Result**(*offset: int*, *low_contrast_visibility: float*, *thickness_passed: bool*, *measured_slice_thickness_mm: float*, *thickness_num_slices_combined: int*, *geometry_passed: bool*, *avg_line_distance_mm: float*, *line_distances_mm: list[float]*, *hu_linearity_passed: bool*, *hu_tolerance: float*, *hu_rois: dict*)

Bases: `object`

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

offset: int

low_contrast_visibility: float

thickness_passed: bool

measured_slice_thickness_mm: float

thickness_num_slices_combined: int

geometry_passed: bool

avg_line_distance_mm: float

line_distances_mm: list[float]

hu_linearity_passed: bool

hu_tolerance: float

hu_rois: dict

class pylinac.ct.**CTP528Result**(*start_angle_radians: float*, *mtf_lp_mm: dict*, *roi_settings: dict*)

Bases: object

This class should not be called directly. It is returned by the results_data() method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

start_angle_radians: float

mtf_lp_mm: dict

roi_settings: dict

class pylinac.ct.**CTP515Result**(*cnr_threshold: float*, *num_rois_seen: int*, *roi_settings: dict*, *roi_results: dict*)

Bases: object

This class should not be called directly. It is returned by the results_data() method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

cnr_threshold: float

num_rois_seen: int

roi_settings: dict

roi_results: dict

class pylinac.ct.**CTP486Result**(*uniformity_index: float*, *integral_non_uniformity: float*, *passed: bool*, *rois: dict*)

Bases: object

This class should not be called directly. It is returned by the results_data() method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

uniformity_index: float

integral_non_uniformity: float

passed: bool

rois: dict

class pylinac.ct.**ROIResult**(*name: str*, *value: float*, *stdev: float*, *difference: float | None*, *nominal_value: float | None*, *passed: bool | None*)

Bases: object

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

name: str

value: float

stdev: float

difference: float | None

nominal_value: float | None

passed: bool | None

Module classes (CTP404, etc)

class pylinac.ct.**Slice**(*catphan*, *slice_num: int | None = None*, *combine: bool = True*, *combine_method: str = 'mean'*, *num_slices: int = 0*, *clear_borders: bool = True*, *original_image: ImageLike | None = None*)

Bases: object

Base class for analyzing specific slices of a CBCT dicom set.

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property phan_center: *Point*

Determine the location of the center of the phantom.

class pylinac.ct.CatPhanModule(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *Slice*

Base class for a CTP module.

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the slice_num property of subclass.

combine
[bool] If True, combines the slices +/- num_slices around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if combine is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if combine is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

roi_dist_mm

alias of float

roi_radius_mm

alias of float

preprocess(*catphan*)

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

property slice_num: int

The slice number of the spatial resolution module.

Returns

float

plot_rois(*axis: Axes*) → None

Plot the ROIs to the axis.

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

class pylinac.ct.**CTP404CP503**(*catphan*, *offset: int*, *hu_tolerance: float*, *thickness_tolerance: float*, *scaling_tolerance: float*, *clear_borders: bool = True*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, float | int] | None = None*)

Bases: *CTP404CP504*

Alias for namespace consistency

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance. offset : int hu_tolerance : float thickness_tolerance : float scaling_tolerance : float clear_borders : bool

class pylinac.ct.**CTP404CP504**(*catphan*, *offset: int*, *hu_tolerance: float*, *thickness_tolerance: float*, *scaling_tolerance: float*, *clear_borders: bool = True*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, float | int] | None = None*)

Bases: *CatPhanModule*

Class for analysis of the HU linearity, geometry, and slice thickness regions of the CTP404.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance. offset : int hu_tolerance : float thickness_tolerance : float scaling_tolerance : float clear_borders : bool

preprocess(*catphan*) → None

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

property lcv: float

The low-contrast visibility

plot_linearity(*axis: plt.Axes | None = None*, *plot_delta: bool = True*) → tuple

Plot the HU linearity values to an axis.

Parameters

axis
[None, matplotlib.Axes] The axis to plot the values on. If None, will create a new figure.

plot_delta
[bool] Whether to plot the actual measured HU values (False), or the difference from nominal (True).

property passed_hu: bool

Boolean specifying whether all the ROIs passed within tolerance.

plot_rois(*axis: Axes*) → None

Plot the ROIs onto the image, as well as the background ROIs

property passed_thickness: bool

Whether the slice thickness was within tolerance from nominal.

property meas_slice_thickness: float

The average slice thickness for the 4 wire measurements in mm.

property passed_geometry: bool

Returns whether all the line lengths were within tolerance.

class pylinac.ct.**CTP404CP600**(*catphan*, *offset: int*, *hu_tolerance: float*, *thickness_tolerance: float*, *scaling_tolerance: float*, *clear_borders: bool = True*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, float | int] | None = None*)

Bases: *CTP404CP504*

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance. offset : int hu_tolerance : float thickness_tolerance : float scaling_tolerance : float clear_borders : bool

class pylinac.ct.**CTP404CP604**(*catphan*, *offset: int*, *hu_tolerance: float*, *thickness_tolerance: float*, *scaling_tolerance: float*, *clear_borders: bool = True*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, float | int] | None = None*)

Bases: *CTP404CP504*

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance. offset : int hu_tolerance : float thickness_tolerance : float scaling_tolerance : float clear_borders : bool

class pylinac.ct.**CTP528CP503**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CTP528CP504*

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[`Image` or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

class pylinac.ct.**CTP528CP504**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CatPhanModule*

Class for analysis of the Spatial Resolution slice of the CBCT dicom data set.

A collapsed circle profile is taken of the line-pair region. This profile is search for peaks and valleys. The MTF is calculated from those peaks & valleys.

Attributes

radius2linepairs_mm
[float] The radius in mm to the line pairs.

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the slice_num property of subclass.

combine
[bool] If True, combines the slices +/- num_slices around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if combine is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if combine is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

property mtf: MTF

The Relative MTF of the line pairs, normalized to the first region.

Returns

dict

property radius2linepairs: float

Radius from the phantom center to the line-pair region, corrected for pixel spacing.

plot_rois(*axis: Axes*) → None

Plot the circles where the profile was taken within.

property circle_profile: *CollapsedCircleProfile*

Calculate the median profile of the Line Pair region.

Returns

pylinac.core.profile.CollapsedCircleProfile : A 1D profile of the Line Pair region.

class pylinac.ct.**CTP528CP600**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CTP528CP504*

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the slice_num property of subclass.

combine
[bool] If True, combines the slices +/- num_slices around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if combine is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if combine is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

class pylinac.ct.**CTP528CP604**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CTP528CP504*

Alias for namespace consistency.

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the slice_num property of subclass.

combine
[bool] If True, combines the slices +/- num_slices around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if combine is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if combine is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

class pylinac.ct.**CTP515**(*catphan*, *tolerance: float*, *cnr_threshold: float*, *offset: int*, *contrast_method: str*, *visibility_threshold: float*, *clear_borders: bool = True*)

Bases: *CatPhanModule*

Class for analysis of the low contrast slice of the CTP module. Low contrast is measured by obtaining the average pixel value of the contrast ROIs and comparing that value to the average background value. To obtain a more "human" detection level, the contrast (which is largely the same across different-sized ROIs) is multiplied by the diameter. This value is compared to the contrast threshold to decide if it can be "seen".

Parameters

catphan
[`CatPhanBase` instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[`Image` or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

property rois_visible: int

The number of ROIs "visible".

property window_min: float

Lower bound of CT window/leveling to show on the plotted image. Improves apparent contrast.

property window_max: float

Upper bound of CT window/leveling to show on the plotted image. Improves apparent contrast

class pylinac.ct.**CTP486**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CatPhanModule*

Class for analysis of the Uniformity slice of the CTP module. Measures 5 ROIs around the slice that should all be close to the same value.

Parameters

catphan
[CatPhanBase instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the slice_num property of subclass.

combine
[bool] If True, combines the slices +/- num_slices around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if combine is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if combine is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

plot_profiles(*axis: plt.Axes | None = None*) → None

Plot the horizontal and vertical profiles of the Uniformity slice.

Parameters

axis
[None, matplotlib.Axes] The axis to plot on; if None, will create a new figure.

property overall_passed: bool

Boolean specifying whether all the ROIs passed within tolerance.

property uniformity_index: float

The Uniformity Index. Elstrom et al equation 2. https://www.tandfonline.com/doi/pdf/10.3109/0284186X.2011.590525

property integral_non_uniformity: float

The Integral Non-Uniformity. Elstrom et al equation 1. https://www.tandfonline.com/doi/pdf/10.3109/0284186X.2011.590525

ROI Objects

class pylinac.ct.**HUDiskROI**(*array: np.ndarray | ArrayImage, angle: float, roi_radius: float, dist_from_center: float, phantom_center: tuple | Point, nominal_value: float | None = None, tolerance: float | None = None, background_mean: float | None = None, background_std: float | None = None*)

Bases: *DiskROI*

An HU ROI object. Represents a circular area measuring either HU sample (Air, Poly, ...) or HU uniformity (bottom, left, ...).

Parameters

nominal_value
The nominal pixel value of the HU ROI.

tolerance
The roi pixel value tolerance.

property value_diff: float
The difference in HU between measured and nominal.

property passed: bool
Boolean specifying if ROI pixel value was within tolerance of the nominal value.

property plot_color: str
Return one of two colors depending on if ROI passed.

class pylinac.ct.**ThicknessROI**(*array*, *width*, *height*, *angle*, *dist_from_center*, *phantom_center*)

Bases: *RectangleROI*

A rectangular ROI that measures the angled wire rod in the HU linearity slice which determines slice thickness.

Parameters

width
[number] Width of the rectangle. Must be positive

height
[number] Height of the rectangle. Must be positive.

center
[Point, iterable, optional] Center point of rectangle.

as_int
[bool] If False (default), inputs are left as-is. If True, all inputs are converted to integers.

property long_profile: *FWXMProfile*
The profile along the axis perpendicular to ramped wire.

property wire_fwhm: float
The FWHM of the wire in pixels.

property plot_color: str
The plot color.

class pylinac.ct.**GeometricLine**(*geo_roi1:* Point, *geo_roi2:* Point, *mm_per_pixel: float*, *tolerance: int | float*)

Bases: *Line*

Represents a line connecting two nodes/ROIs on the Geometry Slice.

Attributes

nominal_length_mm
[int, float] The nominal distance between the geometric nodes, in mm.

Parameters

geo_roi1
[GEO_ROI] One of two ROIs representing one end of the line.

geo_roi2
[GEO_ROI] The other ROI which is the other end of the line.

mm_per_pixel
[float] The mm/pixel value.

tolerance
[int, float] The tolerance of the geometric line, in mm.

property passed: bool
Whether the line passed tolerance.

property pass_fail_color: str
Plot color for the line, based on pass/fail status.

property length_mm: float
Return the length of the line in mm.

Helper Functions

pylinac.ct.**combine_surrounding_slices**(*dicomstack:* DicomImageStack, *nominal_slice_num: int*, *slices_plusminus: int = 1*, *mode: str = 'mean'*) → array

Return an array that is the combination of a given slice and a number of slices surrounding it.

Parameters

dicomstack
[*~pylinac.core.image.DicomImageStack*] The CBCT DICOM stack.

nominal_slice_num
[int] The slice of interest (along 3rd dim).

slices_plusminus: int
How many slices plus and minus to combine (also along 3rd dim).

mode
[{'mean', 'median', 'max}] Specifies the method of combination.

Returns

combined_array
[numpy.array] The combined array of the DICOM stack slices.

pylinac.ct.**get_regions**(*slice_or_arr: Slice | np.array*, *fill_holes: bool = False*, *clear_borders: bool = True*, *threshold: str = 'otsu'*) → tuple[np.array, list, int]

Get the skimage regions of a black & white image.

6.8 ACR Phantoms

6.8.1 Overview

New in version 3.2.

Warning: These algorithms have only a limited amount of testing data and results should be scrutinized. Further, the algorithm is more likely to change in the future when a more robust test suite is built up. If you'd like to submit data, enter it here.

The ACR module provides routines for automatically analyzing DICOM images of the ACR CT 464 phantom and Large MR phantom. It can load a folder or zip file of images, correcting for translational and rotational offsets.

Phantom reference information is drawn from the ACR CT solution article and the analysis is drawn from the ACR CT testing article. MR analysis is drawn from the ACR Guidance document.

Warning: Due to the rectangular ROIs on the MRI phantom analysis, rotational errors should be <= 1 degree. Translational errors are still accounted for however for any reasonable amount.

6.8.2 Typical Use

The ACR CT and MR analyses follows a similar pattern of load/analyze/output as the rest of the library. Unlike the CatPhan analysis, customization is not a goal, as the phantoms and analyses are much more well-defined. I.e. there's less of a use case for custom phantoms in this scenario. CT is mostly used here but is interchangeable with the MRI class.

To use the ACR analysis, import the class:

```
from pylinac import ACRCT, ACRMRILarge
```

And then load, analyze, and view the results:

- **Load images** – Loading can be done with a directory or zip file:

  ```
  acr_ct_folder = r"C:/CT/ACR/Sept 2021"
  ct = ACRCT(acr_ct_folder)
  acr_mri_folder = r"C:/MRI/ACR/Sept 2021"
  mri = ACRMRILarge(acr_mri_folder)
  ```

  or load from zip:

```
acr_ct_zip = r"C:/CT/ACR/Sept 2021.zip"
ct = ACRCT.from_zip(acr_ct_zip)
```

- **Analyze** – Analyze the dataset:

```
ct.analyze()
```

- **View the results** – Reviewing the results can be done in text or dict format as well as images:

```
# print text to the console
print(ct.results())
# view analyzed image summary
ct.plot_analyzed_image()
# view images independently
ct.plot_images()
# save the images
ct.save_analyzed_image()
# or
ct.save_images()
# finally, save a PDF
ct.publish_pdf()
```

6.8.3 Choosing an MR Echo

With MRI, a dual echo scan can be obtained. These can result in a combined DICOM dataset but are distinct acquisitions. To select between multiple echos, use the `echo_number` parameter:

```
from pylinac import ACRMRILarge

mri = ACRMRILarge(...)  # load zip or dir with dual echo image set
mri.analyze(echo_number=2)
mri.results()
```

If no echo number is passed, the first and lowest echo number is selected and analyzed.

6.8.4 Customizing MR/CT Modules

To customize aspects of the MR analysis modules, subclass the relevant module and set the attribute in the analysis class. E.g. to customize the "Slice1" MR module:

```
from pylinac.acr import ACRMRILarge, MRSlice1Module


class Slice1Modified(MRSlice1Module):
    """Custom location for the slice thickness ROIs"""

    thickness_roi_settings = {
        "Top": {"width": 100, "height": 4, "distance": -3},
        "Bottom": {"width": 100, "height": 4, "distance": 2.5},
    }
```

(continues on next page)

(continued from previous page)

```
# now pass to the MR analysis class
class MyMRI(ACRMRILarge):
    slice1 = Slice1Modified


# use as normal
mri = MyMRI(...)
mri.analyze(...)
```

There are 4 modules in ACR MRI Large analysis that can be overridden. The attribute name should stay the same but the name of the subclassed module can be anything as long as it subclasses the original module:

```
class ACRMRILarge:
    # overload these as you wish. The attribute name cannot change.
    slice1 = MRSlice1Module
    geometric_distortion = GeometricDistortionModule
    uniformity_module = MRUniformityModule
    slice11 = MRSlice11PositionModule


class ACRCT:
    ct_calibration_module = CTModule
    low_contrast_module = LowContrastModule
    spatial_resolution_module = SpatialResolutionModule
    uniformity_module = UniformityModule
```

6.8.5 Customizing module offsets

Customizing the module offsets in the ACR module is easier than for the CT module. To do so, simply override any relevant constant like so:

```
import pylinac

pylinac.acr.MR_SLICE11_MODULE_OFFSET_MM = 95

mri = pylinac.ACRMRILarge(...)  # will use offset above
```

The options for module offsets are as follows along with their default value:

```
# CT
CT_UNIFORMITY_MODULE_OFFSET_MM = 70
CT_SPATIAL_RESOLUTION_MODULE_OFFSET_MM = 100
CT_LOW_CONTRAST_MODULE_OFFSET_MM = 30

# MR
MR_SLICE11_MODULE_OFFSET_MM = 100
MR_GEOMETRIC_DISTORTION_MODULE_OFFSET_MM = 40
MR_UNIFORMITY_MODULE_OFFSET_MM = 60
```

6.8.6 Advanced Use

Using `results_data`

Using the ACR module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *`results_data()`* method which returns a *`ACRCTResult`* instance. For MRI this is *`results_data()`* method and *`ACRMRIResult`* respectively.

Continuing from above:

```
data = ct.results_data()
data.ct_module.roi_radius_mm
# and more

# return as a dict
data_dict = ct.results_data(as_dict=True)
data_dict["ct_module"]["roi_radius_mm"]
...
```

6.8.7 MRI Algorithm

The ACR MR analysis is based on the official guidance document. Because the guidance document is extremely specific (nice job ACR!) only a few highlights are given here. The guidance is followed as reasonably close as possible.

Allowances

- Multiple MR sequences can be present in the dataset.
- The phantom can have significant cartesian shifts.

Restrictions

- There should be 11 slices per scan (although multiple echo scans are allowed) per the guidance document (section 0.3).
- The phantom should have very little pitch, yaw, or roll (<1 degree).

Analysis

Section 0.4 specifies the 7 tests to perform. Pylinac can perform 6 of these 7. It cannot yet perform the low-contrast visibility test.

- **Geometric Accuracy** - The geometric accuracy is measured using profiles of slice 5. The only difference is that pylinac will use the 60th percentile pixel value of the image as a high-pass filter so that minor background fluctuations are removed and then take the FWHM of several profiles of this new image. The width between the two pixels defining the FWHM is the diameter.
- **High Contrast** - High contrast is hard to measure for the ACR MRI phantom simply because it does not use line pairs, but rather offset dots as well as the qualitative description in the guidance document about how to score these. Pylinac measures the high-contrast by sampling a circular ROI on the left ROI (phantom right) set. This is the baseline which all other measurements will be normalized to. The actual dot-ROIs are sampled by taking a circular ROI of the row-based set and the column-based set. Each row-based ROI is evaluated against the other

row-based ROIs. The same is done for column-based ROIs. The ROIs use the maximum and minimum pixel values inside the sample ROI. No dot-counting is performed.

Tip: It is suggested to perform the contrast measurement visually and compare to pylinac values to establish a cross-comparison ratio. After a ratio has been established, the pylinac MTF can be used as the baseline value moving forward.

- **Slice thickness** - Slice thickness is measured using the FWHM of two rectangular ROIs. This is very similar to the guidance document explanation.

  Slice thickness is defined the same as in the guidance document:

  $$Thickness = 0.2 * \frac{Top * Bottom}{Top + Bottom}$$

- **Slice Position** - Slice position accuracy is measured very similarly to the manual method described in the document: "The display level setting … should be set to a level roughly half that of the signal in the bright, all-water portions of the phantom." For each vertical bar, the pixel nearest to the mid-value between min and max of the rectangular ROI is used as the bar position:

  $$position_{bar} = \frac{ROI_{max} - ROI_{min}}{2} + ROI_{min}$$

  The difference in positions between the bars is the value reported.

- **Uniformity** - Uniformity is measured using a circular ROI at the center of the phantom and ROIs to the top, bottom, left, and right of the phantom, very similar to the guidance document.

  The percent integral uniformity (PIU) is defined as:

  $$PIU = 100 * (1 - \frac{high - low}{high + low})$$

  Instead of using the WL/WW to find the low and high 1cm^2 ROI, pylinac uses the 1st and 99th percentile of pixel values inside the central ROI.

  The ghosting ratio is defined the same as the ACR guidance document:

  $$ghosting_{ratio} = |\frac{(top + bottom) - (left + right)}{2 * ROI_{large}}|$$

  where all values are the median pixel values of their respective ROI. The percent-signal ghosting (PSG) is:

  $$PSG = ghosting_{ratio} * 100$$

6.8.8 API Documentation

class pylinac.acr.**ACRCT**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`ct_calibration_module`
alias of `CTModule`

`low_contrast_module`
alias of `LowContrastModule`

`spatial_resolution_module`
alias of `SpatialResolutionModule`

`uniformity_module`
alias of `UniformityModule`

`plot_analyzed_subimage`(**args*, ***kwargs*)
Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

`save_analyzed_subimage`(**args*, ***kwargs*)
Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`analyze`() → None
Analyze the ACR CT phantom

`plot_analyzed_image`(*show: bool = True*, ***plt_kwargs*) → Figure
Plot the analyzed image

Parameters

show
Whether to show the image.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

`save_analyzed_image`(*filename: str | Path | BytesIO*, ***plt_kwargs*) → None
Save the analyzed image to disk or stream

Parameters

filename
Where to save the image to

plt_kwargs
Keywords to pass to matplotlib for figure customization.

`plot_images`(*show: bool = True*, ***plt_kwargs*) → dict[str, Figure]
Plot all the individual images separately

Parameters

show
Whether to show the images.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

save_images(*directory: Path | str | None = None*, *to_stream: bool = False*, ***plt_kwargs*) → list[Path | BytesIO]

Save separate images to disk or stream.

Parameters

directory
The directory to write the images to. If None, will use current working directory

to_stream
Whether to write to stream or disk. If True, will return streams. Directory is ignored in that scenario.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

find_phantom_roll(*func=<function ACRCT.<lambda>>*) → float

Determine the "roll" of the phantom.

Only difference of base method is that we sort the ROIs by size, not by being in the center since the two we're looking for are both right-sided.

results() → str

Return the results of the analysis as a string. Use with print().

results_data(*as_dict=False*) → *ACRCTResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_origin_slice()` → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`classmethod from_demo_images()`

Construct a CBCT object from the demo images.

`classmethod from_url(`*url: str*, *check_uid: bool = True*`)`

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

`classmethod from_zip(`*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*`)`

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

class pylinac.acr.**ACRCTResult**(*phantom_model: str*, *phantom_roll_deg: float*, *origin_slice: int*, *num_images: int*, *ct_module:* CTModuleOutput, *uniformity_module:* UniformityModuleOutput, *low_contrast_module:* LowContrastModuleOutput, *spatial_resolution_module:* SpatialResolutionModuleOutput)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

phantom_model: str

phantom_roll_deg: float

origin_slice: int

num_images: int

ct_module: *CTModuleOutput*

uniformity_module: *UniformityModuleOutput*

low_contrast_module: *LowContrastModuleOutput*

spatial_resolution_module: *SpatialResolutionModuleOutput*

class pylinac.acr.**CTModuleOutput**(*offset: int*, *roi_distance_from_center_mm: int*, *roi_radius_mm: int*, *roi_settings: dict*, *rois: dict*)

Bases: object

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

class pylinac.acr.**UniformityModuleOutput**(*offset: int*, *roi_distance_from_center_mm: int*, *roi_radius_mm: int*, *roi_settings: dict*, *rois: dict*, *center_roi_stdev: float*)

Bases: *CTModuleOutput*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

center_roi_stdev: float

class pylinac.acr.**SpatialResolutionModuleOutput**(*offset: int*, *roi_distance_from_center_mm: int*, *roi_radius_mm: int*, *roi_settings: dict*, *rois: dict*, *lpmm_to_rmtf: dict*)

Bases: *CTModuleOutput*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

lpmm_to_rmtf: dict

class pylinac.acr.**LowContrastModuleOutput**(*offset: int*, *roi_distance_from_center_mm: int*, *roi_radius_mm: int*, *roi_settings: dict*, *rois: dict*, *cnr: float*)

Bases: *CTModuleOutput*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

cnr: float

class pylinac.acr.**ACRMRILarge**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: CatPhanBase

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`slice1`
alias of `MRSlice1Module`

`geometric_distortion`
alias of `GeometricDistortionModule`

`uniformity_module`
alias of `MRUniformityModule`

`slice11`
alias of `MRSlice11PositionModule`

`plot_analyzed_subimage`(**args*, ***kwargs*)
Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

`save_analyzed_subimage`(**args*, ***kwargs*)
Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`localize()` → None

Find the slice number of the catphan's HU linearity module and roll angle

`find_phantom_roll()` → float

Determine the "roll" of the phantom. This algorithm uses the circular left-upper hole on slice 1 as the reference

Returns

float : the angle of the phantom in **degrees**.

`analyze`(*echo_number: int | None = None*) → None

Analyze the ACR CT phantom

Parameters

echo_number:
The echo to analyze. If not passed, uses the minimum echo number found.

`plot_analyzed_image`(*show: bool = True*, ***plt_kwargs*) → Figure

Plot the analyzed image

Parameters

show
Whether to show the image.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

`plot_images`(*show: bool = True*, ***plt_kwargs*) → dict[str, Figure]

Plot all the individual images separately

Parameters

show
Whether to show the images.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

save_images(*directory: Path | str | None = None*, *to_stream: bool = False*, ***plt_kwargs*) → list[Path | BytesIO]

Save separate images to disk or stream.

Parameters

directory
The directory to write the images to. If None, will use current working directory

to_stream
Whether to write to stream or disk. If True, will return streams. Directory is ignored in that scenario.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_str: bool = True*) → str | tuple

Return the results of the analysis as a string. Use with print().

results_data(*as_dict: bool = False*) → *ACRMRIResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

property catphan_size: float

The expected size of the phantom in pixels, based on a 20cm wide phantom.

find_origin_slice() → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

find_phantom_axis()

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

class pylinac.acr.**ACRMRIResult**(*phantom_model: str*, *phantom_roll_deg: float*, *origin_slice: int*, *num_images: int*, *slice1:* MRSlice1ModuleOutput, *slice11:* MRSlice11ModuleOutput, *uniformity_module:* MRUniformityModuleOutput, *geometric_distortion_module:* MRGeometricDistortionModuleOutput)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

phantom_model: str

phantom_roll_deg: float

origin_slice: int

num_images: int

slice1: *MRSlice1ModuleOutput*

slice11: *MRSlice11ModuleOutput*

uniformity_module: *MRUniformityModuleOutput*

geometric_distortion_module: *MRGeometricDistortionModuleOutput*

class pylinac.acr.**MRSlice11ModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *bar_difference_mm: float*, *slice_shift_mm: float*)

Bases: object

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

offset: int

roi_settings: dict

rois: dict

bar_difference_mm: float

slice_shift_mm: float

class pylinac.acr.**MRSlice1ModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *bar_difference_mm: float*, *slice_shift_mm: float*, *measured_slice_thickness_mm: float*, *row_mtf_50: float*, *col_mtf_50: float*)

Bases: object

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

offset: int

roi_settings: dict

rois: dict

bar_difference_mm: float

slice_shift_mm: float

measured_slice_thickness_mm: float

row_mtf_50: float

col_mtf_50: float

class pylinac.acr.**MRUniformityModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *ghost_roi_settings: dict*, *ghost_rois: dict*, *psg: float*, *ghosting_ratio: float*, *piu_passed: bool*, *piu: float*)

Bases: object

This class should not be called directly. It is returned by the results_data() method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

offset: int

roi_settings: dict

rois: dict

ghost_roi_settings: dict

ghost_rois: dict

psg: float

ghosting_ratio: float

piu_passed: bool

piu: float

class pylinac.acr.**MRGeometricDistortionModuleOutput**(*offset: int*, *profiles: dict*, *distances: dict*)

Bases: object

This class should not be called directly. It is returned by the results_data() method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

offset: int

profiles: dict

distances: dict

6.9 "Cheese" Phantoms

New in version 3.9.

Warning: These algorithms have only a limited amount of testing data and results should be scrutinized. Further, the algorithm is more likely to change in the future when a more robust test suite is built up. If you'd like to submit data, enter it here. Thanks!

The Cheese module provides routines for automatically analyzing DICOM images of phantoms commonly called "cheese" phantoms, defined by round phantoms with holes where the user can insert plugs, usually of known density. The primary use case is performing HU calibration or verification although some plugs allow for additional functionality such as spatial resolution. It can load a folder or zip file of images, correcting for translational and rotational offsets.

6.9.1 Phantoms Supported

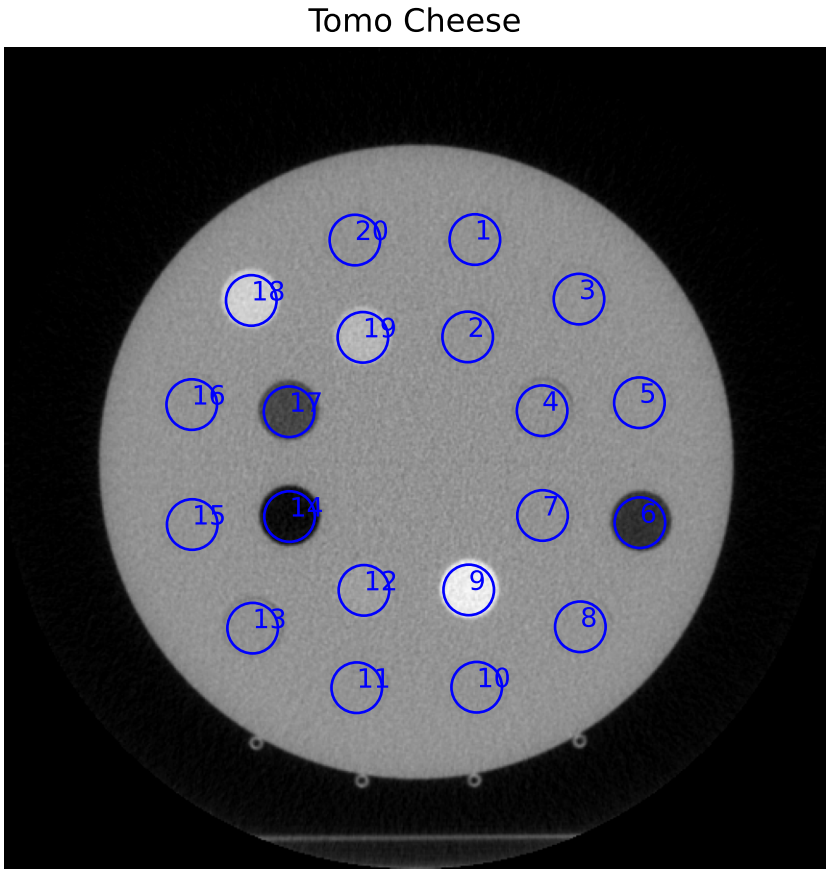
The following phantoms are supported:

- Tomotherapy Cheese
- CIRS Electron Density

6.9.2 Running the Demo

To run one of the Cheese phantom demos, create a script or start an interpreter and input:

```
from pylinac import TomoCheese

TomoCheese.run_demo()
```



Results will be also be printed to the console:

```
- TomoTherapy Cheese Phantom Analysis -
- HU Module -
ROI 1 median: 17.0, stdev: 37.9
ROI 2 median: 20.0, stdev: 44.2
ROI 3 median: 23.0, stdev: 36.9
```

(continues on next page)

(continued from previous page)

```
ROI 4 median: 1.0, stdev: 45.7
ROI 5 median: 17.0, stdev: 37.6
ROI 6 median: -669.0, stdev: 39.6
ROI 7 median: 14.5, stdev: 45.8
ROI 8 median: 26.0, stdev: 38.6
ROI 9 median: 653.0, stdev: 47.4
ROI 10 median: 25.0, stdev: 36.7
ROI 11 median: 24.0, stdev: 35.3
ROI 12 median: 102.0, stdev: 46.2
ROI 13 median: 8.0, stdev: 38.1
ROI 14 median: -930.0, stdev: 43.8
ROI 15 median: 23.0, stdev: 36.3
ROI 16 median: 15.0, stdev: 37.1
ROI 17 median: -516.0, stdev: 45.1
ROI 18 median: 448.0, stdev: 38.1
ROI 19 median: 269.0, stdev: 45.3
ROI 20 median: 15.0, stdev: 37.9
```

6.9.3 Typical Use

The cheese phantom analyses follows a similar pattern of load/analyze/output as the rest of the library. Unlike the CatPhan analysis, tolerances are not applied and comparison to known values is not the goal. There are two reasons for this: 1) The plugs are interchangable and thus the reference values are not necessarily constant. 2) Evaluation against a reference is not the end goal as described in the *philosophy*. Thus, measured values are provided; what you do with them is your business.

To use the Tomo Cheese analysis, import the class:

```
from pylinac import TomoCheese
```

And then load, analyze, and view the results:

- **Load images** – Loading can be done with a directory or zip file:

  ```
  cheese_folder = r"C:/TomoTherapy/QA/September"
  cheese = TomoCheese(cheese_folder)
  ```

  or load from zip:

  ```
  cheese_zip = r"C:/TomoTherapy/QA/September.zip"
  cheese = TomoCheese.from_zip(cheese_zip)
  ```

- **Analyze** – Analyze the dataset:

  ```
  cheese.analyze()
  ```

- **View the results** – Reviewing the results can be done in text or dictionary format as well as images:

  ```
  # print text to the console
  print(cheese.results())
  # return a dictionary or dataclass
  results = cheese.results_data()
  # view analyzed image summary
  ```

(continues on next page)

(continued from previous page)

```
cheese.plot_analyzed_image()
# save the images
cheese.save_analyzed_image()
# finally, save a PDF
cheese.publish_pdf()
```

6.9.4 Plotting density

An HU-to-density curve can be plotted if an ROI configuration is passed to the `analyze` parameter like so:

```
import pylinac

density_info = {
    "1": {"density": 1.0},
    "3": {"density": 3.05},
}  # add more as needed. all keys must have a dict with 'density' defined
tomo = pylinac.TomoCheese(...)
tomo.analyze(roi_config=density_info)
tomo.plot_density_curve()  # in this case, ROI 1 and 3 will be plotted vs the stated␣
↪density
```

This will plot a simple HU vs density graph.

Note: The keys of the configuration must be strings matching the ROI number on the phantom. I.e. 1 matches to "ROI 1", etc.

Note: Not all ROI densities have to be defined. Any ROI between 1 and 20 can be set.

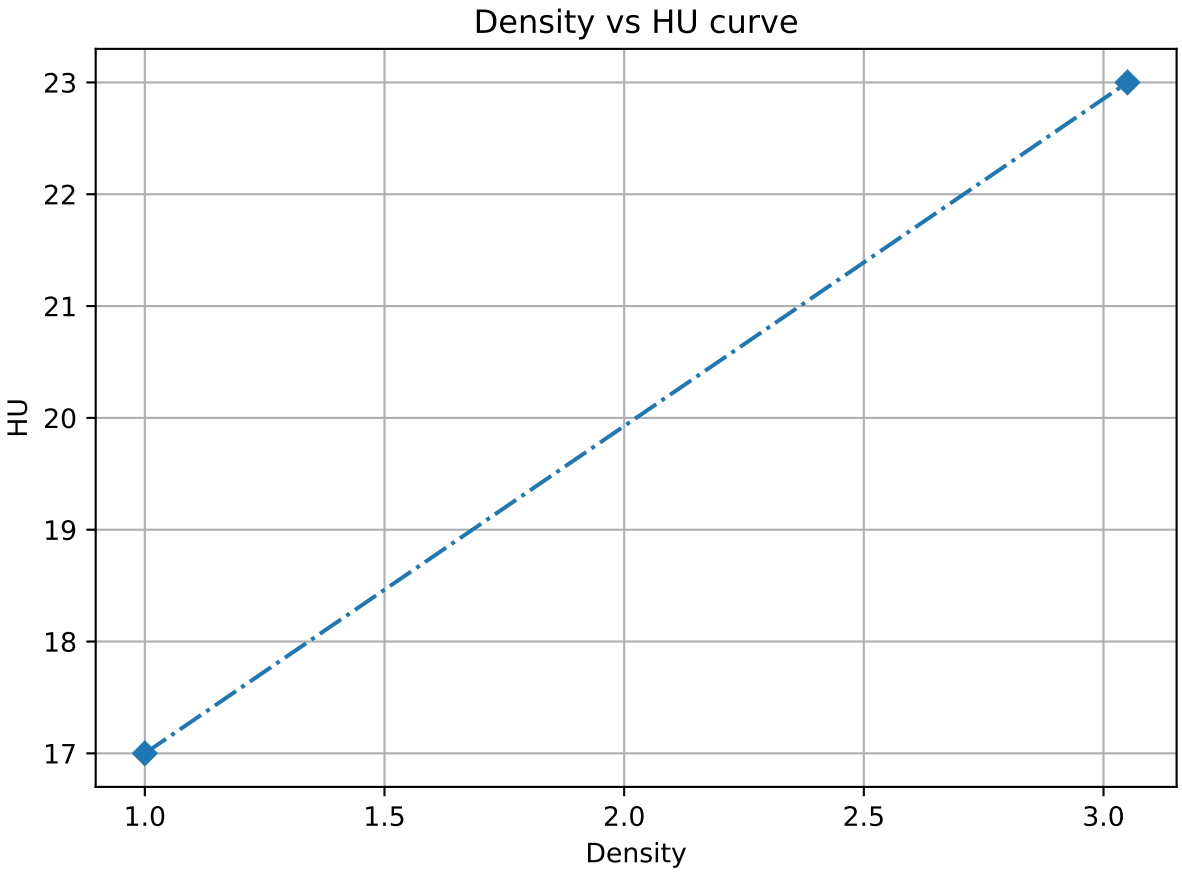
6.9.5 Extending for other phantoms

While new commercial cheese-like phantoms will continue to be added to this module, creating new classes is relatively easy. The following steps show how this can be accomplished.

1. Create a new class "module" that inherits from `CheeseModule`. This class contains information about the ROIs, such as the distance and angle away from the center. You can use the `TomoCheeseModule` as a guide in the source code. An example:

```
from pylinac.cheese import CheeseModule


class SwissCheeseModule(CheeseModule):
    common_name = "Swiss cheese phantom"
    roi_settings = {  # configuration of each ROI.
        "1": {  # each ROI should have a string key and the following keys
            "angle": 90,
            "distance": 45,
            "radius": 6,
        },
```

(continues on next page)


Density vs HU curve
HU
Density
23
22
21
20
19
18
17
1.0
1.5
2.0
2.5
3.0

(continued from previous page)

```
        "2": {
            "angle": 45,
            "distance": 80,
            "radius": 6,
        },
        "3": {...},
    }
```

Note: Not all ROIs have to be defined. E.g. if you are only interested in 5 ROIs out of 20 then simply configure those 5.

2. Create a new class that inherits from `CheesePhantomBase`. This will define the phantom itself:

```
from pylinac.cheese import CheesePhantomBase


class SwissCheesePhantom(CheesePhantomBase):
    model = "Swiss Cheese Phantom"
    # generally this is just the radius of a normal ROI
    air_bubble_radius_mm = 14
    # This is the radius in mm to a "ring" of ROIs that is used for localization␣
↪and roll determination.
    # Generally speaking, set the value to the ring that contains the highest ROI␣
↪HUs.
    localization_radius = 110
    # minimum number of images that should be in the dataset
    min_num_images = 10
    # the radius of the phantom itself
    catphan_radius_mm = 150
    # set this to the module we just created above
    module_class = SwissModule
    # Optional: for the best type inference when using an IDE, set this as well to␣
↪the new module. Note it's only a type annotation!!
    module: SwissModule
```

3. Use the class as normal. The base classes contain all the infrastructure code for analysis and plotting.

```
swiss = SwissCheesePhantom("my/swiss/data")
swiss.analyze()
swiss.plot_analyzed_image()
```

6.9.6 Algorithm

The TomoCheese algorithm leverages a lot of infrastructure from the CatPhan algorithm. It is not based on a manual.

Allowances

- The images can be any size.
- The phantom can have significant translation in all 3 directions.
- The phantom can have significant roll and moderate yaw and pitch.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom cannot touch any edge of the FOV.
- There must be at least one ROI in the "outer" ROI ring that is higher than water/background phantom. This has to do with the automatic roll compensation.

  Note: This is not strictly required but will assist in accurate sub-degree roll compensation.

Pre-Analysis

The pre-analysis is almost exactly the same as the *CatPhan pre-analysis*.

Analysis

- **Determine image properties** – Automatic roll compensation is attempted by creating a circular profile at the radius of the "outer" ROIs. This profile is then searched for peaks, which correspond to high-density plugs that have been inserted. If a peak is not found, no correction is applied and the phantom is assumed to be at 0. This would occur if all plugs have been filled with water/background plugs. If a peak is found, i.e. a plug has been inserted with HU detectably above water, the center of the peak is determined which would correspond to the center of the ROI. The distance to the nearest nominal ROI is calculated. If the value is <5 degrees, the roll compensation is applied. If the value is >5 degrees, the compensation is not applied and the phantom is assumed to be at 0.
- **Measure HU values of each plug** – Based on the nominal spacing and roll compensation (if applied), each plug area is sampled for the median and standard deviation values.

6.9.7 API Documentation

class pylinac.cheese.**TomoCheese**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: *CheesePhantomBase*

A class for analyzing the TomoTherapy 'Cheese' Phantom containing insert holes and plugs for HU analysis.

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

module_class
alias of *TomoCheeseModule*

static run_demo(*show: bool = True*)
Run the Tomotherapy Cheese demo

results_data(*as_dict: bool = False*) → *TomoCheeseResult* | dict
Return the results of the analysis as a structure dataclass

analyze(*roi_config: dict | None = None*) → None
Analyze the Tomo Cheese phantom.

Parameters

roi_config
[dict] The configuration of the ROIs, specifically the known densities.

property catphan_size: float
The expected size of the phantom in pixels, based on a 20cm wide phantom.

find_origin_slice() → int
Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e.

it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

find_phantom_axis()

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

find_phantom_roll(*func: Callable | None = None*) → float

Examine the phantom for the maximum HU delta insert position. Roll the phantom by the measured angle to the nearest nominal angle if nearby. If not nearby, default to 0

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs: dict*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage() → None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

plot_density_curve(*show: bool = True*, ***plt_kwargs: dict*)

Plot the densities of the ROIs vs the measured HU. This will sort the ROIs by measured HU before plotting.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

publish_pdf(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

`save_analyzed_subimage()` → None
Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`class pylinac.cheese.CIRS062M(`*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*`)`

Bases: *CheesePhantomBase*

A class for analyzing the CIRS Electron Density Phantom containing insert holes and plugs for HU analysis.

See Also

https://www.cirsinc.com/products/radiation-therapy/electron-density-phantom/

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`module_class`

alias of *CIRSHUModule*

`classmethod from_demo_images()`

Construct a CBCT object from the demo images.

`find_origin_slice()` → int

We override to lower the minimum variation required. This is ripe for refactor, but I'd like to add a few more phantoms first to get the full picture required.

`analyze(roi_config: dict | None = None)` → None

Analyze the Tomo Cheese phantom.

Parameters

roi_config
[dict] The configuration of the ROIs, specifically the known densities.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`find_phantom_roll(func: Callable | None = None)` → float

Examine the phantom for the maximum HU delta insert position. Roll the phantom by the measured angle to the nearest nominal angle if nearby. If not nearby, default to 0

`classmethod from_url(url: str, check_uid: bool = True)`

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs: dict*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

plot_analyzed_subimage() → None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

`plot_density_curve`(*show: bool = True*, ***plt_kwargs: dict*)

Plot the densities of the ROIs vs the measured HU. This will sort the ROIs by measured HU before plotting.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

`plot_side_view`(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

`publish_pdf`(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`results_data`(*as_dict: bool = False*) → *CheeseResult* | dict

Return the results of the analysis as a structure dataclass

`save_analyzed_image`(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

`save_analyzed_subimage`() → None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`class pylinac.cheese.TomoCheeseModule`(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: `CheeseModule`

The pluggable module with user-accessible holes.

The ROIs of the inner circle are ~45 degrees apart. The ROIs of the outer circle are ~30 degrees apart.

Parameters

catphan
[`CatPhanBase` instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[`Image` or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property phan_center: *Point*

Determine the location of the center of the phantom.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

plot_rois(*axis: Axes*) → None

Plot the ROIs to the axis. We add the ROI # to help the user differentiate

preprocess(*catphan*)

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

roi_dist_mm

alias of `float`

property slice_num: int

The slice number of the spatial resolution module.

Returns

float

class pylinac.cheese.**CIRSHUModule**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: `CheeseModule`

The pluggable module with user-accessible holes.

The ROIs of each circle are ~45 degrees apart.

Parameters

catphan
[`CatPhanBase` instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[`Image` or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property phan_center: *Point*

Determine the location of the center of the phantom.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

plot_rois(*axis: Axes*) → None

Plot the ROIs to the axis. We add the ROI # to help the user differentiate

preprocess(*catphan*)

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

roi_dist_mm
alias of `float`

property slice_num: int
The slice number of the spatial resolution module.

Returns

float

class pylinac.cheese.**CheeseResult**(*origin_slice: int*, *num_images: int*, *phantom_roll: float*, *rois: dict*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

origin_slice: int

num_images: int

phantom_roll: float

rois: dict

class pylinac.cheese.**TomoCheeseResult**(*origin_slice: int*, *num_images: int*, *phantom_roll: float*, *rois: dict*, *roi_1: dict*, *roi_2: dict*, *roi_3: dict*, *roi_4: dict*, *roi_5: dict*, *roi_6: dict*, *roi_7: dict*, *roi_8: dict*, *roi_9: dict*, *roi_10: dict*, *roi_11: dict*, *roi_12: dict*, *roi_13: dict*, *roi_14: dict*, *roi_15: dict*, *roi_16: dict*, *roi_17: dict*, *roi_18: dict*, *roi_19: dict*, *roi_20: dict*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

origin_slice: int

num_images: int

phantom_roll: float

rois: dict

roi_1: dict

roi_2: dict

roi_3: dict

roi_4: dict

roi_5: dict

roi_6: dict

roi_7: dict

roi_8: dict

roi_9: dict

roi_10: dict

roi_11: dict

roi_12: dict

roi_13: dict

roi_14: dict

roi_15: dict

roi_16: dict

roi_17: dict

roi_18: dict

roi_19: dict

roi_20: dict

class pylinac.cheese.**CheesePhantomBase**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A base class for doing cheese-like phantom analysis. A subset of catphan analysis where only one module is assumed.

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`analyze`(*roi_config: dict | None = None*) → None

Analyze the Tomo Cheese phantom.

Parameters

roi_config
[dict] The configuration of the ROIs, specifically the known densities.

`find_phantom_roll`(*func: Callable | None = None*) → float

Examine the phantom for the maximum HU delta insert position. Roll the phantom by the measured angle to the nearest nominal angle if nearby. If not nearby, default to 0

`plot_analyzed_image`(*show: bool = True*, ***plt_kwargs: dict*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis as a string. Use with print().

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`plot_density_curve`(*show: bool = True*, ***plt_kwargs: dict*)

Plot the densities of the ROIs vs the measured HU. This will sort the ROIs by measured HU before plotting.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

`publish_pdf`(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`save_analyzed_subimage`() → None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

`plot_analyzed_subimage`() → None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

results_data(*as_dict: bool = False*) → *CheeseResult* | dict

Return the results of the analysis as a structure dataclass

property catphan_size: float

The expected size of the phantom in pixels, based on a 20cm wide phantom.

find_origin_slice() → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

find_phantom_axis()

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

6.10 Quart

New in version 3.2.

6.10.1 Overview

The Quart module provides routines for automatically analyzing DICOM images of the Quart DVT phantom typically used with the Halcyon linac system. It can load a folder or zip file of images, correcting for translational and rotational offsets.

New in version 3.2.

Warning: These algorithms have only a limited amount of testing data and results should be scrutinized. Further, the algorithm is more likely to change in the future when a more robust test suite is built up. If you'd like to submit data, enter it here.

6.10.2 Typical Use

The Quart phantom analysis follows a similar pattern of load/analyze/output as the rest of the library. Unlike the CatPhan analysis, customization is not a goal, as the phantoms and analyses are much more well-defined. I.e. there's less of a use case for custom phantoms in this scenario.

To use the Quart analysis, import the class:

```
from pylinac import QuartDVT
from pylinac.quart import QuartDVT  # equivalent import
```

And then load, analyze, and view the results:

- **Load images** – Loading can be done with a directory or zip file:

  ```
  quart_folder = r"C:/CT/Quart/Sept 2021"
  quart = QuartDVT(quart_folder)
  ```

  or load from zip:

  ```
  quart_folder = (
      r"C:/CT/Quart/Sept 2021.zip"  # this contains all the DICOM files of the scan
  )
  quart = QuartDVT.from_zip(quart_folder)
  ```

- **Analyze** – Analyze the dataset:

```
quart.analyze()
```

- **View the results** – Reviewing the results can be done in text or dict format as well as images:

```
# print text to the console
print(quart.results())
# view analyzed image summary
quart.plot_analyzed_image()
# view images independently
quart.plot_images()
# save the images
quart.save_images()
# finally, save a PDF
quart.publish_pdf("myquart.pdf")
```

6.10.3 Hypersight

New in version 3.17.

The Hypersight variant of the Quart phantom includes a water ROI in the HU module. A sister class can be used to also analyze this phantom: *HypersightQuartDVT* and will include an additional ROI analysis of the water bubble.

The class can be used interchangeably with the normal class and throughout this documentation.

6.10.4 Advanced Use

Using `results_data`

Using the Quart module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *results_data()* method which returns a *QuartDVTResult* instance.

Continuing from above:

```
data = quart.results_data()
data.hu_module.roi_radius_mm
# and more

# return as a dict
data_dict = quart.results_data(as_dict=True)
data_dict["hu_module"]["roi_radius_mm"]
...
```

6.10.5 Algorithm

The Quart algorithm is nearly the same as the *CBCT Algorithm*. The image loading and localization use the same type of logic.

High-Resolution

For high-resolution resolvability, the Quart manual does describe an equation for calculating the MTF using the line-spread function (LSF) of the phantom edge. For simplicity, we use the Varian Halcyon IPA document, which outlines a similar logic with specific measurements of the -700 -> -200 HU distance using a vertical and horizontal profile.

Within pylinac, to reduce the number of input parameters and also match commissioning values, these are the values used. The result is the distance in mm from these two HU values.

Note: The images in pylinac are "grounded", meaning -1000 -> 0. So the actual algorithm search values are +300 HU (-700 + 1000) and +800 HU (-200 + 1000).

CNR/SNR

While normally the *contrast* algorithm is chosen by the user, for the Quart phantom it is hardcoded based on the equations in the manual. Specifically, contrast to noise is defined as:

$$\frac{|Polystyrene - Acrylic|}{Acrylic}$$

where the values are the median pixel value of the given ROI. Poly was given as a possible recommendations in the Quart user manual. Acrylic is the base material of the phantom, i.e. background.

Note: The numerator is an absolute value.

The signal to noise is defined as:

$$\frac{Polystyrene + 1000}{\sigma_{Polystyrene}}$$

where σ is the standard deviation of the Polystyrene ROI pixel values. The poly ROI was chosen by us to match the selection for the CNR equation.

6.10.6 API Documentation

class pylinac.quart.**QuartDVT**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: `CatPhanBase`

A class for loading and analyzing CT DICOM files of a Quart phantom that comes with the Halcyon. Analyzes: HU Uniformity, Image Scaling & HU Linearity.

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

`hu_module_class`
alias of *QuartHUModule*

`uniformity_module_class`
alias of *QuartUniformityModule*

`geometry_module_class`
alias of *QuartGeometryModule*

`static run_demo`(*show: bool = True*)
Run the Quart algorithm with a head dataset.

`analyze`(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *cnr_threshold: int | float = 5*)
Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto', which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`plot_analyzed_image`(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

`plot_analyzed_subimage`(**args*, ***kwargs*) → None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

results(*as_str: bool = True*) → str | tuple[str, ...]

Return the results of the analysis as a string. Use with print().

results_data(*as_dict: bool = False*) → *QuartDVTResult* | dict

Return results in a data structure for more programmatic use.

plot_images(*show: bool = True*, ***plt_kwargs*) → dict[str, Figure]

Plot all the individual images separately.

Parameters

show
Whether to show the images.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

save_images(*directory: Path | str | None = None*, *to_stream: bool = False*, ***plt_kwargs*) → list[Path] | dict[str, BytesIO]

Save separate images to disk or stream.

Parameters

directory
The directory to write the images to. If None, will use current working directory

to_stream
Whether to write to stream or disk. If True, will return streams. Directory is ignored in that scenario.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

`publish_pdf`(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

`find_origin_slice`() → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

`find_phantom_axis()`

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

`find_phantom_roll`(*func: Callable | None = None*) → float

Determine the "roll" of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

Parameters

func
A callable to sort the air ROIs.

Returns

float : the angle of the phantom in **degrees**.

`classmethod from_demo_images()`

Construct a CBCT object from the demo images.

`classmethod from_url`(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

`classmethod from_zip`(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_side_view(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

save_analyzed_image(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

save_analyzed_subimage(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

class pylinac.quart.**HypersightQuartDVT**(*folderpath: str | Sequence[str] | Path | Sequence[Path] | Sequence[BytesIO]*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Bases: *QuartDVT*

A class for loading and analyzing CT DICOM files of a Quart phantom that comes with the Halcyon, specifically for the Hypersight version, which includes a water ROI. Analyzes: HU Uniformity, Image Scaling & HU Linearity.

Parameters

folderpath
[str, list of strings, or Path to folder] String that points to the CBCT image folder location.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

NotADirectoryError
If folder str passed is not a valid directory.

FileNotFoundError
If no CT images are found in the folder

hu_module
alias of `HypersightQuartHUModule`

hu_module_class
alias of `HypersightQuartHUModule`

analyze(*hu_tolerance: int | float = 40*, *scaling_tolerance: int | float = 1*, *thickness_tolerance: int | float = 0.2*, *cnr_threshold: int | float = 5*)

Single-method full analysis of CBCT DICOM files.

Parameters

hu_tolerance
[int] The HU tolerance value for both HU uniformity and linearity.

scaling_tolerance
[float, int] The scaling tolerance in mm of the geometric nodes on the HU linearity slice (CTP404 module).

thickness_tolerance
[float, int] The tolerance of the thickness calculation in mm, based on the wire ramps in the CTP404 module.

> **Warning:** Thickness accuracy degrades with image noise; i.e. low mAs images are less accurate.

low_contrast_tolerance
[int] The number of low-contrast bubbles needed to be "seen" to pass.

cnr_threshold
[float, int] The threshold for "detecting" low-contrast image. See RTD for calculation info.

Deprecated since version 3.0: Use visibility parameter instead.

zip_after
[bool] If the CT images were not compressed before analysis and this is set to true, pylinac will compress the analyzed images into a ZIP archive.

contrast_method
The contrast equation to use. See *Low contrast*.

visibility_threshold
The threshold for detecting low-contrast ROIs. Use instead of `cnr_threshold`. Follows the Rose equation. See *Visibility*.

thickness_slice_straddle
The number of extra slices **on each side** of the HU module slice to use for slice thickness determination. The rationale is that for thin slices the ramp FWHM can be very noisy. I.e. a 1mm slice might have a 100% variation with a low-mAs protocol. To account for this, slice thicknesses < 3.5mm have 1 slice added on either side of the HU module (so 3 total slices) and then averaged. The default is 'auto', which follows the above logic. Set to an integer to explicitly use a certain amount of padding. Typical values are 0, 1, and 2.

> **Warning:** This is the padding **on either side**. So a value of 1 => 3 slices, 2 => 5 slices, 3 => 7 slices, etc.

expected_hu_values
An optional dictionary of the expected HU values for the HU linearity module. The keys are the ROI names and the values are the expected HU values. If a key is not present or the parameter is None, the default values will be used.

`property catphan_size: float`

The expected size of the phantom in pixels, based on a 20cm wide phantom.

find_origin_slice() → int

Using a brute force search of the images, find the median HU linearity slice.

This method walks through all the images and takes a collapsed circle profile where the HU linearity ROIs are. If the profile contains both low (<800) and high (>800) HU values and most values are the same (i.e. it's not an artifact), then it can be assumed it is an HU linearity slice. The median of all applicable slices is the center of the HU slice.

Returns

int
The middle slice of the HU linearity module.

find_phantom_axis()

We fit all the center locations of the phantom across all slices to a 1D poly function instead of finding them individually for robustness.

Normally, each slice would be evaluated individually, but the RadMachine jig gets in the way of detecting the HU module (). To work around that in a backwards-compatible way we instead look at all the slices and if the phantom was detected, capture the phantom center. ALL the centers are then fitted to a 1D poly function and passed to the individual slices. This way, even if one slice is messed up (such as because of the phantom jig), the poly function is robust to give the real center based on all the other properly-located positions on the other slices.

find_phantom_roll(*func: Callable | None = None*) → float

Determine the "roll" of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

Parameters

func
A callable to sort the air ROIs.

Returns

float : the angle of the phantom in **degrees**.

classmethod from_demo_images()

Construct a CBCT object from the demo images.

classmethod from_url(*url: str*, *check_uid: bool = True*)

Instantiate a CBCT object from a URL pointing to a .zip object.

Parameters

url
[str] URL pointing to a zip archive of CBCT images.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

classmethod from_zip(*zip_file: str | zipfile.ZipFile | BinaryIO*, *check_uid: bool = True*, *memory_efficient_mode: bool = False*)

Construct a CBCT object and pass the zip file.

Parameters

zip_file
[str, ZipFile] Path to the zip file or a ZipFile object.

check_uid
[bool] Whether to enforce raising an error if more than one UID is found in the dataset.

memory_efficient_mode
[bool] Whether to use a memory efficient mode. If True, the DICOM stack will be loaded on demand rather than all at once. This will reduce the memory footprint but will be slower by ~25%. Default is False.

Raises

FileExistsError : If zip_file passed was not a legitimate zip file. FileNotFoundError : If no CT images are found in the folder

geometry_module_class

alias of *QuartGeometryModule*

localize() → None

Find the slice number of the catphan's HU linearity module and roll angle

property mm_per_pixel: float

The millimeters per pixel of the DICOM images.

property num_images: int

The number of images loaded.

plot_analyzed_image(*show: bool = True*, ***plt_kwargs*) → None

Plot the images used in the calculation and summary data.

Parameters

show
[bool] Whether to plot the image or not.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

`plot_analyzed_subimage`(**args*, ***kwargs*) → None

Plot a specific component of the CBCT analysis.

Parameters

subimage
[{'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof', 'side'}] The subcomponent to plot. Values must contain one of the following letter combinations. E.g. `linearity`, `linear`, and `lin` will all draw the HU linearity values.

- `hu` draws the HU linearity image.
- `un` draws the HU uniformity image.
- `sp` draws the Spatial Resolution image.
- `lc` draws the Low Contrast image (if applicable).
- `mtf` draws the RMTF plot.
- `lin` draws the HU linearity values. Used with `delta`.
- `prof` draws the HU uniformity profiles.
- `side` draws the side view of the phantom with lines of the module locations.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

show
[bool] Whether to actually show the plot.

`plot_images`(*show: bool = True*, ***plt_kwargs*) → dict[str, Figure]

Plot all the individual images separately.

Parameters

show
Whether to show the images.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

`plot_side_view`(*axis: Axes*) → None

Plot a view of the scan from the side with lines showing detected module positions

`publish_pdf`(*filename: str | Path*, *notes: str | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_str: bool = True*) → str | tuple[str, ...]

Return the results of the analysis as a string. Use with print().

`results_data`(*as_dict: bool = False*) → *QuartDVTResult* | dict

Return results in a data structure for more programmatic use.

`static run_demo`(*show: bool = True*)

Run the Quart algorithm with a head dataset.

`save_analyzed_image`(*filename: str | Path | BinaryIO*, ***kwargs*) → None

Save the analyzed summary plot.

Parameters

filename
[str, file object] The name of the file to save the image to.

kwargs :
Any valid matplotlib kwargs.

`save_analyzed_subimage`(*filename: str | BinaryIO*, *subimage: str = 'hu'*, *delta: bool = True*, ***kwargs*) → plt.Figure | None

Save a component image to file.

Parameters

filename
[str, file object] The file to write the image to.

subimage
[str] See `plot_analyzed_subimage()` for parameter info.

delta
[bool] Only for use with `lin`. Whether to plot the HU delta or actual values.

save_images(*directory: Path | str | None = None*, *to_stream: bool = False*, ***plt_kwargs*) → list[Path] | dict[str, BytesIO]

Save separate images to disk or stream.

Parameters

directory
The directory to write the images to. If None, will use current working directory

to_stream
Whether to write to stream or disk. If True, will return streams. Directory is ignored in that scenario.

plt_kwargs
Keywords to pass to matplotlib for figure customization.

uniformity_module_class

alias of *QuartUniformityModule*

class pylinac.quart.**QuartHUModule**(*catphan*, *offset: int*, *hu_tolerance: float*, *thickness_tolerance: float*, *scaling_tolerance: float*, *clear_borders: bool = True*, *thickness_slice_straddle: str | int = 'auto'*, *expected_hu_values: dict[str, float | int] | None = None*)

Bases: *CTP404CP504*

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance. offset : int hu_tolerance : float thickness_tolerance : float scaling_tolerance : float clear_borders : bool

property meas_slice_thickness: float

The average slice thickness for the 4 wire measurements in mm.

property signal_to_noise: float

Calculate the SNR based on the suggested procedure in the manual: SNR = (HU + 1000) / sigma, where HU is the mean HU of a chosen insert and sigma is the stdev of the HU insert. We choose to use the Polystyrene as the target HU insert

property contrast_to_noise: float

Calculate the CNR based on the suggested procedure in the manual: CNR = abs(HU_target - HU_background) / sigma, where HU_target is the mean HU of a chosen insert, HU_background is the mean HU of the background insert and sigma is the stdev of the HU background. We choose to use the Polystyrene as the target HU insert and Acrylic (base phantom material) as the background

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property lcv: float

The low-contrast visibility

property passed_geometry: bool

Returns whether all the line lengths were within tolerance.

property passed_hu: bool

Boolean specifying whether all the ROIs passed within tolerance.

property passed_thickness: bool

Whether the slice thickness was within tolerance from nominal.

property phan_center: *Point*

Determine the location of the center of the phantom.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

plot_linearity(*axis: plt.Axes | None = None*, *plot_delta: bool = True*) → tuple

Plot the HU linearity values to an axis.

Parameters

axis
[None, matplotlib.Axes] The axis to plot the values on. If None, will create a new figure.

plot_delta
[bool] Whether to plot the actual measured HU values (False), or the difference from nominal (True).

plot_rois(*axis: Axes*) → None

Plot the ROIs onto the image, as well as the background ROIs

preprocess(*catphan*) → None

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

property slice_num: int

The slice number of the spatial resolution module.

Returns

float

class pylinac.quart.**QuartUniformityModule**(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CTP486*

Class for analysis of the Uniformity slice of the CTP module. Measures 5 ROIs around the slice that should all be close to the same value.

Parameters

catphan
[`CatPhanBase` instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[`Image` or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

property integral_non_uniformity: float

The Integral Non-Uniformity. Elstrom et al equation 1. https://www.tandfonline.com/doi/pdf/10.3109/0284186X.2011.590525

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property overall_passed: bool

Boolean specifying whether all the ROIs passed within tolerance.

property phan_center: *Point*

Determine the location of the center of the phantom.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

plot_profiles(*axis: plt.Axes | None = None*) → None

Plot the horizontal and vertical profiles of the Uniformity slice.

Parameters

axis
[None, matplotlib.Axes] The axis to plot on; if None, will create a new figure.

`plot_rois`(*axis: Axes*) → None
Plot the ROIs to the axis.

`preprocess`(*catphan*)
A preprocessing step before analyzing the CTP module.

Parameters

catphan : ~*pylinac.cbct.CatPhanBase* instance.

`property slice_num: int`
The slice number of the spatial resolution module.

Returns

float

`property uniformity_index: float`
The Uniformity Index. Elstrom et al equation 2. https://www.tandfonline.com/doi/pdf/10.3109/0284186X.2011.590525

`class` pylinac.quart.`QuartGeometryModule`(*catphan*, *tolerance: float | None = None*, *offset: int = 0*, *clear_borders: bool = True*)

Bases: *CatPhanModule*

Class for analysis of the Uniformity slice of the CTP module. Measures 5 ROIs around the slice that should all be close to the same value.

Parameters

catphan
[`CatPhanBase` instance.] The catphan instance.

slice_num
[int] The slice number of the DICOM array desired. If None, will use the `slice_num` property of subclass.

combine
[bool] If True, combines the slices +/- `num_slices` around the slice of interest to improve signal/noise.

combine_method
[{'mean', 'max'}] How to combine the slices if `combine` is True.

num_slices
[int] The number of slices on either side of the nominal slice to combine to improve signal/noise; only applicable if `combine` is True.

clear_borders
[bool] If True, clears the borders of the image to remove any ROIs that may be present.

original_image
[Image or None] The array of the slice. This is a bolt-on parameter for optimization. Leaving as None is fine, but can increase analysis speed if 1) this image is passed and 2) there is no combination of slices happening, which is most of the time.

plot_rois(*axis: Axes*)

Plot the ROIs to the axis.

distances() → dict[str, float]

The measurements of the phantom size for the two lines in mm

high_contrast_resolutions() → dict

The distance in mm from the -700 HU index to the -200 HU index.

This calculates the distance on each edge of the horizontal and vertical geometric profiles for a total of 4 measurements. The result is the average of the 4 values. The DICOM data is already HU-corrected so -1000 => 0. This means we will search for 300 HU (-1000 + 700) and 800 HU (-1000 + 200) respectively.

This cuts the profile in half, searches for the highest-gradient index (where the phantom edge is), then further cuts it down to +/-10 pixels. The 300/800 HU are then found from linear interpolation. It was found that artifacts in the image could drastically influence these values, so hence the +/-10 subset.

Assumptions: -The phantom does not cross the halfway point of the image FOV (i.e. not offset by an obscene amount). -10 pixels about the phantom edge is adequate to capture the full dropoff. -300 and 800 HU values will be in the profile

mean_high_contrast_resolution() → float

Mean high-contrast resolution

is_phantom_in_view() → bool

Whether the phantom appears to be within the slice.

property phan_center: *Point*

Determine the location of the center of the phantom.

property phantom_roi: RegionProperties

Get the Scikit-Image ROI of the phantom

The image is analyzed to see if: 1) the CatPhan is even in the image (if there were any ROIs detected) 2) an ROI is within the size criteria of the catphan 3) the ROI area that is filled compared to the bounding box area is close to that of a circle

plot(*axis: Axes*)

Plot the image along with ROIs to an axis

preprocess(*catphan*)

A preprocessing step before analyzing the CTP module.

Parameters

catphan : *~pylinac.cbct.CatPhanBase* instance.

roi_dist_mm
alias of `float`

roi_radius_mm
alias of `float`

property slice_num: int
The slice number of the spatial resolution module.

Returns

float

class pylinac.quart.**QuartDVTResult**(*phantom_model: str*, *phantom_roll_deg: float*, *origin_slice: int*, *num_images: int*, *hu_module:* QuartHUModuleOutput, *uniformity_module:* QuartUniformityModuleOutput, *geometric_module:* QuartGeometryModuleOutput)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

phantom_model: str

phantom_roll_deg: float

origin_slice: int

num_images: int

hu_module: *QuartHUModuleOutput*

uniformity_module: *QuartUniformityModuleOutput*

geometric_module: *QuartGeometryModuleOutput*

class pylinac.quart.**QuartHUModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *measured_slice_thickness_mm: float*, *signal_to_noise: float*, *contrast_to_noise: float*)

Bases: `object`

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

class pylinac.quart.**QuartUniformityModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *passed: bool*)

Bases: `object`

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

class pylinac.quart.**QuartGeometryModuleOutput**(*offset: int*, *roi_settings: dict*, *rois: dict*, *distances: dict*, *high_contrast_distances: dict*, *mean_high_contrast_distance: float*)

Bases: `object`

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

6.11 Log Analyzer

6.11.1 Overview

The log analyzer module reads and parses Varian linear accelerator machine logs, both Dynalogs and Trajectory logs. The module also calculates actual and expected fluences as well as performing gamma evaluations. Data is structured to be easily accessible and easily plottable.

Unlike most other modules of pylinac, the log analyzer module has no end goal. Data is parsed from the logs, but what is done with that info, and which info is analyzed is up to the user.

Features:

- **Analyze Dynalogs or Trajectory logs** - Either platform is supported. Tlog versions 2.1, 3.0, and 4.0 are supported.
- **Read in both .bin and .txt Trajectory log files** - Read in the machine data from both .bin and .txt files to get all the information recorded. See the `txt` attribute.
- **Save Trajectory log data to CSV** - The Trajectory log binary data format does not allow for easy export of data. Pylinac lets you do that so you can use Excel or other software that you use with Dynalogs.
- **Plot or analyze any axis** - Every data axis (e.g. gantry, y1, beam holds, MLC leaves) can be accessed and plotted: the actual, expected, and even the difference.
- **Calculate fluences and gamma** - Besides reading in the MLC positions, pylinac calculates the actual and expected fluence as well as the gamma map; DTA and threshold values are adjustable.
- **Anonymize logs** - Both dynalogs and trajectory logs can be "anonymized" by removing the Patient ID from the filename(s) and file data.

6.11.2 Concepts

Because the `log_analyzer` module functions without an end goal, the data has been formatted for easy exploration. However, there are a few concepts that should be grasped before diving in.

- **Log Sections** - Upon log parsing, all data is placed into data structures. Varian has designated 4 sections for Trajectory logs: *Header*, *Axis Data*, *Subbeams*, and *CRC*. The *Subbeams* are only applicable for auto-sequenced beams and all v3.0 logs, and the *CRC* is specific to the Trajectory log. The *Header* and *Axis Data* however, are common to both Trajectory logs and Dynalogs.

Note: Dynalogs do not have explicit sections like the Trajectory logs, but pylinac formats them to have these two data structures for consistency.

- **Leaf Indexing & Positions** - Varian leaf identification is 1-index based, over against Python's 0-based indexing. Thus, indexing the first MLC leaf would be `[1]`, not `[0]`.

  Warning: When slicing or analyzing leaf data, keep the Varian 1-index base in mind.

  Leaf data is stored in a dictionary, with the leaf number as the key, from 1 up to the number of MLC leaves. E.g. if the machine has a Millennium 120 standard MLC model, leaf data will have 120 dictionary items from 1 to 120. Leaves from each bank have an offset of half the number of leaves. I.e. leaves A1 and B1 = 1 and 61. Thus, leaves 61-120 correspond to the B-bank, while leaves 1-60 correspond to the A-bank. This can be described by a function $(A_n, B_n) = (n, n + N_{leaves}/2)$, where n is the leaf number and N_{leaves} is the number of leaves.

- **Units** - Units follow the Trajectory log specification: linear axes are in cm, rotational axes in degrees, and MU for dose.

  Note: Dynalog files are inherently in mm for collimator and gantry axes, tenths of degrees for rotational axes, and MLC positions are not at isoplane. For consistency, Dynalog values are converted to Trajectory log specs, meaning linear axes, both collimator and MLCs are in cm at isoplane, and rotational axes are in degrees. Dynalog MU is always from 0 to 25000 no matter the delivered MU (i.e. it's relative), unless it was a VMAT delivery, in which case the gantry position is substituted in the dose fraction column.

  Warning: Dynalog VMAT files replace the dose fraction column with the gantry position. Unfortunately, because of the variable dose rate of Varian linacs the gantry position is not a perfect surrogate for dose, but there is no other choice. Thus, fluence calculations will use the relative gantry movement as the dose in fluence calculations.

- **All data Axes are similar** - Log files capture machine data in "control cycles", aka "snapshots" or "heartbeats". Let's assume a log has captured 100 control cycles. Axis data that was captured will all be similar (e.g. gantry, collimator, jaws). They will all have an *actual* and sometimes an *expected* value for each cycle. Pylinac formats these as 1D numpy arrays along with a difference array if applicable. Each of these arrays can be quickly plotted for visual analysis. See *Axis* for more info.

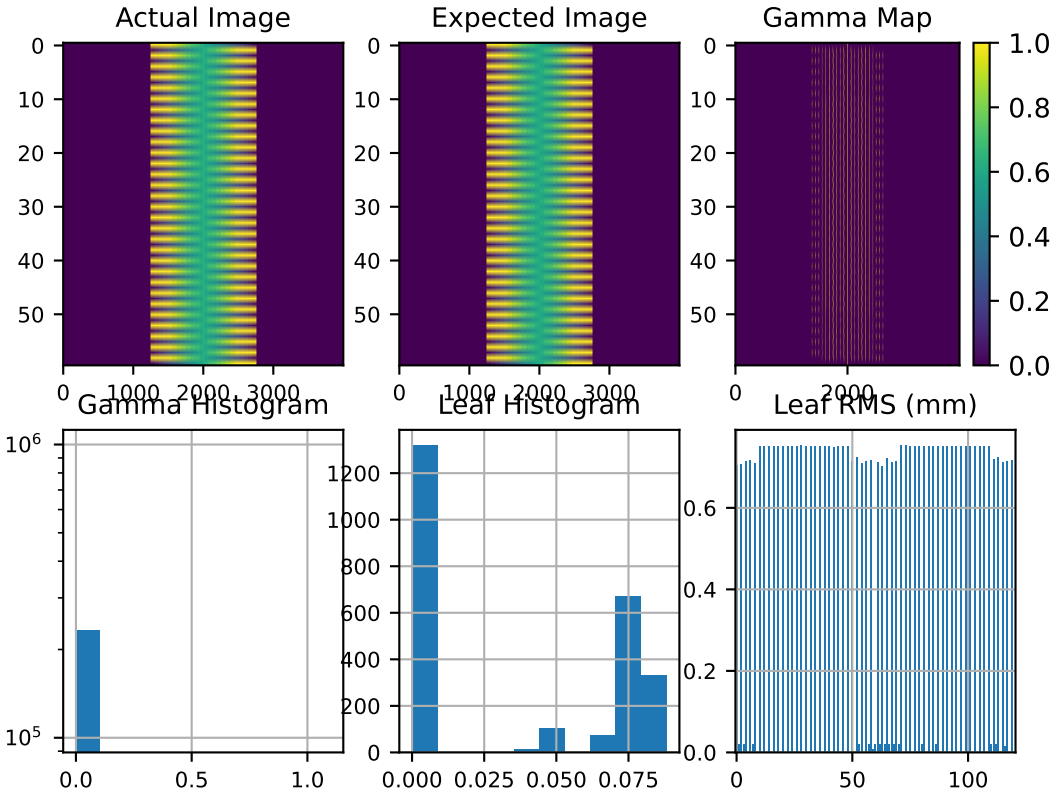
6.11.3 Running the Demos

As usual, the module comes with demo files and methods:

```
from pylinac import Dynalog

Dynalog.run_demo()
```

Which will output the following:

```
Results of file: C:\Users\James\Dropbox\Programming\Python\Projects\pylinac\pylinac\demo_
↪files\AQA.dlg
Average RMS of all leaves: 0.037 cm
Max RMS error of all leaves: 0.076 cm
95th percentile error: 0.088 cm
Number of beam holdoffs: 20
Gamma pass %: 18.65
Gamma average: 0.468
```

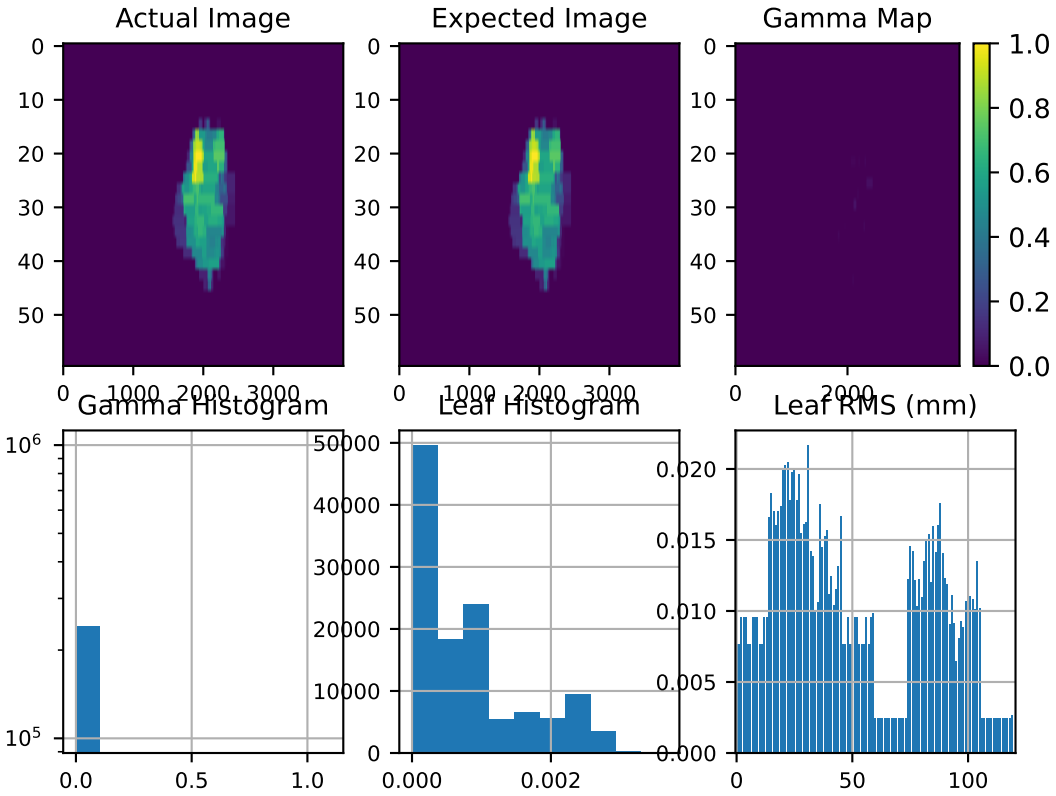

Actual Image
Expected Image
Gamma Map
Gamma Histogram
Leaf Histogram
Leaf RMS (mm)

Your file location will be different, but the values should be the same. The same can be done using the demo Trajectory log:

```
from pylinac import TrajectoryLog

TrajectoryLog.run_demo()
```

Which will give:

```
Results of file: C:\Users\James\Dropbox\Programming\Python\Projects\pylinac\pylinac\demo_
↪files\Tlog.bin
Average RMS of all leaves: 0.001 cm
Max RMS error of all leaves: 0.002 cm
95th percentile error: 0.002 cm
Number of beam holdoffs: 19
Gamma pass %: 100.00
Gamma average: 0.002
```



Note that you can also save data in a PDF report:

```
tlog = ...
tlog.publish_pdf("mytlog.pdf")
```

6.11.4 Loading Data

Loading Single Logs

Logs can be loaded two ways. The first way is through the main helper function *load_log()*.

Note: If you've used pylinac versions <1.6 the helper function is new and can be a replacement for `MachineLog` and `MachineLogs`, depending on the context as discussed below.

```
from pylinac import load_log

log_path = "C:/path/to/tlog.bin"
log = load_log(log_path)
```

In addition, a folder, ZIP archive, or URL can also be passed:

```
log2 = load_log("http://myserver.com/logs/2.dlg")
```

Note: If loading from a URL the object can be a file or ZIP archive.

Pylinac will automatically infer the log type and load it into the appropriate data structures for analysis. The *load_log()* function is a convenient wrapper around the classes within the log analysis module. However, logs can be instantiated a second way: directly through the classes.

```
from pylinac import Dynalog, TrajectoryLog

dlog_path = "C:/path/to/dlog.dlg"
dlog = Dynalog(dlog_path)

tlog_path = "C:/path/to/tlog.bin"
tlog = TrajectoryLog(tlog_path)
```

Loading Multiple Logs

Loading multiple files is also possible using the *load_log()* function as listed above. The logs can also be directly instantiated by using *MachineLogs*. Acceptable inputs include a folder and zip archive.

```
from pylinac import load_log, MachineLogs

path_to_folder = "C:/path/to/dir"

# from folder; equivalent
logs = MachineLogs(path_to_folder)
logs = load_log(path_to_folder)

# from ZIP archive
logs = load_log("path/to/logs.zip")
```

6.11.5 Working with the Data

Working with the log data is straightforward once the data structures and Axes are understood (See *Concepts* for more info). Pylinac follows the data structures specified by Varian for trajectory logs, with a *Header* and *Axis Data* structure, and possibly a *Subbeams* structure if the log is a Trajectory log and was autosequenced. For accessible attributes, see *TrajectoryLog*. The following sections explore each major section of log data and the data structures pylinac creates to assist in data analysis.

Note: It may be helpful to also read the log specification format in parallel with this guide. It is easier to see that pylinac follows the log specifications and where the info comes from. Log specifications are on MyVarian.com.

Working with the Header

Header information is essentially anything that isn't axis measurement data; it's metadata about the file, format, machine configuration, etc. Because of the different file formats, there are separate classes for Trajectory log and Dynalog headers. The classes are:

- *TrajectoryLogHeader*
- *DynalogHeader*

Header attributes are listed in the class API docs by following the above links. For completeness they are also listed here. For Trajectory logs:

- `header`
- `version`
- `header_size`
- `sampling_interval`
- `num_axes`
- `axis_enum`
- `samples_per_axis`
- `num_mlc_leaves`
- `axis_scale`
- `num_subbeams`
- `is_truncated`
- `num_snapshots`
- `mlc_model`

For Dynalogs the following header information is available:

- `version`
- `patient_name`
- `plan_filename`
- `tolerance`
- `num_mlc_leaves`
- `clinac_scale`

Example

Let's explore the header of the demo trajectory log:

```
>>> tlog = TrajectoryLog.from_demo()
>>> tlog.header.header
'VOSTL'
>>> tlog.header.version
2.1
>>> tlog.header.num_subbeams
2
```

Working with Axis Data

Axis data is all the information relating to the measurements of the various machine axes and is accessible under the `axis_data` attribute. This includes the gantry, collimator, MLCs, etc. Trajectory logs capture more information than Dynalogs, and additionally hold the expected positions not only for MLCs but also for all axes. Every measurement axis has *`Axis`* as its base; they all have similar methods to access and plot the data (see *Plotting & Saving Axes/Fluences*). However, not all attributes are axes. Pylinac adds properties to the axis data structure for ease of use (e.g. the number of snapshots) For Trajectory logs the following attributes are available, based on the *`TrajectoryLogAxisData`* class:

- `collimator`
- `gantry`
- `jaws`

  Note: The `jaws` attribute is a data structure to hold all 4 jaw axes; see *`JawStruct`*

- `couch`

  Note: The `couch` attribute is a data structure to hold lateral, longitudinal, etc couch positions; see *`CouchStruct`*

- `mu`
- `beam_hold`
- `control_point`
- `carriage_A`
- `carriage_B`
- `mlc`

  Note: The `mlc` attribute is a data structure to hold leaf information; see *`MLC`* for attributes and the *Working with MLC Data* section for more info.

Dynalogs have similar attributes, derived from the *`DynalogAxisData`* class:

- `collimator`
- `gantry`

- jaws

  Note: The jaws attribute is a data structure to hold all 4 jaw axes; see *JawStruct*

- num_snapshots
- mu
- beam_hold
- beam_on
- previous_segment_num
- prior_dose_index
- next_dose_index
- carriage_A
- carriage_B
- mlc

  Note: The mlc attribute is a data structure to hold leaf information; see *MLC* for attributes and the *Working with MLC Data* section for more info.

Example

Let's access a few axis data attributes:

```
>>> log = Dynalog.from_demo()
>>> log.axis_data.mu.actual  # a numpy array
array([  0, 100, ...
>>> log.axis_data.num_snapshots
99
>>> log.axis_data.gantry.actual
array([ 180, 180, 180, ...
```

Working with MLC Data

Although MLC data is acquired and included in Trajectory logs and Dynalogs, it is not always easy to parse. Additionally, a physicist may be interested in the MLC metrics of a log (RMS, etc). Pylinac provides tools for accessing MLC raw data as well as helper methods and properties via the *MLC* class. Note that this class is consistent between Trajectory logs and Dynalogs. This class is reachable through the axis_data attribute as mlc.

Accessing Leaf data

Leaf data for any leaf is available under the `leaf_axes` attribute which is a dict. The leaves are keyed by the leaf number and the value is an *Axis*. Example:

```
>>> log = Dynalog.from_demo()
>>> log.axis_data.mlc.leaf_axes[1].actual  # numpy array of the 'actual' values for leaf
↪#1
array([ 7.56374, ...
>>> log.axis_data.mlc.leaf_axes[84].difference  # actual values minus the planned values␣
↪for leaf 84
array([-0.001966, ...
```

MLC helper methods/properties

Beyond direct MLC data, pylinac provides a number of helper methods and properties to make working with MLC data easier and more helpful. All the methods are listed in the *MLC* class, but some examples of use are given here:

```
>>> log = Dynalog.from_demo()
>>> log.axis_data.mlc.get_error_percentile(percentile=95)  # get an MLC error percentile␣
↪value
0.08847
>>> log.axis_data.mlc.leaf_moved(12)  # did leaf 12 move during treatment?
False
>>> log.axis_data.mlc.get_RMS_avg()  # get the average RMS error
0.03733
>>> log.axis_data.mlc.get_RMS_avg('A')  # get the average RMS error for bank A
0.03746
>>> log.axis_data.mlc.num_leaves  # the number of MLC leaves
120
>>> log.axis_data.mlc.num_moving_leaves  # the number of leaves that moved during␣
↪treatment
60
```

Working with Fluences

Fluences created by the MLCs can also be accessed and viewed. Fluences are accessible under the `fluence` attribute. There are three subclasses that handle the fluences: The fluence actually delivered is in *ActualFluence*, the fluence planned is in *ExpectedFluence*, and the gamma of the fluences is in *GammaFluence*. Each fluence must be calculated, however pylinac makes reasonable defaults and has a few shortcuts. The actual and expected fluences can be calculated to any resolution in the leaf-moving direction. Some examples:

```
>>> log = Dynalog.from_demo()
>>> log.fluence.actual.calc_map()  # calculate the actual fluence; returns a numpy array
array([ 0, 0, ...
>>> log.fluence.expected.calc_map(resolution=1)  # calculate at 1mm resolution
array([ 0, 0, ...
>>> log.fluence.gamma.calc_map(distTA=0.5, doseTA=1, resolution=0.1)  # change the gamma␣
↪criteria
array([ 0, 0, ...
>>> log.fluence.gamma.pass_prcnt  # the gamma passing percentage
```

(continues on next page)

(continued from previous page)

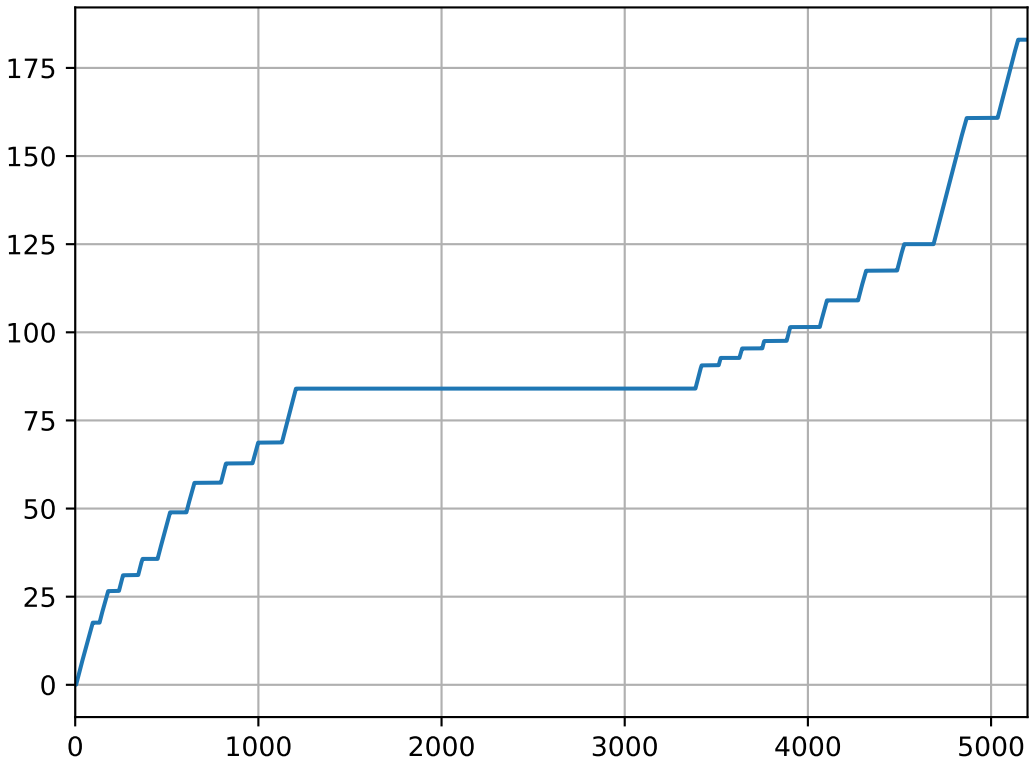
```
99.82
>>> log.fluence.gamma.avg_gamma  # the average gamma value
0.0208
```

Plotting & Saving Axes/Fluences

Each and every axis of the log can be accessed as a numpy array and/or plotted. For each axis the "actual" array/plot is always available. Dynalogs only have expected values for the MLCs. Trajectory logs have the actual and expected values for all axes. Additionally, if an axis has actual and expected arrays, then the difference is also available.

Example of plotting the MU actual:

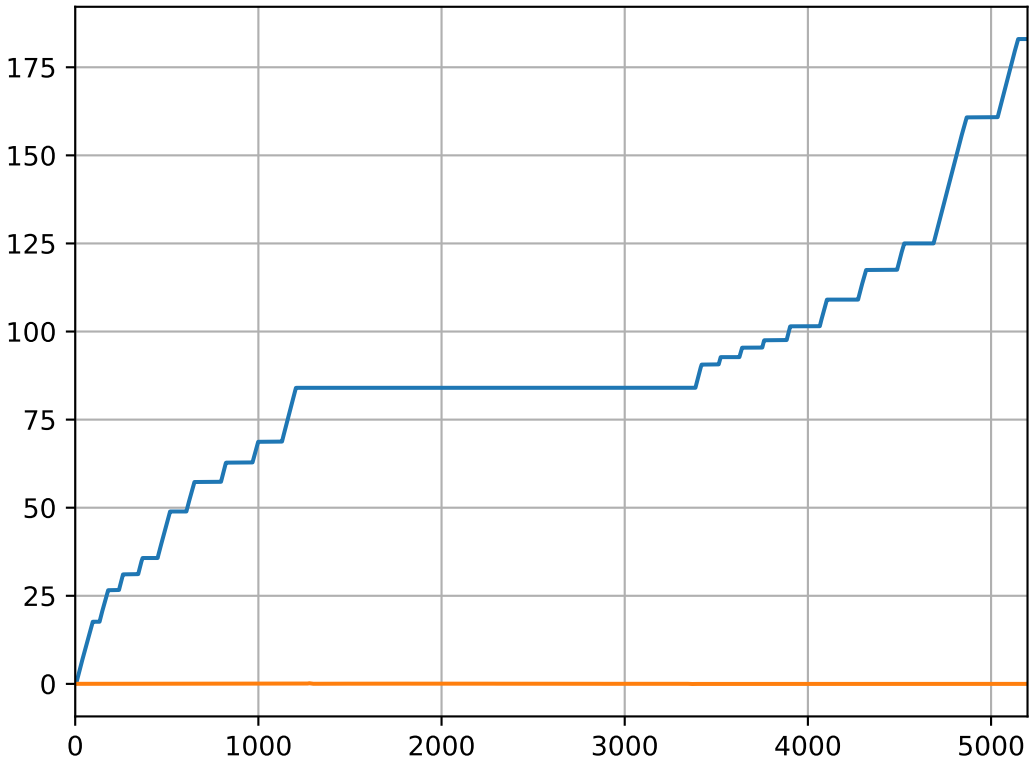
```
import pylinac

log = pylinac.TrajectoryLog.from_demo()
log.axis_data.mu.plot_actual()
```



Plot the Gantry difference:

```
log.axis_data.gantry.plot_difference()
```

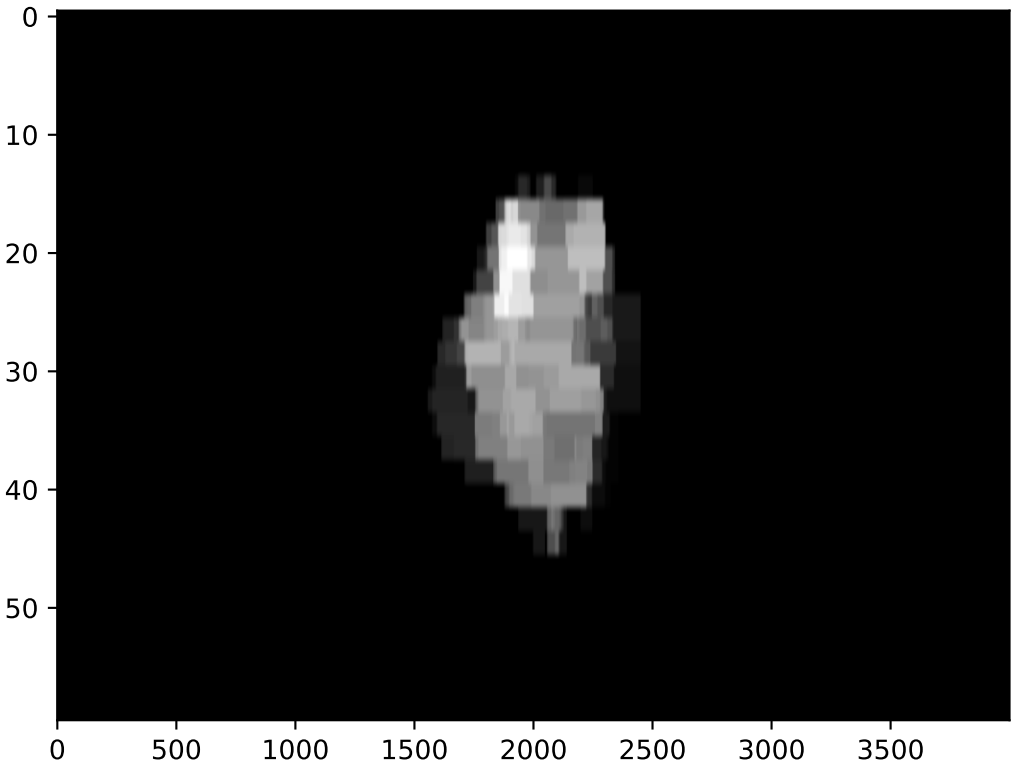
Axis plots are just as easily saved:


175
150
125
100
75
50
25
0
0
1000
2000
3000
4000
5000

```
log.axis_data.gantry.save_plot_difference(filename="gantry diff.png")
```

Now, lets plot the actual fluence:

```
log.fluence.actual.calc_map()
log.fluence.actual.plot_map()
```

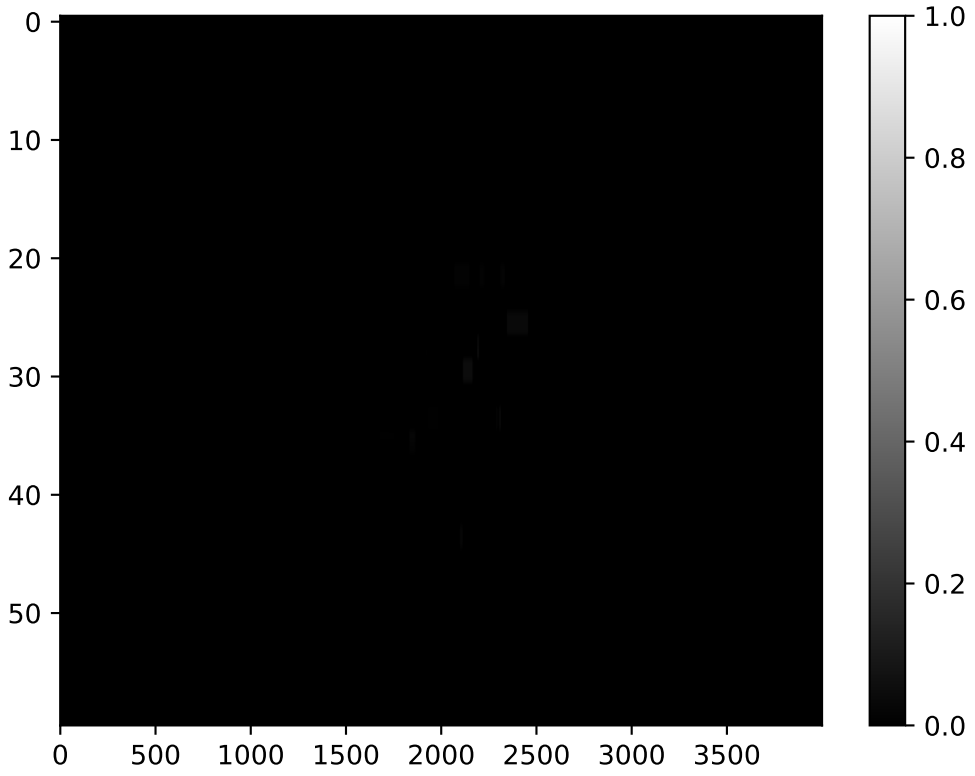


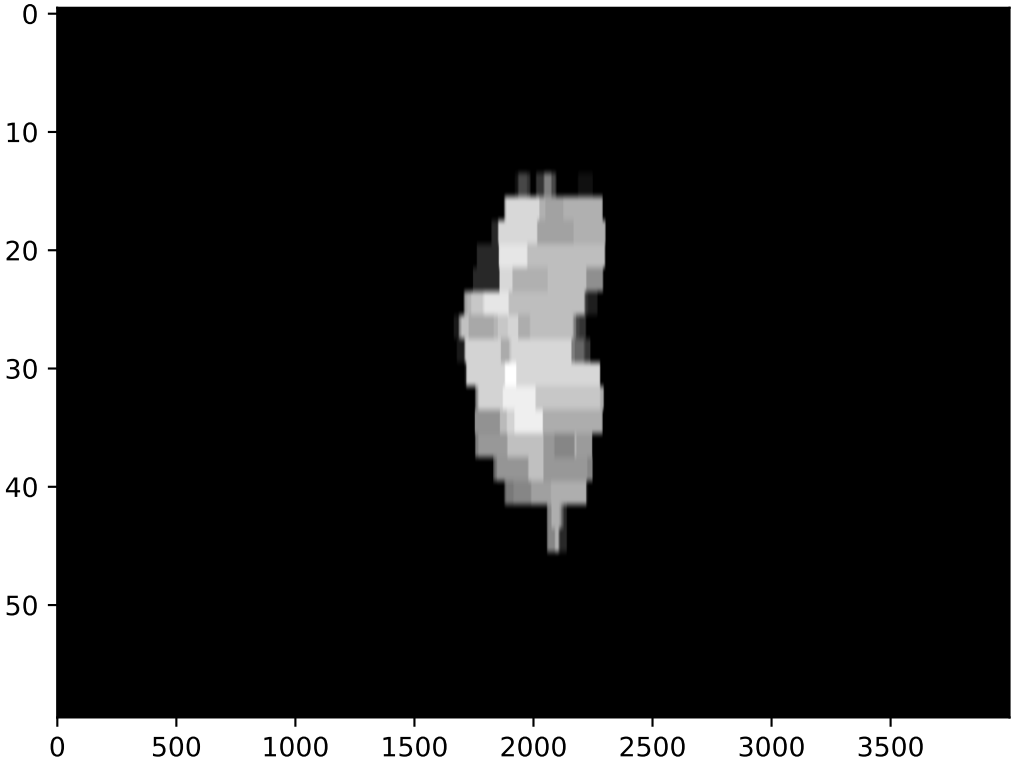
And the fluence gamma. But note we must calculate the gamma first, passing in any DoseTA or DistTA parameters:

```
log.fluence.gamma.calc_map()
log.fluence.gamma.plot_map()
```

Additionally, you can calculate and view the fluences of subbeams if you're working with trajectory logs:

```
log.subbeams[0].fluence.gamma.calc_map()
log.subbeams[0].fluence.actual.plot_map()
```


0
10
20
30
40
50
0
500
1000
1500
2000
2500
3000
3500
1.0
0.8
0.6
0.4
0.2
0.0


0
10
20
30
40
50
0
500
1000
1500
2000
2500
3000
3500

6.11.6 Converting Trajectory logs to CSV

If you already have the log files, you obviously have a record of treatment. However, trajectory logs are in binary format and are not easily readable without tools like pylinac. You can save trajectory logs in a more readable format through the *to_csv()* method. This will write the log to a comma-separated variable (CSV) file, which can be read with Excel and many other programs. You can do further or specialized analysis with the CSV files if you wish, without having to use pylinac:

```
log = TrajectoryLog.from_demo()
log.to_csv()
```

6.11.7 Anonymizing Logs

Machine logs can be anonymized two ways. The first is using the *anonymize()* method, available to both Trajectory logs and Dynalogs. Example script:

```
tlog = TrajectoryLog.from_demo()
tlog.anonymize()
dlog = Dynalog.from_demo()
dlog.anonymize()
```

The other way is the use the module function *anonymize()*. This function will anonymize a single log file or a whole directory. If you plan on anonymizing a lot of logs, use this method as it is threaded and is much faster:

```
from pylinac.log_analyzer import anonymize

log_file = "path/to/tlog.bin"
anonymize(log_file)
log_dir = "path/to/log/folder"
anonymize(log_dir)  # VERY fast
```

6.11.8 Batch Processing

Batch processing/loading of log files is helpful when dealing with one file at a time is too cumbersome. Pylinac allows you to load logs of an entire directory via *MachineLogs*; individual log files can be accessed, and a handful of batch methods are included.

Example

Let's assume all of your logs for the past week are in a folder. You'd like to quickly see what the average gamma is of the files:

```
>>> from pylinac import MachineLogs
>>> log_dir = r"C:\path\to\log\directory"
>>> logs = MachineLogs(log_dir)
>>> logs.avg_gamma(resolution=0.2)
0.03  # or whatever
```

You can also append to *MachineLogs* to have two or more different folders combined:

```
>>> other_log_dir = r"C:\different\path"
>>> logs.append(other_log_dir)
```

Trajectory logs in a MachineLogs instance can also be converted to CSV, just as for a single instance of TrajectoryLog:

```
>>> logs.to_csv()  # only converts trajectory logs; dynalogs are already basically CSV␣
↪files
```

Note: Batch processing methods (like *avg_gamma()* can take a while if numerous logs have been loaded, so be patient. You can also use the `verbose=True` argument in batch methods to see how the process is going.

6.11.9 API Documentation

Main interface

These are the classes and functions a typical user may interface with.

`pylinac.log_analyzer.load_log`(*file_or_dir: str*, *exclude_beam_off: bool = True*, *recursive: bool = True*) → *TrajectoryLog* | *Dynalog* | *MachineLogs*

Load a log file or directory of logs, either dynalogs or Trajectory logs.

Parameters

file_or_dir
[str] String pointing to a single log file or a directory that contains log files.

exclude_beam_off
[bool] Whether to include snapshots where the beam was off.

recursive
[bool] Whether to recursively search a directory. Irrelevant for single log files.

Returns

One of *Dynalog*, *TrajectoryLog*, *MachineLogs*.

`pylinac.log_analyzer.anonymize`(*source: str*, *inplace: bool = False*, *destination: bool | None = None*, *recursive: bool = True*)

Quickly anonymize an individual log or directory of logs. For directories, threaded execution is performed, making this much faster (10-20x) than loading a `MachineLogs` instance of the folder and using the `.anonymize()` method.

Note: Because `MachineLog` instances are not overly memory-efficient, you *may* run into `MemoryError` issues. To avoid this, try not to anonymize more than ~3000 logs at once.

Parameters

source
[str] Points to a local log file (e.g. .dlg or .bin file) or to a directory containing log files.

inplace
[bool] Whether to edit the file itself, or created an anonymized copy and leave the original.

destination
[str, None] Where the put the anonymized logs. Must point to an existing directory. If None, will place the logs in their original location.

recursive
[bool] Whether to recursively enter sub-directories below the root source folder.

class pylinac.log_analyzer.**Dynalog**(*filename*, *exclude_beam_off: bool = True*)

Bases: LogBase

Class for loading, analyzing, and plotting data within a Dynalog file.

Attributes

header : *DynalogHeader* axis_data : *DynalogAxisData* fluence : *FluenceStruct*

property a_logfile: str
Path of the A* dynalog file.

property b_logfile: str
Path of the B* dynalog file.

property num_beamholds: int
Return the number of times the beam was held.

classmethod from_demo(*exclude_beam_off: bool = True*)
Load and instantiate from the demo dynalog file included with the package.

static run_demo()
Run the Dynalog demo.

publish_pdf(*filename: str*, *notes: str = None*, *metadata: dict = None*, *open_file: bool = False*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`static identify_other_file`(*first_dlg_file: str*, *raise_find_error: bool = True*) → str

Return the filename of the corresponding dynalog file.

For example, if the A*.dlg file was passed in, return the corresponding B*.dlg filename. Can find both A- and B-files.

Parameters

first_dlg_file
[str] The absolute file path of the dynalog file.

raise_find_error
[bool] Whether to raise an error if the file isn't found.

Returns

str
The absolute file path to the corresponding dynalog file.

`anonymize`(*inplace: bool = False*, *destination: str | None = None*, *suffix: str | None = None*) → list[str]

Save an anonymized version of the log.

For dynalogs, this replaces the patient ID in the filename(s) and the second line of the log with `Anonymous<suffix>`. This will rename both A* and B* logs if both are present in the same directory.

Note: Anonymization is only available for logs loaded locally (i.e. not from a URL or a data stream). To anonymize such a log it must be first downloaded or written to a file, then loaded in.

Note: Anonymization is done to the log **file** itself. The current instance of `MachineLog` will not be anonymized.

Parameters

inplace
[bool] If False (default), creates an anonymized **copy** of the log(s). If True, **renames and replaces** the content of the log file.

destination
[str, optional] A string specifying the directory where the newly anonymized logs should be placed. If None, will place the logs in the same directory as the originals.

suffix
[str, optional] An optional suffix that is added after `Anonymous` to give specificity to the log.

Returns

list
A list containing the paths to the newly written files.

class pylinac.log_analyzer.**TrajectoryLog**(*filename: str | BinaryIO*, *exclude_beam_off: bool = True*)

Bases: LogBase

A class for loading and analyzing the data of a Trajectory log.

Attributes

header : *~pylinac.log_analyzer.TrajectoryLogHeader*, which has the following attributes: axis_data : *~pylinac.log_analyzer.TrajectoryLogAxisData* fluence : *~pylinac.log_analyzer.FluenceStruct* subbeams : *~pylinac.log_analyzer.SubbeamManager*

property txt_filename: str

The name of the associated .txt file for the .bin file. The file may or may not be available.

anonymize(*inplace: bool = False*, *destination: str | None = None*, *suffix: str | None = None*) → list[str]

Save an anonymized version of the log.

The patient ID in the filename is replaced with Anonymous<suffix> for the .bin file. If the associated .txt file is in the same directory it will similarly replace the patient ID in the filename with Anonymous<suffix>. Additionally, the *Patient ID* row will be replaced with Patient ID: Anonymous<suffix>. For v4+ logs, the Patient ID field in the Metadata structure will be replaced with Anonymous<suffix>.

Note: Anonymization is only available for logs loaded locally (i.e. not from a URL or a data stream). To anonymize such a log it must be first downloaded or written to a file, then loaded in.

Note: Anonymization is done to the log **file** itself. The current instance of MachineLog will not be anonymized.

Parameters

inplace
[bool] If False (default), creates an anonymized **copy** of the log(s). If True, **renames and replaces** the content of the log file.

destination
[str, optional] A string specifying the directory where the newly anonymized logs should be placed. If None, will place the logs in the same directory as the originals.

suffix
[str, optional] An optional suffix that is added after Anonymous to give specificity to the log.

Returns

list
A list containing the paths to the newly written files.

classmethod from_demo(*exclude_beam_off: bool = True*)
Load and instantiate from the demo trajetory log file included with the package.

static run_demo()
Run the Trajectory log demo.

to_csv(*filename: str | None = None*) → str
Write the log to a CSV file.

Parameters

filename
[None, str] If None (default), the CSV filename will be the same as the filename of the log. If a string, the filename will be named so.

Returns

str
The full filename of the newly created CSV file.

publish_pdf(*filename: str | BinaryIO*, *metadata: dict = None*, *notes: str | list = None*, *open_file: bool = False*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

property num_beamholds: int
Return the number of times the beam was held.

property is_hdmlc: bool

Whether the machine has an HDMLC or not.

class pylinac.log_analyzer.**MachineLogs**(*folder: str*, *recursive: bool = True*)

Bases: list

Read in machine logs from a directory. Inherits from list. Batch methods are also provided.

Parameters

folder

[str] The directory of interest. Will walk through and process any logs, Trajectory or dynalog, it finds. Non-log files will be skipped.

recursive

[bool] Whether to walk through subfolders of passed directory. Only used if `folder` is a valid log directory.

Examples

Load a directory upon initialization:

```
>>> log_folder = r'C:\path\log\directory'
>>> logs = MachineLogs(log_folder)
```

Batch methods include determining the average gamma and average gamma pass value:

```
>>> logs.avg_gamma()
>>> 0.05 # or whatever it is
>>> logs.avg_gamma_pct()
>>> 97.2
```

classmethod from_zip(*zfile: str*)

Instantiate from a ZIP archive.

Parameters

zfile

[str] Path to the zip archive.

property num_logs: int

The number of logs currently loaded.

property num_tlogs: int

The number of Trajectory logs currently loaded.

property num_dlogs: int

The number of Trajectory logs currently loaded.

load_folder(*directory: str*, *recursive: bool = True*)

Load log files from a directory and append to existing list.

Parameters

directory
[str, None] The directory of interest. If a string, will walk through and process any logs, Trajectory or dynalog, it finds. Non-log files will be skipped. If None, files must be loaded later using .load_dir() or .append().

recursive
[bool] If True (default), will walk through subfolders of passed directory. If False, will only search root directory.

report_basic_parameters() → None

Report basic parameters of the logs.

- Number of logs
- Average gamma value of all logs
- Average gamma pass percent of all logs

append(*obj*, *recursive: bool = True*) → None

Append a log. Overloads list method.

Parameters

obj
[str, Dynalog, TrajectoryLog] If a string, must point to a log file. If a directory, must contain log files. If a Dynalog or Trajectory log instance, then simply appends.

recursive
[bool] Whether to walk through subfolders of passed directory. Only applicable if obj was a directory.

avg_gamma(*doseTA: int | float = 1*, *distTA: int | float = 1*, *threshold: int | float = 0.1*, *resolution: int | float = 0.1*) → float

Calculate and return the average gamma of all logs. See *calc_map()* for further parameter info.

avg_gamma_pct(*doseTA: int | float = 1*, *distTA: int | float = 1*, *threshold: int | float = 0.1*, *resolution: int | float = 0.1*) → float

Calculate and return the average gamma pass percent of all logs. See *calc_map()* for further parameter info.

to_csv() → list[str]

Write trajectory logs to CSV. If there are both dynalogs and trajectory logs, only the trajectory logs will be written. File names will be the same as the original log file names.

Returns

list
A list of all the filenames of the newly created CSV files.

anonymize(*inplace: bool = False*, *suffix: str | None = None*)

Save anonymized versions of the logs.

For dynalogs, this replaces the patient ID in the filename(s) and the second line of the log with 'Anonymous<suffix>`. This will rename both A* and B* logs if both are present in the same directory.

For trajectory logs, the patient ID in the filename is replaced with *Anonymous<suffix>* for the .bin file. If the associated .txt file is in the same directory it will similarly replace the patient ID in the filename with *Anonymous<suffix>*. Additionally, the *Patient ID* row will be replaced with *Patient ID: Anonymous<suffix>*.

Note: Anonymization is only available for logs loaded locally (i.e. not from a URL or a data stream). To anonymize such a log it must be first downloaded or written to a file, then loaded in.

Note: Anonymization is done to the log *file* itself. The current instance(s) of *MachineLog* will not be anonymized.

Parameters

inplace
[bool] If False (default), creates an anonymized *copy* of the log(s). If True, *renames and replaces* the content of the log file.

suffix
[str, optional] An optional suffix that is added after *Anonymous* to give specificity to the log.

Returns

list
A list containing the paths to the newly written files.

class pylinac.log_analyzer.**Graph**(*value*)

Bases: Enum

An enumeration.

GAMMA = 'gamma'

HISTOGRAM = 'histogram'

RMS = 'rms'

class pylinac.log_analyzer.**MLCBank**(*value*)

Bases: Enum

An enumeration.

A = 'A'

B = 'B'

BOTH = 'both'

class pylinac.log_analyzer.**Fluence**(*value*)

Bases: Enum

An enumeration.

ACTUAL = 'actual'

EXPECTED = 'expected'

GAMMA = 'gamma'

class pylinac.log_analyzer.**TreatmentType**(*value*)

Bases: Enum

An enumeration.

STATIC_IMRT = 'Static IMRT'

DYNAMIC_IMRT = 'Dynamic IMRT'

VMAT = 'VMAT'

IMAGING = 'Imaging'

Supporting Classes

You generally won't have to interface with these unless you're doing advanced behavior.

class pylinac.log_analyzer.**Metadata**(*stream: BinaryIO*, *num_axes: int*)

Bases: object

Metadata field for Trajectory logs v4.0+.

> **Warning:** The TrueBeam log file spec says that there is a reserved section of the same size as v3.0 following this section. That is NOT TRUE. It is actually offset by the size of the metadata; meaning 1024 - (64 + num_axes * 8) - 745.

class pylinac.log_analyzer.**Axis**(*actual: np.ndarray*, *expected: np.ndarray | None = None*)

Bases: object

Represents an 'Axis' of a Trajectory log or dynalog file, holding actual and potentially expected and difference values.

Attributes

Parameters are Attributes

Parameters

actual
[numpy.ndarray] The array of actual position values.

expected
[numpy.ndarray, optional] The array of expected position values. Not applicable for dynalog axes other than MLCs.

property difference: ndarray
Return an array of the difference between actual and expected positions.

Returns

numpy.ndarray
Array the same length as actual/expected.

plot_actual() → None
Plot the actual positions as a matplotlib figure.

plot_expected() → None
Plot the expected positions as a matplotlib figure.

plot_difference() → None
Plot the difference of positions as a matplotlib figure.

class pylinac.log_analyzer.**MLC**(*log_type*, *snapshot_idx: np.ndarray | None = None*, *jaw_struct=None*, *hdmlc: bool = False*, *subbeams=None*)

Bases: object

The MLC class holds MLC information and retrieves relevant data about the MLCs and positions.

Parameters

log_type: *Dynalog*, *TrajectoryLog*
The log type.

snapshot_idx
[array, list] The snapshots to be considered for RMS and error calculations (can be all snapshots or just when beam was on).

jaw_struct
[*JawStruct*] The jaw structure.

hdmlc
[boolean] If False (default), indicates a regular MLC model (e.g. Millennium 120). If True, indicates an HD MLC model (e.g. Millennium 120 HD).

Attributes

leaf_axes
[dict containing *Axis*] The dictionary is keyed by the leaf number, with the Axis as the value.

> **Warning:** Leaf numbers are 1-index based to correspond with Varian convention.

classmethod from_dlog(*dlog*, *jaws*, *snapshot_data: np.ndarray*, *snapshot_idx: list | np.ndarray*)
Construct an MLC structure from a Dynalog

classmethod from_tlog(*tlog*, *subbeams*, *jaws*, *snapshot_data*, *snapshot_idx*, *column_iter*)
Construct an MLC instance from a Trajectory log.

property num_pairs: int
Return the number of MLC pairs.

property num_leaves: int
Return the number of MLC leaves.

property num_snapshots: int
Return the number of snapshots used for MLC RMS & Fluence calculations.

> **Warning:** This number may not be the same as the number of recorded snapshots in the log since the snapshots where the beam was off may not be included. See `MachineLog.load()`

property num_moving_leaves: int
Return the number of leaves that moved.

property moving_leaves: ndarray
Return an array of the leaves that moved during treatment.

add_leaf_axis(*leaf_axis: LeafAxis*, *leaf_num: int*) → None
Add a leaf axis to the MLC data structure.

Parameters

leaf_axis
[LeafAxis] The leaf axis to be added.

leaf_num
[int] The leaf number.

> **Warning:** Leaf numbers are 1-index based to correspond with Varian convention.

leaf_moved(*leaf_num: int*) → bool
Return whether the given leaf moved during treatment.

Parameters

leaf_num : int

> **Warning:** Leaf numbers are 1-index based to correspond with Varian convention.

pair_moved(*pair_num: int*) → bool

Return whether the given pair moved during treatment.

If either leaf moved, the pair counts as moving.

Parameters

pair_num : int

> **Warning:** Pair numbers are 1-index based to correspond with Varian convention.

get_RMS_avg(*bank: MLCBank = MLCBank.BOTH*, *only_moving_leaves: bool = False*)

Return the overall average RMS of given leaves.

Parameters

bank :
Specifies which bank(s) is desired.

only_moving_leaves
[boolean] If False (default), include all the leaves. If True, will remove the leaves that were static during treatment.

> **Warning:** The RMS and error will nearly always be lower if all leaves are included since non-moving leaves have an error of 0 and will drive down the average values. Convention would include all leaves, but prudence would use only the moving leaves to get a more accurate assessment of error/RMS.

Returns

float

get_RMS_max(*bank: MLCBank = MLCBank.BOTH*) → float

Return the overall maximum RMS of given leaves.

Parameters

bank :
Specifies which bank(s) is desired.

Returns

float

get_RMS_percentile(*percentile: int | float = 95*, *bank: MLCBank = MLCBank.BOTH*, *only_moving_leaves: bool = False*)

Return the n-th percentile value of RMS for the given leaves.

Parameters

percentile
[int] RMS percentile desired.

bank :
Specifies which bank(s) is desired.

only_moving_leaves
[boolean] If False (default), include all the leaves. If True, will remove the leaves that were static during treatment.

> **Warning:** The RMS and error will nearly always be lower if all leaves are included since non-moving leaves have an error of 0 and will drive down the average values. Convention would include all leaves, but prudence would use only the moving leaves to get a more accurate assessment of error/RMS.

get_RMS(*leaves_or_bank: str | MLCBank | Iterable*) → np.ndarray

Return an array of leaf RMSs for the given leaves or MLC bank.

Parameters

leaves_or_bank
[sequence of numbers, {'a', 'b', 'both'}] If a sequence, must be a sequence of leaf numbers desired. If a string, it specifies which bank (or both) is desired.

Returns

numpy.ndarray
An array for the given leaves containing the RMS error.

get_leaves(*bank: MLCBank = MLCBank.BOTH*, *only_moving_leaves: bool = False*) → list

Return a list of leaves that match the given conditions.

Parameters

bank
[{'A', 'B', 'both'}] Specifies which bank(s) is desired.

only_moving_leaves
[boolean] If False (default), include all the leaves. If True, will remove the leaves that were static during treatment.

get_error_percentile(*percentile: int | float = 95*, *bank: MLCBank = MLCBank.BOTH*, *only_moving_leaves: bool = False*) → float

Calculate the n-th percentile error of the leaf error.

Parameters

percentile
[int] RMS percentile desired.

bank
[{'A', 'B', 'both'}] Specifies which bank(s) is desired.

only_moving_leaves
[boolean] If False (default), include all the leaves. If True, will remove the leaves that were static during treatment.

> **Warning:** The RMS and error will nearly always be lower if all leaves are included since non-moving leaves have an error of 0 and will drive down the average values. Convention would include all leaves, but prudence would use only the moving leaves to get a more accurate assessment of error/RMS.

create_error_array(*leaves: Sequence[int]*, *absolute: bool = True*) → ndarray

Create and return an error array of only the leaves specified.

Parameters

leaves
[sequence] Leaves desired.

absolute
[bool] If True, (default) absolute error will be returned. If False, error signs will be retained.

Returns

numpy.ndarray
An array of size leaves-x-num_snapshots

create_RMS_array(*leaves: Sequence[int]*) → ndarray

Create an RMS array of only the leaves specified.

Parameters

leaves
[sequence] Leaves desired.

Returns

numpy.ndarray
An array of size leaves-x-num_snapshots

leaf_under_y_jaw(*leaf_num: int*) → bool
Return a boolean specifying if the given leaf is under one of the y jaws.

Parameters

leaf_num : int

get_snapshot_values(*bank_or_leaf:* MLCBank | *Iterable = MLCBank.BOTH*, *dtype: str = 'actual'*) → np.ndarray
Retrieve the snapshot data of the given MLC bank or leaf/leaves

Parameters

bank_or_leaf
[str, array, list] If a str, specifies what bank ('A', 'B', 'both'). If an array/list, specifies what leaves (e.g. [1,2,3])

dtype
[{'actual', 'expected'}] The type of MLC snapshot data to return.

Returns

ndarray
An array of shape (number of leaves - x - number of snapshots). E.g. for an MLC bank and 500 snapshots, the array would be (60, 500).

plot_mlc_error_hist(*show: bool = True*) → None
Plot an MLC error histogram.

save_mlc_error_hist(*filename: str*, ***kwargs*) → None
Save the MLC error histogram to file.

plot_rms_by_leaf(*show: bool = True*) → None
Plot RMSs by leaf.

save_rms_by_leaf(*filename: str*, ***kwargs*) → None
Save the RMS-leaf to file.

class pylinac.log_analyzer.**DynalogHeader**(*dlogdata*)
Bases: *Structure*

Attributes

version
[str] The Dynalog version letter.

patient_name
[str] Patient information.

plan_filename
[str] Filename if using standalone. If using Treat =<6.5 will produce PlanUID, Beam Number. Not yet implemented for this yet.

tolerance
[int] Plan tolerance.

num_mlc_leaves
[int] Number of MLC leaves.

clinac_scale
[int] Clinac scale; 0 -> Varian scale, 1 -> IEC 60601-2-1 scale

`class` pylinac.log_analyzer.**DynalogAxisData**(*log*, *dlogdata*)

Bases: `object`

Attributes

num_snapshots
[int] Number of snapshots recorded.

mu
[*Axis*] Current dose fraction

> **Note:** This *can* be gantry rotation under certain conditions. See Dynalog file specs.

previous_segment_num
[*Axis*] Previous segment *number*, starting with zero.

beam_hold
[*Axis*] Beam hold state; 0 -> holdoff not asserted (beam on), 1 -> holdoff asserted, 2 -> carriage in transition

beam_on
[*Axis*] Beam on state; 1 -> beam is on, 0 -> beam is off

prior_dose_index
[*Axis*] Previous segment dose index or previous segment gantry angle.

next_dose_index
[*Axis*] Next segment dose index.

gantry
[*Axis*] Gantry data in degrees.

collimator
[*Axis*] Collimator data in degrees.

jaws
[`Jaw_Struct`] Jaw data structure. Data in cm.

carriage_A
[*Axis*] Carriage A data. Data in cm.

carriage_B
[*Axis*] Carriage B data. Data in cm.

mlc
[*MLC*] MLC data structure. Data in cm.

Read the dynalog axis data.

class pylinac.log_analyzer.**TrajectoryLogHeader**(*file: BinaryIO*)

Bases: object

Attributes

header
[str] Header signature: 'VOSTL'.

version
[str] Log version.

header_size
[int] Header size; fixed at 1024.

sampling_interval
[int] Sampling interval in milliseconds.

num_axes
[int] Number of axes sampled.

axis_enum
[int] Axis enumeration; see the Tlog file specification for more info.

samples_per_axis
[numpy.ndarray] Number of samples per axis; 1 for most axes, for MLC it's # of leaves and carriages.

num_mlc_leaves
[int] Number of MLC leaves.

axis_scale
[int] Axis scale; 1 -> Machine scale, 2 -> Modified IEC 61217.

num_subbeams
[int] Number of subbeams, if autosequenced.

is_truncated
[int] Whether log was truncated due to space limitations; 0 -> not truncated, 1 -> truncated

num_snapshots
[int] Number of snapshots, cycles, heartbeats, or whatever you'd prefer to call them.

mlc_model
[int] The MLC model; 2 -> NDS 120 (e.g. Millennium), 3 -> NDS 120 HD (e.g. Millennium 120 HD)

class pylinac.log_analyzer.**TrajectoryLogAxisData**(*log*, *file*, *subbeams*)

Bases: object

Attributes

collimator
[*Axis*] Collimator data in degrees.

gantry
[*Axis*] Gantry data in degrees.

jaws
[*JawStruct*] Jaw data structure. Data in cm.

couch
[*CouchStruct*] Couch data structure. Data in cm.

mu
[*Axis*] MU data in MU.

beam_hold
[*Axis*] Beam hold state. Beam *pauses* (e.g. Beam Off button pressed) are not recorded in the log. Data is automatic hold state. 0 -> Normal; beam on. 1 -> Freeze; beam on, dose servo is temporarily turned off. 2 -> Hold; servo holding beam. 3 -> Disabled; beam on, dose servo is disable via Service.

control_point
[*Axis*] Current control point.

carriage_A
[*Axis*] Carriage A data in cm.

carriage_B
[*Axis*] Carriage B data in cm.

mlc
[*MLC*] MLC data structure; data in cm.

`class pylinac.log_analyzer.SubbeamManager(file, header)`

Bases: `object`

One of 4 subsections of a trajectory log. Holds a list of Subbeams; only applicable for auto-sequenced beams.

`post_hoc_metadata(axis_data)`

From the Axis Data, perform post-hoc analysis and set metadata to the subbeams. Gives the subbeams more information, as not much is given directly in the logs.

`class pylinac.log_analyzer.Subbeam(file, log_version: float)`

Bases: `object`

Data structure for trajectory log "subbeams". Only applicable for auto-sequenced beams.

Attributes

control_point
[int] Internally-defined marker that defines where the plan is currently executing.

mu_delivered
[float] Dose delivered in units of MU.

rad_time
[float] Radiation time in seconds.

sequence_num
[int] Sequence number of the subbeam.

beam_name
[str] Name of the subbeam.

property gantry_angle: float
Median gantry angle of the subbeam.

property collimator_angle: float
Median collimator angle of the subbeam.

property jaw_x1: float
Median X1 position of the subbeam.

property jaw_x2: float
Median X2 position of the subbeam.

property jaw_y1: float
Median Y1 position of the subbeam.

property jaw_y2: float
Median Y2 position of the subbeam.

class pylinac.log_analyzer.**FluenceStruct**(*mlc_struct=None*, *mu_axis:* Axis | *None = None*, *jaw_struct=None*)

Bases: object

Structure for data and methods having to do with fluences.

Attributes

actual
[*FluenceBase*] The actual fluence delivered.

expected
[*FluenceBase*] The expected, or planned, fluence.

gamma
[*GammaFluence*] The gamma structure regarding the actual and expected fluences.

class pylinac.log_analyzer.**FluenceBase**(*mlc_struct=None*, *mu_axis:* Axis | *None = None*, *jaw_struct=None*)

Bases: object

An abstract base class to be used for the actual and expected fluences.

Attributes

array
[numpy.ndarray] An array representing the fluence map; will be num_mlc_pairs-x-400/resolution. E.g., assuming a Millennium 120 MLC model and a fluence resolution of 0.1mm, the resulting matrix will be 60-x-4000.

resolution
[int, float] The resolution of the fluence calculation; -1 means calculation has not been done yet.

Parameters

mlc_struct : MLC_Struct mu_axis : BeamAxis jaw_struct : Jaw_Struct

is_map_calced(*raise_error: bool = False*) → bool

Return a boolean specifying whether the fluence has been calculated.

calc_map(*resolution: float = 0.1*, *equal_aspect: bool = False*) → ndarray

Calculate a fluence pixel map.

Image calculation is done by adding fluence snapshot by snapshot, and leaf pair by leaf pair. Each leaf pair is analyzed separately. First, to optimize, it checks if the leaf is under the y-jaw. If so, the fluence is left at zero; if not, the leaf (or jaw) ends are determined and the MU fraction of that snapshot is added to the total fluence. All snapshots are iterated over for each leaf pair until the total fluence matrix is built.

Parameters

resolution
[int, float] The resolution in mm of the fluence calculation in the leaf-moving direction.

equal_aspect
[bool] If True, make the y-direction the same resolution as x. If False, the y-axis will be equal to the number of leaves.

Returns

numpy.ndarray
A numpy array reconstructing the actual fluence of the log. The size will be the number of MLC pairs by 400 / resolution since the MLCs can move anywhere within the 40cm-wide linac head opening.

plot_map(*show: bool = True*) → None

Plot the fluence; the fluence (pixel map) must have been calculated first.

save_map(*filename: str*, ***kwargs*) → None

Save the fluence map figure to a file.

class pylinac.log_analyzer.**ActualFluence**(*mlc_struct=None*, *mu_axis: Axis | None = None*, *jaw_struct=None*)

Bases: *FluenceBase*

The actual fluence object

Parameters

mlc_struct : MLC_Struct mu_axis : BeamAxis jaw_struct : Jaw_Struct

class pylinac.log_analyzer.**ExpectedFluence**(*mlc_struct=None*, *mu_axis:* Axis | *None = None*, *jaw_struct=None*)

Bases: *FluenceBase*

The expected fluence object.

Parameters

mlc_struct : MLC_Struct mu_axis : BeamAxis jaw_struct : Jaw_Struct

class pylinac.log_analyzer.**GammaFluence**(*actual_fluence:* ActualFluence, *expected_fluence:* ExpectedFluence, *mlc_struct*)

Bases: *FluenceBase*

Gamma object, including pixel maps of gamma, binary pass/fail pixel map, and others.

Attributes

array
[numpy.ndarray] The gamma map. Only available after calling calc_map()

passfail_array
[numpy.ndarray] The gamma pass/fail map; pixels that pass (<1.0) are set to 0, while failing pixels (>=1.0) are set to 1.

distTA
[int, float] The distance to agreement value used in gamma calculation.

doseTA
[int, float] The dose to agreement value used in gamma calculation.

threshold
[int, float] The threshold percent dose value, below which gamma was not evaluated.

pass_prcnt
[float] The percent of pixels passing gamma (<1.0).

avg_gamma
[float] The average gamma value.

Parameters

actual_fluence
[ActualFluence] The actual fluence object.

expected_fluence
[ExpectedFluence] The expected fluence object.

mlc_struct
[MLC_Struct] The MLC structure, so fluence can be calculated from leaf positions.

calc_map(*doseTA: int | float = 1*, *distTA: int | float = 1*, *threshold: int | float = 0.1*, *resolution: int | float = 0.1*, *calc_individual_maps: bool = False*) → np.ndarray

Calculate the gamma from the actual and expected fluences.

The gamma calculation is based on Bakai et al eq.6, which is a quicker alternative to the standard Low gamma equation.

Parameters

doseTA
[int, float] Dose-to-agreement in percent; e.g. 2 is 2%.

distTA
[int, float] Distance-to-agreement in mm.

threshold
[int, float] The dose threshold percentage of the maximum dose, below which is not analyzed.

resolution
[int, float] The resolution in mm of the resulting gamma map in the leaf-movement direction.

calc_individual_maps
[bool] Not yet implemented. If True, separate pixel maps for the distance-to-agreement and dose-to-agreement are created.

Returns

numpy.ndarray
A num_mlc_leaves-x-400/resolution numpy array.

plot_map(*show: bool = True*)

Plot the fluence; the fluence (pixel map) must have been calculated first.

histogram(*bins: list | None = None*) → tuple[np.ndarray, np.ndarray]

Return a histogram array and bin edge array of the gamma map values.

Parameters

bins
[sequence] The bin edges for the gamma histogram; see numpy.histogram for more info.

Returns

histogram
[numpy.ndarray] A 1D histogram of the gamma values.

bin_edges
[numpy.ndarray] A 1D array of the bin edges. If left as None, the class default will be used (self.bins).

plot_histogram(*scale: str = 'log'*, *bins: list | None = None*, *show: bool = True*) → None

Plot a histogram of the gamma map values.

Parameters

scale
[{'log', 'linear'}] Scale of the plot y-axis.

bins
[sequence] The bin edges for the gamma histogram; see numpy.histogram for more info.

save_histogram(*filename: str*, *scale: str = 'log'*, *bins: list | None = None*, ***kwargs*) → None
Save the histogram plot to file.

plot_passfail_map() → None
Plot the binary gamma map, only showing whether pixels passed or failed.

class pylinac.log_analyzer.**JawStruct**(*x1: HeadAxis*, *y1: HeadAxis*, *x2: HeadAxis*, *y2: HeadAxis*)
Bases: object

Jaw Axes data structure.

Attributes

x1 : *Axis* y1 : *Axis* x2 : *Axis* y2 : *Axis*

class pylinac.log_analyzer.**CouchStruct**(*vertical: CouchAxis*, *longitudinal: CouchAxis*, *lateral: CouchAxis*, *rotational: CouchAxis*, *pitch: CouchAxis | None = None*, *roll: CouchAxis | None = None*)
Bases: object

Couch Axes data structure.

class pylinac.log_analyzer.**NotALogError**
Bases: OSError

Machine log error. Indicates that the passed file is not a valid machine log file.

class pylinac.log_analyzer.**NotADynalogError**
Bases: OSError

Dynalog error. Indicates that the passed file is not a valid dynalog file.

class pylinac.log_analyzer.**DynalogMatchError**
Bases: OSError

Dynalog error. Indicates that the associated file of the dynalog passed in (A file if B passed in & vic versa) cannot be found. Ensure associated file is in the same folder and has the same name as the passed file, except the first letter.

6.12 Picket Fence

6.12.1 Overview

The picket fence module is meant for analyzing EPID images where a "picket fence" MLC pattern has been made. Physicists regularly check MLC positioning through this test. This test can be done using film and one can "eyeball" it, but this is the 21st century and we have numerous ways of quantifying such data. This module attains to be one of them. It can load in an EPID dicom image (or superimpose multiple images) and determine the MLC peaks, error of each MLC pair to the picket, and give a few visual indicators for passing/warning/failing.

Features:

- **Analyze any MLC type** - Both default MLCs and custom MLCs can be used.
- **Easy-to-read pass/warn/fail overlay** - Analysis gives you easy-to-read tools for determining the status of an MLC pair.
- **Any Source-to-Image distance** - Whatever your clinic uses as the SID for picket fence, pylinac can account for it.
- **Account for panel translation** - Have an off-CAX setup? No problem. Translate your EPID and pylinac knows.
- **Account for panel sag** - If your EPID sags at certain angles, just tell pylinac and the results will be shifted.

6.12.2 Concepts

Although most terminology will be familiar to a clinical physicist, it is still helpful to be clear about what means what. A "picket" is the line formed by several MLC pairs all at the same position. There is usually some ideal gap between the MLCs, such as 0.5, 1, or 2 mm. An "MLC position" is, for pylinac's purposes, the center of the FWHM of the peak formed by one MLC pair at one picket. Thus, one picket fence image may have anywhere between a few to a dozen pickets, formed by as few as 10 MLC pairs up to all 60 pairs.

Pylinac presents the analyzed image in such a way that allows for quick assessment; additionally, all elements atop the image can optionally be turned off. Pylinac by default will plot the image, the determined MLC positions, "guard rails", and a semi-transparent overlay of the MLC error magnitude and translucent boxes over failed leaves. The guard rails are two lines parallel to the fitted picket or side of the picket, offset by the tolerance passed to *analyze()*. Thus, if a tolerance of 0.5 mm is passed, each guard rail is 0.5 mm to the left and right of the invisible picket. Ideally, MLC positions will all be within these guard rails, i.e. within tolerance, and will be colored blue. If they are outside the tolerance they are turned red with a larger box overlaid for easy identification. If an "action tolerance" is also passed to *analyze()*, MLC positions that are below tolerance but above the action tolerance are turned magenta.

Additionally, pylinac provides a semi-transparent colored overlay so that an "all clear" or a "pair(s) failed" status is easily seen and not inadvertently overlooked. If any MLC position is outside the action tolerance or the absolute tolerance, the MLC pair/leaf area is colored the corresponding color. In this way, not every position needs be looked at.

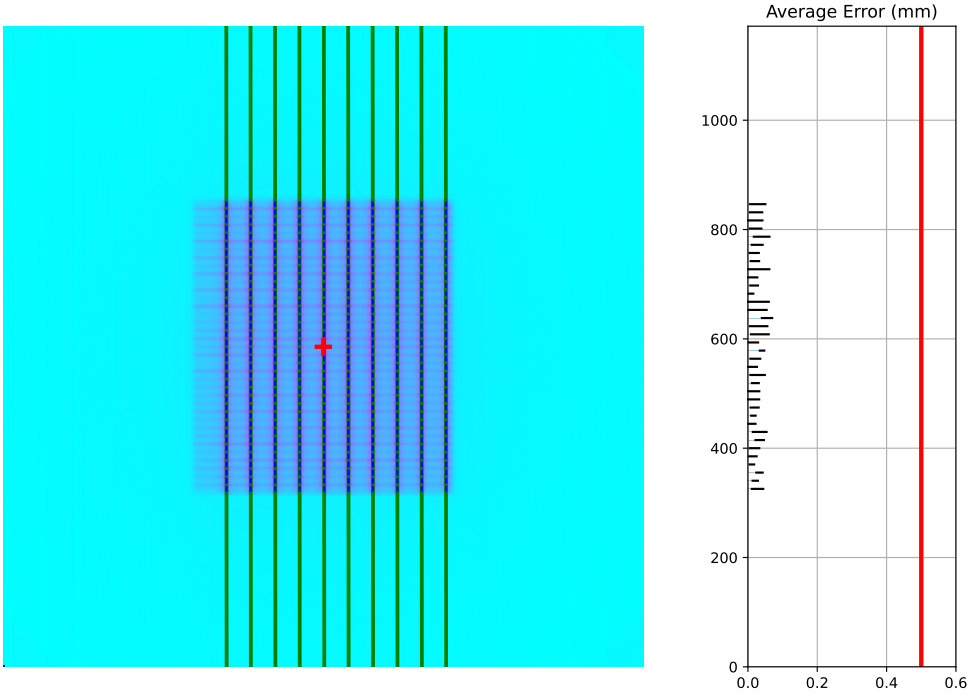
6.12.3 Running the Demo

To run the picketfence demo, create a script or start in interpreter and input:

```
from pylinac import PicketFence

PicketFence.run_demo()
```

Results will be printed to the console and a figure showing the analyzed picket fence image will pop up:

```
Picket Fence Results:
100.0% Passed
Median Error: 0.062mm
Max Error: 0.208mm on Picket: 3, Leaf: 22
```



Finally, you can save the results to a PDF report:

```
pf = PicketFence.from_demo()
pf.analyze()
pf.publish_pdf(filename="PF Oct-2018.pdf")
```

6.12.4 Acquiring the Image

The easiest way to acquire a picket fence image is using the EPID. In fact, pylinac will only analyze images acquired via an EPID, as the DICOM image it produces carries important information about the SID, pixel/mm conversion, etc. Depending on the EPID type and physicist, either the entire array of MLCs can be imaged at once, or only the middle leaves are acquired. Changing the SID can also change how many leaves are imaged. For analysis by pylinac, the SID does not matter, nor EPID type, nor panel translation.

6.12.5 Typical Use

Picket Fence tests are recommended to be done weekly. With automatic software analysis, this can be a trivial task. Once the test is delivered to the EPID, retrieve the DICOM image and save it to a known location. Then import the class:

```
from pylinac import PicketFence
```

The minimum needed to get going is to:

- **Load the image** – As with most other pylinac modules, loading images can be done by passing the image string directly, or by using a UI dialog box to retrieve the image manually. The code might look like either of the following:

  ```
  pf_img = r"C:/QA Folder/June/PF_6_21.dcm"
  pf = PicketFence(pf_img)
  ```

  You may also load multiple images that become superimposed (e.g. an MLC & Jaw irradiation):

  ```
  img1 = r"path/to/image1.dcm"
  img2 = r"path/to/image2.dcm"
  pf = PicketFence.from_multiple_images([img1, img2])
  ```

  As well, you can use the demo image provided:

  ```
  pf = PicketFence.from_demo_image()
  ```

  You can also change the MLC type:

  ```
  pf = PicketFence(pf_img, mlc="HD")
  ```

  In this case, we've set the MLCs to be HD Millennium. For more options and to customize the MLC configuration, see *Customizing MLCs*.

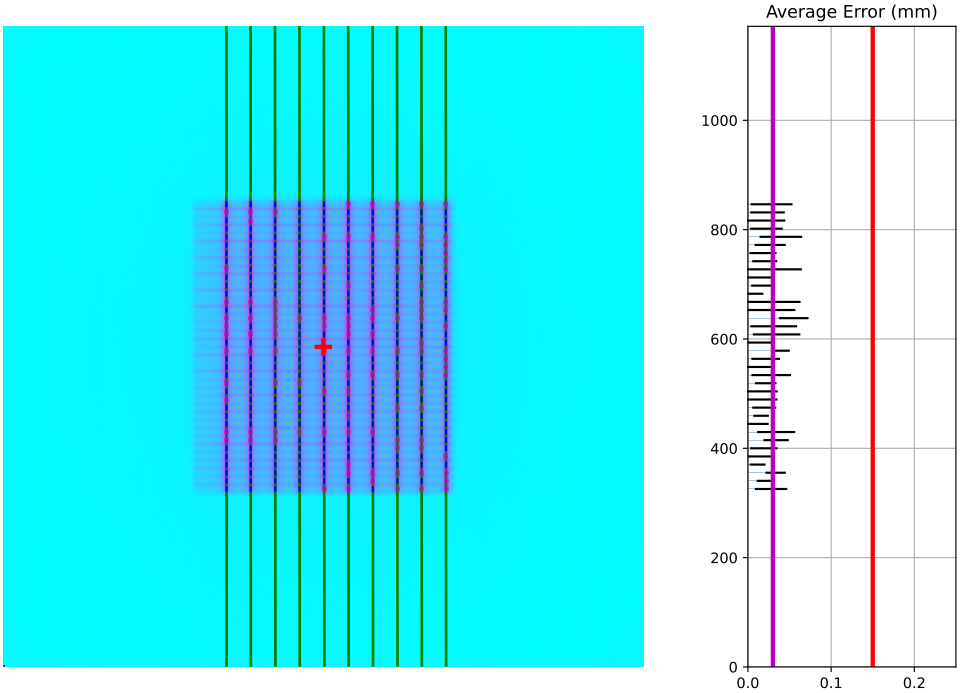
- **Analyze the image** – Once the image is loaded, tell PicketFence to start analyzing the image. See the Algorithm section for details on how this is done. While defaults exist, you may pass in a tolerance as well as an "action" tolerance (meaning that while passing, action should be required above this tolerance):

  ```
  pf.analyze(
      tolerance=0.15, action_tolerance=0.03
  )  # tight tolerance to demo fail & warning overlay
  ```

- **View the results** – The PicketFence class can print out the summary of results to the console as well as draw a matplotlib image to show the image, MLC peaks, guard rails, and a color overlay for quick assessment:

```
# print results to the console
print(pf.results())
# view analyzed image
pf.plot_analyzed_image()
```

which results in:



The plot is also able to be saved to PNG:

```
pf.save_analyzed_image("mypf.png")
```

Or you may save to PDF:

```
pf.publish_pdf("mypf.pdf")
```

6.12.6 Analyzing individual leaves

Historically, MLC pairs were evaluated together; i.e. the center of the picket was determined and compared to the idealized picket. In v3.0+, an option to analyze each leaf of the MLC kiss was added. This will create 2 pickets per gap, one on either side and compare the measurements of each leaf. For backwards compatibility, this option is opt-in. This option also requires a nominal gap value to be passed. To analyze individual leaves:

```
from pylinac import PicketFence

pf = PicketFence(...)
```

(continues on next page)

(continued from previous page)

```
pf.analyze(..., separate_leaves=True, nominal_gap_mm=2)
...
```

Note: Don't forget that you will always need to pass a correct `nominal_gap_mm` value when analyzing separated leaves. A good starting point is the nominal gap (e.g. 2mm in the DICOM plan) + DLG.

The gap value is the combined values of the planned gap, MLC DLG, and EPID scatter effects. This is required since the expected position is no longer at the center of the MLC kiss, but offset to the side and depends on the above effects. You will likely have to determine this for yourself given the different MLCs and EPID combinations make a dynamic computation difficult.

6.12.7 Individual leaf detection vs combined

Despite the above, I personally (JK) don't like the individual leaf analysis approach. I have found the combined method more robust (in terms of analysis). The biggest problem with individual leaf analysis is that the expected leaf width is not just simply the DICOM separation and must be empirically determined. I will describe some of the issues the PF test is meant to or can solve w/r/t individual analysis vs combined.

- **One leaf error**: When one single leaf has an error. This is the quintessential example for PF.

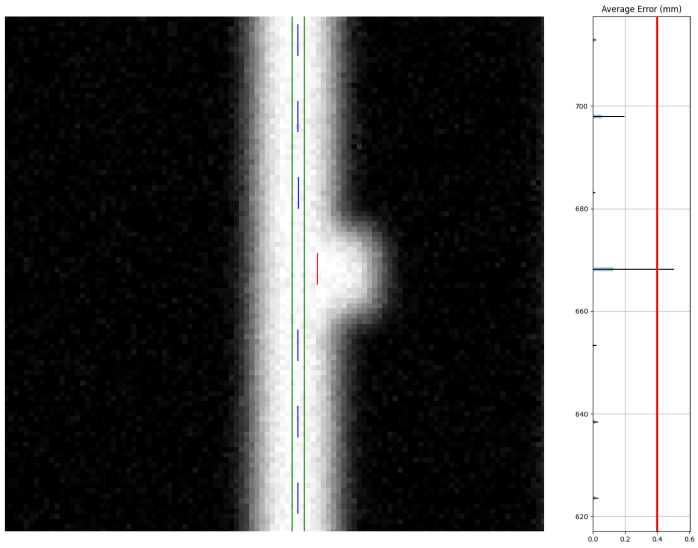


Fig. 1: Combined analysis

Assuming the opposite leaf has no error (see other issues below), the error of a combined analysis is half of the error of the leaf. Over against the argument that it is important to test each leaf, the simple answer is that using a tolerance of half the acceptable error will catch this. I.e. a tolerance of 0.1mm will catch an erroneous leaf up of 0.2mm or more.

- **Both leaves offset (unilateral)**: When both leaves are offset to one side.

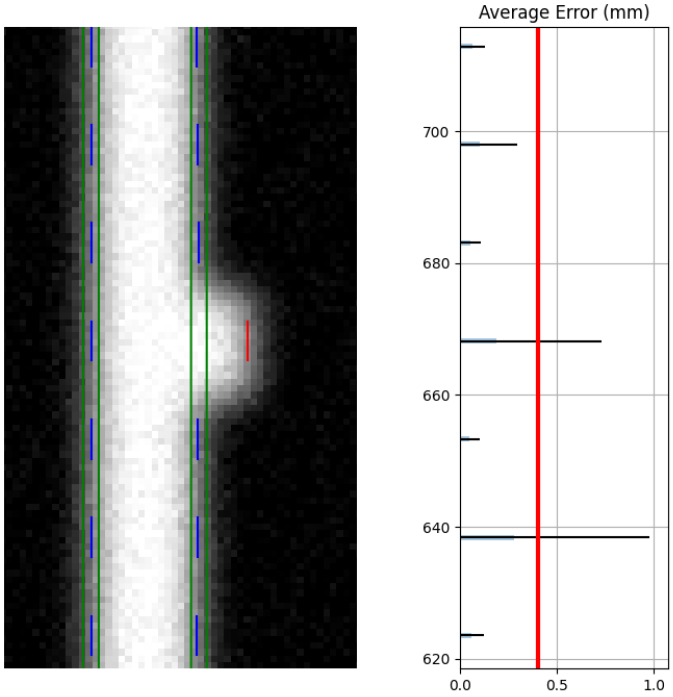


Fig. 2: Separate analysis

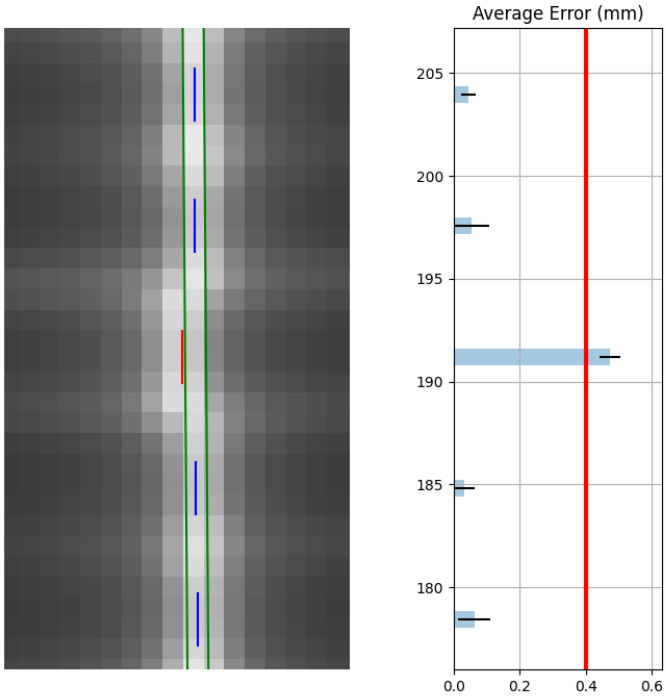


Fig. 3: Combined analysis

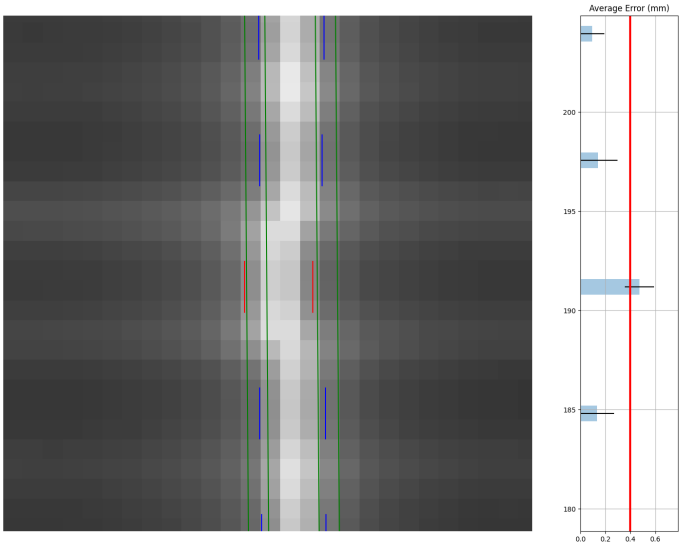


Fig. 4: Separate analysis with the same tolerance

As the images show, both analyses detect the problem. This makes sense given that the error was the same direction for both leaves.

- **Both leaves offset (mirrored)**: When both leaves have an offset error, but in opposite directions. This is the only drawback to the combined method.

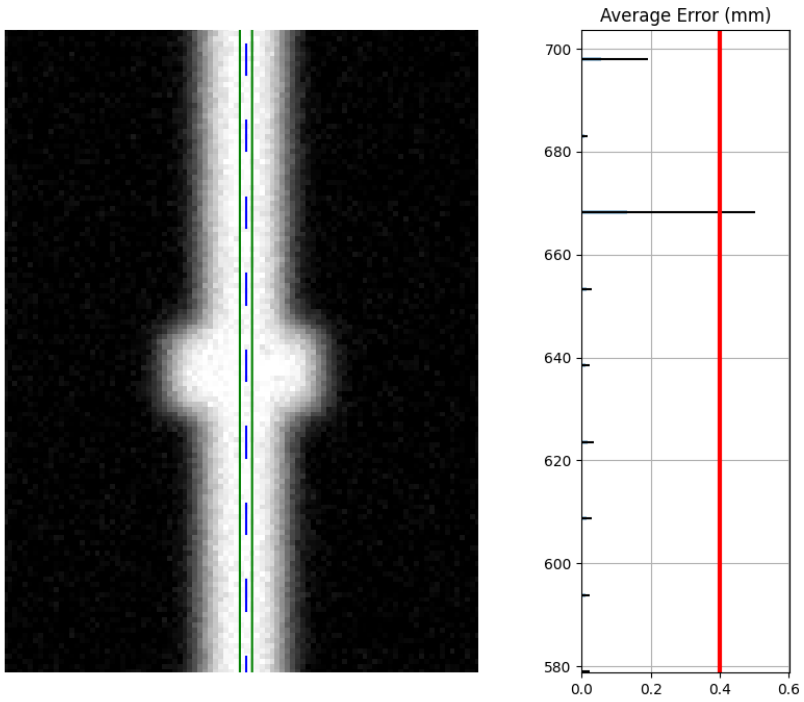


Fig. 5: Combined analysis

Clearly, the separate analysis is advantageous here in terms of detecting the error. The chance of MLC leaves being off by the same amount in opposite directions seems extraordinarily rare. The more likely error would be that the picket width for all leaves is too wide or too narrow. Such a scenario would be easily caught with a DLG test.

To be clear, I'm not against individual leaf analysis, but my anecdotal experience leans toward combined analysis being more robust. Combined with other QA typically performed, I don't think the medical physics community is all out of whack because they use the combined method vs individual analysis. Use what works for you but realize the strengths of each. Finally, remember that physician contours vary a lot, sometimes by a factor or more. This dwarfs any 0.1mm error of the leaf that we might squabble about. For the scenarios you actually need that 0.1mm, such as SRS, the patient plan QA is the most important factor in determining whether a problem exists.

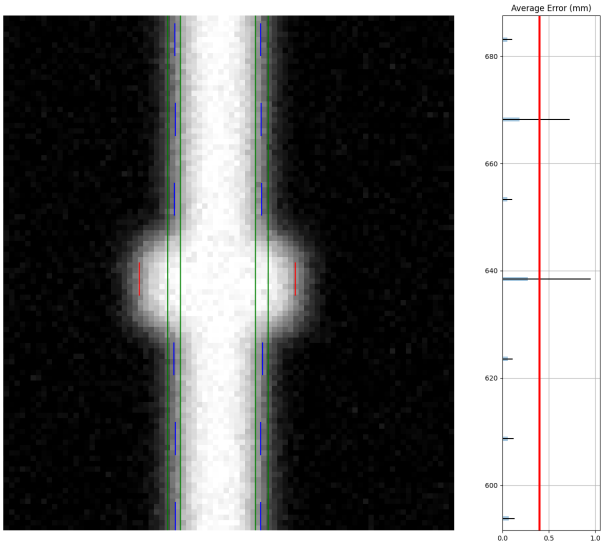


Fig. 6: Separate analysis

6.12.8 Plotting a histogram

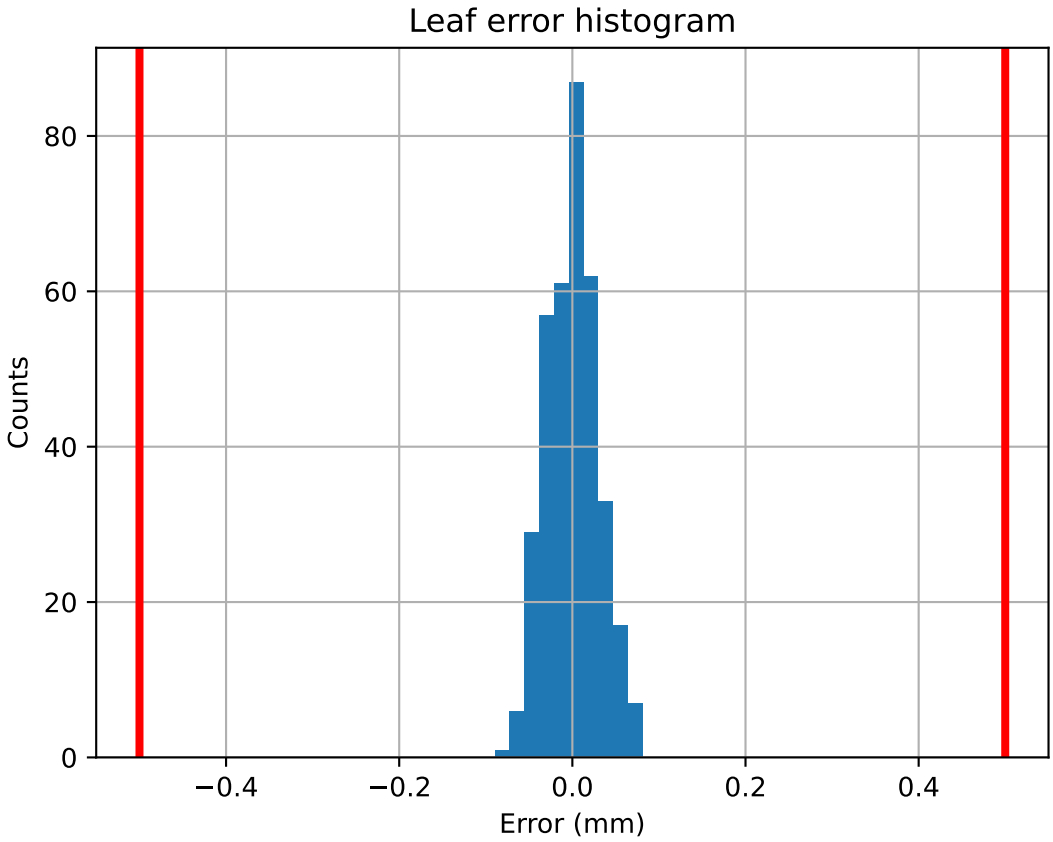
As of v3.0, you may plot a histogram of the error data like so:

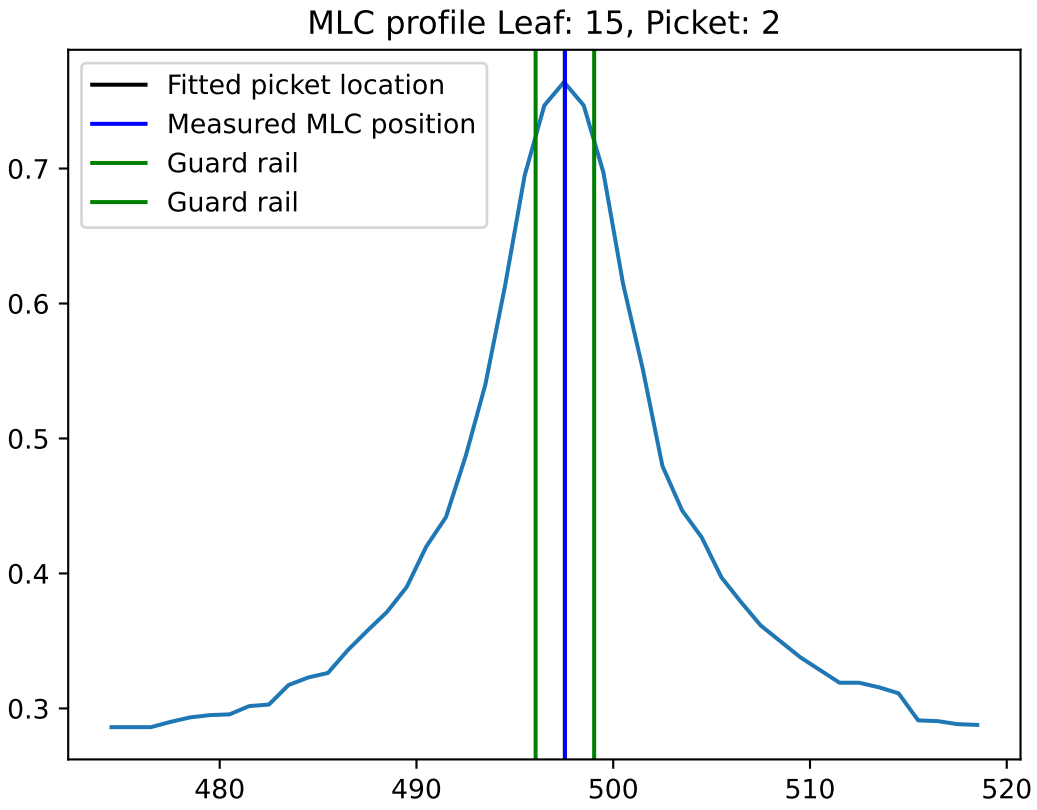
```
from pylinac import PicketFence
pf = PicketFence.from_demo_image()
pf.analyze()
pf.plot_histogram()
```

6.12.9 Plotting a leaf profile

As of v3.0, you may plot an individual leaf profile like so:

```
from pylinac import PicketFence
pf = PicketFence.from_demo_image()
pf.analyze()
pf.plot_leaf_profile(leaf=15, picket=2)
```


Leaf error histogram
Counts
0
20
40
60
80
−0.4
−0.2
0.0
0.2
0.4
Error (mm)


MLC profile Leaf: 15, Picket: 2
Fitted picket location
Measured MLC position
Guard rail
Guard rail
0.7
0.6
0.5
0.4
0.3
480
490
500
510
520

6.12.10 Using a Machine Log

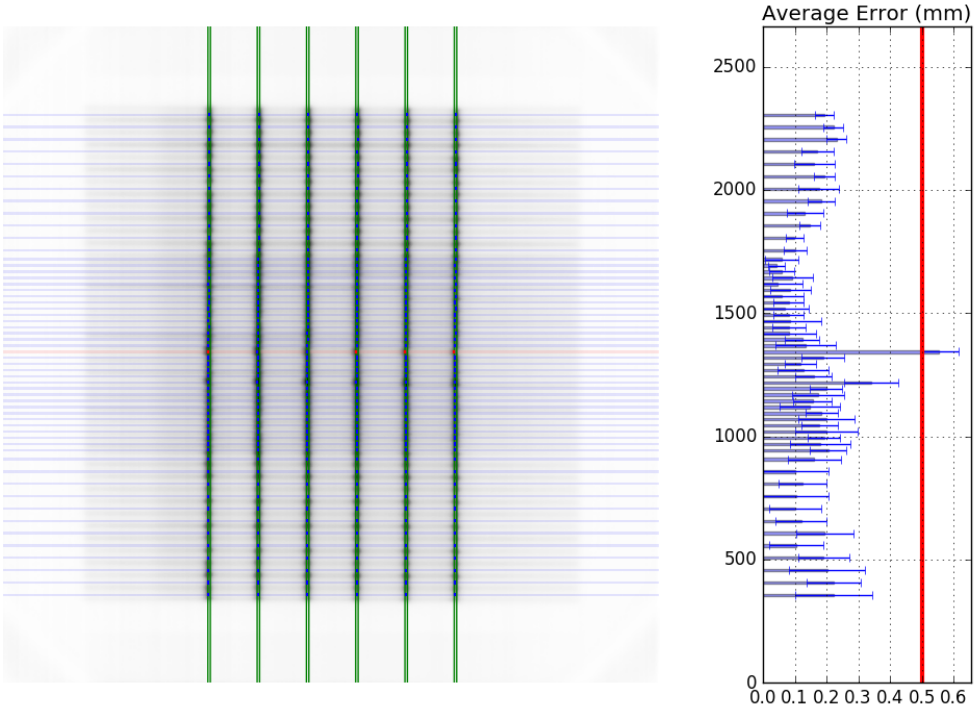
As of v1.4, you can load a machine log along with your picket fence image. The algorithm will use the expected fluence of the log to determine where the pickets should be instead of fitting to the MLC peaks. Usage looks like this:

```
from pylinac import PicketFence

pf = PicketFence("my/pf.dcm", log="my/pf_log.bin")
...
```

Everything else is the same except the measurements are **absolute**.

> **Warning:** While using a machine log makes the MLC peak error absolute, there may be EPID twist or sag that will exaggerate differences that may or may not be real. Be sure to understand how your imager moves during your picket fence delivery. Even TrueBeams are not immune to EPID twist.

Results will look similar. Here's an example of the results of using a log:



6.12.11 Customizing MLCs

As of v2.5, MLC configuration is set a priori (vs empirical determination as before) and the user can also create custom MLC types. Pylinac was only able to handle Millennium and HD Millennium previously.

Preset configurations

Use a specific preset config:

```
from pylinac.picketfence import PicketFence, MLC

pf = PicketFence(pf_img, mlc=MLC.MILLENNIUM)
```

The built-in presets can be seen in attrs of the *MLC* class.

Creating and using a custom configuration

Using a custom configuration is very easy. You must create and then pass in a custom *MLCArrangement*. Leaf arrangements are sets of tuples with the leaf number and leaf width. An example will make this clear:

```
from pylinac.picketfence import PicketFence, MLCArrangement

# recreate a standard Millennium MLC with 10 leaves of 10mm width, then 40 leaves of 5mm,
↪ then 10 of 10mm again.
mlc_setup = MLCArrangement(leaf_arrangement=[(10, 10), (40, 5), (10, 10)])
# add an offset for Halcyon-style or odd-numbered leaf setups
mlc_setup_offset = MLCArrangement(leaf_arrangement=..., offset=2.5)  # offset is in mm

# pass it in to the mlc parameter
pf = PicketFence("path/to/img", mlc=mlc_setup)

# proceed as normal
pf.analyze(...)
...
```

6.12.12 Acquiring good images

The following are general tips on getting good images that pylinac will analyze easily. These are in addition to the algorithm allowances and restrictions:

- Keep your pickets away from the edges. That is, in the direction parallel to leaf motion keep the pickets at least 1-2cm from the edge.
- If you use wide-gap pickets, give a reasonable amount of space between the pickets and keep the gap wider than the picket. I.e. don't have 5mm spacing between 20mm pickets.
- If you use Y-jaws, leave them open 1-2 leaves more than the leaves you want to measure. For example. if you just want to analyze the "central" leaves and set Y-jaws to +/-10cm, the leaves at the edge may not be caught by the algorithm (although see the `edge_threshold` parameter of `analyze`). To avoid having to tweak the algorithm, just open the jaws a bit more.
- Don't put anything else in the beam path. This might sound obvious, but I'm continually surprised at the types of images people try to use/take. No, pylinac cannot account for the MV phantom you left on the couch when you took your PF image.

- Keep the leaves parallel to an edge. I.e. as close to 0, 90, 270 as possible.

6.12.13 Tips & Tricks

Use `results_data`

Using the picketfence module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *analyze()* method which returns a *PFResult* instance.

Note: While the pylinac tooling may change under the hood, this object should remain largely the same and/or expand. Thus, using this is more stable than accessing attrs directly.

Continuing from above:

```
data = pf.results_data()
data.max_error_mm
data.tolerance_mm
# and more

# return as a dict
data_dict = pf.results_data(as_dict=True)
data_dict["max_error_mm"]
...
```

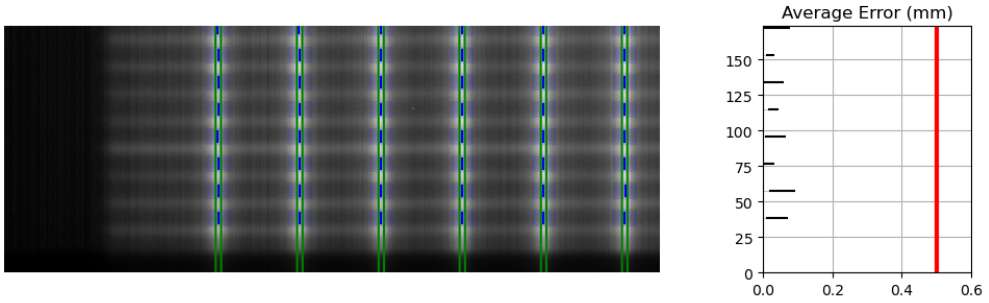
EPID sag

For older linacs, the EPID can also sag at certain angles. Because pylinac assumes a perfect panel, sometimes the analysis will not be centered exactly on the MLC leaves. If you want to correct for this, simply pass the EPID sag in mm:

```
pf = PicketFence(r"C:/path/saggyPF.dcm")
pf.analyze(sag_adjustment=0.6)
```

Edge leaves

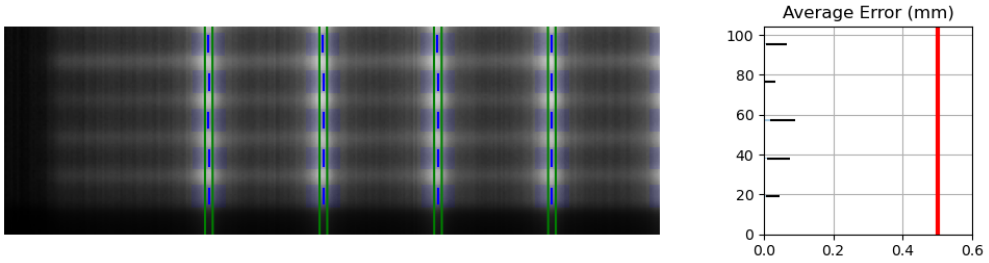
For some images, the leaves at the edge of the image or adjacent to the jaws may not be detected. See the image below:



This is caused by the algorithm filtering and can be changed through an analysis parameter. Increase the number to catch more edge leaves:

```
pf = PicketFence(...)
pf.analyze(..., edge_threshold=3)
...
```

This results with the edge leaves now being caught in this case. You may need to experiment with this number a few times:



6.12.14 Benchmarking the algorithm

With the image generator module we can create test images to test the picket fence algorithm on known results. This is useful to isolate what is or isn't working if the algorithm doesn't work on a given image and when commissioning pylinac.

Note: Some results here are not perfect. This is because the image generator module cannot necessarily generate pickets of exactly a given gap. The pickets are simulated by setting the pixel values. A gap is rounded to the closest pixel equivalent of the desired gap size; this may not be perfectly symmetric. This affects the error when doing separate leaf analysis and also when evaluating the distance from the CAX. Further, many of these have small amounts of random noise applied on purpose.

Perfect Up-Down Image

Below, we generate a DICOM image with slits representing pickets. Several realistic side-effects are not here (such as tongue and groove), but this is perfect for testing. Think of this as the equivalent of measuring a 10x10cm field on the linac vs TPS dose before moving on to VMAT plans.

The script will generate the file, but you can also download it here: `perfect_up_down.dcm`.

```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

# the file name to write the DICOM image to disk to
pf_file = "perfect_.dcm"
# create a PF image with 5 pickets with 40mm spacing between them and 3mm gap. Also␣
↪applies a gaussian filter to simulate the leaf edges.
generate_picketfence(
    simulator=AS1200Image(sid=1000),
    field_layer=PerfectFieldLayer,
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
    ],
    pickets=5,
    picket_spacing_mm=40,
    picket_width_mm=3,
    orientation=Orientation.UP_DOWN,
)
# load it just like any other
pf = pylinac.PicketFence(pf_file)
pf.analyze(separate_leaves=False, nominal_gap_mm=4)
print(pf.results_data())
pf.plot_analyzed_image()
```

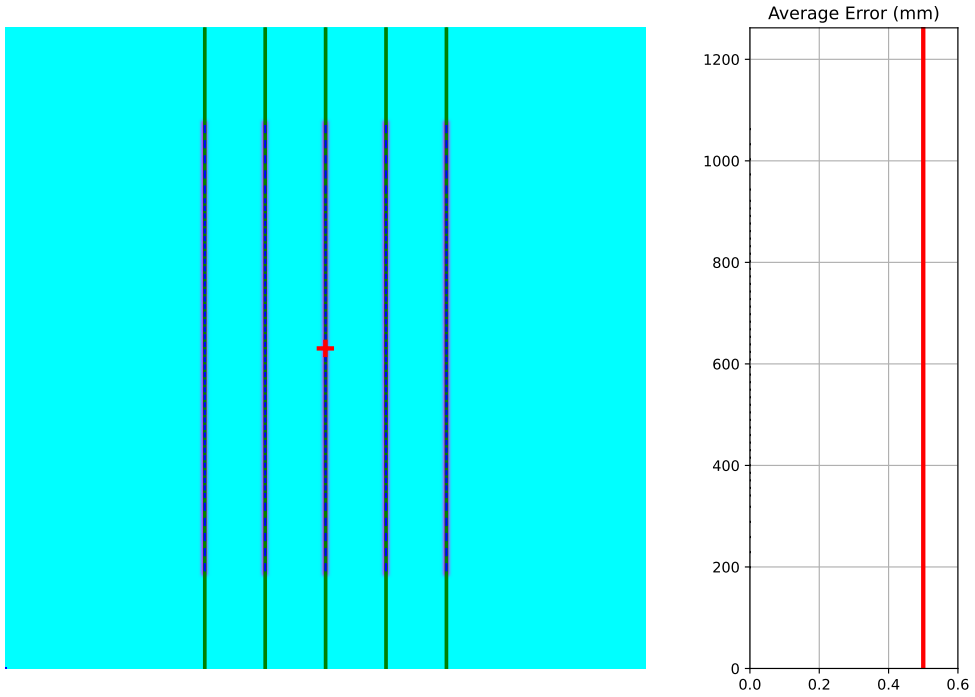
As you can see, the error is zero, the pickets are perfectly straight up and down, and everything looks good.

Perfect Left-Right

Generated file: `perfect_left_right.dcm`.

```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "perfect_left_right.dcm"
generate_picketfence(
    simulator=AS1200Image(sid=1000),
    field_layer=PerfectFieldLayer,
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
```

(continues on next page)



(continued from previous page)

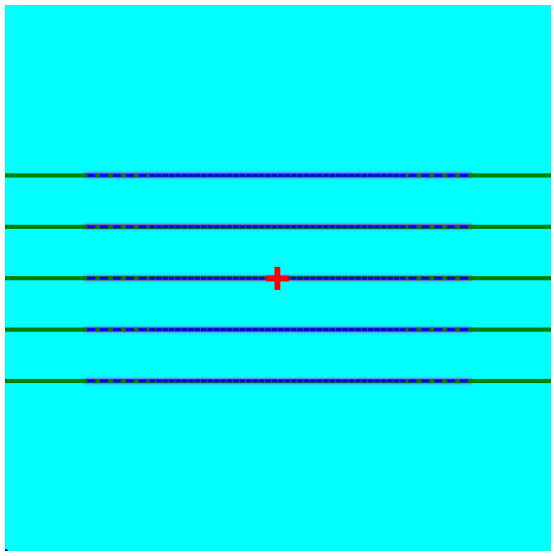
```
    ],
    pickets=5,
    picket_spacing_mm=40,
    picket_width_mm=3,
    orientation=Orientation.LEFT_RIGHT,
)

pf = pylinac.PicketFence(pf_file)
pf.analyze(separate_leaves=False, nominal_gap_mm=4)
print(pf.results_data())
pf.plot_analyzed_image()
```

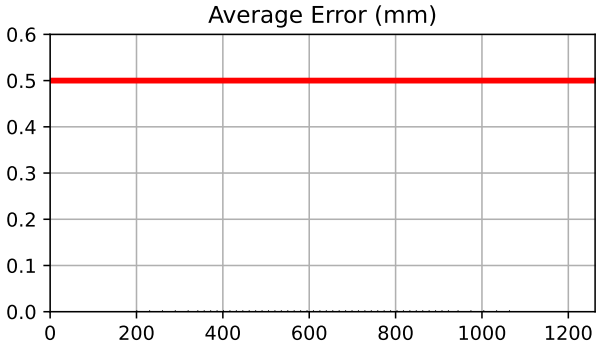
Noisy, Wide-gap Image

Generated file: `noisy_wide_gap_up_down.dcm`.

```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "noisy_wide_gap_up_down.dcm"
generate_picketfence(
```

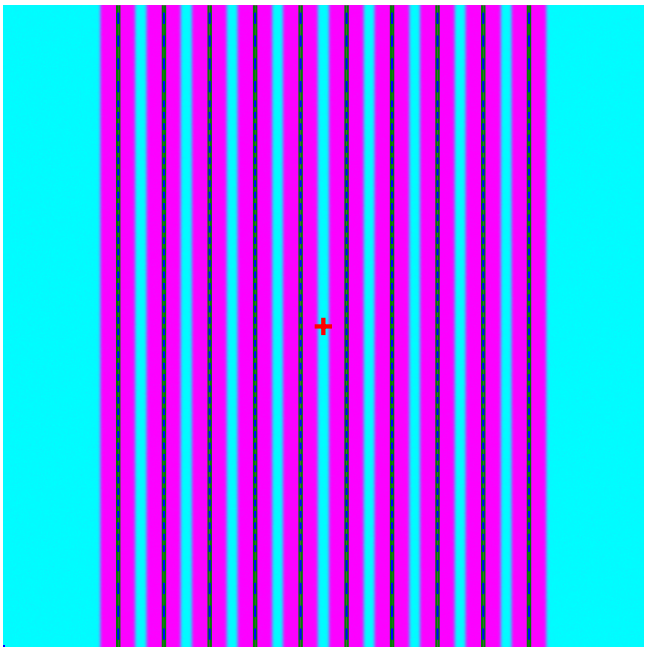
(continues on next page)

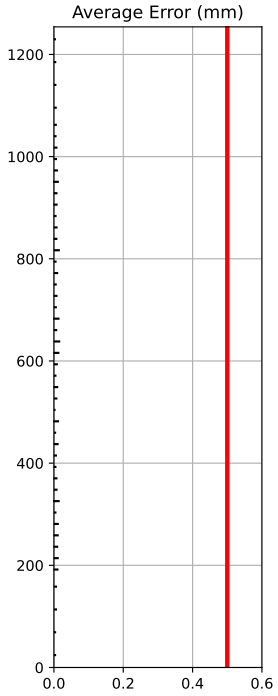



Average Error (mm)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
200
400
600
800
1000
1200

(continued from previous page)

```
    simulator=AS1200Image(sid=1500),
    field_layer=PerfectFieldLayer,  # this applies a non-uniform intensity about the CAX,
↪ simulating the horn effect
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
        RandomNoiseLayer(sigma=0.03)  # add salt & pepper noise
    ],
    pickets=10,
    picket_spacing_mm=20,
    picket_width_mm=10,  # wide-ish gap
    orientation=Orientation.UP_DOWN,
)

pf = pylinac.PicketFence(pf_file)
pf.analyze()
print(pf.results_data())
pf.plot_analyzed_image()
```





Individual Leaf Analysis

Let's now analyze individual leaves using the `separate_leaves` parameter. This uses the same image base as above; note that the analysis is different.

Generated file: `separated_wide_gap_up_down.dcm`.

```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "separated_wide_gap_up_down.dcm"
generate_picketfence(
    simulator=AS1200Image(sid=1500),
    field_layer=PerfectFieldLayer,  # this applies a non-uniform intensity about the CAX,
↪ simulating the horn effect
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
        RandomNoiseLayer(sigma=0.03)  # add salt & pepper noise
    ],
    pickets=10,
    picket_spacing_mm=20,
    picket_width_mm=10,  # wide-ish gap
    orientation=Orientation.UP_DOWN,
)

pf = pylinac.PicketFence(pf_file)
pf.analyze(separate_leaves=True, nominal_gap_mm=10)
print(pf.results())
print(pf.results_data())
pf.plot_analyzed_image()
```

Note that this image has an error of ~0.1mm. This is due to the rounding of pixel values when generating the picket. I.e. it's not always possible to generate an exactly 10mm gap, but instead is rounded to the nearest pixel equivalent of 10mm.

Rotated

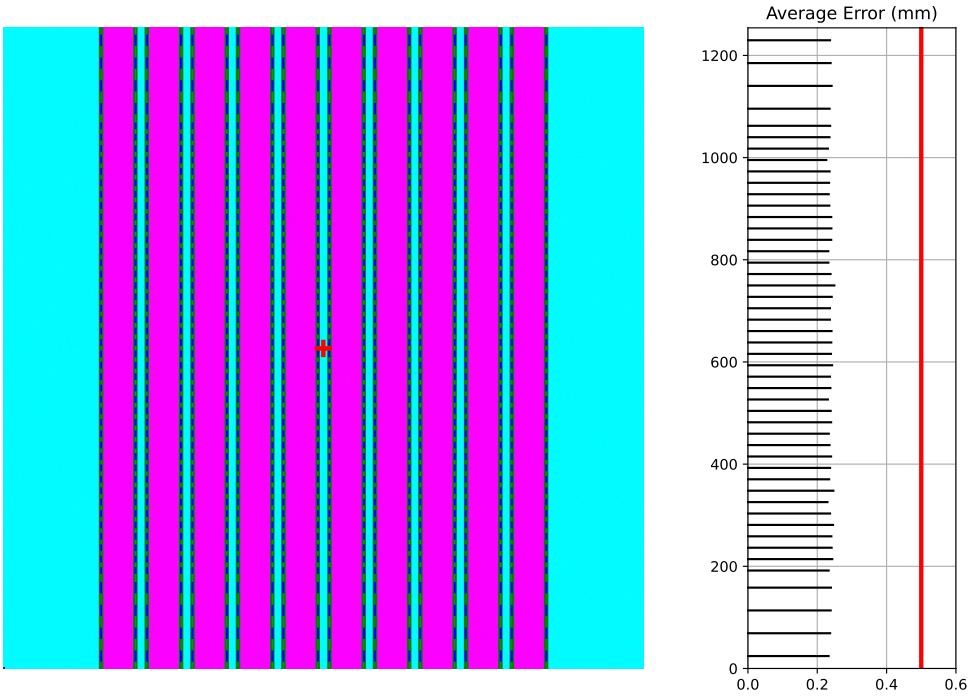
Let's analyze a slightly rotated image of 2 degrees. Recall that pylinac is limited to ~5 degrees of rotation (depending on picket size).

The image generator doesn't do the rotation, but is applied later after loading.

Generated file: `rotated_up_down.dcm`.

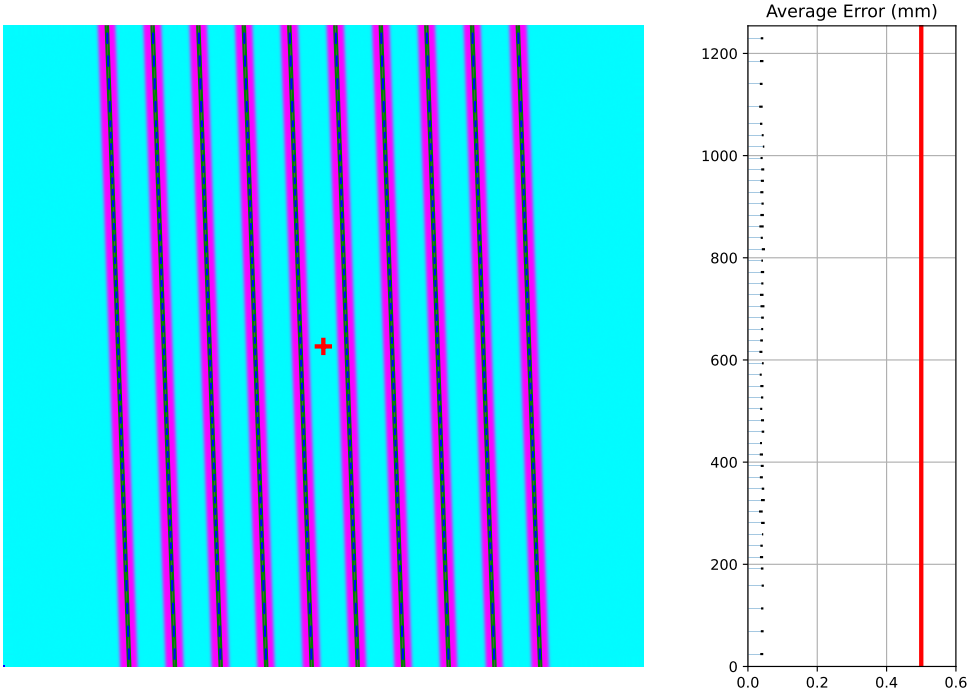
```
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation
```

(continues on next page)



(continued from previous page)

```
pf_file = "rotated_up_down.dcm"
generate_picketfence(
    simulator=AS1200Image(sid=1500),
    field_layer=PerfectFieldLayer,  # this applies a non-uniform intensity about the CAX,
↪ simulating the horn effect
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
        RandomNoiseLayer(sigma=0.01)  # add salt & pepper noise
    ],
    pickets=10,
    picket_spacing_mm=20,
    picket_width_mm=5,
    orientation=Orientation.UP_DOWN,
)

pf = pylinac.PicketFence(pf_file)
# here's where we rotate
pf.image.array = ndimage.rotate(pf.image, -2, reshape=False, mode='nearest')
pf.analyze(separate_leaves=False, nominal_gap_mm=5)
print(pf.results())
print(pf.results_data())
pf.plot_analyzed_image()
```



Offset pickets

In this example, we offset the pickets to simulate an error where the picket was delivered at the wrong x-distance. Lots of physicists cite this as a possibility (or expect their QA software to catch it) but I've never seen it. If you have let me know!

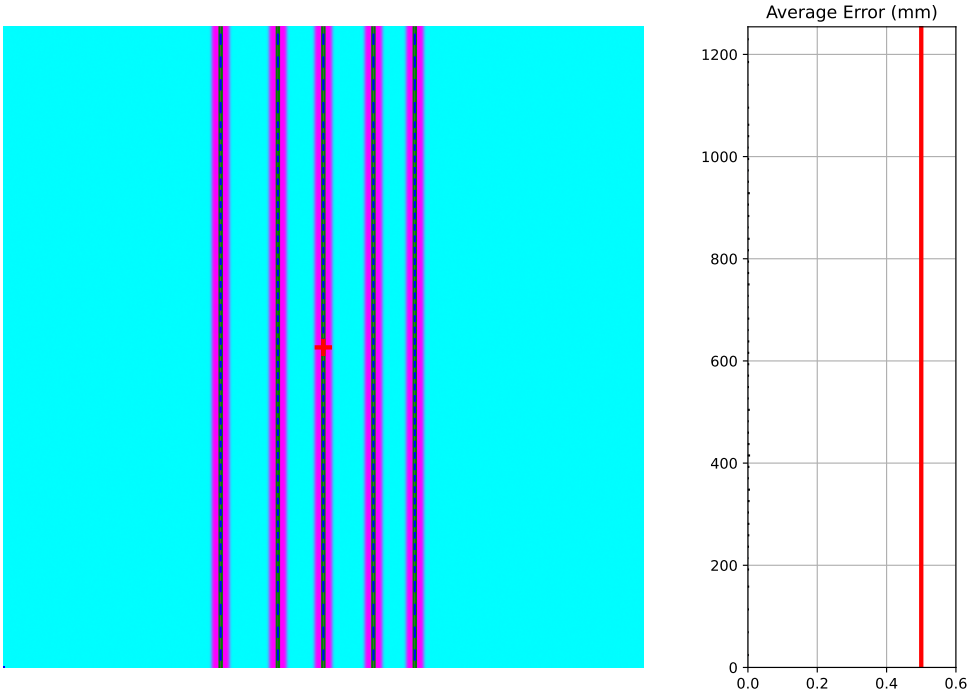
Generated file: `offset_picket.dcm`.

```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "offsetpicket.dcm"
generate_picketfence(
    simulator=AS1200Image(sid=1500),
    field_layer=PerfectFieldLayer,  # this applies a non-uniform intensity about the CAX,
↪ simulating the horn effect
    file_out=pf_file,
    final_layers=[
        GaussianFilterLayer(sigma_mm=1),
        RandomNoiseLayer(sigma=0.01)  # add salt & pepper noise
    ],
    pickets=5,
    picket_spacing_mm=20,
    picket_width_mm=5,
```

(continues on next page)

(continued from previous page)

```
    picket_offset_error=[-5, 0, 0, 2, 0],  # array of errors; length must match the␣
↪number of pickets
    orientation=Orientation.UP_DOWN,
)

pf = pylinac.PicketFence(pf_file)
pf.analyze()
print(pf.results())
print(pf.results_data())
pf.plot_analyzed_image()
```



Which produces the following output:

```
...
Picket offsets from CAX (mm): 45.0 19.9 0.0 -22.0 -40.1
...
```

The results still show passing. However, note the printed picket offsets from the CAX. The first picket is off by 5mm and the 4th is off by 2mm (as we introduced).

Erroneous leaves

In this example we introduce errors simulating leaves opening farther than they should.

Generated file: `erroneous_leaves.dcm`.

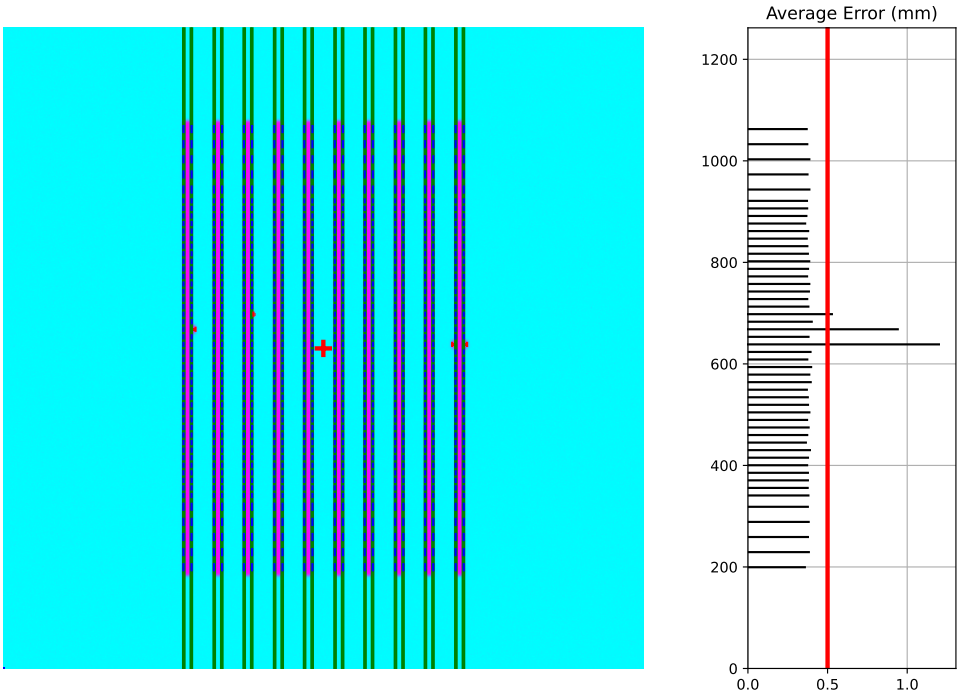
```
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,␣
↪PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "erroneous_leaves.dcm"
generate_picketfence(
        simulator=AS1200Image(sid=1000),
        field_layer=PerfectFieldLayer,  # this applies a non-uniform intensity about the␣
↪CAX, simulating the horn effect
        file_out=pf_file,
        final_layers=[
            PerfectFieldLayer(field_size_mm=(5, 10), cax_offset_mm=(2.5, 90)),  # a 10mm␣
↪gap centered over the picket
            PerfectFieldLayer(field_size_mm=(5, 5), cax_offset_mm=(12.5, -87.5)),  # a 2.
↪5mm extra opening of one leaf
            PerfectFieldLayer(field_size_mm=(5, 5), cax_offset_mm=(22.5, -49)),  # a 1mm␣
↪extra opening of one leaf
            GaussianFilterLayer(sigma_mm=1),
            RandomNoiseLayer(sigma=0.03)  # add salt & pepper noise
        ],
        pickets=10,
        picket_spacing_mm=20,
        picket_width_mm=5,  # wide-ish gap
        orientation=Orientation.UP_DOWN,
)

pf = pylinac.PicketFence(pf_file)
pf.analyze(separate_leaves=True, nominal_gap_mm=5)
print(pf.results())
print(pf.results_data())
pf.plot_analyzed_image()
```

6.12.15 Algorithm

The picket fence algorithm uses expected lateral positions of the MLCs and samples those regions for the center of the FWHM to determine the MLC positions:

Allowances

- The image can be any size.
- Various leaf sizes can be analyzed (e.g. 5 and 10mm leaves for standard Millennium).
- Any MLC can be analyzed. See *Customizing MLCs*
- The image can be either orientation (pickets going up-down or left-right).
- The image can be at any SSD.
- Any EPID type can be used (aS500, aS1000, aS1200).



- The EPID panel can have an x or y offset (i.e. translation).

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The image must be a DICOM image acquired via the EPID.
- The delivery must be parallel or nearly-parallel (<~5°) to an image edge; i.e. the collimator should be at 0, 90, or 270 degrees.

Pre-Analysis

- **Check for noise** – Dead pixels can cause wild values in an otherwise well-behaved image. These values can disrupt analysis, but pylinac will try to detect the presence of noise and will apply a median filter if detected.
- **Check image inversion** – Upon loading, the image is sampled near all 4 corners for pixel values. If it is greater than the mean pixel value of the entire image the image is inverted.
- **Determine orientation** – The image is summed along each axis. Pixel percentile values of each axis sum are sampled. The axis with a greater difference in percentile values is chosen as the orientation (The picket axis, it is argued, will have more pixel value variation than the axis parallel to leaf motion.)
- **Adjust for EPID sag** – If a nonzero value is passed for the sag adjustment, the image is shifted along the axis of the pickets; i.e. a +1 mm adjustment for an Up-Down picket image will move expected MLC positions up 1 mm.

Analysis

- **Find the pickets** – The mean profile of the image perpendicular to the MLC travel direction is taken. Major peaks are assumed to be pickets.

- **Find FWHM at each MLC position** – For each picket, a sample of the image in the MLC travel direction is taken at each MLC position. The center of the FWHM of the picket for that MLC position is recorded.
- **Fit the picket to the positions & calculate error** – Once all the MLC positions are determined, the positions from each peak of a picket are fitted to a 1D polynomial which is considered the ideal picket. Differences of each MLC position to the picket polynomial fit at that position are determined, which is the error. When plotted, errors are tested against the tolerance and action tolerance as appropriate.

6.12.16 Troubleshooting

First, check the general *Troubleshooting* section. Specific to the picket fence analysis, there are a few things you can do.

- **Set the image inversion** - If you get an error like this: `ValueError: max() arg is an empty sequence`, one issue may be that the image has the wrong inversion (negative values are positive, etc). Set the analyze flag `invert` to `True` to invert the image from the automatic detection. Additionally, if you're using wide pickets, the image inversion could be wrong. If the pickets are wider than the "valleys" between the pickets this will almost always result in a wrong inversion.
- **Crop the edges** - This is far and away the most common problem. Elekta is notorious for having noisy/bad edges. Pass a larger value into the constructor:

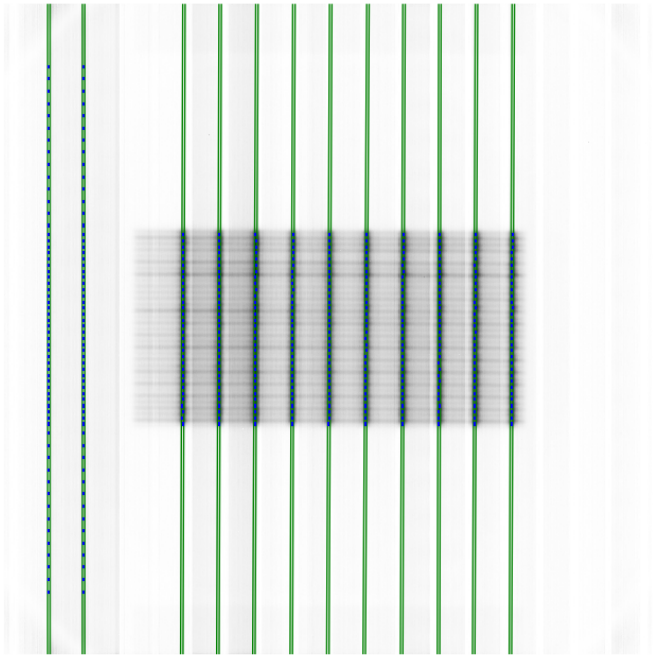
```
pf = PicketFence(..., crop_mm=7)
```

- **Apply a filter upon load** - While pylinac tries to correct for unreasonable noise in the image before analysis, there may still be noise that causes analysis to fail. A way to check this is by applying a median filter upon loading the image:

```
pf = PicketFence("mypf.dcm", filter=5)  # vary the filter size depending on the␣
↪image
```

  Then try performing the analysis.

- **Check for streak artifacts** - It is possible in certain scenarios (e.g. TrueBeam dosimetry mode) to have noteworthy artifacts in the image like so:



If the artifacts are in the same direction as the pickets then it is possible pylinac is tripping on these artifacts. You can reacquire the image in another mode or simply try again in the same mode. You may also try cropping the image to exclude the artifact:

```
pf = PicketFence("mypf.dcm")
pf.image.array = mypf.image.array[200:400, 150:450]  # or whatever values you want
```

- **Set the number of pickets** - If pylinac is catching too many pickets you can set the number of pickets to find with *analyze()*.
- **Crop the image** - For Elekta images, the 0th column is often an extreme value. For any Elekta image, it is suggested to crop the image. You can crop the image like so:

```
pf = PicketFence(r"my/pf.dcm")
pf.image.crop(pixels=3)
pf.analyze()
...
```

6.12.17 API Documentation

Main classes

These are the classes a typical user may interface with.

class pylinac.picketfence.**PicketFence**(*filename: str | Path | BinaryIO*, *filter: int | None = None*, *log: str | None = None*, *use_filename: bool = False*, *mlc:* MLC | MLCArrangement | *str = MLC.MILLENNIUM*, *crop_mm: int = 3*, *image_kwargs: dict | None = None*)

Bases: `object`

A class used for analyzing EPID images where radiation strips have been formed by the MLCs. The strips are assumed to be parallel to one another and normal to the image edge; i.e. a "left-right" or "up-down" orientation is assumed. Further work could follow up by accounting for any angle.

Parameters

filename
Name of the file as a string or a file-like object.

filter
If None (default), no filtering will be done to the image. If an int, will perform median filtering over image of size `filter`.

log
Path to a log file corresponding to the delivery. The expected fluence of the log file is used to construct the pickets. MLC peaks are then compared to an absolute reference instead of a fitted picket.

use_filename
If False (default), no action will be performed. If True, the filename will be searched for keywords that describe the gantry and/or collimator angle. For example, if set to True and the file name was "PF_gantry45.dcm" the gantry would be interpreted as being at 45 degrees.

mlc
The MLC model of the image. Must be an option from the enum `MLCs` or an *MLCArrangement*.

crop_mm
The number of mm to crop from all edges. Elekta is infamous for having columns of dead pixels on the side of their images. These need to be cleaned up first. For Varian images, this really shouldn't make a difference unless the pickets are very close to the edge. Generally speaking, they shouldn't be for the best accuracy.

classmethod from_url(*url: str*, *filter: int = None*, *image_kwargs: dict | None = None*)

Instantiate from a URL.

classmethod from_demo_image(*filter: int | None = None*)

Construct a PicketFence instance using the demo image.

classmethod from_multiple_images(*path_list: Iterable[str | Path]*, *stretch_each: bool = True*, *method: str = 'mean'*, ***kwargs*)

Load and superimpose multiple images and instantiate a PF object.

Parameters

path_list
[iterable] An iterable of path locations to the files to be loaded/combined.

stretch_each
[bool] Whether to stretch each image individually before combining. See `load_multiples`.

method
[{'sum', 'mean'}] The method to combine the images. See `load_multiples`.

kwargs
Passed to *load_multiples()*.

classmethod from_bb_setup(**args*, *bb_image: str | Path | BinaryIO*, *bb_diameter: float*, ***kwargs*)

Construct a PicketFence instance using a BB setup image to find the CAX first. The CAX of the PF image is then overridden w/ the BB location from the first image.

Thank the French for this.

property passed: bool

Boolean specifying if all MLC positions were within tolerance.

property percent_passing: float

Return the percentage of MLC positions under tolerance.

property max_error: float

Return the maximum error found.

property max_error_picket: int

Return the picket number where the maximum error occurred.

picket_width_stat(*picket: int*, *metric: str = 'max'*) → float

Get the statistic of the picket width for the given picket.

Parameters

picket
The picket number to analyze.

metric
The metric to use. One of 'max', 'median', 'mean', 'min'.

property max_error_leaf: int | str

Return the leaf/leaf pair that had the maximum error. This will be a single int value (i.e. either/both A and B) for classic analysis or a fully-qualified name for separate analysis. E.g. A43

failed_leaves() → list[int] | list[str]

A list of the failed leaves. Either the leaf number or the bank+leaf number if using separate leaves.

property abs_median_error: float

Return the median error found.

property num_pickets: int

Return the number of pickets determined.

property mean_picket_spacing: float

The average distance between pickets in mm.

plot_leaf_profile(*leaf: str | int*, *picket: int*, *show: bool = True*)

Plot the leaf profile of a given leaf pair parallel to leaf motion.

Parameters

leaf
The leaf to plot. If `separate_leaves` is True, this will be a string like "A15" or "B33". If `separate_leaves` is False, this must be an int, like `15` or `33`.

picket
An int of the picket number. Pickets start from the 0-side of an image. E.g. for left-right PFs, this would start on the left; for up-down this would start at the bottom.

save_leaf_profile(*filename: str | Path | BinaryIO*, *leaf: str | int*, *picket: int*, ***kwargs*)

Save the leaf profile plot to disk or stream. See plot_leaf_profile for parameter hints. Kwargs are passed to matplotlib.savefig()

static run_demo(*tolerance: float = 0.5*, *action_tolerance: float | None = None*) → None

Run the Picket Fence demo using the demo image. See analyze() for parameter info.

analyze(*tolerance: float = 0.5*, *action_tolerance: float | None = None*, *num_pickets: int | None = None*, *sag_adjustment: float | int = 0*, *orientation: Orientation | str | None = None*, *invert: bool = False*, *leaf_analysis_width_ratio: float = 0.4*, *picket_spacing: float | None = None*, *height_threshold: float = 0.5*, *edge_threshold: float = 1.5*, *peak_sort: str = 'peak_heights'*, *required_prominence: float = 0.2*, *fwxm: int = 50*, *separate_leaves: bool = False*, *nominal_gap_mm: float = 3*, *central_axis: Point | None = None*) → None

Analyze the picket fence image.

Parameters

tolerance
The tolerance of difference in mm between an MLC pair position and the picket fit line.

action_tolerance
If None (default), no action tolerance is set or compared to. If an int or float, the MLC pair measurement is also compared to this tolerance. Must be lower than tolerance. This value is usually meant to indicate that a physicist should take an "action" to reduce the error, but should not stop treatment.

num_pickets
The number of pickets in the image. A helper parameter to limit the total number of pickets, only needed if analysis is catching more pickets than there really are.

sag_adjustment
The amount of shift in mm to apply to the image to correct for EPID sag. For Up-Down picket images, positive moves the image down, negative up. For Left-Right picket images, positive moves the image left, negative right.

orientation
If None (default), the orientation is automatically determined. If for some reason the determined orientation is not correct, you can pass it directly using this parameter. If passed a string with 'u' (e.g. 'up-down', 'u-d', 'up') it will set the orientation of the pickets as going up-down. If passed a string with 'l' (e.g. 'left-right', 'lr', 'left') it will set it as going left-right.

invert
If False (default), the inversion of the image is automatically detected and used. If True, the image inversion is reversed from the automatic detection. This is useful when runtime errors are encountered.

leaf_analysis_width_ratio
The ratio of the leaf width to use as part of the evaluation. E.g. if the ratio is 0.5, the center half of the leaf will be used. This helps avoid tongue and groove influence.

picket_spacing
If None (default), the spacing between pickets is determined automatically. If given, it should be an int or float specifying the number of **PIXELS** apart the pickets are.

height_threshold
The threshold that the MLC peak needs to be above to be considered a picket (vs background). Lower if not all leaves are being caught. Note that for FFF beams this would very likely need to be lowered.

edge_threshold
The threshold of pixel value standard deviation within the analysis window of the MLC leaf to be considered a full leaf. This is how pylinac removes MLCs that are eclipsed by the jaw. This also is how to omit or catch leaves at the edge of the field. Raise to catch more edge leaves.

peak_sort
Either 'peak_heights' or 'prominences'. This is the method for determining the peaks. Usually not needed unless the wrong number of pickets have been detected. See the scipy.signal.find_peaks function for more information.

required_prominence
The required height of the picket (not individual MLCs) to be considered a peak. Pylinac takes a mean of the image axis perpendicular to the leaf motion to get an initial guess of the peak locations and also to determine picket spacing. Changing this can be useful for wide-gap tests where the shape of the beam horns can form two or more local maximums in the picket area. Increase if for wide-gap images that are catching too many pickets. Consider lowering for FFF beams if there are analysis issues.

> **Warning:** We do not recommend performing FFF wide-gap PF tests. Make your FFF pickets narrow or measure with a flat beam instead.

fwxm
For each MLC kiss, the profile is a curve from low to high to low. The FWXM (0-100) is the height to use to measure to determine the center of the curve, which is the surrogate for MLC kiss position. I.e. for each MLC kiss, what height of the picket should you use to actually determine the center location? It is unusual to change this. If you have something in the way (we've seen crazy examples with a BB in the way) you may want to increase this.

separate_leaves
Whether to analyze leaves individually (each tip) or as a set (combined, center of the picket). False is the default for backwards compatibility.

nominal_gap_mm
The expected gap of the pickets in mm. Only used when separate leaves is True. Due to the DLG and EPID scattering, this value will have to be determined by you with a known good delivery.

central_axis
The central axis of the beam. If None (default), the CAX is automatically determined. This is used for French regulations where the CAX is set to the BB location from a separate image.

`plot_analyzed_image`(*guard_rails: bool = True*, *mlc_peaks: bool = True*, *overlay: bool = True*, *leaf_error_subplot: bool = True*, *show: bool = True*, *figure_size: str | tuple = 'auto'*) → None

Plot the analyzed image.

Parameters

guard_rails
Do/don't plot the picket "guard rails" around the ideal picket

mlc_peaks
Do/don't plot the detected MLC peak positions.

overlay
Do/don't plot the alpha overlay of the leaf status.

leaf_error_subplot
If True, plots a linked leaf error subplot adjacent to the PF image plotting the average and standard deviation of leaf error.

show
Whether to display the plot. Set to false for saving to a figure, etc.

figure_size
Either 'auto' or a tuple. If auto, the figure size is set depending on the orientation. If a tuple, this is the figure size to use.

save_analyzed_image(*filename: str | io.BytesIO*, *guard_rails: bool = True*, *mlc_peaks: bool = True*, *overlay: bool = True*, *leaf_error_subplot: bool = False*, ***kwargs*) → None

Save the analyzed figure to a file. See *plot_analyzed_image()* for further parameter info.

results(*as_list: bool = False*) → str

Return results of analysis. Use with print().

results_data(*as_dict=False*) → *PFResult* | dict

Present the results data and metadata as a dataclass, dict, or tuple. The default return type is a dataclass.

publish_pdf(*filename: str | io.BytesIO*, *notes: str = None*, *open_file: bool = False*, *metadata: dict = None*, *bins: int = 10*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

bins: int
Number of bins to show for the histogram

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

mlc_skew() → float

Apparent rotation in degrees of the MLC. This could be conflated with the EPID skew, so be careful when interpreting this value.

plot_histogram(*bins: int = 10*, *show: bool = True*) → None

Plot a histogram of the leaf errors

save_histogram(*filename: [str, Path, BinaryIO]*, *bins: int = 10*, ***kwargs*) → None

Save a histogram of the leaf errors

property orientation: *Orientation*

The orientation of the image, either Up-Down or Left-Right.

class pylinac.picketfence.**MLCArrangement**(*leaf_arrangement: list[tuple[int, float]]*, *offset: float = 0*)

Bases: object

Construct an MLC array

Parameters

leaf_arrangement
Description of the leaf arrangement. List of tuples containing the number of leaves and leaf width. E.g. (10, 5) is 10 leaves with 5mm widths.

offset
The offset in mm of the leaves. Used for asymmetric arrangements. E.g. -2.5mm will shift the arrangement 2.5mm to the left.

class pylinac.picketfence.**Orientation**(*value*)

Bases: Enum

Possible orientations of the image

UP_DOWN = 'Up-Down'

LEFT_RIGHT = 'Left-Right'

class pylinac.picketfence.**MLC**(*value*)

Bases: Enum

The pre-built MLC types

MILLENNIUM = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name': 'Millennium'}

HD_MILLENNIUM = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name': 'HD Millennium'}

BMOD = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name': 'B Mod'}

```
AGILITY = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name':
'Agility'}

MLCI = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name':
'MLCi'}

HALCYON_DISTAL = {'arrangement': <pylinac.picketfence.MLCArrangement object>,
'name': 'Halcyon distal'}

HALCYON_PROXIMAL = {'arrangement': <pylinac.picketfence.MLCArrangement object>,
'name': 'Halcyon proximal'}
```

class pylinac.picketfence.**PFResult**(*tolerance_mm: float*, *action_tolerance_mm: float*, *percent_leaves_passing: float*, *number_of_pickets: int*, *absolute_median_error_mm: float*, *max_error_mm: float*, *max_error_picket: int*, *max_error_leaf: str | int*, *mean_picket_spacing_mm: float*, *offsets_from_cax_mm: list[float]*, *passed: bool*, *failed_leaves: list[str] | list[int]*, *mlc_skew: float*, *picket_widths: dict[int, dict[str, float]]*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

```
tolerance_mm: float

action_tolerance_mm: float

percent_leaves_passing: float

number_of_pickets: int

absolute_median_error_mm: float

max_error_mm: float

max_error_picket: int

max_error_leaf: str | int

mean_picket_spacing_mm: float

offsets_from_cax_mm: list[float]

passed: bool

failed_leaves: list[str] | list[int]

mlc_skew: float

picket_widths: dict[int, dict[str, float]]
```

Supporting Classes

You generally won't have to interface with these unless you're doing advanced behavior.

class pylinac.picketfence.**PFDicomImage**(*path: str*, ***kwargs*)

Bases: *LinacDicomImage*

A subclass of a DICOM image that checks for noise and inversion when instantiated. Can also adjust for EPID sag.

Parameters

path
[str, file-object] The path to the file or the data stream.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

dpi
[int, float] The dots-per-inch of the image, defined at isocenter.

Note: If a DPI tag is found in the image, that value will override the parameter, otherwise this one will be used.

---

sid
[int, float] The Source-to-Image distance in mm.

sad
[float] The Source-to-Axis distance in mm.

raw_pixels
[bool] Whether to apply pixel intensity correction to the DICOM data. Typically, Rescale Slope, Rescale Intercept, and other tags are included and meant to be applied to the raw pixel data, which is potentially compressed. If True, no correction will be applied. This is typically used for scenarios when you want to match behavior to older or different software.

adjust_for_sag(*sag: int*, *orientation: str | Orientation*) → None

Roll the image to adjust for EPID sag.

property center: *Point*

Override the central axis call in the event we passed it directly

class pylinac.picketfence.**Picket**(*mlc_measurements: list[MLCValue]*, *log_fits*, *orientation: Orientation*, *image: PFDicomImage*, *tolerance: float*, *separate_leaves: bool*, *nominal_gap: float*)

Bases: `object`

Holds picket information in a Picket Fence test.

get_fit() → poly1d

The fit of a polynomial to the MLC measurements.

skew() → float

The slope/skew of the picket

property dist2cax: float

The distance from the CAX to the picket, in mm.

property left_guard_separated: Sequence[poly1d]

The line representing the left-sided guard rails. When not doing separate analysis, the left and right rails will overlap.

property right_guard_separated

The line representing the right-sided guard rails.

add_guards_to_axes(*axis: Axes*, *color: str = 'g'*) → None

Plot guard rails to the axis.

class pylinac.picketfence.**MLCValue**(*picket_num: int*, *approx_idx: int*, *leaf_width: float*, *leaf_center: float*, *picket_spacing: float*, *orientation:* Orientation, *leaf_analysis_width_ratio: float*, *tolerance: float*, *action_tolerance: float | None*, *leaf_num: int*, *approx_peak_val: float*, *image_window: np.ndarray*, *image:* PFDicomImage, *fwxm: int*, *separate_leaves: bool*, *nominal_gap_mm: float*)

Bases: `object`

Representation of an MLC kiss or of each MLC about a kiss.

property full_leaf_nums: Sequence[str | int]

The fully-qualified leaf names. This will be the simple leaf number for traditional analysis or the bank+leaf num for separate leaves.

plot2axes(*axes: plt.Axes*, *width: float | int = 1*) → None

Plot the measurement to the axes.

property passed: Sequence[bool]

Whether the MLC kiss or leaf was within tolerance.

property passed_action: Sequence[bool] | None

Whether the MLC kiss or leaf was within the action tolerance.

property bg_color: Sequence[str]

The color of the measurement when the PF image is plotted, based on pass/fail status.

property picket_positions: Sequence[float]

The position(s) of the pickets in mm

property error: Sequence[float]

The error (difference) of the MLC measurement and the picket fit. If using individual leaf analysis, returns both errors otherwise return one.

property max_abs_error: float

The maximum absolute error

property marker_lines: list[*Line*]

The line(s) representing the MLC measurement position. When using separated leaves there are two lines. Traditional analysis returns one.

plot_overlay2axes(*axes*) → None

Create a rectangle overlay with the width of the error. I.e. it stretches from the picket fit to the MLC position. Gives more visual size to the

6.13 Winston-Lutz

6.13.1 Overview

The Winston-Lutz module loads and processes EPID images that have acquired Winston-Lutz type images.

Features:

- **Couch shift instructions** - After running a WL test, get immediate feedback on how to shift the couch. Couch values can also be passed in and the new couch values will be presented so you don't have to do that pesky conversion. "Do I subtract that number or add it?"
- **Automatic field & BB positioning** - When an image or directory is loaded, the field CAX and the BB are automatically found, along with the vector and scalar distance between them.
- **Isocenter size determination** - Using backprojections of the EPID images, the 3D gantry isocenter size and position can be determined *independent of the BB position*. Additionally, the 2D planar isocenter size of the collimator and couch can also be determined.
- **Image plotting** - WL images can be plotted separately or together, each of which shows the field CAX, BB and scalar distance from BB to CAX.
- **Axis deviation plots** - Plot the variation of the gantry, collimator, couch, and EPID in each plane as well as RMS variation.
- **File name interpretation** - Rename DICOM filenames to include axis information for linacs that don't include such information in the DICOM tags. E.g. "myWL_gantry45_coll0_couch315.dcm".

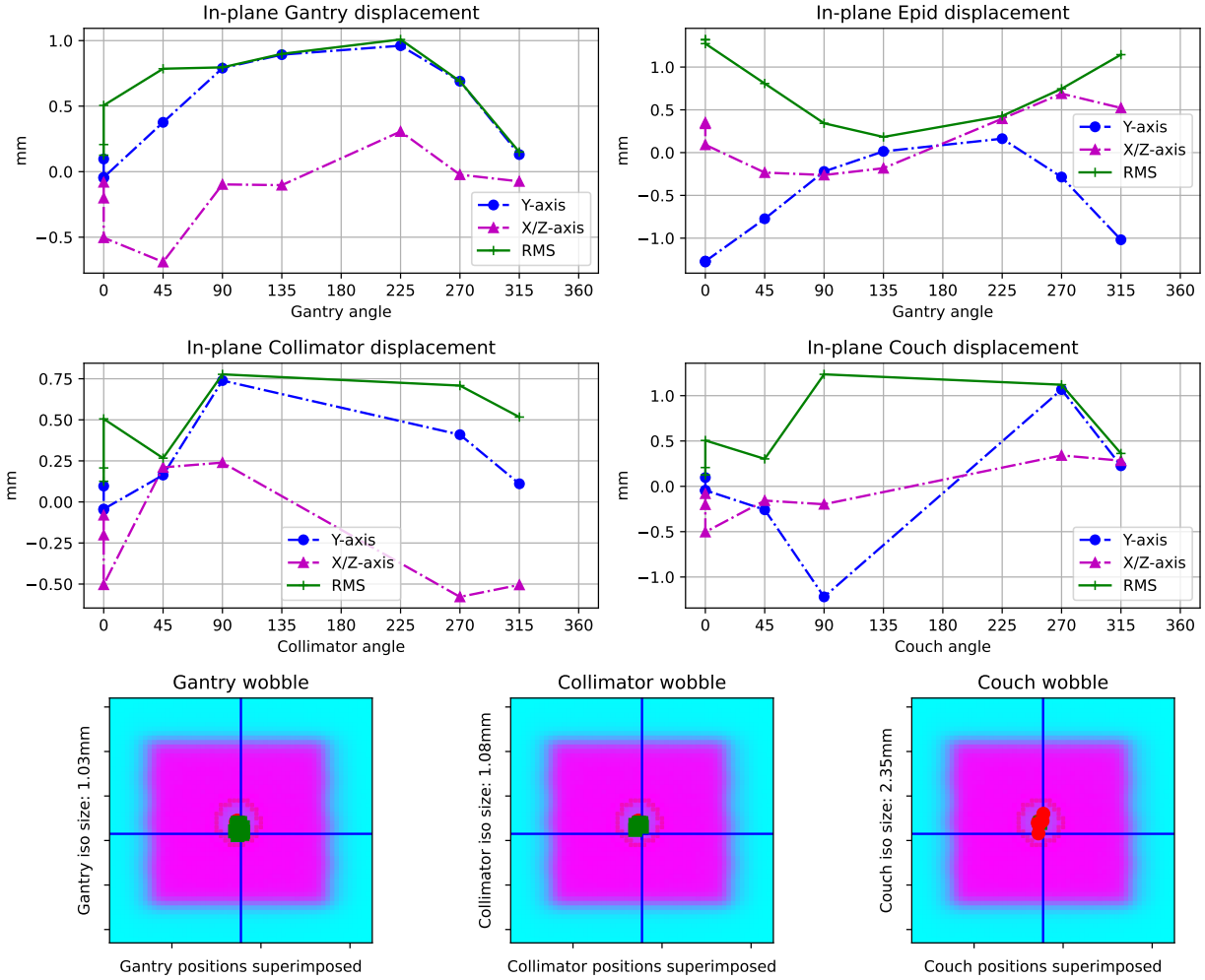
6.13.2 Running the Demo

To run the Winston-Lutz demo, create a script or start an interpreter session and input:

```
from pylinac import WinstonLutz

WinstonLutz.run_demo()
```

Results will be printed to the console and a figure showing the zoomed-in images will be generated:

```
Winston-Lutz Analysis
==============================
Number of images: 17
Maximum 2D CAX->BB distance: 1.23mm
Median 2D CAX->BB distance: 0.69mm
Shift to iso: facing gantry, move BB: RIGHT 0.36mm; OUT 0.36mm; DOWN 0.20mm
Gantry 3D isocenter diameter: 1.05mm (9/17 images considered)
Maximum Gantry RMS deviation (mm): 1.03mm
Maximum EPID RMS deviation (mm): 1.31mm
Gantry+Collimator 3D isocenter diameter: 1.11mm (13/17 images considered)
Collimator 2D isocenter diameter: 1.09mm (7/17 images considered)
Maximum Collimator RMS deviation (mm): 0.79
Couch 2D isocenter diameter: 2.32mm (7/17 images considered)
Maximum Couch RMS deviation (mm): 1.23
```


In-plane Gantry displacement
mm
Gantry angle
Y-axis
X/Z-axis
RMS
In-plane Epid displacement
mm
Gantry angle
Y-axis
X/Z-axis
RMS
In-plane Collimator displacement
mm
Collimator angle
Y-axis
X/Z-axis
RMS
In-plane Couch displacement
mm
Couch angle
Y-axis
X/Z-axis
RMS
Gantry wobble
Gantry iso size: 1.03mm
Gantry positions superimposed
Collimator wobble
Collimator iso size: 1.08mm
Collimator positions superimposed
Couch wobble
Couch iso size: 2.35mm
Couch positions superimposed

6.13.3 Image Acquisition

The Winston-Lutz module will only load EPID images. The images can be from any EPID however, and any SID. To ensure the most accurate results, a few simple tips should be followed. Note that these are not unique to pylinac; most Winston-Lutz analyses require these steps:

- The BB should be fully within the field of view.
- The MLC field should be symmetric.
- The BB should be <2cm from the isocenter.

Axis Values

Pylinac uses the *Image types & output definitions* definition to bin images. Regardless of the axis values, pylinac will calculate some values like max/median BB->CAX distance. Other values such as gantry iso size will only use Reference and Gantry image types as defined in the linked section. We recommend reviewing the analysis definitions and acquiring images according to the values you are interested in. Some examples are below. Note that these are not mutually exclusive:

- Simple max distance to BB: Any axis values; any combination of axis values are allowed.
- Gantry iso size: Gantry value can be any; all other axes must be 0.
- Collimator iso size: Collimator value can be any; all other axes must be 0.

If, e.g., all axis values are combinations of axes then gantry iso size will not be calculated. Further, the `plot_analyzed_image` method assumes Gantry, Collimator, and/or Couch image sets. If only combinations are passed, this image will be empty. A good practice is also to acquire a reference image if possible, meaning all axes at 0.

6.13.4 Coordinate Space

Note: In pylinac 2.3, the coordinates changed to be compliant with IEC 61217. Compared to previous versions, the Y and Z axis have been swapped. The new Z axis has also flipped which way is positive.

When interpreting results from a Winston-Lutz test, it's important to know the coordinates, origin, etc. Pylinac uses IEC 61217 coordinate space. Colloquial descriptions are as if standing at the foot of the couch looking at the gantry.

- **X-axis** - Lateral, or left-right, with right being positive.
- **Y-axis** - Superior-Inferior, or in-out, with sup/in being positive.
- **Z-axis** - Anterior-Posterior, or up-down, with up/anterior being positive.

Passing a coordinate system

New in version 3.6.

It is possible to pass in your machine's coordinate scale/system to the analyze parameter like so:

```
from pylinac.winston_lutz import WinstonLutz, MachineScale

wl = WinstonLutz(...)
```

(continues on next page)

(continued from previous page)

```
wl.analyze(..., machine_scale=MachineScale.VARIAN_IEC)
...
```

This will change the BB shift vector and shift instructions accordingly. If you don't use the shift vector or instructions then you won't need to worry about this parameter.

6.13.5 Typical Use

Analyzing a Winston-Lutz test is simple. First, let's import the class:

```
from pylinac import WinstonLutz
```

From here, you can load a directory:

```
my_directory = "path/to/wl_images"
wl = WinstonLutz(my_directory)
```

You can also load a ZIP archive with the images in it:

```
wl = WinstonLutz.from_zip("path/to/wl.zip")
```

Now, analyze it:

```
wl.analyze(bb_size_mm=5)
```

And that's it! You can now view images, print the results, or publish a PDF report:

```
# plot all the images
wl.plot_images()
# plot an individual image
wl.images[3].plot()
# save a figure of the image plots
wl.save_plots("wltest.png")
# print to PDF
wl.publish_pdf("mywl.pdf")
```

If you want to shift the BB based on the results and perform the test again there is a method for that:

```
print(wl.bb_shift_instructions())
# LEFT: 0.1mm, DOWN: 0.22mm, ...
```

You can also pass in your couch coordinates and the new values will be generated:

```
print(wl.bb_shift_instructions(couch_vrt=0.41, couch_lng=96.23, couch_lat=0.12))
# New couch coordinates (mm): VRT: 0.32; LNG: 96.11; LAT: 0.11
```

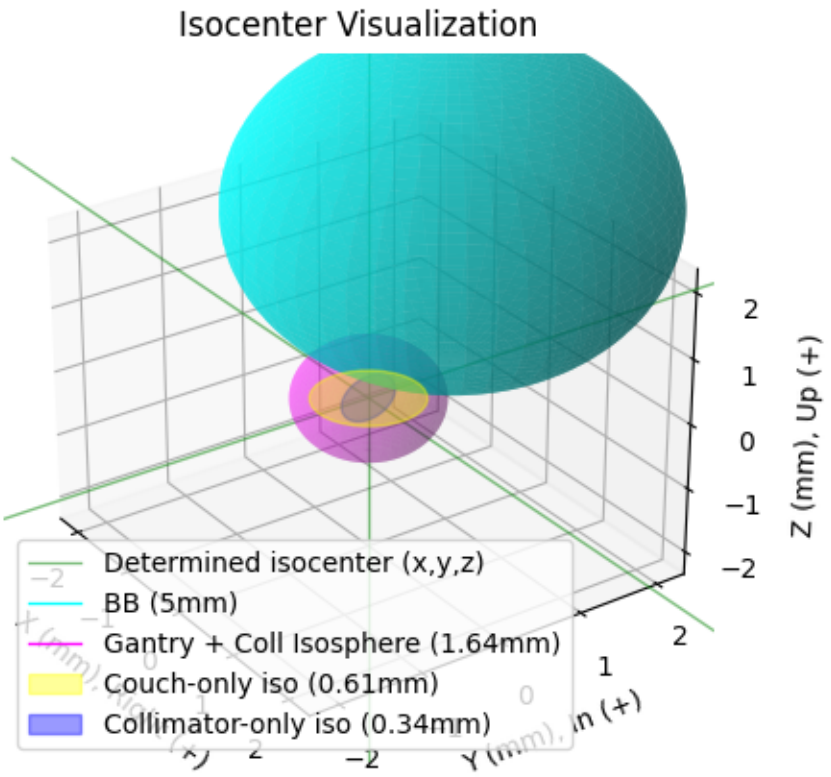
6.13.6 Visualizing the Isocenter-to-BB

New in version 3.17.

The isocenter and BB can be visualized together after analysis by calling `plot_location`:

```
wl = WinstonLutz(...)
wl.analyze(...)
wl.plot_location()
```

This will result in a 3D plot visualizing the BB (true physical size) and the isocenter (true physical size) in the room coordinates like so:



6.13.7 Accessing data

Changed in version 3.0.

Using the WL module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *results_data()* method which returns a *WinstonLutzResult* instance.

Note: While the pylinac tooling may change under the hood, this object should remain largely the same and/or expand. Thus, using this is more stable than accessing attrs directly.

Continuing from above:

```
data = wl.results_data()
data.num_total_images
data.max_2d_cax_to_bb_mm
# and more

# return as a dict
data_dict = wl.results_data(as_dict=True)
data_dict["num_total_images"]
...
```

6.13.8 Accessing individual images

Each image can be plotted and otherwise accessed easily:

```
wl = WinstonLutz(...)
# access first image
wl.images[
    0
]  # these are subclasses of the pylinac.core.image.DicomImage class, with a few special␣
↪props
# plot 3rd image
wl.images[
    0
].plot()  # the plot method is special to the WL module and shows the BB, EPID, and␣
↪Field CAX.
# get 2D x/y vector of an image
wl.images[
    4
].cax2bb_vector  # this is a Vector with a .x and .y attribute. Note that x and y are in␣
↪respect to the image, not the fixed room coordinates.
```

6.13.9 Analyzing a single image

You may optionally analyze a single image if that is your preference. Obviously, no 3D computations are performed.

Note: This is the same class used under the hood for the `WinstonLutz` images, so any attribute you currently use with something like `wl.images[2].cax2bb_vector` will work for the below with a direct call: `wl2d.cax2bb_vector`.

```
from pylinac import WinstonLutz2D

wl2d = WinstonLutz2D("my/path/...")
wl2d.analyze(bb_size_mm=4)  # same as WinstonLutz class
wl2d.plot()
...
```

This class does not have all the methods that `WinstonLutz` has for mostly obvious reasons and lower likelihood of being used directly.

6.13.10 Passing in Axis values

If your linac EPID images do not include axis information (such as Elekta) there are two ways to pass the data in. First, you can specify it in the file name. Any and all of the three axes can be defined. If one is not defined and is not in the DICOM tags, it will default to 0. The syntax to define the axes: "<*>gantry0<*>coll0<*>couch0<*>". There can be any text before, after, or in between each axis definition. However, the axes numerical value **must** immediately follow the axis name. Axis names are also fixed. The following examples are valid:

- MyWL-gantry0-coll90-couch315.dcm
- gantry90_stuff_coll45-couch0.dcm
- abc-couch45-gantry315-coll0.dcm
- 01-gantry0-abcd-coll30couch10abc.dcm
- abc-gantry30.dcm
- coll45abc.dcm

The following are invalid:

- mywl-gantry=0-coll=90-couch=315.dcm
- gan45_collimator30-table270.dcm

Using the filenames within the code is done by passing the `use_filenames=True` flag to the init method:

```
my_directory = "path/to/wl_images"
wl = WinstonLutz(my_directory, use_filenames=True)
```

Note: If using filenames any relevant axes must be defined, otherwise they will default to zero. For example, if the acquisition was at gantry=45, coll=15, couch=0 then the filename must include both the gantry and collimator in the name (<...gantry45...coll15....dcm>). For this example, the couch need not be defined since it is 0.

The other way of inputting axis information is passing the `axis_mapping` parameter to the constructor. This is a dictionary with the filenames as keys and a tuple of ints for the gantry, coll, and couch:

```
directory = "path/to/wl/dir"
mapping = {
    "file1.dcm": (0, 0, 0),
    "file2.dcm": (90, 315, 45),
}  # add more as needed
wl = WinstonLutz(directory=directory, axis_mapping=mapping)
# analyze as normal
wl.analyze(...)
```

Note: The filenames should be local to the directory. In the above example the full paths would be `path/to/wl/dir/file1.dcm`, and `path/to/wl/dir/file2.dcm`.

6.13.11 Changing BB detection size

To change the size of BB pylinac is expecting you can pass the size to the analyze method:

```
import pylinac

wl = WinstonLutz(...)
wl.analyze(bb_size_mm=3)
...
```

6.13.12 Low-density BBs

If using a phantom with a BB that has a lower density that than the surrounding material, pass the `low_density_bb` parameter:

```
import pylinac

wl = WinstonLutz(...)
wl.analyze(..., low_density_bb=True)
...
```

6.13.13 kV Analysis/Imaging-only iso evaluation

It is possible to analyze kV WL images and/or analyze a WL set and only focus on the imaging iso. In this case there are two parameters you likely need to adjust: `open_field` and `low_density_bb`. The first will set the field center to the image center. It is assumed the field is not of interest or the field cannot be measured, such as a fully-open kV image. Use this anytime the radiation iso is not of interest. For large-field WL images, you may need to set the `low_density_bb` parameter to True. This is because the automatic inversion of the WL module assumes a small field is being delivered. For large field deliveries, kV or MV, see about flipping this parameter if the analysis fails.

6.13.14 CBCT Analysis

New in version 3.16.

> **Warning:** This feature is still experimental. Use with caution.

It's possible to take and load a CBCT dataset of a BB using the `from_cbct` and `from_cbct_zip` class methods. The CBCT dataset is loaded as a 3D numpy array. Projections at the 4 faces of the array (top, left, bottom, right) are created into pseudo-cardinal angle DICOMs. These DICOMs are then loaded as normal images and analyzed.

Listing 1: Example of loading and analyzing a CBCT dataset of a WL BB

```
wl = WinstonLutz.from_cbct("my/cbct/dir")
# OR
wl = WinstonLutz.from_cbct_zip("my/cbct.zip")
# ensure to set low density and open field to True
wl.analyze(low_density_bb=True, open_field=True, bb_size_mm=3)
# use as normal
```

(continues on next page)

(continued from previous page)

```
print(wl.results())
print(wl.results_data())
print(wl.bb_shift_instructions())
wl.plot_images()
```

Warning: The CBCT analysis comes with a few caveats:

- Analyzing the image will override the `low_density_bb` and `open_field` flags to always be True; it does not matter what is passed in `analyze`.
- No axis deviation information is available, i.e. couch/coll/gantry walkout.
- There are always 4 images generated.
- The generated images are not true DICOMs and thus do not have all the DICOM tags.

6.13.15 Using TIFF images

New in version 3.12.

The WL module can handle TIFF images on a provisional basis.

Warning: This is still experimental and caution is warranted. Even though there is an automatic noise/edge cleaner, cropping images to remove markers and/or film scan artifacts is encouraged.

To load TIFF images, extra parameters must be passed. Specifically, the `sid` and potentially the `dpi` parameters must be added. Additionally, `axis_mapping` must be populated. This is how pylinac can convert the images into rudimentary dicom images. The `dpi` parameter is only needed if the TIFF images do not have a resolution tag. Pylinac will give a specific error if `dpi` wasn't passed and also wasn't in the TIFF tags.

Note: Although it is technically possible to load both DICOM and TIFF together in one dataset it is not encouraged.

```
from pylinac import WinstonLutz

my_tiff_images = list(Path(...), Path(...))
wl_tiff = WinstonLutz(
    my_tiff_images,
    sid=1000,
    dpi=212,
    axis_mapping={"g0.tiff": (0, 0, 0), "g270.tiff": (270, 0, 0)},
)
# now analyze as normal
wl_tiff.analyze(...)
print(wl_tiff.results())
```

Note that other `.from...` methods are available such as `.from_zip`:

```
from pylinac import WinstonLutz
```

(continues on next page)

(continued from previous page)

```
my_tiff_zip = "../files/tiffs.zip"
# same inputs as above
wl_tiff = WinstonLutz.from_zip(my_tiff_zip, dpi=...)
```

6.13.16 Image types & output definitions

The following terms are used in pylinac's WL module and are worth defining.

Image axis definitions/Image types Images are classified into 1 of 6 image types, depending on the position of the axes. The image type is then used for determining whether to use the image for the given calculation. Image types allow the module to isolate the analysis to a given axis if needed. E.g. for gantry iso size, as opposed to overall iso size, only the gantry should be moving so that no other variables influence it's calculation.

- **Reference**: This is when all axes are at value 0 (gantry=coll=couch=0).
- **Gantry**: This is when all axes but gantry are at value 0, e.g. gantry=45, coll=0, couch=0.
- **Collimator**: This is when all axes but collimator are at value 0.
- **Couch**: This is when all axes but the couch are at value 0.
- **GB Combo**: This is when either the gantry or collimator are non-zero but the couch is 0.
- **GBP Combo**: This is where the couch is kicked and the gantry and/or collimator are rotated.

Analysis definitions Given the above terms, the following calculations are performed.

- **Maximum 2D CAX->BB distance (scalar, mm)**: Analyzes all images individually for the maximum 2D distance from rad field center to the BB.
- **Median 2D CAX->BB distance (scalar, mm)**: Same as above but the median.
- **Shift of BB to isocenter (vector, mm)**: The instructions of how to move the BB/couch in order to place the BB at the determined isocenter.
- **Gantry 3D isocenter diameter (scalar, mm)**: Analyzes only the gantry axis images (see above image types). Applies backprojection of the CAX in 3D and then minimizes a sphere that touches all the 3D backprojection lines.
- **Gantry+Collimator 3D isocenter diameter (scalar, mm)**: Same as above but also considers Collimator and GB Combo images.
- **[Couch, Collimator] 2D isocenter diameter (scalar, mm)**: Analyzes only the collimator or couch images to determine the size of the isocenter according to the axis in question. The maximum distance between any of the points is the isocenter size. The couch and collimator are treated separately for obvious reasons. If no images are given that rotate about the axis in question (e.g. cardinal gantry angles only) the isocenter size will default to 0.
- **[Maximum, All][Gantry, Collimator, Couch, GB Combo, GBP Combo, EPID] RMS deviation (array of scalars, mm)**: Analyzes the images for the axis in question to determine the overall RMS inclusive of all 3 coordinate axes (vert, long, lat). I.e. this is the overall displacement as a function of the axis in question. For EPID, the displacement is calculated as the distance from image center to BB for all images with couch=0. If no images are given that rotate about the axis in question (e.g. cardinal gantry angles only) the isocenter size will default to 0.

6.13.17 Algorithm

The Winston-Lutz algorithm is based on the works of Winkler et al, Du et al, and Low et al. Winkler found that the collimator and couch iso could be found using a minimum optimization of the field CAX points. They also found that the gantry isocenter could by found by "backprojecting" the field CAX as a line in 3D coordinate space, with the BB being the reference point. This method is used to find the gantry isocenter size.

Low determined the geometric transformations to apply to 2D planar images to calculate the shift to apply to the BB. This method is used to determine the shift instructions. Specifically, equations 6 and 9.

Note: If doing research, it is very important to note that Low implicitly used the "Varian" coordinate system. This is an old coordinate system and any new Varian linac actually uses IEC 61217. However, because the gantry and couch definitions are different, the matrix definitions are technically incorrect when using IEC 61217. By default, Pylinac assumes the images are in IEC 61217 scale and will internally convert it to varian scale to be able to use Low's equations. To use a different scale use the `machine_scale` parameter, shown here *Passing a coordinate system*. Also see *Machine Scale*.

The algorithm works like such:

Allowances

- The images can be acquired with any EPID (aS500, aS1000, aS1200) at any SID.
- The BB does not need to be near the real isocenter to determine isocenter sizes, but does affect the 2D image analysis.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The BB must be fully within the field of view.
- The BB must be within 2.0cm of the real isocenter.
- The images must be acquired with the EPID.
- The linac scale should be IEC 61217.

Analysis

- **Find the field CAX** – The spread in pixel values (max - min) is divided by 2, and any pixels above the threshold is associated with the open field. The pixels are converted to black & white and the center of mass of the pixels is assumed to be the field CAX.
- **Find the BB** – The algorithm for finding the BB can be found here: *Algorithm*.

Note: Strictly speaking, the following aren't required analyses, but are explained for fullness and clarity.

- **Backproject the CAX for gantry images** – Based on the vector of the BB to the field CAX and the gantry angle, a 3D line projection of the CAX is constructed. The BB is considered at the origin. Only images where the couch was at 0 are used for CAX projection lines.
- **Determine gantry isocenter size** - Using the backprojection lines, an optimization function is run to minimize the maximum distance to any line. The optimized distance is the isocenter radius.
- **Determine collimator isocenter size** - The maximum distance between any two field CAX locations is calculated for all collimator images.

- **Determine couch isocenter size** - Instead of using the BB as the non-moving reference point, which is now moving with the couch, the Reference image (gantry = collimator = couch = 0) CAX location is the reference. The maximum distance between any two BB points is calculated and taken as the isocenter size.

Note: Collimator iso size is always in the plane normal to the gantry, while couch iso size is always in the x-z plane.

6.13.18 Benchmarking the Algorithm

With the image generator module we can create test images to test the WL algorithm on known results. This is useful to isolate what is or isn't working if the algorithm doesn't work on a given image and when commissioning pylinac. It is common, especially with the WL module, to question the accuracy of the algorithm. Since no linac is perfect and the results are sub-millimeter, discerning what is true error vs algorithmic error can be difficult. The image generator module is a perfect solution since it can remove or reproduce the former error.

Perfect Delivery

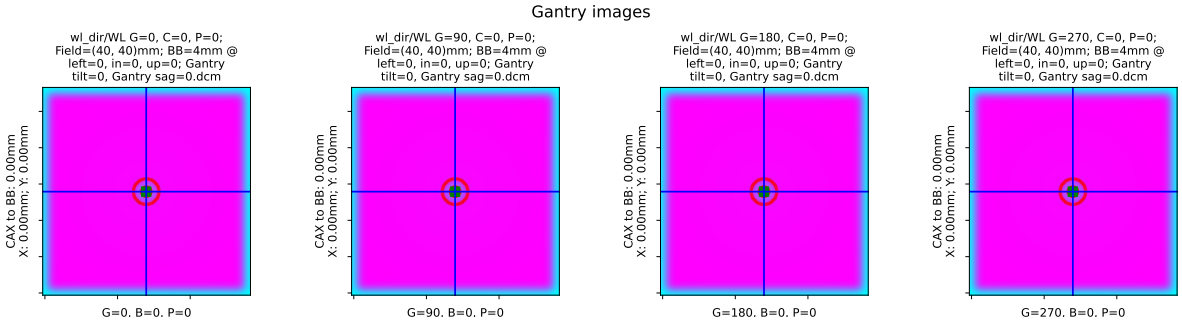
Let's deliver a set of perfect images. This should result in near-0 deviations and isocenter size. The utility function used here will produce 4 images at the 4 cardinal gantry angles with all other axes at 0, with a BB of 4mm diameter, and a field size of 4x4cm:

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz,
)

wl_dir = 'wl_dir'
generate_winstonlutz(
    AS1200Image(),
    FilteredFieldLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    field_size_mm=(40, 40),
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4)
wl.plot_images()
```

which has an output of:

```
Winston-Lutz Analysis
================================
Number of images: 4
Maximum 2D CAX->BB distance: 0.00mm
Median 2D CAX->BB distance: 0.00mm
Shift to iso: facing gantry, move BB: RIGHT 0.00mm; IN 0.00mm; UP 0.00mm
```

(continues on next page)



(continued from previous page)

```
Gantry 3D isocenter diameter: 0.00mm (4/4 images considered)
Maximum Gantry RMS deviation (mm): 0.00mm
Maximum EPID RMS deviation (mm): 0.00mm
Gantry+Collimator 3D isocenter diameter: 0.00mm (4/4 images considered)
Collimator 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Collimator RMS deviation (mm): 0.00
Couch 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Couch RMS deviation (mm): 0.00
```

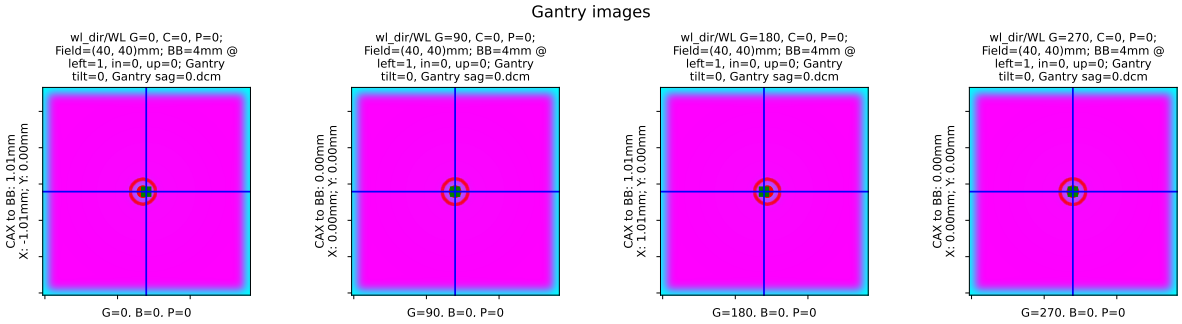
As shown, we have perfect results.

Offset BB

Let's now offset the BB by 1mm to the left:

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz,
)

wl_dir = 'wl_dir'
generate_winstonlutz(
    AS1200Image(),
    FilteredFieldLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    field_size_mm=(40, 40),
    offset_mm_left=1,
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4)
wl.plot_images()
```

with an output of:




```
Winston-Lutz Analysis
=================================
Number of images: 4
Maximum 2D CAX->BB distance: 1.01mm
Median 2D CAX->BB distance: 0.50mm
Shift to iso: facing gantry, move BB: RIGHT 1.01mm; IN 0.00mm; UP 0.00mm
Gantry 3D isocenter diameter: 0.00mm (4/4 images considered)
Maximum Gantry RMS deviation (mm): 1.01mm
Maximum EPID RMS deviation (mm): 0.00mm
Gantry+Collimator 3D isocenter diameter: 0.00mm (4/4 images considered)
Collimator 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Collimator RMS deviation (mm): 0.00
Couch 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Couch RMS deviation (mm): 0.00
```

We have correctly found that the max distance is 1mm and the required shift to iso is 1mm to the right (since we placed the bb to the left).

Gantry Tilt

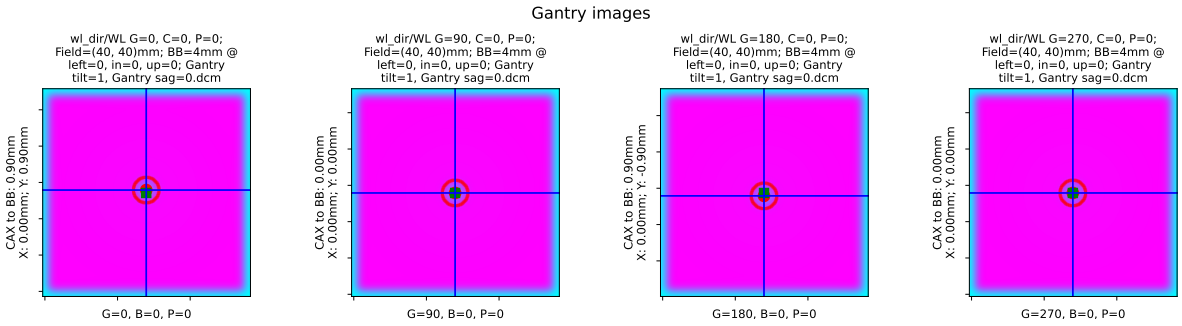
We can simulate gantry tilt, where at 0 and 180 the gantry tilts forward and backward respectively. We use a realistic value of 1mm. Note that everything else is perfect:

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz,
)

wl_dir = 'wl_dir'
generate_winstonlutz(
    AS1200Image(),
    FilteredFieldLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    field_size_mm=(40, 40),
    gantry_tilt=1,
```

(continues on next page)

(continued from previous page)

```
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4)
wl.plot_images()
```



with output of:

```
Winston-Lutz Analysis
=================================
Number of images: 4
Maximum 2D CAX->BB distance: 0.90mm
Median 2D CAX->BB distance: 0.45mm
Shift to iso: facing gantry, move BB: LEFT 0.00mm; IN 0.00mm; UP 0.00mm
Gantry 3D isocenter diameter: 1.79mm (4/4 images considered)
Maximum Gantry RMS deviation (mm): 0.90mm
Maximum EPID RMS deviation (mm): 0.90mm
Gantry+Collimator 3D isocenter diameter: 1.79mm (4/4 images considered)
Collimator 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Collimator RMS deviation (mm): 0.00
Couch 2D isocenter diameter: 0.00mm (1/4 images considered)
Maximum Couch RMS deviation (mm): 0.00
```

Note that since the tilt is symmetric the shift to iso is 0 despite our non-zero median distance. I.e. we are at iso, the iso just isn't perfect and we are thus at the best possible position.

Perfect Multi-Axis

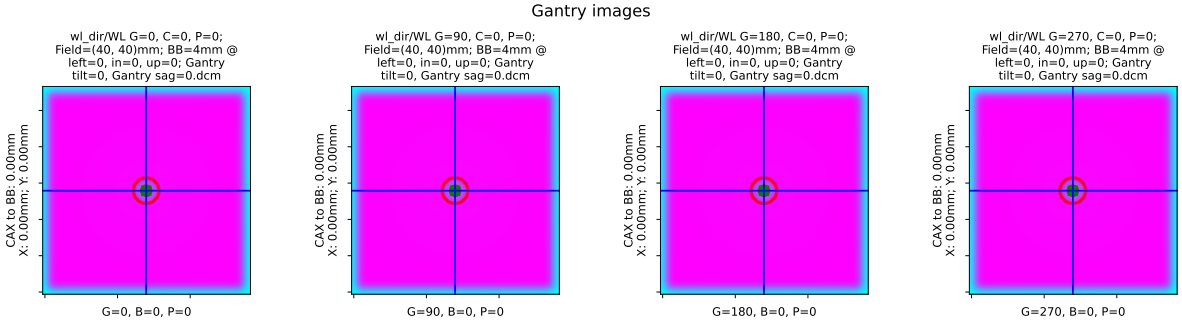
We can also vary the axis data for the images produced. Below we create a typical multi-axis WL with varying gantry, collimator, and couch (cardinal values for axis of interest with all other axes at 0):

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz,
)

wl_dir = 'wl_dir'
```

(continues on next page)

(continued from previous page)

```
generate_winstonlutz(
    AS1200Image(),
    FilteredFieldLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    field_size_mm=(40, 40),
    image_axes=[(0, 0, 0), (0, 90, 0), (0, 270, 0),
                (90, 0, 0), (180, 0, 0), (270, 0, 0),
                (0, 0, 90), (0, 0, 270)]
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4)
wl.plot_images()
```



Perfect Cone

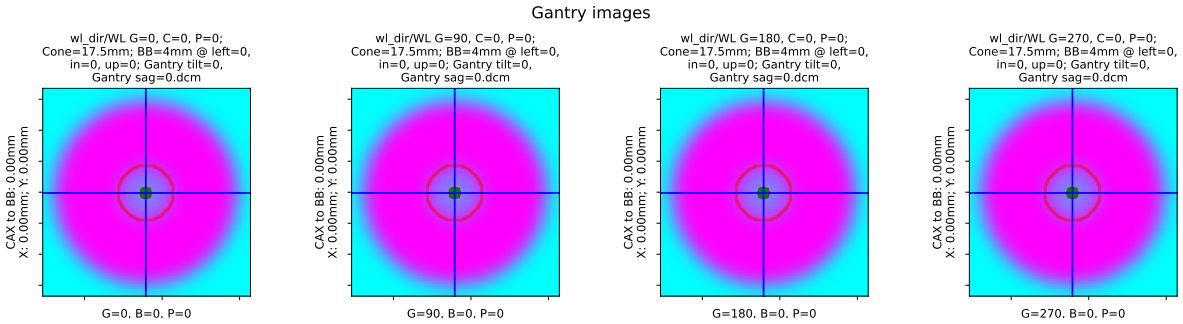
We can also look at simulated cone WL images. Here we use the 17.5mm cone:

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz, generate_winstonlutz_cone, FilterFreeConeLayer,
)

wl_dir = 'wl_dir'
generate_winstonlutz_cone(
    AS1200Image(),
    FilterFreeConeLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    cone_size_mm=17.5,
)

wl = pylinac.WinstonLutz(wl_dir)
```

(continues on next page)

(continued from previous page)

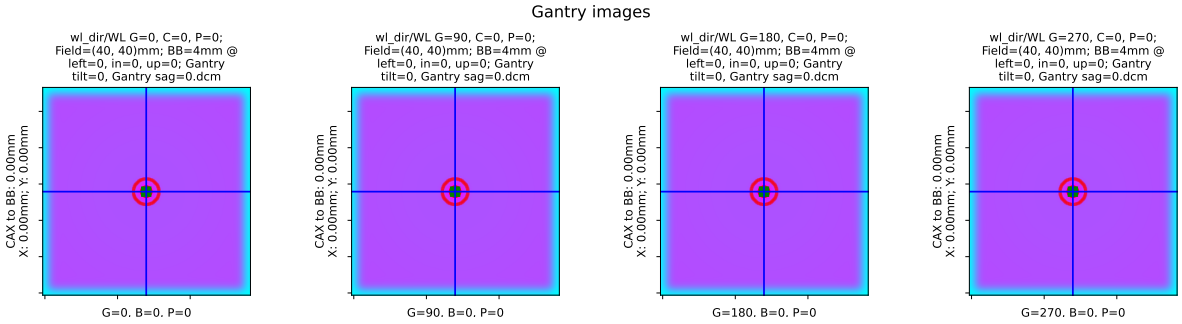
```
wl.analyze(bb_size_mm=4)
wl.plot_images()
```



Low-density BB

Simulate a low-density BB surrounded by higher-density material:

```
import pylinac
from pylinac.core.image_generator import (
    GaussianFilterLayer,
    FilteredFieldLayer,
    AS1200Image,
    RandomNoiseLayer,
    generate_winstonlutz,
)

wl_dir = 'wl_dir'
generate_winstonlutz(
    AS1200Image(),
    FilteredFieldLayer,
    dir_out=wl_dir,
    final_layers=[GaussianFilterLayer(),],
    bb_size_mm=4,
    field_size_mm=(40, 40),
    field_alpha=0.6,  # set the field to not max out
    bb_alpha=0.3  # normally this is negative to attenuate the beam, but here we␣
↪increase the signal
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4, low_density_bb=True)
wl.plot_images()
```



6.13.19 API Documentation

class pylinac.winston_lutz.**WinstonLutz**(*directory: str | list[str] | Path*, *use_filenames: bool = False*, *axis_mapping: dict[str, tuple[int, int, int]] | None = None*, *axes_precision: int | None = None*, *dpi: float | None = None*, *sid: float | None = None*)

Bases: `object`

Class for performing a Winston-Lutz test of the radiation isocenter.

Parameters

directory
: [str, list[str]] Path to the directory of the Winston-Lutz EPID images or a list of the image paths

use_filenames: bool
: Whether to try to use the file name to determine axis values. Useful for Elekta machines that do not include that info in the DICOM data. This is mutually exclusive to axis_mapping. If True, axis_mapping is ignored.

axis_mapping: dict
: An optional way of instantiating by passing each file along with the axis values. Structure should be <filename>: (<gantry>, <coll>, <couch>).

axes_precision: int | None
: How many significant digits to represent the axes values. If None, no precision is set and the input/DICOM values are used raw. If set to an integer, rounds the axes values (gantry, coll, couch) to that many values. E.g. gantry=0.1234 => 0.1 with precision=1. This is mostly useful for plotting/rounding (359.9=>0) and if using the `keyed_image_details` with `results_data`.

dpi
: The dots-per-inch setting. Only needed if using TIFF images and the images do not contain the resolution tag. An error will raise if dpi is not passed and the TIFF resolution cannot be determined.

sid
: The Source-to-Image distance in mm. Only needed when using TIFF images.

machine_scale: *MachineScale*

image_type
: alias of *WinstonLutz2D*

images: list[*WinstonLutz2D*]

classmethod from_demo_images(***kwargs*)
: Instantiate using the demo images.

Parameters

kwargs
See parameters of the __init__ method for details.

`classmethod from_zip`(*zfile: str | BinaryIO*, ***kwargs*)
Instantiate from a zip file rather than a directory.

Parameters

zfile
Path to the archive file.

kwargs
See parameters of the __init__ method for details.

`classmethod from_url`(*url: str*, ***kwargs*)
Instantiate from a URL.

Parameters

url
[str] URL that points to a zip archive of the DICOM images.

kwargs
See parameters of the __init__ method for details.

`classmethod from_cbct_zip`(*file: Path | str*, *raw_pixels: bool = False*, ***kwargs*)
Instantiate from a zip file containing CBCT images.

Parameters

file
Path to the archive file.

raw_pixels
If True, uses the raw pixel values of the DICOM files. If False, uses the rescaled Hounsfield units. Generally, this should be true.

kwargs
See parameters of the __init__ method for details.

`classmethod from_cbct`(*directory: Path | str*, *raw_pixels: bool = False*, ***kwargs*)
Create a 4-angle WL test from a CBCT dataset.

The dataset is loaded and the array is "viewed" from top, bottom, left, and right to create the 4 angles. The dataset has to be rescaled so that the z-axis spacing is equal to the x/y axis. This is because the typical slice thickness is much larger than the in-plane resolution.

Parameters

directory
The directory containing the CBCT DICOM files.

raw_pixels
If True, uses the raw pixel values of the DICOM files. If False, uses the rescaled Hounsfield units. Generally, this should be true.

kwargs
See parameters of the __init__ method for details.

`static run_demo()` → None

Run the Winston-Lutz demo, which loads the demo files, prints results, and plots a summary image.

`analyze(`*bb_size_mm: float = 5*, *machine_scale:* MachineScale = *MachineScale.IEC61217*, *low_density_bb: bool = False*, *open_field: bool = False*`)` → None

Analyze the WL images.

Parameters

bb_size_mm
The expected diameter of the BB in mm. The actual size of the BB can be +/-2mm from the passed value.

machine_scale
The scale of the machine. Shift vectors depend on this value.

low_density_bb
Set this flag to True if the BB is lower density than the material surrounding it.

open_field
If True, sets the field center to the EPID center under the assumption the field is not the focus of interest or is too wide to be calculated. This is often helpful for kV WL analysis where the blades are wide open and even then the blade edge is of less interest than simply the imaging iso vs the BB.

`property gantry_iso_size: float`

The diameter of the 3D gantry isocenter size in mm. Only images where the collimator and couch were at 0 are used to determine this value.

`property gantry_coll_iso_size: float`

The diameter of the 3D gantry isocenter size in mm *including collimator and gantry/coll combo images*. Images where the couch!=0 are excluded.

`property collimator_iso_size: float`

The 2D collimator isocenter size (diameter) in mm. The iso size is in the plane normal to the gantry.

`property couch_iso_size: float`

The diameter of the 2D couch isocenter size in mm. Only images where the gantry and collimator were at zero are used to determine this value.

`property bb_shift_vector: Vector`

The shift necessary to place the BB at the radiation isocenter. The values are in the coordinates defined in the documentation.

The shift is based on the paper by Low et al. See online documentation for more.

bb_shift_instructions(*couch_vrt: float | None = None*, *couch_lng: float | None = None*, *couch_lat: float | None = None*) → str

Returns a string describing how to shift the BB to the radiation isocenter looking from the foot of the couch. Optionally, the current couch values can be passed in to get the new couch values. If passing the current couch position all values must be passed.

Parameters

couch_vrt
[float] The current couch vertical position in cm.

couch_lng
[float] The current couch longitudinal position in cm.

couch_lat
[float] The current couch lateral position in cm.

axis_rms_deviation(*axis: Axis | tuple[Axis, ...] = Axis.GANTRY*, *value: str = 'all'*) → Iterable | float

The RMS deviations of a given axis/axes.

Parameters

axis
[('Gantry', 'Collimator', 'Couch', 'Epid', 'GB Combo', 'GBP Combo')] The axis desired.

value
[{'all', 'range'}] Whether to return all the RMS values from all images for that axis, or only return the maximum range of values, i.e. the 'sag'.

cax2bb_distance(*metric: str = 'max'*) → float

The distance in mm between the CAX and BB for all images according to the given metric.

Parameters

metric
[{'max', 'median', 'mean'}] The metric of distance to use.

cax2epid_distance(*metric: str = 'max'*) → float

The distance in mm between the CAX and EPID center pixel for all images according to the given metric.

Parameters

metric
[{'max', 'median', 'mean'}] The metric of distance to use.

plot_axis_images(*axis: Axis = Axis.GANTRY*, *show: bool = True*, *ax: plt.Axes | None = None*) → None

Plot all CAX/BB/EPID positions for the images of a given axis.

For example, axis='Couch' plots a reference image, and all the BB points of the other images where the couch was moving.

Parameters

axis
[{'Gantry', 'Collimator', 'Couch', 'GB Combo', 'GBP Combo'}] The images/markers from which accelerator axis to plot.

show
[bool] Whether to actually show the images.

ax
[None, matplotlib.Axes] The axis to plot to. If None, creates a new plot.

`plot_location`(*show: bool = True*, *viewbox_mm: float | None = None*, *plot_bb: bool = True*, *plot_isocenter_sphere: bool = True*, *plot_couch_iso: bool = True*, *plot_coll_iso: bool = True*, *show_legend: bool = True*)

Plot the isocenter and size as a sphere in 3D space relative to the BB. The iso is at the origin.

Only images where the couch was at zero are considered.

Parameters

show
[bool] Whether to plot the image.

viewbox_mm
[float] The default size of the 3D space to plot in mm in each axis.

plot_bb
[bool] Whether to plot the BB location; the size is also considered.

plot_isocenter_sphere
[bool] Whether to plot the gantry + collimator isocenter size.

plot_couch_iso
[bool] Whether to plot the couch-plane-only isocenter size. This will be zero if there are no images where the couch rotated.

plot_coll_iso
[bool] Whether to plot the collimator-plane-only isocenter size. This is shown along the Z/Y plane only to differentiate from the couch iso visualization. The collimator plane is always normal to the gantry angle. This will be zero if there are no images where the collimator rotated.

show_legend
[bool] Whether to show the legend.

`plot_images`(*axis: Axis = Axis.GANTRY*, *show: bool = True*, *split: bool = False*, ***kwargs*)

Plot a grid of all the images acquired.

Four columns are plotted with the titles showing which axis that column represents.

Parameters

axis
[{'Gantry', 'Collimator', 'Couch', 'GB Combo', 'GBP Combo', 'All'}] The axis to plot.

show
[bool] Whether to show the image.

split
[bool] Whether to show/plot the images individually or as one large figure.

save_images(*filename: str | BinaryIO*, *axis:* Axis = *Axis.GANTRY*, ***kwargs*) → None
Save the figure of *plot_images()* to file. Keyword arguments are passed to *matplotlib.pyplot.savefig()*.

Parameters

filename
[str] The name of the file to save to.

axis
The axis to save.

save_images_to_stream(***kwargs*) → dict[str, BytesIO]
Save the individual image plots to stream

plot_summary(*show: bool = True*, *fig_size: tuple | None = None*) → None
Plot a summary figure showing the gantry sag and wobble plots of the three axes.

save_summary(*filename: str | BinaryIO*, ***kwargs*) → None
Save the summary image.

results(*as_list: bool = False*) → str
Return the analysis results summary.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → *WinstonLutzResult* | dict
Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

publish_pdf(*filename: str*, *notes: str | list[str] | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

class pylinac.winston_lutz.**WinstonLutzResult**(*num_gantry_images: int*, *num_gantry_coll_images: int*, *num_coll_images: int*, *num_couch_images: int*, *num_total_images: int*, *max_2d_cax_to_bb_mm: float*, *median_2d_cax_to_bb_mm: float*, *mean_2d_cax_to_bb_mm: float*, *max_2d_cax_to_epid_mm: float*, *median_2d_cax_to_epid_mm: float*, *mean_2d_cax_to_epid_mm: float*, *gantry_3d_iso_diameter_mm: float*, *max_gantry_rms_deviation_mm: float*, *max_epid_rms_deviation_mm: float*, *gantry_coll_3d_iso_diameter_mm: float*, *coll_2d_iso_diameter_mm: float*, *max_coll_rms_deviation_mm: float*, *couch_2d_iso_diameter_mm: float*, *max_couch_rms_deviation_mm: float*, *image_details: list[*WinstonLutz2DResult*]*, *keyed_image_details: dict[str,* WinstonLutz2DResult*]*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

num_gantry_images: int

num_gantry_coll_images: int

num_coll_images: int

num_couch_images: int

num_total_images: int

max_2d_cax_to_bb_mm: float

median_2d_cax_to_bb_mm: float

mean_2d_cax_to_bb_mm: float

max_2d_cax_to_epid_mm: float

median_2d_cax_to_epid_mm: float

mean_2d_cax_to_epid_mm: float

gantry_3d_iso_diameter_mm: float

max_gantry_rms_deviation_mm: float

max_epid_rms_deviation_mm: float

gantry_coll_3d_iso_diameter_mm: float

coll_2d_iso_diameter_mm: float

max_coll_rms_deviation_mm: float

couch_2d_iso_diameter_mm: float

max_couch_rms_deviation_mm: float

image_details: list[*WinstonLutz2DResult*]

keyed_image_details: dict[str, *WinstonLutz2DResult*]

class pylinac.winston_lutz.**WinstonLutz2D**(*file: str | BinaryIO | Path*, *use_filenames: bool = False*, ***kwargs*)

Bases: *LinacDicomImage*

Holds individual Winston-Lutz EPID images, image properties, and automatically finds the field CAX and BB.

Parameters

file
: [str] Path to the image file.

use_filenames: bool
: Whether to try to use the file name to determine axis values. Useful for Elekta machines that do not include that info in the DICOM data.

analyze(*bb_size_mm: float = 5*, *low_density_bb: bool = False*, *open_field: bool = False*) → None
: Analyze the image. See WinstonLutz.analyze for parameter details.

to_axes() → str
: Give just the axes values as a human-readable string

property epid: *Point*
: Center of the EPID panel

property cax_line_projection: *Line*
: The projection of the field CAX through space around the area of the BB. Used for determining gantry isocenter size.

Returns

Line
The virtual line in space made by the beam CAX.

property cax2bb_vector: *Vector*
The vector in mm from the CAX to the BB.

property cax2bb_distance: float
The scalar distance in mm from the CAX to the BB.

property cax2epid_vector: *Vector*
The vector in mm from the CAX to the EPID center pixel

property cax2epid_distance: float
The scalar distance in mm from the CAX to the EPID center pixel

plot(*ax: plt.Axes | None = None*, *show: bool = True*, *clear_fig: bool = False*) → plt.Axes
Plot the image, zoomed-in on the radiation field, along with the detected BB location and field CAX location.

Parameters

ax
[None, matplotlib Axes instance] The axis to plot to. If None, will create a new figure.

show
[bool] Whether to actually show the image.

clear_fig
[bool] Whether to clear the figure first before drawing.

save_plot(*filename: str*, ***kwargs*)
Save the image plot to file.

property variable_axis: Axis
The axis that is varying.

There are five types of images:

- Reference : All axes are at 0.
- Gantry: All axes but gantry at 0.
- Collimator : All axes but collimator at 0.
- Couch : All axes but couch at 0.
- Combo : More than one axis is not at 0.

results_data(*as_dict: bool = False*) → *WinstonLutz2DResult* | dict
Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

class pylinac.winston_lutz.**WinstonLutz2DResult**(*variable_axis: 'str'*, *cax2epid_vector: 'Vector'*, *cax2epid_distance: 'float'*, *cax2bb_distance: 'float'*, *cax2bb_vector: 'Vector'*, *bb_location: 'Point'*, *field_cax: 'Point'*)

Bases: *ResultBase*

variable_axis: str

cax2epid_vector: *Vector*

cax2epid_distance: float

cax2bb_distance: float

cax2bb_vector: *Vector*

bb_location: *Point*

field_cax: *Point*

6.14 Winston-Lutz Multi-Target

6.14.1 Overview

New in version 3.9.

The Multi-Target Winston-Lutz (MTWL) is an advanced test category meant to measure multiple locations away from isocenter, typically to represent multi-lesion SRS cases. The MTWL module can analyze images with any number of BBs in any arrangement. It is generalizable such that new phantom analyses can be created quickly.

Technically, there are two flavors of multi-target WL: multi-field and single field. An example of a multi-field WL is the SNC MultiMet. Each field is centered around each BB. The BB position is compared to that of the field. This is closest to what the patient experiences since it incorporates both the gantry/coll/couch deviations as well as the MLCs.

An example of a single-field multi-target WL is Machine Performance Check. The BBs are compared to the known positions. This removes the error of the MLCs to isolate just the gantry/coll/couch.

Currently, only the multi-field flavor is supported, but work on the single-field flavor will occur to support things like secondary checks of MPC.

This is why the class is called `WinstonLutzMultiTargetMultiField` as there will be an anticipated `WinstonLutzMultiTargetSingleField`.

6.14.2 Differences from single-target WL

Warning: The MTWL algorithm is new and provisional. There are a number of limitations with the algorithm. Hopefully, these are removed in future updates. The algorithm is still considered valuable even with these limitations which is why it is released.

Important: In a nutshell, the MTWL analyzes BB positions only, whereas vanilla WL provides more machine-related data as well as BB position data.

Unlike the single-target WL algorithm (aka "vanilla" WL), there are more limitations to acquisition and outputs. This should improve over time, but for now you can think of the MTWL as a subset of the vanilla WL algorithm:

- Utility methods such as loading images are the same.
- Outputs related to the BBs are different.

- BB size is not a parameter but is part of the BB arrangement.
- Single images cannot be analyzed.
- Axis deviations (Gantry wobble, etc) are not yet available.
- Couch rotation images are dropped as they cannot yet be handled.
- Interpreting filenames is not yet allowed.

See the following sections for more info.

- *Image Acquisition*
- *Supported Phantoms*

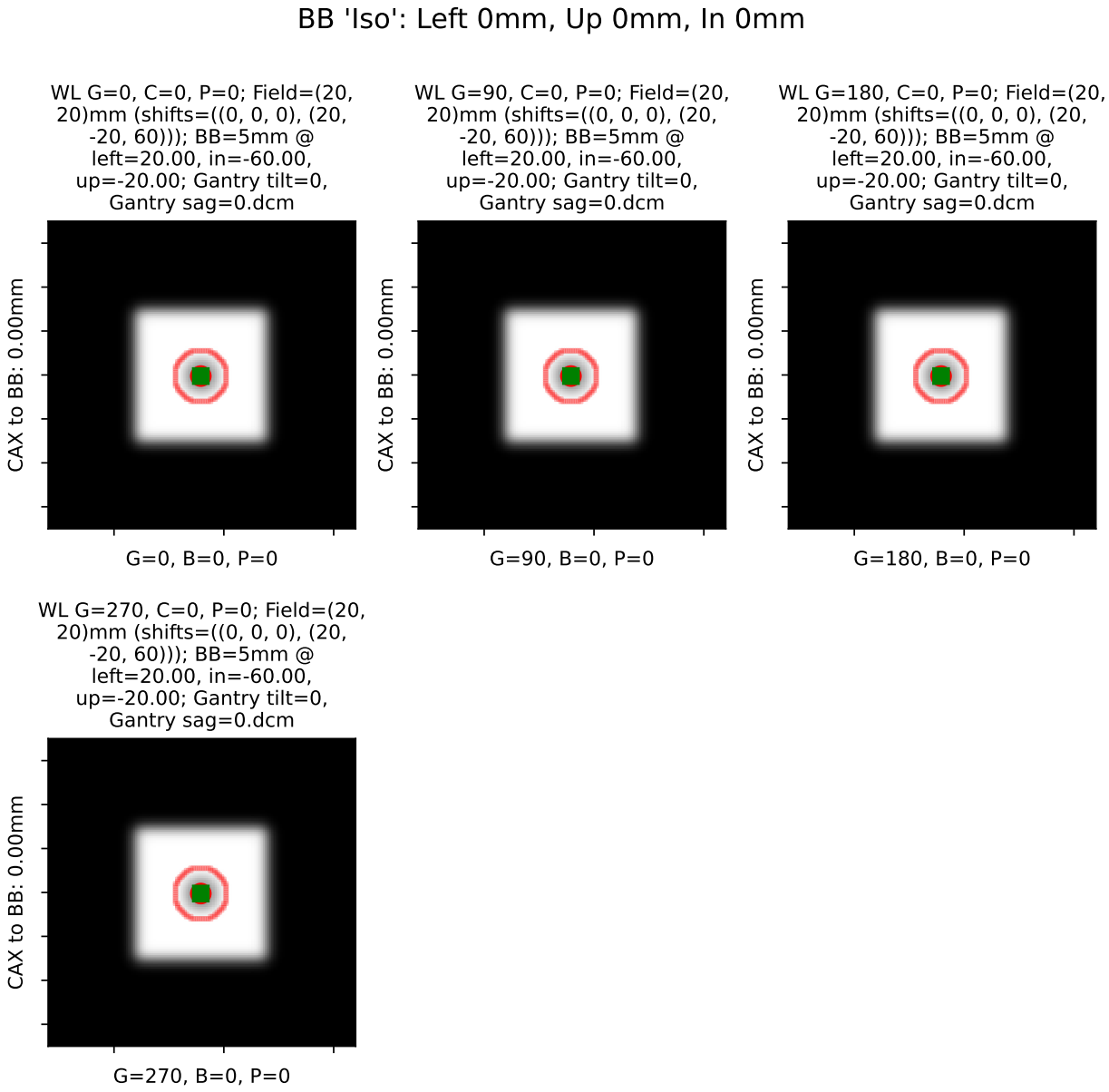
6.14.3 Running the Demo

To run the multi-target Winston-Lutz demo, create a script or start an interpreter session and input:

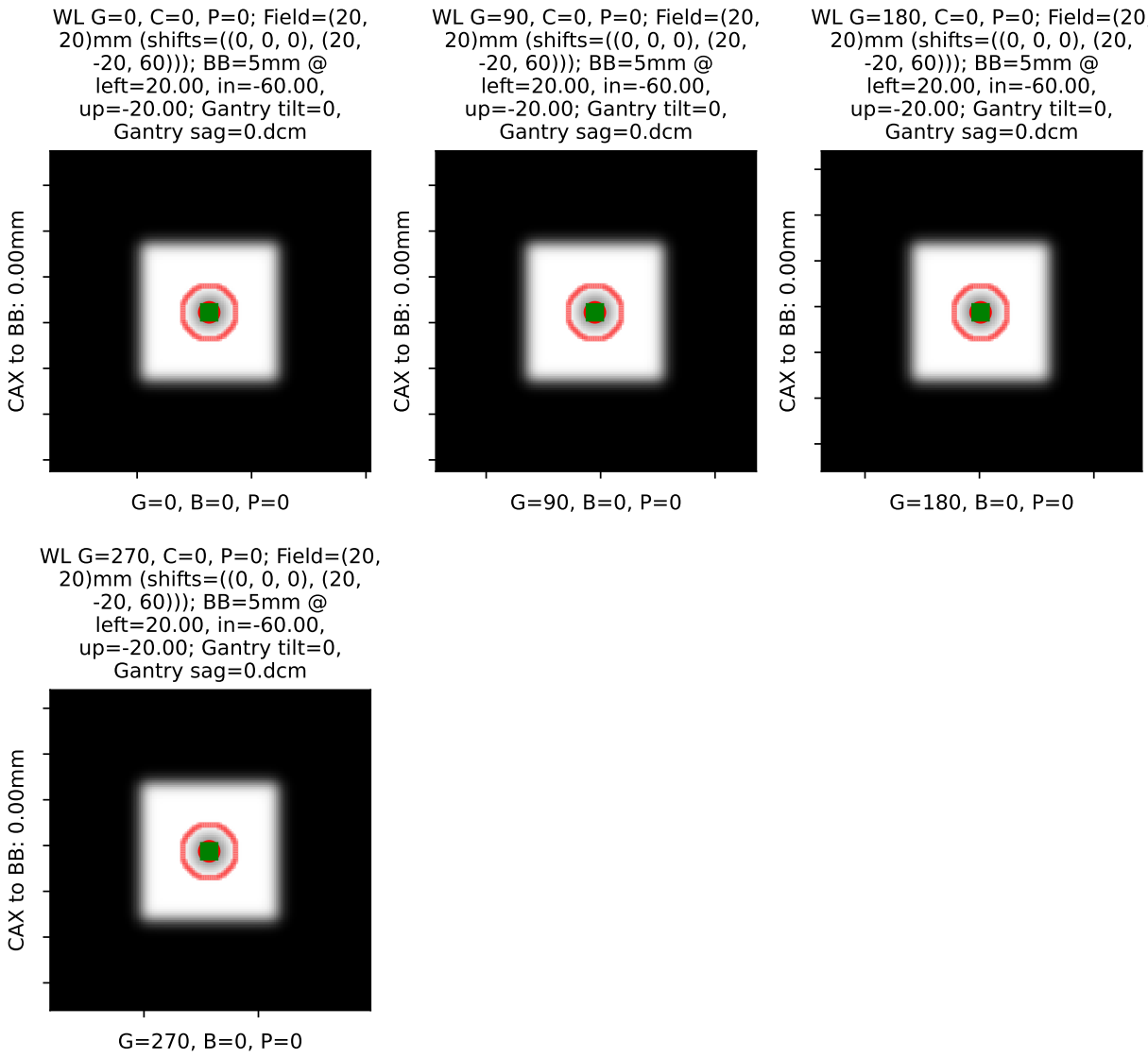
```
from pylinac import WinstonLutzMultiTargetMultiField

WinstonLutzMultiTargetMultiField.run_demo()
```

Results will be printed to the console and a figure showing the zoomed-in images will be generated:

```
Winston-Lutz Multi-Target Multi-Field Analysis
==============================================
Number of images: 4

2D distances
============
Max 2D distance of any BB: 0.00 mm
Mean 2D distance of any BB: 0.00 mm
Median 2D distance of any BB: 0.00 mm

  BB #  Description
------  -----------------------------------------
     0  'Iso': Left 0mm, Up 0mm, In 0mm
     1  'Left,Down,In': Left 20mm, Down 20mm, In 60mm

Image                    G    Co    Ch    BB #0    BB #1
-------------------    ---  ----  ----  -------  -------
=0, Gantry sag=0.dcm     0     0     0        0        0
=0, Gantry sag=0.dcm    90     0     0        0        0
=0, Gantry sag=0.dcm   180     0     0        0        0
=0, Gantry sag=0.dcm   270     0     0        0        0
```


BB 'Iso': Left 0mm, Up 0mm, In 0mm
WL G=0, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=0, B=0, P=0
WL G=90, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=90, B=0, P=0
WL G=180, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=180, B=0, P=0
WL G=270, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=270, B=0, P=0

BB 'Left,Down,In': Left 20mm, Down 20mm, In 60mm



6.14.4 Image Acquisition

The Winston-Lutz module will only load EPID images. The images can be from any EPID however and any SID. To ensure the most accurate results the following should be noted:

- Images with a rotated couch are dropped and not analyzed (yet) but will not cause an error.
- The BBs should not occlude each other.
- The BBs should be >5mm apart in any given image.
- The radiation fields should have >5mm separation in any given image.
- The BB and radiation field should be <=5 mm away from the nominal location given by the arrangement.

6.14.5 Coordinate Space

The MTWL algorithm uses the same coordinate system as the vanilla WL. *Coordinate Space*.

Passing a coordinate system

No coordinate system is passed or used (yet).

Note: This is a target for the MTWL algorithm, so expect this to change in the future.

6.14.6 Supported Phantoms

Currently, only the MultiMet-WL cube from SNC is supported. However, the algorithm is generalized and can be easily adapted to analyze other phantoms. See *Custom BB Arrangements*.

6.14.7 Typical Use

Analyzing a multi-target Winston-Lutz test is simple. First, let's import the class:

```
from pylinac import WinstonLutzMultiTargetMultiField
from pylinac.winston_lutz import BBArrangement
```

From here, you can load a directory:

```
my_directory = "path/to/wl_images"
wl = WinstonLutzMultiTargetMultiField(my_directory)
```

You can also load a ZIP archive with the images in it:

```
wl = WinstonLutzMultiTargetMultiField.from_zip("path/to/wl.zip")
```

Now, analyze it. Unlike the vanilla WL algorithm, we have to pass the BB arrangement to know where the BBs should be in space. *Preset phantoms* exist, or a custom arrangement can be passed.

```
wl.analyze(bb_arrangement=BBArrangement.SNC_MULTIMET)
```

And that's it! You can now view images, print the results, or publish a PDF report:

```
# plot all the images
wl.plot_images()
# save figures of the image plots for each bb
wl.save_images(prefix="snc")
# print to PDF
wl.publish_pdf("mymtwl.pdf")
```

6.14.8 Changing BB detection size

To change the size of BB pylinac is expecting you must change it in the BB arrangement. This allows phantoms with multiple BB sizes to still be analyzed. See *Custom BB Arrangements*

6.14.9 Custom BB Arrangements

The MTWL algorithm uses a priori BB arrangements. I.e. you need to know where the BBs **should** exist in space relative to isocenter. The MTWL algorithm is flexible to accommodate any reasonable arrangement of BBs.

To create a custom arrangement, say for an in-house phantom or commercial phantom not yet supported, define the BB offsets and size like so. Use negative values to move the other direction:

```
my_special_phantom_bbs = [
    {
        "offset_left_mm": 0,
        "offset_up_mm": 0,
        "offset_in_mm": 0,
        "bb_size_mm": 5,
        "rad_size_mm": 20,
    },  # 5mm BB at iso
    {
        "offset_left_mm": 30,
        "offset_up_mm": 0,
        "offset_in_mm": 0,
        "bb_size_mm": 4,
        "rad_size_mm": 20,
    },  # 4mm BB 30mm to left of iso
    {
        "offset_left_mm": 0,
        "offset_up_mm": -20,
        "offset_in_mm": 10,
        "bb_size_mm": 5,
        "rad_size_mm": 20,
    },  # BB DOWN 20mm and in 10mm
    ...,  # keep going as needed
]
```

Pass it to the algorithm like so:

```
wl = WinstonLutzMultiTargetMultiField(...)
wl.analyze(bb_arrangement=my_special_phantom_bbs)
...
```

6.14.10 Algorithm

The MTWL algorithm is based on the vanilla WL algorithm. For each BB and image combination, the image is searched at the nominal location for the BB and radiation field. If it's not found it will be skipped for that combo. The BB must be detected in at least one image or an error will be raised.

The algorithm works like such:

Allowances

- The images can be acquired with any EPID (aS500, aS1000, aS1200) at any SID.
- The image can have any number of BBs.
- The BBs can be at any 3D location.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- Each BB and radiation field must be within 5mm of the expected position in x and y in the EPID plane. I.e. it must be <=7mm in scalar distance.
- BBs must not occlude or be <5 mm from each other in any 2D image.
- Images with a rotated couch are dropped and not analyzed (yet) but will not cause an error.
- The radiation fields should have >5mm separation in any given image.

Analysis

This algorithm is performed for each BB and image combination:

- **Find the field center** – The spread in pixel values (max - min) is divided by 2, and any pixels above the threshold is associated with the open field. The pixels are converted to black & white and the center of mass of the pixels is assumed to be the field center.
- **Find the BB** – The image is converted to binary based on pixel values *both* above the 50% threshold as above, and below the upper threshold. The upper threshold is an iterative value, starting at the image maximum value, that is lowered slightly when the BB is not found. If the binary image has a reasonably circular ROI, is approximately the right size, and is within 5mm of the expected BB position, the BB is considered found and the pixel-weighted center of mass of the BB is considered the BB location.
- **Evaluate against the field position** – Once the measured BB and field positions are known, both the scalar distance and vector from the field position to the measured BB position is determined.

6.14.11 Benchmarking the Algorithm

With the image generator module we can create test images to test the WL algorithm on known results. This is useful to isolate what is or isn't working if the algorithm doesn't work on a given image and when commissioning pylinac. It is common, especially with the WL module, to question the accuracy of the algorithm. Since no linac is perfect and the results are sub-millimeter, discerning what is true error vs algorithmic error can be difficult. The image generator module is a perfect solution since it can remove or reproduce the former error.

Note: With the introduction of the MTWL algorithm, so to a multi-target synthetic image generator has been created: *generate_winstonlutz_multi_bb_multi_field()*.

Warning: The image generator is limited in accuracy to ~1/2 pixel because creating the image requires a row or column to be set. E.g. a 5mm field with a 0.336mm pixel size means we need to create a field of 14.88 pixels wide. We can only set the field to be 14 or 15 pixels, so the nearest field size of 15 pixels or 5.04mm is set.

2-BB Perfect Delivery

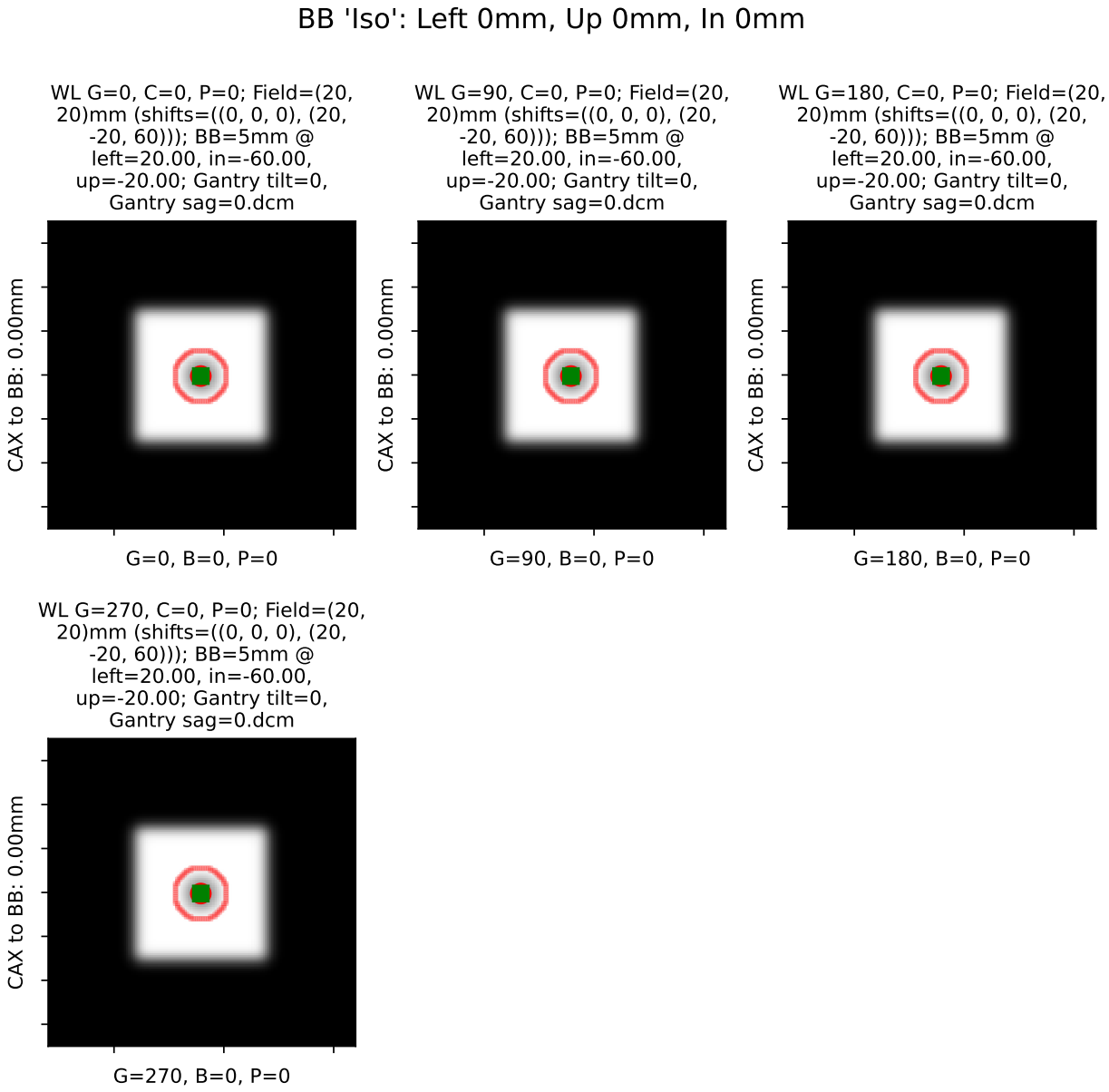
Create a perfect set of fields with 1 BB at iso and another 20mm left, 20mm down, and 60mm inward (this is the same as the demo, but is good for explanation).

```
import pylinac
from pylinac.core.image_generator import simulators, layers, generate_winstonlutz_multi_
↪bb_multi_field

wl_dir = 'wl_dir'
generate_winstonlutz_multi_bb_multi_field(
        simulator=simulators.AS1200Image(sid=1000),
        field_layer=layers.PerfectFieldLayer,
        final_layers=[layers.GaussianFilterLayer(sigma_mm=1),],
        dir_out=wl_dir,
        field_offsets=((0, 0, 0), (20, -20, 60)),
        field_size_mm=(20, 20),
        bb_offsets=[[0, 0, 0], [20, -20, 60]],
)
arrange = (
    {'name': 'Iso', 'offset_left_mm': 0, 'offset_up_mm': 0, 'offset_in_mm': 0, 'bb_size_
↪mm': 5, 'rad_size_mm': 20},
    {'name': 'Left,Down,In', 'offset_left_mm': 20, 'offset_up_mm': -20, 'offset_in_mm':␣
↪60, 'bb_size_mm': 5, 'rad_size_mm': 20},)

wl = pylinac.WinstonLutzMultiTargetMultiField(wl_dir)
wl.analyze(bb_arrangement=arrange)
print(wl.results())
wl.plot_images()
```

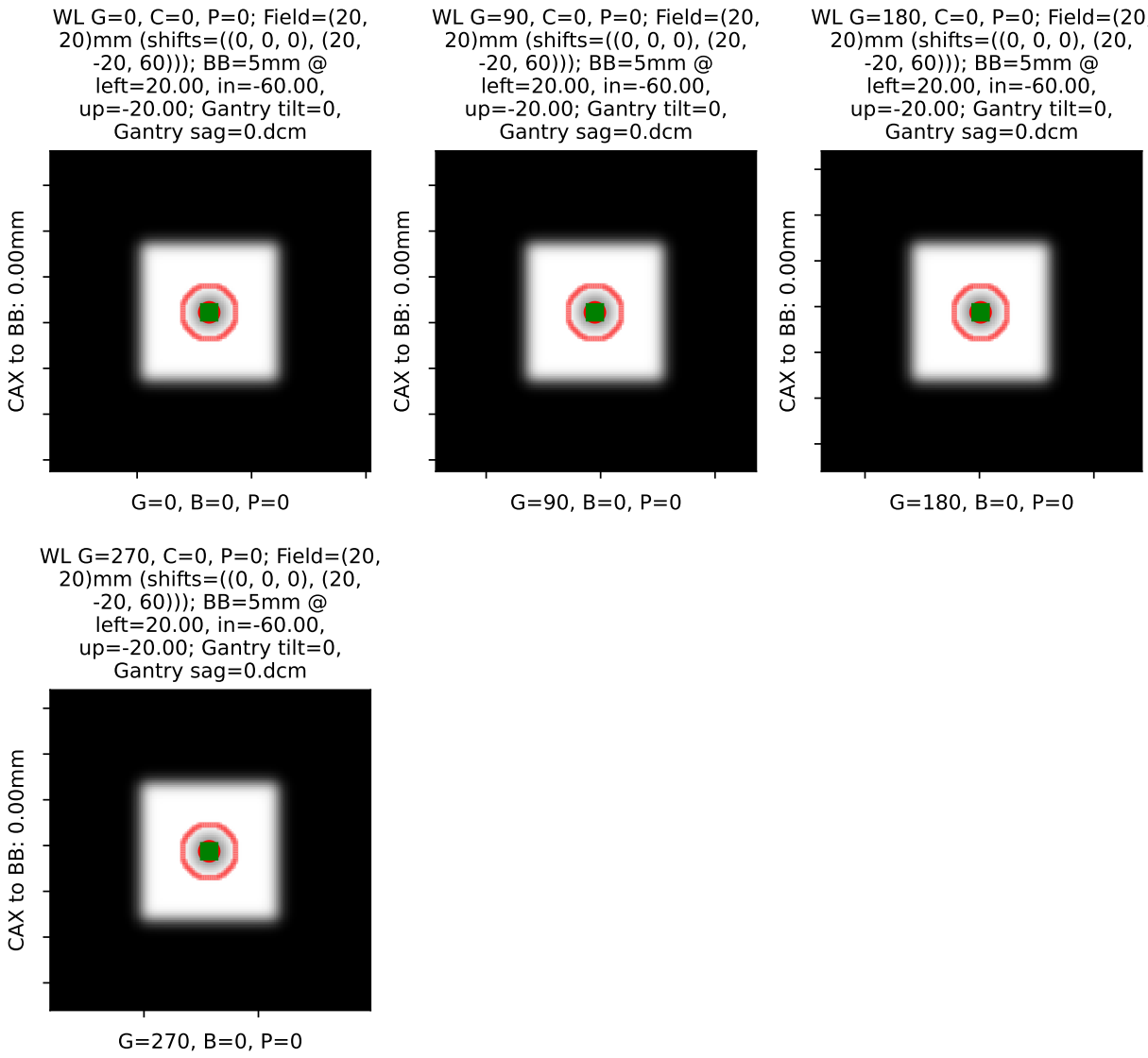
which has an output of:

```
Winston-Lutz Multi-Target Multi-Field Analysis
==============================================
Number of images: 4

2D distances
============
Max 2D distance of any BB: 0.00 mm
Mean 2D distance of any BB: 0.00 mm
Median 2D distance of any BB: 0.00 mm

  BB #  Description
------  -------------------------------------------
     0  'Iso': Left 0mm, Up 0mm, In 0mm
     1  'Left,Down,In': Left 20mm, Down 20mm, In 60mm
```

(continues on next page)


BB 'Iso': Left 0mm, Up 0mm, In 0mm
WL G=0, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=0, B=0, P=0
WL G=90, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=90, B=0, P=0
WL G=180, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=180, B=0, P=0
WL G=270, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=20.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=270, B=0, P=0

BB 'Left,Down,In': Left 20mm, Down 20mm, In 60mm



(continued from previous page)

```
Image                  G    Co    Ch    BB #0    BB #1
--------------------  ---  ----  ----  -------  -------
=0, Gantry sag=0.dcm    0     0     0        0        0
=0, Gantry sag=0.dcm   90     0     0        0        0
=0, Gantry sag=0.dcm  180     0     0        0        0
=0, Gantry sag=0.dcm  270     0     0        0        0
```

As shown, we have perfect results.

Offset BBs

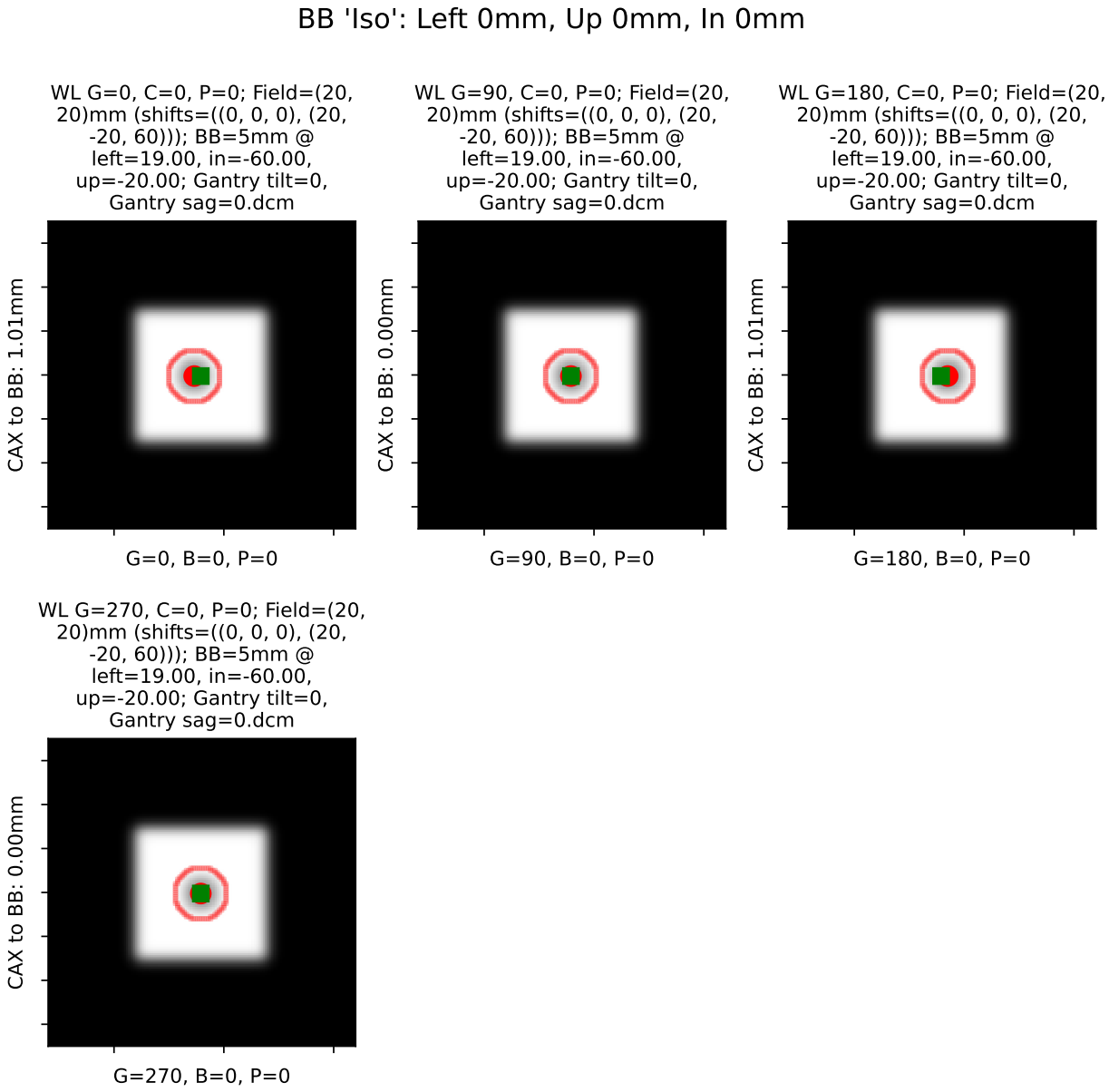
Let's now offset both BBs by 1mm to the left:

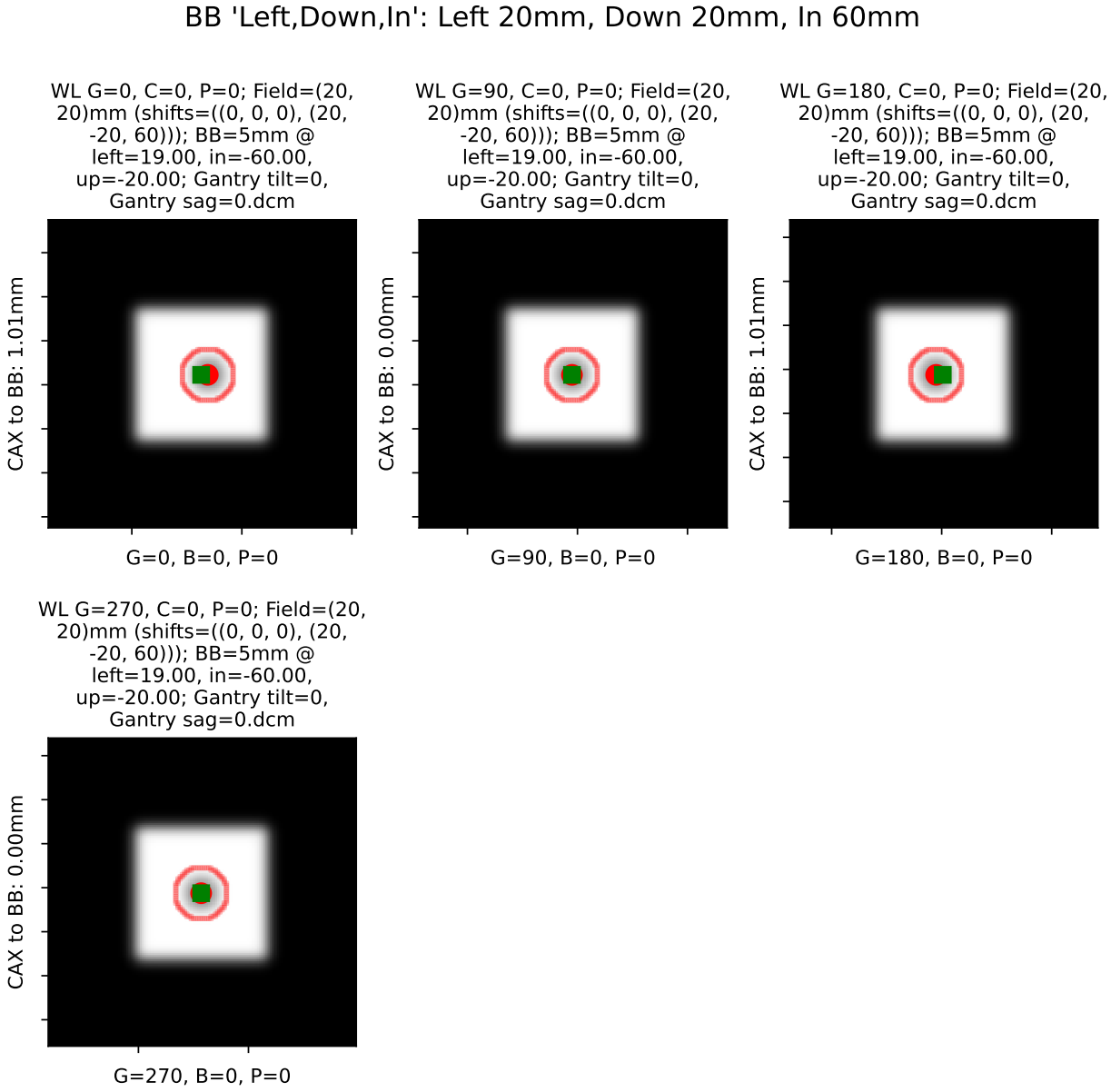
```
import pylinac
from pylinac.core.image_generator import simulators, layers, generate_winstonlutz_multi_
↪bb_multi_field

wl_dir = 'wl_dir'
generate_winstonlutz_multi_bb_multi_field(
        simulator=simulators.AS1200Image(sid=1000),
        field_layer=layers.PerfectFieldLayer,
        final_layers=[layers.GaussianFilterLayer(sigma_mm=1),],
        dir_out=wl_dir,
        field_offsets=((0, 0, 0), (20, -20, 60)),
        field_size_mm=(20, 20),
        bb_offsets=[[1, 0, 0], [19, -20, 60]],  # here's the offset
)
arrange = (
    {'name': 'Iso', 'offset_left_mm': 0, 'offset_up_mm': 0, 'offset_in_mm': 0, 'bb_size_
↪mm': 5, 'rad_size_mm': 20},
    {'name': 'Left,Down,In', 'offset_left_mm': 20, 'offset_up_mm': -20, 'offset_in_mm':␣
↪60, 'bb_size_mm': 5, 'rad_size_mm': 20},)

wl = pylinac.WinstonLutzMultiTargetMultiField(wl_dir)
wl.analyze(bb_arrangement=arrange)
print(wl.results())
wl.plot_images()
```

with an output of:

```
Winston-Lutz Multi-Target Multi-Field Analysis
==============================================
Number of images: 4

2D distances
============
Max 2D distance of any BB: 1.01 mm
Mean 2D distance of any BB: 1.01 mm
Median 2D distance of any BB: 1.01 mm

  BB #  Description
------  -------------------------------------------
```

(continues on next page)


BB 'Iso': Left 0mm, Up 0mm, In 0mm
WL G=0, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 1.01mm
G=0, B=0, P=0
WL G=90, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=90, B=0, P=0
WL G=180, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 1.01mm
G=180, B=0, P=0
WL G=270, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=270, B=0, P=0


BB 'Left,Down,In': Left 20mm, Down 20mm, In 60mm
WL G=0, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 1.01mm
G=0, B=0, P=0
WL G=90, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=90, B=0, P=0
WL G=180, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 1.01mm
G=180, B=0, P=0
WL G=270, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.00, in=-60.00, up=-20.00; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 0.00mm
G=270, B=0, P=0

(continued from previous page)

```
    0  'Iso': Left 0mm, Up 0mm, In 0mm
    1  'Left,Down,In': Left 20mm, Down 20mm, In 60mm

Image                   G    Co    Ch    BB #0    BB #1
--------------------  ---  ----  ----  -------  -------
=0, Gantry sag=0.dcm    0     0     0     1.01     1.01
=0, Gantry sag=0.dcm   90     0     0     0        0
=0, Gantry sag=0.dcm  180     0     0     1.01     1.01
=0, Gantry sag=0.dcm  270     0     0     0        0
```

Both BBs report a shift of 1mm. Note this is only in 0 and 180. A left shift would not be captured at 90/270.

Random error

Let's now add random error:

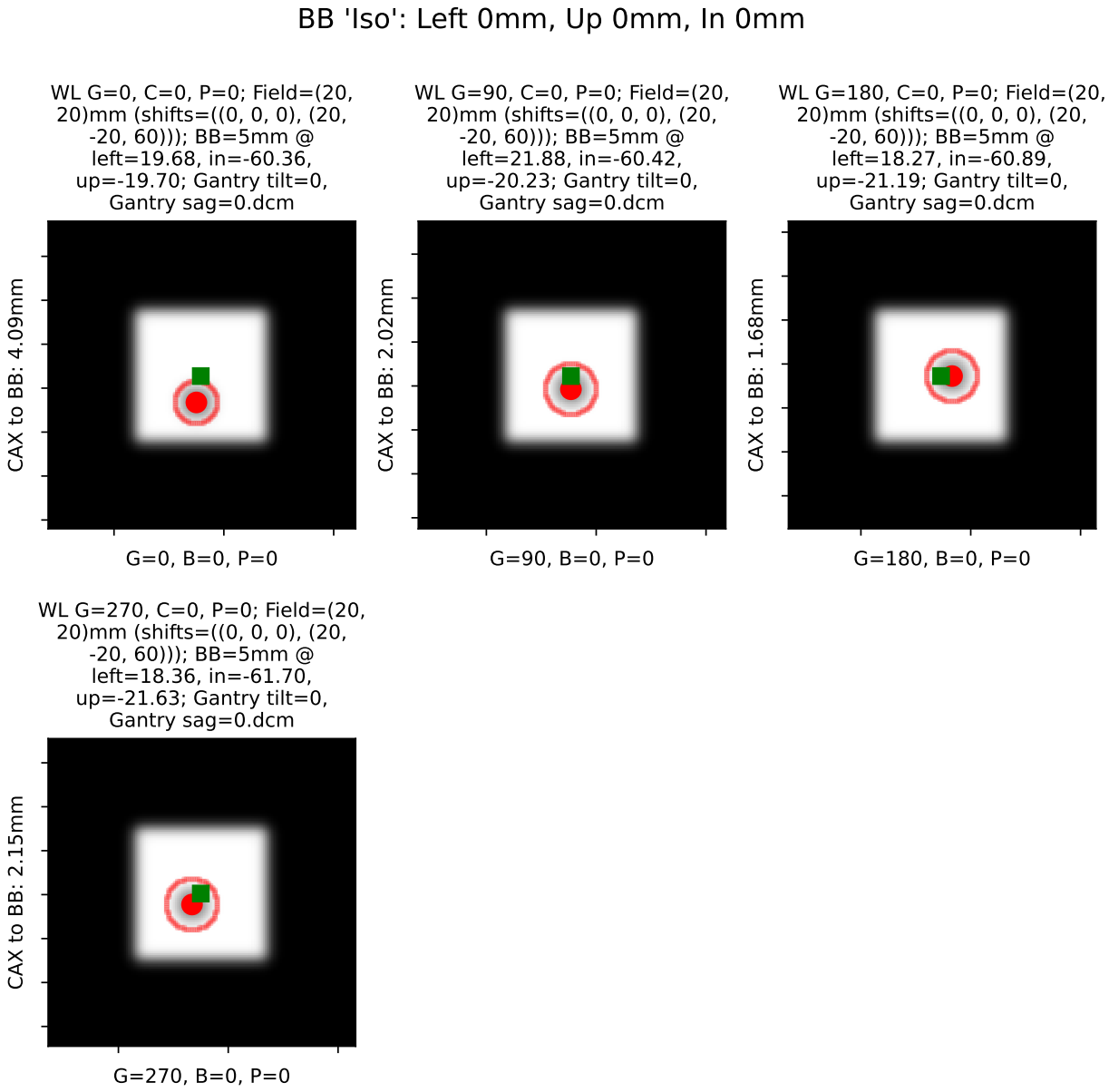
Note: The error is random so performing this again will change the results slightly.

```
import pylinac
from pylinac.core.image_generator import simulators, layers, generate_winstonlutz_multi_
↪bb_multi_field

wl_dir = 'wl_dir'
generate_winstonlutz_multi_bb_multi_field(
        simulator=simulators.AS1200Image(sid=1000),
        field_layer=layers.PerfectFieldLayer,
        final_layers=[layers.GaussianFilterLayer(sigma_mm=1),],
        dir_out=wl_dir,
        field_offsets=((0, 0, 0), (20, -20, 60)),
        field_size_mm=(20, 20),
        bb_offsets=[[0, 0, 0], [20, -20, 60]],
        jitter_mm=2  # here we add random noise
)
arrange = (
    {'name': 'Iso', 'offset_left_mm': 0, 'offset_up_mm': 0, 'offset_in_mm': 0, 'bb_size_
↪mm': 5, 'rad_size_mm': 20},
    {'name': 'Left,Down,In', 'offset_left_mm': 20, 'offset_up_mm': -20, 'offset_in_mm':␣
↪60, 'bb_size_mm': 5, 'rad_size_mm': 20},)

wl = pylinac.WinstonLutzMultiTargetMultiField(wl_dir)
wl.analyze(bb_arrangement=arrange)
print(wl.results())
wl.plot_images()
```

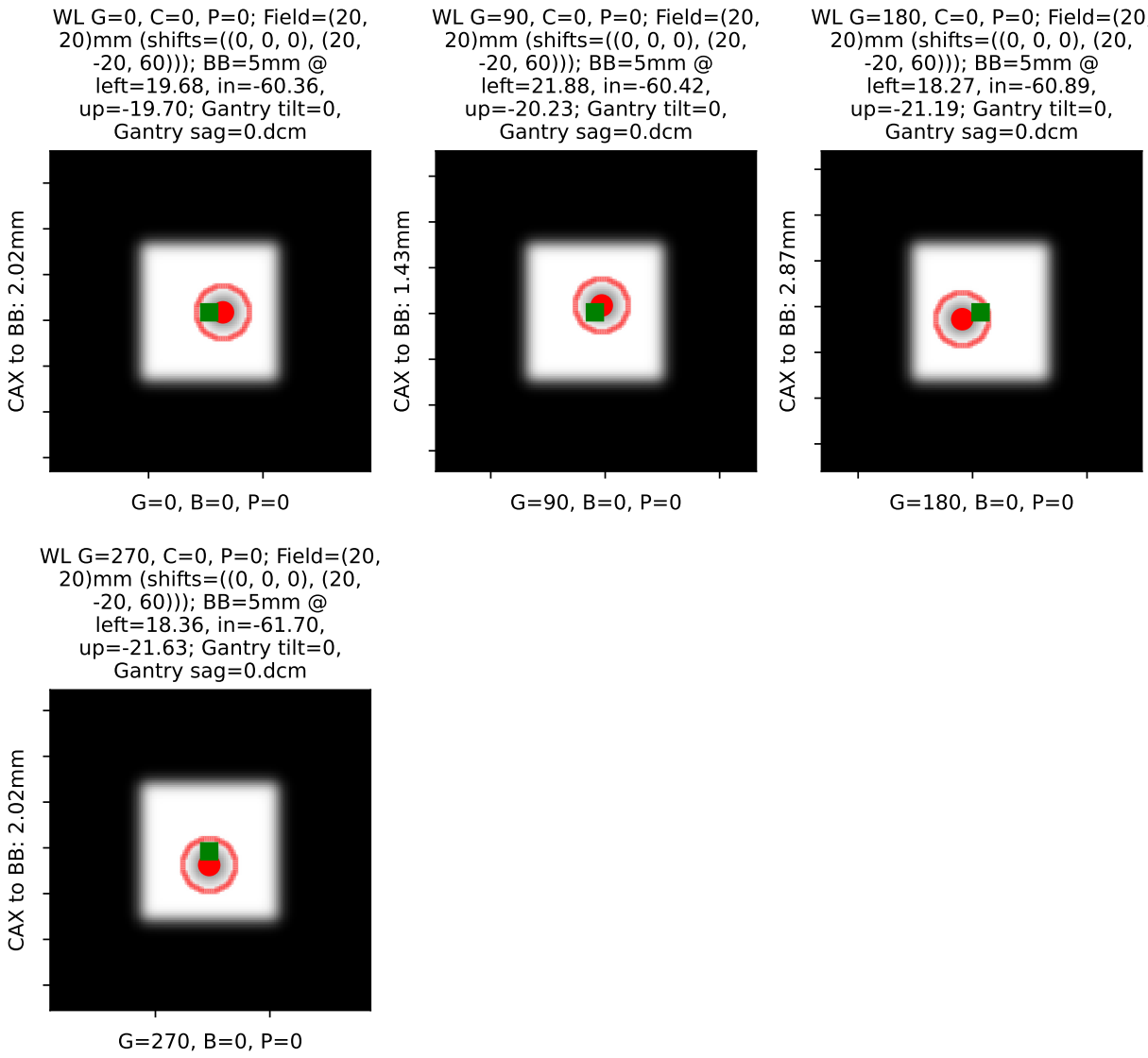
with an output of:

```
Winston-Lutz Multi-Target Multi-Field Analysis
==============================================
Number of images: 4
```

(continues on next page)


BB 'Iso': Left 0mm, Up 0mm, In 0mm
WL G=0, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=19.68, in=-60.36, up=-19.70; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 4.09mm
G=0, B=0, P=0
WL G=90, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=21.88, in=-60.42, up=-20.23; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 2.02mm
G=90, B=0, P=0
WL G=180, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=18.27, in=-60.89, up=-21.19; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 1.68mm
G=180, B=0, P=0
WL G=270, C=0, P=0; Field=(20, 20)mm (shifts=((0, 0, 0), (20, -20, 60))); BB=5mm @ left=18.36, in=-61.70, up=-21.63; Gantry tilt=0, Gantry sag=0.dcm
CAX to BB: 2.15mm
G=270, B=0, P=0

BB 'Left,Down,In': Left 20mm, Down 20mm, In 60mm



(continued from previous page)

```
2D distances
============
Max 2D distance of any BB: 3.38 mm
Mean 2D distance of any BB: 2.82 mm
Median 2D distance of any BB: 2.82 mm

  BB #  Description
------  ------------------------------------------
     0  'Iso': Left 0mm, Up 0mm, In 0mm
     1  'Left,Down,In': Left 20mm, Down 20mm, In 60mm

Image                    G    Co    Ch    BB #0    BB #1
-------------------   ---  ----  ----  -------  -------
=0, Gantry sag=0.dcm    0     0     0     2.25     0.34
=0, Gantry sag=0.dcm   90     0     0     2.15     2.77
=0, Gantry sag=0.dcm  180     0     0     2.13     3.38
=0, Gantry sag=0.dcm  270     0     0     2.25     2.42
```

6.14.12 API Documentation

class pylinac.winston_lutz.**WinstonLutzMultiTargetMultiField**(**args*, ***kwargs*)

Bases: *WinstonLutz*

We cannot yet handle non-0 couch angles so we drop them. Analysis fails otherwise

analyzed_images: dict[str, list[*WinstonLutz2DMultiTarget*]]

image_type

alias of *WinstonLutz2DMultiTarget*

bb_arrangement: Sequence[dict]

images: Sequence[*WinstonLutz2DMultiTarget*]

classmethod from_demo_images()

Instantiate using the demo images.

static run_demo()

Run the Winston-Lutz MT MF demo, which loads the demo files, prints results, and plots a summary image.

analyze(*bb_arrangement: Sequence[dict]*)

Analyze the WL images.

Parameters

bb_arrangement

The arrangement of the BBs in the phantom. A dict with offset and BB size keys. See the `BBArrangement` class for keys and syntax.

plot_images(*show: bool = True*, ***kwargs*)

Make a plot for each BB. Each plot contains the analysis of that BB on each image it was found.

save_images(*prefix: str = ''*, ***kwargs*)

Save the figure of *plot_images()* to file as PNG. Keyword arguments are passed to *matplotlib.pyplot.savefig()*.

Parameters

prefix
[str] The prefix name of the file to save to. The BB name is appended to the prefix.

save_images_to_stream(***kwargs*) → dict[str, BytesIO]

Save the individual image plots to stream

cax2bb_distance(*bb: str*, *metric: str = 'max'*) → float

The distance in mm between the CAX and BB for all images according to the given metric.

Parameters

metric
[{'max', 'median', 'mean'}] The metric of distance to use.

bb
[str] The BB to analyze

results_data(*as_dict: bool = False*) → *WinstonLutzMultiTargetMultiFieldResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

plot_summary(*show: bool = True*, *fig_size: tuple | None = None*)

Plot a summary figure showing the gantry sag and wobble plots of the three axes.

plot_axis_images(*axis:* Axis *= Axis.GANTRY*, *show: bool = True*, *ax: plt.Axes | None = None*)

Plot all CAX/BB/EPID positions for the images of a given axis.

For example, axis='Couch' plots a reference image, and all the BB points of the other images where the couch was moving.

Parameters

axis
[{'Gantry', 'Collimator', 'Couch', 'GB Combo', 'GBP Combo'}] The images/markers from which accelerator axis to plot.

show
[bool] Whether to actually show the images.

ax
[None, matplotlib.Axes] The axis to plot to. If None, creates a new plot.

property max_bb_deviation_2d: float

The maximum distance from any measured BB to its nominal position

property mean_bb_deviation_2d: float

The mean distance from any measured BB to its nominal position

property median_bb_deviation_2d: float

The median distance from any measured BB to its nominal position

results(*as_list: bool = False*) → str

Return the analysis results summary.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

publish_pdf(*filename: str*, *notes: str | list[str] | None = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object)] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

class pylinac.winston_lutz.**WinstonLutz2DMultiTarget**(**args*, ***kwargs*)

Bases: *WinstonLutz2D*

A 2D image of a WL delivery, but where multiple BBs are in use.

Parameters

file
[str] Path to the image file.

use_filenames: bool
Whether to try to use the file name to determine axis values. Useful for Elekta machines that do not include that info in the DICOM data.

as_analyzed(*bb_location: dict*) → *WinstonLutz2DMultiTarget*

Analyze the image of the multi-BB setup. We return a copy of the WL image because we analyze images more than once. Each "analyzed" image is really the analysis of a BB/image combo.

Parameters

bb_location
An iterable of dictionaries. Each dict contains keys for the offsets and size of the BB in mm. Use the `BBArrangement` class as a guide.

plot(*ax: plt.Axes | None = None*, *show: bool = True*, *clear_fig: bool = False*) → plt.Axes

Plot the image, zoomed-in on the radiation field, along with the detected BB location and field CAX location.

Parameters

ax
[None, matplotlib Axes instance] The axis to plot to. If None, will create a new figure.

show
[bool] Whether to actually show the image.

clear_fig
[bool] Whether to clear the figure first before drawing.

location_near_nominal(*region: RegionProperties*, *location: dict*) → bool

Determine whether the given BB ROI is near where the BB is expected to be

results_data(*as_dict: bool = False*) → *WinstonLutz2DResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

class pylinac.winston_lutz.**WinstonLutzMultiTargetMultiFieldResult**(*num_total_images: int*, *max_2d_field_to_bb_mm: float*, *median_2d_field_to_bb_mm: float*, *mean_2d_field_to_bb_mm: float*, *bb_arrangement: Iterable[dict]*, *bb_maxes: dict*)

Bases: *ResultBase*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

num_total_images: int

max_2d_field_to_bb_mm: float

median_2d_field_to_bb_mm: float

mean_2d_field_to_bb_mm: float

bb_arrangement: Iterable[dict]

bb_maxes: dict

6.15 Planar Imaging

6.15.1 Overview

The planar imaging module analyzes phantom images taken with the kV or MV imager in 2D. The following phantoms are supported:

- Leeds TOR 18
- Standard Imaging QC-3
- Standard Imaging QC-kV
- Las Vegas
- Elekta Las Vegas
- Doselab MC2 MV
- Doselab MC2 kV
- SNC kV
- SNC MV
- PTW EPID QC

Features:

- **Automatic phantom localization** - Set up your phantom any way you like; automatic positioning, angle, and inversion correction mean you can set up how you like, nor will setup variations give you headache.
- **High and low contrast determination** - Analyze both low and high contrast ROIs. Set thresholds as you see fit.

6.15.2 Feature table

Feature/Phantom	Can be inverted?	SSD setting	Auto-centering	Auto-rotation
Doselab MC2 (MV)	No	Manual	Yes	Semi (+/-5 from 0)
Doselab MC2 (kV)	No	Manual	Yes	Semi (+/-5 from 0)
Las Vegas	No	Manual	Yes	No (0)
Elekta Las Vegas	No	Manual	Yes	No (0)
Leeds TOR	Yes	Manual	Yes	Yes
PTW EPID QC	No	Manual	Yes	No (0)
SNC MV	No	Manual	Yes	No (45)
SNC MV (12510)	No	Manual	Yes	No (45)
SNC kV	No	Manual	Yes	No (135)
SI QC-3 (MV)	No	Manual	Yes	Semi (+/-5 from 45/135)
SI QC kV	No	Manual	Yes	Semi (+/-5 from 45/135)
IBA Primus A	No	Manual	Yes (+/-2cm)	Semi (+/-5 from 0,90,270)

6.15.3 Typical module usage

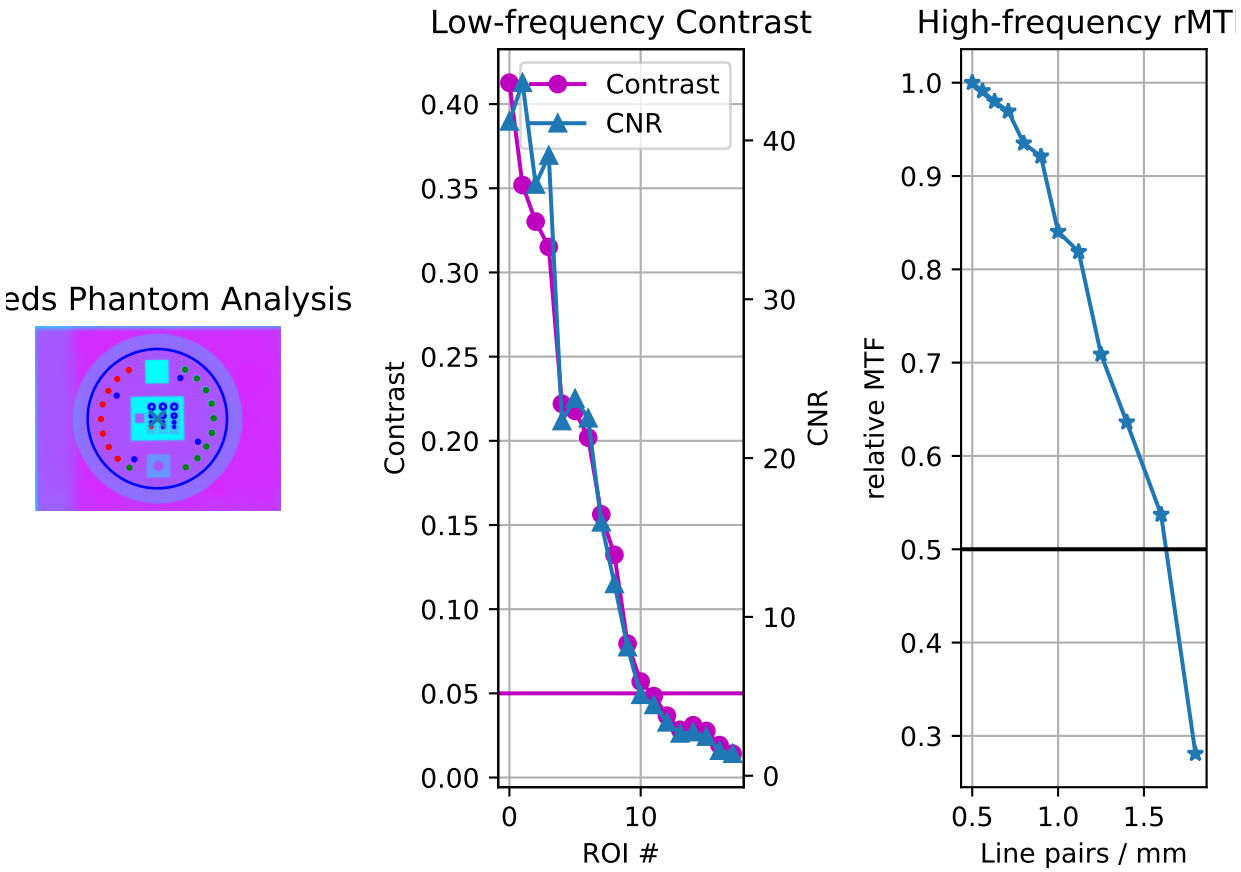
The following snippets can be used with any of the phantoms in this module; they all have the same or very similar methods. We will use the LeedsTOR for the example, but plug in any phantom from this module.

Running the Demo

To run the demo of any phantom, create a script or start an interpreter session and input:

```
from pylinac import LeedsTOR  # or LasVegas, DoselabMC2kV, etc

LeedsTOR.run_demo()
```

A figure showing the phantom, low contrast plot, and RMTF will be generated:



Typical Use

Import the class:

```
from pylinac import (
    LeedsTOR,
)  # or whatever phantom you like from the planar imaging module
```

The minimum needed to get going is to:

- **Load image** – Load the planar image as you would any other class: by passing the path directly to the constructor:

  ```
  leeds = LeedsTOR("my/leeds.dcm")
  ```

  Alternatively, a URL can be passed:

  ```
  leeds = LeedsTOR.from_url("http://myserver.com/leeds")
  ```

  You may also use the demo image:

  ```
  leeds = LeedsTOR.from_demo_image()
  ```

- **Analyze the images** – Analyze the image using the *analyze()* method. The low and high contrast thresholds can be specified:

  ```
  leeds.analyze(low_contrast_threshold=0.01, high_contrast_threshold=0.5)
  ```

  Additionally, you may specify the SSD of the phantom. By default, SAD and 5cm up from SID are searched:

  ```
  leeds.analyze(..., ssd=1400)
  ```

- **View the results** – The results of analysis can be viewed with the *plot_analyzed_image()* method.

  ```
  leeds.plot_analyzed_image()
  ```

  Note that each subimage can be turned on or off.

  ```
  # don't show the low contrast plot
  leeds.plot_analyzed_image(low_contrast=False)
  ```

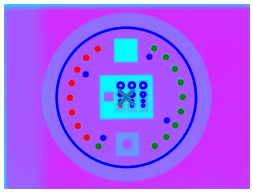
  The figure can also be saved:

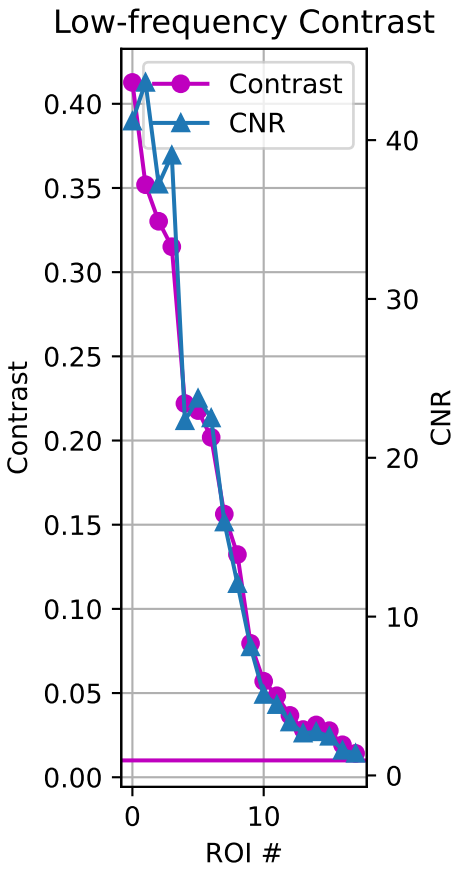
  ```
  leeds.save_analyzed_image("myprofile.png")
  ```

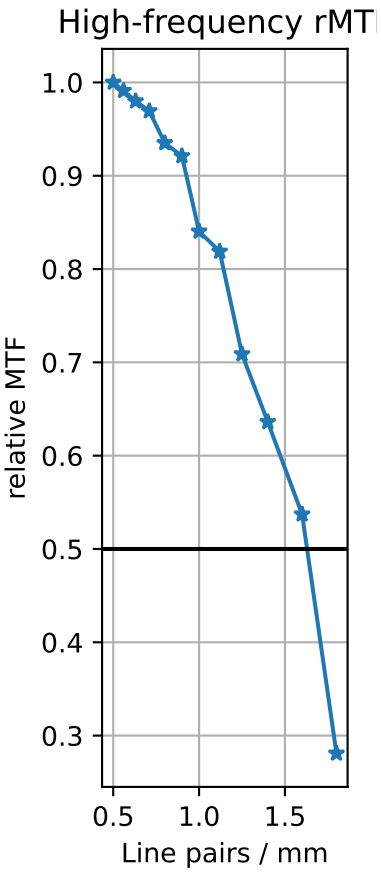
  A PDF report can also be generated:

  ```
  leeds.publish_pdf("leeds_october16.pdf")
  ```

eds Phantom Analysis







6.15.4 Leeds TOR Phantom

The Leeds phantom is used to measure image quality metrics for the kV imager of a linac. It contains both high and low contrast ROIs.

Note: There are two phantom analysis routines. The *LeedsTOR* class is for newer phantoms that have a red ring on the outside. Older Leeds phantoms may have a blue label containing the serial number and model on the back. Use the *LeedsTORBlue* class for these phantoms. The difference is small ROI location shifts.

Image Acquisition

You can acquire the images any way you like. Just ensure that the phantom is not touching a field edge. It is also recommended by the manufacturer to rotate the phantom to a non-cardinal angle so that pixel aliasing does not occur for the high-contrast line pairs.

Algorithm

Leeds phantom analysis is straightforward: find the phantom in the image, then sample ROIs at the appropriate locations.

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any size kV imager.
- The phantom can be at any distance.
- The phantom can be at any angle.
- The phantom can be flipped either way.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must not be touching or close to any image edges.
- The blades should be fully or mostly open to correctly invert the image. This may not result in a complete failure, but you may have to force-invert the analysis if this case isn't true (i.e. `myleeds.analyze(invert=True)`).
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – The Leeds phantom is found by performing a Canny edge detection algorithm to the image. The thin structures found are sifted by finding appropriately-sized ROIs. This may include the outer phantom edge and the metal ring just inside. The average central position of the circular ROIs is set as the phantom center.
- **Determine phantom angle** – To find the rotational angle of the phantom, a similar process is employed, but square-like features are searched for in the edge detection image. Because there are two square areas, the ROI with the highest attenuation (lead) is chosen. The angle between the phantom center and the lead square center is set as the angle.

- **Determine rotation direction** – The phantom might be placed upside down. To keep analysis consistent, a circular profile is sampled at the radius of the low contrast ROIs starting at the lead square. Peaks are searched for on each semicircle. The side with the most peaks is the side with the higher contrast ROIs. Analysis is always done counter-clockwise. If the ROIs happen to be clockwise, the image is flipped left-right and angle/center inverted.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.
- **Calculate high contrast** – Again, because the phantom position and angle are known, offsets are applied to sample the high contrast line pair regions. For each sample, the relative MTF is calculated. See *Modulation Transfer Function*.

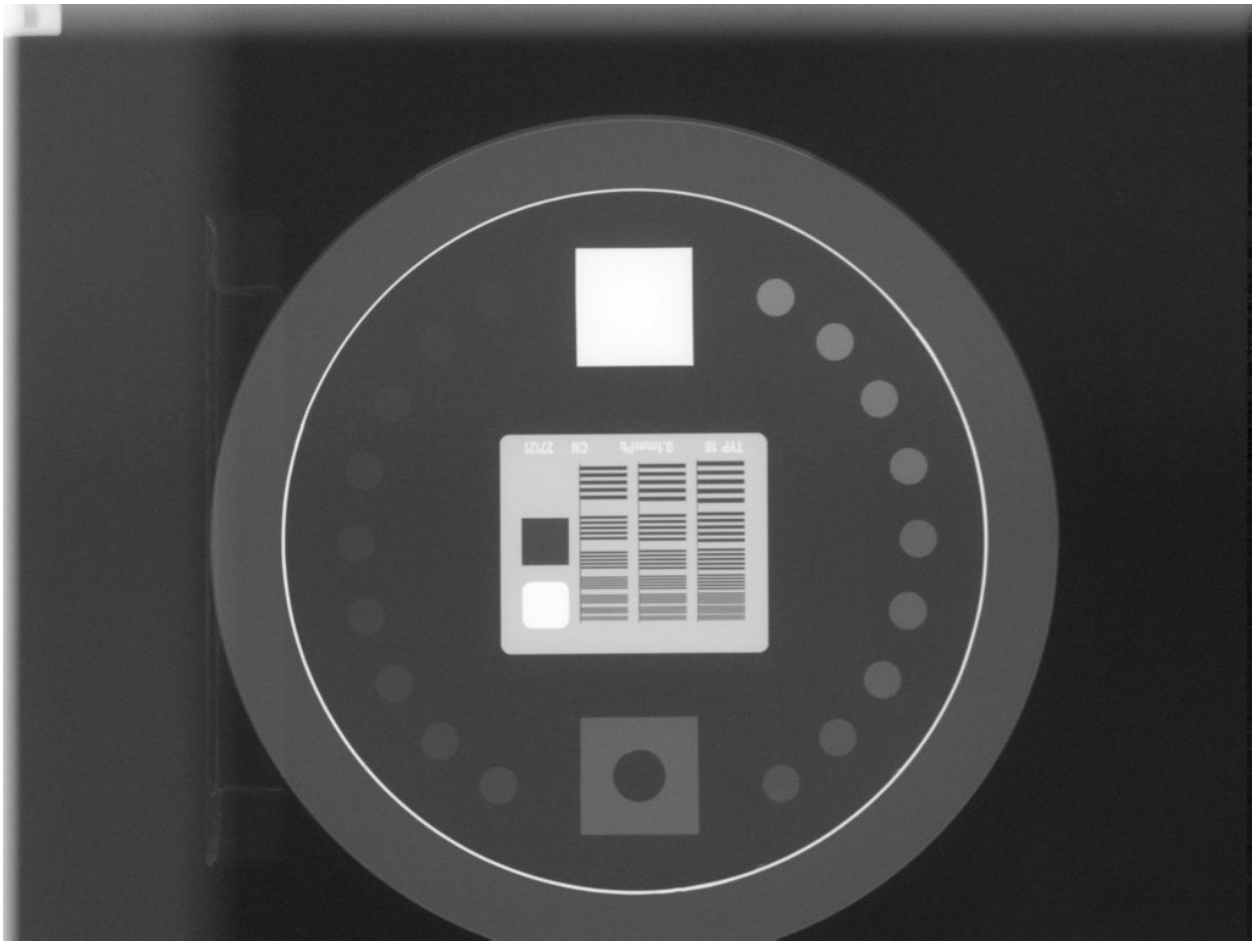
Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

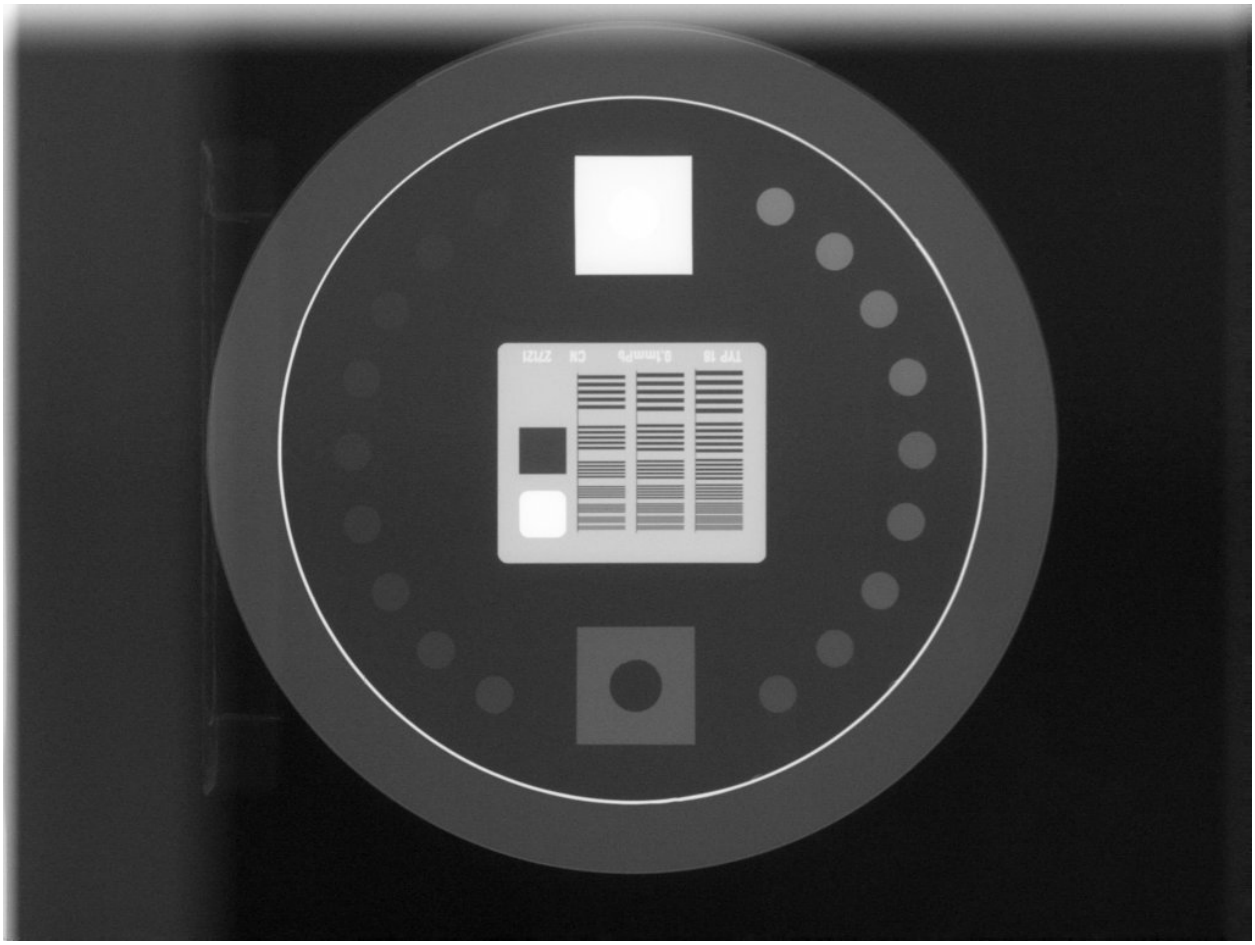
Troubleshooting

If you're having trouble getting the Leeds phantom analysis to work, first check out the *Troubleshooting* section. If the issue is not listed there, then it may be one of the issues below.

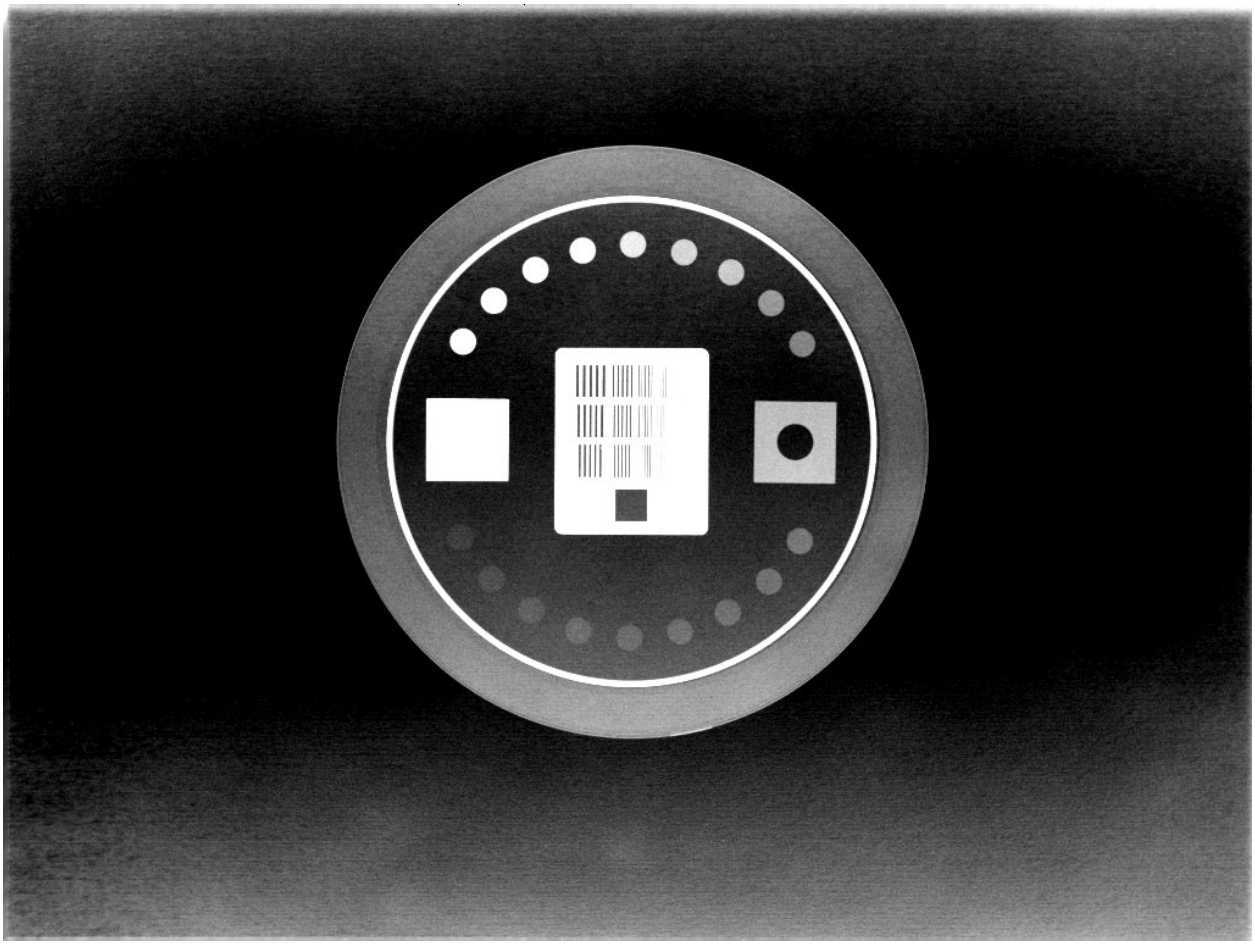
The most common reason for failing is having the phantom near an image edge. The resulting error is usually that the phantom angle cannot be determined. For example, this image would throw an error:



The below image also fails. Technically, the phantom is in the image, but the top blade skews the pixel values such that the phantom edge cannot be properly found at the top. This fails to identify the true phantom edge, causing the angle to also not be found:

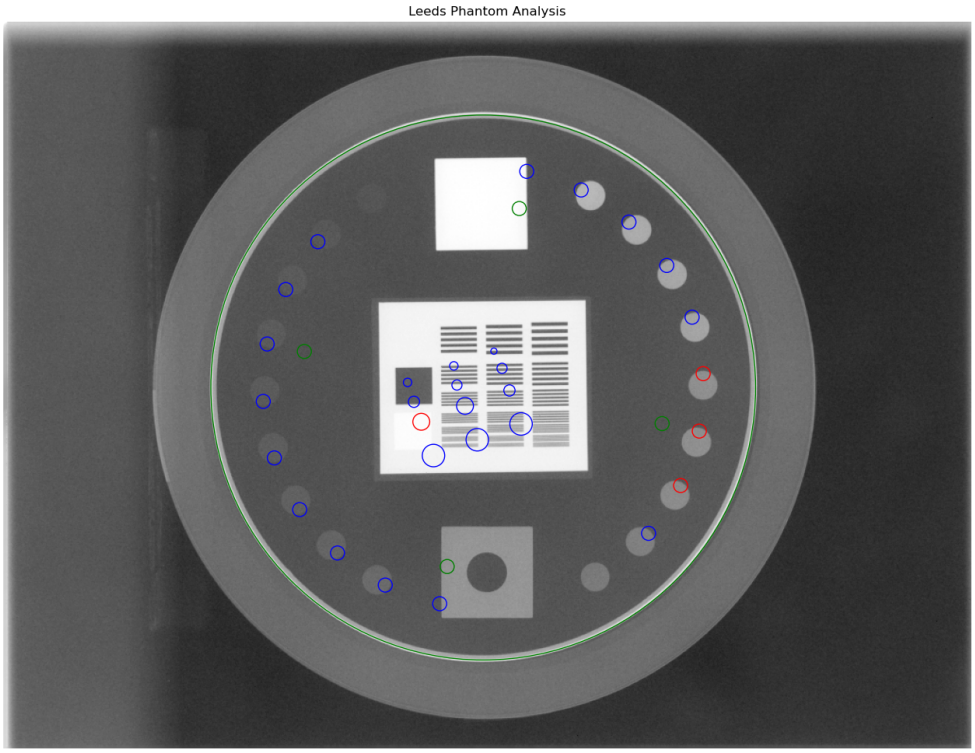


Another problem is that the image may have a non-uniform background. This can cause pylinac's automatic inversion correction to incorrectly invert the image. For example, this image falsely inverts:



When analyzed, the angle is 180 degrees opposite the lead square, causing the ROIs to be flipped 180 degrees. To correct this problem, pass `invert=True` to *analyze()*. This will force pylinac to invert the image the opposite way and correctly identify the lead square.

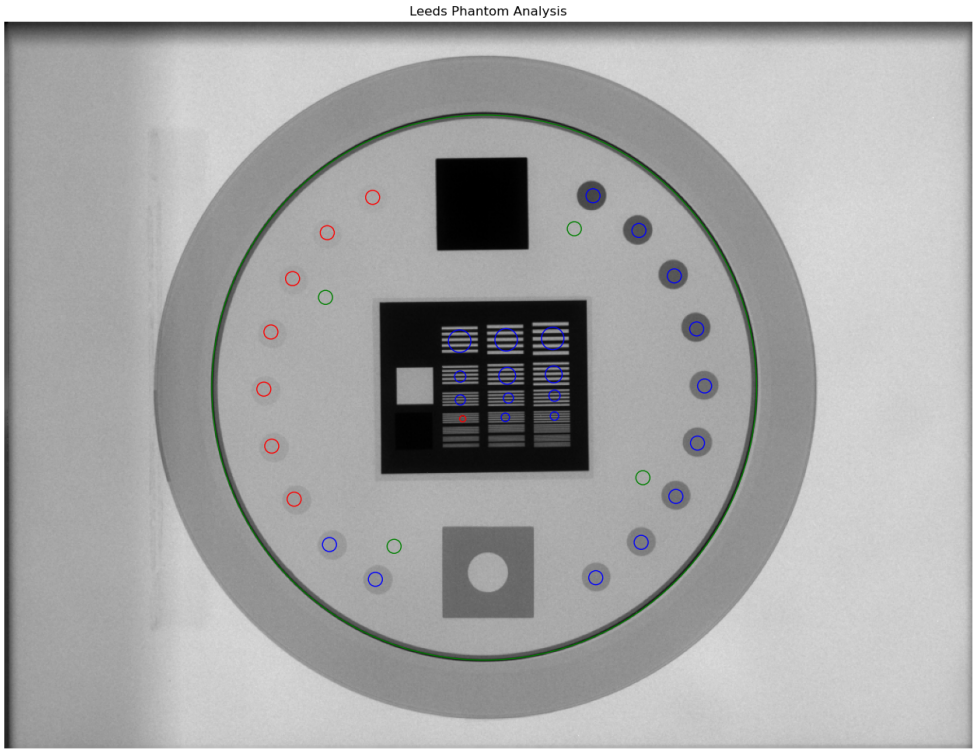
Another common problem is an offset analysis, as shown below:



This is caused by a wrong inversion.

Note: If the image flash is dark, then the image inversion is very likely wrong.

Again, pass `invert=True` to the `analyze` method. This is the same image but with invert=True:



6.15.5 PTW EPID QC Phantom

The PTW EPID QC phantom is an MV imaging quality assurance phantom and has high and low contrast regions, just as the Leeds phantom, but with different geometric configurations.

Image Acquisition

The EPID QC phantom appears to have a specific setup as recommended by the manufacturer. The phantom should have the high-contrast line pairs at the top of the image and low contrast at the bottom. The rotation is not automatically determined, so you should take care when setting up the phantom to be well-positioned.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.
- The images can be acquired with the phantom at any SSD.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must be at 0 degrees.
- The phantom must not be touching any image edges.
- The phantom should have the high-contrast linen pair regions toward the gantry stand/top.
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – A Canny edge search is performed on the image. Connected edges that are semi-round and angled are thought to possibly be the phantom. Of the ROIs, the one with the longest axis is said to be the phantom edge. The center of the bounding box of the ROI is set as the phantom center.
- **Determine phantom radius** – The major axis length of the ROI determined above serves as the phantom radius.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.
- **Calculate high contrast** – Again, because the phantom position and angle are known, offsets are applied to sample the high contrast line pair regions. For each sample, the relative MTF is calculated. See *Modulation Transfer Function*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

6.15.6 Standard Imaging QC-3 Phantom

The Standard Imaging phantom is an MV imaging quality assurance phantom and has high and low contrast regions, just as the Leeds phantom, but with different geometric configurations.

Image Acquisition

The Standard Imaging phantom has a specific setup as recommended by the manufacturer. The phantom should be angled 45 degrees, with the "1" pointed toward the gantry stand and centered along the CAX. For best results when using pylinac, open the jaws to fully cover the EPID, or at least give 1-2cm flash around the phantom edges.

Warning: If using the acrylic jig that comes with the phantom, place a spacer of at least a few mm between the jig and the phantom. E.g. a slice of foam on each angled edge. This is because the edge detection of the phantom may fail at certain energies with the phantom abutted to the acrylic jig.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.
- The images can be acquired with the phantom at any SSD.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must be at 45 degrees.
- The phantom must not be touching any image edges.
- The phantom should have the "1" pointing toward the gantry stand.
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – A Canny edge search is performed on the image. Connected edges that are semi-round and angled are thought to possibly be the phantom. Of the ROIs, the one with the longest axis is said to be the phantom edge. The center of the bounding box of the ROI is set as the phantom center.
- **Determine phantom radius and angle** – The major axis length of the ROI determined above serves as the phantom radius. The orientation of the edge ROI serves as the phantom angle.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.
- **Calculate high contrast** – Again, because the phantom position and angle are known, offsets are applied to sample the high contrast line pair regions. For each sample, the relative MTF is calculated. See *Modulation Transfer Function*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

Troubleshooting

If you're having issues with the StandardImaging class, make sure you have correctly positioned the phantom as per the manufacturer's instructions (also see *Image Acquisition*). One issue that may arise is incorrect inversion. If the jaws are closed tightly around the phantom, the automatic inversion correction may falsely invert the image, just as for the Leeds phantom. If you have an image that looks inverted or just plain weird, add `invert=True` to *`analyze()`*. If this doesn't help, reshoot the phantom with the jaws open.

6.15.7 Las Vegas Phantom

The Las Vegas phantom is for MV image quality testing and includes low contrast regions of varying contrast and size. There is also a *`ElektaLasVegas`* class that is very similar. This section covers both styles.

Image Acquisition

The Las Vegas phantom has a recommended position as stated on the phantom. Pylinac will however account for shifts and inversions. Best practices for the Las Vegas phantom:

- Keep the phantom from a couch edge or any rails.
- The field edge should be >=5mm from the phantom edge, preferably 10+mm.
- The orientation should have the largest "holes" towards the right side although this can be accounted for as an `analyze` parameter.
- The angle should be as close to 0 as possible, given above, although this can be accounted for as an `analyze` parameter.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must not be touching any image edges.
- The phantom should be at a cardinal angle (0, 90, 180, or 270 degrees) relative to the EPID.
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – A Canny edge search is performed on the image. Connected edges that are semi-round and angled are thought to possibly be the phantom. Of the ROIs, the one with the longest axis is said to be the phantom edge. The center of the bounding box of the ROI is set as the phantom center.

- **Determine phantom radius and angle** – The major axis length of the ROI determined above serves as the phantom radius. The orientation of the edge ROI serves as the phantom angle.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

6.15.8 Doselab MC2 MV & kV

The Doselab MC2 phantom is for both kV & MV image quality testing and includes low and high contrast regions of varying contrast. There are two high contrast sections, one intended for kV and one for MV.

Image Acquisition

The Doselab phantom has a recommended position as stated on the phantom. Pylinac will however account for shifts and inversions. Best practices for the Doselab phantom:

- Keep the phantom away from a couch edge or any rails.
- Center the phantom along the CAX.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must not be touching any image edges.
- The phantom should be at 45 degrees relative to the EPID.
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – A canny edge search is performed on the image. Connected edges that are semi-round and angled are thought to possibly be the phantom. Of the ROIs, the one with the longest axis is said to be the phantom edge. The center of the bounding box of the ROI is set as the phantom center.
- **Determine phantom radius and angle** – The major axis length of the ROI determined above serves as the phantom radius. The orientation of the edge ROI serves as the phantom angle.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

6.15.9 SNC MV & kV

The SNC MV and kV phantoms are for kV & MV image quality testing and includes low and high contrast regions of varying contrast.

Image Acquisition

The SNC phantoms typically use the angled setup jig. Best practices for the Doselab phantom:

- Keep the phantom away from a couch edge or any rails.
- Center the phantom along the CAX.
- Use the angled setup jig.
- For the MV phantom, have the longer side point inferiorly (i.e. **away** from the stand).
- For the kV phantom, have the longer side point superiorly (i.e. **toward** the stand).

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must not be touching any image edges.
- The phantom should be at 45 degrees relative to the EPID.
- The phantom should be centered near the CAX (<1-2cm).

Pre-Analysis

- **Determine phantom location** – A canny edge search is performed on the image. Connected edges that are semi-round and angled are thought to possibly be the phantom. Of the ROIs, the one with the longest axis is said to be the phantom edge. The center of the bounding box of the ROI is set as the phantom center.
- **Determine phantom radius** – The major axis length of the ROI determined above serves as the phantom radius.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

6.15.10 IBA Primus A

The IBA Primus A phantom is used for kV image analysis and includes low and high contrast regions of varying contrast.

Image Acquisition

Lay the phantom on the couch with the wedge step circle facing the top/gun and high-res square facing the bottom/target.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom must not be touching any image edges.
- The phantom should be at 0, 90, or 270 +/-5 degrees relative to the EPID where 0 is facing the gun.
- The dynamic wedge steps should be facing the gun side; the high-resolution square should be facing the target side.
- The phantom should be centered near the CAX (<2cm).

Pre-Analysis

- **Determine phantom location** – A Canny edge search is performed on the image. The ROI that approximates the size of the central crosshair of the phantom and is nearly at the center of the image is used as the phantom center location
- **Determine phantom radius** – The size of the above crosshair ROI is used as the basis for the phantom radius.
- **Fine-tune phantom angle** – The phantom angle is assumed to be around 0 (wedge steps facing gun), but fine-tuning is performed so that sensitive ROIs like MTF can be had with high accuracy. This is performed by taking a circular profile about the phantom at the radius of the wedge steps. The two areas of highest gradient will be at the first and last wedge steps. The center between these two points is the angle at which the phantom is "pointing" and will be used as the updated angle.

  > **Warning:** If the gradients cannot be found or if the determined angle is >5 degrees (caused by bad inversion, e.g.) a warning will be printed to the console and a default of 0 will be used.

Analysis

- **Calculate low contrast** – Because the phantom center and angle are known, the angles to the ROIs can also be known. From here, the contrast can be known; see *Contrast*.
- **Calculate high contrast** – Again, because the phantom position and angle are known, offsets are applied to sample the high contrast line pair regions. For each sample, the relative MTF is calculated. See *Modulation Transfer Function*.

Post-Analysis

- **Determine passing low and high contrast ROIs** – For each low and high contrast region, the determined value is compared to the threshold. The plot colors correspond to the pass/fail status.

6.15.11 Standard Imaging FC-2

The FC-2 phantom is for testing light/radiation coincidence.

Image Acquisition

The FC-2 phantom should be placed on the couch at 100cm SSD.

- Keep the phantom away from a couch edge or any rails.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom should be at a cardinal angle (0, 90, 180, or 270 degrees) relative to the EPID.
- The phantom should be centered near the CAX (<1cm).
- The phantom should be +/- 1cm from 100cm SSD.

Pre-Analysis

- **Determine BB set to use** – There are two sets of BBs, one for 10x10cm and another for 15x15cm. To get the maximum accuracy, the larger set is used if a 15x15cm field is irradiated. The field size is determined and if it's >14cm then the algorithm will look for the larger set. Otherwise, it will look for the smaller 4.

Analysis

- **Get BB centroid** – Once the BB set is chosen, image windows look for the BBs in a 1x1cm square. Once it finds them, the centroid of all 4 BBs is calculated.
- **Determine field center** – The field size is measured along the center of the image in the inplane and crossplane direction. A 5mm strip is averaged and used to reduce noise.

Post-Analysis

- **Comparing centroids** – The irradiated field centroid is compared to the EPID/image center as well as the the BB centroid. The field size is also reported.

Customizing behavior

The BB window as well as the expected BB positions, and field strip size can be overridden like so:

```
from pylinac import StandardImagingFC2


class MySIFC2(StandardImagingFC2):
    bb_sampling_box_size_mm = (
        20  # look at a 20x20mm window for the BB at the expected position
    )
    # change the 10x10 BB expected positions. This is in mm relative to the CAX.
    bb_positions_10x10 = {
        "TL": [-30, -30],
        "BL": [-30, 30],
        "TR": [30, -30],
        "BR": [30, 30],
    }
    bb_positions_15x15 = ...  # same as above
    field_strip_width_mm = 10  # 10mm strip in x and y to determine field size


# use as normal
fc2 = MySIFC2(...)
```

6.15.12 Doselab RLf

New in version 3.15.

The Doselab RLf is for testing light/radiation coincidence. See also *DoselabRLf*.

Image Acquisition

The RLf phantom should be placed on the couch at 100cm SSD.

- Keep the phantom away from a couch edge or any rails.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom should be at a cardinal angle (0, 90, 180, or 270 degrees) relative to the EPID.
- The phantom should be centered near the CAX (<2mm).

Analysis

- **Get BB centroid** – An image window looks for each BB on the inner side of each edge. After finding the BBs, the centroid is calculated.

  Note: The inner 10x10 BBs are always used regardless of the field size. This is because the BB detection is more robust when the BBs are away from a field edge. This also means that 10x10 analysis is slightly less robust that 15x15 analysis all else being equal.

- **Determine field center** – The field size is measured along the center of the image in the inplane and crossplane direction. A 5mm strip is averaged and used to reduce noise.

Post-Analysis

- **Comparing centroids** – The irradiated field centroid is compared to the EPID/image center as well as the the BB centroid. The field size is also reported.

6.15.13 IsoAlign

New in version 3.15.

The IsoAlign phantom is for testing light/radiation coincidence. See also *IsoAlign*.

Image Acquisition

The phantom should be placed on the couch at 100cm SSD.

- Keep the phantom away from a couch edge or any rails.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom should be at a cardinal angle (0, 90, 180, or 270 degrees) relative to the EPID.
- The phantom should be centered near the CAX (<2mm).

Analysis

- **Get BB centroid** – An image window looks for the central BB as well as 1 BB in each cardinal direction. After finding the BBs, the centroid is calculated.
- **Determine field center** – The field size is measured along the center of the image in the inplane and crossplane direction. A 10mm strip is averaged and used to reduce noise.

Post-Analysis

- **Comparing centroids** – The irradiated field centroid is compared to the EPID/image center as well as the the BB centroid. The field size is also reported.

6.15.14 IMT L-Rad

New in version 3.2.

The IMT L-Rad phantom is for testing light/radiation coincidence. Unlike the FC-2 phantom, the L-Rad's BBs are all the way at the edge of the phantom. This means for any size below 20x20cm those BBs can't be seen. Even at 20x20, the field edge partially obscures the BBs. For this reason, we only use the central BB for detection.

Image Acquisition

The L-Rad phantom should be placed on the couch at 100cm SSD.

- Keep the phantom away from a couch edge or any rails.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

> **Warning:** Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom should be at a cardinal angle (0, 90, 180, or 270 degrees) relative to the EPID.
- The phantom should be centered near the CAX (<3mm).

Analysis

- **Get BB centroid** – An image window looks for the central BB in a 1.2x1.2cm square. Once it finds it, the centroid is calculated.
- **Determine field center** – The field size is measured along the center of the image in the inplane and crossplane direction. A 5mm strip is averaged and used to reduce noise.

Post-Analysis

- **Comparing centroids** – The irradiated field centroid is compared to the EPID/image center as well as the the BB centroid. The field size is also reported.

6.15.15 SNC FSQA

New in version 3.3.

The SNC FSQA phantom is for light/radiation coincidence. It contains markers which guide the physicist on how to position the light field for either a 10x10 or 15x15cm field. There is also an offset BB 4cm at the top right of the image. Because of both *the philosophy* of pylinac on light/rad and also because pylinac is a library and not a GUI, there is no interaction to find the edge markers. Instead, we use the one offset BB and then offset that point back 4cm in each direction to get a "virtual center". This center is compared to the field center and EPID center. The expectation is that the physicist set up their field to the markers using the light field at the time of acquisition.

Image Acquisition

The FSQA phantom should be placed on the couch at 100cm SSD.

- Keep the phantom away from a couch edge or any rails.
- Keep the phantom as close to 0 degrees rotation as possible.

Algorithm

The algorithm works like such:

Allowances

- The images can be acquired at any SID.
- The images can be acquired with any EPID.

Restrictions

Warning: Analysis can fail or give unreliable results if any Restriction is violated.

- The phantom should be at 0 degrees relative to the EPID.
- The phantom should be roughly centered along the CAX (<3mm).

Analysis

- **Get BB centroid** – An image window looks for the top-right offset BB in a 1.2x1.2cm square. Once it finds it, a "virtual center" centroid is calculated by shifting the detected BB location by 4cm in each direction.
- **Determine field center** – The field size is measured along the center of the image in the inplane and crossplane direction. A 5mm strip is averaged and used to reduce noise.

Post-Analysis

- **Comparing centroids** – The irradiated field centroid is compared to the EPID/image center as well as the the BB centroid. The field size is also reported.

6.15.16 Creating a custom phantom

In the event you would like to analyze a phantom that pylinac does not analyze out of the box, the pylinac planar imaging module structure allows for generating new phantom analysis types quickly and easily. The benefit of this design is that with a few simple definitions you inherit a strong base of methods (e.g. plotting and PDF reports come for free).

Creating a new class involves a few different steps but can be done in a few minutes. The following is a guide for custom phantoms:

1. Subclass the `ImagePhantomBase` class:

```
from pylinac.planar_imaging import ImagePhantomBase


class CustomPhantom(ImagePhantomBase):
    pass
```

2. Define the `common_name`. This is the name shown in plots and PDF reports.

```
class CustomPhantom(ImagePhantomBase):
    common_name = "Custom Phantom v2.0"
```

3. If the phantom has a high-contrast measurement object, define the ROI locations.

```
class CustomPhantom(ImagePhantomBase):
    ...
    high_contrast_roi_settings = {
        "roi 1": {
            "distance from center": 0.5,
            "angle": 30,
            "roi radius": 0.05,
            "lp/mm": 0.2,
        },
        # add as many ROIs as are needed
    }
```

Note: The exact values of your ROIs will need to be empirically determined. This usually involves an iterative process of adjusting the values until the values are satisfactory based on the ROI sample alignment to the actual ROIs.

4. If the phantom has a low-contrast measurement object, define the sample ROI and background ROI locations.

```
class CustomPhantom(ImagePhantomBase):
    ...
    low_contrast_roi_settings = {
        "roi 1": {
            "distance from center": 0.5,
            "angle": 30,
            "roi radius": 0.05,
        },  # no lp/mm key
        # add as many ROIs as are needed
    }
    low_contrast_background_roi_settings = {
        "roi 1": {"distance from center": 0.3, "angle": -45, "roi radius": 0.02},
```

(continues on next page)

(continued from previous page)

```
        # add as many ROIs as are needed
    }
```

Note: The exact values of your ROIs will need to be empirically determined. This usually involves an iterative process of adjusting the values until the values are satisfactory based on the ROI sample alignment to the actual ROIs.

5. Set the "detection conditions", which is the list of rules that must be true to properly detect the phantom ROI. E.g. the phantom should be near the center of the image. Detection conditions must always have a specific signature as shown below:

```
def my_special_detection_condition(
    region: RegionProperties, instance: object, rtol: float
) -> bool:
    # region is a scikit regionprop (https://scikit-image.org/docs/dev/api/skimage.
↪measure.html#skimage.measure.regionprops)
    # instance == self of the phantom
    # rtol is relative tolerance of agreement. Don't have to use this.
    do_stuff  # e.g. is the region size and position correct?
    return bool(result)  # must always return a boolean


class CustomPhantom(ImagePhantomBase):
    detection_conditions = [
        my_special_detection_condition,
    ]  # list of conditions; add as many as you want.
```

6. Optionally, add a phantom outline object. This helps visualize the algorithm's determination of the size, center, and angle. If no object is defined, then no outline will be shown. This step is optional.

```
class CustomPhantom(ImagePhantomBase):
    ...
    phantom_outline_object = {
        "Circle": {"radius ratio": 0.5}
    }  # to create a circular outline
    # or...
    phantom_outline_object = {
        "Rectangle": {"width ratio": 0.5, "height ratio": 0.3}
    }  # to create a rectangular outline
```

At this point you could technically call it done. You would need to always override the angle, center, and size values in the analyze method however. To automate this part you will need to fill in the associated logic. You can use whatever method you like. What I have found most useful is to use an edge detection algorithm and find the outline of the phantom.

```
class CustomPhantom(ImagePhantomBase):
    ...

    def _phantom_center_calc(self) -> Point:
        # do stuff in here to determine the center point location.
        # don't forget to return as a Point item (pylinac.core.geometry.Point).
        ...
```

(continues on next page)

(continued from previous page)

```
    def _phantom_radius_calc(self) -> float:
        # do stuff in here to return a float that represents the phantom radius value.
        # This value does not have to relate to a physical measure. It simply defines a␣
↪value that the ROIs scale by.
        ...

    def _phantom_angle_calc(self) -> float:
        # do stuff in here to return a float that represents the angle of the phantom.
        # Again, this value does not have to correspond to reality; it simply offsets␣
↪the ROIs.
        # You may also return a constant if you like for any of these.
        ...
```

Congratulations! You now have a fully-functioning custom phantom. By using the base class and the predefined attributes and methods, the plotting and PDF report functionality comes for free.

6.15.17 Usage tips, tweaks, & troubleshooting

Set the SSD of your phantom

If your phantom is at a non-standard distance (!= 1000mm), e.g. sitting on the EPID panel, you can specify its distance via the `ssd` parameter.

Warning: The `ssd` should be in mm, not cm. Pylinac is moving toward consistent units on everything and it will be mm for distance.

```
from pylinac import StandardImagingQC3

qc = StandardImagingQC3(...)
qc.analyze(..., ssd=1500)  # distance to the phantom in mm.
```

Adjust an ROI on an existing phantom

To adjust an ROI, override the relevant attribute or create a subclass. E.g. to move the 2nd ROI of the high-contrast ROI set of the QC-3 phantom:

```
from pylinac import StandardImagingQC3

StandardImagingQC3.high_contrast_roi_settings["roi 1"][
    "distance from center"
] = 0.05  # overrides that one setting
qc3 = StandardImagingQC3(...)

# or


class TweakedStandardImagingQC3(StandardImagingQC3):
```

(continues on next page)

(continued from previous page)

```
    high_contrast_roi_settings = {
        "roi 1": ...
    }  # note that you must replace ALL the values


qc3 = TweakedStandardImagingQC3(...)
```

Calculating Uniformity

The uniformity of the phantom can be found by using the *percent_integral_uniformity()* method. This uses the same equation as ACR for CT uniformity. See the "Uniformity" section under *ACR Analysis* for more information.

The PIU is calculated over all the low-contrast ROIs and the lowest (worst) PIU is returned.

For robustness, the 1st and 99th percentiles are used rather than the min/max. The true min/max can be influenced by salt and paper noise. To use the true min and max, set the percentiles to 0 and 100 respectively:

Note: This will not be reflected in the `results_data` structure.

```
leeds = LeedsTOR(...)
leeds.analyze(...)
print(leeds.percent_integral_uniformity(percentiles=(0, 100)))  # uses the true min/max
```

Warning: This equation was chosen because it is common and understood, but it does come with pitfalls. It is not designed to handle negative values or 0. The calculated result may be misleading if these conditions exist.

Calculate a specific MTF

To calculate a specific MTF value, i.e. the frequency at a given MTF%:

```
dl = DoselabMC2kV(...)
dl.analyze(...)
print(dl.mtf.relative_resolution(x=50))  # 50% rMTF
```

Get/View the contrast of a low-contrast ROI

```
leeds = LeedsTOR(...)
leeds.analyze(...)
print(leeds.low_contrast_rois[1].contrast)  # get the 2nd ROI contrast value
```

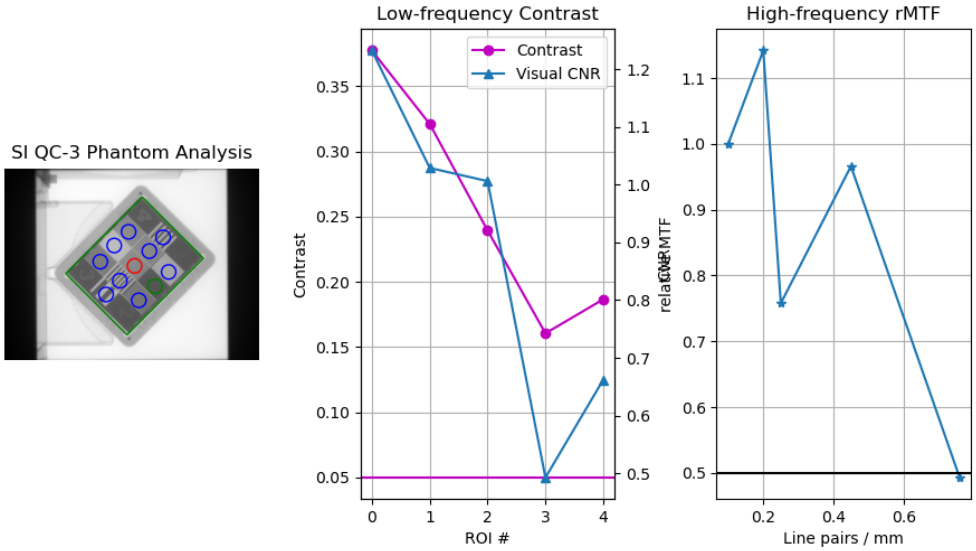
Loosen the ROI finding conditions

If for some reason you have a need to loosen the existing phantom-finding algorithm conditions you can do so fairly easily by overloading the current tooling:

```
from pylinac.planar_imaging import is_right_size, is_centered, LeedsTOR


def is_right_size_loose(region, instance, rtol=0.3):  # rtol default is 0.1
    return is_right_size(region, instance, rtol)


# set the new condition for whatever
LeedsTOR.detection_conditions = [is_right_size_loose, is_centered]
# proceed as normal
myleeds = LeedsTOR(...)
```

6.15.18 Wrong phantom angle

It may sometimes be that the angle of the phantom appears incorrect, or the results appear incorrect. E.g. here is a QC-3 phantom:



The ROIs appear correct, the but the contrast and MTF do not monotonically decrease, indicating a problem. In this case, it is because the image acquisition rules were not followed. For the QC-3, the "1" should always point toward the gantry, as per the manual. When oriented this way, the results will be correct.

6.15.19 Light/Radiation philosophy

Pylinac (or rather the author) has an opinionated philosophy about light vs radiation that affects the way light/radiation analysis is performed. In our opinion, light/rad **using a phantom** is antiquated as EPIDs are robust enough nowadays to be quite reliable, at least for Varian machines. By using something as simple as graph paper after mechanical measurements, a light field can be set and a simple open field delivered. This open field size and CAX offset can be compared to the nominal values set by the physicist at the time of acquisition.

Short of using CCD cameras or specialty equipment like phosphorus, there is no true way to know the light field measurement. All we have is what the physicist set up to. If the physicist sets up to a nominal size like 10x10, then a radiation field measurement can be compared to that rather simply with common field analysis. E.g if the measured field size was 10.1x10.6mm then the error between light and rad is 0.1 and 0.6mm respectively. The CAX offset follows the same logic.

You may disagree, but this is here for the purposes of explaining our philosophy and why light/rad does (or does not do) what it does.

We provide these light/rad routines because customers ask for them, not because we recommend them.

6.15.20 API Documentation

class pylinac.planar_imaging.**LeedsTOR**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Leeds TOR phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor`() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**LeedsTORBlue**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *LeedsTOR*

The Leeds TOR (Blue) is for analyzing older Leeds phantoms which have slightly offset ROIs compared to the newer, red-ring variant.

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

classmethod from_demo_image()

Instantiate and load the demo image.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale

the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_url`(*url: str*)

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`
The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity`(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`
The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`
The skimage region of the phantom outline.

`plot_analyzed_image`(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

static run_demo() → None

Run the Leeds TOR phantom analysis demonstration.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**StandardImagingQC3**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

classmethod from_demo_image()

Instantiate and load the demo image.

static run_demo() → None

Run the Standard Imaging QC-3 phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99)) → float | None`

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float

The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties

The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**StandardImagingQCkV**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *StandardImagingQC3*

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Standard Imaging QC-3 phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor`() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**LasVegas**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo()

Run the Las Vegas phantom analysis demonstration.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis. Overridden because ROIs seen is based on visibility, not CNR.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

results_data(*as_dict: bool = False*) → PlanarResult | dict

Overridden because ROIs seen is based on visibility, not CNR

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`
Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`
The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99)) → float | None`
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`
The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`
The skimage region of the phantom outline.

`plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False, **plt_kwargs: dict) → tuple[list[Figure], list[str]]`
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**ElektaLasVegas**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *LasVegas*

Elekta's variant of the Las Vegas.

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo()

Run the Elekta Las Vegas phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis. Overridden because ROIs seen is based on visibility, not CNR.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`results_data`(*as_dict: bool = False*) → PlanarResult | dict

Overridden because ROIs seen is based on visibility, not CNR

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**DoselabMC2MV**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *DoselabMC2kV*

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Doselab MC2 MV-area phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99))` → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`

The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`

The skimage region of the phantom outline.

`plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False, **plt_kwargs: dict)` → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**DoselabMC2kV**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Doselab MC2 kV-area phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor`() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**SNCMV**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *SNCkV*

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Sun Nuclear MV-QA phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99))` → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`

The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`

The skimage region of the phantom outline.

`plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False, **plt_kwargs: dict)` → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**SNCMV12510**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *SNCMV*

The older SNC MV QA phantom w/ model number 1251000

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`static run_demo`() → None

Run the Sun Nuclear MV-QA phantom analysis demonstration.

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**SNCkV**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Sun Nuclear kV-QA phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99))` → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`

The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`

The skimage region of the phantom outline.

`plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False, **plt_kwargs: dict)` → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**PTWEPIDQC**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Standard Imaging QC-3 phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*image: bool = True*, *low_contrast: bool = True*, *high_contrast: bool = True*, *show: bool = True*, *split_plots: bool = False*, ***plt_kwargs: dict*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`results`(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

`save_analyzed_image`(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling`() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor`() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**IBAPrimusA**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

property phantom_angle: float

Cache this; calculating the angle is expensive

static run_demo() → None

Run the Standard Imaging QC-3 phantom analysis demonstration.

analyze(*low_contrast_threshold: float = 0.05*, *high_contrast_threshold: float = 0.5*, *invert: bool = False*, *angle_override: float | None = None*, *center_override: tuple | None = None*, *size_override: float | None = None*, *ssd: float | Literal['auto'] = 'auto'*, *low_contrast_method: str = 'Michelson'*, *visibility_threshold: float = 100*) → None

Analyze the phantom using the provided thresholds and settings.

Parameters

low_contrast_threshold
[float] This is the contrast threshold value which defines any low-contrast ROI as passing or failing.

high_contrast_threshold
[float] This is the contrast threshold value which defines any high-contrast ROI as passing or failing.

invert
[bool] Whether to force an inversion of the image. This is useful if pylinac's automatic inversion algorithm fails to properly invert the image.

angle_override
[None, float] A manual override of the angle of the phantom. If None, pylinac will automatically determine the angle. If a value is passed, this value will override the automatic detection.

Note: 0 is pointing from the center toward the right and positive values go counterclockwise.

---

center_override
[None, 2-element tuple] A manual override of the center point of the phantom. If None, pylinac will automatically determine the center. If a value is passed, this value will override the automatic detection. Format is (x, y)/(col, row).

size_override
[None, float] A manual override of the relative size of the phantom. This size value is used to scale the positions of the ROIs from the center. If None, pylinac will automatically determine the size. If a value is passed, this value will override the automatic sizing.

Note: This value is not necessarily the physical size of the phantom. It is an arbitrary value.

---

ssd
The SSD of the phantom itself in mm. If set to "auto", will first search for the phantom at the SAD, then at 5cm above the SID.

low_contrast_method
The equation to use for calculating low contrast.

visibility_threshold
The threshold for whether an ROI is "seen".

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99))` → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`

The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`

The skimage region of the phantom outline.

`plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False, **plt_kwargs: dict)` → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

image
[bool] Show the image.

low_contrast
[bool] Show the low contrast values plot.

high_contrast
[bool] Show the high contrast values plot.

show
[bool] Whether to actually show the image when called.

split_plots
[bool] Whether to split the resulting image into individual plots. Useful for saving images into individual files.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

Parameters

as_list
[bool] Whether to return as a list of strings vs single string. Pretty much for internal usage.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_mtf.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

class pylinac.planar_imaging.**StandardImagingFC2**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: ImagePhantomBase

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the Standard Imaging FC-2 phantom analysis demonstration.

analyze(*invert: bool = False*, *fwxm: int = 50*) → None

Analyze the FC-2 phantom to find the BBs and the open field and compare to each other as well as the EPID.

Parameters

invert
[bool] Whether to force-invert the image from the auto-detected inversion.

fwxm
[int] The FWXM value to use to detect the field. For flattened fields, the default of 50 should be fine. For FFF fields, consider using a lower value such as 25-30.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

property field_epid_offset_mm: *Vector*

Field offset from CAX using vector difference

property field_bb_offset_mm: *Vector*

Field offset from BB centroid using vector difference

results_data(*as_dict: bool = False*) → LightRadResult | dict

Return the results as a dict or dataclass

plot_analyzed_image(*show: bool = True*, ***kwargs*) → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

show
[bool] Whether to actually show the image when called.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`classmethod from_demo_image()`

Instantiate and load the demo image.

`classmethod from_url(url: str)`

Parameters

url
[str] The URL to the image.

`property magnification_factor: float`

The mag factor of the image based on SSD vs SAD

`percent_integral_uniformity(percentiles: tuple[float, float] = (1, 99)) → float | None`

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

`property phantom_bbox_size_px: float`

The phantom bounding box size in pixels^2 at the isoplane.

`property phantom_ski_region: RegionProperties`

The skimage region of the phantom outline.

`window_ceiling() → float | None`

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor() → float | None`

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**IMTLRad**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *StandardImagingFC2*

The IMT light/rad phantom: https://www.imtqa.com/products/l-rad

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

analyze(*invert: bool = False*, *fwxm: int = 50*) → None

Analyze the FC-2 phantom to find the BBs and the open field and compare to each other as well as the EPID.

Parameters

invert
[bool] Whether to force-invert the image from the auto-detected inversion.

fwxm
[int] The FWXM value to use to detect the field. For flattened fields, the default of 50 should be fine. For FFF fields, consider using a lower value such as 25-30.

property field_bb_offset_mm: *Vector*

Field offset from BB centroid using vector difference

property field_epid_offset_mm: *Vector*

Field offset from CAX using vector difference

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float

The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float

The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties

The skimage region of the phantom outline.

plot_analyzed_image(*show: bool = True*, ***kwargs*) → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

show
[bool] Whether to actually show the image when called.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

results_data(*as_dict: bool = False*) → LightRadResult | dict

Return the results as a dict or dataclass

static run_demo() → None

Run the Standard Imaging FC-2 phantom analysis demonstration.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

`window_ceiling()` → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

`window_floor()` → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

`class pylinac.planar_imaging.DoselabRLf(`*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*`)`

Bases: *StandardImagingFC2*

The Doselab light/rad phantom

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

`static run_demo()` → None

Run the Doselab RFl phantom analysis demonstration.

`analyze(`*invert: bool = False*, *fwxm: int = 50*`)` → None

Analyze the FC-2 phantom to find the BBs and the open field and compare to each other as well as the EPID.

Parameters

invert
[bool] Whether to force-invert the image from the auto-detected inversion.

fwxm
[int] The FWXM value to use to detect the field. For flattened fields, the default of 50 should be fine. For FFF fields, consider using a lower value such as 25-30.

property field_bb_offset_mm: *Vector*

Field offset from BB centroid using vector difference

property field_epid_offset_mm: *Vector*

Field offset from CAX using vector difference

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float

The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float

The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties

The skimage region of the phantom outline.

plot_analyzed_image(*show: bool = True*, ***kwargs*) → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

show
[bool] Whether to actually show the image when called.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

results_data(*as_dict: bool = False*) → LightRadResult | dict

Return the results as a dict or dataclass

save_analyzed_image(*filename: None | str | BinaryIO = None*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**IsoAlign**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *StandardImagingFC2*

The PTW Iso-Align light/rad phantom

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

static run_demo() → None

Run the phantom analysis demonstration.

analyze(*invert: bool = False*, *fwxm: int = 50*) → None

Analyze the FC-2 phantom to find the BBs and the open field and compare to each other as well as the EPID.

Parameters

invert
[bool] Whether to force-invert the image from the auto-detected inversion.

fwxm
[int] The FWXM value to use to detect the field. For flattened fields, the default of 50 should be fine. For FFF fields, consider using a lower value such as 25-30.

property field_bb_offset_mm: *Vector*

Field offset from BB centroid using vector difference

property field_epid_offset_mm: *Vector*

Field offset from CAX using vector difference

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float

The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None

Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float

The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties

The skimage region of the phantom outline.

plot_analyzed_image(*show: bool = True*, ***kwargs*) → tuple[list[Figure], list[str]]

Plot the analyzed image.

Parameters

show
[bool] Whether to actually show the image when called.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]

Return the results of the analysis.

results_data(*as_dict: bool = False*) → LightRadResult | dict

Return the results as a dict or dataclass

save_analyzed_image(*filename: None | str | BinaryIO = None*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

class pylinac.planar_imaging.**SNCFSQA**(*filepath: str | BinaryIO | Path*, *normalize: bool = True*, *image_kwargs: dict | None = None*)

Bases: *StandardImagingFC2*

SNC light/rad phantom. See the 'FSQA' phantom and specs: https://www.sunnuclear.com/products/suncheck-machine.

Unlike other light/rad phantoms, this does not have at least a centered BB. The edge markers are in the penumbra and thus detecting them is difficult. We thus detect the one offset marker in the top right of the image. This is offset by 4cm in each direction. We can then assume that the phantom center is -4cm from this point, creating a 'virtual center' so we have an apples-to-apples comparison.

Parameters

filepath
[str] Path to the image file.

normalize: bool
Whether to "ground" and normalize the image. This can affect contrast measurements, but for backwards compatibility this is True. You may want to set this to False if trying to compare with other software.

image_kwargs
[dict] Keywords passed to the image load function; this would include things like DPI or SID if applicable

analyze(*invert: bool = False*, *fwxm: int = 50*) → None

Analyze the FC-2 phantom to find the BBs and the open field and compare to each other as well as the EPID.

Parameters

invert
[bool] Whether to force-invert the image from the auto-detected inversion.

fwxm
[int] The FWXM value to use to detect the field. For flattened fields, the default of 50 should be fine. For FFF fields, consider using a lower value such as 25-30.

property field_bb_offset_mm: *Vector*

Field offset from BB centroid using vector difference

property field_epid_offset_mm: *Vector*

Field offset from CAX using vector difference

classmethod from_demo_image()

Instantiate and load the demo image.

classmethod from_url(*url: str*)

Parameters

url
[str] The URL to the image.

property magnification_factor: float
The mag factor of the image based on SSD vs SAD

percent_integral_uniformity(*percentiles: tuple[float, float] = (1, 99)*) → float | None
Calculate and return the percent integral uniformity (PIU). This uses a similar equation as ACR does for CT protocols. The PIU is calculated over all the low contrast ROIs and the lowest (worst) PIU is returned.

If the phantom does not contain low-contrast ROIs, None is returned.

property phantom_bbox_size_px: float
The phantom bounding box size in pixels^2 at the isoplane.

property phantom_ski_region: RegionProperties
The skimage region of the phantom outline.

plot_analyzed_image(*show: bool = True*, ***kwargs*) → tuple[list[Figure], list[str]]
Plot the analyzed image.

Parameters

show
[bool] Whether to actually show the image when called.

publish_pdf(*filename: str*, *notes: str = None*, *open_file: bool = False*, *metadata: dict | None = None*, *logo: Path | str | None = None*)
Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra data to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

results(*as_list: bool = False*) → str | list[str]
Return the results of the analysis.

results_data(*as_dict: bool = False*) → LightRadResult | dict

Return the results as a dict or dataclass

static run_demo() → None

Run the Standard Imaging FC-2 phantom analysis demonstration.

save_analyzed_image(*filename: None | str | BinaryIO = None*, *to_streams: bool = False*, ***kwargs*) → dict[str, BinaryIO] | list[str] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

filename
[None, str, stream] A string representing where to save the file to. If split_plots and to_streams are both true, leave as None as newly-created streams are returned.

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

window_ceiling() → float | None

The value to use as the maximum when displaying the image. Helps show contrast of images, specifically if there is an open background

window_floor() → float | None

The value to use as the minimum when displaying the image (see https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html) Helps show contrast of images, specifically if there is an open background

6.16 Field Analysis

6.16.1 Overview

The field analysis module (`pylinac.field_analysis`) allows a physicist to analyze metrics from an EPID to measure penumbra, field width, etc. Additionally, protocols can be used which can calculate flatness & symmetry. The module is very flexible, allowing users to choose different types of interpolation, normalization, centering, etc. Users can also creat custom protocols to perform other types of field analysis within the main pylinac flow.

The module implements traditional analysis like FWHM as well as new methods as outlined in the pre-publication of the NCS-33 report which include edge fitting for FFF fields as well as a "top" calculation for the center position of FFF beams.

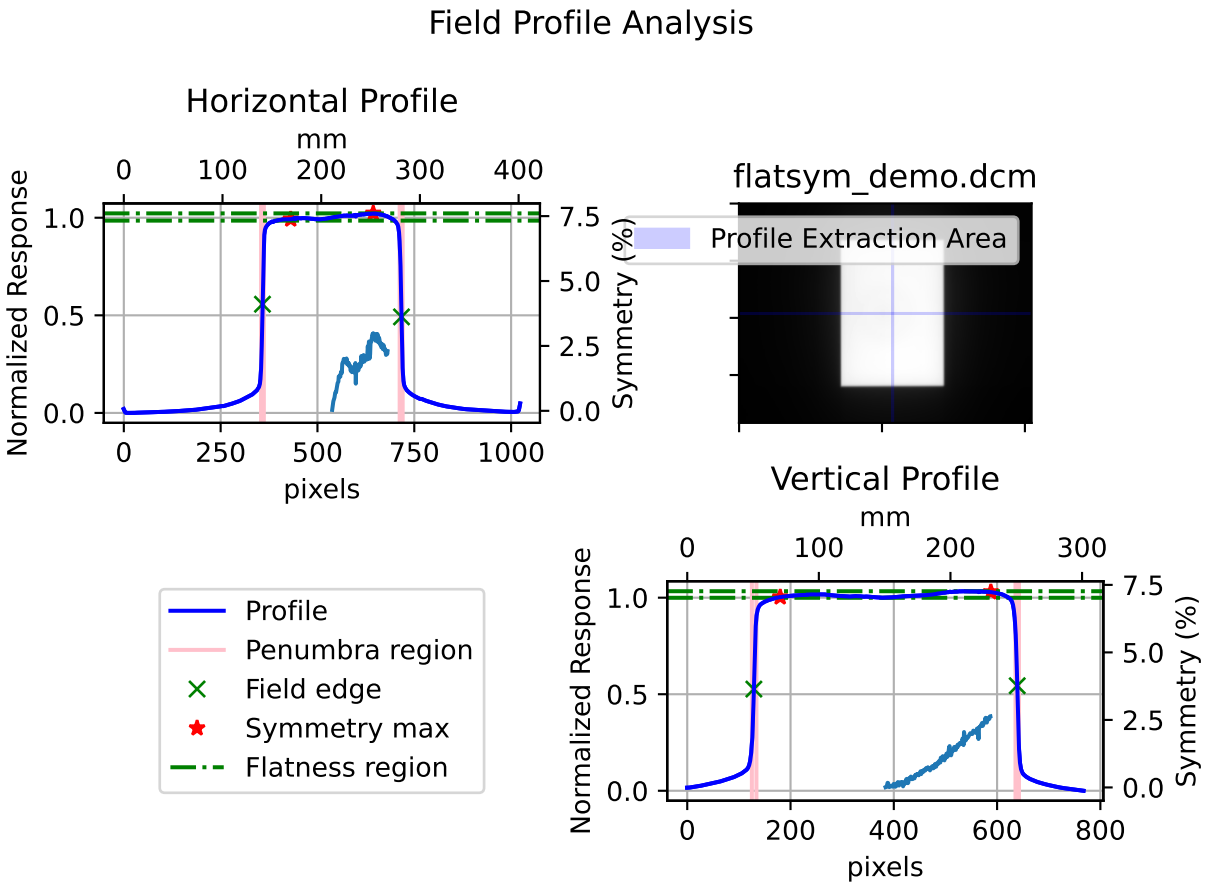
Note: This is not a purely faithful implementation of NCS-33. There are a few differences w/r/t how stringent field widths are applied. E.g. the "top" calculation in NCS-33 is over the central 5cm. Pylinac simply uses a field width ratio which may or may not be 5cm.

The module's main class is *`FieldAnalysis`* which is used for EPID images.

6.16.2 Running the Demo

To run the demo, import the main class and run the demo method:

```
from pylinac import FieldAnalysis

FieldAnalysis.run_demo()
```



Which will also result in the following output:

```
Field Analysis Results
----------------------
File: E:\OneDrive - F...demo_files\flatsym_demo.dcm
Protocol: VARIAN
Centering method: Beam center
Normalization method: Beam center
Interpolation: Linear
Edge detection method: Inflection Derivative

Penumbra width (20/80):
Left: 2.7mm
Right: 3.0mm
Top: 3.9mm
```

(continues on next page)

(continued from previous page)

```
Bottom: 2.8mm

Field Size:
Horizontal: 140.9mm
Vertical: 200.3mm

CAX to edge distances:
CAX -> Top edge: 99.8mm
CAX -> Bottom edge: 100.5mm
CAX -> Left edge: 60.4mm
CAX -> Right edge: 80.5mm

Top slope: -0.006%/mm
Bottom slope: 0.044%/mm
Left slope: 0.013%/mm
Right slope: 0.014%/mm

Protocol data:
--------------
Vertical symmetry: -2.631%
Horizontal symmetry: -3.006%

Vertical flatness: 1.700%
Horizontal flatness: 1.857%
```

6.16.3 Typical Use

In most instances, a physicist is interested in quickly calculating the flatness, symmetry, or both of the image in question. The `field_analysis` module allows you to do this easily and quickly.

To get started, import the *FieldAnalysis* class:

```
from pylinac import FieldAnalysis
```

Loading images is easy and just like any other module:

```
# from a file
my_file = r"C:/my/QA/folder/img.dcm"
my_img = FieldAnalysis(path=my_file)
```

Alternatively, you can load data from a 2D device array:

```
from pylinac import DeviceFieldAnalysis

# Profiler file
my_file = r"C:/my/profiler/data.prm"
my_img = DeviceFieldAnalysis(path=my_file)
```

If you don't have an image you can load the demo image:

```
my_img = FieldAnalysis.from_demo_image()
```

You can then calculate several field metrics with the *analyze()* method:

```
my_img.analyze()
```

After analysis, the results can be printed, plotted, or saved to a PDF:

```
print(my_img.results())  # print results as a string
my_img.plot_analyzed_image()  # matplotlib image
my_img.publish_pdf(filename="flatsym.pdf")  # create PDF and save to file
my_img.results_data()  # dict of results
```

6.16.4 Analyze Options

The analysis algorithm allows the user to change numerous parameters about the analysis including automatic/manual centering, profile extraction width, field width ratio, penumbra values, interpolation, normalization, and edge detection. See *pylinac.field_analysis.FieldAnalysis.analyze()* for details on each parameter

The protocol can also be specified here; this is where both default and custom algorithms like flatness and symmetry can be used. See *Protocol Definitions* for the common flatness/symmetry algorithms provided out of the box. For custom protocols, see *Creating & Using Custom Protocols*.

```
from pylinac import Protocol, Centering, Edge, Normalization, Interpolation

my_img.analyze(
    protocol=Protocol.ELEKTA,
    centering=Centering.BEAM_CENTER,
    in_field_ratio=0.8,
    is_FFF=True,
    interpolation=Interpolation.SPLINE,
    interpolation_resolution_mm=0.2,
    edge_detection_method=Edge.INFLECTION_HILL,
)
```

Centering

There are 3 centering options: manual, beam center, and geometric center.

Manual

Manual centering means that you as the user specify the position of the image that the profiles are taken from.

```
from pylinac import FieldAnalysis, Centering

fa = FieldAnalysis(...)
fa.analyze(..., centering=Centering.MANUAL)  # default is the middle of the image

# or specify a custom location
fa.analyze(..., centering=Centering.MANUAL, vert_position=0.3, horiz_position=0.8)
# take profile at 30% width (i.e. left side) and 80% height
```

Beam center

This is the default for EPID images/`FieldAnalysis`. It first looks for the field to find the approximate center along each axis. Then it extracts the profiles and continues. This is helpful if you always want to be at the center of the field, even for offset fields or wedges.

```
from pylinac import FieldAnalysis, Centering

fa = FieldAnalysis(...)
fa.analyze(...)  # nothing special needed as it's the default

# You CANNOT specify a position. These values will be ignored
fa.analyze(..., centering=Centering.BEAM_CENTER, vert_position=0.3, horiz_position=0.8)
# this is allowed but will result in the same result as above
```

Geometric center

This is the default for 2D device arrays/`DeviceFieldAnalysis`. It will always find the middle pixel and extract the profiles from there. This is helpful if you always want to be at the center of the image.

```
from pylinac import FieldAnalysis, Centering

fa = FieldAnalysis(...)
fa.analyze(...)  # nothing special needed as it's the default

# You CANNOT specify a position. These values will be ignored
fa.analyze(
    ..., centering=Centering.GEOMETRIC_CENTER, vert_position=0.3, horiz_position=0.8
)
# this is allowed but will result in the same result as above
```

Edge detection

Edge detection is important for determining the field width and beam center (which is often used for symmetry). There are 3 detection strategies: FWHM, inflection via derivative, and inflection via the Hill/sigmoid/4PNLR function.

FWHM

The full-width half-max strategy is traditional and works for flat beams. It can give poor values for FFF beams.

```
from pylinac import FieldAnalysis, Edge

fa = FieldAnalysis(...)
fa.analyze(..., edge_detection_method=Edge.FWHM)
```

Inflection (derivative)

The inflection point via the derivative is useful for both flat and FFF beams, and is thus the default for `FieldAnalysis`. The method will find the positions of the max and min derivative of the values. Using a 0-crossing of the 2nd derivative can be tripped up by noise so it is not used.

Note: This method is recommended for high spatial resolution images such as the EPID, where the derivative has several points to use at the beam edge. It is not recommended for 2D device arrays.

```
from pylinac import FieldAnalysis, Edge

fa = FieldAnalysis(...)  # nothing special needed as it's the default

# you may also specify the edge smoothing value. This is a gaussian filter applied to␣
↪the derivative just for the purposes of finding the min/max derivative.
# This is to ensure the derivative is not caught by some noise. It is usually not␣
↪necessary to change this.
fa = FieldAnalysis(..., edge_smoothing_ratio=0.005)
```

Inflection (Hill)

The inflection point via the Hill function is useful for both flat and FFF beams The fitting of the function is best for low-resolution data, and is thus the default for `DeviceFieldAnalysis`. The Hill function, the sigmoid function, and 4-point non-linear regression belong to a family of logistic equations to fit a dual-curved value. Since these fit a function to the data the resolution problems are eliminated. Some examples can be seen here. The generalized logistic function has helpful visuals as well here.

The function used here is:

$f(x) = A + \frac{B-A}{1+\frac{C}{x}^D}$

where A is the low asymptote value (~0 on the left edge of a field), B is the high asymptote value (~1 for a normalized beam on the left edge), C is the inflection point of the sigmoid curve, and D is the slope of the sigmoid.

The function is fitted to the edge data of the field on each side to return the function. From there, the inflection point, penumbra, and slope can be found.

Note: This method is recommended for low spatial resolution images such as 2D device arrays, where there is very little data at the beam edges. While it can be used for EPID images as well, the fit can have small errors as compared to the direct data. The fit, however, is much better than a linear or even spline interpolation at low resolutions.

```
from pylinac import FieldAnalysis, Edge

fa = FieldAnalysis(..., edge_detection_method=Edge.INFLECTION_HILL)

# you may also specify the hill window. This is the size of the window (as a ratio) to␣
↪use to fit the field edge to the Hill function.
fa = FieldAnalysis(
    ..., edge_detection_method=Edge.INFLECTION_HILL, hill_window_ratio=0.05
)
# i.e. use a 5% field width about the edges to fit the Hill function.
```

Note: When using this method, the fitted Hill function will also be plotted on the image. Further, the exact field edge marker (green x) may not align with the Hill function fit. This is just a rounding issue due to the plotting mechanism. The field edge is really using the Hill fit under the hood.

6.16.5 Normalization

There are 4 options for interpolation: `None`, `GEOMETRIC_CENTER`, `BEAM_CENTER`, and `MAX`. These should be self-explanatory, especially in light of the centering explanations.

```
from pylinac import FieldAnalysis, Normalization

fa = FieldAnalysis(...)
fa.analyze(..., normalization_method=Normalization.BEAM_CENTER)
```

6.16.6 Interpolation

There are 3 options for interpolation: `NONE`, `LINEAR`, and `SPLINE`.

None

A method of `NONE` will obviously apply no interpolation. Other interpolation parameters (see below) are ignored. This is the default method for `DeviceFieldAnalysis`

Note: When plotting the data, if interpolation is None and the data is from a device, the data will be plotted as individual markers (+). If interpolation is applied to device data or it is a DICOM/EPID image, the data is plotted as a line.

Linear

This will apply a linear interpolation to the original data. Along with this, the parameter interpolation_resolution_mm determine the amount of interpolation. E.g. a value of 0.1 will resample the data to get data points 0.1mm apart. This is the default method for `FieldAnalysis`.

```
from pylinac import FieldAnalysis, Interpolation

fa = FieldAnalysis(...)
fa.analyze(..., interpolation=Interpolation.LINEAR, interpolation_resolution_mm=0.1)
```

Spline

This will apply a cubic spline interpolation to the original data. Along with this, the parameter interpolation_resolution_mm determine the amount of interpolation. E.g. a value of 0.1 will resample the data to get data points 0.1mm apart.

```
from pylinac import FieldAnalysis, Interpolation

fa = FieldAnalysis(...)
fa.analyze(..., interpolation=Interpolation.SPLINE, interpolation_resolution_mm=0.1)
```

6.16.7 Protocol Definitions

There are multiple definitions for both flatness and symmetry. Your machine vendor uses certain equations, or your clinic may use a specific definition. Pylinac has a number of built-in definitions which you can use. Know also that you can create your own if you don't like/want to extend these *Creating & Using Custom Protocols*.

None

Technically, you are allowed a "None" protocol (`Protocol.NONE`), which just means nothing beyond the basic field analysis is performed. If you just want the penumbra, distances to CAX, etc, without flatness/symmetry/custom algos then this is for you.

Varian

This is the default protocol if you don't specify one (`Protocol.VARIAN`). Two metrics are included, flatness & symmetry.

```
from pylinac import FieldAnalysis, Protocol

fa = FieldAnalysis(...)
fa.analyze(protocol=Protocol.VARIAN, ...)
...
```

Flatness

Flatness is defined by the variation (difference) across the field values within the field width.

$flatness = 100 * |D_{max} - D_{min}|/(D_{max} + D_{min})$

If the field width is set to, e.g. 80%, then the flatness is calculated over all the values within that 80%. Flatness is a scalar and always positive.

Symmetry

Symmetry is defined as the Point Difference:

$symmetry = 100 * max(|L_{pt} - R_{pt}|)/D_{CAX}$

where L_{pt} and R_{pt} are equidistant from the beam center.

Symmetry is calculated over the specified field width (e.g. 80%) as set in by `analyze()`. Symmetry can be positive or negative. A negative value means the right side is higher. A positive value means the left side is higher.

Elekta

This is specified by passing `protocol=Protocol.ELEKTA` to `analyze`.

```
from pylinac import FieldAnalysis, Protocol

fa = FieldAnalysis(...)
fa.analyze(protocol=Protocol.ELEKTA, ...)
...
```

Flatness

Flatness is defined by the ratio of max/min across the field values within the field width.

$flatness = 100 * D_{max}/D_{min}$

If the field width is set to, e.g. 80%, then the flatness is calculated over all the values within that 80%. Flatness is a scalar and always positive.

Symmetry

Symmetry is defined as the Point Difference Quotient (aka IEC):

$symmetry = 100 * max(|L_{pt}/R_{pt}|, |R_{pt}/L_{pt}|)$

where L_{pt} and R_{pt} are equidistant from the beam center.

Symmetry is calculated over the specified field width (e.g. 80%) as set in by `analyze()`. Symmetry can be positive or negative. A negative value means the right side is higher. A positive value means the left side is higher.

Siemens

This is specified by passing `protocol=Protocol.SIEMENS` to `analyze`.

```
from pylinac import FieldAnalysis, Protocol

fa = FieldAnalysis(...)
fa.analyze(protocol=Protocol.SIEMENS, ...)
...
```

Flatness

Flatness is defined by the variation (difference) across the field values within the field width.

$flatness = 100 * |D_{max} - D_{min}|/(D_{max} + D_{min})$

If the field width is set to, e.g. 80%, then the flatness is calculated over all the values within that 80%. Flatness is a scalar and always positive.

Symmetry

Symmetry is defined as the ratio of area on each side about the CAX:

$symmetry = 100 * (A_{left} - A_{right})/(A_{left} + A_{right})$

Symmetry is calculated over the specified field width (e.g. 80%) as set in by `analyze()`. Symmetry can be positive or negative. A negative value means the right side is higher. A positive value means the left side is higher.

6.16.8 Creating & Using Custom Protocols

Protocols allow the user to perform specific image metric algorithms. This includes things like flatness & symmetry. Depending on the protocol, different methods of determining the flatness/symmetry/whatever exist. Pylinac provides a handful of protocols out of the box, but it is easy to add your own custom algorithms.

To create a custom protocol you must 1) create custom algorithm functions, 2) create a protocol class that 3) inherits from Enum and 4) defines a dictionary with a `calc`, `unit`, and `plot` key/value pair. The `plot` key is optional; it allows you to plot something if you also want to see your special algorithm (e.g. if it used a fitting function and you want to plot the fitted values).

- `calc` should be a function to calculate a specific, singular value such as flatness.
- `unit` should be a string that specifies the unit of `calc`. If it is unitless leave it as an empty string (`''`)
- `plot` is **OPTIONAL** and is a function that can plot something to the profile views (e.g. a fitting function)

The `calc` and `plot` values should be functions with a specific signature as shown in the example below:

```
import enum


# create the custom algorithm functions
# the ``calc`` function must have the following signature
def my_special_flatness(
    profile: SingleProfile, in_field_ratio: float, **kwargs
) -> float:
    # do whatever. Must return a float. ``profile`` will be called twice, once for the␣
↪vertical profile and horizontal profile.
    # the kwargs are passed to ``analyze`` and can be used here for your own purposes (e.
↪g. fitting parameters)
    my_special_value = kwargs.get("funkilicious")
    flatness = profile(...)
    return flatness


# custom plot function for the above flatness function
# This is OPTIONAL
```

(continues on next page)

(continued from previous page)

```
# If you do implement this, it must have the following signature
def my_special_flatness_plot(instance, profile: SingleProfile, axis: plt.Axes) -> None:
    # instance is the FieldAnalysis instance; i.e. it's basically ``self``.
    # do whatever; typically, you will do an axis.plot()
    axis.plot(...)


# custom protocol MUST inherit from Enum
class MySpecialProtocols(enum.Enum):
    # note you can specify several protocols if you wish
    PROTOCOL_1 = {
        # for each protocol, you can specify any number of metrics to calculate. E.g. 2␣
↪symmetry calculations
        "my flatness": {
            "calc": my_special_flatness,
            "unit": "%",
            "plot": my_special_flatness_plot,
        },
        "my symmetry": ...,
        "my other flatness metric": ...,
    }
    PROTOCOL_2 = ...


# proceed as normal
fa = FieldAnalysis(...)
fa.analyze(protocol=MySpecialProtocols.PROTOCOL_1, ...)
...
```

Passing in custom parameters

You may pass custom parameters to these custom algorithms via the analyze method as simple keyword arguments:

```
fa = FieldAnalysis(...)
fa.analyze(..., my_special_variable=42)
```

The parameter will then be passed to the custom functions:

```
def my_special_flatness(
    profile: SingleProfile, in_field_ratio: float, **kwargs
) -> float:
    my_special_value = kwargs.get("my_special_variable")  # 42
    flatness = profile(...)
    return flatness
```

Note: The *SingleProfile* passed to the functions is very powerful and can calculate numerous helpful data for you such as the field edges, minimum/maximum value within the field, and much more. Read the SingleProfile documentation before creating a custom algorithm.

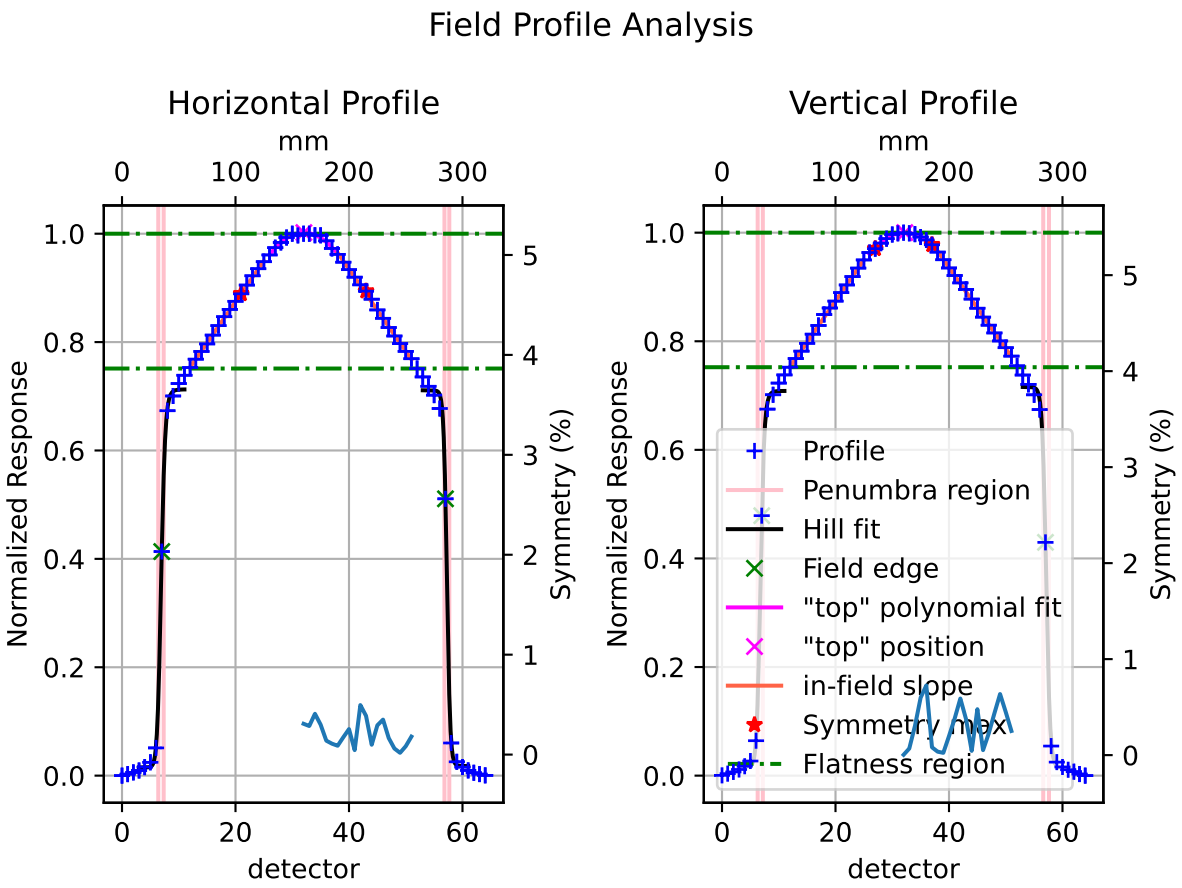
6.16.9 FFF fields

The field analysis module can handle FFF beams, or more specifically, calculating extra metrics associated with FFF fields. These metrics are largely from the NCS-33 pre-publication and include the "top" position, and the slopes of the field on each side.

These metrics are always calculated (even for flat beams), but will be shown in the `results()` output and also on the plotted image of `plot_analyzed_image()` if the `is_FFF` flag is true.

The easiest way to demonstrate this is through the Device demo, which is an FFF field:

```
from pylinac import DeviceFieldAnalysis, Protocol

fa = DeviceFieldAnalysis.from_demo_image()
fa.analyze(protocol=Protocol.VARIAN, is_FFF=True)
fa.plot_analyzed_image()
```



"Top" metric

The "top" metric is the fitted position of the peak of a FFF beam. It uses the central region of the field as specified by the `slope_exclusion_ratio`. E.g. if the value is 0.3 it will use the central 30% field width.

The central region is fitted to a 2nd order polynomial and then the max of the polynomial is found. That value is the "top" position. This helps to account for noise in the profile.

When printing results for an FFF beam there will be a section like so:

```
'Top' vertical distance from CAX: 1.3mm
'Top' horizontal distance from CAX: 0.6mm
'Top' vertical distance from beam center: 1.7mm
'Top' horizontal distance from beam center: 0.3mm
```

Field slope

For FFF beams, an additional metric is calculated: the slope of each side of the field. Since traditional flatness algorithms aren't tuned for FFF beams they can be noisy or non-sensible. By measuring the slope of each side of the field the flatness can be measured more accurately (as a slope) for trending and consistency purposes.

The slope is calculated in the regions between the field width edges and the slope exclusion ratio. E.g. a field width ratio of 0.8 and a slope exclusion ratio of 0.4 will mean that the regions between +/-0.4 (0.8/2) from the CAX to +/-0.2 (0.4/2) will be used to fit linear regressions.

When printing results for an FFF beam there will be a section like so:

```
Top slope: 0.292%/mm
Bottom slope: -0.291%/mm
Left slope: 0.295%/mm
Right slope: -0.296%/mm
```

6.16.10 Accessing data

Changed in version 3.0.

Using the module in your own scripts? While the analysis results can be printed out, if you intend on using them elsewhere (e.g. in an API), they can be accessed the easiest by using the *results_data()* method which returns a *FieldResult* instance.

Note: While the pylinac tooling may change under the hood, this object should remain largely the same and/or expand. Thus, using this is more stable than accessing attrs directly.

You can access most data you get from `results()`:

```
fa = FieldAnalysis(...)
fa.analyze(...)
data = fa.results_data()

data.top_penumbra_mm
data.beam_center_to_left_mm
```

You may also access protocol data in the `protocol_results` dictionary. These results must be in a dictionary because the protocol names and fields are dynamic and not known a priori.

```
data.protocol_results["flatness_vertical"]
data.protocol_results["symmetry_horizontal"]
```

The keys of this dict are defined by the protocol names. Using the example from the *Creating & Using Custom Protocols* section, we would access that custom protocol data as:

```
data.protocol_results["my flatness_vertical"]
data.protocol_results["my flatness_horizontal"]
```

because the protocol name was `my flatness`.

6.16.11 Algorithm

There is little of a true "algorithm" in `field_analysis` other than analyzing profiles. Thus, this section is more terminology and notekeeping.

Allowances

- The image can be any size.
- The image can be EPID (actually just DICOM) or a 2D device array file.
- The image can be either inversion (Radiation is dark or light).
- The image can be off-center.

Restrictions

- The module is only meant for photon analysis at the moment (there are sometimes different equations for electrons for the same definition name).
- Analysis is limited to normal/parallel directions. Thus if the image is rotated there is no way to account for it other than rotating the image before analysis.

Analysis

- **Extract profiles** - With the positions given, profiles are extracted and analyzed according to the method specified (see *Protocol Definitions*). For symmetry calculations that operate around the CAX, the CAX must first be determined, which is the center of the FWHM of the profile.

6.16.12 API Documentation

Main classes

These are the classes a typical user may interface with.

class pylinac.field_analysis.**FieldAnalysis**(*path: str | BinaryIO*, *filter: int | None = None*, *image_kwargs: dict | None = None*)

Bases: `object`

Class for analyzing the various parameters of a radiation image, most commonly an open image from a linac.

Parameters

path
The path to the image.

filter
If None, no filter is applied. If an int, a median filter of size n pixels is applied. Generally, a good idea. Default is None for backwards compatibility.

image: image.ImageLike

vert_profile: *SingleProfile*

horiz_profile: *SingleProfile*

classmethod from_demo_image()
Load the demo image into an instance.

static run_demo() → None
Run the Field Analysis demo by loading the demo image, print results, and plot the profiles.

analyze(*protocol:* Protocol = *Protocol.VARIAN*, *centering:* Centering | *str* = *Centering.BEAM_CENTER*, *vert_position: float = 0.5*, *horiz_position: float = 0.5*, *vert_width: float = 0*, *horiz_width: float = 0*, *in_field_ratio: float = 0.8*, *slope_exclusion_ratio: float = 0.2*, *invert: bool = False*, *is_FFF: bool = False*, *penumbra: tuple[float, float] = (20, 80)*, *interpolation:* Interpolation | *str* | *None* = *Interpolation.LINEAR*, *interpolation_resolution_mm: float = 0.1*, *ground: bool = True*, *normalization_method: Normalization* | *str* = *Normalization.BEAM_CENTER*, *edge_detection_method:* Edge | *str* = *Edge.INFLECTION_DERIVATIVE*, *edge_smoothing_ratio: float = 0.003*, *hill_window_ratio: float = 0.15*, ***kwargs*) → None
Analyze the image to determine parameters such as field edges, penumbra, and/or flatness & symmetry.

Parameters

protocol
[*Protocol*] The analysis protocol. See *Protocol Definitions* for equations.

centering
[Centering] The profile extraction position technique. Beam center will determine the beam center and take profiles through the middle. Geometric center will simply take profiles centered about the image in both axes. Manual will use the values of *vert_position* and *horiz_position* as the position. See *Centering*.

vert_position
The distance ratio of the image to sample. E.g. at the default of 0.5 the profile is extracted in the middle of the image. 0.0 is at the left edge of the image and 1.0 is at the right edge of the image.

Note: This value only applies when centering is MANUAL.

horiz_position
The distance ratio of the image to sample. E.g. at the default of 0.5 the profile is extracted in the middle of the image. 0.0 is at the top edge of the image and 1.0 is at the bottom edge of the image.

Note: This value only applies when centering is MANUAL.

vert_width
The width ratio of the image to sample. E.g. at the default of 0.0 a 1 pixel wide profile is extracted. 0.0 would be 1 pixel wide and 1.0 would be the vertical image width.

horiz_width
The width ratio of the image to sample. E.g. at the default of 0.0 a 1 pixel wide profile is extracted. 0.0 would be 1 pixel wide and 1.0 would be the horizontal image width.

in_field_ratio
The ratio of the field width to use for protocol values. E.g. 0.8 means use the 80% field width.

slope_exclusion_ratio
This is the ratio of the field to use to 1) calculate the "top" of an FFF field as well as 2) exclude from the "slope" calculation of each side of the field. Alternatively, this also defines the area to use for the slope calculation. E.g. an *in_field_ratio* of 0.8 and *slope_exclusion_ratio* of 0.2 means the central 20% of the field is used to fit and calculate the "top", while the region on either side of the central 20% between the central 80% is used to calculate a slope on either side using linear regression.

Note: While the "top" is always calculated, it will not be displayed in plots if the *is_FFF* parameter is false.

---

invert
Whether to invert the image. Setting this to True will override the default inversion. This is useful if pylinac's automatic inversion is incorrect.

is_FFF
This is a flag to display the "top" calculation and slopes on either side of the field.

penumbra
A tuple of (lower, higher) % of the penumbra to calculate. E.g. (20, 80) will calculate the penumbra width at 20% and 80%.

Note: The exact height of the penumbra depends on the edge detection method. E.g. FWHM will result in calculating penumbra at 20/80% of the field max, but if something like inflection is used, the penumbra height will be 20/50*100*inflection height and 80/50*100*inflection height.

---

ground
Whether to ground the profile (set min value to 0). Helpful most of the time.

interpolation
Interpolation technique to use. See *Interpolation*.

interpolation_resolution_mm
The resolution that the interpolation will scale to. E.g. if the native dpmm is 2 and the resolution is set to 0.1mm the data will be interpolated to have a new dpmm of 10 (1/0.1).

normalization_method
How to pick the point to normalize the data to. See *Normalization*.

edge_detection_method
The method by which to detect the field edge. FWHM is reasonable most of the time except for FFF beams. Inflection-derivative will use the max gradient to determine the field edge. Note that this may not be the 50% height. In fact, for FFF beams it shouldn't be. Inflection methods are better for FFF and other unusual beam shapes. See *Edge detection*.

edge_smoothing_ratio
The ratio of the length of the values to use as the sigma for a Gaussian filter applied before searching

for the inflection. E.g. 0.005 with a profile of 1000 points will result in a sigma of 5. This helps make the inflection point detection more robust to noise. Increase for noisy data.

hill_window_ratio
The ratio of the field size to use as the window to fit the Hill function. E.g. 0.2 will using a window centered about each edge with a width of 20% the size of the field width. Only applies when the edge detection is `INFLECTION_HILL`.

kwargs
Use these to pass parameters to custom protocol functions. See *Creating & Using Custom Protocols*.

`results`(*as_str=True*) → str

Get the results of the analysis.

Parameters

as_str
If True, return a simple string. If False, return a list of each line of text.

`results_data`(*as_dict: bool = False*) → *FieldResult* | dict

Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

`publish_pdf`(*filename: str*, *notes: str | list = None*, *open_file: bool = False*, *metadata: dict = None*, *logo: Path | str | None = None*) → None

Publish (print) a PDF containing the analysis, images, and quantitative results.

Parameters

filename
[(str, file-like object}] The file to write the results to.

notes
[str, list of strings] Text; if str, prints single line. If list of strings, each list item is printed on its own line.

open_file
[bool] Whether to open the file using the default program after creation.

metadata
[dict] Extra stream to be passed and shown in the PDF. The key and value will be shown with a colon. E.g. passing {'Author': 'James', 'Unit': 'TrueBeam'} would result in text in the PDF like: ———— Author: James Unit: TrueBeam ————

logo: Path, str
A custom logo to use in the PDF report. If nothing is passed, the default pylinac logo is used.

`plot_analyzed_image`(*show: bool = True*, *grid: bool = True*, *split_plots: bool = False*, ***plt_kwargs*) → tuple[list[Figure], list[str]]

Plot the analyzed image. Shows parameters such as flatness & symmetry.

Parameters

show
Whether to show the plot when called.

grid
Whether to show a grid on the profile plots

split_plots
[bool] Whether to plot the image and profiles on individual figures. Useful for saving individual plots.

plt_kwargs
[dict] Keyword args passed to the plt.figure() method. Allows one to set things like figure size.

save_analyzed_image(*filename: None | str | Path | BinaryIO = None*, *split_plots: bool = False*, *to_streams: bool = False*, ***kwargs*) → list[str] | dict[str, BinaryIO] | None

Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()

Parameters

split_plots: bool
If split_plots is True, multiple files will be created that append a name. E.g. *my_file.png* will become *my_file_image.png*, *my_file_vertical.png*, etc. If to_streams is False, a list of new filenames will be returned

to_streams: bool
This only matters if split_plots is True. If both of these are true, multiple streams will be created and returned as a dict.

class pylinac.field_analysis.**FieldResult**(*protocol: Protocol*, *protocol_results: dict*, *centering_method: Centering*, *normalization_method: Normalization*, *interpolation_method: Interpolation*, *edge_detection_method: Edge*, *top_penumbra_mm: float*, *bottom_penumbra_mm: float*, *left_penumbra_mm: float*, *right_penumbra_mm: float*, *geometric_center_index_x_y: tuple[float, float]*, *beam_center_index_x_y: tuple[float, float]*, *field_size_vertical_mm: float*, *field_size_horizontal_mm: float*, *beam_center_to_top_mm: float*, *beam_center_to_bottom_mm: float*, *beam_center_to_left_mm: float*, *beam_center_to_right_mm: float*, *cax_to_top_mm: float*, *cax_to_bottom_mm: float*, *cax_to_left_mm: float*, *cax_to_right_mm: float*, *top_position_index_x_y: tuple[float, float]*, *top_horizontal_distance_from_cax_mm: float*, *top_vertical_distance_from_cax_mm: float*, *top_horizontal_distance_from_beam_center_mm: float*, *top_vertical_distance_from_beam_center_mm: float*, *left_slope_percent_mm: float*, *right_slope_percent_mm: float*, *top_slope_percent_mm: float*, *bottom_slope_percent_mm: float*, *top_penumbra_percent_mm: float = 0*, *bottom_penumbra_percent_mm: float = 0*, *left_penumbra_percent_mm: float = 0*, *right_penumbra_percent_mm: float = 0*, *central_roi_mean: float = 0*, *central_roi_max: float = 0*, *central_roi_std: float = 0*, *central_roi_min: float = 0*)

Bases: *DeviceResult*

This class should not be called directly. It is returned by the `results_data()` method. It is a dataclass under the hood and thus comes with all the dunder magic.

Use the following attributes as normal class attributes.

In addition to the below attrs, custom protocol data will also be attached under the `protocol_results` attr as a dictionary with keys like so: `<protocol name>_vertical` and `<protocol name>_horizontal` for each protocol item.

E.g. a protocol item of `symmetry` will result in `symmetry_vertical` and `symmetry_horizontal`.

central_roi_mean: float = 0

central_roi_max: float = 0

central_roi_std: float = 0

central_roi_min: float = 0

class pylinac.field_analysis.**DeviceResult**(*protocol: 'Protocol'*, *protocol_results: 'dict'*, *centering_method: 'Centering'*, *normalization_method: 'Normalization'*, *interpolation_method: 'Interpolation'*, *edge_detection_method: 'Edge'*, *top_penumbra_mm: 'float'*, *bottom_penumbra_mm: 'float'*, *left_penumbra_mm: 'float'*, *right_penumbra_mm: 'float'*, *geometric_center_index_x_y: 'tuple[float, float]'*, *beam_center_index_x_y: 'tuple[float, float]'*, *field_size_vertical_mm: 'float'*, *field_size_horizontal_mm: 'float'*, *beam_center_to_top_mm: 'float'*, *beam_center_to_bottom_mm: 'float'*, *beam_center_to_left_mm: 'float'*, *beam_center_to_right_mm: 'float'*, *cax_to_top_mm: 'float'*, *cax_to_bottom_mm: 'float'*, *cax_to_left_mm: 'float'*, *cax_to_right_mm: 'float'*, *top_position_index_x_y: 'tuple[float, float]'*, *top_horizontal_distance_from_cax_mm: 'float'*, *top_vertical_distance_from_cax_mm: 'float'*, *top_horizontal_distance_from_beam_center_mm: 'float'*, *top_vertical_distance_from_beam_center_mm: 'float'*, *left_slope_percent_mm: 'float'*, *right_slope_percent_mm: 'float'*, *top_slope_percent_mm: 'float'*, *bottom_slope_percent_mm: 'float'*, *top_penumbra_percent_mm: 'float' = 0*, *bottom_penumbra_percent_mm: 'float' = 0*, *left_penumbra_percent_mm: 'float' = 0*, *right_penumbra_percent_mm: 'float' = 0*)

Bases: *ResultBase*

protocol: *Protocol*

protocol_results: dict

centering_method: *Centering*

normalization_method: Normalization

interpolation_method: *Interpolation*

edge_detection_method: *Edge*

top_penumbra_mm: float

bottom_penumbra_mm: float

left_penumbra_mm: float

right_penumbra_mm: float

geometric_center_index_x_y: tuple[float, float]

beam_center_index_x_y: tuple[float, float]

field_size_vertical_mm: float

field_size_horizontal_mm: float

beam_center_to_top_mm: float

beam_center_to_bottom_mm: float

beam_center_to_left_mm: float

beam_center_to_right_mm: float

cax_to_top_mm: float

cax_to_bottom_mm: float

cax_to_left_mm: float

cax_to_right_mm: float

top_position_index_x_y: tuple[float, float]

top_horizontal_distance_from_cax_mm: float

top_vertical_distance_from_cax_mm: float

top_horizontal_distance_from_beam_center_mm: float

top_vertical_distance_from_beam_center_mm: float

left_slope_percent_mm: float

right_slope_percent_mm: float

top_slope_percent_mm: float

bottom_slope_percent_mm: float

top_penumbra_percent_mm: float = 0

bottom_penumbra_percent_mm: float = 0

left_penumbra_percent_mm: float = 0

right_penumbra_percent_mm: float = 0

class pylinac.field_analysis.**Device**(*value*)

Bases: Enum

2D array device Enum.

PROFILER = {'detector spacing (mm)': 5, 'device': <class 'pylinac.core.io.SNCProfiler'>}

class pylinac.field_analysis.**Protocol**(*value*)

Bases: Enum

Protocols to analyze additional metrics of the field. See *Protocol Definitions*

NONE = {}

VARIAN = {'flatness': {'calc': <function flatness_dose_difference>, 'plot': <function plot_flatness>, 'unit': '%'}, 'symmetry': {'calc': <function symmetry_point_difference>, 'plot': <function plot_symmetry_point_difference>, 'unit': '%'}}

SIEMENS = {'flatness': {'calc': <function flatness_dose_difference>, 'plot': <function plot_flatness>, 'unit': ''}, 'symmetry': {'calc': <function symmetry_area>, 'plot': <function plot_symmetry_area>, 'unit': ''}}

ELEKTA = {'flatness': {'calc': <function flatness_dose_ratio>, 'plot': <function plot_flatness>, 'unit': ''}, 'symmetry': {'calc': <function symmetry_pdq_iec>, 'plot': <function plot_symmetry_pdq>, 'unit': ''}}

class pylinac.field_analysis.**Centering**(*value*)

Bases: Enum

See *Centering*

MANUAL = 'Manual'

BEAM_CENTER = 'Beam center'

GEOMETRIC_CENTER = 'Geometric center'

class pylinac.field_analysis.**Interpolation**(*value*)

Bases: Enum

Interpolation Enum

NONE = None

LINEAR = 'Linear'

SPLINE = 'Spline'

class pylinac.field_analysis.**Edge**(*value*)

Bases: Enum

Edge detection Enum

FWHM = 'FWHM'

INFLECTION_DERIVATIVE = 'Inflection Derivative'

INFLECTION_HILL = 'Inflection Hill'

Supporting Classes

You generally won't have to interface with these unless you're doing advanced behavior.

pylinac.field_analysis.**flatness_dose_difference**(*profile:* SingleProfile, *in_field_ratio: float = 0.8*, ***kwargs*) → float

The Varian specification for calculating flatness. See *Varian*.

pylinac.field_analysis.**flatness_dose_ratio**(*profile:* SingleProfile, *in_field_ratio: float = 0.8*, ***kwargs*) → float

The Elekta specification for calculating flatness. See *Elekta*.

pylinac.field_analysis.**symmetry_point_difference**(*profile:* SingleProfile, *in_field_ratio: float*, ***kwargs*) → float

Calculation of symmetry by way of point difference equidistant from the CAX. See *Varian*.

A negative value means the right side is higher. A positive value means the left side is higher.

pylinac.field_analysis.**symmetry_area**(*profile:* SingleProfile, *in_field_ratio: float*, ***kwargs*) → float

Ratio of the area under the left and right profile segments. See *Siemens*.

A negative value indicates the right side is higher; a positive value indicates the left side is higher.

pylinac.field_analysis.**symmetry_pdq_iec**(*profile:* SingleProfile, *in_field_ratio: float*, ***kwargs*) → float

Symmetry calculation by way of PDQ IEC. See *Elekta*.

A negative value means the right side is higher. A positive value means the left side is higher.

6.17 Core Modules

The following is the API documentation for the core modules of pylinac. These can be used directly, or as the base for mixin classes or methods.

6.17.1 Image Module

This module holds classes for image loading and manipulation.

pylinac.core.image.**equate_images**(*image1:* DicomImage | ArrayImage | FileImage | LinacDicomImage, *image2:* DicomImage | ArrayImage | FileImage | LinacDicomImage) → tuple[*DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*, *DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*]

Crop and resize two images to make them:

- The same pixel dimensions
- The same DPI

The usefulness of the function comes when trying to compare images from different sources. The best example is calculating gamma on a machine log fluence and EPID image. The physical and pixel dimensions must be normalized, the SID normalized

Parameters

image1
[{*ArrayImage*, *DicomImage*, *FileImage*}] Must have DPI and SID.

image2
[{*ArrayImage*, *DicomImage*, *FileImage*}] Must have DPI and SID.

Returns

image1
[*ArrayImage*] The first image equated.

image2
[*ArrayImage*] The second image equated.

pylinac.core.image.**is_image**(*path: str | io.BytesIO | ImageLike | np.ndarray*) → bool
Determine whether the path is a valid image file.

Returns

bool

pylinac.core.image.**retrieve_image_files**(*path: str*) → list[str]
Retrieve the file names of all the valid image files in the path.

Returns

list
Contains strings pointing to valid image paths.

pylinac.core.image.**load**(*path: str | Path | ImageLike | np.ndarray | BinaryIO*, ***kwargs*) → ImageLike
Load a DICOM image, JPG/TIF/BMP image, or numpy 2D array.

Parameters

path
[str, file-object] The path to the image file or data stream or array.

kwargs
See *FileImage*, *DicomImage*, or *ArrayImage* for keyword arguments.

Returns

:*FileImage*, *ArrayImage*, or *DicomImage*
Return type depends on input image.

Examples

Load an image from a file and then apply a filter:

```
>>> from pylinac.core.image import load
>>> my_image = r"C:\QA\image.tif"
>>> img = load(my_image)  # returns a FileImage
>>> img.filter(5)
```

Loading from an array is just like loading from a file:

```
>>> arr = np.arange(36).reshape(6, 6)
>>> img = load(arr)  # returns an ArrayImage
```

pylinac.core.image.**load_url**(*url: str*, *progress_bar: bool = True*, ***kwargs*) → *DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*

Load an image from a URL.

Parameters

url
[str] A string pointing to a valid URL that points to a file.

Note: For some images (e.g. Github), the raw binary URL must be used, not simply the basic link.

---

progress_bar: bool
Whether to display a progress bar of download status.

pylinac.core.image.**load_multiples**(*image_file_list: Sequence*, *method: str = 'mean'*, *stretch_each: bool = True*, ***kwargs*) → *DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*

Combine multiple image files into one superimposed image.

Parameters

image_file_list
[list] A list of the files to be superimposed.

method
[{'mean', 'max', 'sum'}] A string specifying how the image values should be combined.

stretch_each
[bool] Whether to normalize the images being combined by stretching their high/low values to the same values across images.

kwargs :
Further keyword arguments are passed to the load function and stretch function.

Examples

Load multiple images:

```
>>> from pylinac.core.image import load_multiples
>>> paths = ['starshot1.tif', 'starshot2.tif']
>>> superimposed_img = load_multiples(paths)
```

class pylinac.core.image.**BaseImage**(*path: str | Path | BytesIO | ImageLike | np.ndarray | BufferedReader*)

Bases: `object`

Base class for the Image classes.

Attributes

path
[str] The path to the image file.

array
[numpy.ndarray] The actual image pixel array.

Parameters

path
[str] The path to the image.

classmethod from_multiples(*filelist: list[str]*, *method: str = 'mean'*, *stretch: bool = True*, ***kwargs*) → *DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*

Load an instance from multiple image items. See *load_multiples()*.

property center: *Point*

Return the center position of the image array as a Point. Even-length arrays will return the midpoint between central two indices. Odd will return the central index.

property physical_shape: float, float

The physical size of the image in mm.

date_created(*format: str = '%A, %B %d, %Y'*) → str

The date the file was created. Tries DICOM data before falling back on OS timestamp. The method use one or more inputs of formatted code, where % means a placeholder and the letter the time unit of interest. For a full description of the several formatting codes see strftime() documentation.

Parameters

format
[str] %A means weekday full name, %B month full name, %d day of the month as a zero-padded decimal number and %Y year with century as a decimal number.

Returns

str
The date the file was created.

plot(*ax: plt.Axes = None*, *show: bool = True*, *clear_fig: bool = False*, *metric_kwargs: dict | None = None*, ***kwargs*) → plt.Axes

Plot the image.

Parameters

ax
[matplotlib.Axes instance] The axis to plot the image to. If None, creates a new figure.

show
[bool] Whether to actually show the image. Set to false when plotting multiple items.

clear_fig
[bool] Whether to clear the prior items on the figure before plotting.

metric_kwargs
[dict] kwargs passed to the metric plot method.

kwargs
kwargs passed to plt.imshow()

plot_metrics(*show: bool = True*) → list[figure]

Plot any additional figures from the metrics.

Returns a list of figures of the metrics. These metrics are not drawn on the original image but rather are something complete separate. E.g. a profile plot or a histogram of the metric.

filter(*size: float | int = 0.05*, *kind: str = 'median'*) → None

Filter the profile in place.

Parameters

size
[int, float] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

crop(*pixels: int = 15*, *edges: tuple[str, ...] = ('top', 'bottom', 'left', 'right')*) → None

Removes pixels on all edges of the image in-place.

Parameters

pixels
[int] Number of pixels to cut off all sides of the image.

edges
[tuple] Which edges to remove from. Can be any combination of the four edges.

`flipud`() → None
Flip the image array upside down in-place. Wrapper for np.flipud()

`fliplr`() → None
Flip the image array upside down in-place. Wrapper for np.fliplr()

`invert`() → None
Invert (imcomplement) the image.

`bit_invert`() → None
Invert the image bit-wise

`roll`(*direction: str = 'x'*, *amount: int = 1*) → None
Roll the image array around in-place. Wrapper for np.roll().

Parameters

direction
[{'x', 'y'}] The axis to roll over.

amount
[int] The amount of elements to roll over.

`rot90`(*n: int = 1*) → None
Wrapper for numpy.rot90; rotate the array by 90 degrees CCW n times.

`rotate`(*angle: float*, *mode: str = 'edge'*, **args*, ***kwargs*)
Rotate the image counter-clockwise. Simple wrapper for scikit-image. See https://scikit-image.org/docs/stable/api/skimage.transform.html#skimage.transform.rotate. All parameters are passed to that function.

`threshold`(*threshold: float*, *kind: str = 'high'*) → None
Apply a high- or low-pass threshold filter.

Parameters

threshold
[int] The cutoff value.

kind
[str] If `high` (default), will apply a high-pass threshold. All values above the cutoff are left as-is. Remaining points are set to 0. If `low`, will apply a low-pass threshold.

`as_binary`(*threshold: int*) → *DicomImage* | *ArrayImage* | *FileImage* | *LinacDicomImage*
Return a binary (black & white) image based on the given threshold.

Parameters

threshold
[int, float] The threshold value. If the value is above or equal to the threshold it is set to 1, otherwise to 0.

Returns

ArrayImage

dist2edge_min(*point:* Point | *tuple*) → float

Calculates distance from given point to the closest edge.

Parameters

point : geometry.Point, tuple

Returns

float

ground() → float

Ground the profile in place such that the lowest value is 0.

Note: This will also "ground" profiles that are negative or partially-negative. For such profiles, be careful that this is the behavior you desire.

Returns

float
The amount subtracted from the image.

normalize(*norm_val: str | float | None = None*) → None

Normalize the profile to the given value.

Parameters

value
[number or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

check_inversion(*box_size: int = 20*, *position: float, float = (0.0, 0.0)*) → None

Check the image for inversion by sampling the 4 image corners. If the average value of the four corners is above the average pixel value, then it is very likely inverted.

Parameters

box_size
[int] The size in pixels of the corner box to detect inversion.

position
[2-element sequence] The location of the sampling boxes.

`check_inversion_by_histogram`(*percentiles: float, float, float = (5, 50, 95)*) → bool

Check the inversion of the image using histogram analysis. The assumption is that the image is mostly background-like values and that there is a relatively small amount of dose getting to the image (e.g. a picket fence image). This function looks at the distance from one percentile to another to determine if the image should be inverted.

Parameters

percentiles
[3-element tuple] The 3 percentiles to compare. Default is (5, 50, 95). Recommend using (x, 50, y). To invert the other way (where pixel value is *decreasing* with dose, reverse the percentiles, e.g. (95, 50, 5).

Returns

bool: Whether an inversion was performed.

`gamma`(*comparison_image:* DicomImage | ArrayImage | FileImage | LinacDicomImage, *doseTA: float = 1*, *distTA: float = 1*, *threshold: float = 0.1*, *ground: bool = True*, *normalize: bool = True*) → ndarray

Calculate the gamma between the current image (reference) and a comparison image.

New in version 1.2.

The gamma calculation is based on Bakai et al eq.6, which is a quicker alternative to the standard Low gamma equation.

Parameters

comparison_image
[{*ArrayImage*, *DicomImage*, or *FileImage*}] The comparison image. The image must have the same DPI/DPMM to be comparable. The size of the images must also be the same.

doseTA
[int, float] Dose-to-agreement in percent; e.g. 2 is 2%.

distTA
[int, float] Distance-to-agreement in mm.

threshold
[float] The dose threshold percentage of the maximum dose, below which is not analyzed. Must be between 0 and 1.

ground
[bool] Whether to "ground" the image values. If true, this sets both datasets to have the minimum value at 0. This can fix offset errors in the data.

normalize
[bool] Whether to normalize the images. This sets the max value of each image to the same value.

Returns

gamma_map
[numpy.ndarray] The calculated gamma map.

See Also

equate_images()

compute(*metrics: list[MetricBase] | MetricBase*) → Any | dict[str, Any]

Compute the given metrics on the image.

This can be called multiple times to compute different metrics. Metrics are appended on each call. This allows for modification of the image between metric calls as well as the ability to compute different metrics on the same image that might depend on earlier metrics.

Metrics are both returned and stored in the `metrics` attribute. The `metrics` attribute will store all metrics every calculated. The metrics returned are only those passed in the `metrics` argument.

Parameters

metrics
[list[MetricBase] | MetricBase] The metric(s) to compute.

class pylinac.core.image.**XIM**(*file_path: str | Path*, *read_pixels: bool = True*)

Bases: *BaseImage*

A class to open, read, and/or export an .xim image, Varian's custom image format which is 99.999% PNG

This had inspiration from a number of places: - https://gist.github.com/1328/7da697c71f9c4ef12e1e - https://medium.com/@duhroach/how-png-works-f1174e3cc7b7 - https://www.mathworks.com/matlabcentral/answers/419228-how-to-write-for-loop-and-execute-data - https://www.w3.org/TR/PNG-Filters.html - https://bitbucket.org/dmoderesearchtools/ximreader/src/master/

Parameters

file_path
The path to the file of interest.

read_pixels
Whether to read and parse the pixel information. Doing so is quite slow. Set this to false if, e.g., you are searching for images only via tags or doing a pre-filtering of image selection.

array: np.ndarray

properties: dict

property dpmm: float
The dots/mm value of the XIM images. The value appears to be in cm in the file.

save_as(*file: str*, *format: str | None = None*) → None

Save the image to a NORMAL format. PNG is highly suggested. Accepts any format supported by Pillow. Ironically, an equivalent PNG image (w/ metadata) is ~50% smaller than an .xim image.

> **Warning:** Any format other than PNG will not include the properties included in the .xim image!

Parameters

file
The file to save the image to. E.g. my_xim.png

format
The format to save the image as. Uses the Pillow logic, which will infer the format if the file name has one.

class pylinac.core.image.**DicomImage**(*path: str | Path | BytesIO | BufferedReader*, ***, *dtype: np.dtype | None = None*, *dpi: float = None*, *sid: float = None*, *sad: float = 1000*, *raw_pixels: bool = False*)

Bases: *BaseImage*

An image from a DICOM RTImage file.

Attributes

metadata
[pydicom Dataset] The dataset of the file as returned by pydicom without pixel data.

Parameters

path
[str, file-object] The path to the file or the data stream.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

dpi
[int, float] The dots-per-inch of the image, defined at isocenter.

> **Note:** If a DPI tag is found in the image, that value will override the parameter, otherwise this one will be used.

sid
[int, float] The Source-to-Image distance in mm.

sad
[float] The Source-to-Axis distance in mm.

raw_pixels
[bool] Whether to apply pixel intensity correction to the DICOM data. Typically, Rescale Slope, Rescale Intercept, and other tags are included and meant to be applied to the raw pixel data, which is potentially

compressed. If True, no correction will be applied. This is typically used for scenarios when you want to match behavior to older or different software.

classmethod from_dataset(*dataset: Dataset*)

Create a DICOM image instance from a pydicom Dataset.

save(*filename: str | Path*) → str | Path

Save the image instance back out to a .dcm file.

Parameters

filename
[str, Path] The filename to save the DICOM file as.

Returns

A string pointing to the new filename.

property z_position: float

The z-position of the slice. Relevant for CT and MR images.

property slice_spacing: float

Determine the distance between slices. The spacing can be greater than the slice thickness (i.e. gaps). Uses the absolute version as it can apparently be negative: https://dicom.innolitics.com/ciods/nm-image/nm-reconstruction/00180088

This attempts to use the slice spacing attr and if it doesn't exist, use the slice thickness attr

property sid: float

The Source-to-Image in mm.

property sad: float

The source to axis (iso) in mm

property dpi: float

The dots-per-inch of the image, defined at isocenter.

property dpmm: float

The Dots-per-mm of the image, defined at isocenter. E.g. if an EPID image is taken at 150cm SID, the dpmm will scale back to 100cm.

property cax: *Point*

The position of the beam central axis. If no DICOM translation tags are found then the center is returned. Uses this tag: https://dicom.innolitics.com/ciods/rt-beams-delivery-instruction/rt-beams-delivery-instruction/00741020/00741030/3002000d

class pylinac.core.image.**LinacDicomImage**(*path: str | Path | BinaryIO*, *use_filenames: bool = False*, *axes_precision: int | None = None*, ***kwargs*)

Bases: *DicomImage*

DICOM image taken on a linac. Also allows passing of gantry/coll/couch values via the filename.

Parameters

path
[str, file-object] The path to the file or the data stream.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

dpi
[int, float] The dots-per-inch of the image, defined at isocenter.

Note: If a DPI tag is found in the image, that value will override the parameter, otherwise this one will be used.

---

sid
[int, float] The Source-to-Image distance in mm.

sad
[float] The Source-to-Axis distance in mm.

raw_pixels
[bool] Whether to apply pixel intensity correction to the DICOM data. Typically, Rescale Slope, Rescale Intercept, and other tags are included and meant to be applied to the raw pixel data, which is potentially compressed. If True, no correction will be applied. This is typically used for scenarios when you want to match behavior to older or different software.

property gantry_angle: float
Gantry angle of the irradiation.

property collimator_angle: float
Collimator angle of the irradiation.

property couch_angle: float
Couch angle of the irradiation.

class pylinac.core.image.**FileImage**(*path: str | Path | BinaryIO*, *, *dpi: float | None = None*, *sid: float | None = None*, *dtype: np.dtype | None = None*)

Bases: *BaseImage*

An image from a "regular" file (.tif, .jpg, .bmp).

Attributes

info
[dict] The info dictionary as generated by Pillow.

sid
[float] The SID value as passed in upon construction.

Parameters

path
[str, file-object] The path to the file or a data stream.

dpi
[int, float] The dots-per-inch of the image, defined at isocenter.

Note: If a DPI tag is found in the image, that value will override the parameter, otherwise this one will be used.

---

sid
[int, float] The Source-to-Image distance in mm.

dtype
[numpy.dtype] The data type to cast the array as. If None, will use the datatype stored in the file. If the file is multi-channel (e.g. RGB), it will be converted to int32

property dpi: float | None
The dots-per-inch of the image, defined at isocenter.

property dpmm: float | None
The Dots-per-mm of the image, defined at isocenter. E.g. if an EPID image is taken at 150cm SID, the dpmm will scale back to 100cm.

class pylinac.core.image.**ArrayImage**(*array: ndarray*, *, *dpi: float | None = None*, *sid: float | None = None*, *dtype=None*)

Bases: *BaseImage*

An image constructed solely from a numpy array.

Parameters

array
[numpy.ndarray] The image array.

dpi
[int, float] The dots-per-inch of the image, defined at isocenter.

Note: If a DPI tag is found in the image, that value will override the parameter, otherwise this one will be used.

---

sid
[int, float] The Source-to-Image distance in mm.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

property dpmm: float | None
The Dots-per-mm of the image, defined at isocenter. E.g. if an EPID image is taken at 150cm SID, the dpmm will scale back to 100cm.

property dpi: float | None

The dots-per-inch of the image, defined at isocenter.

class pylinac.core.image.DicomImageStack(*folder: str | Path*, *dtype: np.dtype | None = None*, *min_number: int = 39*, *check_uid: bool = True*, *raw_pixels: bool = False*)

Bases: LazyDicomImageStack

A class that loads and holds a stack of DICOM images (e.g. a CT dataset). The class can take a folder or zip file and will read CT images. The images must all be the same size. Supports indexing to individual images.

Attributes

images
[list] Holds instances of *DicomImage*. Can be accessed via index; i.e. self[0] == self.images[0].

Examples

Load a folder of Dicom images >>> from pylinac import image >>> img_folder = r"folder/qa/cbct/june" >>> dcm_stack = image.DicomImageStack(img_folder) # loads and sorts the images >>> dcm_stack.plot(3) # plot the 3rd image

Load a zip archive >>> img_folder_zip = r"archive/qa/cbct/june.zip" # save space and zip your CBCTs >>> dcm_stack = image.DicomImageStack.from_zip(img_folder_zip)

Load as a certain data type >>> dcm_stack_uint32 = image.DicomImageStack(img_folder, dtype=np.uint32)

Load a folder with DICOM CT images.

Parameters

folder
[str] Path to the folder.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

classmethod from_zip(*zip_path: str | Path*, *dtype: np.dtype | None = None*)

Load a DICOM ZIP archive.

Parameters

zip_path
[str] Path to the ZIP archive.

dtype
[dtype, None, optional] The data type to cast the image data as. If None, will use whatever raw image format is.

plot_3view()

Plot the stack in 3 views: axial, coronal, and sagittal.

pylinac.core.image.**tiff_to_dicom**(*tiff_file: str | Path | BytesIO*, *sid: float*, *gantry: float*, *coll: float*, *couch: float*, *dpi: float | None = None*) → Dataset

Converts a TIFF file into a **simplistic** DICOM file. Not meant to be a full-fledged tool. Used for conversion so that tools that are traditionally oriented towards DICOM have a path to accept TIFF. Currently used to convert files for WL.

Note: This will convert the image into an uint16 datatype to match the native EPID datatype.

Parameters

tiff_file
The TIFF file to be converted.

sid
The Source-to-Image distance in mm.

dpi
The dots-per-inch value of the TIFF image.

gantry
The gantry value that the image was taken at.

coll
The collimator value that the image was taken at.

couch
The couch value that the image was taken at.

pylinac.core.image.**gamma_2d**(*reference: ndarray*, *evaluation: ndarray*, *dose_to_agreement: float = 1*, *distance_to_agreement: int = 1*, *gamma_cap_value: float = 2*, *global_dose: bool = True*, *dose_threshold: float = 5*, *fill_value: float = nan*) → ndarray

Compute a 2D gamma of two 2D numpy arrays. This does NOT do size or spatial resolution checking. It performs an element-by-element evaluation. It is the responsibility of the caller to ensure the reference and evaluation have comparable spatial resolution.

The algorithm follows Table I of D. Low's 2004 paper: Evaluation of the gamma dose distribution comparison method: https://aapm.onlinelibrary.wiley.com/doi/epdf/10.1118/1.1598711

This is similar to the gamma_1d function for profiles, except we must search a 2D grid around the reference point.

Parameters

reference
The reference 2D array.

evaluation
The evaluation 2D array.

dose_to_agreement
The dose to agreement in %. E.g. 1 is 1% of global reference max dose.

distance_to_agreement
The distance to agreement in **elements**. E.g. if the value is 4 this means 4 elements from the reference point under calculation. Must be >0

gamma_cap_value
The value to cap the gamma at. E.g. a gamma of 5.3 will get capped to 2. Useful for displaying data with a consistent range.

global_dose
Whether to evaluate the dose to agreement threshold based on the global max or the dose point under evaluation.

dose_threshold
The dose threshold as a number between 0 and 100 of the % of max dose under which a gamma is not calculated. This is not affected by the global/local dose normalization and the threshold value is evaluated against the global max dose, period.

fill_value
The value to give pixels that were not calculated because they were under the dose threshold. Default is NaN, but another option would be 0. If NaN, allows the user to calculate mean/median gamma over just the evaluated portion and not be skewed by 0's that should not be considered.

`pylinac.core.image.`**`z_position`**(*metadata: Dataset*) → float
The 'z-position' of the image. Relevant for CT and MR images.

6.17.2 Geometry Module

Module for classes that represent common geometric objects or patterns.

`pylinac.core.geometry.`**`tan`**(*degrees: float*) → float
Calculate the tangent of the given degrees.

`pylinac.core.geometry.`**`atan`**(*x: float*, *y: float*) → float
Calculate the degrees of a given x/y from the origin

`pylinac.core.geometry.`**`cos`**(*degrees: float*) → float
Calculate the cosine of the given degrees.

`pylinac.core.geometry.`**`sin`**(*degrees: float*) → float
Calculate the sine of the given degrees.

class `pylinac.core.geometry.`**`Point`**(*x: float | tuple | Point = 0*, *y: float = 0*, *z: float = 0*, *idx: int | None = None*, *value: float | None = None*, *as_int: bool = False*)
Bases: `object`

A geometric point with x, y, and z coordinates/attributes.

Parameters

x
[number-like, Point, iterable] x-coordinate or iterable type containing all coordinates. If iterable, values are assumed to be in order: (x,y,z).

y
[number-like, optional] y-coordinate

idx
[int, optional] Index of point. Useful for sequential coordinates; e.g. a point on a circle profile is sometimes easier to describe in terms of its index rather than x,y coords.

value
[number-like, optional] value at point location (e.g. pixel value of an image)

as_int
[boolean] If True, coordinates are converted to integers.

`distance_to`(*thing:* Point | Circle) → float
Calculate the distance to the given point.

Parameters

thing
[Circle, Point, 2 element iterable] The other thing to calculate distance to.

`as_array`(*only_coords: bool = True*) → ndarray
Return the point as a numpy array.

`class pylinac.core.geometry.Circle`(*center_point:* Point | *Iterable = (0, 0)*, *radius: float = 0*)
Bases: `object`

A geometric circle with center Point, radius, and diameter.

Parameters

center_point
[Point, optional] Center point of the wobble circle.

radius
[float, optional] Radius of the wobble circle.

`property diameter: float`
Get the diameter of the circle.

`plot2axes`(*axes: Axes*, *edgecolor: str = 'black'*, *fill: bool = False*, *text: str = ''*, *fontsize: str = 'medium'*) → None
Plot the Circle on the axes.

Parameters

axes
[matplotlib.axes.Axes] An MPL axes to plot to.

edgecolor
[str] The color of the circle.

fill
[bool] Whether to fill the circle with color or leave hollow.

text: str
If provided, plots the given text at the center. Useful for identifying ROIs on a plotted image apart.

fontsize: str
The size of the text, if provided. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html for options.

as_dict() → dict

Convert to dict. Useful for dataclasses/Result

class pylinac.core.geometry.**Vector**(*x: float = 0*, *y: float = 0*, *z: float = 0*)

Bases: object

A vector with x, y, and z coordinates.

as_scalar() → float

Return the scalar equivalent of the vector.

distance_to(*thing:* Circle | Point) → float

Calculate the distance to the given point.

Parameters

thing

[Circle, Point, 2 element iterable] The other point to calculate distance to.

pylinac.core.geometry.**vector_is_close**(*vector1:* Vector, *vector2:* Vector, *delta: float = 0.1*) → bool

Determine if two vectors are with delta of each other; this is a simple coordinate comparison check.

class pylinac.core.geometry.**Line**(*point1:* Point | *tuple[float, float]*, *point2:* Point | *tuple[float, float]*)

Bases: object

A line that is represented by two points or by m*x+b.

Notes

Calculations of slope, etc are from here: http://en.wikipedia.org/wiki/Linear_equation and here: http://www.mathsisfun.com/algebra/line-equation-2points.html

Parameters

point1

[Point] One point of the line

point2

[Point] Second point along the line.

property m: float

Return the slope of the line.

m = (y1 - y2)/(x1 - x2)

From: http://www.purplemath.com/modules/slope.htm

property b: float

Return the y-intercept of the line.

b = y - m*x

y(*x*) → float

Return y-value along line at position x.

x(*y*) → float

Return x-value along line at position y.

property center: *Point*

Return the center of the line as a Point.

property length: float

Return length of the line, if finite.

distance_to(*point:* Point) → float

Calculate the minimum distance from the line to a point.

Equations are from here: http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html #14

Parameters

point
[Point, iterable] The point to calculate distance to.

plot2axes(*axes: Axes*, *width: float = 1*, *color: str = 'w'*, ***kwargs*) → Line3D

Plot the line to an axes.

Parameters

axes
[matplotlib.axes.Axes] An MPL axes to plot to.

color
[str] The color of the line.

class pylinac.core.geometry.**Rectangle**(*width: float*, *height: float*, *center:* Point | *tuple*, *as_int: bool = False*)

Bases: object

A rectangle with width, height, center Point, top-left corner Point, and bottom-left corner Point.

Parameters

width
[number] Width of the rectangle. Must be positive

height
[number] Height of the rectangle. Must be positive.

center
[Point, iterable, optional] Center point of rectangle.

as_int
[bool] If False (default), inputs are left as-is. If True, all inputs are converted to integers.

property br_corner: *Point*

The location of the bottom right corner.

property bl_corner: *Point*

The location of the bottom left corner.

property tl_corner: *Point*

The location of the top left corner.

property tr_corner: *Point*

The location of the top right corner.

plot2axes(*axes: Axes*, *edgecolor: str = 'black'*, *angle: float = 0.0*, *fill: bool = False*, *alpha: float = 1*, *facecolor: str = 'g'*, *label=None*, *text: str = ''*, *fontsize: str = 'medium'*, *text_rotation: float = 0*)

Plot the Rectangle to the axes.

Parameters

axes
[matplotlib.axes.Axes] An MPL axes to plot to.

edgecolor
[str] The color of the circle.

angle
[float] Angle of the rectangle.

fill
[bool] Whether to fill the rectangle with color or leave hollow.

text: str
If provided, plots the given text at the center. Useful for identifying ROIs on a plotted image apart.

fontsize: str
The size of the text, if provided. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html for options.

text_rotation: float
The rotation of the text in degrees.

6.17.3 Profile Module

See *Profiles & 1D Metrics*.

6.17.4 I/O Module

I/O helper functions for pylinac.

pylinac.core.io.**is_dicom**(*file: str | Path*) → bool

Boolean specifying if file is a proper DICOM file.

This function is a pared down version of read_preamble meant for a fast return. The file is read for a proper preamble ('DICM'), returning True if so, and False otherwise. This is a conservative approach.

Parameters

file
[str] The path to the file.

See Also

pydicom.filereader.read_preamble pydicom.filereader.read_partial

pylinac.core.io.**is_dicom_image**(*file: str | Path | BinaryIO*) → bool
Boolean specifying if file is a proper DICOM file with a image

Parameters

file
[str] The path to the file.

See Also

pydicom.filereader.read_preamble pydicom.filereader.read_partial

pylinac.core.io.**retrieve_dicom_file**(*file: str | Path | BinaryIO*) → pydicom.FileDataset
Read and return the DICOM dataset.

Parameters

file
[str] The path to the file.

class pylinac.core.io.**TemporaryZipDirectory**(*zfile: str | Path | BinaryIO*, *delete: bool = True*)
Bases: TemporaryDirectory

Creates a temporary directory that unpacks a ZIP archive. Shockingly useful

Parameters

zfile
[str] String that points to a ZIP archive.

delete
[bool] Whether to delete the temporary directory when the context manager exits.

pylinac.core.io.**retrieve_filenames**(*directory: str | Path*, *func: Callable | None = None*, *recursive: bool = True*, ***kwargs*) → list[str]
Retrieve file names in a directory.

Parameters

directory
[str] The directory to walk over recursively.

func
[function, None] The function that validates if the file name should be kept. If None, no validation will be performed and all file names will be returned.

recursive
[bool] Whether to search only the root directory.

kwargs
Additional arguments passed to the func parameter.

pylinac.core.io.**retrieve_demo_file**(*name: str*, *force: bool = False*) → Path

Retrieve the demo file either by getting it from file or from a URL.

If the file is already on disk it returns the file name. If the file isn't on disk, get the file from the URL and put it at the expected demo file location on disk for lazy loading next time.

Parameters

name
[str] The suffix to the url (location within the S3 bucket) pointing to the demo file.

pylinac.core.io.**is_url**(*url: str*) → bool

Determine whether a given string is a valid URL.

Parameters

url : str

Returns

bool

pylinac.core.io.**get_url**(*url: str*, *destination: str | Path | None = None*, *progress_bar: bool = True*) → str

Download a URL to a local file.

Parameters

url
[str] The URL to download.

destination
[str, None] The destination of the file. If None is given the file is saved to a temporary directory.

progress_bar
[bool] Whether to show a command-line progress bar while downloading.

Returns

filename
[str] The location of the downloaded file.

Notes

Progress bar use/example adapted from tqdm documentation: https://github.com/tqdm/tqdm

class pylinac.core.io.**SNCProfiler**(*path: str*, *gain_row: int = 20*, *detector_row: int = 106*, *bias_row: int = 107*, *calibration_row: int = 108*, *data_row: int = -1*, *data_columns: slice = slice(5, 259, None)*)

Bases: `object`

Load a file from a Sun Nuclear Profiler device. This accepts .prs files.

Parameters

path
[str] Path to the .prs file.

detector_row
The row that contains the detector data.

bias_row
The row that contains the bias data.

calibration_row
The row that contains the calibration data.

data_row
The row that contains the data.

data_columns
The range of columns that the data is in. Usually, there are some columns before and after the real data.

to_profiles(*n_detectors_row: int = 63*, ***kwargs*) → tuple[*SingleProfile*, *SingleProfile*, *SingleProfile*, *SingleProfile*]

Convert the SNC data to SingleProfiles. These can be analyzed directly or passed to other modules like flat/sym.

Parameters

n_detectors_row
[int] The number of detectors in a given row. Note that they Y profile includes 2 extra detectors from the other 3.

6.17.5 ROI Module

pylinac.core.roi.**bbox_center**(*region: RegionProperties*) → *Point*

Return the center of the bounding box of an scikit-image region.

Parameters

region
A scikit-image region as calculated by skimage.measure.regionprops().

Returns

point : *Point*

class pylinac.core.roi.**DiskROI**(*array: np.ndarray*, *angle: float*, *roi_radius: float*, *dist_from_center: float*, *phantom_center: tuple* | Point)

Bases: *Circle*

An class representing a disk-shaped Region of Interest.

Parameters

array
[ndarray] The 2D array representing the image the disk is on.

angle
[int, float] The angle of the ROI in degrees from the phantom center.

roi_radius
[int, float] The radius of the ROI from the center of the phantom.

dist_from_center
[int, float] The distance of the ROI from the phantom center.

phantom_center
[tuple] The location of the phantom center.

property pixel_value: float

The median pixel value of the ROI.

property std: float

The standard deviation of the pixel values.

circle_mask() → ndarray

Return a mask of the image, only showing the circular ROI.

plot2axes(*axes: plt.Axes | None = None*, *edgecolor: str = 'black'*, *fill: bool = False*, *text: str = ''*, *fontsize: str = 'medium'*) → None

Plot the Circle on the axes.

Parameters

axes
[matplotlib.axes.Axes] An MPL axes to plot to.

edgecolor
[str] The color of the circle.

fill
[bool] Whether to fill the circle with color or leave hollow.

text: str
If provided, plots the given text at the center. Useful for differentiating ROIs on a plotted image.

fontsize: str
The size of the text, if provided. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html for options.

`as_dict`() → dict

Convert to dict. Useful for dataclasses/Result

class pylinac.core.roi.**LowContrastDiskROI**(*array: np.ndarray | ArrayImage*, *angle: float*, *roi_radius: float*, *dist_from_center: float*, *phantom_center: tuple | Point*, *contrast_threshold: float | None = None*, *contrast_reference: float | None = None*, *cnr_threshold: float | None = None*, *contrast_method: str = 'Michelson'*, *visibility_threshold: float | None = 0.1*)

Bases: *DiskROI*

A class for analyzing the low-contrast disks.

Parameters

contrast_threshold
[float, int] The threshold for considering a bubble to be "seen".

`property signal_to_noise: float`

The signal-to-noise ratio. Cast to numpy first to use numpy overflow handling.

`property contrast_to_noise: float`

The contrast to noise ratio of the ROI. Cast to numpy first to use numpy overflow handling.

`property michelson: float`

The Michelson contrast

`property weber: float`

The Weber contrast

`property rms: float`

The root-mean-square contrast

`property ratio: float`

The ratio contrast

property contrast: float

The contrast of the bubble. Uses the contrast method passed in the constructor. See https://en.wikipedia.org/wiki/Contrast_(vision).

property cnr_constant: float

The contrast-to-noise value times the bubble diameter.

property visibility: float

The visual perception of CNR. Uses the model from A Rose: https://www.osapublishing.org/josa/abstract.cfm?uri=josa-38-2-196. See also here: https://howradiologyworks.com/x-ray-cnr/. Finally, a review paper here: http://xrm.phys.northwestern.edu/research/pdf_papers/1999/burgess_josaa_1999.pdf Importantly, the Rose model is not applicable for high-contrast use cases.

property contrast_constant: float

The contrast value times the bubble diameter.

property passed: bool

Whether the disk ROI contrast passed.

property passed_visibility: bool

Whether the disk ROI's visibility passed.

property passed_contrast_constant: bool

Boolean specifying if ROI pixel value was within tolerance of the nominal value.

property passed_cnr_constant: bool

Boolean specifying if ROI pixel value was within tolerance of the nominal value.

property plot_color: str

Return one of two colors depending on if ROI passed.

property plot_color_constant: str

Return one of two colors depending on if ROI passed.

property plot_color_cnr: str

Return one of two colors depending on if ROI passed.

as_dict() → dict

Dump important data as a dictionary. Useful when exporting a *results_data* output

percentile(*percentile: float*) → float

Return the pixel value at the given percentile.

property std: float

The std within the ROI.

property max: float

The max pixel value of the ROI.

property min: float

The min pixel value of the ROI.

class pylinac.core.roi.**HighContrastDiskROI**(*array: np.ndarray*, *angle: float*, *roi_radius: float*, *dist_from_center: float*, *phantom_center: tuple | Point*, *contrast_threshold: float*)

Bases: *DiskROI*

A class for analyzing the high-contrast disks.

Parameters

contrast_threshold
[float, int] The threshold for considering a bubble to be "seen".

property max: ndarray
The max pixel value of the ROI.

property min: ndarray
The min pixel value of the ROI.

class pylinac.core.roi.**RectangleROI**(*array*, *width*, *height*, *angle*, *dist_from_center*, *phantom_center*)
Bases: *Rectangle*

Class that represents a rectangular ROI.

Parameters

width
[number] Width of the rectangle. Must be positive

height
[number] Height of the rectangle. Must be positive.

center
[Point, iterable, optional] Center point of rectangle.

as_int
[bool] If False (default), inputs are left as-is. If True, all inputs are converted to integers.

property pixel_array: ndarray
The pixel array within the ROI.

property pixel_value: float
The pixel array within the ROI.

property mean: float
The mean value within the ROI.

property std: float
The std within the ROI.

property min: float
The min value within the ROI.

property max: float
The max value within the ROI.

6.17.6 Mask Module

Module for processing "masked" arrays, i.e. binary images.

pylinac.core.mask.**bounding_box**(*array: np.ndarray*)

Get the bounding box values of an ROI in a 2D array.

6.17.7 Utilities Module

Utility functions for pylinac.

pylinac.core.utilities.**convert_to_enum**(*value: str | Enum | None*, *enum: type[Enum]*) → Enum

Convert a value to an enum representation from an enum value if needed

class pylinac.core.utilities.**OptionListMixin**

Bases: object

A mixin class that will create a list of the class attributes. Used for enum-like classes

class pylinac.core.utilities.**ResultBase**

Bases: object

pylinac_version: str

date_of_analysis: datetime

pylinac.core.utilities.**clear_data_files**()

Delete all demo files, image classifiers, etc from the demo folder

pylinac.core.utilities.**assign2machine**(*source_file: str*, *machine_file: str*)

Assign a DICOM RT Plan file to a specific machine. The source file is overwritten to contain the machine of the machine file.

Parameters

source_file
[str] Path to the DICOM RTPlan file that contains the fields/plan desired (e.g. a Winston Lutz set of fields or Varian's default PF files).

machine_file
[str] Path to a DICOM RTPlan file that has the desired machine. This is easily obtained from pushing a plan from the TPS for that specific machine. The file must contain at least one valid field.

pylinac.core.utilities.**is_close**(*val: float*, *target: float | Sequence*, *delta: float = 1*)

Return whether the value is near the target value(s).

Parameters

val
[number] The value being compared against.

target
[number, iterable] If a number, the values are simply evaluated. If a sequence, each target is compared to `val`. If any values of `target` are close, the comparison is considered True.

Returns

bool

`pylinac.core.utilities.`**`simple_round`**(*number: float | int*, *decimals: int | None = 0*) → float | int

Round a number to the given number of decimals. Fixes small floating number errors. If decimals is None, no rounding is performed

`pylinac.core.utilities.`**`is_iterable`**(*object*) → bool

Determine if an object is iterable.

`class` `pylinac.core.utilities.`**`Structure`**(***kwargs*)

Bases: `object`

A simple structure that assigns the arguments to the object.

`pylinac.core.utilities.`**`decode_binary`**(*file: BinaryIO*, *dtype: type[int] | type[float] | type[str] | str | np.dtype*, *num_values: int = 1*, *cursor_shift: int = 0*, *strip_empty: bool = True*) → int | float | str | np.ndarray | list

Read in a raw binary file and convert it to given data types.

Parameters

file
The open file object.

dtype
The expected data type to return. If int or float and num_values > 1, will return numpy array.

num_values
The expected number of dtype to return

Note: This is not the same as the number of bytes.

---

cursor_shift
[int] The number of bytes to move the cursor forward after decoding. This is used if there is a reserved section after the read-in segment.

strip_empty
[bool] Whether to strip trailing empty/null values for strings.

6.17.8 Contrast Module

class pylinac.core.contrast.**Contrast**

Bases: *OptionListMixin*

Contrast calculation technique. See *Visibility*

MICHELSON = 'Michelson'

WEBER = 'Weber'

RATIO = 'Ratio'

RMS = 'Root Mean Square'

DIFFERENCE = 'Difference'

pylinac.core.contrast.**visibility**(*array: ndarray*, *radius: float*, *std: float*, *algorithm: str*) → float

The visual perception of CNR. Uses the model from A Rose: https://www.osapublishing.org/josa/abstract.cfm?uri=josa-38-2-196. See also here: https://howradiologyworks.com/x-ray-cnr/. Finally, a review paper here: http://xrm.phys.northwestern.edu/research/pdf_papers/1999/burgess_josaa_1999.pdf Importantly, the Rose model is not applicable for high-contrast use cases.

This uses the `contrast` function under the hood. Consult before using.

Parameters

array
The numpy array of the contrast ROI or a 2-element array containing the individual inputs. See `contrast` for more.

radius
The radius of the contrast ROI

std
Standard deviation of the array. This can sometimes be obtained from another ROI, so it is a separate parameter.

algorithm
The contrast method. See *Contrast* for options.

pylinac.core.contrast.**contrast**(*array: ndarray*, *algorithm: str*) → float

Generic contrast function. Different algorithms have different inputs, so caution is advised. When possible, the exact contrast function is preferred.

For Michelson and RMS algorithms, the input array can be any ordinary numpy array. For Weber and Ratio algorithms, the array is assumed to be a 2-element array.

Parameters

array
The numpy array of the ROI or 2-element input array. This is used in combination with the method.

algorithm
The contrast method. See `Contrast` for options.

`pylinac.core.contrast.rms`(*array: ndarray*) → float

The root-mean-square contrast. Requires values be within 0 and 1.

`pylinac.core.contrast.difference`(*feature: float*, *background: float*) → float

The simple absolute difference between the feature ROI and background ROI. This can be useful if the default CNR formula is desired (since pylinac CNR is based on the contrast algorithm chosen.

See also:

https://en.wikipedia.org/wiki/Contrast-to-noise_ratio

`pylinac.core.contrast.michelson`(*array: ndarray*) → float

The Michelson contrast. Used for sinusoidal patterns. Ranges from 0 to 1.

See also:

https://en.wikipedia.org/wiki/Contrast_(vision)#Michelson_contrast

`pylinac.core.contrast.weber`(*feature: float*, *background: float*) → float

The Weber contrast. Used for patterns with a small feature within a large background. Ranges from 0 to infinity.

For backwards compatibility with previous versions, the absolute difference is used, making the range 0 to infinity vs -1 to infinity.

See also:

https://en.wikipedia.org/wiki/Contrast_(vision)#Weber_contrast

> **Danger:** The default definition does not use the absolute value. We only use it here for backwards compatibility.

`pylinac.core.contrast.ratio`(*feature: float*, *reference: float*) → float

The ratio of luminescence

6.18 Image Generator

6.18.1 Overview

New in version 2.4.

The image generator module allows users to generate simulated radiation images. This module is different than other modules in that the goal here is non-deterministic. There are no phantom analysis routines here. What is here started as a testing concept for pylinac itself, but has uses for advanced users of pylinac who wish to build their own tools.

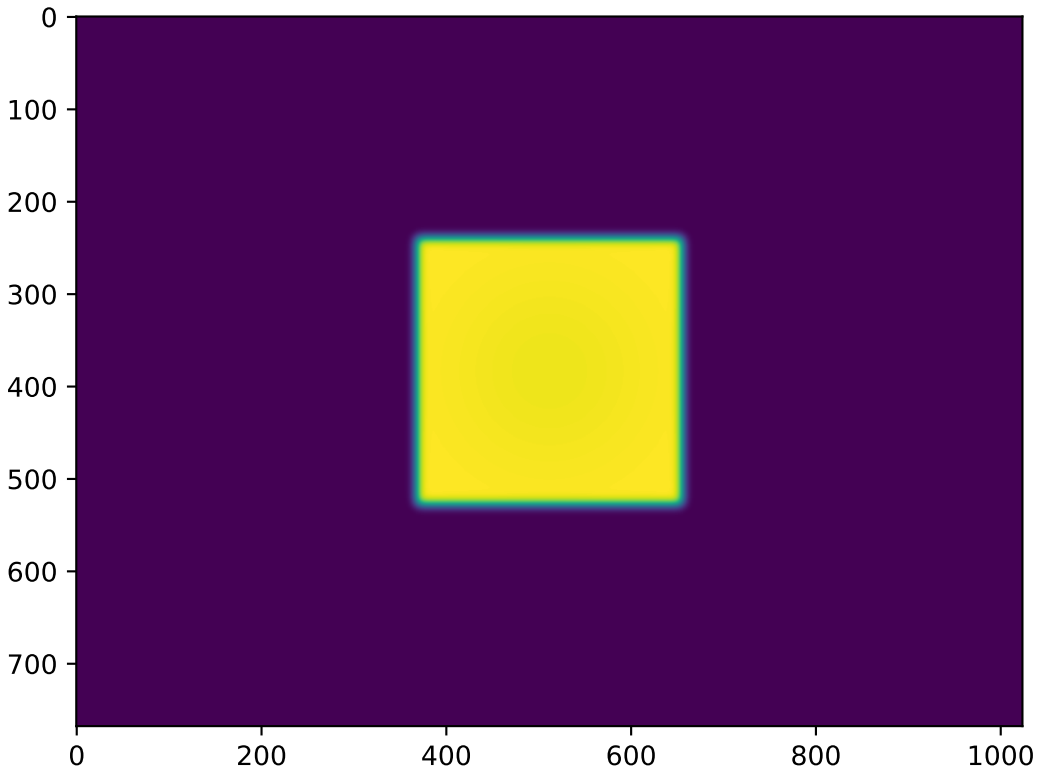
> **Warning:** This feature is currently experimental and untested.

The module allows users to create a pipeline ala keras, where layers are added to an empty image. The user can add as many layers as they wish.

6.18.2 Quick Start

The basics to get started are to import the image simulators and layers from pylinac and add the layers as desired.

```
from matplotlib import pyplot as plt

from pylinac.core.image_generator import AS1000Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1000 = AS1000Image()  # this will set the pixel size and shape automatically
as1000.add_layer(FilteredFieldLayer(field_size_mm=(50, 50)))  # create a 50x50mm square␣
↪field
as1000.add_layer(GaussianFilterLayer(sigma_mm=2))  # add an image-wide gaussian to␣
↪simulate penumbra/scatter
as1000.generate_dicom(file_out_name="my_AS1000.dcm", gantry_angle=45)  # create a DICOM␣
↪file with the simulated image
# plot the generated image
plt.imshow(as1000.image)
```



6.18.3 Layers & Simulators

Layers are very simple structures. They usually have constructor arguments specific to the layer and always define an `apply` method with the signature `.apply(image, pixel_size) -> image`. The apply method returns the modified image (a numpy array). That's it!

Simulators are also simple and define the parameters of the image to which layers are added. They have `pixel_size` and `shape` properties and always have an `add_layer` method with the signature `.add_layer(layer: Layer)`. They also have a `generate_dicom` method for dumping the image along with mostly stock metadata to DICOM.

6.18.4 Extending Layers & Simulators

This module is meant to be extensible. That's why the structures are defined so simply. To create a custom simulator, inherit from `Simulator` and define the pixel size and shape. Note that generating DICOM does not come for free:

```
from pylinac.core.image_generator.simulators import Simulator


class AS5000(Simulator):
    pixel_size = 0.12
    shape = (5000, 5000)


# use like any other simulator
```

To implement a custom layer, inherit from `Layer` and implement the `apply` method:

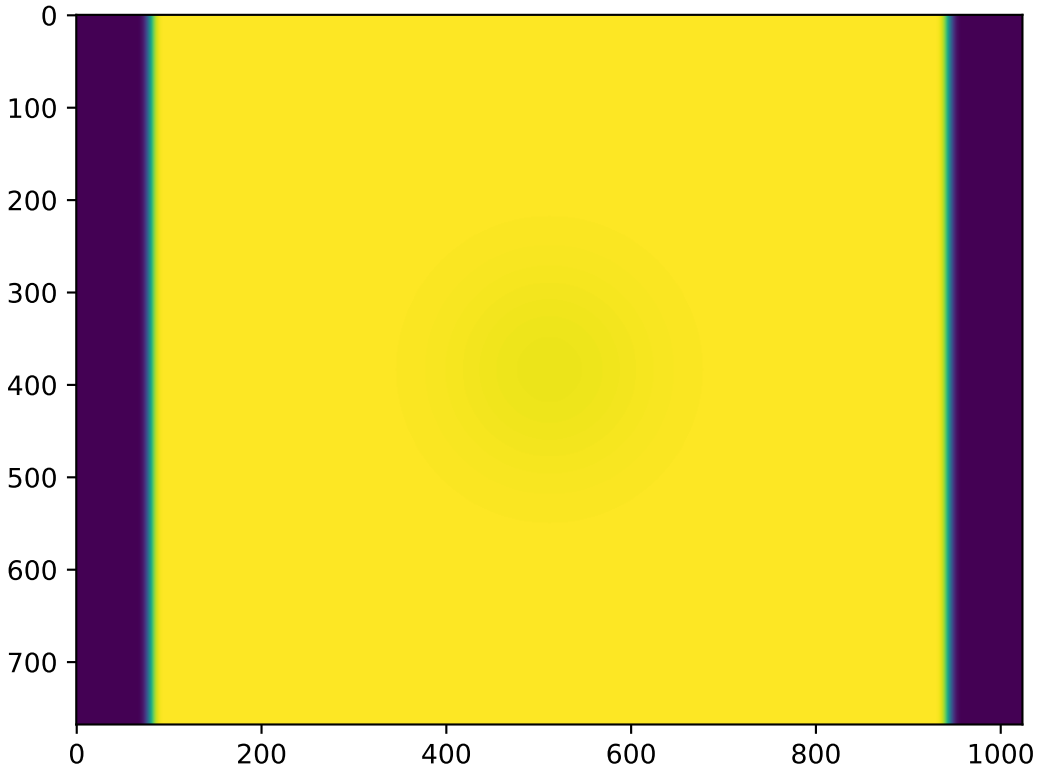
```
from pylinac.core.image_generator.layers import Layer


class MyAwesomeLayer(Layer):
    def apply(image, pixel_size):
        # do stuff here
        return image


# use
from pylinac.core.image_generator import AS1000Image

as1000 = AS1000Image()
as1000.add_layer(MyAwesomeLayer())
...
```

6.18.5 Examples

Let's make some images!

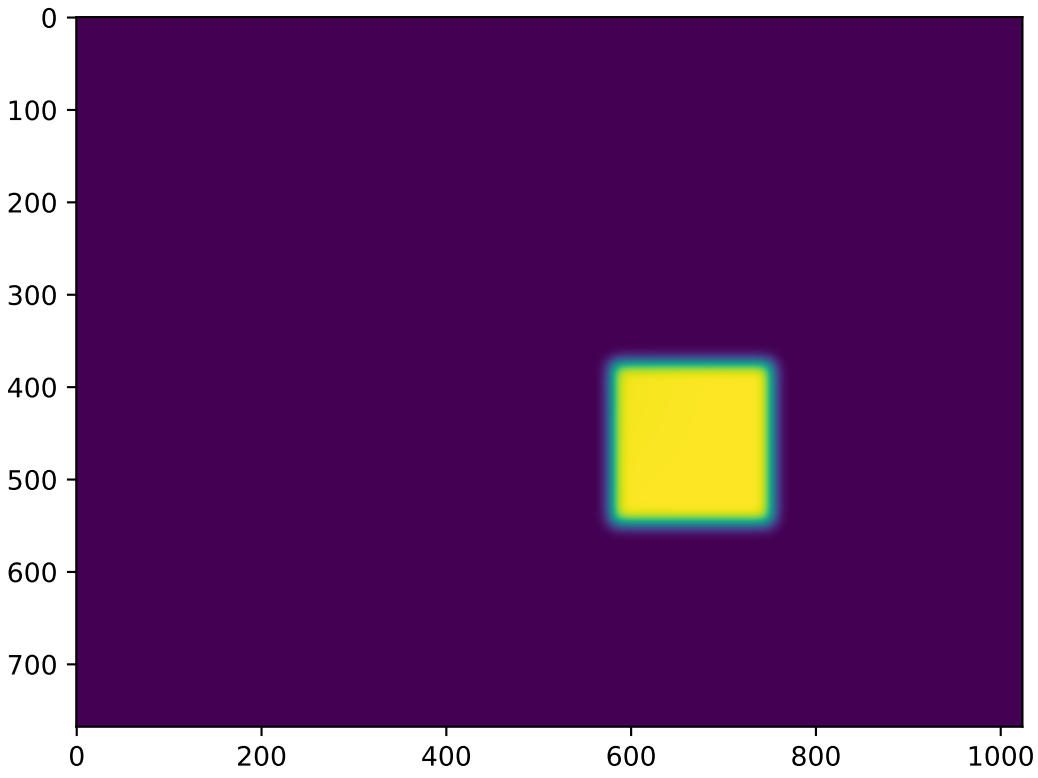
Simple Open Field

```
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1000Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1000 = AS1000Image()  # this will set the pixel size and shape automatically
as1000.add_layer(FilteredFieldLayer(field_size_mm=(150, 150)))  # create a 50x50mm␣
↪square field
as1000.add_layer(GaussianFilterLayer(sigma_mm=2))  # add an image-wide gaussian to␣
↪simulate penumbra/scatter
# plot the generated image
plt.imshow(as1000.image)
```



Off-center Open Field

```
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1000Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1000 = AS1000Image()  # this will set the pixel size and shape automatically
as1000.add_layer(FilteredFieldLayer(field_size_mm=(30, 30), cax_offset_mm=(20, 40)))
as1000.add_layer(GaussianFilterLayer(sigma_mm=3))
# plot the generated image
plt.imshow(as1000.image)
```



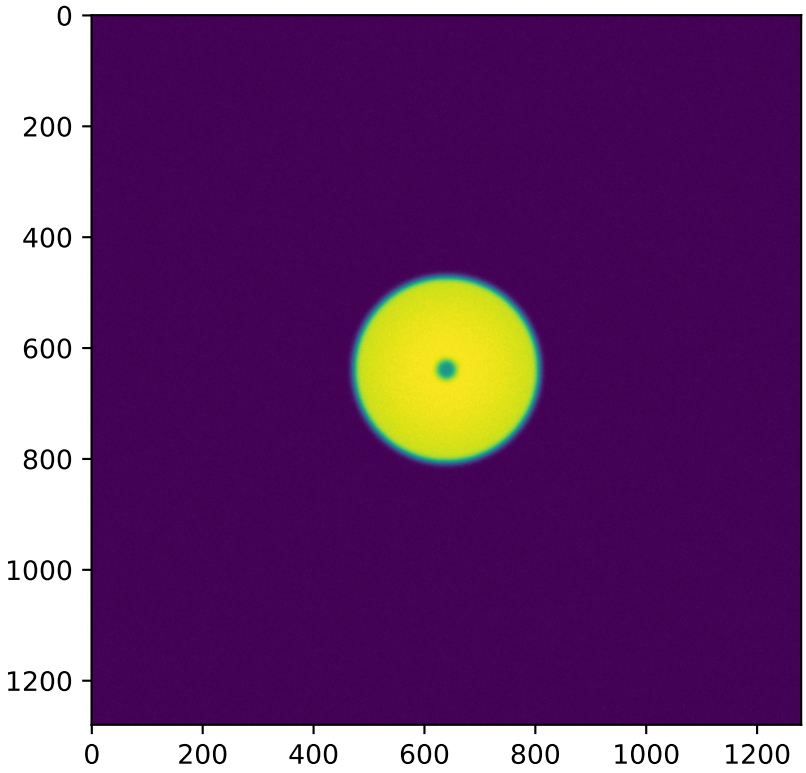
Winston-Lutz FFF Cone Field with Noise

```
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1200Image
from pylinac.core.image_generator.layers import FilterFreeConeLayer, GaussianFilterLayer,
↪ PerfectBBLayer, RandomNoiseLayer

as1200 = AS1200Image()
as1200.add_layer(FilterFreeConeLayer(50))
```

(continues on next page)

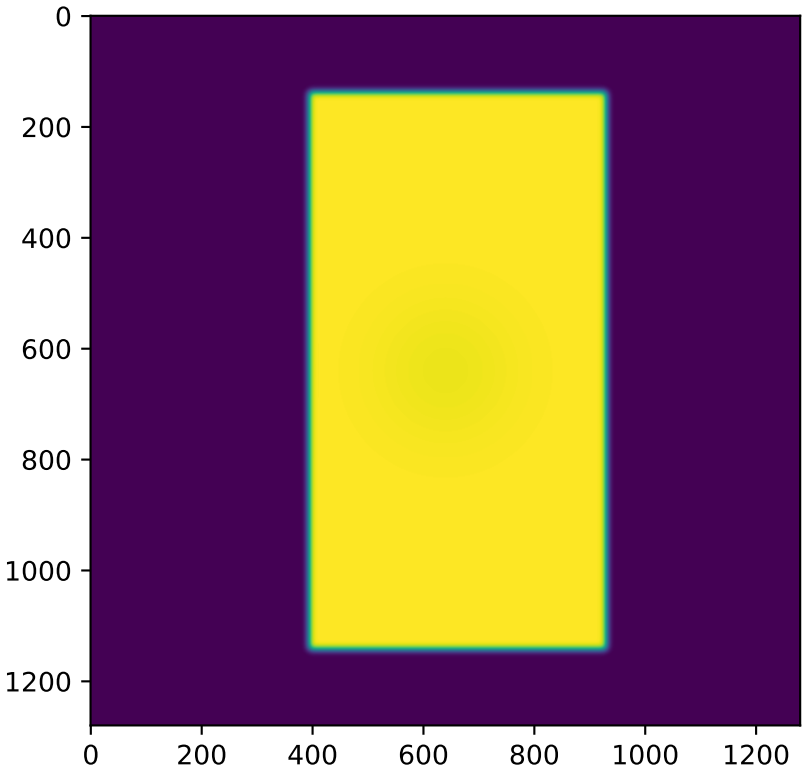
(continued from previous page)

```
as1200.add_layer(PerfectBBLayer(bb_size_mm=5))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=0.02))
# plot the generated image
plt.imshow(as1200.image)
```



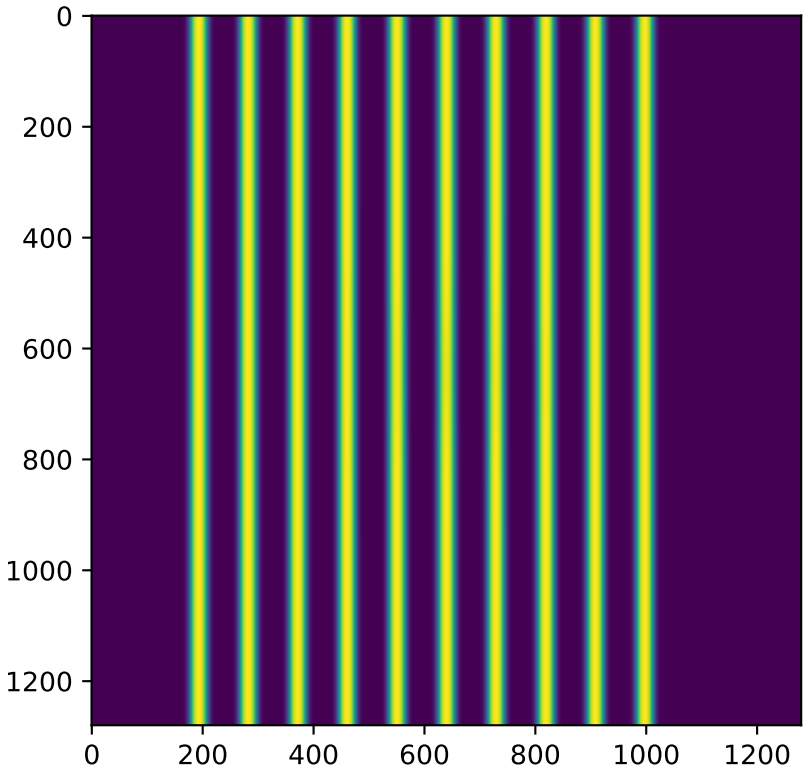
VMAT DRMLC

```
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1200Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1200 = AS1200Image()
as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, -40)))
as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, -10)))
as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, 20)))
as1200.add_layer(FilteredFieldLayer((150, 20), cax_offset_mm=(0, 50)))
as1200.add_layer(GaussianFilterLayer())
plt.imshow(as1200.image)
plt.show()
```


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200

Picket Fence

```
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1200Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1200 = AS1200Image()
height = 350
width = 4
offsets = range(-100, 100, 20)
for offset in offsets:
    as1200.add_layer(FilteredFieldLayer((height, width), cax_offset_mm=(0, offset)))
as1200.add_layer(GaussianFilterLayer())
plt.imshow(as1200.image)
plt.show()
```



Starshot

Simulating a starshot requires a small trick as angled fields cannot be handled by default. The following example rotates the image after every layer is applied.

Note: Rotating the image like this is a convenient trick but note that it will rotate the entire existing image including all previous layers. This will also possibly erroneously adjust the horn effect simulation. Use with caution.

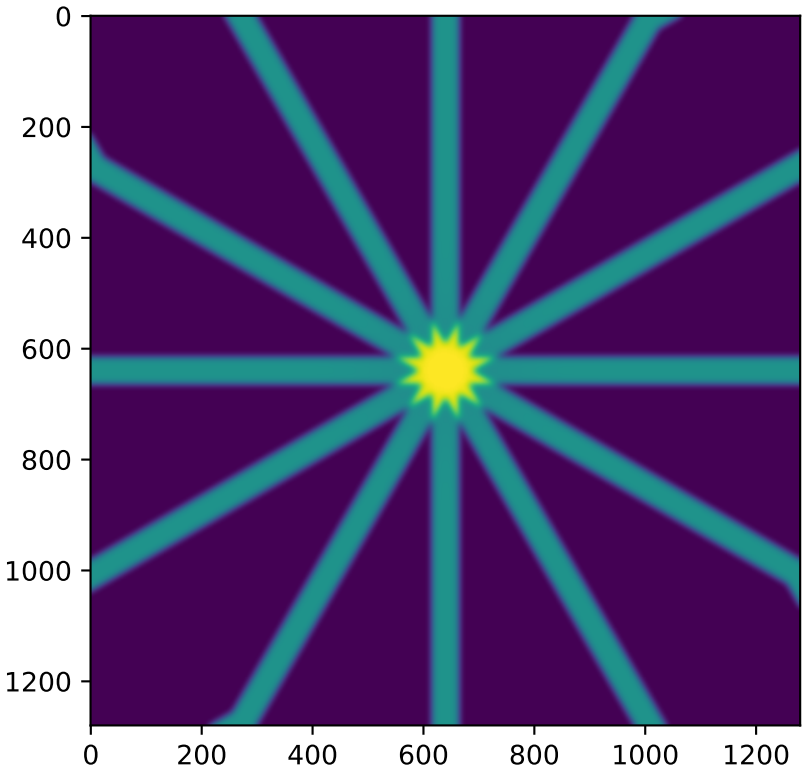
```
from scipy import ndimage
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1200Image
from pylinac.core.image_generator.layers import FilteredFieldLayer, GaussianFilterLayer

as1200 = AS1200Image()
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.image = ndimage.rotate(as1200.image, 30, reshape=False, mode='nearest')
as1200.add_layer(FilteredFieldLayer((250, 7), alpha=0.5))
as1200.add_layer(GaussianFilterLayer())
plt.imshow(as1200.image)
plt.show()
```

Helper utilities

Using the new utility functions of v2.5+ we can construct full dicom files of picket fence and winston-lutz sets of images:

```
from pylinac.core.image_generator import generate_picketfence, generate_winstonlutz
from pylinac.core import image_generator

sim = image_generator.simulators.AS1000Image()
field_layer = image_generator.layers.FilteredFieldLayer  # could also do FilterFreeLayer
generate_picketfence(
    simulator=Simulator,
    field_layer=FilteredFieldLayer,
    file_out="pf_image.dcm",
    pickets=11,
    picket_spacing_mm=20,
    picket_width_mm=2,
    picket_height_mm=300,
    gantry_angle=0,
)
# we now have a pf image saved as 'pf_image.dcm'
```

(continues on next page)


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200

(continued from previous page)

```
# create a set of WL images
# this will create 4 images (via image_axes len) with an offset of 3mm to the left
# the function is smart enough to correct for the offset w/r/t gantry angle.
generate_winstonlutz(
    simulator=sim,
    field_layer=field_layer,
    final_layers=[GaussianFilterLayer()],
    gantry_tilt=0,
    dir_out="./wl_dir",
    offset_mm_left=3,
    image_axes=[[0, 0, 0], [180, 0, 0], [90, 0, 0], [270, 0, 0]],
)
```

6.18.6 Tips & Tricks

- The `FilteredFieldLayer` and `FilterFree<Field, Cone>Layer` have gaussian filters applied to create a first-order approximation of the horn(s) of the beam. It doesn't claim to be super-accurate, it's just to give some reality to the images. You can adjust the magnitude of these parameters to simulate other energies (e.g. sharper horns) when defining the layer.
- The `Perfect...Layer` s do not apply any energy correction as above.
- Use `alpha` to adjust the intensity of the layer. E.g. the BB layer has a default alpha of -0.5 to simulate attenuation. This will subtract out up to half of the possible dose range existing on the image thus far (e.g. an open image of alpha 1.0 will be reduced to 0.5 after a BB is layered with alpha=-0.5). If you want to simulate a thick material like tungsten you can adjust the alpha to be lower (more attenuation). An alpha of 1 means full radiation, no attenuation (like an open field).
- Generally speaking, don't apply more than one `GaussianFilterLayer` since they are additive. A good rule is to apply one filter at the end of your layering.
- Apply `ConstantLayer` s at the beginning rather than the end.

Warning: Pylinac uses unsigned int16 datatypes (native EPID dtype). To keep images from flipping bits when adding layers, pylinac will clip the values. Just be careful when, e.g. adding a `ConstantLayer` at the end of a layering. Better to do this at the beginning.

6.18.7 API Documentation

Layers

class pylinac.core.image_generator.layers.**PerfectConeLayer**(*cone_size_mm: float = 10*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = 1.0*)

Bases: `Layer`

A cone without flattening filter effects

Parameters

cone_size_mm
Cone size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

apply(*image: ndarray*, *pixel_size: float*, *mag_factor: float*) → ndarray

Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**FilterFreeConeLayer**(*cone_size_mm: float = 10*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = 1.0*, *filter_magnitude: float = 0.4*, *filter_sigma_mm: float = 80*)

Bases: *PerfectConeLayer*

A cone with flattening filter effects.

Parameters

cone_size_mm
Cone size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

filter_magnitude
The magnitude of the CAX peak. Larger values result in "pointier" fields.

filter_sigma_mm
Proportional to the width of the CAX peak. Larger values produce wider curves.

apply(*image: ndarray*, *pixel_size: float*, *mag_factor: float*) → ndarray

Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**PerfectFieldLayer**(*field_size_mm: float, float = (10, 10)*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = 1.0*)

Bases: Layer

A square field without flattening filter effects

Parameters

field_size_mm
Field size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

apply(*image: ndarray*, *pixel_size: float*, *mag_factor: float*) → array

Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**FilteredFieldLayer**(*field_size_mm: float, float = (10, 10)*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = 1.0*, *gaussian_height: float = 0.03*, *gaussian_sigma_mm: float = 32*)

Bases: *PerfectFieldLayer*

A square field with flattening filter effects

Parameters

field_size_mm
Field size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

gaussian_height
The intensity of the "horns", or more accurately, the CAX dip. Proportional to the max value allowed for the data type. Increase to make the horns more prominent.

gaussian_sigma_mm
The width of the "horns". A.k.a. the CAX dip width. Increase to create a wider horn effect.

apply(*image: array*, *pixel_size: float*, *mag_factor: float*) → array

Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**FilterFreeFieldLayer**(*field_size_mm: float, float = (10, 10)*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = 1.0*, *gaussian_height: float = 0.4*, *gaussian_sigma_mm: float = 80*)

Bases: *FilteredFieldLayer*

A square field with flattening filter free (FFF) effects

Parameters

field_size_mm
Field size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

gaussian_height
The magnitude of the CAX peak. Larger values result in "pointier" fields.

gaussian_sigma_mm
Proportional to the width of the CAX peak. Larger values produce wider curves.

apply(*image: array*, *pixel_size: float*, *mag_factor: float*) → array
Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**PerfectBBLayer**(*bb_size_mm: float = 5*, *cax_offset_mm: float, float = (0, 0)*, *alpha: float = -0.5*)

Bases: *PerfectConeLayer*

A BB-like layer. Like a cone, but with lower alpha (i.e. higher opacity)

Parameters

cone_size_mm
Cone size in mm at the iso plane

cax_offset_mm
The offset in mm. (out, right)

alpha
The intensity of the layer. 1 is full saturation/radiation. 0 is none.

apply(*image: ndarray*, *pixel_size: float*, *mag_factor: float*) → ndarray
Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**GaussianFilterLayer**(*sigma_mm: float = 2*)

Bases: Layer

A Gaussian filter. Simulates the effects of scatter on the field

apply(*image: array*, *pixel_size: float*, *mag_factor: float*) → array
Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**RandomNoiseLayer**(*mean: float = 0.0*, *sigma: float = 0.001*)

Bases: Layer

A salt and pepper noise, simulating dark current

apply(*image: array*, *pixel_size: float*, *mag_factor: float*) → array
Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

class pylinac.core.image_generator.layers.**ConstantLayer**(*constant: float*)

Bases: Layer

A constant layer. Can be used to simulate scatter or background.

apply(*image: array*, *pixel_size: float*, *mag_factor: float*) → array

Apply the layer. Takes a 2D array and pixel size value in and returns a modified array.

Simulators

class pylinac.core.image_generator.simulators.**AS500Image**(*sid: float = 1500*)

Bases: Simulator

Simulates an AS500 EPID image.

Parameters

sid

Source to image distance in mm.

as_dicom(*gantry_angle: float = 0.0*, *coll_angle: float = 0.0*, *table_angle: float = 0.0*) → Dataset

Create and return a pydicom Dataset. I.e. create a pseudo-DICOM image.

add_layer(*layer: Layer*) → None

Add a layer to the image

class pylinac.core.image_generator.simulators.**AS1000Image**(*sid: float = 1500*)

Bases: Simulator

Simulates an AS1000 EPID image.

Parameters

sid

Source to image distance in mm.

as_dicom(*gantry_angle: float = 0.0*, *coll_angle: float = 0.0*, *table_angle: float = 0.0*) → Dataset

Create and return a pydicom Dataset. I.e. create a pseudo-DICOM image.

add_layer(*layer: Layer*) → None

Add a layer to the image

class pylinac.core.image_generator.simulators.**AS1200Image**(*sid: float = 1500*)

Bases: Simulator

Simulates an AS1200 EPID image.

Parameters

sid
Source to image distance in mm.

add_layer(*layer: Layer*) → None
Add a layer to the image

as_dicom(*gantry_angle: float = 0.0*, *coll_angle: float = 0.0*, *table_angle: float = 0.0*) → Dataset
Create and return a pydicom Dataset. I.e. create a pseudo-DICOM image.

Helpers

pylinac.core.image_generator.utils.**generate_picketfence**(*simulator: Simulator*, *field_layer: type[*FilterFreeFieldLayer | FilteredFieldLayer | PerfectFieldLayer*]*, *file_out: str*, *final_layers: list[Layer] = None*, *pickets: int = 11*, *picket_spacing_mm: float = 20*, *picket_width_mm: int = 2*, *picket_height_mm: int = 300*, *gantry_angle: int = 0*, *orientation: Orientation = Orientation.UP_DOWN*, *picket_offset_error: Sequence | None = None*) → None

Create a mock picket fence image. Will always be up-down.

Parameters

simulator
The image simulator

field_layer
The primary field layer

file_out
The name of the file to save the DICOM file to.

final_layers
Optional layers to apply at the end of the procedure. Useful for noise or blurring.

pickets
The number of pickets

picket_spacing_mm
The space between pickets

picket_width_mm
Picket width parallel to leaf motion

picket_height_mm
Picket height parallel to leaf motion

gantry_angle
Gantry angle; sets the DICOM tag.

pylinac.core.image_generator.utils.**generate_winstonlutz**(*simulator: Simulator*, *field_layer: type[Layer]*, *dir_out: str*, *field_size_mm: tuple[float, float] = (30, 30)*, *final_layers: list[Layer] | None = None*, *bb_size_mm: float = 5*, *offset_mm_left: float = 0*, *offset_mm_up: float = 0*, *offset_mm_in: float = 0*, *image_axes: int, int, int, ... = ((0, 0, 0), (90, 0, 0), (180, 0, 0), (270, 0, 0))*, *gantry_tilt: float = 0*, *gantry_sag: float = 0*, *clean_dir: bool = True*, *field_alpha: float = 1.0*, *bb_alpha: float = -0.5*) → list[str]

Create a mock set of WL images, simulating gantry sag effects. Produces one image for each item in image_axes.

Parameters

simulator
The image simulator

field_layer
The primary field layer simulating radiation

dir_out
The directory to save the images to.

field_size_mm
The field size of the radiation field in mm

final_layers
Layers to apply after generating the primary field and BB layer. Useful for blurring or adding noise.

bb_size_mm
The size of the BB. Must be positive.

offset_mm_left
How far left (lat) to set the BB. Can be positive or negative.

offset_mm_up
How far up (vert) to set the BB. Can be positive or negative.

offset_mm_in
How far in (long) to set the BB. Can be positive or negative.

image_axes
List of axis values for the images. Sequence is (Gantry, Coll, Couch).

gantry_tilt
The tilt of the gantry that affects the position at 0 and 180. Simulates a simple cosine function.

gantry_sag
The sag of the gantry that affects the position at gantry=90 and 270. Simulates a simple sine function.

clean_dir
Whether to clean out the output directory. Useful when iterating.

field_alpha
The normalized alpha (i.e. signal) of the radiation field. Use in combination with bb_alpha such that the sum of the two is always <= 1.

bb_alpha
The normalized alpha (in the case of the BB think of it as attenuation) of the BB against the radiation field. More negative values attenuate (remove signal) more.

pylinac.core.image_generator.utils.**generate_winstonlutz_cone**(*simulator: Simulator*, *cone_layer: type[FilterFreeConeLayer] | type[PerfectConeLayer]*, *dir_out: str*, *cone_size_mm: float = 17.5*, *final_layers: list[Layer] | None = None*, *bb_size_mm: float = 5*, *offset_mm_left: float = 0*, *offset_mm_up: float = 0*, *offset_mm_in: float = 0*, *image_axes: int, int, int, ... = ((0, 0, 0), (90, 0, 0), (180, 0, 0), (270, 0, 0))*, *gantry_tilt: float = 0*, *gantry_sag: float = 0*, *clean_dir: bool = True*) → list[str]

Create a mock set of WL images with a cone field, simulating gantry sag effects. Produces one image for each item in image_axes.

Parameters

simulator
The image simulator

cone_layer
The primary field layer simulating radiation

dir_out
The directory to save the images to.

cone_size_mm
The field size of the radiation field in mm

final_layers
Layers to apply after generating the primary field and BB layer. Useful for blurring or adding noise.

bb_size_mm
The size of the BB. Must be positive.

offset_mm_left
How far left (lat) to set the BB. Can be positive or negative.

offset_mm_up
How far up (vert) to set the BB. Can be positive or negative.

offset_mm_in
How far in (long) to set the BB. Can be positive or negative.

image_axes
List of axis values for the images. Sequence is (Gantry, Coll, Couch).

gantry_tilt
The tilt of the gantry in degrees that affects the position at 0 and 180. Simulates a simple cosine function.

gantry_sag
The sag of the gantry that affects the position at gantry=90 and 270. Simulates a simple sine function.

clean_dir
Whether to clean out the output directory. Useful when iterating.

pylinac.core.image_generator.utils.**generate_winstonlutz_multi_bb_multi_field**(*simulator: Simulator*, *field_layer: type[Layer]*, *dir_out: str*, *field_offsets: list[list[float]]*, *bb_offsets: list[list[float]] | list[dict[str, float]]*, *field_size_mm: tuple[float, float] = (20, 20)*, *final_layers: list[Layer] | None = None*, *bb_size_mm: float = 5*, *image_axes: int, int, int, ... = ((0, 0, 0), (90, 0, 0), (180, 0, 0), (270, 0, 0))*, *gantry_tilt: float = 0*, *gantry_sag: float = 0*, *clean_dir: bool = True*, *jitter_mm: float = 0*, *align_to_pixels: bool = True*) → list[str]

Create a mock set of WL images, simulating gantry sag effects. Produces one image for each item in image_axes. This will also generate multiple BBs on the image, one per item in *offsets*. Each offset should be a list of the shifts of the BB relative to isocenter like so: [<left>, <up>, <in>] OR an arrangement from the WL module.

Parameters

simulator
The image simulator

field_layer
The primary field layer simulating radiation

dir_out
The directory to save the images to.

field_offsets
A list of lists containing the shift of the fields. Format is the same as bb_offsets.

bb_offsets
A list of lists containing the shift of the BBs from iso; each sublist should be a 3-item list/tuple of left, up, in. Negative values are acceptable and will go the opposite direction.

field_size_mm
The field size of the radiation field in mm

final_layers
Layers to apply after generating the primary field and BB layer. Useful for blurring or adding noise.

bb_size_mm
The size of the BB. Must be positive.

image_axes
List of axis values for the images. Sequence is (Gantry, Coll, Couch).

gantry_tilt
The tilt of the gantry in degrees that affects the position at 0 and 180. Simulates a simple cosine function.

gantry_sag
The sag of the gantry that affects the position at gantry=90 and 270. Simulates a simple sine function.

clean_dir
Whether to clean out the output directory. Useful when iterating.

jitter_mm
The amount of jitter to add to the in/left/up location of the BB in MM.

6.19 Images

Pylinac deals nearly exclusively with DICOM image data. Film has been actively avoided where possible because of 1) the increased use and technological advances of EPIDs. EPID data also contains useful tags that give contextual information about the acquisition (unless you use Elekta). And 2) film images tend to be much messier in general; they often have markings on them such as a pin prick, marker writing to identify the image, or flash on the edges of the image where the scanner and film edge did not line up. That being said, pylinac can generally handle DICOM images and general images (PNG, JPG, etc) relatively well.

The `image` module within pylinac is quite powerful and flexible to do arbitrary operations for use in custom algorithms. For example, images can be loaded easily, filters applied, cropped, rotated, and more with straightforward methods.

6.19.1 How data is loaded

Pylinac uses the excellent `pydicom` library to load DICOM images. The pydicom dataset is stored in pylinac images under the `metadata` attribute.

For non-DICOM images (JPG, PNG, TIFF, etc), the `Pillow` library is used.

To load an image, the easiest way is like so:

```
from pylinac import image

my_dcm = image.load("path/to/my/image.dcm")
my_dcm.metadata.GantryAngle  # the GantryAngle tag of the DICOM file
# these won't have the metadata property as they aren't DICOM
```

(continues on next page)

(continued from previous page)

```
my_tiff = image.load("path/to/my/image.tiff")
my_jpg = image.load("path/to/my/image.jpg")
```

See the *load()* function for details. This will return an image-like object ready for plotting or manipulation. Note that *XIM* images are handled separately.

We can also test whether a file is image-like without causing an error if it's not:

```
from pylinac import image

is_image = image.is_image("path/to/questionable.file")  # True or False
```

6.19.2 Image Manipulation

To manipulate an image, such as cropping, simply run the method. Some examples:

```
from pylinac import image

my_dcm = image.load(...)
my_dcm.filter(size=0.01, kind="median")
my_dcm.fliplr()  # flip the image left-right
my_dcm.ground()  # set minimum value to 0; useful for images with short dynamic range
my_dcm.crop(pixels=30, edges=("top", "left"))
my_dcm.normalize()  # normalize values to 1.0
my_dcm.rot90(n=1)  # rotate the image by 90 degrees
my_dcm.bit_invert()  # flip the image so that dark is light and light is dark. Useful␣
↪for EPID images.
my_dcm.plot()  # plot the image for visualization
```

These and similar methods are available to all types of images. However, some image types have additional properties and methods. For a DICOM that is from a linac EPID, we have a few extras. We need to load it specifically:

```
from pylinac import image

my_linac_dcm = image.LinacDicomImage("path/to/image.dcm")
my_linac_dcm.cax()  # a Point instance. E.g. (x=550, y=550)
my_linac_dcm.dpmm()  # the dots/mm at isocenter. Will account for the SID.
```

6.19.3 TIFF to DICOM

Pylinac will often internally convert TIFF images to pseudo-DICOM files so that the same methods are available as a DICOM. To do so:

```
from pylinac import image

image.tiff_to_dicom(
    tiff_file="path/to/image.tiff",
    dicom_file="my_new_dicom.dcm",
    sid=1000,
    gantry=0,
    coll=0,
```

(continues on next page)

(continued from previous page)

```
    couch=0,
    dpi=280,
)
```

We will now have a file in our working directory named `my_new_dicom.dcm` that is, for all intents and purposes, a DICOM file. It can be loaded with `image.load()` or `pydicom` like any normal DICOM.

6.19.4 Gamma

We can compute the gamma between two arrays or images using *gamma_2d()*:

```
import matplotlib.pyplot as plt
from pylinac import image

ref = image.load("reference_dicom.dcm")
eval = image.load("eval_dicom.dcm")

gamma = image.gamma_2d(
    reference=ref,
    evaluation=eval,
    dose_to_agreement=2,
    distance_to_agreement=3,
    global_dose=True,
    ...,
)

# gamma is a numpy array the same size as the reference/eval image
plt.imshow(gamma)
```

6.19.5 Pixel Data & Inversion

This is the most common issue when dealing with image analysis. The inversion, meaning the pixel value to radiation fluence relationship, of pylinac images used to be a simple imcompliment, meaning inverting the data while respecting the bit ranges, since most images' raw pixel data was inverted. However, to handle newer EPID images that included more and better pixel relationships, this has changed in v3.0.

Note: The axiom for pylinac (for v3.0+) is that higher pixel values == more radiation == lighter/whiter display

Image pixel values will proceed through the following conditions. The first condition that matches will be executed:

- If the `raw_pixels` parameter is set to `True`, no tags will be searched and the values from the DICOM file will be used directly. E.g.

  ```
  from pylinac.core import image

  dcm = image.load("my_dcm_file.dcm", raw_pixels=True)
  # OR
  dcm = image.DicomImage("my_dcm_file.dcm", raw_pixels=True)
  ```

  New in version 3.13.

- If the image has the Rescale Slope, Rescale Intercept and the Pixel Intensity Relationship Sign attributes, all of them are applied with a simple linear correction: $P_{corrected} = Sign * Slope * P_{raw} + Intercept$ Images from newer linac platforms appear more likely to have this attribute.
- If the image only has the Rescale Slope and Rescale Intercept but not the relationship tag then it is applied as: $P_{corrected} = Slope * P_{raw} + Intercept$. This is the most common scenario encountered to date.

  Note: It is possible that the slope has a negative value which is implicitly applying a relationship and would be equivalent to the first case, however, older images often have a simple positive slope relationship.

- If the image does not have these two tags, then an imcompliment is applied: $newarray = -oldarray + max(oldarray) + min(oldarray)$. Very old images will likely reach this condition.

Note: If your image appears to be incorrectly inverted, missing tags are likely why. Pylinac has parameters to force the inversion of the image if the end result is wrong. Furthermore, some modules perform another inversion check at runtime. This is mostly historical but was done because some images were always expected to have a certain relationship and the tag logic above was not applied consistently (both new and old images were imcomplimented, causing differences). For those modules, tags were not used but a simple histogram analysis which expects the irradiated part of the image to be either centrally located or most of the image to NOT be irradiated. This is how pylinac historically worked around this issue and got reliable results across image eras. However with this new logic, there may be analysis differences for those images. It is more correct to follow the tags but for backwards compatibility the module-specific inversion checks remain.

6.20 XIM images

Images ending in `.xim` are generally produced by a Varian TrueBeam or newer linac. They are images with additional tags. Unfortunately, they are written in binary into a custom format so using a typical image library will not work.

The binary file specification appears to be unofficial, but it does work. You can find the spec here which comes from this repo: https://bitbucket.org/dmoderesearchtools/ximreader/src/master/

Warning: Rant ahead.

The XIM images used a custom compression format. Why they chose to use a custom format is beyond me. Moreso, the format they chose was that of a PNG algorithm, but not as good. So, XIM images are just PNG images but with a custom lookup table and property tags. Everyday PNG format would've worked just as well. It's possible this is security by obscurity or NIH syndrome.

6.20.1 Loading an XIM image

To load an XIM images use the *XIM* class:

```
from pylinac.core.image import XIM


my_xim_file = r"C:\TDS\H12345\QA\image.xim"
xim_img = XIM(my_xim_file)
```

(continues on next page)

(continued from previous page)

```
# plot the image
xim_img.plot()

# see the XIM properties
print(xim_img.properties)
```

Reconstructing the image pixels is relatively slow (~1s for AS1200 image) thanks to the custom compression format, so if you are only searching through the properties you can skip reconstructing the pixels. Skipping the pixels and only reading the properties is relatively fast (order of milliseconds):

```
from pylinac.core.image import XIM


my_xim_files = [r"C:\TDS\H12345\QA\image.xim", ...]
files_to_analyze = []
for file in my_xim_files:
    # will load relatively fast
    xim_img = XIM(file, read_pixels=False)
    if xim_img.properties["AcquisitionMode"] == "Highres":
        files_to_analyze.append(file)

# now load the pixel data only for the files we're interested in
for file in files_to_analyze:
    xim_img = XIM(file)
    # image is available, do what you want
    xim_img.plot()
```

An XIM image has all the utility methods other pylinac images do, so use this to your advantage:

```
from pylinac.core.image import XIM


my_xim_file = r"C:\TDS\H12345\QA\image.xim"
xim_img = XIM(my_xim_file)

# process
xim_img.crop(pixels=30)
xim_img.filter()
xim_img.fliplr()
...
```

6.20.2 Exporting images

Exporting `*.xim` images is easy. The PNG format is recommended because its ~1/2 the size of the original xim image and will also include the properties. PNG is also lossless, so all information is retained. PNG images can usually be viewed easily across many devices and OSs and also loads very fast.

```
from pylinac.core.image import XIM


my_xim_file = r"C:\TDS\H12345\QA\image.xim"
```

(continues on next page)

(continued from previous page)

```
xim_img = XIM(my_xim_file)

xim_img.save_as("myxim.png")
# saved to PNG!
```

6.20.3 Reading exported images

To load the image in python you can use any library that reads PNG. Pillow is recommended. Opening these files are usually very fast (order of milliseconds), so if you plan on doing research or analysis of a large number of .xim images, it may be worth it to export to PNG en masse and then perform the analysis.

```
import numpy as np
import PIL.Image
import matplotlib.pyplot as plt

xim_img = PIL.Image.open("myxim.png")

# numpy array of the pixels
xim_array = np.asarray(xim_img)

# plot it
plt.imshow(xim_array)
plt.show()
```

To read the properties of an XIM file that was saved to PNG we may to have to load from strings. PNG tags are all strings, and some xim properties are arrays or numbers. In order to easily save it, we convert them all to strings. In order to get the native datatype for non-string types we cast to the inferred type. For numbers, use `float` and for lists use `json`:

```
import json
import PIL.Image

xim_img = PIL.Image.open("myxim.png")

system_version = xim_img.info["AcquisitionSystemVersion"]
# "2.7.304.16" already a string so no change needed

couch_lat = xim_img.info["CouchLat"]
# '100.39021332'  it's a string even though it looks like a number
# convert to the original type:
couch_lat_num = float(couch_lat)

# MLCs are a list; we need json
mlc_a_string = xim_img.info["MLCLeafsA"]
# '[20.6643, 20.6992, ...]'
mlc_a_list = json.loads(mlc_a_string)
# now it's a normal list: [20.6643, 20.6992, ...]
```

6.21 Contrast

Contrast is used in the catphan and planar imaging modules. There are two contrasts that are evaluated: high contrast and low contrast. High contrast is also called spatial resolution, and refers to the ability of the device to resolve high contrast objects that are abutting. This is usually measured with line pairs or a high-contrast point. Low contrast refers to the ability of the device to measure differences between two similarly-attenuating materials. The materials and regions need not be abutting as for high contrast.

Depending on who you ask/read, there are multiple definitions of contrast. For high contrast, this is less contentious than low contrast. We describe here the equations used or offered in pylinac to calculate contrast. A summary of methods can be read here: https://en.wikipedia.org/wiki/Display_contrast

6.21.1 High contrast

High contrast calculations are performed by analyzing multiple ROIs and calculating the maximum and minimum pixel value from each ROI. An ROI is used for each high contrast region (e.g. each line pair region). The contrast is first calculated, then normalized. The high contrast calculation uses the *Michelson* algorithm.

$$\frac{\frac{I_{max}-I_{min}}{I_{max}+I_{min}}}{\max\left(\frac{I_{max}-I_{min}}{I_{max}+I_{min}}\right)}$$

where $I = 1...n$ line pair ROIs. See also the *MTF* section.

6.21.2 Low contrast

Low contrast calculations are also performed by analyzing multiple ROIs, but each ROI has only one value: the median pixel value. These pixel values are compared to a reference ROI. However, that comparison is different depending on who you ask. Previously, pylinac gave only the Michelson contrast as the low contrast option. However, there are now multiple options available.

Important: The combination of low contrast and ROI size, aka visibility, is handled in the next section. Do not confuse low contrast with visibility/perception.

For all below I is the given ROI and R is the reference ROI.

Michelson

Note: For backwards compatibility with older pylinac versions, Michelson is the default contrast algorithm.

Michelson is a good algorithm for contrast and is the pylinac default for low contrast calculations. It is the only algorithm used for high contrast evaluation. The Michelson contrast can range from 0 to 1. The official definition is:

$$\frac{I_{max} - I_{min}}{I_{max} + I_{min}}$$

This is how high-contrast ROIs are evaluated, where the ROI contains both the high and low values together.

When applied to low contrast, there are usually two ROIs, the ROI of the contrast value in question and some reference value, usually the background. For low-contrast evaluation, the equation is the same, but is restated given the two

individual ROIs:

$$\frac{I_{mean} - R_{mean}}{I_{mean} + R_{mean}}$$

where I is the ROI of the contrast region in question and R is the background ROI, usually placed somewhere within the phantom area that is uniform.

It is primarily used in high-contrast, periodic pattern situations, although it can be used in most situations.

An example calculation:

```
from pylinac.core import contrast

my_roi = np.array((1.17, 1.31, 1.26, ...))
c = contrast.michelson(my_roi)
```

See also:

Wikipedia

Weber

The Weber algorithm is generally used when a large image has a small feature and the majority of the image is background. However, how the "feature" and "background" values are calculated is ambiguous. The Weber contrast value ranges from -1 to infinity.

The official definition is:

$$\frac{I - I_{background}}{I_{background}}$$

Within pylinac, this is interpreted to be the following:

$$\frac{|I_{mean} - R_{mean}|}{R_{mean}}$$

where I is the ROI of the contrast region in question and R is the background ROI, usually placed somewhere within the phantom area that is uniform.

Important: For historical reasons, the numerator is the absolute difference. This means the range is from 0 to infinity vs -1 to infinity. The repercussions is that contrast is symmetric. I.e. -1 and +1 both go to +1.

An example calculation:

```
from pylinac.core import contrast

feature_value = np.max(my_array)
background = np.median(my_array)
c = contrast.weber(feature=feature_value, background=background)
```

See also:

Wikipedia.

Ratio

The ratio algorithm is simply the value of interest over the reference or background value.

$$\frac{feature}{reference}$$

Within pylinac, this is interpreted as:

$$\frac{I_{mean}}{R_{mean}}$$

where I is the ROI of the contrast region in question and R is the background ROI, usually placed somewhere within the phantom area that is uniform.

An example calculation:

```
from pylinac.core import contrast

feature_value = np.max(my_array)
reference = np.min(my_array)
c = contrast.ratio(feature=feature_value, reference=reference)
```

Difference

The difference algorithm is the absolute difference of the feature ROI and the reference or background ROI. You might prefer this algorithm if you want to have a strictly normal definition of CNR like this.

$$|feature - background|$$

Note: The absolute difference is used; i.e. the difference algorithm is symmetric.

Within pylinac, this is interpreted as:

$$I_{mean} - R_{mean}$$

where I is the ROI of the contrast region in question and R is the background/reference ROI, usually placed somewhere within the phantom area that is uniform.

An example calculation:

```
from pylinac.core import contrast

c = contrast.ratio(feature=10, background=5)
```

See also:

Wikipedia.

Root-mean-square

The RMS algorithm is another good general algorithm for evaluating contrast in myriad situations. It is defined as:

$$\sqrt{\frac{1}{M*N}*\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}(I_{i,j}-\bar{I})^2}$$

where an image/array is of size M by N. $\bar{I}$ is the average intensity of the image. $I_{i,j}$ is the element at the i-th and j-th position within the image array dimensions.

Warning: RMS calculations require the input values to be within the range 0 and 1. You might need to normalize your image/array first.

An example calculation:

```
from pylinac.core import contrast

my_roi = np.array((0.34, 0.67, 0.44, ...))
c = contrast.rms(my_roi)
```

See also:

Wikipedia

6.21.3 Visibility

Visibility is the ability for humans to detect signal against noise within a certain context. Visibility is a component of low contrast detectability. Traditionally, low contrast is evaluated irrespective of the size of the object. However, as a phantom like the Las Vegas or CatPhan 515 module shows, a large-sized object with small contrast might be seen, but a small-sized object of the same contrast might not. This is referred to as visibility. Visibility following the Rose model is defined as:

$$Visibility \approx C*\sqrt{Area*N}$$

where C is contrast and N is the number of photons.

Within pylinac, this equation is interpreted as:

$$Visibility(I,R) = Contrast(I,R)*\sqrt{Area(I)*DQE(I)} = Contrast(I,R)*\frac{\sqrt{\pi*radius^2}}{I_{std}}$$

where contrast is an option from the *low contrast methods* and $\pi*radius^2$ is the area of the ROI, which is assumed to be circular. I is the contrast image/array and R is the reference image/array.

Note: What is meant by "noise" is unclear in the literature. Technically, it was meant to be the detective quantum efficiency (DQE) which correlates to the number of photons counted. For simplicity and ease of understanding, the standard deviation, aka noise, of the ROI works as a simple inverse surrogate. I.e.

$$\sqrt{DQE(I)} \approx \sqrt{N} \approx \frac{1}{stdev_I}$$

Note: Pylinac ROIs are smaller than that actual size of the contrast ROI on the phantom. Uncertainty in the phantom detection algorithm means that the ROIs must be smaller to allow a small localization tolerance in the algorithm. Thus, visibility is a very specific number that depends on the size of the **sampling** ROI.

6.21.4 Contrast-to-noise ratio

The contrast to noise ratio (CNR) is defined as follows:

$$CNR(I) = \frac{Contrast(I)}{noise(I)} = \frac{Contrast(I)}{stdev(I)}$$

where contrast is an option from the low contrast methods. If you prefer the classic definition of CNR then use the "Difference" algorithm.

6.22 Modulation Transfer Function

The modulation transfer function (MTF) is used in CBCT and planar imaging metrics to describe high-contrast characteristics of the imaging system. An excellent introduction is here. In pylinac, MTF is calculated using equation 3 of the above reference, which is also the *Michelson* contrast definition.

$$contrast = \frac{I_{max} - I_{min}}{I_{max} + I_{min}}$$

Then, all the contrasts are normalized to the largest one, resulting in a normalized MTF or rMTF (relative). Pylinac only reports rMTF values. This is the first of two inputs. The other is the line pair spacing. The spacing is usually provided by the phantom manufacturer. The rMTF is the plotted against the line pair/mm values. Also from this data the MTF at a certain percentage (e.g. 50%) can be determined in units of lp/mm.

However, it's important to know what I_{max} and I_{min} means here. For a line pair set, each bar and space-between is one contrast value. Thus, one contrast value is calculated for each bar/space combo. For phantoms with areas of the same spacing (e.g. the Leeds), all bars and spaces are the same and thus we can use an area-based ROI for the input to the contrast equation.

6.23 Machine Scale

A machine scale or coordinate system is a specific reference framework used in the context of machines or equipment. It provides a standardized way to define positions, orientations, and measurements within the machine or equipment.

Other examples of coordinate systems are Cartesian, Polar, and Spherical.

Within the context of pylinac and machines, we're generally referring to IEC61217 or other systems defined by a manufacturer.

It is sometimes required to convert coordinate systems. An example is Dan Low's Winston-Lutz paper, where the coordinate system must be corrected for the equations to work correctly. See *here*.

6.23.1 Converting scales

To convert the values from one machine scale to another, use the *convert()* function.

```
from pylinac.core.scale import convert, MachineScale

gantry = 0
coll = 90
couch = 45

new_gantry, new_coll, new_couch = convert(
    input_scale=MachineScale.Varian,
    output_scale=MachineScale.IEC61217,
    gantry=gantry,
    collimator=coll,
    rotation=couch,
)
```

class pylinac.core.scale.**MachineScale**(*value*)

Bases: Enum

Possible machine scales. Used for specifying input and output scales for conversion. The enum keys are conversion functions for each axis relative to IEC 61217

pylinac.core.scale.**convert**(*input_scale:* MachineScale, *output_scale:* MachineScale, *gantry: float*, *collimator: float*, *rotation: float*)

Convert from one coordinate scale to another. Returns gantry, collimator, rotation.

6.24 Profiles & 1D Metrics

Profiles, in the context of pylinac, are 1D arrays of data that contain a single radiation field. Colloquially, these are what physicists might call an "inplane profile" or "crossplane profile" although the usage here is not restricted to those orientations.

6.24.1 Use Cases

Typical use cases for profiles are:

- Calculating a metric such as flatness, symmetry, penumbra, etc.
- Finding the center of the field.
- Finding the field edges.

6.24.2 Assumptions & Constraints

- There is one single, large radiation field in the profile.
- The radiation field should not be at the edge of the array.
- The radiation field should have a higher pixel value than the background. I.e. it should not be inverted.
- The field does not have to be normalized, but generally it should be. Plugins may assume normalized data.
- The field can be "horned" (i.e. have a dip in the middle) and/or contain a peak from an FFF field.
- The field can be off-center, but the penumbra should be fully contained in the array.
- The field can be skewed (e.g. a wedge field).
- The field can have any resolution or no resolution.

6.24.3 Basic Usage

Out of the box, the profile classes can be used to find the center of the field, field width, and the field edges.

```
from pylinac.core.profile import FWXMProfile

profile = FWXMProfile(..., fwxm_height=50)
print(profile.center_idx)  # print the center of the field position
print(profile.field_edge_idx(side="left"))  # print the left field edge position
print(profile.field_width_px)  # print the field width in pixels
profile.plot()  # plot the profile
```

However, the real power of the profile classes is in the plugins that can be used to calculate custom metrics. See the *plugins* section for more information.

6.24.4 Legacy vs New Classes

The legacy class for analyzing profiles is *SingleProfile*. This class is frozen and will not receive updates.

The modern classes for analyzing profiles are *FWXMProfile*, *InflectionDerivativeProfile*, *HillProfile*. These classes deal with **arrays only**.

For physical profiles, i.e. something where the values have a physical size or location like an EPID profile or water tank scan, use the following classes: *FWXMProfilePhysical*, *InflectionDerivativeProfilePhysical*, *HillProfilePhysical*.

Important: You will almost always want the `...Physical` variants of the classes as most profiles are from physical sources. Furthermore, some plugins will not work with the non-physical classes.

The difference between `SingleProfile` and the other classes is that `SingleProfile` is a swiss army knife. It can do almost everything the other classes can do and considerably more. However, the other classes are more specialized and thus more robust as well as a lot clearer and focused. Internally, pylinac uses the new specialized classes. The new classes also allow for *plugins* to be written and used much easier than with `SingleProfile`.

The `SingleProfile` class is more complicated to both read and use than the specialized classes. It's also harder to test and maintain. Thus, the specialized classes have come about as a response.

6.24.5 What class to use

Note: Throughout the documentation examples, the FWXM variety is used, but all examples can be replaced with the other variants.

The following list is a guide to what class to use:

- PDDs, TPRs, and similar depth-focused scans:
  - Use any of the classes.
- *FWXMProfile*/*FWXMProfilePhysical*
  - Use when the beam is "flat"; no FFF beams.
  - Can handle somewhat noisy data.
  - Robust to background noise and asymmetric background data.
- *InflectionDerivativeProfile*/*InflectionDerivativeProfilePhysical*:
  - Use either when the beam is flat or peaked.
  - Is not robust to salt and pepper noise or gently-sloping data.
  - The profile should have relatively sharp slopes compared to the background and central region.
  - This is a good default *ceteris parabus*.
- *HillProfile*/*HillProfilePhysical*:
  - Use either when the beam is flat or peaked.
  - It is robust to salt and pepper noise.
  - Do not use with sparse data such as IC profiler or similar where little data exists in the penumbra region.

6.24.6 Metric Plugins

The new profile classes discussed above allow for plugins to be written that can calculate metrics of the profile. For example, a penumbra plugin could be written that calculates the penumbra of the profile.

Several plugins are provided out of the box, and writing new plugins is straightforward.

Metrics are calculated by passing it as a list to the `calculate` method. The examples below show its usage.

Built-in Plugins

The following plugins are available out of the box:

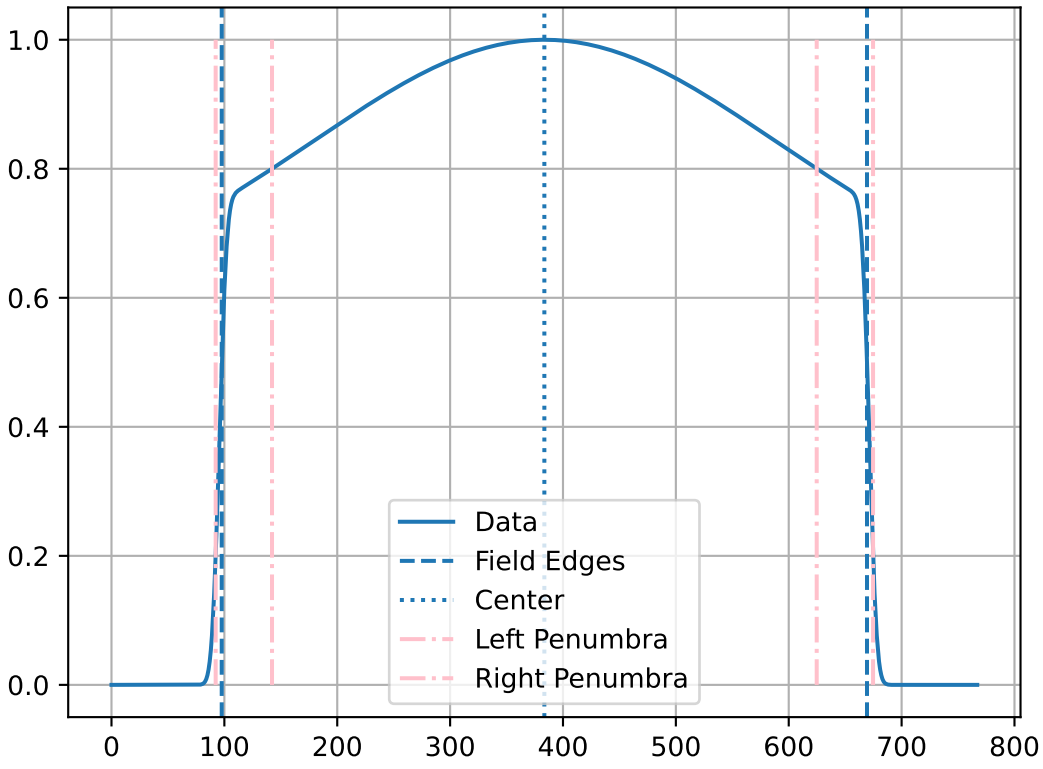
Penumbra Right

`PenumbraRightMetric` This plugin calculates the right penumbra of the profile. The upper and lower bounds can be passed in as arguments. The default is 80/20.

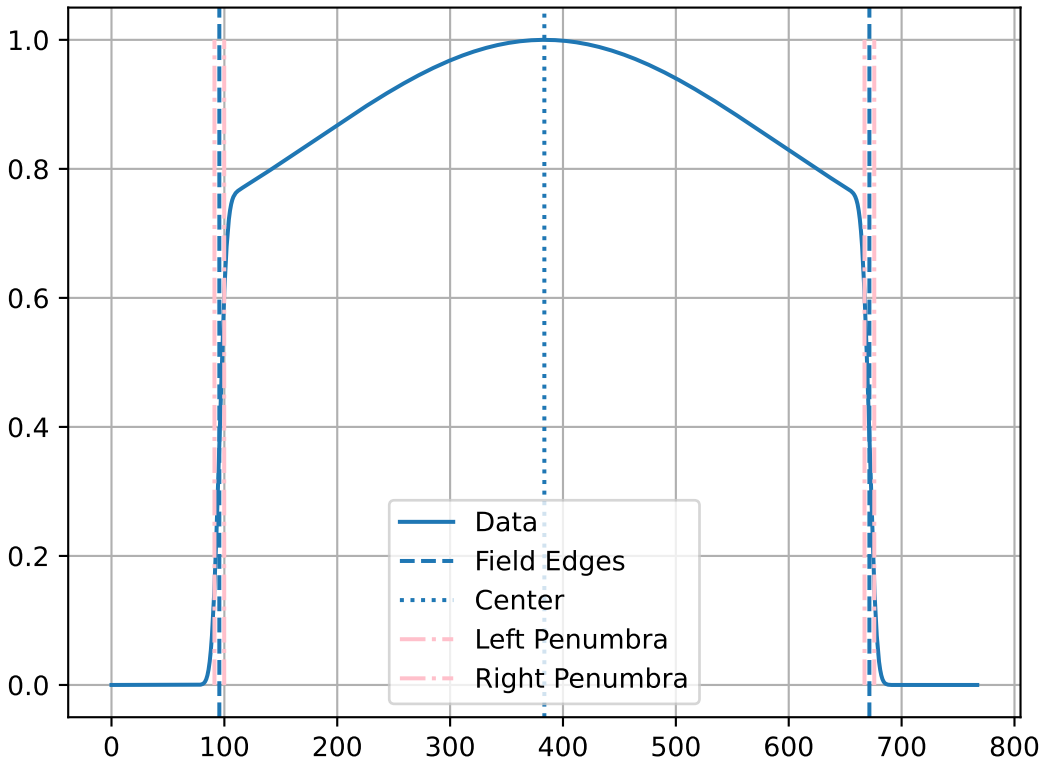
Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[PenumbraRightMetric(upper=80, lower=20)])
```

Note: When using Inflection derivative or Hill profiles, the penumbra is based on the height of the edge, not the maximum value of the profile. E.g. assume an FFF profile normalized to 1.0 and penumbra bounds of 80/20. The penumbra will not be at 50% height for Inflection derivative or Hill profiles. If it is detected at 0.4, or 40% height, the lower penumbra will be set to 20% (0.2) * 2 * 0.4, or 0.16. The upper penumbra will be 80% (0.8) * 2 * 0.4, or 0.64. This is because the penumbra bound is based on the height of the field edge, not the maximum value of the profile. This is best illustrated with a plot. We use the `FWXMProfilePhysical` class first to show its *inappropriate* use with FFF beams:



Note the upper penumbra is well-past the "shoulder" region and thus the penumbra is not accurate. Now let's use the `InflectionDerivativeProfilePhysical` class:



When analyzing flat beams, the `FWXMProfile` class is appropriate and will give similar results to the other two classes.

Penumbra Left

`PenumbraLeftMetric` This plugin calculates the left penumbra of the profile. The upper and lower bounds can be passed in as arguments. The default is 80/20.

Flatness (Difference)

`FlatnessDifferenceMetric` This plugin calculates the flatness difference of the profile. The in-field ratio can be passed in as an argument. The default is 0.8.

The flatness equation is:

$$flatness = 100 * \frac{D_{max} - D_{min}}{D_{max} + D_{min}} \in field$$

The equation does not track which side the flatness is higher or lower on. The value can range from 0 to 100. A perfect value is 0.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[FlatnessDifferenceMetric(in_field_ratio=0.8)])
```

Flatness (Ratio)

`FlatnessRatioMetric` This plugin calculates the flatness ratio of the profile. The in-field ratio can be passed in as an argument. The default is 0.8.

The flatness equation is:

$$flatness = 100 * \frac{D_{max}}{D_{min}} \in field$$

The equation does not track which side the flatness is higher or lower on. The value will range from 100 to ∞. A perfect value is 100.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[FlatnessRatioMetric(in_field_ratio=0.8)])
```

Symmetry (Point Difference)

`SymmetryPointDifferenceMetric` This plugin calculates the symmetry point difference of the profile. The in-field ratio can be passed in as an argument. The default is 0.8.

The symmetry point difference equation is:

$$symmetry = 100 * \frac{max(L_{pt} - R_{pt})}{D_{CAX}} \in field$$

where L_{pt} and R_{pt} are equidistant from the beam center. Symmetry can be positive or negative. The max refers to the point with the maximum difference between the left and right points. If the largest absolute value is negative, that is the value used.

Note: Unlike the point difference quotient, this metric is signed. A negative value means the right side is higher. A positive value means the left side is higher.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[SymmetryPointDifferenceMetric(in_field_ratio=0.8)])
```

Symmetry (Point Difference Quotient)

`SymmetryPointDifferenceQuotientMetric` This plugin calculates the symmetry point difference of the profile defined as the Point Difference Quotient (aka IEC). The in-field ratio can be passed in as an argument. The default is 0.8.

The symmetry point difference equation is:

$$symmetry = 100 * max(\frac{L_{pt}}{R_{pt}}, \frac{R_{pt}}{L_{pt}}) \in field$$

where L_{pt} and R_{pt} are equidistant from the beam center. This value can range from 100 to ∞. A perfect value is 100.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[SymmetryPointDifferenceQuotientMetric(in_field_ratio=0.8)])
```

Symmetry (Area)

`SymmetryAreaMetric` This plugin calculates the symmetry area of the profile. The in-field ratio can be passed in as an argument. The default is 0.8.

The symmetry area equation is:

$$symmetry = 100 * \frac{area_{left} - area_{right}}{area_{left} + area_{right}} \in field$$

where $area_{left}$ and $area_{right}$ are the areas under the left and right sides of the profile, centered about the beam center.

The value is signed. A negative value means the right side is higher and vice versa. The value can range from -100 to $+100$. A perfect value is 0.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[SymmetryAreaMetric(in_field_ratio=0.8)])
```

Top Position

`TopPositionMetric` This plugin calculates the distance from the "top" of the field to the beam center. This is typically used for FFF beams.

The calculation is based on the NCS-33 report. The central part of the field, by default the central 20%, is fitted to a 2nd order polynomial. The maximum of the polynomial is the "top" and the distance to the field center is calculated.

Example usage:

```
profile = FWXMProfile(...)
profile.compute(metrics=[TopDistanceMetric(top_region_ratio=0.2)])
```

Dmax

Dmax This plugin calculates the distance of the maximum value of the profile, usually called "Dmax".

A polynomial fit is used to find the maximum value of the profile. The maximum value of the polynomial fit is the determined Dmax. The window of the polynomial fit can be adjusted using the `window_mm` parameter.

```
profile = FWXMProfile(...)
profile.compute(metrics=[Dmax(window_mm=30)])
```

Important: It is expected that the x-values of the profile are given in mm! I.e. `FWXMProfile(..., x_values=...)`.

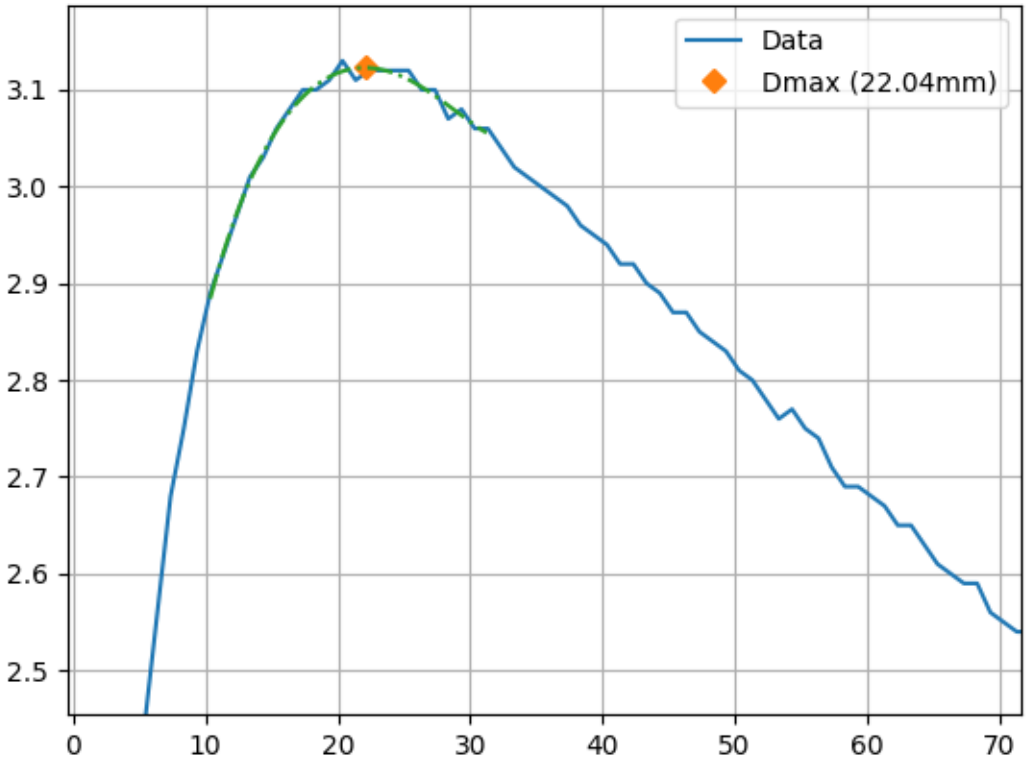
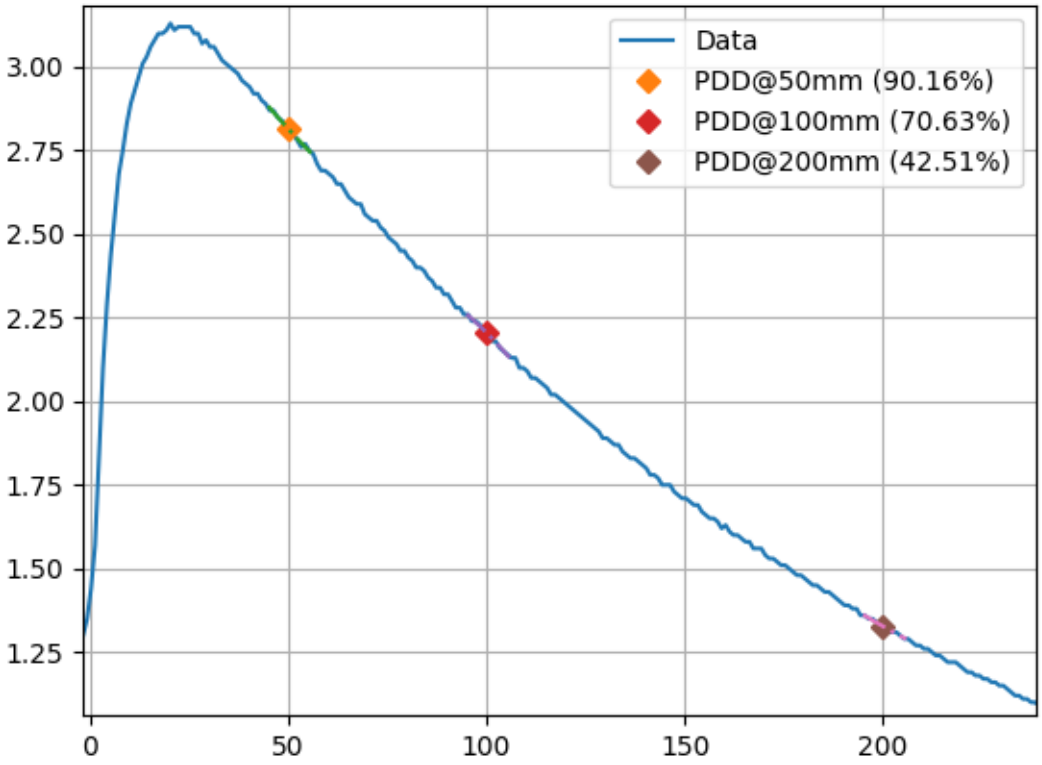


Fig. 7: Zoomed in plot of a profile showing the polynomial fit used to find Dmax.

PDD

PDD This plugin calculates the percentage depth dose (PDD) of the profile at a given depth. A polynomial fit is performed around the desired depth and then the value of the polynomial at the desired depth is returned.

```
profile = FWXMProfile(...)
profile.compute(metrics=[PDD(depth_mm=100)])
```



Ratio to Dmax

Since the PDD is a ratio of the maximum dose, the dmax is also calculated using, by default, a polynomial fit. I.e. if you ask for a PDD at 10 cm, two polynomial fits are done: one around 10 cm and one around the maximum and the ratio * 100 is the returned PDD. To override this behavior, set `normalize_to='max'`. Using `max` will simply normalize the depth value (still using a poly fit) to the maximum value of the profile.

Important: It is expected that the x-values of the profile are given in mm! I.e. `FWXMProfile(..., x_values=...)`.

Accessing metrics

There are two ways to access the metrics calculated by a profile (what is returned by the metric's `calculate` method). The first is what is returned by the `compute` method:

```
profile = FWXMProfile(...)
penum = profile.compute(metrics=PenumbraRightMetric())
print(penum)  # prints the penumbra value
```

We can also access the metric's calculation by accessing the `metric_values` attribute of the profile:

```
profile = FWXMProfile(...)
profile.compute(metrics=[PenumbraRightMetric()])
print(profile.metric_values["Right Penumbra"])  # prints the penumbra value
```

Note:

- The key within a profile's `metric_values` dictionary attribute is the value of the plugin's `name` attribute.
- Either 1 or multiple (as a list) metrics can be passed to the `compute` method.
- There are metrics included in pylinac. See the *built-in* section.

Writing plugins

To write a plugin, create a class with the following conditions:

- It inherits from `ProfileMetric`.
- It implements a `calculate()` method that returns something.
- It should also have a `name` attribute.

  Note: This can be handled either by a class attribute or dynamically using a property.

- (Optional) It implements a `plot` method can be declared that will plot the metric on the profile plot, although this is not required.

Note:

- Within the plugin, `self.profile` is available and will be the profile itself. This is so we can access the profile's attributes and methods.
- The `calculate` method can return anything, but a float is normal.
- The `plot` method must take a `matplotlib.plt.Axes` object as an argument and return nothing. But a plot method is optional.

Center index example

For an example, let us write a plugin that calculates the value of the center index and also plots it.

```
import matplotlib.pyplot as plt

from pylinac.core.profile import ProfileMetric


class CenterMetric(ProfileMetric):
    name = "Center Index"  # human-readable string

    def calculate(self) -> float:
        """Return the index of the center of the profile."""
        return self.profile.center_idx

    def plot(self, axis: plt.Axes) -> None:
        """Plot the center index."""
        axis.plot(
            self.profile.center_idx,
            self.profile.y_at_x(self.profile.center_idx),
            "o",
            color="red",
            markersize=10,
            label=self.name,
        )
```

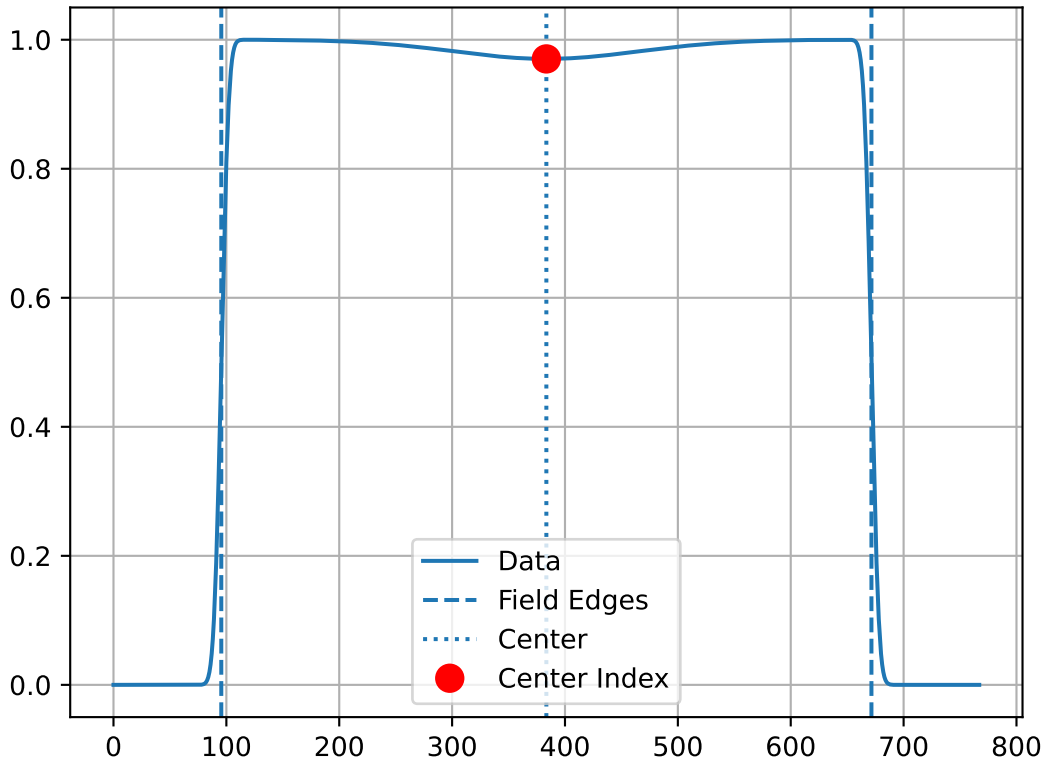
We can now pass this metric to the profile class' `compute` method. We will use the image generator to create an image we will extract a profile from.

```
import matplotlib.pyplot as plt

from pylinac.core.profile import FWXMProfile, ProfileMetric
from pylinac.core.image_generator import AS1000Image, FilteredFieldLayer,␣
↪GaussianFilterLayer
from pylinac.core.array_utils import normalize


# same as above; included so we can plot
class CenterMetric(ProfileMetric):
    name = 'Center Index'  # human-readable string

    def calculate(self) -> float:
        """Return the index of the center of the profile."""
        return self.profile.center_idx

    def plot(self, axis: plt.Axes) -> None:
        """Plot the center index."""
        axis.plot(self.profile.center_idx, self.profile.y_at_x(self.profile.center_idx),
↪'o', color='red',
                  markersize=10, label=self.name)

# this is our set up to get a nice profile
```

(continues on next page)

(continued from previous page)

```
as1000 = AS1000Image()
as1000.add_layer(
    FilteredFieldLayer(field_size_mm=(100, 100))
)
as1000.add_layer(
    GaussianFilterLayer(sigma_mm=2)
)  # add an image-wide gaussian to simulate penumbra/scatter

# pull out the profile array
array = normalize(as1000.image[:, as1000.shape[1] // 2])

# create the profile
profile = FWXMProfile(array)

# compute the metric with our plugin
profile.compute(metrics=CenterMetric())

# plot the profile
profile.plot()
```



6.24.7 Resampling

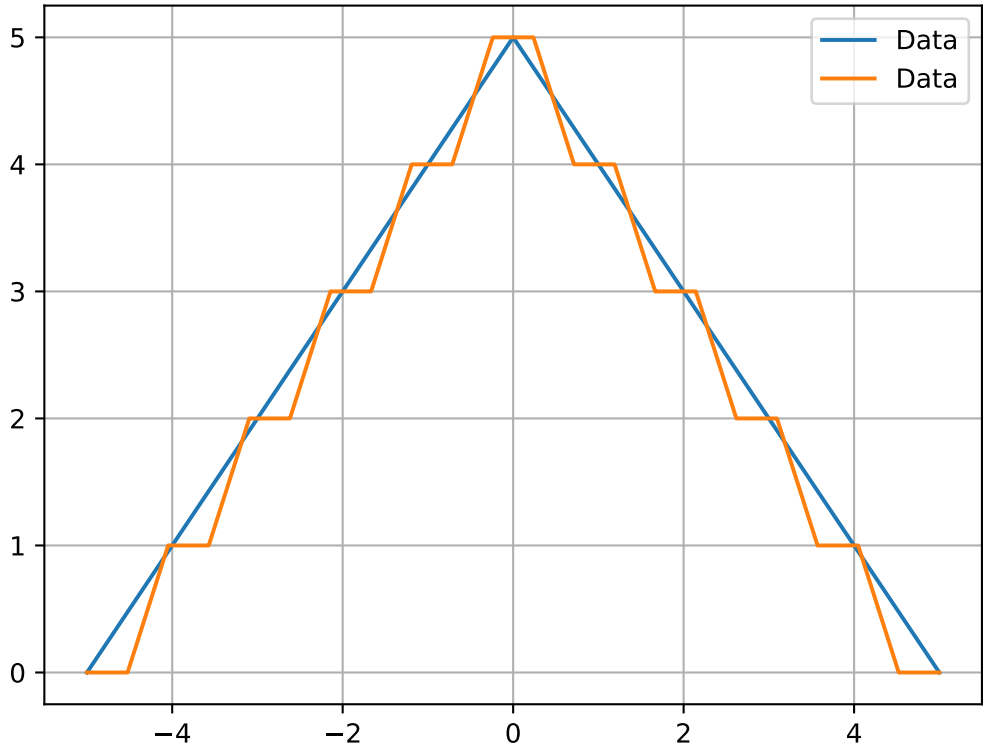
Resampling a profile is the process of interpolating the profile data to a new resolution and can be done easily using `as_resampled`:

```
from pylinac.core.profile import FWXMProfilePhysical

profile = FWXMProfilePhysical(my_array, dpmm=3)
profile_resampled = profile.as_resampled(interpolation_resolution_mm=0.1)
```

This will create a **new** profile that is resampled to 0.1 mm resolution. The new profile's `dpmm` attribute is also updated. The original profile is not modified.

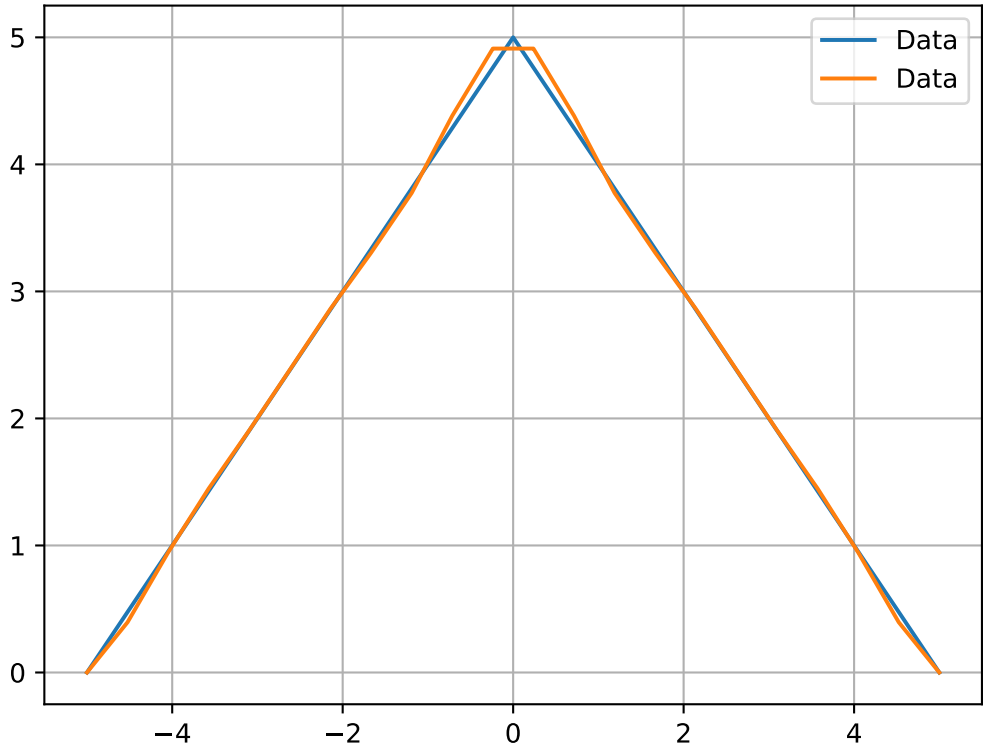
Warning: Resampling will respect the input datatype. If the array is an integer type and has a small range, the resampled array may be truncated. For example, if the array is an unsigned 16-bit integer (native EPID) and the range of values varies from 100 to 200, the resampled array will appear to be step-wise.

```
import numpy as np
from matplotlib import pyplot as plt

from pylinac.core.profile import FWXMProfile

y = np.array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0], dtype=int)
x = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], dtype=int)

prof = FWXMProfile(values=y, x_values=x)
prof_interp = prof.as_resampled(interpolation_factor=2)
ax = prof.plot(show=False, show_field_edges=False, show_center=False)
prof_interp.plot(show=True, axis=ax, show_field_edges=False, show_center=False)
```



Compare this to a float array:

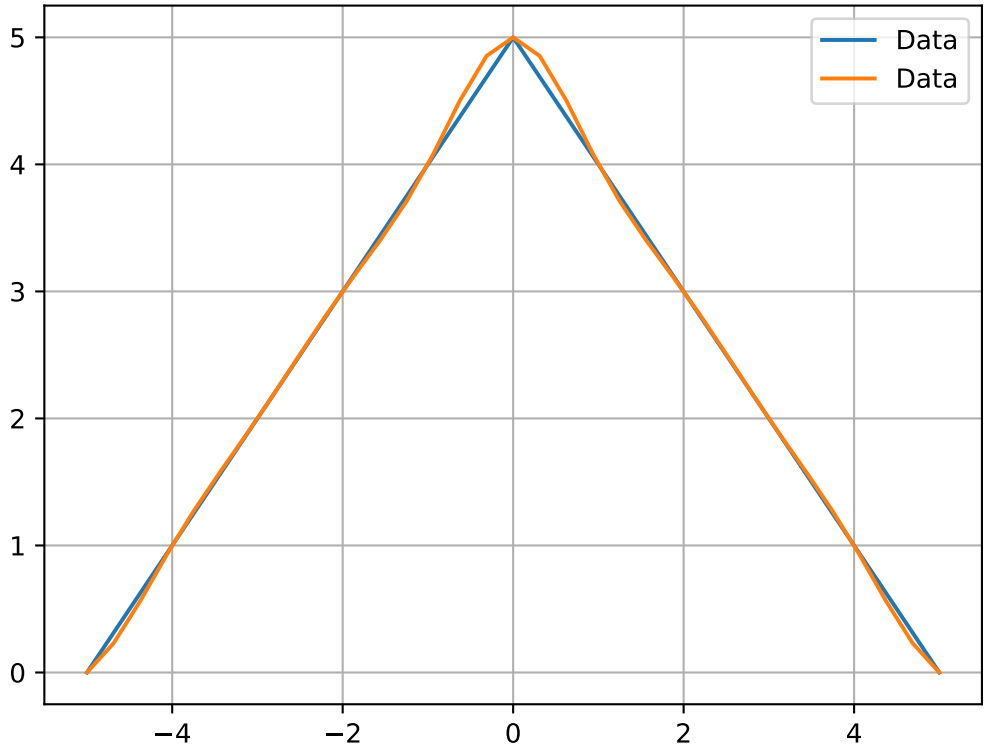
```
import numpy as np
from matplotlib import pyplot as plt

from pylinac.core.profile import FWXMProfile

y = np.array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0], dtype=float)
x = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], dtype=float)

prof = FWXMProfile(values=y, x_values=x)
prof_interp = prof.as_resampled(interpolation_factor=2)
ax = prof.plot(show=False, show_field_edges=False, show_center=False)
prof_interp.plot(show=True, axis=ax, show_field_edges=False, show_center=False)
```



This float array is interpolated better, although there is still some apparent spline interpolation fit error.

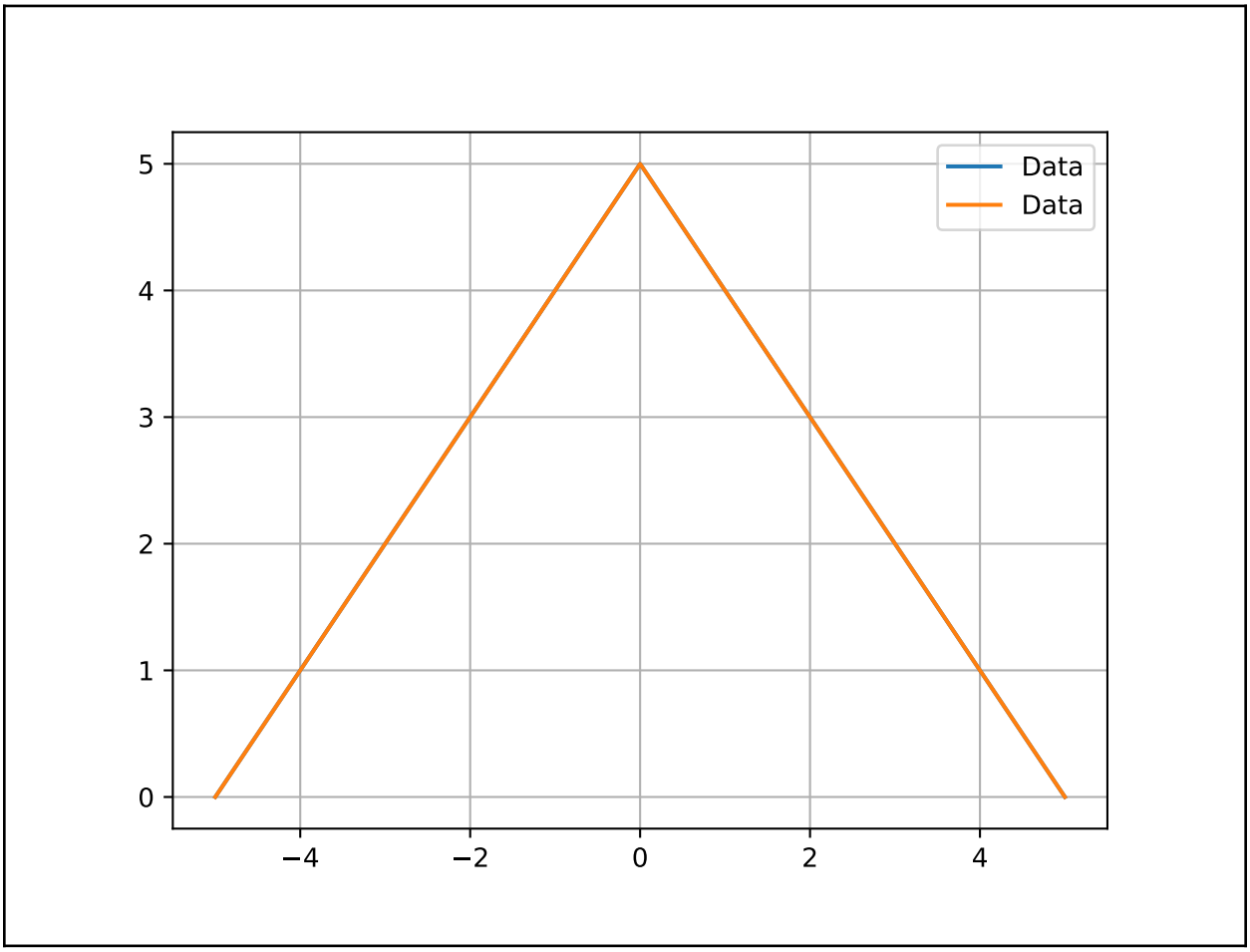
This second issue can be resolved by using an odd-sized interpolation factor:

```
import numpy as np
from matplotlib import pyplot as plt

from pylinac.core.profile import FWXMProfile

y = np.array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0], dtype=float)
x = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], dtype=float)

prof = FWXMProfile(values=y, x_values=x)
prof_interp = prof.as_resampled(interpolation_factor=3)  # not 2
ax = prof.plot(show=False, show_field_edges=False, show_center=False)
prof_interp.plot(show=True, axis=ax, show_field_edges=False, show_center=False)
```



Better, but still not perfect. Most profiles do not look like this however. This is an extreme example. However, even here we can improve things by using linear interpolation. This is done by setting the `order` parameter to 1:

```
import numpy as np
from matplotlib import pyplot as plt

from pylinac.core.profile import FWXMProfile

y = np.array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0], dtype=float)
x = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], dtype=float)

prof = FWXMProfile(values=y, x_values=x)
prof_interp = prof.as_resampled(interpolation_factor=3, order=1)  # order=1 => linear
ax = prof.plot(show=False, show_field_edges=False, show_center=False)
prof_interp.plot(show=True, axis=ax, show_field_edges=False, show_center=False)
```



Note: Resampling can be used for both upsampling and downsampling.

Important: The parameters for `as_resampled` are slightly different between the physical and non-physical classes. For physical classes, the new resolution is in mm/pixels. For non-physical classes, the new resolution is a simple factor like 5x or 10x the original resolution.

Important: Resampling is not the same as smoothing. Smoothing is the process of removing noise from the profile. Resampling is the process of changing the resolution of the profile. To apply a filter, use the *`filter()`* method:

```
from pylinac.core.profile import FWXMProfile

profile = FWXMProfile(...)
profile.filter(size=5, kind="gaussian")
```

Warning: When resampling a physical profile, it is important to know that interpolation must account for the physical size of the pixels and how that affects the edge of the array. Simply resampling the array without accounting

for the physical size of the pixels will result in a profile that is not accurate at the edges. The simplest way to visualize this is shown in the `grid_mode` parameter of scipy's zoom function.

Multiple resampling

Profiles can be resampled multiple times, but it is important to set `grid_mode=False` on secondary resamplings. This is because the physical size of the pixels is already accounted for in the first resampling. If `grid_mode=True` is used on secondary resamplings, the profile edges will not accurately represent the physical size and position of the pixels:

```
from pylinac.core.profile import FWXMProfilePhysical

profile = FWXMProfilePhysical(my_array, dpmm=3)
profile_resampled = profile.as_resampled(interpolation_resolution_mm=0.1)
# use grid_mode=False on secondary resamplings
profile_resampled2 = profile_resampled.as_resampled(
    interpolation_resolution_mm=0.05, grid_mode=False
)

# if we resample to 0.05mm directly from the original it will be the same as the above
profile_resampled3 = profile.as_resampled(interpolation_resolution_mm=0.05)
assert len(profile_resampled2) == len(profile_resampled3)
# assert the left edge is at the same physical location
assert profile_resampled2.x_values[0] == profile_resampled3.x_values[0]
```

6.24.8 API

class pylinac.core.profile.**SingleProfile**(*values: np.ndarray*, *dpmm: float = None*, *interpolation: Interpolation | str | None = Interpolation.LINEAR*, *ground: bool = True*, *interpolation_resolution_mm: float = 0.1*, *interpolation_factor: float = 10*, *normalization_method: Normalization | str = Normalization.BEAM_CENTER*, *edge_detection_method: Edge | str = Edge.FWHM*, *edge_smoothing_ratio: float = 0.003*, *hill_window_ratio: float = 0.1*, *x_values: np.ndarray | None = None*)

Bases: `ProfileMixin`

A profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

Parameters

values
The profile numpy array. Must be 1D.

dpmm
The dots (pixels) per mm. Pass to get info like beam width in distance units in addition to pixels

interpolation
Interpolation technique.

ground
Whether to ground the profile (set min value to 0). Helpful most of the time.

interpolation_resolution_mm
The resolution that the interpolation will scale to. **Only used if dpmm is passed and interpolation is set**. E.g. if the dpmm is 0.5 and the resolution is set to 0.1mm the data will be interpolated to have a new dpmm of 10 (1/0.1).

interpolation_factor
The factor to multiply the data by. **Only used if interpolation is used and dpmm is NOT passed**. E.g. 10 will perfectly decimate the existing data according to the interpolation method passed.

normalization_method
How to pick the point to normalize the data to.

edge_detection_method
The method by which to detect the field edge. FWHM is reasonable most of the time except for FFF beams. Inflection-derivative will use the max gradient to determine the field edge. Note that this may not be the 50% height. In fact, for FFF beams it shouldn't be. Inflection methods are better for FFF and other unusual beam shapes.

edge_smoothing_ratio
Only applies to INFLECTION_DERIVATIVE and INFLECTION_HILL.

The ratio of the length of the values to use as the sigma for a Gaussian filter applied before searching for the inflection. E.g. 0.005 with a profile of 1000 points will result in a sigma of 5. This helps make the inflection point detection more robust to noise. Increase for noisy data.

hill_window_ratio
The ratio of the field size to use as the window to fit the Hill function. E.g. 0.2 will using a window centered about each edge with a width of 20% the size of the field width. **Only applies when the edge detection is INFLECTION_HILL**.

x_values
The x-values of the profile, if any. If None, will generate a simple range(len(values)).

`resample`(*interpolation_factor: int = 10*, *interpolation_resolution_mm: float = 0.1*) → *SingleProfile*

Resample the profile at a new resolution. Returns a new profile

`geometric_center`() → dict

The geometric center (i.e. the device center)

`beam_center`() → dict

The center of the detected beam. This can account for asymmetries in the beam position (e.g. offset jaws)

`fwxm_data`(*x: int = 50*) → dict

Return the width at X-Max, where X is the percentage height.

Parameters

x
The percent height of the profile. E.g. x = 50 is 50% height, i.e. FWHM.

`field_data`(*in_field_ratio: float = 0.8*, *slope_exclusion_ratio=0.2*) → dict

Return the width at X-Max, where X is the percentage height.

Parameters

in_field_ratio
In Field Ratio: 1.0 is the entire detected field; 0.8 would be the central 80%, etc.

slope_exclusion_ratio
Ratio of the field width to use as the cutoff between "top" calculation and "slope" calculation. Useful for FFF beams. This area is centrally located in the field. E.g. 0.2 will use the central 20% of the field to calculate the "top" value. To calculate the slope of each side, the field width between the edges of the in_field_ratio and the slope exclusion region are used.

> **Warning:** The "top" value is always calculated. For FFF beams this should be reasonable, but for flat beams this value may end up being non-sensible.

`inflection_data`() → dict

Calculate the profile inflection values using either the 2nd derivative or a fitted Hill function.

> **Note:** This only applies if the edge detection method is *INFLECTION_...*.

Parameters

`penumbra`(*lower: int = 20*, *upper: int = 80*)

Calculate the penumbra of the field. Dependent on the edge detection method.

Parameters

lower
The lower % of the beam to use. If the edge method is FWHM, this is the typical % penumbra you're thinking. If the inflection method is used it will be the value/50 of the inflection point value. E.g. if the inflection point is perfectly at 50% with a `lower` of 20, then the penumbra value here will be 20% of the maximum. If the inflection point is at 30% of the max value (say for a FFF beam) then the lower penumbra will be `lower/50` of the inflection point or `0.3*lower/50`.

upper
Upper % of the beam to use. See lower for details.

`field_calculation`(*in_field_ratio: float = 0.8*, *calculation: str = 'mean'*, *slope_exclusion_ratio: float = 0.2*) → float | tuple[float, float]

Perform an operation on the field values of the profile. This function is useful for determining field symmetry and flatness.

Parameters

in_field_ratio
Ratio of the field width to use in the calculation.

calculation
[{'mean', 'median', 'max', 'min', 'area'}] Calculation to perform on the field values.

gamma(*evaluation_profile: SingleProfile*, *distance_to_agreement: float = 1*, *dose_to_agreement: float = 1*, *gamma_cap_value: float = 2*, *dose_threshold: float = 5*, *global_dose: bool = True*, *fill_value: float = nan*) → ndarray

Calculate a 1D gamma. The passed profile is the evaluation profile. The instance calling this method is the reference profile. This profile must have the *dpmm* value given at instantiation so that physical spacing can be evaluated. The evaluation profile is resampled to be the same resolution as the reference profile.

Note: The difference between this method and the *gamma_1d* function is that 1) this is computed on Profile instances and 2) this validates the physical spacing of the profiles.

Parameters

evaluation_profile
The evaluation profile. This profile must have the *dpmm* value given at instantiation so that physical spacing can be evaluated.

distance_to_agreement
Distance in **mm** to search

dose_to_agreement
Dose in % of either global or local reference dose

gamma_cap_value
The value to cap the gamma at. E.g. a gamma of 5.3 will get capped to 2. Useful for displaying data with a consistent range.

global_dose
Whether to evaluate the dose to agreement threshold based on the global max or the dose point under evaluation.

dose_threshold
The dose threshold as a number between 0 and 100 of the % of max dose under which a gamma is not calculated. This is not affected by the global/local dose normalization and the threshold value is evaluated against the global max dose, period.

fill_value
The value to give pixels that were not calculated because they were under the dose threshold. Default is NaN, but another option would be 0. If NaN, allows the user to calculate mean/median gamma over just the evaluated portion and not be skewed by 0's that should not be considered.

plot(*show: bool = True*) → None

Plot the profile.

bit_invert() → None

Invert the profile bit-wise.

convert_to_dtype(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size

[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind

[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float

Ground the profile such that the lowest value is 0.

Returns

float

The minimum value that was used as the grounding value.

invert() → None

Invert the profile.

normalize(*norm_val: str | float | None = None*) → None

Normalize the profile to the given value.

Parameters

norm_val

[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

stretch(*min: float = 0*, *max: float = 1*) → None

'Stretch' the profile to the min and max parameter values.

Parameters

min

[number] The new minimum of the values

max

[number] The new maximum value.

class pylinac.core.profile.**FWXMProfile**(*values: np.array*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str | Normalization = Normalization.NONE*, *fwxm_height: float = 50*)

Bases: `ProfileBase`

A profile that has one large signal, e.g. a radiation beam profile and data derived from it is based on the Full-Width X-Maximum to find the edge indices

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

field_edge_idx(*side: Literal['right', 'left']*) → float

The edge index of the given side using the FWXM methodology

as_resampled(*interpolation_factor: float = 10*, *order: int = 3*) → *FWXMProfile*

Resample the profile at a new resolution. Returns a new profile.

Parameters

interpolation_factor
[float] The factor to zoom the profile by. E.g. 10 means the profile will be 10x larger.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

bit_invert() → None

Invert the profile bit-wise.

property **center_idx: float**

The center index of the profile. Halfway between the field edges.

compute(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

convert_to_dtype(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

field_indices(*in_field_ratio: float*)

Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

field_values(*in_field_ratio: float = 0.8*) → ndarray

The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

property field_width_px: float

The field width of the profile in pixels

field_x_values(*in_field_ratio: float*) → ndarray

Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size

[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind

[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float

Ground the profile such that the lowest value is 0.

Returns

float

The minimum value that was used as the grounding value.

invert() → None

Invert the profile.

normalize(*norm_val: str | float | None = None*) → None

Normalize the profile to the given value.

Parameters

norm_val

[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes

Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None

'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

x_at_x(*x: float*) → ndarray

Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float

Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float

Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int

Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float

Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.core.profile.**FWXMProfilePhysical**(*values: np.array*, *dpmm: float*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str | Normalization = Normalization.NONE*, *fwxm_height: float = 50*)

Bases: PhysicalProfileMixin, *FWXMProfile*

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

as_resampled(*interpolation_resolution_mm: float = 0.1*, *order: int = 3*, *grid: bool = True*) → *FWXMProfilePhysical*

Resample the physical profile at a new resolution. Returns a new profile.

Parameters

interpolation_resolution_mm
[float] The resolution to resample to in mm. E.g. 0.1 means the profile will be 0.1 mm resolution.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

grid
[bool] Whether to use grid mode when zooming. See parameter `grid_mode` in `zoom()` for more information. This should be true unless you are resampling an already-resampled physical array.

Warnings

This method will respect the input datatype of the numpy array. If the input array is a float, the output array will be a float. This can cause issues for int arrays with a small range. E.g. if the range is only 10, interpolation will look more step-like than smooth. If this is the case, convert the array to a float before passing it to this method. The array is not automatically converted to float in this case to respect the original dtype. However, a warning will be produced.

bit_invert() → None

Invert the profile bit-wise.

property center_idx: float

The center index of the profile. Halfway between the field edges.

compute(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

convert_to_dtype(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

field_edge_idx(*side: Literal['right', 'left']*) → float

The edge index of the given side using the FWXM methodology

field_indices(*in_field_ratio: float*)

Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

field_values(*in_field_ratio: float = 0.8*) → ndarray

The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

property field_width_mm: float

The field width of the profile in mm

property field_width_px: float

The field width of the profile in pixels

field_x_values(*in_field_ratio: float*) → ndarray

Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size
[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float
Ground the profile such that the lowest value is 0.

Returns

float
The minimum value that was used as the grounding value.

invert() → None
Invert the profile.

normalize(*norm_val: str | float | None = None*) → None
Normalize the profile to the given value.

Parameters

norm_val
[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes
Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None
'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

x_at_x(*x: float*) → ndarray
Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float
Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float

Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int

Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float

Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.core.profile.**InflectionDerivativeProfile**(*values: np.array*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str | Normalization = Normalization.NONE*, *edge_smoothing_ratio: float = 0.003*)

Bases: `ProfileBase`

A profile that has one large signal, e.g. a radiation beam profile and data derived from it is based on the Full-Width X-Maximum

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

field_edge_idx(*side: str*) → float

The edge index of the given side using the second derivative methodology

as_resampled(*interpolation_factor: float = 10*, *order: int = 3*) → *InflectionDerivativeProfile*

Resample the profile at a new resolution. Returns a new profile.

Parameters

interpolation_factor
[float] The factor to zoom the profile by. E.g. 10 means the profile will be 10x larger.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

Warnings

This method will respect the input datatype of the numpy array. If the input array is a float, the output array will be a float. This can cause issues for int arrays with a small range. E.g. if the range is only 10, interpolation will look more step-like than smooth. If this is the case, convert the array to a float before passing it to this method. The array is not automatically converted to float in this case to respect the original dtype. However, a warning will be produced.

bit_invert() → None

Invert the profile bit-wise.

property center_idx: float

The center index of the profile. Halfway between the field edges.

compute(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

`convert_to_dtype`(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

`field_indices`(*in_field_ratio: float*)

Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

`field_values`(*in_field_ratio: float = 0.8*) → ndarray

The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

`property field_width_px: float`

The field width of the profile in pixels

`field_x_values`(*in_field_ratio: float*) → ndarray

Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

`filter`(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size
[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

`ground`() → float

Ground the profile such that the lowest value is 0.

Returns

float
The minimum value that was used as the grounding value.

invert() → None
Invert the profile.

normalize(*norm_val: str | float | None = None*) → None
Normalize the profile to the given value.

Parameters

norm_val
[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes
Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None
'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

x_at_x(*x: float*) → ndarray
Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float
Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float
Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int
Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float
Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.core.profile.**InflectionDerivativeProfilePhysical**(*values: np.array*, *dpmm: float*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str | Normalization = Normalization.NONE*, *edge_smoothing_ratio: float = 0.003*)

Bases: PhysicalProfileMixin, *InflectionDerivativeProfile*

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

as_resampled(*interpolation_resolution_mm: float = 0.1*, *order: int = 3*, *grid: bool = True*) → *InflectionDerivativeProfilePhysical*

Resample the physical profile at a new resolution. Returns a new profile.

Parameters

interpolation_resolution_mm
[float] The resolution to resample to in mm. E.g. 0.1 means the profile will be 0.1 mm resolution.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

grid
[bool] Whether to use grid mode when zooming. See parameter `grid_mode` in `zoom()` for more information. This should be true unless you are resampling an already-resampled physical array.

Warnings

This method will respect the input datatype of the numpy array. If the input array is a float, the output array will be a float. This can cause issues for int arrays with a small range. E.g. if the range is only 10, interpolation will look more step-like than smooth. If this is the case, convert the array to a float before passing it to this method. The array is not automatically converted to float in this case to respect the original dtype. However, a warning will be produced.

bit_invert() → None

Invert the profile bit-wise.

property center_idx: float

The center index of the profile. Halfway between the field edges.

compute(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

convert_to_dtype(*dtype: type[dtype]*) → None
Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

field_edge_idx(*side: str*) → float
The edge index of the given side using the second derivative methodology

field_indices(*in_field_ratio: float*)
Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

field_values(*in_field_ratio: float = 0.8*) → ndarray
The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

property field_width_mm: float
The field width of the profile in mm

property field_width_px: float
The field width of the profile in pixels

field_x_values(*in_field_ratio: float*) → ndarray
Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None
Filter the profile.

Parameters

size
[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float
Ground the profile such that the lowest value is 0.

Returns

float
The minimum value that was used as the grounding value.

invert() → None
Invert the profile.

normalize(*norm_val: str | float | None = None*) → None
Normalize the profile to the given value.

Parameters

norm_val
[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes

Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None

'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

x_at_x(*x: float*) → ndarray

Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float

Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float

Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int

Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float

Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.core.profile.**HillProfile**(*values: np.array*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str = Normalization.NONE*, *edge_smoothing_ratio: float = 0.003*, *hill_window_ratio: float = 0.1*)

Bases: *InflectionDerivativeProfile*

A profile that has one large signal, e.g. a radiation beam profile and data derived from it is based on the Full-Width X-Maximum

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

field_edge_idx(*side: str*) → float

The edge index of the given side using the FWXM methodology

as_resampled(*interpolation_factor: float = 10*, *order: int = 3*) → *HillProfile*

Resample the profile at a new resolution. Returns a new profile.

Parameters

interpolation_factor
[float] The factor to zoom the profile by. E.g. 10 means the profile will be 10x larger.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

Warnings

This method will respect the input datatype of the numpy array. If the input array is a float, the output array will be a float. This can cause issues for int arrays with a small range. E.g. if the range is only 10, interpolation will look more step-like than smooth. If this is the case, convert the array to a float before passing it to this method. The array is not automatically converted to float in this case to respect the original dtype. However, a warning will be produced.

`bit_invert`() → None

Invert the profile bit-wise.

`property center_idx: float`

The center index of the profile. Halfway between the field edges.

`compute`(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

`convert_to_dtype`(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

`field_indices`(*in_field_ratio: float*)

Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

`field_values`(*in_field_ratio: float = 0.8*) → ndarray

The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

`property field_width_px: float`

The field width of the profile in pixels

`field_x_values`(*in_field_ratio: float*) → ndarray

Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size
: [float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
: [{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float

Ground the profile such that the lowest value is 0.

Returns

float
: The minimum value that was used as the grounding value.

invert() → None

Invert the profile.

normalize(*norm_val: str | float | None = None*) → None

Normalize the profile to the given value.

Parameters

norm_val
: [number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes

Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None

'Stretch' the profile to the min and max parameter values.

Parameters

min
: [number] The new minimum of the values

max
: [number] The new maximum value.

x_at_x(*x: float*) → ndarray

Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float

Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float

Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int

Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float

Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.core.profile.**HillProfilePhysical**(*values: np.array*, *dpmm: float*, *x_values: np.array | None = None*, *ground: bool = False*, *normalization: str | Normalization = Normalization.NONE*, *edge_smoothing_ratio: float = 0.003*, *hill_window_ratio: float = 0.1*)

Bases: PhysicalProfileMixin, *HillProfile*

A 1D profile that has one large signal, e.g. a radiation beam profile. Signal analysis methods are given, mostly based on FWXM and on Hill function calculations. Profiles with multiple peaks are better suited by the MultiProfile class.

as_resampled(*interpolation_resolution_mm: float = 0.1*, *order: int = 3*, *grid: bool = True*) → *HillProfilePhysical*

Resample the physical profile at a new resolution. Returns a new profile.

Parameters

interpolation_resolution_mm
[float] The resolution to resample to in mm. E.g. 0.1 means the profile will be 0.1 mm resolution.

order
[int] The order of the spline interpolation. 1 is linear, 3 is cubic, etc.

grid
[bool] Whether to use grid mode when zooming. See parameter `grid_mode` in `zoom()` for more information. This should be true unless you are resampling an already-resampled physical array.

Warnings

This method will respect the input datatype of the numpy array. If the input array is a float, the output array will be a float. This can cause issues for int arrays with a small range. E.g. if the range is only 10, interpolation will look more step-like than smooth. If this is the case, convert the array to a float before passing it to this method. The array is not automatically converted to float in this case to respect the original dtype. However, a warning will be produced.

bit_invert() → None

Invert the profile bit-wise.

property center_idx: float

The center index of the profile. Halfway between the field edges.

compute(*metrics: Iterable[ProfileMetric] | ProfileMetric*) → Any | dict[str, Any]

Compute metric(s) on the profile.

Unlike other modules, calling `compute` is not strictly necessary. Only call it if there are metrics to calculate.

Parameters

metrics: iterable of ProfileMetric | ProfileMetric
List of metrics to calculate. If only one metric is desired, it can be passed directly.

Returns

dict | list
A dictionary of metric names and values if multiple metrics were given. If only one metric was given, the value of that metric is returned.

convert_to_dtype(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

field_edge_idx(*side: str*) → float

The edge index of the given side using the FWXM methodology

field_indices(*in_field_ratio: float*)

Return the indices of the left and right edge of the field, given the in-field ratio. Importantly, this will use the same rounding behavior as field_values.

field_values(*in_field_ratio: float = 0.8*) → ndarray

The array of values of the profile within the 'field' area. This is typically 80% of the detected field width.

property field_width_mm: float

The field width of the profile in mm

property field_width_px: float

The field width of the profile in pixels

field_x_values(*in_field_ratio: float*) → ndarray

Return the x-values of the field, given the in-field ratio. This is helpful when plotting the field to include the proper x-values as well.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size
[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

ground() → float

Ground the profile such that the lowest value is 0.

Returns

float
The minimum value that was used as the grounding value.

invert() → None

Invert the profile.

normalize(*norm_val: str | float | None = None*) → None

Normalize the profile to the given value.

Parameters

norm_val
[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*show: bool = True*, *axis: plt.Axes | None = None*, *show_field_edges: bool = True*, *show_grid: bool = True*, *show_center: bool = True*) → plt.Axes

Plot the profile along with relevant overlays to point out features.

stretch(*min: float = 0*, *max: float = 1*) → None

'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

x_at_x(*x: float*) → ndarray

Deprecated alias for x_at_x_idx

x_at_x_idx(*x: float | np.ndarray*) → np.ndarray | float

Return the physical x-value at the given index. When no x-values are provided, these are the same. However, physical dimensions can be different than the index.

x_at_y(*y: float | np.ndarray*, *side: str*) → np.ndarray | float

Interpolated y-values. Can use floats as indices.

x_idx_at_x(*x: float*) → int

Return the **index** of the x-value closest to the given x-value.

y_at_x(*x: float | np.ndarray*) → np.ndarray | float

Interpolated y-values. The x-value is the physical position, not the index. However, if no x-values were provided, these will be the same.

class pylinac.metrics.profile.**PenumbraRightMetric**(*lower: float = 20*, *upper: float = 80*, *color='pink'*, *ls='-.'*)

Bases: *PenumbraLeftMetric*

calculate() → float

Calculate the left penumbra in mm. We first find the edge point and then return the distance from the lower penumbra value to upper penumbra value. The trick is that wherever the field edge is, is assumed to be 50% height. It's okay if it's not actually (like for FFF).

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

plot(*axis: Axes*)

Plot the metric on the given axis.

class pylinac.metrics.profile.**PenumbraLeftMetric**(*lower: float = 20*, *upper: float = 80*, *color='pink'*, *ls='-.'*)

Bases: ProfileMetric

calculate() → float

Calculate the left penumbra in mm. We first find the edge point and then return the distance from the lower penumbra value to upper penumbra value. The trick is that wherever the field edge is, is assumed to be 50% height. It's okay if it's not actually (like for FFF).

plot(*axis: Axes*)

Plot the metric on the given axis.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.metrics.profile.**SymmetryPointDifferenceMetric**(*in_field_ratio: float = 0.8*, *color='magenta'*, *linestyle='--'*, *max_sym_range: float = 2*, *min_sym_range: float = -2*)

Bases: ProfileMetric

Symmetry using the point difference method.

calculate() → float

Calculate the symmetry ratio of the profile.

plot(*axis: plt.Axes*, *markers: str, str = ('^', 'v')*) → None

Plot the metric on the given axis.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.metrics.profile.**SymmetryPointDifferenceQuotientMetric**(*in_field_ratio: float = 0.8*, *color='magenta'*, *linestyle='--'*, *max_sym_range: float = 2*, *min_sym_range: float = 0*)

Bases: *SymmetryPointDifferenceMetric*

Symmetry as defined by IEC.

plot(*axis: plt.Axes*, *markers: str, str = ('x', 'x')*) → None

Plot the metric on the given axis.

calculate() → float

Calculate the symmetry ratio of the profile.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.metrics.profile.**TopDistanceMetric**(*top_region_ratio: float = 0.2*, *color='orange'*)

Bases: ProfileMetric

The distance from an FFF beam's "top" to the center of the field. Similar, although not 100% faithful to NCS-33. The NCS report uses the middle 5cm but we use a field ratio. In practice, this shouldn't make a difference.

calculate() → float

Calculate the distance from the top to the field center. Positive means the top is to the right, negative means the top is to the left.

plot(*axis: Axes*)

Plot the top point and the fitted curve.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.metrics.profile.**FlatnessRatioMetric**(*in_field_ratio: float = 0.8*, *color='g'*, *linestyle='-.'*)

Bases: *FlatnessDifferenceMetric*

Flatness as (apparently) defined by IEC.

calculate() → float

Calculate the flatness ratio of the profile.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

plot(*axis: Axes*) → None

Plot the points of largest flatness difference as well as the search bounding box.

class pylinac.metrics.profile.**FlatnessDifferenceMetric**(*in_field_ratio: float = 0.8*, *color='g'*, *linestyle='-.'*)

Bases: ProfileMetric

Flatness as defined by IAEA Rad Onc Handbook pg 196: https://www-pub.iaea.org/MTCD/Publications/PDF/Pub1196_web.pdf

calculate() → float

Calculate the flatness ratio of the profile.

plot(*axis: Axes*) → None

Plot the points of largest flatness difference as well as the search bounding box.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.metrics.profile.**PDD**(*depth_mm: float*, *window_mm: float = 10*, *poly_order: int = 2*, *normalize_to: Literal['fit', 'max'] = 'fit'*, *dmax_window_mm: float = 20*, *dmax_poly_order: int = 5*, *color: str | None = None*, *linestyle: str | None = '-.'*)

Bases: *Dmax*

The PDD at a given depth.

This will fit a polynomial to the profile in a window around the depth of interest and calculate the y-value of the polynomial at the depth of interest. This is the un-normalized value. We then have to normalize to the Dmax. The original PDD is then set as PDD/Dmax to give a true percentage.

Parameters

depth_mm
The depth at which to calculate the PDD.

window_mm
The width of the window to use for the fit. The window will be centered around the depth of interest.

poly_order
The order of the polynomial to use for the fit. See UnivariateSpline for more information. Generally, an order between 1 and 2 is recommended.

normalize_to
The value to normalize the PDD to. Either "fit" or "max". If "fit", the Dmax is calculated using the default Dmax metric using the `dmax_window_mm` and `dmax_poly_order` parameters. If "max", the maximum value of the profile is used.

dmax_window_mm
The width of the window to use for the Dmax calculation. Only used if `normalize_to` is "fit".

dmax_poly_order
The order of the polynomial to use for the Dmax calculation. Only used if `normalize_to` is "fit".

color
The color of the PDD point.

linestyle
The linestyle of the fit line.

property name

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

calculate() → float

Calculate the PDD of the profile.

This fits a polynomial to the profile in a window around the depth of interest and returns the y-value of the polynomial at the depth of interest.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

plot(*axis: Axes*)

Plot the PDD point and polynomial fit.

class pylinac.metrics.profile.**Dmax**(*window_mm: float = 20*, *poly_order: int = 5*, *color: str | None = None*, *linestyle: str | None = '-.'*)

Bases: ProfileMetric

Find the Dmax of the profile. This is a special case of the PDD metric.

Parameters

window_mm
The width of the window to use for the fit. The window will be centered around the maximum value point, which is used as the initial guess for the fit.

poly_order
The order of the polynomial to use for the fit. See UnivariateSpline for more information. Generally, an order between 3 and 5 is recommended.

color
The color of the Dmax point.

linestyle
The linestyle of the fit line.

calculate() → float

Calculate the Dmax of the profile.

We find the maximum value of the profile and then fit a polynomial to the profile in a window around the maximum value. The Dmax is the x-value of the polynomial's maximum value.

plot(*axis: Axes*)

Plot the PDD point and polynomial fit.

inject_profile(*profile: ProfileBase*) → None

Inject the profile into the metric class. We can't do this at instantiation because we don't have the profile yet. We also don't want to force the user to have to save it manually as they might forget. Finally, we want to have it around for any method we might use.

class pylinac.core.profile.**CollapsedCircleProfile**(*center: Point*, *radius: float*, *image_array: ndarray*, *start_angle: int = 0*, *ccw: bool = True*, *sampling_ratio: float = 1.0*, *width_ratio: float = 0.1*, *num_profiles: int = 20*)

Bases: CircleProfile

A circular profile that samples a thick band around the nominal circle, rather than just a 1-pixel-wide profile to give a mean value.

Parameters

width_ratio
[float] The "thickness" of the band to sample. The ratio is relative to the radius. E.g. if the radius is 20 and the width_ratio is 0.2, the "thickness" will be 4 pixels.

num_profiles
[int] The number of profiles to sample in the band. Profiles are distributed evenly within the band.

See Also

`CircleProfile` : Further parameter info.

as_dict() → dict

Convert to dict. Useful for dataclasses/Result

bit_invert() → None

Invert the profile bit-wise.

convert_to_dtype(*dtype: type[dtype]*) → None

Convert the profile datatype to another datatype while retaining the values relative to the datatype min/max

property diameter: float

Get the diameter of the circle.

filter(*size: float = 0.05*, *kind: str = 'median'*) → None

Filter the profile.

Parameters

size
[float, int] Size of the median filter to apply. If a float, the size is the ratio of the length. Must be in the range 0-1. E.g. if size=0.1 for a 1000-element array, the filter will be 100 elements. If an int, the filter is the size passed.

kind
[{'median', 'gaussian'}] The kind of filter to apply. If gaussian, *size* is the sigma value.

find_fwxm_peaks(*threshold: float | int = 0.3*, *min_distance: float | int = 0.05*, *max_number: int = None*, *search_region: tuple[float, float] = (0.0, 1.0)*) → tuple[np.array, np.array]

Overloads Profile to also map the peak locations to the image.

find_peaks(*threshold: float | int = 0.3*, *min_distance: float | int = 0.05*, *max_number: int = None*, *search_region: tuple[float, float] = (0.0, 1.0)*) → tuple[np.array, np.array]

Overloads Profile to also map peak locations to the image.

find_valleys(*threshold: float | int = 0.3*, *min_distance: float | int = 0.05*, *max_number: int = None*, *search_region: tuple[float, float] = (0.0, 1.0)*) → tuple[np.array, np.array]

Overload Profile to also map valley locations to the image.

ground() → float

Ground the profile such that the lowest value is 0.

Returns

float
The minimum value that was used as the grounding value.

invert() → None
Invert the profile.

normalize(*norm_val: str | float | None = None*) → None
Normalize the profile to the given value.

Parameters

norm_val
[number or 'max' or None] If a number, normalize the array to that number. If None, normalizes to the maximum value.

plot(*ax: plt.Axes | None = None*) → None
Plot the profile.

Parameters

ax: plt.Axes
An axis to plot onto. Optional.

roll(*amount: int*) → None
Roll the profile and x and y coordinates.

stretch(*min: float = 0*, *max: float = 1*) → None
'Stretch' the profile to the min and max parameter values.

Parameters

min
[number] The new minimum of the values

max
[number] The new maximum value.

property x_locations: array
The x-locations of the profile values.

property y_locations: array
The x-locations of the profile values.

property size: float
The elemental size of the profile.

plot2axes(*axes: Axes | None = None*, *edgecolor: str = 'black'*, *fill: bool = False*, *plot_peaks: bool = True*) → None
Add 2 circles to the axes: one at the maximum and minimum radius of the ROI.

See Also

`plot2axes()` : Further parameter info.

6.25 Images & 2D Metrics

New in version 3.16.

Pylinac images can now have arbitrary metrics calculated on them, similar to profiles. This can be useful for calculating and finding values and regions of interest in images. The system is quite flexible and allows for any number of metrics to be calculated on an image. Furthermore, this allows for re-usability of metrics, as they can be applied to any image.

6.25.1 Use Cases

- Calculate the mean pixel value of an area of an image.
- Finding an object in the image.
- Calculating the distance between two objects in an image.

6.25.2 Tool Legend

Use Case	Constraint	Class
Find the location of a BB in the image	The BB size and location is known approximately	*SizedDiskLocator*
Find the ROI properties of a BB in the image	The BB size and location is known approximately	*SizedDiskRegion*
Find the location of *N* BBs in the image	The BB size is known approximately	*GlobalSizedDiskLocator*
Find the location of a square field in an image	The field size is known approximately	*GlobalSizedFieldLocator*
Find the locations of *N* square fields in an image	The field size is not known	*GlobalFieldLocator*
Find the location of a circular field in an image	The field size and location are known approximately	*SizedDiskLocator* (`invert=False`)
Find the ROI properties of a circular field in an image	The field size and location are known approximately	*SizedDiskRegion* (`invert=False`)

6.25.3 Basic Usage

To calculate metrics on an image, simply pass the metric(s) to the `compute` method of the image:

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
metric = img.compute(
    metrics=DiskLocator(
        expected_position=(100, 100),
```

(continues on next page)

(continued from previous page)

```
        search_window=(30, 30),
        radius=10,
        radius_tolerance=2,
    )
)
print(metric)
```

You may compute multiple metrics by passing a list of metrics:

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
metrics = img.compute(
    metrics=[
        # disk 1
        DiskLocator(
            expected_position=(100, 100),
            search_window=(30, 30),
            radius=10,
            radius_tolerance=2,
        ),
        # disk 2
        DiskLocator(
            expected_position=(200, 200),
            search_window=(30, 30),
            radius=10,
            radius_tolerance=2,
        ),
    ]
)
print(metrics)
```

Metrics might have something to plot on the image. If so, the `plot` method of the image will plot the metric(s) on the image:

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
metrics = img.compute(
    metrics=[
        # disk 1
        DiskLocator(
            expected_position=(100, 100),
            search_window=(30, 30),
            radius=10,
            radius_tolerance=2,
        ),
        # disk 2
        DiskLocator(
            expected_position=(200, 200),
```

(continues on next page)

(continued from previous page)

```
            search_window=(30, 30),
            radius=10,
            radius_tolerance=2,
        ),
    ]
)
img.plot()  # plots the image with the BB positions overlaid
```

6.25.4 Built-in Metrics

Sized Disk Locator

Note: The values provided below are in pixels. The following sections show how variants of how to use the metrics using physical units and relative to the center of the image.

Here's an example of using the *SizedDiskLocator*:

Listing 2: Search for a disk 100 pixels right and 100 pixels down from the top left of the image

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
img.compute(
    metrics=[
        DiskLocator(
            expected_position=(100, 100),
            search_window=(30, 30),
            radius=10,
            radius_tolerance=2,
        )
    ]
)
img.plot()
```

This will search for a disk (BB) in the image at the expected position and window size for a disk of a given radius and tolerance. If the disk is found, the location will be returned as a *Point* object. If the disk is not found, a `ValueError` will be raised.

Using physical units

While pixels are useful, it is sometimes easier to use physical units.

To perform the same Disk/BB location using mm instead of pixels:

Listing 3: Search for a disk 30mm right and 30mm down from the top left of the image

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
img.compute(
    metrics=[
        # these are all in mm
        DiskLocator.from_physical(
            expected_position_mm=(30, 30),
            search_window_mm=(10, 10),
            radius_mm=4,
            radius_tolerance_mm=2,
        )
    ]
)
img.plot()
```

Relative to center

We can also specify the expected position relative to the center of the image.

Important: We can do this using pixels OR physical units.

This will look for the disk/BB 30 pixels right and 30 pixels down from the center of the image:

Listing 4: Relative to center using pixels

```
from pylinac.core.image import DicomImage
from pylinac.metrics.image import DiskLocator, DiskRegion

img = DicomImage("my_image.dcm")
img.compute(
    metrics=[
        # these are all in pixels
        DiskLocator.from_center(
            expected_position=(30, 30),
            search_window=(10, 10),
            radius=4,
            radius_tolerance=2,
        )
    ]
)
img.plot()
```

This will look for the disk/BB 30mm right and 30mm down from the center of the image:

Listing 5: Relative to center using physical units

```
img.compute(
    metrics=[
        # these are all in mm
        DiskLocator.from_center_physical(
            expected_position_mm=(30, 30),
            search_window_mm=(10, 10),
            radius_mm=4,
            radius_tolerance_mm=2,
        )
    ]
)
img.plot()
```

Sized Disk Region

The *SizedDiskRegion* metric is the same as the *SizedDiskLocator*, but instead of returning the location, it returns a scikit-image regionprops object that is the region of the disk. This allows one to then calculate things like the weighted centroid, area, etc.

It also supports the same class methods as the *SizedDiskLocator* metric.

Global Sized Disk Locator

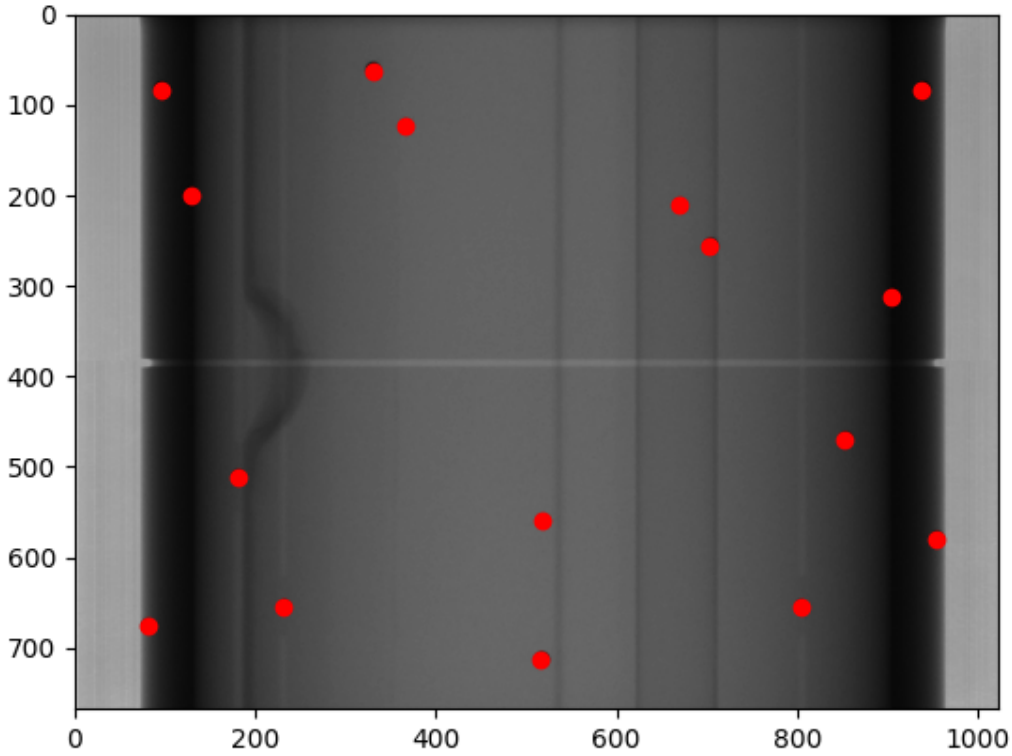
New in version 3.17.

The *GlobalSizedDiskLocator* metric is similar to the *SizedDiskLocator* metric except that it searches the entire image for disks/BB, not just a small window. This is useful for finding the BB in images where the BB is not in the expected location or unknown. This is also efficient for finding BBs in images, even if the locations are known.

For example, here is an example analysis of an MPC image:

```
from pylinac.core.image import XIM
from pylinac.metrics.image import GlobalDiskLocator

img = XIM("my_image.xim")
bbs = img.compute(
    metrics=GlobalDiskLocator(
        radius_mm=3.5,
        radius_tolerance_mm=1.5,
        min_number=10,
    )
)
img.plot()
```

This will result in an image like so:



Global Sized Field Locator

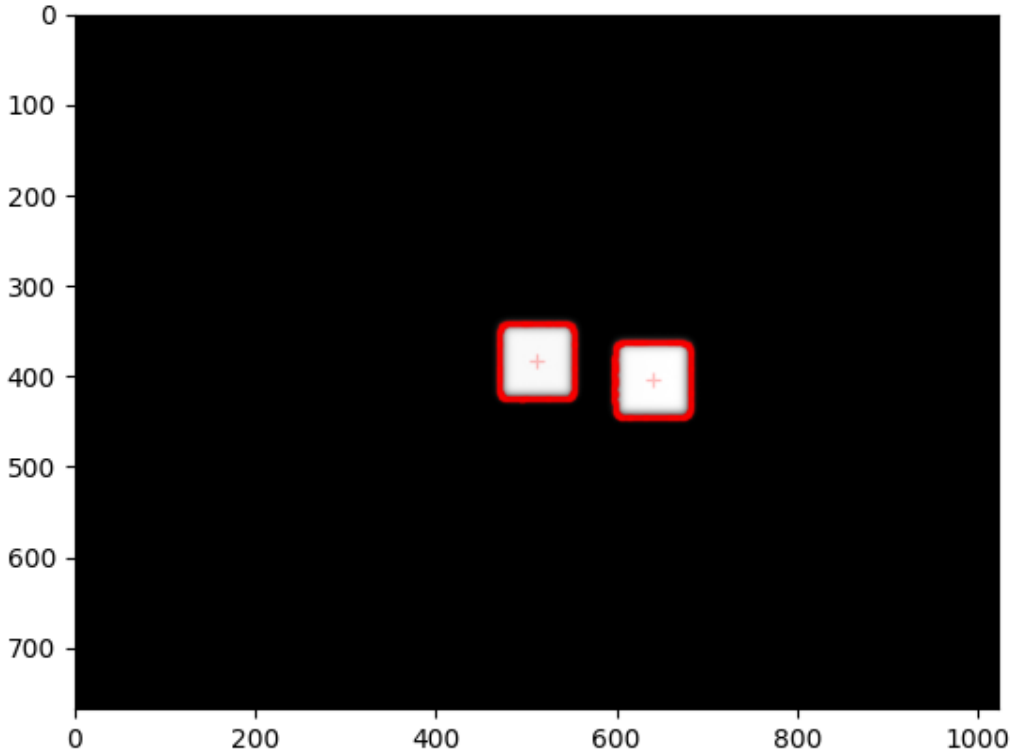
New in version 3.17.

The *GlobalSizedFieldLocator* metric is similar to the *GlobalSizedDiskLocator* metric. This is useful for finding one or more fields in images where the field is not in the expected location or unknown. This is also efficient when multiple fields are present in the image.

The locator will find the weighted center of the field(s) and return the location(s) as a *Point* objects. The boundary of the detected field(s) will be plotted on the image in addition to the center.

The locator will use pixels by default, but also has a `from_physical` class method to use physical units.

An example plot of finding multiple fields can be seen below:



For example:

Listing 6: Search for at least 2 fields of size 30x30 pixels with a tolerance of 4 pixels & plot

```
img = DicomImage("my_image.dcm")
img.compute(
    metrics=GlobalSizedFieldLocator(
        field_width_px=30, field_height_px=30, field_tolerance_px=4, max_number=2
    )
)
img.plot()  # this will plot the image with the fields overlaid
```

Constraints

- The field is expected to be mostly rectangular. I.e. not a cone.
- The field must not be touching an edge of the image. Such fields will be ignored.
- The field must be at least 10% of the maximum pixel value. This is to avoid finding artifacts. I.e. If the maximum pixel value is 10,000, then the pixels within the field must be at least 1,000 to be detected. Anything under 10% will be ignored.

Using physical units

To perform a similar field location using mm instead of pixels:

Listing 7: Search for at least 2 fields of size 30x30mm with a tolerance of 4mm

```
img = DicomImage("my_image.dcm")
img.compute(
    metrics=GlobalSizedFieldLocator.from_physical(
        field_width_mm=30, field_height_mm=30, field_tolerance_mm=4, max_number=2
    )
)
```

Usage tips

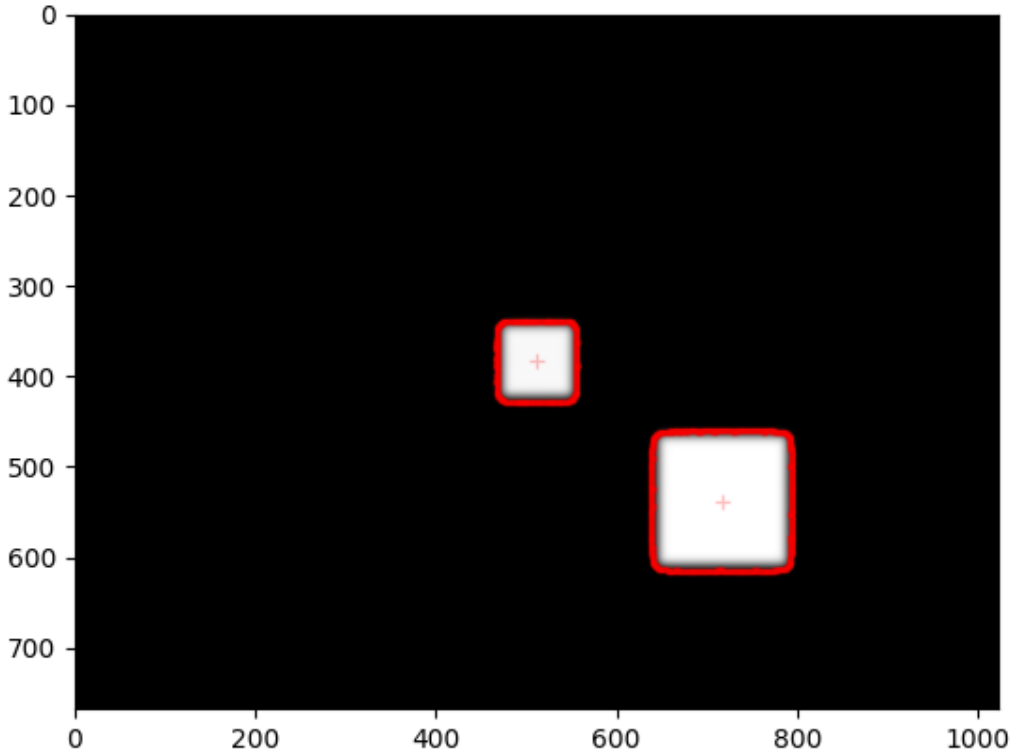
- Whenever possible, set the `max_number` parameter. This can **greatly** speed up the computation for several reasons. First, it will stop searching once the number of fields is found. Second, the thresholding algorithm will have a much better initial guess and also a better step size. This is because the approximate area of the field is known relative to the total image size.
- The `field_tolerance_<mm|px>` parameter can be relatively tight if the `max_number` parameter is set. Without a `max_number` parameter, you may have to increase the field tolerance to find all fields.

Global Field Locator

New in version 3.17.

The `GlobalFieldLocator` metric will find fields within an image, but does not require the field to be a specific size. It will find anything field-like in the image. The logic is similar to the `GlobalSizedFieldLocator` metric otherwise.

For example:



Listing 8: Search for any fields within the image, regardless of size

```
img = DicomImage("my_image.dcm")
img.compute(metrics=GlobalFieldLocator(max_number=2))
img.plot()  # this will plot the image with the fields overlaid
```

6.25.5 Writing Custom Plugins

The power of the plugin architecture is that you can write your own metrics and use them on any image as well as reuse them where needed.

To write a custom plugin, you must

- Inherit from the *MetricBase* class
- Specify a `name` attribute.
- Implement the `calculate` method.
- (Optional) Implement the `plot` method if you want the metric to plot on the image.

Warning: Do not modify the image in the `calculate` method as this will affect the image for other metrics and/or plotting.

For example, let's built a simple plugin that finds and plots an "X" at the center of the image:

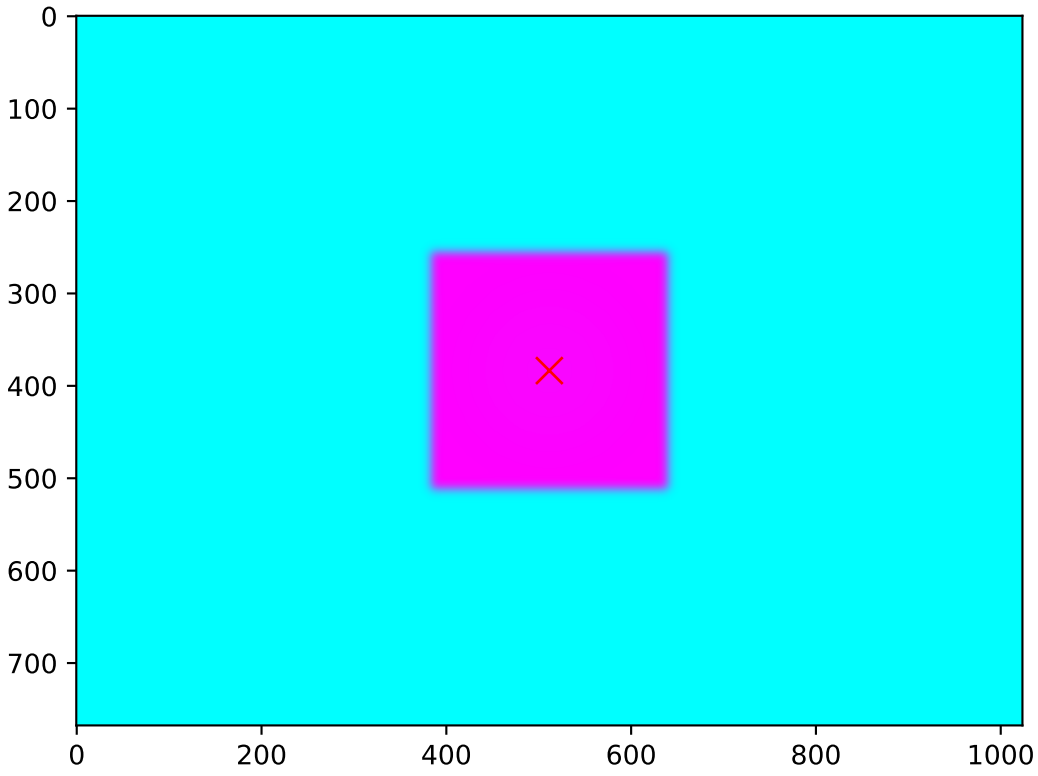
```
from pylinac.core.image_generator import AS1000Image, FilteredFieldLayer,␣
↪GaussianFilterLayer
from pylinac.core.image import DicomImage
from pylinac.metrics.image import MetricBase

class ImageCenterMetric(MetricBase):
    name = "Image Center"

    def calculate(self):
        return self.image.center

    def plot(self, axis: plt.Axes):
        axis.plot(self.image.center.x, self.image.center.y, 'rx', markersize=10)

# now we create an image to compute over
as1000 = AS1000Image(sid=1000)  # this will set the pixel size and shape automatically
as1000.add_layer(
    FilteredFieldLayer(field_size_mm=(100, 100))
)  # create a 100x100mm square field
as1000.add_layer(
    GaussianFilterLayer(sigma_mm=2)
)  # add an image-wide gaussian to simulate penumbra/scatter
ds = as1000.as_dicom()

# now we can compute the metric on the image
img = DicomImage.from_dataset(ds)
center = img.compute(metrics=ImageCenterMetric())
print(center)
img.plot()
```

6.25.6 Algorithm

The algorithms for the metrics are similar to each other. They all use a thresholding algorithm to find the object(s) of interest. For the example below, we will be using the first image from the Winston Lutz demo dataset:

1. The image is cropped if needed. This applies to metrics that have an expected position. I.e. non-`Global` variants of the metrics. This cropped section is called the "sample".

   Important: Image analyses using the *`WinstonLutz`* class are always cropped like this.


0
100
200
300
400
500
600
700
0
200
400
600
800
1000

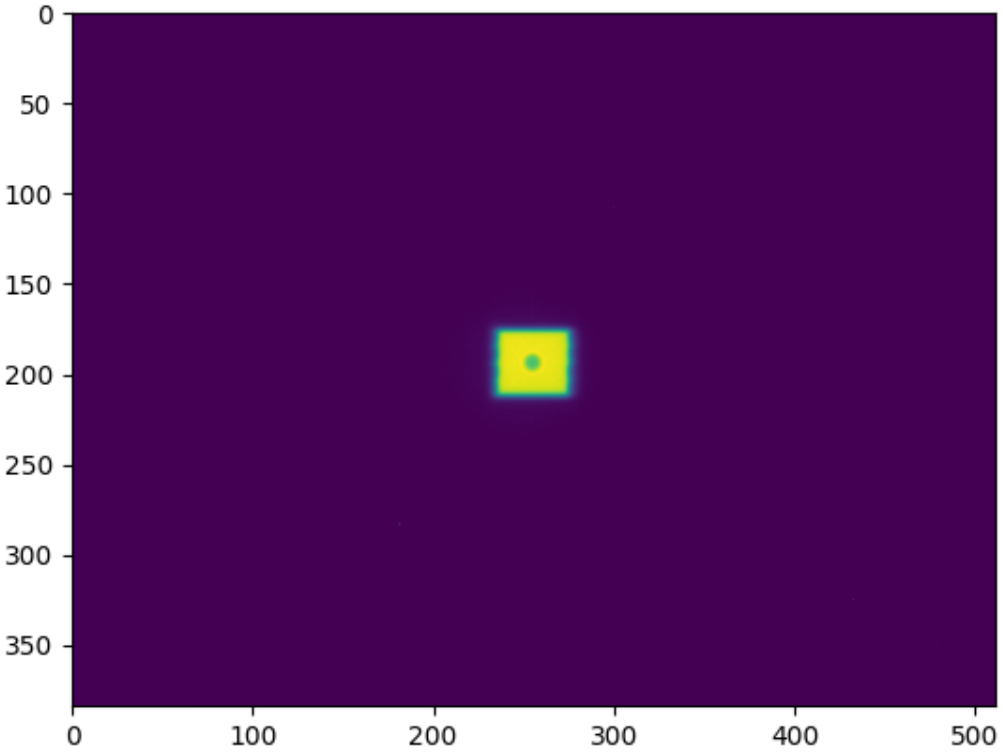
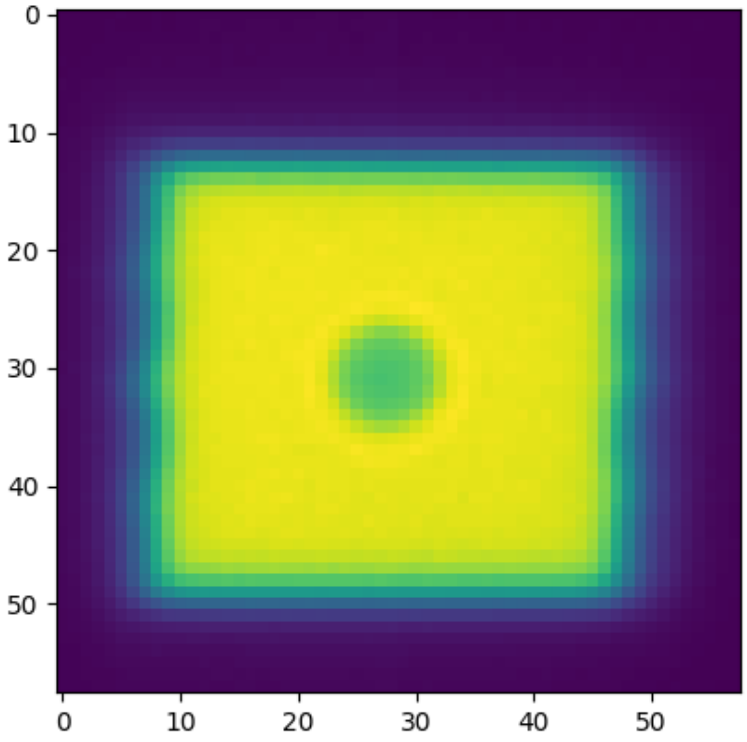
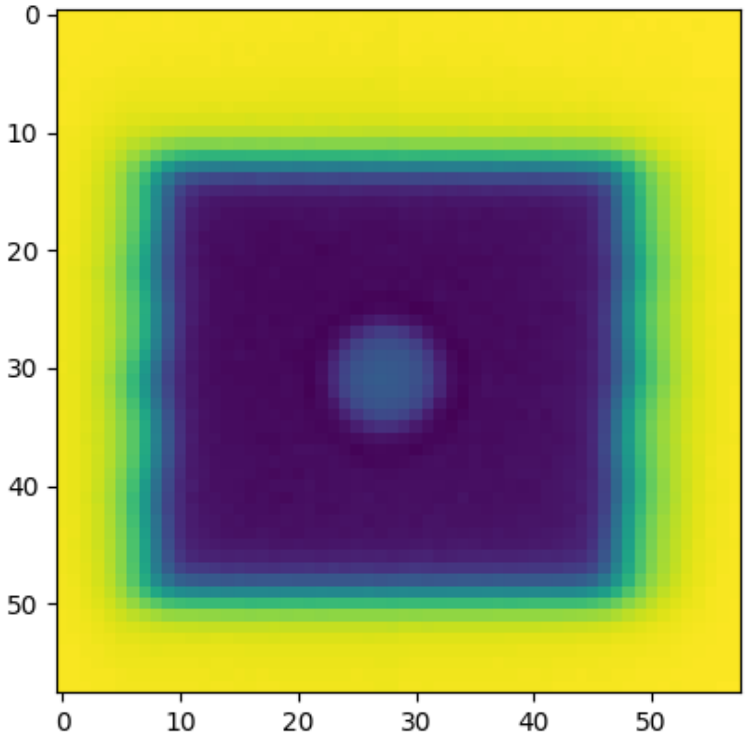


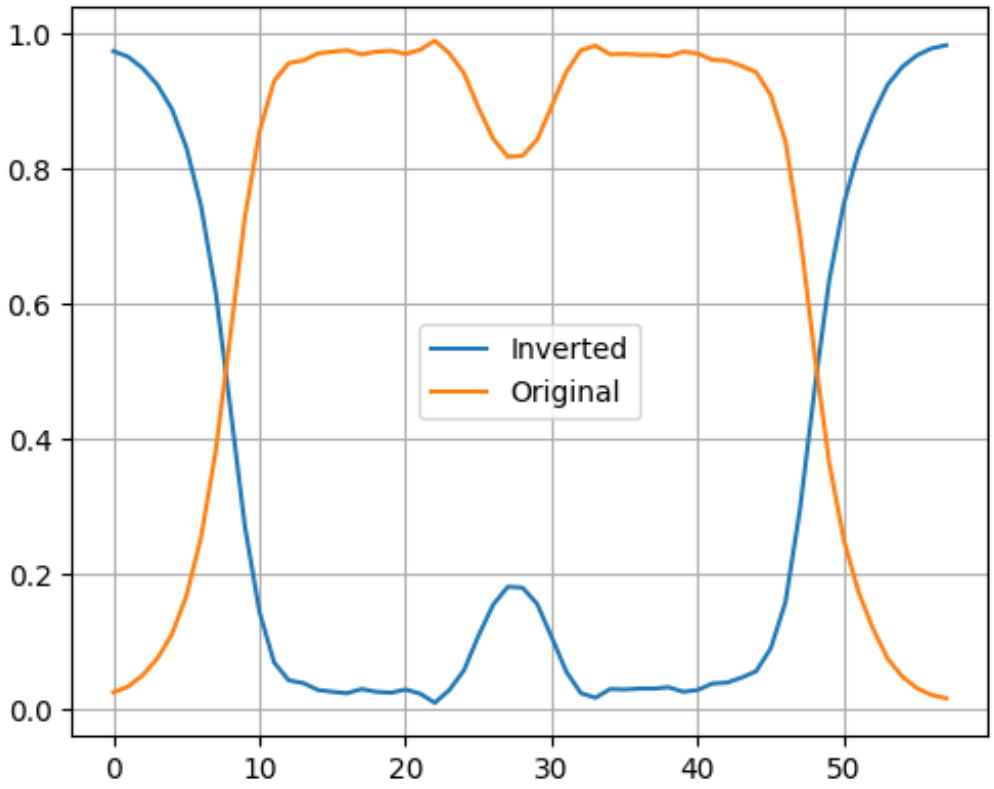
Fig. 8: Original image to be analyzed.



2. The sample might be inverted depending on the `invert` parameter. This applies to disk/BB metrics. The reason for inversion is that in the case of a BB within an open field, the pixel values within the BB ROI will be lower than the field surrounding it. Because pylinac uses weighted pixel values to find the center of the object, the pixel values need to be proportional to the "impact" of the object.



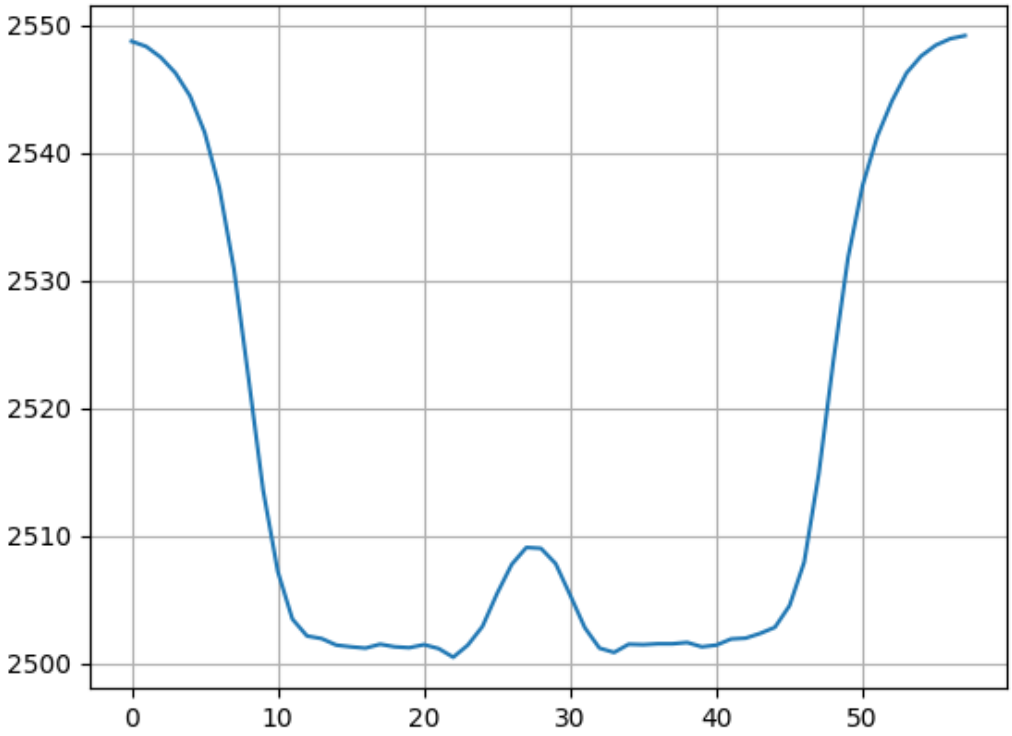
To make this clear, the profiles are plotted below with and without inversion. Because pylinac uses the weighted centroid of the pixels, we want the BB to "weigh" more at the center than the edges of the BB.



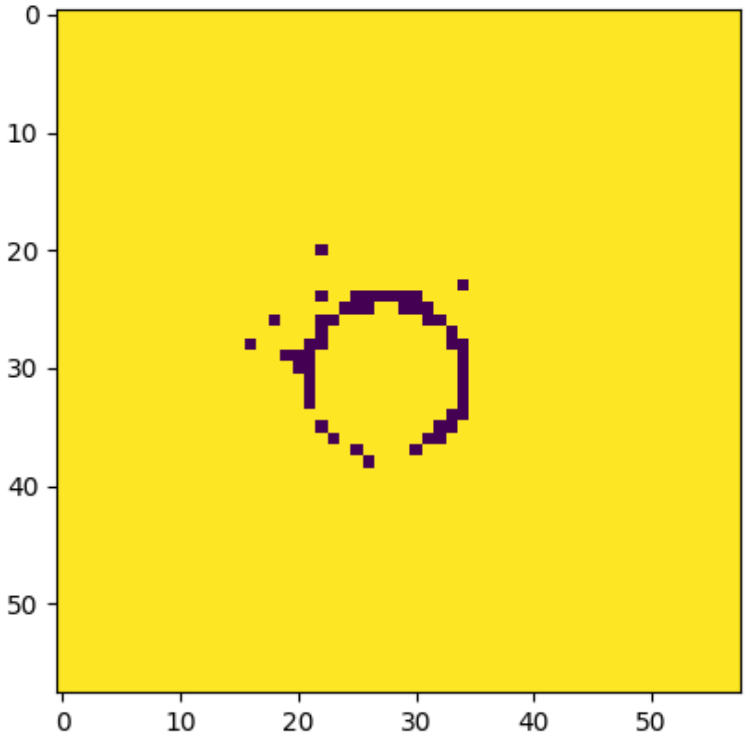
3. The sample is normalized to a range of 0-1. This is done because sometimes the attenuation of a BB can be relatively small compared to the surrounding field. Doing so ensures the central pixels' weighting is large and the BB edge pixel values is relatively small.

Note: For `Global` metrics, this often won't change the image very much since the entire image is searched.

The image below shows why normalization is helpful. When not normalized, the BB pixel values can be very similar to the region outside the BB. E.g. the value at index 20 (outside the BB in the field area) is ~2502 while the peak BB value at index 27 is ~2509. With such a small difference, the weighted centroid will be very similar to a simple centroid, which is not as accurate as it's sensitive to "edge" pixels of the blob.



4. The image is then converted to binary using a threshold. The threshold starts at 0.02. E.g. for this threshold value any pixel below 0.02 is set to 0 and everything above it is set to 1. The resulting binary image is analyzed to see if each "blob" passed all of the `detection_conditions`.

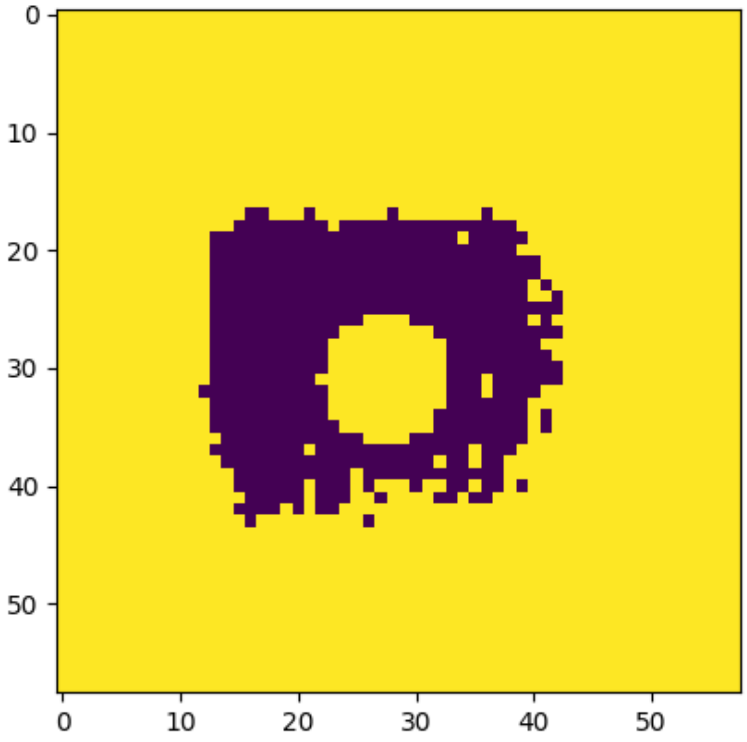


The detection conditions can be anything and generally measure blob properties such as area, eccentricity, etc. Each blob is a regionprops . The blob has a vast number of properties that are calculated and can be measured. E.g. the disk detection conditions are that the blob is 1) round, 2) has an area within the expected range given the BB size, and 3) has a circumference within the expected range given the BB size.

The image above isn't very helpful because the BB isn't really visible here.

If no blob passes all the detection conditions, the threshold is increased by 0.02 and the process is repeated.

Here is the image when the detection conditions pass:



Clearly, this threshold works well and the BB is identifiable.

5. If there is a `max_number` parameter, the thresholding algorithm will stop once that number is reached. This is because the thresholding algorithm can be slow, and once the expected number of objects is found there is no need to continue.

Important: Setting a `max_number` does not guarantee that **only** that number of objects will be found. It only guarantees that once that number is found, the thresholding algorithm will stop. This is because the thresholding algorithm could find multiple objects at a given threshold value.

If the `max_number` parameter is not set, the thresholding algorithm will continue until the threshold reaches 1.0.

6. If the `min_number` parameter is set and the thresholding algorithm stops (i.e. reaches 1.0), a `ValueError` is raised.

Note: The `min_number` and `max_number` parameters are only available for the `Global` variants of the metrics. They are both fixed to 1 for single object metrics.

---

7. If no blobs are found that match all the detection conditions after the threshold reaches 1.0, a `ValueError` is raised.
8. For each blob that is detected, the boundary of the binary threshold of that blob is tracked. This is solely used for plotting.
9. At each pass of the threshold, if more objects are detected, they are compared and potentially de-duplicated. This

is because if, e.g., we are looking for two objects and one is found at a given threshold, it will likely continue to be found again and again at higher thresholds while we are searching for the second object.

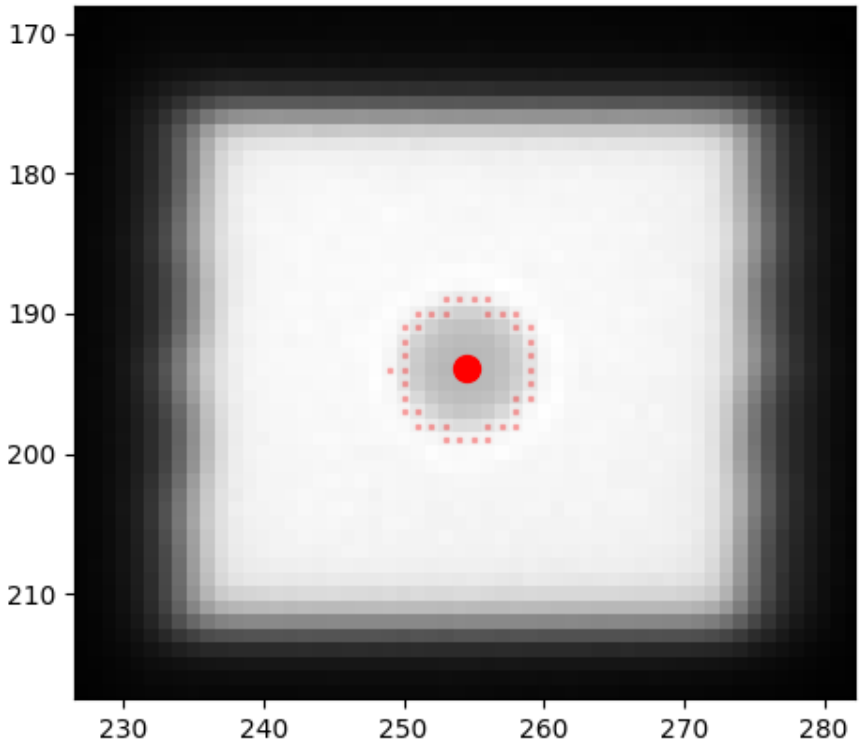
This comparison is done by comparing the centroids of the blobs. If the centroids are within the `min_separation_mm`, the first blob is retained and the subsequent blob is discarded. If the centroid is sufficiently far away, the blob is retained.

Note that it's also possible the original blob is no longer seen at higher thresholds. E.g. multiple small fields each delivering different MUs. In all cases, regions/points are always kept even if it is no longer detected at higher thresholds. I.e. the regions/points are add-only.

10. For `Locator` metrics, the weighted centroid of the blob is returned as a *`Point`* object. For `Region` metrics, the scikit-image regionprops object is returned.

Important: I can't stress this enough: **the weighted centroid is not the same as the simple centroid**. The weighted centroid is the centroid of the blob *accounting for the pixel value itself*. This will not be nearly as sensitive to "edge" pixels being included as a simple centroid. The simple centroid is the average of the pixel locations *not accounting for the pixel value*. You will find some other commercial software uses only the simple centroid, which is not as accurate as it's far more sensitive to edge pixels being included in the threshold.

Here is the plot of the final image with the BB location and threshold boundary overlaid:



Note: Note the single pixel on the left side boundary of the BB. A different threshold may or may not have included that pixel. The benefit of the weighted centroid is that the normalized, inverted sample value at that pixel is ~0 whereas the pixels at the center of the BB are ~1. The weighted centroid thus is hardly affected by that pixel, so it's robust to a

change in threshold.

6.25.7 API

class pylinac.metrics.image.**MetricBase**

Bases: ABC

Base class for any 2D metric. This class is abstract and should not be instantiated.

The subclass should implement the `calculate` method and the `name` attribute.

As a best practice, the `image_compatibility` attribute should be set to a list of image classes that the metric is compatible with. Image types that are not in the list will raise an error. This allows compatibility to be explicit. However, by default this is None and no compatibility checking is done.

inject_image(*image:* BaseImage)

Inject the image into the metric.

context_calculate() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

abstract calculate() → Any

Calculate the metric. Can return anything

plot(*axis: Axes*, ***kwargs*) → None

Plot the metric

additional_plots() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

class pylinac.metrics.image.**SizedDiskLocator**(*expected_position: Point | tuple[float, float], search_window: tuple[float, float], radius: float, radius_tolerance: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name: str = 'Disk Region'*)

Bases: *SizedDiskRegion*

Calculates the weighted centroid of a disk/BB as a Point in an image where the disk is near an expected position and size.

Finds BBs globally within an image.

Parameters

radius_mm
[float] The radius of the BB in mm.

radius_tolerance_mm
[float] The tolerance of the BB radius in mm.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean. The functions should be used to determine whether the regionprops object is a BB.

min_number
[int] The minimum number of BBs to find. If not found, an error is raised.

max_number
[int, None] The maximum number of BBs to find. If None, no maximum is set.

min_separation_mm
[float] The minimum distance between BBs in mm. If BBs are found that are closer than this, they are deduplicated.

name
[str] The name of the metric.

`calculate`() → *Point*

Get the weighted centroid of the region prop of the BB.

`plot`(*axis: Axes*, *show_boundaries: bool = True*, *color: str = 'red'*, *markersize: float = 3*, *alpha: float = 0.25*) → None

Plot the BB center

`additional_plots`() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

`context_calculate`() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

`classmethod from_center`(*expected_position: Point | tuple[float, float]*, *search_window: tuple[float, float]*, *radius: float*, *radius_tolerance: float*, *detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>)*, *invert: bool = True*, *name='Disk Region'*)

Create a DiskRegion from a center point.

`classmethod from_center_physical`(*expected_position_mm: Point | tuple[float, float]*, *search_window_mm: tuple[float, float]*, *radius_mm: float*, *radius_tolerance_mm: float = 0.25*, *detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>)*, *invert: bool = True*, *name='Disk Region'*)

Create a DiskRegion using physical dimensions from the center point.

classmethod from_physical(*expected_position_mm: Point | tuple[float, float], search_window_mm: tuple[float, float], radius_mm: float, radius_tolerance_mm: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name='Disk Region'*)

Create a DiskRegion using physical dimensions.

inject_image(*image:* BaseImage)

Inject the image into the metric.

class pylinac.metrics.image.**SizedDiskRegion**(*expected_position: Point | tuple[float, float], search_window: tuple[float, float], radius: float, radius_tolerance: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name: str = 'Disk Region'*)

Bases: *GlobalSizedDiskLocator*

A metric to find a disk/BB in an image where the BB is near an expected position and size. This will calculate the scikit-image regionprops of the BB.

Finds BBs globally within an image.

Parameters

radius_mm
[float] The radius of the BB in mm.

radius_tolerance_mm
[float] The tolerance of the BB radius in mm.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean. The functions should be used to determine whether the regionprops object is a BB.

min_number
[int] The minimum number of BBs to find. If not found, an error is raised.

max_number
[int, None] The maximum number of BBs to find. If None, no maximum is set.

min_separation_mm
[float] The minimum distance between BBs in mm. If BBs are found that are closer than this, they are deduplicated.

name
[str] The name of the metric.

classmethod from_physical(*expected_position_mm: Point | tuple[float, float], search_window_mm: tuple[float, float], radius_mm: float, radius_tolerance_mm: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name='Disk Region'*)

Create a DiskRegion using physical dimensions.

classmethod from_center(*expected_position: Point | tuple[float, float], search_window: tuple[float, float], radius: float, radius_tolerance: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name='Disk Region'*)

Create a DiskRegion from a center point.

classmethod from_center_physical(*expected_position_mm: Point | tuple[float, float], search_window_mm: tuple[float, float], radius_mm: float, radius_tolerance_mm: float = 0.25, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_right_size_bb>, <function is_round>, <function is_right_circumference>, <function is_symmetric>, <function is_solid>), invert: bool = True, name='Disk Region'*)

Create a DiskRegion using physical dimensions from the center point.

calculate() → RegionProperties

Find the scikit-image regiongprops of the BB.

This will apply a high-pass filter to the image iteratively. The filter starts at a very low percentile and increases until a region is found that meets the detection conditions.

plot(*axis: Axes*, *show_boundaries: bool = True*, *color: str = 'red'*, *markersize: float = 3*, *alpha: float = 0.25*) → None

Plot the BB center

additional_plots() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

context_calculate() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

inject_image(*image:* BaseImage)

Inject the image into the metric.

class pylinac.metrics.image.**GlobalSizedDiskLocator**(*radius_mm: float, radius_tolerance_mm: float, detection_conditions: list[Callable[[RegionProperties, ...], bool]] = (<function is_round>, <function is_right_size_bb>, <function is_right_circumference>), min_number: int = 1, max_number: int | None = None, min_separation_mm: float = 5, name='Global Disk Locator'*)

Bases: *MetricBase*

Finds BBs globally within an image.

Parameters

radius_mm
[float] The radius of the BB in mm.

radius_tolerance_mm
[float] The tolerance of the BB radius in mm.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean. The functions should be used to determine whether the regionprops object is a BB.

min_number
[int] The minimum number of BBs to find. If not found, an error is raised.

max_number
[int, None] The maximum number of BBs to find. If None, no maximum is set.

min_separation_mm
[float] The minimum distance between BBs in mm. If BBs are found that are closer than this, they are deduplicated.

name
[str] The name of the metric.

`calculate`() → list[*Point*]

Find up to N BBs/disks in the image. This will look for BBs at every percentile range. Multiple BBs may be found at different threshold levels.

`plot`(*axis: Axes*, *show_boundaries: bool = True*, *color: str = 'red'*, *markersize: float = 3*, *alpha: float = 0.25*) → None

Plot the BB centers

`additional_plots`() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

`context_calculate`() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

`inject_image`(*image:* BaseImage)

Inject the image into the metric.

class pylinac.metrics.image.**GlobalSizedFieldLocator**(*field_width_px: float*, *field_height_px: float*, *field_tolerance_px: float*, *min_number: int = 1*, *max_number: int | None = None*, *name: str = 'Field Finder'*, *detection_conditions: list[callable] = (<function is_right_square_perimeter>, <function is_right_area_square>)*)

Bases: *MetricBase*

Finds fields globally within an image.

Parameters

field_width_px
[float] The width of the field in px.

field_height_px
[float] The height of the field in px.

field_tolerance_px
[float] The tolerance of the field size in px.

min_number
[int] The minimum number of fields to find. If not found, an error is raised.

max_number
[int, None] The maximum number of fields to find. If None, no maximum is set.

name
[str] The name of the metric.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean.

`classmethod from_physical`(*field_width_mm: float*, *field_height_mm: float*, *field_tolerance_mm: float*, *min_number: int = 1*, *max_number: int | None = None*, *name: str = 'Field Finder'*, *detection_conditions: list[callable] = (<function is_right_square_perimeter>, <function is_right_area_square>)*)

Construct an instance using physical dimensions.

Parameters

field_width_mm
[float] The width of the field in mm.

field_height_mm
[float] The height of the field in mm.

field_tolerance_mm
[float] The tolerance of the field size in mm.

min_number
[int] The minimum number of fields to find. If not found, an error is raised.

max_number
[int, None] The maximum number of fields to find. If None, no maximum is set.

name
[str] The name of the metric.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean.

`calculate`() → list[*Point*]

Find up to N fields in the image. This will look for fields at every percentile range. Multiple fields may be found at different threshold levels.

`plot`(*axis: Axes*, *show_boundaries: bool = True*, *color: str = 'red'*, *markersize: float = 3*, *alpha: float = 0.25*) → None

Plot the BB centers and boundary of detection.

additional_plots() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

context_calculate() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

inject_image(*image:* BaseImage)

Inject the image into the metric.

class pylinac.metrics.image.**GlobalFieldLocator**(*min_number: int = 1*, *max_number: int | None = None*, *name: str = 'Field Finder'*, *detection_conditions: list[callable] = (<function is_right_square_perimeter>*, *<function is_right_area_square>)*)

Bases: *GlobalSizedFieldLocator*

Finds fields globally within an image, irrespective of size.

additional_plots() → list[figure]

Plot additional information on a separate figure as needed.

This should NOT show the figure. The figure will be shown via the `metric_plots` method. Calling show here would block other metrics from plotting their own separate metrics.

calculate() → list[*Point*]

Find up to N fields in the image. This will look for fields at every percentile range. Multiple fields may be found at different threshold levels.

context_calculate() → Any

Calculate the metric. This also checks the image hash to attempt to ensure no changes were made.

inject_image(*image:* BaseImage)

Inject the image into the metric.

plot(*axis: Axes*, *show_boundaries: bool = True*, *color: str = 'red'*, *markersize: float = 3*, *alpha: float = 0.25*) → None

Plot the BB centers and boundary of detection.

classmethod from_physical(**args*, ***kwargs*)

Construct an instance using physical dimensions.

Parameters

field_width_mm
[float] The width of the field in mm.

field_height_mm
[float] The height of the field in mm.

field_tolerance_mm
[float] The tolerance of the field size in mm.

min_number
[int] The minimum number of fields to find. If not found, an error is raised.

max_number
[int, None] The maximum number of fields to find. If None, no maximum is set.

name
[str] The name of the metric.

detection_conditions
[list[callable]] A list of functions that take a regionprops object and return a boolean.

6.26 Troubleshooting

6.26.1 General

Things always go wrong in real life. If you tried analyzing your data in pylinac and it threw an error, you can try a few simple things to fix it.

- First, **See if the demo works** - If not, pylinac may not have installed correctly or you may not have a dependency or the minimum version of a dependency.
- Second, **Check the error** - If it's an error that makes sense, maybe you just forgot something; e.g. analyzing an image before it's loaded will raise an error. Asking for a gamma result of a fluence before calculating the fluence will raise an error. Such things are easy to fix.
- Third, **Check the Troubleshooting section of the specific module** - Each module may fail in different ways, and also have different methods of resolution.
- And if none of those work, **Post a question on the forum** - You may have found a bug and it needs fixing!
  - Forum

6.26.2 Loading TIFF Files

Loading TIFF files can be tricky since there are many variations of the TIFF image format. Pillow is the package for image I/O in Python and is what pylinac uses. But sometimes even Pillow has trouble. If you've tried loading a TIFF file and it just doesn't seem to be working you can try two things:

- Install `pillow` with `conda`; when installing this way more libraries are installed (vs. `pip`) that pillow can leverage.
- Resave the image with another program. While I can't tell you exactly what will work, one solution that's worked for me is using GIMP. It's free; just open up your TIFF files and then export them back to TIFF. It may not seem like that should change anything, but my anecdotal evidence is that every TIFF image that didn't work that I reconverted using GIMP allowed me to read it in, especially when combined with the above strategy.

6.27 Contributing

There are several ways you can contribute to the pylinac project no matter your skill level. Read on for more info.

6.27.1 Submitting bugs

The easiest way to contribute is to report bugs. Submit bugs via a Github issue here.

6.27.2 Submitting files

Another easy way to improve pylinac is to submit QA files for the testing repository. Files are treated anonymously and are added to the test suite so that the package will become more robust. You can submit files here.

6.27.3 Suggesting ideas

Ideas are always welcome (though they might not get implemented). You can submit new ideas here.

6.27.4 Commit changes

Now you're serious about contributing. Awesome! Pylinac has mostly had one maintainer but we are looking for newcomers to contribute. There are a few things to know before contributing:

- Make an issue first, whether it be for a bug fix or feature request. This helps track the progress and put it on the roadmap appropriately.
- See if there are any other tools out there that can solve the problem. We don't want to write code just to write code. Pylinac should solve a problem that no one else has solved, do it better than existing solutions, or do so openly (vs closed). If another library already does this, the justification takes more work.
- Evaluate the work involved. This includes reviewing any existing modules or 3rd party tools that can help solve the problem.
- At this point, it looks like you're going to write some code. Make sure you have a *development environment* set up.
- Propose the framework. This includes making the files and boilerplate for the new module/functions. This will allow others to evaluate and make suggestions to the framework before the work actually starts.
- Once the framework is agreed upon the code can start flowing. Get to it!
- If you're unsure, just ask.

6.27.5 Setting up a development environment

If you want to contribute code, you'll need to set up a development environment. This is easy to do with the following steps. If you're new to git, see the git guide. Be sure to ask for help if you need it!

- Fork the pylinac repository.
- Clone your forked repository to your local machine: `git clone` followed by the URL of your forked repository. Some IDEs also have a GUI for this. See also *Get a Distribution Stack*.
- Create a virtual environment. This is optional but highly recommended. See the Python guide to set up a new venv.
- Install the requirements and developer requirements: `pip install -r requirements.txt -r requirements-dev.txt`
- Create a new branch for your work: `git checkout -b my_new_branch`
- Make your changes

- Write tests for your changes. Most test modules in pylinac have a 1:1 correlation with the library modules and thus the expected location should be straightforward.
- Run the tests locally: `pytest`
- If the tests pass, commit your changes: `git commit -m "my new feature!"`
- Push your changes to your forked repository: `git push origin my_new_branch`
- Make a new pull request to the main pylinac repository.

6.28 Changelog

6.28.1 v 3.18.0

Picket Fence

- The `from_multiple_images` method now no longer uses the demo image as a placeholder. This was causing an error when using this method within RadMachine as it was trying to load the demo image.
- A new method is available for picket fence instances: `picket_width_stat`. This will return a statistic for a given picket. This is useful for determining the consistency of the MLCs.
- A new item is available in `results_data`: `picket_widths`. This metric will provide the max, min, median, and mean of the picket widths for all MLC pairs across a picket. This is another way to test MLC consistency.

CT

- CatPhan, Quart, Cheese, and ACR phantom analyses now have a new parameter option: `memory_efficient_mode`. This mode will use dramatically less memory than the default implementation. This is useful for large datasets or limited resources on the machine running the process. This does come at a ~25-80% speed penalty depending on the size of the dataset. Larger datasets will have a larger penalty.
- In the `results` method, the CTP528 (spatial resolution) and CTP486 (uniformity) sections have been swapped. This is so that the resulting PDF text and images on each page matches. Previously, the PDF text and images for these two modules were switched.

Winston-Lutz

- The `results()` method of the `WinstonLutz` class will now also report the mean distance from the BB to the CAX in mm.
- The Winston-Lutz algorithm now uses the new *`SizedDiskLocator`* internal class (see below). This was introduced in pylinac 3.16. The algorithm is very similar to the existing WL algorithm.
- A new parameter has been added to `analyze()`: `bb_tolerance_mm`. This gives an acceptable window for finding a BB. E.g. if the BB size is 2mm, the tolerance can be set to 1mm. Alternatively, if the BB is very large, the tolerance can be widened. This was done since very small and very large BBs were sometimes tripping up because of the hardcoded 2mm tolerance. The default tolerance is still 2mm.

  Important: If you use WL with very small BBs (<3mm), we recommend you set the tolerance to 1mm.

- The BB boundary is now plotted. See the "Metrics" section.

- Detection conditions for the WL algorithm can now be set via the `detection_conditions` parameter for `WinstonLutz2D` and set as a class attribute for `WinstonLutz`.

Important: As always, pylinac uses the **weighted** centroid of the detected pixels. If the boundary seems to include an extraneous pixel, it should minimally affect the BB location.

Metrics

- There is a new `metrics` module in pylinac. Existing metrics have been moved into this module.

  E.g. instead of `from pylinac.core.metrics import SizedDiskLocator` you would now do `from pylinac.metrics.image import SizedDiskLocator`. Image-based metrics are now under `pylinac.metrics.image`. Profile-based metrics are now under `pylinac.metrics.profile`. Individual feature detection functions are now under `pylinac.metrics.features`.

  For backward compatibility (even though metrics are relatively new feature), the old import locations will still work but will raise a deprecation warning.

- The documentation for metrics has been updated considerably. See *Images & 2D Metrics*.
- The detection algorithm for disk/field metrics has been written out; see *Algorithm*.
- The `DiskLocator` class was renamed to `SizedDiskLocator`.
- The `DiskRegion` class was renamed to `SizedDiskRegion`.
- The `GlobalDiskLocator` class was renamed to `GlobalSizedDiskLocator`.
- The `SizedDiskLocator` class now plots the detected boundary of the disk/BB. Because the WL algorithm now uses this class, the WL plots now also include the detected BB boundary.
- A new metric class has been added: `GlobalFieldLocator`. This class will find a number of open fields within an image without having to know the field size beforehand. See *Global Sized Field Locator* for more.
- Previously, metrics would allow the image to be modified. The metric would copy the image temporarily. However, a memory bug would cause large numbers of images to use inordinate amounts of memory. Now, images cannot be permanently modified. A hash check will be run before and after the calculation to ensure the image array has not been modified and will raise an error if it has.
- Calling `plot` now allows to pass a `metric_kwargs` parameter. This allows the user to pass arguments to the underlying metric's `plot` method. This is useful for customizing the plot.
- A new metric `PDD` has been added. This will calculate the percent depth dose at a given depth using a polynomial fit.
- A new metric `Dmax` has been added. This will calculate the maximum dose using a polynomial fit.
- Profiles will now be sorted to have the x-values always be increasing.
- A bug was fixed when descending x-values for a profile were passed. This was causing the center index to be faulty.

6.28.2 v 3.17.0

Metrics

- Another metric is now available for 2D image analysis: `GlobalDiskLocator`. This metric will find a number of BBs/disks within an image. This is useful for finding BBs in an image without knowing where they might be. This is relatively efficient if there are multiple BBs in the image compared with using the `DiskLocator` class multiple times, even when the BB locations are known.
- The metric *`GlobalSizedFieldLocator`* is also available. This metric will find a number of open fields within an image. See *Global Sized Field Locator* for more.

Planar Imaging

- A new method is available for planar phantom analyses: *`percent_integral_uniformity()`*. This method will calculate the percent integral uniformity (PIU) over the low-contrast ROIs. This result will also be included in the `results_data` structure. This is not done for light/rad phantoms.
- If a phantom had a completely homogeneous array for an ROI, the `results_data` call would fail due to a division by 0 error. This has now been fixed such than an error is not raised. However, the resulting CNR and SNR will be a special case of `float('inf')`. This was encountered with a very low kVp analysis of the Doselab MC2 kV/MV phantom.

Picket Fence

- The Halcyon MLC configurations were incorrect and have now been fixed. Thanks to Dominic Rafferty for pointing this out. Previously, it was using a similar configuration as the TrueBeam out of lack of experience with the system. The new configuration was based on this paper.

Winston-Lutz

- Normal Winston-Lutz analyses (not multi-target/multi-field) can now plot a visualization of the BB position relative to the determined isocenter. After analyzing a WL set, call `plot_location()`. See *Visualizing the Isocenter-to-BB*.

CT

- A new class *`CIRS062M`* is now available. This will analyze the CIRS electron density phantom.
- The base class for cheese phantoms (*`CheesePhantomBase`*) now has a default implementation for `results_data`. Previously, it did not and required the user to create one when extending the phantom analysis to a new type.
- The *`TomoCheese`* phantom's output from `results_data` has an additional key: `rois`. This is a dictionary of all the ROIs with the name of the ROI (usually the number) as the key. The data in the `rois` dict is the same information as in the `roi_<n>` elements. In retrospect, a simple dictionary is far more extensible when the number of ROIs vary. I.e. `results_data()['rois']['1']` is the same as `results_data()['roi_1']`. The `roi_<n>` keys were left for backwards compatibility.
- A new class `HypersightQuartDVT` has been added that will analyze the Hypersight variant of the Quart phantom, which includes an additional water ROI.

ACR

- The `z_position` property for DICOM stacks (used in CT and MRI) was using `SliceLocation` if the tag existed and `ImagePositionPatient[-1]` if it did not exist. The `SliceLocation` tag however is apparently relative. This caused problems for the ACR MRI module on properly-acquired datasets. The `ImagePositionPatient` tag is now the primary lookup key and `SliceLocation` is only used if the former tag is unavailable.

Starshot

- The `from_multiple_images` method no longer requires the demo image. The demo image was just a placeholder to set up initial values.

Profiles

The following applies to the `SingleProfile` classes:

- Passing *decreasing* x-values to `SingleProfile` would usually result in an error because the measured width would be negative. An error will now be raised if the x-values are decreasing.
- Profiles that had non-integer increments in the x-values were not returning the right field values. I.e. when calling `.field_data()['field values']` and non-integer x-values were passed at instantiation the values were not correct. Given the `SingleProfile` class is now frozen, it is recommended to not pass non-integer x-values and/or skip passing x-values to the profile.

The following applies to the `<FWXM|InflectionDerivative|Hill>Profile` classes:

- The same error of passing *decreasing* x-values as above was also detected in the new `<FWXM|InflectionDerivative|Hill>Profile` classes. Given these classes are the new standard, they have been fully fixed and can now handle decreasing x-values.
- Profiles that had non-integer increments in the x-values were not returning the right field values. I.e. when calling `.field_values()` and non-integer x-values were passed at instantiation the values were not correct. This has been fixed.
- The `x_at_x` method has been renamed to `x_at_x_idx`. A deprecation warning will be raised. The method will be removed in 3.18.
- The `y_at_x` and `x_at_y` and `x_at_x_idx` methods now all return a numpy array instead of a float.
- A new method has been added: `field_x_values`. This returns a numpy array of x-values that corresponds to the y-values that are returned when using `field_values`. This is useful for plotting the field values to the correct x-values.
- The `SymmetryPointDifferenceMetric` class' plot method now uses "x" for the markers instead of "^" and "v".

6.28.3 v 3.16.0

Planar Imaging

- `results_data` for planar imaging phantoms (Leeds, SNC kV/MV, Doselab MC2, etc) will now return a `low_contrast_rois` dict that contains relevant info for each low-contrast ROI.

Winston-Lutz

- The Winston Lutz module can now load CBCT datasets of a scanned BB. This is still experimental and may have bugs. Caution is warranted. See *CBCT Analysis*.

CBCT

- Passing expected HU values for ROIs is now much easier by passing a dictionary to the `.analyze()` method. See *Custom HU values*.

Profiles

- Profile analysis has been completely revamped. The existing `SingleProfile` class still exists and will not be deprecated immediately. It is frozen and will not receive updates.
- New profile classes were written that are more generalizable and extensible. These can be read about in the documentation below.
- The new profile classes also have a new plugin system for computing custom metrics. This allows for much more user-friendly, readable, and extensible code for both myself and users.
- A new documentation section has been added for profiles: *Profiles & 1D Metrics*. This section describes the various profile classes and how to use them.
- Internally, pylinac now uses these new profile classes. Existing calculations should be the same.
- Calculating custom profile metrics (such as symmetry or flatness) is now much easier using these new classes. The field analysis module will get a "v2" that will use these new classes and allow for these easy-to-write custom metrics.

Core

Image

- Similar to the new profile plugin architecture, 2D images also have a new plugin metric system. See the new documentation: *Images & 2D Metrics*.
- The `DicomImage` class has a new class method: *`from_dataset()`*. This allows one to create a Dicom image from a pydicom dataset directly.

Image Generator

- The `Simulator` class and its subclasses has a new method: *`as_dicom()`*. This method will perform the same action as `generate_dicom`, but instead of saving to file, will return the pydicom Dataset.

6.28.4 v 3.15.0

Winston-Lutz

- For the MultiTargetMultiField Winston Lutz analysis, non-zero couch angles are not allowed. However, the check for this was limited to 0-5 degrees. Couch values that were on the other side of 0 were not being included. Couch angles between 355-5 degrees are now allowed as originally intended.

Planar Imaging

- The Doselab RLf light/rad phantom has been added as an analysis options: *Doselab RLf*.
- The IsoAlign light/rad phantom has been added as an analysis options: *IsoAlign*.

CT

- The catphan detection was failing if the phantom jig was touching the phantom at the center of a module. This has been fixed.
- A rounding error was fixed where the extent check was failing because of floating point rounding differences. This was causing an error to be raised when the scan extent was just slightly smaller (or appeared to be smaller) than the configuration extent.

ACR

- The ACR MRI phantom analysis was sometimes failing because the slice thickness check was failing. This was caused by a slightly inappropriate use of the profile module, causing instability under certain conditions. The MRI analysis should be more stable. Quantitative results should be the same.

VMAT

- The standard deviation for each VMAT segment is now available as the `.stdev` property of the segment.

```
vmat = DRMLC(...)
vmat.analyze(...)
data = vmat.results_data()
print(data.segments[0].stdev)  # first segment stdev
```

Core

- When saving a DICOM image, the pixel values were not "unscaling" the raw pixel values. I.e. the scaled values were being saved back to the DICOM file. If the image was then read in again, the values would be scaled twice. This has been fixed and DICOM images can now, for the most part, go "round trip" without the raw pixel values changing. An example is below:

```
dcm_image = image.load("my_image.dcm")
dcm_image.array  # this is scaled by the DICOM tags
dcm_image.save(
    "my_output_image.dcm"
)  # the pixel values were written back *as rescaled*
```

(continues on next page)

(continued from previous page)

```
dcm_image2 = image.load("my_output_image.dcm")
dcm_image2.array  # this was scaling by the DICOM tags *again*
```

Warning: If the DICOM pixel values have been modified, such as concatenating images together, and the values are too high or too low for the original datatype (usually uint16), the values will be scaled to fit the datatype, with the maximum value being the max of the datatype. A warning will be raised when this occurs.

Most of the time these operations are relative and absolute values don't matter, but it's still something to be aware of.

6.28.5 v 3.14.0

Planar Imaging

- An Elekta variant of the Las Vegas phantom has been added: *ElektaLasVegas*.
- The SSD parameter of now defaults to "auto" (`.analyze(..., ssd="auto")`). Previously, it was set to 1000mm. If "auto", the phantom is first searched at 1000mm (for backwards compatibility). If the phantom isn't found, it then searches at 5cm above the SID value. The 5cm is to account for the physical shroud of most EPID panels. If the phantom isn't found at either of these locations an error is raised. In that case, the SSD should be provided manually, which was already the case previously.

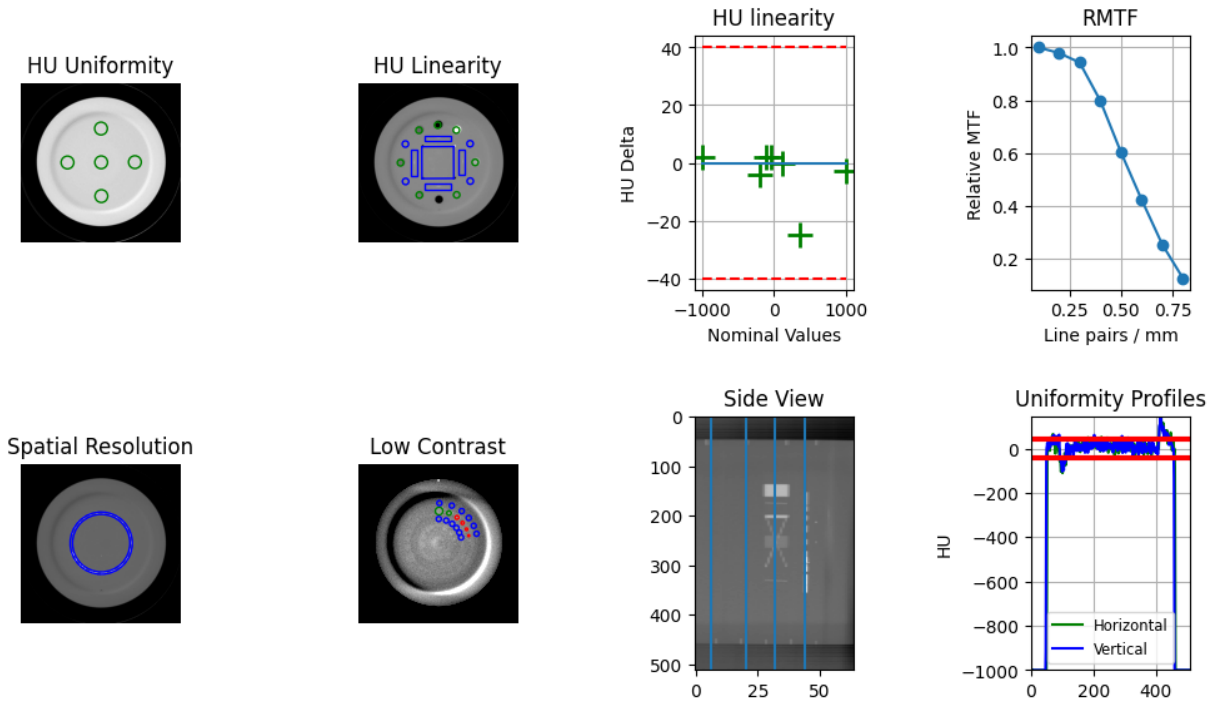
CT

- CBCT, ACR CT/MR, and Quart analyses will now plot a "side view" of the phantom with lines to show where the modules were sampled. This will help visualize if the module slice selection was appropriate.
- A new check for the scan extent vs the configuration extent is now in place. This will check that the physical extent of the scan is large enough to include all the listed modules. If it's not an error will be raised. This improves the error diagnosis when a scan did not include enough data.

Note: This applies to all CT-like algorithms including the ACR analyses.

ACR

- The ACR MRI algorithm now accounts for scans where slices do not abut. E.g. if the slice thickness is 5mm and the spacing between slices is 10mm.
- The ACR MRI high-resolution ROIs have been adjusted slightly to match the increasing test suite data, however, there are still some sets that do not perfectly align. We suggest following the *Customizing MR/CT Modules* section and adjusting the location as needed.
- The ACR MRI algorithm has a new parameter for `analyze`: `echo_number`. This lets the user pick an echo number if the acquisition was a dual echo scan. This is not required however. If the scan is dual-echo and no echo number is passed, the scan with the first echo number is selected. See the *Choosing an MR Echo*.
- The ACR MRI module classes can now be defined at the class-level, similar to the ACR CT. This was changed so that users can more easily change aspects of each module. See the *Customizing MR/CT Modules* section for more.



- The ACR MRI phantom *MRUniformityModuleOutput* had a typo. The property `ghost_rois` was actually spelled `ghose_rois`. Any code using this property should be updated to the correct spelling.
- The ACR MRI *results_data()* method will now return `ROIResult` instances instead of the raw `HUDiskROI` classes as before. This behavior already occurs for the catphan module and will thus make the results similar in structure.

Quart

- The Quart algorithm now measures the high-contrast resolution. It is accessible via the `high_contrast_resolution` method. It is given in the `results` and `results_data` methods as well.

```
from pylinac import QuartDVT

quart = QuartDVT(...)
quart.analyze()
high_res = quart.geometry_module.high_resolution_contrast()
# or
print(quart.results())
# or
high_res = quart.results_data().geometric_module.high_contrast_distance
```

Core

- The *DicomImage* class has two new properties available: `z_location` and `slice_spacing`. These both apply to CT/MR-like datasets.
- A new contrast algorithm, "Difference", has been added. This can be used similar to RMS, Weber, etc. The reason this might be preferred is so that the resulting CNR value is closer to the default algorithm. See *Contrast* for more.
- Contrast values are now case-insensitive. This applies only if you are passing a string for the contrast method.

```
from pylinac import CatPhan504
from pylinac.core.contrast import Contrast

ct = CatPhan504.from_demo_images()
# equivalent
ct.analyze(..., contrast_method="weber")
ct.analyze(..., contrast_method="Weber")
ct.analyze(..., contrast_method=Contrast.WEBER)
```

- Image classes (`DicomImage`, `ArrayImage`, `FileImage`) have a new method: *rotate()*. This is a wrapper for scikit-image that allows rotation of an arbitrary angle. Previously, only rotations of 90 degrees were allowed via the `rot90` method.
- The library `cached_property` was dropped as a requirement since it was introduced in Python 3.8
- The utility function `find_nearest_index` in the `acr` module was moved to `core.array_utils`.
- The utility functions `abs360` and `wrap360` were moved from `core.utilities` to `core.scale`.

6.28.6 v 3.13.0

Warning: As stated in the previous version, v3.13+ will not support Python 3.7. Python 3.8+ is required, matching the PSF's deprecation policy.

Planar Imaging

- The Leeds phantom has had its high-contrast ROIs adjusted to better fit the majority of phantoms encountered. Additionally, due to perceived differences in manufacturing, the high-contrast ROIs are now placed according to the center of the high-contrast block. The block is found after the phantom is found and the ROI configuration is adjusted about this center. We have noticed small differences between the block and the phantom center that are large enough to move the ROIs outside the line pairs. Even this however does not correctly place the ROIs all the time.

  Warning: This may affect your MTF values, but so far it does not significantly change it if the ROIs were already correctly on top of the high contrast pairs. Images where the ROIs were mis-aligned with the line pairs should now better match, so any change should be between noise and a healthy improvement.

  Here are two images comparing the old positions to the new ones for an image that was previously not working:

  Here is the demo image, where the ROIs were working before, showing that the new locations still work.

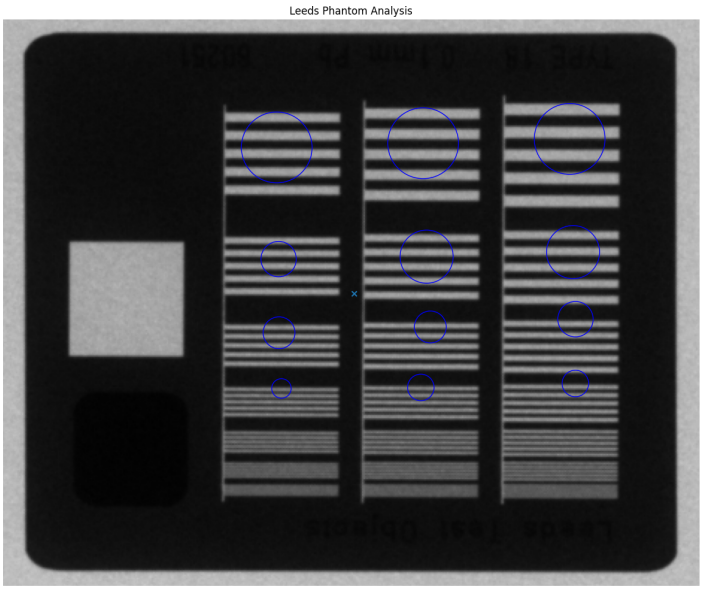


Fig. 9: Previous Leeds ROIs on a poorly-fitting image

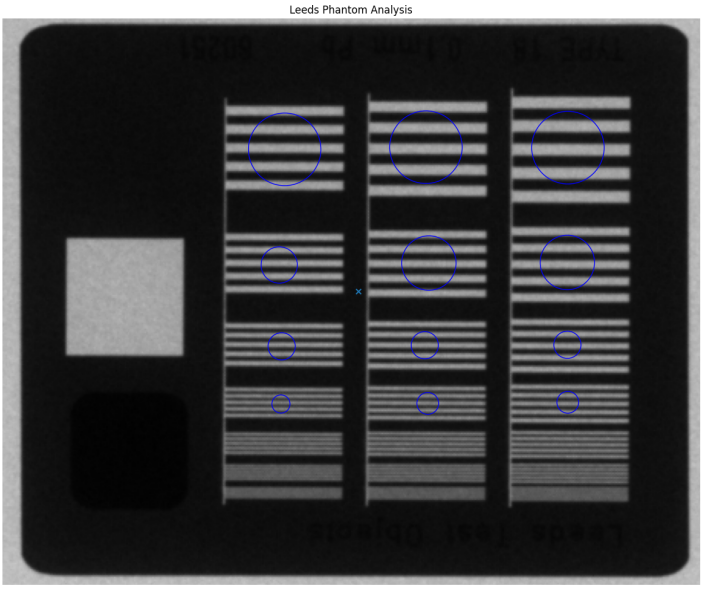


Fig. 10: New Leeds ROIs on the same image

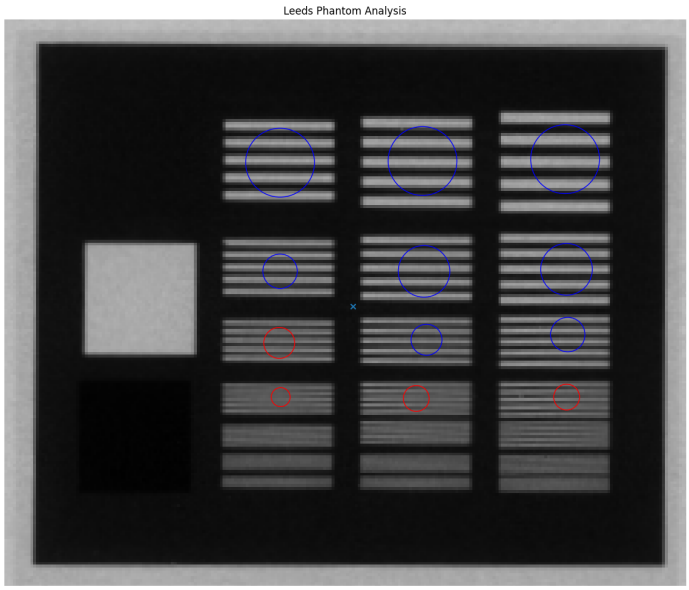


Fig. 11: Previous Leeds ROIs on the demo image

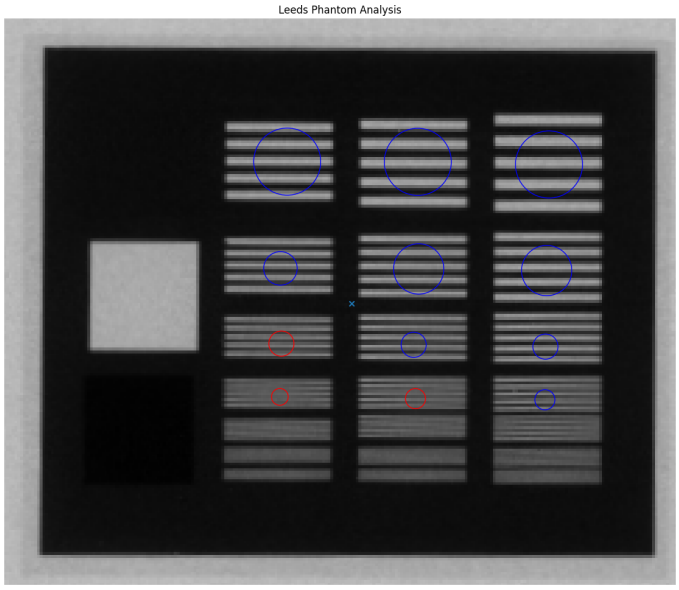


Fig. 12: New Leeds ROIs on the demo image

Note: At this point in time it's unclear where the variation is coming from. This is a best-fit solution to this variation. It's possible there was a revision along the line or the placement tolerances are simply not very tight. We have evidence of other quality issues such as off-center low-contrast ROIs as well. If you know how these differences have come to be let us know!

Finally, if you would like to keep the old ROI locations here is a gist with the old settings: https://gist.github.com/jrkerns/10b62aad7b38c210b9213761447f6155

- Related to above, the high-contrast ROIs have been reduced in size slightly so as not to spill out of the line pair area when there are small discrepancies of location. Testing did not change the MTF significantly from reducing the ROI size.

VMAT

- Three new parameters were added to the `__init__` call: `raw_pixels`. `ground`, and `check_inversion`. These were added to allow users to avoid applying DICOM pixel correction and analysis manipulations. The reason for this is to match the results from other programs such as Doselab. See the new section *Comparison to Doselab*.

Core

- The `DicomImage` class constructor has a new boolean parameter `raw_pixels`. This was implemented for the above VMAT feature, but can be applied to any image if desired. This will not apply any pixel correction tags, and simpy load the values as saved in the DICOM file.

6.28.7 v 3.12.1

Contrast

- The contrast logic was refactored in pylinac 3.12.0. Unfortunately, this used the "vanilla" definition of weber (see Weber). Pylinac versions 3.11 and prior used the absolute difference of the numerator. Using the signed difference caused issues for existing users and workflows. This was unintentional. For backwards compatibility, the definition has been restored to the previous behavior.

6.28.8 v 3.12.0

Warning: This is the last version of pylinac that will support python 3.7 as it will reach end-of-life in July 2023. pylinac v3.13+ will support python 3.8 until October 2024 when python 3.8 is deprecated, etc. You can see the end-of-life chart here.

General

- The *Contrast* class which contains all the contrast algorithms is no longer an enum. It is now a simple class. This should have no effect on the user unless doing something like `Contrast.MICHELSON.value`. No changes are required on the users's part for normal usage patterns.
- The :ref:<contrast> section has been enhanced to provide more details.
- A new core module `contrast` has been created. It contains all contrast-related items. Individual functions can now be called. See the above contrast doc section for examples.

- A new contrast option is available: Root-mean-square. It is available in the `Contrast` class like the existing options. E.g. `leeds.analyze(..., contrast_method=Contrast.RMS)`
- The *Image* section of the documentation has been enhanced with examples for using the core image behavior.
- The *LowContrastDiskROI* now provides properties for all contrast types, not just the selected one. I.e. `<roi>.weber`, `<roi>.rms`, `<roi>.michelson`, and `<roi>.ratio`. The existing `<roi>.contrast` still exists and will respect the passed contrast algorithm as before. This provides a way to compare other contrast algorithms without needing to re-analyze an image.
- There was a bug in the `equate_images` function where same-sized images were causing a zero division error. See here: https://github.com/jrkerns/pylinac/issues/446. Thanks to Luis for finding it!
- The `crop` method for images had a bug where passing `pixels=0` would cause the array to diminish to shape 0 along the axes it was cropping.

Winston-Lutz

- The smallest BB allowed for detection is now ~1mm. This was previously 2mm, but the Varian Exact cube's BB proved to be too small.

> **Warning:** Setting the BB size to a very small value increases the chance of inaccurately detecting the BB when the BB is encased in a block.

CatPhan

- A new analyse parameter has been added `thickness_slice_straddle`. This is to explicitly control the slice combination technique for the slice thickness measurement. The default behavior is backwards-compatible so no changes are needed. Read more here: *Slice Thickness*.
- The parameter `clear_borders` was not being propogated to all submodules for catphan analysis. This is now fixed. Thanks to Chris Williams for finding and fixing the issue here: https://github.com/jrkerns/pylinac/issues/448.

6.28.9 v 3.11.0

General/Core

- The docs now use the `furo` theme.
- A new function is available under the `image` module that converts a TIFF image to a simple DICOM format: `tiff_to_dicom()`.
- Saving a PDF with the default logo will now additionally try to load the logo from the demo file repository if the file is not available locally. This occurs when using pylinac as a Package in RadMachine. This will now allow users to publish PDFs within RadMachine from a custom pylinac package.
- The demo files and PDF references have been removed from the git repository to make shallow clones smaller (e.g. downloading the repo from Github). Demo files are still available publicly as they always have been. No user changes required.
- Type errors should no longer occur for older version of Python.

Cheese Phantoms

- The cheese module has been refactored to be more generalizable so that new cheese-like phantoms can be easily created. Documentation on doing this has been added *here*.
- The `hu` attribute of the `TomoCheese` class has been renamed to `module`. This doesn't affect typical use patterns.

Field Analysis

Warning: TL;DR: Symmetry will statistically go down and Flatness may rise slightly due to an off-by-one bug. For flat DICOM beams, this is insignificant.

A bug was fixed that caused the data considered to be the "field" to be off-by-one. The last element was not included. A visualization can be seen here: https://github.com/jrkerns/pylinac/issues/440. This caused BOTH symmetry and flatness to be affected when using *`FieldAnalysis`* and `DeviceFieldAnalysis` classes.

The value by which the symmetry and flatness will change depends a few factors. The largest factor is the resolution of the original image/dataset. For fields with high resolution, e.g. an AS1200 image, the effects will be smaller than for low-resolution datasets such as the Profiler. The gradient of the beam is also a large factor and FFF beams are the most affected. Interpolation does not have an effect.

To give an idea of when and how much the values will change, the change was performed on all the available data we have for open fields using DICOM and Profiler data and are presented in the table below. Approximately 400 datasets were evaluated.

For DICOM, only flat beams were available for analysis. For all analyses, the field ratio was 0.8, i.e. 80% field width.

Table 1: Symmetry & Flatness changes in % after the bug fix by data and beam type

	Horizontal Symmetry	Horizontal Flatness	Vertical Symmetry	Vertical Flatness
DICOM (Flat)	0	+0.02	+0.01	+0.01
Profiler (Overall)	-0.20	+0.11	-0.26	+0.08
Profiler (Flat)	-0.16	+0.04	-0.09	+0.01
Profiler (FFF)	-0.80	+0.33	-1.26	+0.22
Profiler (Electron)	-0.08	+0.30	-0.52	+0.26

Positive values indicate the value went up, while negative values indicate the average went down.

The data shows that for DICOM data of flat beams, the effect was negligible. This makes sense since an off-by-one error for a field several hundred pixels wide will hardly register. It is the low-resolution datasets that show a difference. The values make general sense in that symmetry generally got better and flatness got somewhat worse. The right-most element was not being evaluated and generally speaking, that's where the beam is starting to fall off. So flatness would likely stay the same or get worse, never get better. Symmetry generally improved because now the calculation is actually being done for the points that are truly opposite it across the CAX. Previously, a given element was being compared to its opposite one element closer to the CAX than it should have been.

FFF beams change the most and this can be attributed to the larger gradients causing larger differences in the calculation for both symmetry and flatness.

I understand that this may cause some consternation because the values are suddenly changing. However, I believe this is an improvement for the better since it is now more accurate. Additionally, symmetry values are generally getting better, which is a good thing. Flatness is usually not within our control either so changes here are bothersome, but know that your energy likely hasn't changed. As always, measure PDD for true energy determination.

Even before this issue was raised, I have been working on refactoring the profile and field analysis modules to be easier to test as well as to extend. Stay tuned.

Thanks to Stephen Terry for pointing this out. We all get better together!

Winston-Lutz

- The WL module can now handle TIFF images. This is still provisional and may have bugs. Caution is warranted. See *Using TIFF images*.

Machine Logs

- Anonymization (`anonymize()`) of trajectory logs now includes the Metadata->Patient ID field in the .bin file for v4+ logs.

6.28.10 v 3.10.0

Machine Logs

- Trajectory Log CSV files now include the Jaw positions (X1, X2, Y1, Y2) as well as couch vert and couch pitch and roll if the couch was a 6D couch.
- Dynalog loading and Trajectory `to_csv` calls will now use UTF-8 encoding by default when reading/writing files.

Tomo

- The `TomoCheese` phantom can now accept density information via an `roi_config` parameter to `analyze`. This is completely optional. See *Plotting density*.
- A new method `plot_density_curve` is available. It requires that an ROI configuration has been passed per above.

Field Analysis

- The `results_data` from a `DeviceFieldAnalysis` was throwing an error previously. It will now return a `DeviceResult`, which is the same as a `FieldResult` save for ROI information since a device is set of profiles and does not have a ROI to speak of.

VMAT

Warning: The `SEGMENT_X_POSITIONS_MM` class attribute has been deprecated. Use the new `roi_config` parameter described below which is a replacement and more.

- The VMAT classes can now accept an ROI configuration dictionary to the `analyze` method. This replaces the `SEGMENT_X_POSITIONS_MM` attribute. This allows the user to pass in the same details as well as ROI names. See the updated *Customizing the analysis* Section.

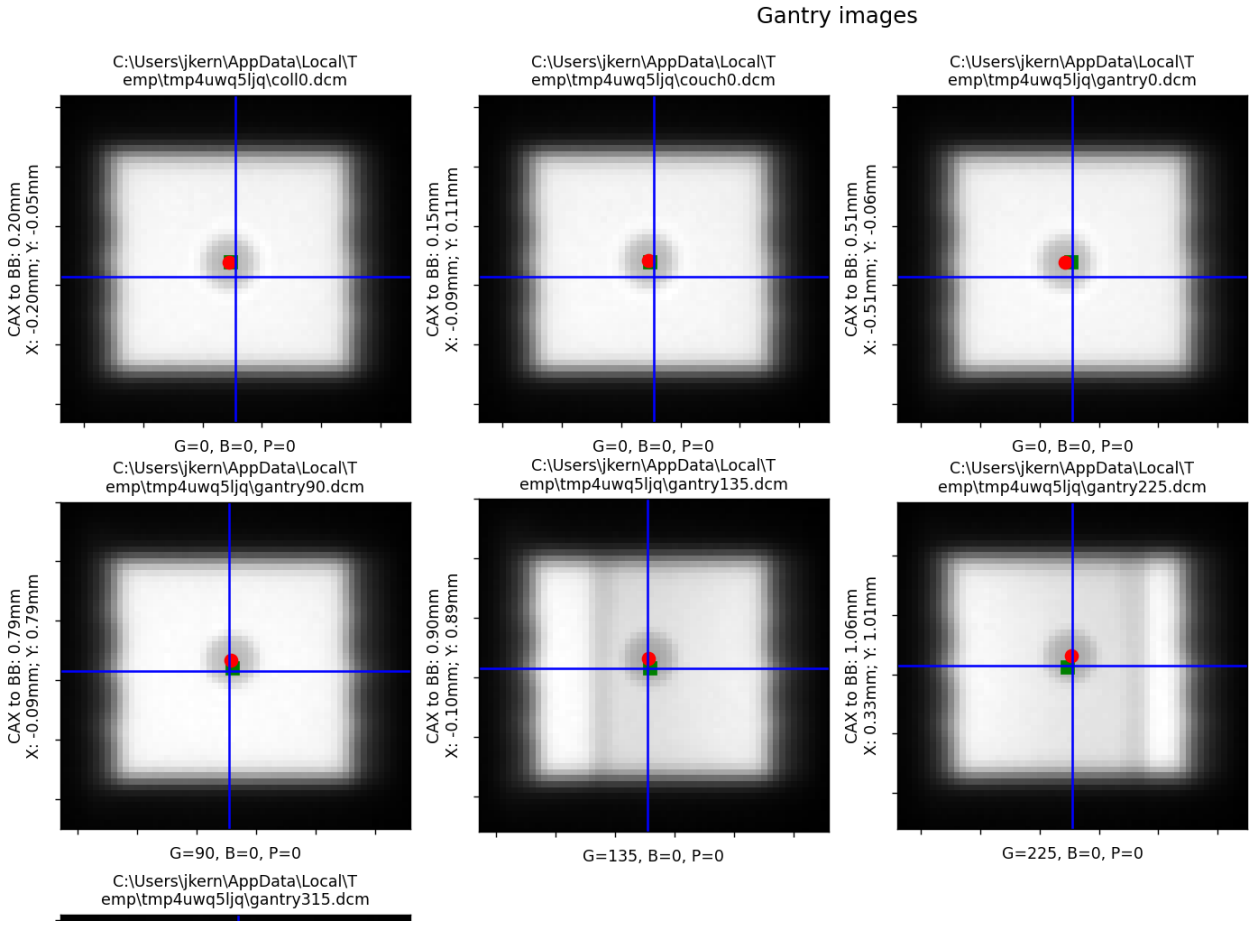
- The `VMATResult` class has a new attribute: `named_segment_data`. This is the exact same data as `segment_data` except it is a dictionary keyed with the same names given in the roi configuration. Note that for backwards compatibility `segment_data` has been kept.
- Plotting the analyzed image now renders the names of the ROIs on the image by default along with the ROI deviation value. A new parameter controls this in the `analyze` method: `show_text`.

Winston-Lutz

- Analyzing kV WL images is now a bit easier. A new parameter `open_field` has been added to the `.analyze` method. Setting this flag to True will set the field center to the center of the image. See the new section: *kV Analysis/Imaging-only iso evaluation*.
- Very small BBs (<2mm) may not be found. Pylinac was never meant to handle BB's smaller than this, but it may have worked. This is now hardcoded because pylinac will add a tolerance of +/-2mm to the input BB size. For inputs of 2mm BB size, this would lead to almost *any* ROI being detected. This is far more likely in phantoms where there is a block + BB vs a BB in air alone. Issues finding very small BBs were resolved with this hard lower limit.

> **Warning:** It is very unlikely but this may break your analysis if your BB is very small (<1.5mm diameter). If you are affected please reach out on the forum and I will provide you a workaround.

- Winston-Lutz individual images will now show the X and Y component of the distance to the BB.



- A new key has been added to the *WinstonLutzResult* class (what is returned from `results_data()`) called `keyed_image_details`. This is a dict that lets the user key off of the axes values. E.g. `data['G0C90B0']` will return the *WinstonLutz2DResult* for that image. This is in contrast to the existing `image_details` attribute that returns a simple list of the results. Images that are taken at the same axes values have a `_{idx}` appended to them. E.g. 3 images at the same position would look like `G0C0B0`, `G0C0B0_1`, and `G0C0B0_2`.

```
wl = WinstonLutz(...)
wl.analyze(...)
results = wl.results_data()
# knowing a priori I had a G90C0B0 image
g90_image_data = results.keyed_image_details["G90B0P0"]
# this is in contrast to having to iterate/search over the images
g90_image_data = [r.gantry_angle == 90 for r in wl.images][0].results_data()
```

- The user can now pass the precision desired for the axes values using a new parameter: `axes_precision`. This lets the user decide how to round (if at all) the axes values. E.g. a gantry at 90.1 with `axes_precision=0` will get rounded to 90. This can be useful with the above if using string keys to get details from a specific image as per the example above. E.g.:

```
# Assume an image set with G=359.9

wl = WinstonLutz(...)  # default, no rounding.
wl.analyze(...)
wl.results_data().keyed_image_details[
    "G359.9B0P0"
]  # we would have to know the delivery was at 359.9 and use the appropriate key

# vs
wl = WinstonLutz(..., axes_precision=0)
wl.analyze(...)
wl.results_data().keyed_image_details[
    "G0B0P0"
]  # whether delivered at G=359.9 or 0.1, this will always round to the nearest␣
↪integer
```

Note: If you consistently deliver images on the "other side of 0" you may want to set `axes_precision=0` which will round to the nearest integer. I.e. if you usually do 359.9 and want it be displayed as 0 do the above. This is helpful for the example above where even if the image was at 359.9 or 89.9, setting `axes_precision=0` will let you use the same consistent key, such as `data['G0C0B0']` rather than having to do `data['G359.9C0B0']`.

Warning: Due to this new axes precision, the default sorting MAY result in a different sorting of the images. This would only affect you if doing `<wl>.images[idx]`. If images are delivered on the "other side of 0" the image will bubble down to the bottom of the stack. I.e. an image delivered as G=359.9, B=0, P=0.1 will now bubble to near the bottom of the stack because the images are sorted first by gantry. Previously, the image would be rounded under the hood to be G=0, B=0, P=0. You can largely restore the prior behavior by passing `axes_precision=0`

Core

- Using `pylinac.core.profile.stretch` is now deprecated and will flag a warning on usage. The only current usage in the library is for `load_multiples` with the parameter `stretch_each=True`. This is unlikely to be used by end users and will be removed in v3.11. A new function of the same name is now available as `pylinac.core.array_utils.stretch`. For the normal use case where an array is to be stretched to have a new minimum and maximum, the result is the same. The use case `stretch(..., fill_dtype=...)` is deprecated as it is confusing and can potentially error out going from integer-like dtypes to float-like dtypes.

  Deprecated since version 3.11.

- A new method `bit_invert` has been added to the Image classes and subclasses as well as Profile classes and subclasses. This lets the user flip the image bit-wise. This is a better alternative than the existing `invert` as it takes into account the datatype. This will eventually become the default inversion method.
- A new method `convert_to_dtype` has been added to the Image and Profile classes and subclasses. This method will let the user pass a new numpy datatype and the array and values will be converted to that new datatype. Unlike a simple datatype casting however, this will keep the relative values to the same w/r/t the datatype max and min. E.g. an array of type uint8 has an element of value 100. Converting this to uint16 would result in a new value of 25,690 (100/255 = 0.392 = x/65535, x = 25,690). This is mostly helpful for combining images together but is a generally-helpful way of converting datatypes regardless of use case.
- The default value for a profile's `normalize` method has changed from `max` to `None`. The same is true of an Image class's `normalize` method. `max` and `None` do the same thing and `max` is still a valid argument. No change is needed by the user.
- Precision for axes values of `LinacDicomImage` s and subclasses are now more consistent and also allow the precision value to be set using a new parameter to the init call: `axes_precision`. Previously, any angle between 359-360 and 0-1 were considered "0". However, this was not true for any other axes value. I.e. the above values were rounded, but no other rounding occurred. This would also only happen if using the automatic DICOM tag values. If the user passed in the axis values directly, they were used as-is. Now, the precision of all axes values can be set using the new `axes_precision` parameter. This will round the axes values to the given precision level. This will apply to both DICOM tag values as well as manually-passed values. The default behavior is to not perform any rounding. The only difference users may notice is that axes values about 359-1 are no longer rounded to 0 by default. To restore this type of behavior pass `axes_precision=0` which will round 359.5+ to 0 and 359.5- to 359.

6.28.11 v 3.9.1

- A missing dependency in the built wheel `tabulate` was added. This only affected users who were trying to use the new `WinstonLutzMultiTargetMultiField` class. This can also be remedied by installing the package on its own: `pip install tabulate`.

6.28.12 v 3.9.0

General

- A new dependency has been added: `tabulate`. This is a Python-only library used for the new multi-target WL module. It is also a dependency of `pandas`, which will likely be a dependency of pylinac in the future.

CatPhan

- ROI details have been added to the *CTP515Result* class.
- Passing `delta` to `save_analyzed_subimage` would fail because the parameter was not being passed. This is now fixed.

Cheese

- A new module for "cheese" phantoms has been created. Only one routine currently exists: the *TomoCheese*, but more will be added later. Documentation for this new phantom can be found here: *"Cheese" Phantoms*.

Winston-Lutz

- Multi-Target, Multi-Field Winston-Lutz is now available. This means phantoms such as the SNC MultiMet can be analyzed. The algorithm is generalized however, and any reasonable configuration of BBs can be analyzed, meaning custom phantoms and new commercial phantoms are easy to make. Read the new section *here*.
- BBs with low density compared to surrounding material can now be analyzed via a new parameter `low_density_bb`. See the *analyze()* method.

Image Generator

- The *generate_winstonlutz()* utility script now accepts a `field_alpha` and `bb_alpha` parameter to set each item respectively.

Bug Fixes

- Certain XIM images were failing to render. This has been fixed.

6.28.13 v 3.8.2

- Using `use_filenames` with `axis_mapping` when instantiating Winston-Lutz would not respect the `use_filenames` flag. Now, `use_filenames` takes precedent. Normally, these should not be used together since they are both trying to set the axis values.

6.28.14 v 3.8.1

- The SNC phantoms (kV, MV, MV 12510) have had their ROI localization algorithms adjusted slightly. These phantoms are commonly used with the acrylic jig. That jig is very dense and often causes issues detecting the phantom separate from the phantom itself. This fix should remove the effect of the acrylic jig and allow any jig to be used, assuming the central ROI area is not occluded.
- Winston-Lutz axis-specific RMS calculations ("Maximum <Gantry | Collimator | Couch> RMS deviation") from the `results` and `results_data` method calls were potentially erroneous if the maximum error was in a "Reference" image (gantry=coll=couch=0). Users are urged to upgrade if using these outputs. Note that the Maximum/Median/Mean 2D CAX->BB distances are unaffected.

6.28.15 v 3.8.0

General

- `.xim` files are now able to be opened. These are Varian-specific images usually taken during MPC or in service mode. Currently, it is not natively integrated into other analyses (e.g. analyzing a .xim picket-fence via `PicketFence(...)`), but depending on the usage it will have more mainstream support in the other modules. However, this will allow the user to export to other, common file formats like png, jpeg, and tiff as well as access the properties of the .xim image such as acquisition mode, MLC positions, etc. Read about it here: *XIM images*.

Image Generator

- The image generator module has had tests added to increase robustness as well as docstrings for the parameters.
- The `RandomNoiseLayer` has been adjusted to provide noise irrespective of the signal. Previously, the noise was dependent on the intensity of the pixel. To be consistent with the intention of applying dark current, the layer now adds noise consistently across the image. The default sigma value has been adjusted to be roughly the same as before.

Picket Fence

- The PDF generated when the orientation was up/down would sometimes occlude the text on the report. The image placement has been adjusted.

Winston Lutz

- The *results_data()* for a normal WL analysis now include the details of each image as well. I.e. Each *WinstonLutzResult* contains *N* *WinstonLutz2DResult* , one for each image, under the `image_details` key.

CBCT

- The MTF returned in `results_data` now includes 10-90 in steps of 10. Previously, only the 80, 50, and 30% were reported.

6.28.16 v 3.7.2

Field Analysis

- Performing a field analysis on a very small field (a few mm) would error out. To get around this, pass a larger `slope_exclusion_ratio` to `analyze()`.

6.28.17 v 3.7.1

Planar Imaging

- The SNC MV 12510 ROIs were slightly downscaled. This caused an issue in contrast and CNR calculation being lower than reality by ~20%. It was introduced in v3.6. Users are encouraged to upgrade if using this specific phantom analysis.

6.28.18 v 3.7.0

General

- Logos can now be passed to any `publish_pdf` method to insert a custom logo (e.g. an institution logo). The size of the logo as it appears on the PDF is fixed.

Picket Fence

- The `max_error_picket` and `max_error_leaf` have been added to the results returned from `<pf>.results_data()`.
- Elekta MLC options have been added to the *MLC* enum.

Planar Imaging

- Inversion detection for the Leeds and PTW EPID QC phantoms have been improved.

  Warning: If you are passing `invert=True` to the analyze method for these phantoms double check the outcome. There is a good chance that parameter can be removed.

- An angle check has been added to the SNC kV phantom. Previously, the angle was hardcoded at 135 degrees per the manufacturer recommendation. It now checks the detected angle. If the value is 135+/-5 degrees the detected angle is passed, otherwise an error is thrown.

CBCT

- The phantom center detection was refactored. This was because the RadMachine jig was touching the CatPhan and causing detection issues on a handful of slices. Unfortunately, these few handful of slices were important to the detection algorithm as they occurred around the HU linearity module for the 604. The phantom center of each slice along the Z axis (in/out) is now detected by fitting a 1D polynomial for all the slices where the phantom is detected. I.e. `x, y = f(z)`. This removes some of the error associated with having something touching the phantom for just a few slices. E.g. a clinic was using BBs on the side of their Catphan for alignment which was causing issues. Situations like these are more likely to be recovered from.

  Note: This change is internal and should not cause issues; all tests passed without modification but there is a small possibility a dataset with some kind of interference will now analyze and cause detection issues.

6.28.19 v 3.6.3

CBCT

- Cropping a catphan dataset before analysis would result in an analysis failure.
- Datasets that had a deep-curve couch very close to the phantom (e.g. head cradles) would fail.

6.28.20 v 3.6.2

CBCT

- The phantom center detection was refactored. This was because the RadMachine jig was touching the CatPhan and causing detection issues on a handful of slices. Unfortunately, these few handful of slices were important to the detection algorithm as they occurred around the HU linearity module for the 604. The phantom center of each slice along the Z axis (in/out) is now detected by fitting a 1D polynomial for all the slices where the phantom is detected. I.e. `x, y = f(z)`. This removes some of the error associated with having something touching the phantom for just a few slices. E.g. a clinic was using BBs on the side of their Catphan for alignment which was causing issues. Situations like these are more likely to be recovered from.

  Note: This change is internal and should not cause issues; all tests passed without modification but there is a small possibility a dataset with some kind of interference will now analyze and cause detection issues.

6.28.21 v 3.6.1

- Fixed a bug with the SNC MV phantom analysis where the ROI scaling for the entire phantom was slightly over-sized.

6.28.22 v 3.6.0

Planar Imaging

- Planar analyses had a discrepancy in the number of low-contrast ROIs "seen" in the plot vs what was given in the numerical results. This is because the numeric results were still using the older method of contrast analysis, which does not take into account the ROI size. The plot uses the newer method of *Visibility*. The quantitative results have been changed to use the visibility.

  Warning: Your detected ROIs may be different moving forward, although the visibility default value in the *analyze()* method was chosen to be as close as possible to the existing contrast results, meaning that the ROIs should be similar out of the gate. If you'd like to still use the older metric it is still available:

```
num_rois_simple_contrast = sum(
    roi.passed for roi in my_planar_phantom.low_contrast_rois
)
```

Picket Fence

- The *max_error_leaf* property will now return an int, where previously it returned a single-element list for classic/combined analysis. I.e. doing <pf>.max_error_leaf used to return something like [42] but now returns 42. The signature type has also been updated to reflect this. This change allows the user to do this: <pf>.plot_leaf_profile(leaf=<pf>.max_error_leaf, picket=<pf>.max_error_picket). Previously, this would fail because the max_error_leaf was a list and the user would have to do ...leaf=<pf>.max_error_leaf[0]....

Note: Users that perform "separate" analysis are unaffected (.analyse(... separate_leaves=True).

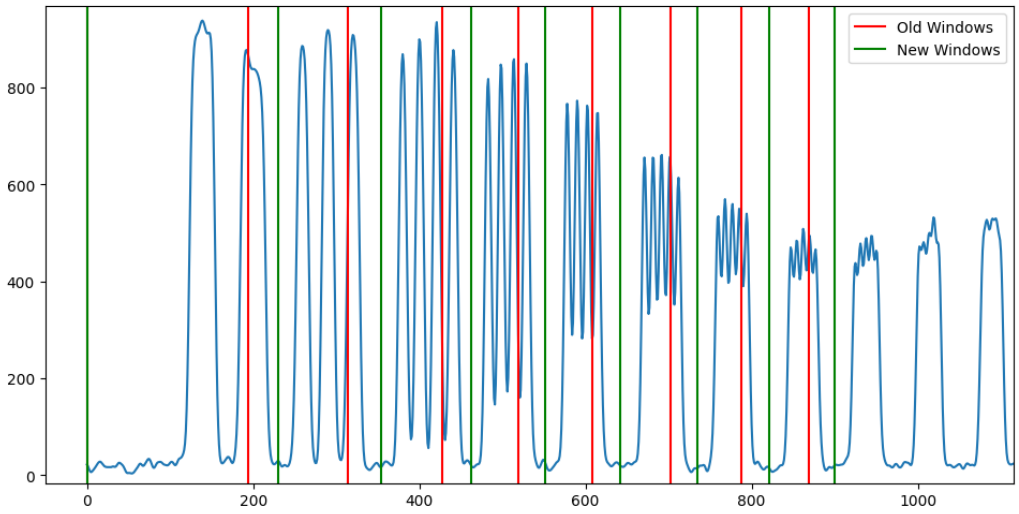
Winston-Lutz

- The BB-finding algorithm has been hardened and can now find the BB even in the presence of artifacts such as the couch. This most often applies when very large fields are used. A side effect is that the BB-finding algorithm is also now faster and reduces analysis time up to 50%.
- The machine coordinate system/scale can now be given as a parameter. This will affect the BB shift vector and shift instructions. The default scale is IEC61217, which was the implicit default previously and is thus backwards-consistent. A small section has been added here: *Passing a coordinate system*.
- Due to the above change, there is no need for the couch_angle_varian_scale property of the WinstonLutz2D class. It has been removed to reduce confusion. Use the new feature above if you had been using/overriding this property.
- A bug was fixed where repeating analysis would give different results. This was because the image pre-processing was being performed each time .analyze() was called. This only applies if you perform .analyze() more than once on the same instance.

Catphan

- The Catphan 600 MTF algorithm had a bug of not using the correct "windows" of peaks/valleys when finding the MTF. Each CatPhan model's high-resolution pairs are at slightly different angles. The 600 was inadvertently using the 504's window positions. This has been updated to use the correct windows. The problem can be visualized below, where the red lines show each MTF resolution window previously, vs the green which is the updated window. The result is that MTF will now be lower than previously because the old windows were sometimes including a peak of the previous line pair, causing the apparent MTF value to be higher than it really was.

Warning: MTF values for the CatPhan 600 will now be ~15% lower than previously due to this bug fix.



Field Analysis

- A visual bug was fixed with the blue ROI display. The horizontal ROI was being offset slightly based on the vertical width. This only applied when the width of the horizontal and vertical parameters were different and is completely visual. No quantitative results are affected.
- The statistics from the central area within the horizontal and vertical windows is now reported. I.e. the stats from the pixel values within the overlap of vertical window and horizontal window are now available like so:

```
fa = FieldAnalysis(...)
fa.analyze(...)
results = fa.results_data()
results.central_roi_max
results.central_roi_mean
...
```

The stats are also available directly from the FieldAnalysis instance:

```
fa = FieldAnalysis(...)
fa.analyze(...)
fa.central_roi.mean
fa.central_roi.max
...
```

If the width is 0 for both parameters a 2x2 matrix is sampled around the central pixel.

Core

- The *RectangleROI* class now has additional statistical results available computed from the pixel array: `.mean`, `.std`, `.min`, `.max`.

6.28.23 v 3.5.0

Planar Imaging

- Older SNC MV phantoms (observed as model #1251000) can now be analyzed with the new *SNCMV12510*. They have a slightly different size and ROI locations but appears to be functionally the same.
- The *IBA Primus A phantom* is now supported.
- Planar image analyses now take into account the image SAD; previously this was assumed to always be 1000mm. This only affects users with non-standard SADs such as proton gantries. Linac-based users should see no difference.
- Most planar phantoms will now show an "x" marker on the analyzed image showing the detected center of the phantom. This can help in evaluating the algorithm's accuracy in phantom detection.
- Two methods, `window_floor` and `window_ceiling`, were added to the image analysis classes. This lets the user define the min and max values of display for plotting the image. These are convenience functions only and currently only affect the Primus phantom, but will likely be adopted for the other phantoms.

Core

- A source-to-axis `sad` property was added to the `DicomImage` class. This property looks up the "RadiationMachineSAD" tag. This was added because non-1000mm SADs are being encountered.
- The `dpmm` property now takes into account the SAD (see above). Previously, the SAD was assumed to be 1000mm. For Linac users there will be no visible change.

Bug Fixes

- The PDFs from planar imaging analyses would have the text collapsed to one line. This has been fixed.
- The planar imaging module was starting to use scikit-image attributes that were introduced in 0.19 inadvertently. This has been fixed. For previous versions, update scikit-image to v0.19 or higher.

6.28.24 v 3.4.0

Picket Fence

- There is now a `skew()` method, returning the skew of each picket.

Planar Imaging

- A new class for analyzing older Leeds phantoms that have a blue label on the back (vs the red ring) has been added: *LeedsTORBlue*

Winston-Lutz

- The *cax2bb_distance()* method can now accept `mean` for the metric.
- The *cax2epid_distance()* method can now accept `mean` for the metric.
- The *results_data()* now includes the mean CAX->BB distance and mean CAX->EPID distance.

CT

- The *CatPhan600* detection has changed to use the bottom Air ROI and the Teflon ROI (just to the right of bottom air ROI). This is because the top air ROI can sometimes (and purposefully) contains a water vial. When inserted, the water vial makes angle detection untenable using this ROI. The result should be <0.5 degrees difference from previous versions, however, it was never 0. The only result this should affect (other than the angle) is the very small ROI low-contrast detection values, as it was found that even with a few tenths of degrees, a single pixel or two would be included or excluded compared to the previous algorithm. This is really a reflection of the sensitivity of the noise, which should likely use a global noise value instead of the local noise.
- Related to above, the same class now will have an extra ROI "Vial" with an expected value of 0. However, if the detected ROI is closer in value to air than water, the ROI will not be evaluated. This gives backwards-compatibility with existing scans that don't use the vial. I.e. if you don't use the water vial nothing should be different.

6.28.25 v 3.3.0

Core

- 1D gamma evaluation between two profiles can now be performed via the new *gamma()* function.
- Resampling of `SingleProfile` can now be done with the *resample()* function. This allows the user to re-sample a profile after it's already been created to achieve a specific interpolation resolution.

Field Analysis

- The `DeviceFieldAnalysis` class has been removed. Only the SNC Profiler was supported and even then it didn't work very well. Further, RadMachine is utilizing profile/file parsing that will be brought to pylinac. This new generalized scan parsing will eventually restore similar behavior, but for now it is deprecated. Sorry

Planar Imaging

- The SNC FSQA light/rad phantom is now able to be analyzed. Docs can be found here: *SNC FSQA*.

Bug Fixes

- #1705 - PDDx for measurements with no lead and PDD < 75 would calculate using the interim equation of 1.267*pdd - 20. This should return the PDD if the PDD<75. This will result in ~0.3% difference for 10MV with PDD just under 75. Depending on the chamber you're using, this could result in a difference of kQ by ~0.0005.
- The planar imaging detection routines have slightly improved robustness. This was caused by using scikit-image's `major_axis_length` property, which is somewhat more finicky than other properties. The detection now uses the `area_bbox` property which appears to curb some edge-case phantom analyses. This should not affect results for images that are already detected properly.
- Linear and Spline interpolation for `SingleProfile` contained an error in how it was interpolating data (it wasn't) at the very edges. The problem is that if we upsample, the left and right ends are not equally sampled. E.g. upsampling a 3-pixel array (0, 1, 2) by 10 normally results in ~20 elements. You interpolate between 0 and 1, and 1 and 2. The first issue is that you do not have a simple X proportion of elements (3 * 10 = 30 but we get 20). Additionally, if these are pixels they have a finite, physical size and technically those values are at the center of the pixels. Thus, you actually need to sample beyond the left and right edges. In the above case you'd really need to sample from approximately -0.5 to 2.5 to get ~10 pixels for each original pixel. We also need to offset the x-values to be back to 0 again from -0.5. We solve this by offsetting the new x-values by a proportion of the sampling ratio. A ratio of 1 (identical sampling) should not have any offset and return the same values. As the ratio goes up, we approach the limit of 0.5 pixels. This follows a proportional relationship with the ratio. The end result actually does not change much in the way of measurement results as nearly every previously-existing tests passed. 2 out of ~50 field analysis tests had a slightly different penumbra measurement and 1 had a slightly changed vert symmetry.

6.28.26 v 3.2.0

General

- The codebase as been blackened. This does not affect functionality but will change code line numbers when comparing to previous versions.
- All internal imports have been converted to relative imports. This does not change functionality but does mean that the pylinac repo can now be easily forked and included as a package in RadMachine. This would typically be done to use a pinned version as the embedded pylinac in RadMachine is updated regularly.
- Pylinac has dropped support for Python 3.6, following the security support timetable. 3.7 support will drop in the next version after June 2023.

Planar Imaging

- The IMT L-Rad light/rad phantom is now able to be analyzed. It is part of the planar imaging module. Docs are *here*.
- The SI QCkV phantom was accidentally analyzing the reference/background ROI. This resulted in a contrast of 0 for the first ROI all the time. This has been removed from the results. Calculations using the average contrast will be affected. Existing ROI analysis values are not affected, but will be off by one if accessing the roi directly. I.e. "roi 3" is now "roi 2", etc as the reference ROI was originally "roi 1".

CT

- The Quart phantom can now be analyzed. Docs are *here*.
- The ACR CT and ACR MRI Large phantom can now be analyzed. These should be considered experimental and subject to breaking changes in future versions until substantial data/tests can be had. Docs are *here*.
- The catphan and quart classes have a new attribute: `hu_origin_slice_variance`. This allows users to override the acceptable variance used to find the HU linearity module. Existing functionality is not changed.

Winston-Lutz

- Axis data can now be passed in as a dictionary. This is mostly for Elekta users. This is an alternative to renaming files. See the updated section on *passing in data*.
- The `ImageManager` class has been removed. The functionality has been absorbed into the existing classes.

6.28.27 v 3.1.0

General

- For the picket fence, field analysis, and planar imaging modules, image keyword args can now be passed on instantiation. This is helpful for images that don't have even basic tags like DPI/DPMM or SID. The keyword args that can be passed are those consumed by *load()*.

```
from pylinac import PicketFence

path = ...  # very sad image that has no DICOM tags for DPI or SID
pf = PicketFence(path, image_kwargs={"dpi": 184, "sid": 1500})
pf.analyze()
...
```

- Matplotlib keyword args can now be passed to most modules that save a figure, allowing the user to specify the figure size and other parameters

```
from pylinac import LeedsTOR

leeds = LeedsTOR.from_demo_image()
leeds.analyze()
leeds.plot_analyzed_image(
    ..., figsize=(10, 10)
)  # figsize is passed to matplotlib to generate a figure of said size
```

- Pylinac is now compatible with scikit-image 0.19

Picket Fence

- Individual leaf errors (on each side of the picket) can now be analyzed. New parameters were introduced to add this and related information needed to compute this. For backwards-compatibility this is set to False. See the *picket fence documentation* and `analyze()` parameter descriptions, specifically the `separate_leaves` and `nominal_gap_mm` parameters.
- Algorithm benchmarking has been added to the PF docs.

Planar Imaging

- The Standard Imaging FC-2 light/rad phantom is now able to be analyzed.
- The Las Vegas contrast analysis has been reverted to pre-3.0 behavior. This is because there is no reference position like there is for other phantoms. Mistakenly, the "reference" was set to the first ROI, but because visibility is dependent on both ROI size and contrast for Las Vegas, the background ROIs outside the milled disc areas have been restored.
- Plots can now be separated. Use `.plot_analyzed_image(... split_plots=True)`. This will now show multiple matplotlib plots.
- You may save analyzed images to individual files. I.e. when splitting per above each plot will be saved to a separate file. See `save_analyzed_image()`. This will return the filenames on disk.
- Finally, you may save split plots to stream using `to_streams`. This will return a dictionary of the plot name (image, low contrast, ...) and stream.

Field Analysis

- The plotting behavior described above for planar imaging is also true now for field analysis.
- Passing a string for centering, interpolation, edge and normalization methods is now an option. E.g. `<field analysis instance>.analyze(..., centering='manual', ...)`.

CBCT

- The catphan module can now accept a list of paths on instantiation. E.g. `Catphan504([path1, path2, path3, ...])`.

Winston-Lutz

- The `plot_summary()` method now allows you to pass a figure size.
- With the above, `save_summary()` also allows you pass the figure size.

Bug Fixes

- #1464 - Off-center CBCT could give faulty slice thickness numbers. The row/col were inverted for the sampling, meaning the left ROI was really sampling the top ROI and vic versa. For an on-center catphan, this would not change the results. Results appear to only have changed if the catphan was 5+ mm off-center. The change of outcome for offsets large than this are indeterminate but likely you weren't getting good results to begin with under that scenario, so it should only improve.
- #405 - The picket fence `results()` were reporting the wrong picket for the maximum error. It was selecting from a wrongly-ordered list, instead giving the picket with the **least** error. Note that the maximum error value was not incorrect, only the reported picket.
- PDF generation for field analysis with a device (i.e. SNC Profiler data) would fail as there was no true image. The PDF generation simply skips the image plotting for devices now.
- #416 - The CBCT docs now correctly state that the slice thickness is based on all the wire profiles, not just the longest two.
- #408 - The Dynalog isoplane correction factor was changed from 1.99614 to 1.96078 to match Varian documentation. This should have a difference of <0.3% of positioning error and should not affect gamma (since the errors canceled out) but would affect comparison to a TPS fluence.

6.28.28 v 3.0.0

Warning: Version 3.0 contains numerous breaking changes (hence the increment). Review the changelog before upgrading.

General

- A new method, `results_data` has been added to most modules (excluding calibration and log analyzer). This is complementary to `results`. `results_data` will return a dataclass or dictionary, which includes pretty much everything in `results` as well as metadata (e.g. pylinac version). This dictionary will be useful for APIs and referencing certain information that will be more stable across versions . Thanks to @crcrewso for the suggestion.
- Nearly all major modules can now handle file objects and streams (Dynalogs cannot yet). These may be passed as would a disk file path.

```
with open("mystarshot.dcm", "rb") as f:
    star = Starshot(f)
    ...
```

- Enums have been added in numerous places to mostly replace string options. E.g. for picket fence instead of specifying "up-down" as the orientation literally, the user now has the option to pass an Enum:

```
from pylinac.picketfence import PicketFence, Orientation

pf = PicketFence(...)
pf.analyze(..., orientation=Orientation.UP_DOWN)  # specify the orientation via an␣
↪Enum
```

The advantage here is two-fold: 1) introspection/autocompletion using your IDE vs remembering/looking up documentation, 2) easier to generate documentation as now we can point to a class with the options. Note however that string options are still available for backwards compatibility.

```
pf = PicketFence(...)
pf.analyze(
    ..., orientation="Up-Down"
)  # specify the orientation via a string. Works the same as above
```

Assuming you'd like to use the string version instead of using enums all over, how do you know the options? Go to the auto-generated documentation of the enum! =) E.g. *Orientation*.

Note: Relying on your IDE is a good idea. A smart one can warn you of incompatible data types.

- The github repo has been "minified" by removing excess demo files and also removing the basic test files. These files are now cloud-hosted and downloaded as needed. This makes `git clone` significantly faster since the repo size has been reduced from ~1.6GB to ~60MB. Note that this does not affect the pip package since that package already had most of this excess data removed.
- Image inversion detection has changed slightly. Some images have proper tags such as rescale slope and intercept. If they do have the tags, they are applied and no inversion is applied. If they do not have the tags, an inversion is then applied. Previously, the tags were applied if they were there, and nothing if not and inversion was ALWAYS applied. This should result in better inversion defaults for images from different machines/platforms and fewer `invert=True` additions. See *Images*.
- A `CONTRAST` enum has been added that can be used for low-contrast analysis of planar images and CBCT images. See *Contrast*.

```
from pylinac.core.roi import Contrast

leeds = LeedsTOR(...)
leeds.analyze(..., low_contrast_method=Contrast.WEBER)
...

ct = CatPhan504(...)
ct.analyze(..., contrast_method=Contrast.MICHELSON)
...
```

- The algorithm for low contrast contrast constant detection has changed slightly. See *Visibility*. This means the # of detected low-contrast ROIs may change for cbct. You may pass in a contrast technique per above and also a visibility threshold. See the `.analyze` method of the respective class.
- The contrast-to-noise property of the LowContrastDiskROI now uses contrast/stdev, where contrast is defined/chosen per above.
- Several LowContrastDiskROI properties have been deprecated such as `contrast_constant`. Use `visibility` instead. The old properties still work but come with a deprecation warning and will be removed in a future release.
- #270 Pylinac had a memory leak that was apparent when running on a server. This was caused by old instances being held in memory from and incorrect usage of the `lru_cache`. This has been fixed.
- Documentation about topics has been added.
- Documentation benchmarking several algorithms has been added. See the "Benchmarking the Algorithm" section for vmat, winston-lutz, and starshot modules. Picket fence will come soon.

Note: Upgrade Hints

Besides the above notes and any module-specific steps, due to the modified method of loading images and inversion, other downstream modules may be affected. This means that some images that needed `invert=True` before may not

need it, and some images that previously worked may need an `invert=True`. So generally, if the image fails when it passed with previous versions, try adding/removing forced inversion first. This should only be an issue for older images. Images generated on new linac platforms should be handled just fine.

Dependencies

A new dependency has been added: `cached_property`.

Field Analysis (previously Flatness/Symmetry)

Danger: This release introduced numerous breaking changes to this module. Existing code will break.

- Two classes are now offered: `FieldAnalysis` and `DeviceFieldAnalysis`.
- Many, many options were added to the *`analyze()`* method. See below and the documentation page for all the details.
- The `flatsym` module has been renamed to `field_analysis` to reflect the generalized nature of the module. Many thanks to Alan Chamberlain (@alanphys) for suggesting and doing the initial implementation for this. This also introduced some early support for NCS-33 , which gives guidance on FFF beams.
- From the above report, a "top" position as well as field slope values are calculated for FFF beams. See *FFF fields*.
- The new module can handle files from devices, specifically the SNC Profiler.
- Extensibility was greatly enhanced. Users can now easily add their own custom analysis routines to the module. See *Creating & Using Custom Protocols*.
- New options for *Centering*, *Normalization*, *Edge detection*, and *Interpolation* were introduced. Each of these can be granularly controlled.

VMAT

- Leveraging the new profile module, the field edge detection has been improved and can detect "wide-gap" or overlapping ROIs more robustly.

Calibration

- #353 The bounds for most functions/methods have been converted to constants. This lets users override the default values should they wish it.

Winston-Lutz

- #366 #333 The analysis will fail if the BB is not detected within 20mm of the center of the field. This should help artifacts from being detected.
- The Winston-Lutz analysis has added an `.analyze` routine, just like all other major modules.
- #358 The user can now pass in an expected BB size. This will help analyses with smaller or very large BBs.
- The `WLImage` class has been renamed to `WinstonLutz2D`. This is to clarify usage as now documentation has been expanded to show using WL with a single image.

Note: Upgrade Hints

- Replace any uses of axis constants (`GANTRY`, `COLLIMATOR`, etc) with the enum version: `Axis.GANTRY`, ...
- Add a `<instance>.analyze(...)` call to each `WinstonLutz` instantiation.
- Set the BB size if needed. The algorithm has a default of 5mm and is relatively forgiving (+/-2mm), but for very small BBs you should set it lower than the default of 5mm. E.g. `.analyze(bb_size_mm=3)`
- If using `WLImage`, rename to `WinstonLutz2D`. Add `.analyze()` calls as well as appropriate.

I/O

- An SNC Profiler file parser has been added: *pylinac.core.io.SNCProfiler*. This can be used standalone, but since the data is not encoded to begin with it's really about handling it as a tool for other modules. Currently, this is being used in the Field Analysis module.

```
from pylinac.core.io import SNCProfiler

snc = SNCProfiler("path/to/data.prs")
snc.data  # ndarray
x, y, pos, neg = snc.to_profiles()  # returns SingleProfiles
```

Planar Imaging

- Sun Nuclear kV and MV phantoms have been added to the arsenal.
- The PTW EPID QC phantom has been added to the arsenal.
- The Standard Imaging QC-kV1 phantom has been added to the arsenal.
- #339 The user can now pass an SSD value for their phantoms. The default is 1000mm, but if you set it on your panel you can pass something like 1400mm.
- The phantom-finding algorithm has been refactored to be more extensible. This does not affect normal users, but reduces the amount of duplicate code. It also makes adding new phantoms easier.
- Generally speaking, the phantoms should all be roughly centered along the CAX. Previously, the phantom could be offset from the CAX. Due to general difficulty in finding the phantom reliably for the majority of clinics, I am enforcing this as a restriction. This shouldn't affect too many people but should make the ROI-finding algorithm better.
- The low contrast background ROI (i.e. the base level of contrast) has been adjusted for some phantoms (QC-3 and Doselab). Previously, it could either be in a "dark" region, meaning a high-attenuation area, or a "light"

region, meaning a low-attenuation area. This has been standardized for all phantoms to be the "light" region. A new doc page for contrast has been added to the online documentation.

- 3 more high-contrast ROIs have been added to the LeedsTOR to help get rMTFs below 50%.
- The SI QC-3 analysis will now handle both typical orientations (gantry 0 and 90), where the "1" is pointing toward the gantry. This produces two different angles. The phantom should still be angled at 45 degrees from a cardinal angle.

Note: Upgrade Hints

- If you have defined any custom phantoms, read the new documentation: *Creating a custom phantom*. Your existing code will likely NOT break but the new format is much easier for extensibility.
- Evaluate the new contrast values versus your existing ones for the QC3 and Doselab phantoms. Moving forward, the above definition of contrast ROI-picking will be used.
- For the LeedsTOR, check the MTF of an existing image. Since adding more high-contrast ROIs, the rMTF may change if you were using a value below the lowest detected value. You do/will get warnings about being below the minimum MTF if you already do so.

Picket Fence

Overall, most code shouldn't need to change from v2.5. From v2.4 or below, the way MLCs are passed and used has changed.

- Wide-gap tests should now work better than before. However, please read the *Acquiring good images* section.
- The `mlc` parameter of the `PicketFence` constructor has been changed to use an Enum or `MLCArrangement`: *`MLC`*. See the *Customizing MLCs* section for more.
- A `crop_mm` parameter has been added to the `PicketFence` constructor. This is for cropping the edges of images. The primary cause of issues with the PF module is dirty/noisy/dead edges.
- The `orientation` parameter of the `analyze` method has been changed to use an Enum or str: *`Orientation`*.
- A `required_prominence` parameter has been added to `analyze`. This is to prevent multiple peaks detection for wide-gap images.
- A `fwxm` parameter has been added to `analyze`. This is to allow the user to set the FWXM height to use for the MLC kiss profile.
- A `results_data` method has been added. See General above.
- The colored rectangular overlay has been reduced in size slightly.

CBCT

- A `contrast` parameter was added to analyze. This uses an Enum and has 3 options; see *Low contrast*.
- A `visibility_threshold` parameter was added and is a replacement for `cnr_threshold`. See the General section and *Visibility*. Compared to `cnr_threshold`, the default value will give approximately the same results for # of low-contrast ROIs "seen". About 30% of the test datasets had a different # detected, but the detected vs expected number were either too high or too low, so there was no single value to perfectly replace the default `cnr_threshold` value.
- With the above, the contrast calculations have been standardized. Compared to previously, the contrast and contrast-to-noise now use the same equation for contrast. Previously, contrast was using the Michelson equation

and contrast-to-noise was using the Weber definition. Now, contrast is always calculated with the definition given during instantiation.

- ROI colors for low contrast ROIs that are "seen" have changed from blue to green to match other modules.

Note: Upgrade Hints

- Change/check the contrast method of *.analyze()*.
- Change/check the visibility threshold of *.analyze()*.
- Verify the # of low contrast ROIs "seen".

Machine logs

- #161 Trajectory logs v4.0 are now supported

6.28.29 v 2.5.0

Warning: There appears to be an issue with reading TIFF images on Windows with libtiff=4.1.0. If you experience TIFF header errors, downgrade libtiff to <4.1.

General

- This release adds utility functions to the image generator module and also a change in configuration of the picket fence module, allowing users to create their own MLC configurations.

Dependencies

- `py-linq` has been added as a dependency. It's pure python so it will not add secondary dependencies.

Picket Fence

- MLC configuration has changed from being empirical to a priori, meaning that leaves are no longer determined, but passed in via configuration. This allows users to configure their own custom MLCs arrangements. See *Customizing MLCs*.
- Linked with the above, the `is_hdmlc` parameter is deprecated and users should now use the `mlc` parameter in the constructor.
- Also due to above, new parameters have been added to the `analyze` method. Please see the documentation for more info.
- The colored overlay is now broken up into the individual leaf kisses rather than one line.
- Several internal classes were removed or overhauled. This should not affect you if you're just using the basic routines like analyze(). `Settings` no longer exists, `MLCMeas` is now `MLCValue`. `PicketManager` no longer exists.

VMAT

- The ROI segment size can now be specified in `analyze`. This is discussed in the new section *Customizing the analysis*.

Image generator

In the previous release, a new image generator module was introduced. This release adds utility scripts for easily creating Winston-Lutz and picket fence image sets. See the Helpers section of the generator documentation.

6.28.30 v 2.4.0

General

Thanks to several contributors for making pull requests in this release!

- A new image generator module has been added. This module can generate custom test images easily: *Image Generator*.
- The core peak-finding functionality used in several modules was refactored to use scipy's implementation. When pylinac was built, such a function did not exist. Now that it does, the custom code has been removed (yay!). The major difference between this implementation and pylinac's is the use of "prominence", which is a concept I had never heard of. The resulting peak-finding functionality is the same for max-value peak-finding. For FWXM peak finding, this can have small differences. The biggest differences would be for profiles that have a very asymmetric "floor". I.e. if one valley on one side of the peak has a very different value than the other side then a difference would be detected. Fortunately, this is a very rare scenario.
- Documentation plots have been updated to be generated on-the-fly. This will result in better agreement with documentation plots vs. what people experience. Previously, some old figures were used that did not match the functionality.
- The GUI function was removed from the pylinac init file. This was causing issues when deploying to Heroku as calls to tkinter caused failures. The GUI should be called from the submodule now:

```
# old
import pylinac

pylinac.gui()

# new
from pylinac.py_gui import gui

gui()
```

Dependencies

Two requirements have been bumped: `scipy>=1.1` and `scikit-image>=0.17`.

CT Module

If you do not perform any advanced functionality, no changes are noteworthy.

The CT module has been reworked to be far more extensible to adjust individual component modules as desired. Previously, only the offset of the modules was easily adjustable. To edit individual modules the user would have to edit the source code directly. Now, the user can subclass individual modules, overload attributes as desired and pass those to the parent CatPhan class. A new tutorial section has been added to the documentation showing examples of this functionality.

- The CTP404 and 528 modules have been refactored into CatPhan-specific classes for easier overloading by appending "CP<model>". E.g. CTP404CP503.
- CTP modules had an inconsistent naming scheme for rois. E.g. CTP404 had `hu_rois` and `bg_hu_rois` while CTP515 had `inner_bg_rois` and `rois`. This has been standardized (mostly) into `rois` for all modules and, where applicable, `background_rois`. Some modules still have **more** relevant attrs, e.g. `thickness_rois` for CTP404, but they all have have `rois`.
- Due to the above refactor, you may notice small differences in the contrast constant value and thus the ROIs "seen".
- HU differences are now signed. Previously the absolute value of the difference was taken.
- HU nominal values have been adjusted to be the mean of the range listed in the CatPhan manuals. The changes are as follows: Air: N/A (this is because most systems have a lower limit of -1000), PMP: -200 -> -196, LDPE: -100 -> -104, Poly: -35 -> -47, Acrylic 120 -> 115, Delrin: 340 -> 365, Teflon: 990 -> 1000, Bone (20%): 240 -> 237, Bone (50%): N/A.

Flatness & Symmetry

The flatness & symmetry module has been updated to allow for profiles of a select width to be analyzed rather than a single pixel profile.

- A `filter` parameter has been added to the constructor. This filter will apply a median filter of pixel size x.
- Due to the new peak-finding function, flatness and symmetry values may be slightly different. In testing, if a filter was not used the values could change by up to 0.3%. However, when a filter was applied the difference was negligible.
- Two new keyword parameters were added to analyze: `vert_width` and `horiz_width`. You can read about their usage in the `analyze` documentation.
- The `plot()` method was renamed to `plot_analyzed_image()` to match the rest of the modules.

Watcher

The watcher script has been officially deprecated for now (it was broken for a long time anyway). A better overall solution is to use something like QATrack+ anyway =).

Bug Fixes

- #325 The Leeds angle detection should be more robust when the phantom angle is very close to 0.
- #313 The catphan CTP486 module had an inverted top and bottom ROI assignment.
- #305 The Leeds `invert` parameter was not being respected.
- #303 Un-inverted WL image analysis would give an error.
- #290 Catphan HU linearity differences are now signed.
- #301 Loading starshots and picket fences from multiple images has been fixed.
- #199 Printing Picket Fence PDFs with a log has been fixed.

6.28.31 v 2.3.2

Bug Fixes

- #285 The SI QC-3 module was incorrectly failing when the phantom was at 140cm due to a faulty mag factor.

6.28.32 v 2.3.1

Bug Fixes

- #281 The ct module had a wrong usage of the new MTF module that caused a break.

6.28.33 v 2.3.0

General

- The dependencies have been updated. Scikit-image min version is now 0.13 from 0.12. There is also no upper pin on numpy or scikit-image.
- The planar imaging module was overhauled.
- An MTF core module was introduced to refactor and standardize the MTF calculations performed across pylinac.
- The Winston-Lutz 2D and 3D algorithms were improved.

Winston Lutz

- The coordinate space definition has changed to be compatible with IEC 61217. This affects how to understand the 3D shift vector. The `bb_shift_instructions` have been modified accordingly to still give colloquial instructions correctly (i.e. "Left 0.3mm").
- The WL module received an internal overhaul with respect to the 3D shift algorithm (i.e. the BB shift vector/instructions). The 3D algorithm was reimplemented according to D Low's 1994 paper. Generally speaking, the results are more stable across multiple datasets, however, you may see individual differences of up to 0.3mm.
- Due to above, the `bb_<axis>_offset` and `epid_<axis>_offset` properties have been removed.
- Two new image categorizations have been added: `GB Combo` and `GBP Combo`. These represent a gantry/collimator combination image with the couch at 0 and gantry/collimator/couch image where all axes are rotated. `GBP Combo` is a replacement for `ALL`. This change should only affect users who explicitly call methods that ask for the image set like `.axis_rms_deviation`, `.plot_axis_images`, etc.
- A new property has been added: `.gantry_coll_iso_size` which calculates the isocenter size using both gantry and collimator images.
- A new property has been added to individual images: `.couch_angle_varian_scale`. This conversion is needed to go from IEC 61217 to "Varian" scale for proper 3D shift vector calculation per the 3D algorithm change. Users likely wouldn't need this, but it's there.
- The 2D CAX->BB vector is improved slightly (#268). Thanks to @brjdenis and @SimonBiggs for bringing this to my attention and helping out.

Planar Imaging

- The Doselab MC2 (MV & kV) phantom has been added to the planar imaging module.
- The planar imaging module has been overhauled. The automatic detection algorithms have been spotty with no easy way of correcting the inputs. Further, each phantom had a few subtle differences making them just different enough to be annoying.
- To this end, the phantom classes have been refactored to consistently use a base class. This means all main methods behave the same and give a standardized output.
- Creating new custom phantom classes is now very easy. A new section of the planar imaging documentation has been added as a guide.
- A `results` method has been added to the base class, thus inherited by all phantom classes.
- The parameter `hi_contrast_threshold` has been refactored to `high_contrast_threshold`.
- The attributes `lc_rois` and `hc_rois` have been refactored to `low_contrast_rois` and `high_contrast_rois`, respectively.
- The `analyze` method now includes new standardized parameters `angle_override`, `size_override`, and `center_override`. Each of these is exactly what it sounds like: overriding pylinac's automatic algorithm. This is useful if the automatic algorithm gives an incorrect value.
- A phantom outline is now displayed on images. This outline is a simple representation and should only be used as a guide to the accuracy of the phantom spatial detection. I.e. you can use this outline to potentially override the center, size, or angle based on the outline.
- The automatic rotation analysis of the phantoms has been problematic. After spending a significant amount of time on the issue a satisfactory solution was not found. Therefore, the default angle or phantoms is that of the recommendation of the manufacturer. I.e. for the QC-3 phantom this means 45 degrees, as is the value when properly set up to the crosshairs.

- High and low contrast ROIs now show as red if they were below the defined threshold.

Core Modules

- A new core module `mtf` has been created to standardize all MTF calculations in pylinac. Previously, these were handled independently. The new module contains one class `MTF` with one method `relative_resolution` to calculate the lp/mm value at the passed rMTF percentage.

Bug Fixes

- This release contains critical fixes. All users of the Winston-Lutz and VMAT modules are strongly encouraged to upgrade as soon as possible.
- #268 The Winston-Lutz BB-finding method contained an error that would cause the BB center to be slightly off-center. After running unit tests, 5/16 datasets had a couch isocenter size difference of >0.2mm. Of those, 3 were around 0.2mm greater and 2 were around 0.2mm smaller. No other changes to iso sizes were detected within the testing tolerance of 0.2mm.
- #204 The VMAT module was sometimes using raw pixel values to calculate the ROI deviations. This would cause the deviations to appear smaller than they should have been if the Rescale and Intercept had been applied to the pixel data.
- #280 The Winston-Lutz 3D BB shift vector was underestimating the shifts by ~30-40%. A new 3D algorithm was implemented.
- #275 Requirements no longer have an upper pinning, although scikit-image minimum version was bumped from 0.12 to 0.13.
- #274 A new MTF module was created to refactor multiple ad hoc implementations.
- #273 The CatPhan HU module detection algorithm was loosened slightly to account for very thin slice scans which have increased noise.

6.28.34 v 2.2.8

General

Although the following changes should really mean a 2.3 release, I consider them small enough that I will keep it a maintenance release.

- An `invert` parameter was added to the `analyze` method of the FlatSym module so the user can override the automatic inversion.
- An `invert` parameter was added to the `analyze` method of the Starshot module so the user can override the automatic inversion.

Bug Fixes

- #272 An `invert` parameter was added to the `analyze` function of the starshot module. This allows the user to force invert the image if pylinac's auto-inversion algorithm is incorrect.
- #264/265 The `results` method for the flatsym module would err out when images with 0 flatness were used.
- #191 The flatsym module was not loading non-DICOM images properly, causing processing failures.
- #202 The rotation determination of the QC-3 phantom was often incorrect. This has temporarily been fixed by hardcoding the angle to 45 degrees. This is a correct assumption if the phantom is being used according to the instructions.
- #263 The FlatSym module was sometimes incorrectly inverting images. This was fixed using a better histogram methodology.
- #266 The deviation of a VMAT ROI was not properly detecting failing segments if the value was negative.
- #267 The `overall_passed` property of the CTP515 module contained an error that would cause an error.
- #271 The line pair/mm values for the CT/CBCT module was inadvertently doubled. I.e. the lines/mm was given, not line *pairs*.

6.28.35 v 2.2.7

Winston-Lutz

- A small change was made to the Winston-Lutz BB finding algorithm to be more robust and use less custom code. The output from WL analyses should be within 0.1mm of previous values.
- A section was added to the documentation to describe how images are classified and the analysis of output from the `.results()` method.

Bug Fixes

- #187 Scipy's imresize function has been deprecated. Functionality was converted to use `skimage.transform.resize()`.
- #185 Winston-Lutz PDF generation had an artifact causing catastrophic failure.
- #183 The Bakai fomula of the gamma calculation had an operational inconsistency such that dose-to-agreement other than 1% would give incorrect values of the gamma value.
- #190 The Catphan module had an inconsistency in the rMTF/spatial resolution determination. Some line pair regions would be detected for some phantoms and not for others. This was caused by the different CatPhan models having slighly different rotations of the CTP528 module. Pylinac now has model-specific boundaries.
- #192 The FlatSym plot would conflate the vertical and horizontal lines shown on the analyzed image. Analysis is unaffected, only the depiction of position.
- #194 The Leeds low contrast ROI color on the analyzed image was not consistent with the contrast plots. ROI color is now based on the pass/fail of the contrast constant, not the contrast.
- #196 Winston-Lutz images with a dense BB and low photon energy could cause BB detection to fail. A better BB-finding algorithm has been implemented.
- #197 EPID RMS deviation would return 0 for the .results() method always. This now calculates correctly.

6.28.36 V 2.2.6

Bug Fixes

- #157 This behavior is revered to pre-2.2.2 behavior to match the DFV and other software.
- **#167 Originally, the fix for this was to raise an error and point to a workaround. At the time the fix was to add a parameter to v2.3.**
  Behavior was able to be changed internally to handle this case without an API change.

6.28.37 V 2.2.5

General

The `watcher` function has had several issues. It has been disabled and will be removed in v2.3.

Bug Fixes

- #173 When forcing inversion of picket fence, the inversion came after the orientation determination, causing orientation to be wrong when inversion was needed.
- #171 The `load_log` function was not working correctly when passing a directory or ZIP archive.
- #172 Calling `publish_pdf` from log_analyzer without passing a filename would fail.
- #169 VMAT Dynalogs were calculating fluence incorrectly for CCW plans due to the gantry angle replacing the dose.
- #160 While addressing #160 initially, Trajectory logs were unknowningly affected. Behavior has been reverted to pre-2.2.2 behavior and documentation changed.

6.28.38 V 2.2.4

Bug Fixes

- #165 Machine log plots and PDFs showing the Leaf RMS were shown in cm, not in mm, as the axis title indicated.
- #167 Picket fence images where the pickets are too close to the edge perpendicular to the pickets will fail. This adds an explicit error and mentions a workaround. The next major version will include a `padding` parameter to apply this workaround.
- #168 Picket fence analyses now crop 2 pixels from every edge. This will allow Elekta images to be analyzed since they inexplicably have a column of dead pixels in EPID images. Should not affect Varian images.

6.28.39 V 2.2.3

Bug Fixes

- #158 Catphan roll determination algorithm has slightly widened the air bubble-finding criterion.

6.28.40 V 2.2.2

Bug Fixes

- #157 Dynalog MLC leaf error was calculated incorrectly. Expected positions were off by a row. Error results should be lower on average.
- #160 Dynalog MLC leaf internal pair mapping (1-61 vs 1-120) was different than documentation. Image calculations should not change.
- #162 The LeedsTOR `angle_offset` in the `.analyze()` method was not being followed by the high-contrast bubbles.
- #144 The LeedsTOR angle determination is much more robust. Previously, only certain orientations of the phantom would correctly identify.

6.28.41 V 2.2.1

Bug Fixes

- #153 Log analyser PDF publishing fix.
- #155 VMAT PDF report had tolerance listed incorrectly (absolute vs percentage) causing most tolerances to appear as zero due to rounding.

6.28.42 V 2.2.0

General

- #131 Typing has been added to almost every function and class in pylinac.
- F-strings have been incorporated. This bumps the minimum version for Python to 3.6.
- The `publish_pdf` method of every module has had its signature changed. Before, not all the signatures matched and only included a few parameters like author and unit name. This has been changed to `filename: str, notes: str, list of str, open_file: bool, metadata: dict`. Filename and open file are straightforward. notes is a string or list of strings that are placed at the bottom of the report (e.g. 'April monthly redo'). Metadata is a dictionary that will print both the key and value at the top of each page of the report (e.g. physicist and date of measurement)
- The TG-51 module has been placed under a new module: *Calibration (TG-51/TRS-398)*. This is because:
- A TRS-398 calibration module has been created *TRS-398*.
- The default colormap for arrays is now Viridis, the matplotlib default.
- A contributor's guide has been added: *Contributing*.
- #141 The Pylinac logo has been included in the package so that PDFs can be generated without needing www access.
- A new dependency has been added: argue which handles input parameters.

Flatness & Symmetry

- #130 The flatsym module has been completely rewritten. Documentation has also been updated and should be consulted given the number of changes: *Field Analysis*.

VMAT

- The overall simplicity of use has been increased by automating & removing several parameters.
- #128 The `VMAT` class has been split into two classes: *DRGS* and *DRMLC*. Although there are now two classes instead of one, the overall simplicity has been increased, such as the following:
  - The `test` parameter in `analyze()` is no longer required and has been removed.
  - The `type` is no longer required in `.from_demo_images()`.
  - The demo method matches the other modules: `.run_demo()`
  - All naming conventions have been deprecated.
- The `x_offset` parameter has been removed. The x-position is now based on the FWHM of the DMLC field itself. This means the x-position is dynamic and automatic.
- The `delivery_types` parameter has been removed. The delivery types of the images are now automatically determined.
- The methods for plotting and saving subimages (each image & the profiles) has been converted to a private method (`_plot_subimage()`, ...). There is little need for a public method to plot individually.

TG-51/Calibration

- #127 A TRS-398 module has been added. There are two main classes: `TRS398Photon` and `TRS398Electron`.
- #129 The TG-51 module has been refactored to add a `TG51ElectronLegacy` and `TG51ElectronModern` calibration class. The Legacy class uses the classic TG-51 values that require a kecal value and a Pgradient measurement. The Modern class uses the equations from Muir & Rogers 2014 to calculate kQ that updates and incorporates the Pgradient and kecal values. While not strictly TG-51, these values are very likely to be incorporated into the next TG-51 addendum as the kQ values for photons already have.
- Certain parameters have been refactored: `volt_high` and `volt_low` have been refactored to `voltage_reference` and `voltage_reduced`, `m_raw`, `m_low`, and `m_opp` have been refactored to `m_reference`, `m_reduced`, and `m_opposite`. These parameters are also the same for the TRS-398 classes (see #127).
- The `kq` function has been separated into three functions: `kq_photon_pdd10x`, `kq_photon_tpr2010`, and `kq_electron`.
- A PDD(20,10) to TPR(20,10) converter function has been added: `tpr2010_from_pdd2010`.
- Pressure and temperature conversion helper functions have been added: `mmHg2kPa`, `mbar2kPa`, `fahrenheit2celsius`. This can be used in either TG-51 or TRS-398 to get TPR without actually needing to measure it.
- Defaults were removed from most functions to avoid possible miscalibration/miscalculation.
- Most parameters of both TG-51 and TRS-398 were changed to be keyword only. This will prevent accidental miscalculations from simple positional argument mismatches.

Bug Fixes

- #138/#139: Too many arguments when plotting the leaf error subplot for picketfence.
- #133: Trajectory log HDMLC status was reversed. This only affected fluence calculations using the `equal_aspect` argument.
- #134: Trajectory log fluence array values were not in absolute MU.

6.28.43 V 2.1.0

General

- After reflection, the package seems to have bloated in some respects. Certain behaviors are only helpful in very few circumstances and are hard to maintain w/ proper testing. They are described below or in their respective sections.
- The command line commands have been deprecated. All commands were simply shortcuts that are just as easy to place in a 1-2 line Python script. There was no good use case for it in the context of how typical physicists work.
- The interactive plotting using MPLD3 has been deprecated. Matplotlib figures and PDF reports should be sufficient. This was a testing nightmare and no use cases have been presented.
- The transition of the method `return_results()` to `results()` is complete. This was baked-in from the very beginning of the package. It is expected that results would return something, nor is there any other corresponding method prefixed with `return_`.
- Pip is now the recommended way to install pylinac. Packaging for conda was somewhat cumbersome. Pylinac itself is just Python and was always installable via pip; it is the dependencies that are complicated. The wheels format seems to be changing that.
- Some dependency minimum versions have been bumped.

CatPhan

- The module was refactored to easily alter existing and add new catphan models.
- The CatPhan HU module classifier has been deprecated. Its accuracy was not as high as the original brute force method. Thus, the `use_classifier` keyword argument is no longer valid.
- CatPhan 604 support was added thanks to contributions and datasets from Alan Chamberlain. More datasets are needed to ensure robust analysis, so please contribute your dataset if it fails analysis.
- The CTP528 slice (High resolution line pairs) behavior was changed to extract the max value from 3 adjacent slices. This was done because sometimes the line pair slice selected was slightly offset from the optimum slice. Using the mean would lower MTF values. While using the max slightly increases the determined MTF from previous versions, the reproducibility was increased across datasets.

Winston-Lutz

- Certain properties have been deprecated such as gantry/coll/couch vector to iso. These are dropped in favor of a cumulative vector.
- A BB shift vector and shift instructions have been added for iterative WL testing. I.e. you can get a BB shift to move the BB to the determined iso easily.

```
import pylinac

wl = pylinac.WinstonLutz.from_demo_images()
print(wl.bb_shift_instructions())
# output: RIGHT 0.29mm; DOWN 0.04mm; OUT 0.41mm
# shift BB and run it again...
```

- Images taken at nonzero couch angles are now correctly accounted for in the BB shift.
- Images now do not take into account shifts along the axis of the beam (#116).
- The name of the file will now not automatically be interpreted if it can. This could cause issues for valid DICOM files that had sufficient metadata. If the image was taken at Gantry of 45 and the file name contained "gantry001" due to, e.g., TrueBeam's default naming convention it would override the DICOM data. (#124)

Picket Fence

- Files can now allow for interpretation by the file name, similar to the WL module. This is helpful for Elekta linacs that may be doing this test (#126).

Core Modules

- `is_dicom` and `is_dicom_image` were moved from the `utilites` module to the `io` module.
- `field_edges()` had the parameter `interpolation` added so that field edges could be computed more accurately (#123)
- A new class was created called `LinacDicomImage`. This is a subclass of `DicomImage` and currently adds smart gantry/coll/couch angle interpretation but may be extended further in the future.

6.28.44 V 2.0.0

General

- Version 2.0 is here! It may or may not be a real major version update worthy of '2.0', but '1.10' just didn't sound as good =)
- A GUI has been added! Most major modules have been added to the GUI. The GUI is a very simple interface that will load files and publish a PDF/process files. To start the gui run the `gui()` function like so:

```
import pylinac

pylinac.gui()
```

You may also start the GUI from the command line:

```
pylinac gui
```

The GUI is a result of a few causes. Many physicists don't know how to code; this should remove that barrier and allow Pylinac to get even more exposure. I have always felt the web was the future, and it likely is, but pylinac should be able to run on it's own, and because a rudimentary GUI is relatively easy, I've finally made it. The GUI is also free to use and has no hosting costs (unlike assuranceQA.com). Also, due to other ventures, a new job, and a newborn, I couldn't devote further time to the assuranceQA site–A native GUI is much easier albeit much more primitive.

- Some module PDF methods now don't require filenames. If one is not passed it will default to the name of the file analyzed. E.g. "abc123.dcm" would become "abc123.pdf". Modules where multiple images may be passed (e.g. a CBCT directory) still requires a filename.
- PDF methods now have a boolean parameter to open the file after publishing: `open_file`.
- A number of dependencies have been bumped. Some were for specific reasons and others were just out of good practice.

Watcher

- Closes #84 Which would overwrite the resulting zip and PDF of initially unzipped CBCTs performed on the same day. I.e. multiple CBCTs would result in only 1 zip/PDF. The image timestamp has been edited so that it will include the hour-minute-second of the CBCT to avoid conflict.
- Closes #86 - Which had a discrepancy between the YAML config setting of the file source directories and what the watcher was looking for.

CatPhan

- Closes #85 Which displayed the nominal CBCT slice width on PDF reports, not the detected width for the CatPhan504 & CatPhan600.
- Closes #89 which had variables swapped in the CatPhan503 PDF.
- The `contrast_threshold` parameter has been renamed to `cnr_threshold`. The meaning and values are the same, but has been renamed to be consistent with other changes to the `roi` module.
- Due to various problems with the SVM classifier, the default setting of the classifier has been set to `False`.

Planar Phantoms

- The Las Vegas phantom has been added to the planar imaging module. It's use case is very similar to the existing planar phantoms:

```
from pylinac import LasVegas

lv = LasVegas("myfile.dcm")
lv.analyze()
lv.publish_pdf()
...
```

- The *pylinac.planar_imaging.LeedsTOR.analyze()* method has an additional parameter: `angle_offset`. From analyzing multiple Leeds images, it has become apparent that the low contrast ROIs are not always perfectly set relative to the phantom. This parameter will allow the user to fine-tune the analysis to perfectly overlay the low contrast ROIs by adding an additional angle offset to the analysis.

Winston-Lutz

- Closes enhancement #63 Files can now have the axis settings interpreted via the file name. E.g: "myWL_gantry90_coll0_couch340.dcm". See *Passing in Axis values* for further info.
- The `x/y/z_offset` properties of the WLImages which were deprecated many versions ago have finally been removed.
- The `collimator/gantry_sag` and associated `plot_gantry_sag` methods have been deprecated. A similar method has been implemented that utilizes the RMS deviation. To achieve the "gantry sag" using RMS errors use the method `axis_rms_deviation` with parameter `value='range'`.

TG-51

- The Electron class has been adjusted to reflect the Muir & Rogers 2014 kecal data which allows the user to calculate kQ from just R50 data.
- The `kq` function now accepts an `r_50` parameter to calculate kQ based on the above data.

Core Modules

- The `Image` class has been fully deprecated and is no longer available. Use the functions available in the *Image Module* instead. See the version 1.4.0 release notes for further details.
- The `remove_edges` method has been deprecated and is now an alias for `crop`. The `crop` method should be used instead. Parameters are exactly the same.

6.28.45 V 1.9.0

General Changes

- This release introduces PDF reports for most major modules. All classes with this functionality have been given a `publish_pdf` method. This method takes an output filename and other optional data like the author, machine/unit, and any custom notes. See e.g. *pylinac.starshot.Starshot.publish_pdf()* or *pylinac.picketfence.PicketFence.publish_pdf()*.
- The watch/process functions have been tweaked to best work on one unit per run. Multiple units/machines should have their own config files. A new article describes how to use the process function with Windows Task Scheduler to regularly pull and analyze files.

CatPhan

- The CatPhan classes, when passed a directory during instantiation, will search through the DICOM files for Series UIDs and analyze the files of the most numerous UID. E.g. if a folder has 80 DICOM images including one set of 60 CBCT images and a total of 20 VMAT and picket fence images, it will find the CBCT files via UID and analyze those, leaving the other images/files alone. This is useful for when all QA images are simply dumped into one folder.
- Raw, uncompressed CatPhan DICOM files can optionally be compressed to a ZIP file after analysis using the new `zip_after` argument in the `analyze` method.

Watcher/Processer

- The `watcher/process` functions have been reworked to produce PDF files rather than PNG/txt files.
- If upgrading the watch/process function from a previous pylinac version be sure to copy/amend the new default YAML config file as new keywords have been added and using old YAML files will error out.
- Several new configuration keywords have been changed/added. In the general section, `use-classifier` has been deprecated in favor of individual module keywords of the same name. This allows a user to use a classifier for, say, picket fence images but not for winston lutz images. A `unit` keyword has been added that specifies which unit the files should be considered to be from. This unit name is passed to the PDF reports that are generated. If you have multiple units, make individual YAML configuration files, one for each unit.
- CatPhan, VMAT, and Winston-Lutz can now take raw, unzipped images as well as the usual ZIP archive. ZIP archives are detected only by keywords as usual. For uncompressed CatPhan images, the analyzer will look for any CatPhan DICOM file groups via UID (see above CatPhan section), analyze them, and then ZIP the images until no further sets can be found. For VMAT and Winston-Lutz if the `use-classifier` setting is true their respective sections in the YAML configuration then an image classifier is used to group images of the given type and then analyze them.

6.28.46 v 1.8.0

General Changes

- This release focuses solely on the CBCT/CatPhan module.
- Pylinac now has a logo! Check out the readme on github or landing page on ReadTheDocs.

Watcher/Processer

- The cbct analysis section has been renamed to `catphan`. Thus, the YAML config file needs to look like the following:

```
# other sections
...

catphan:  # not cbct:
    ...

...
```

CBCT/CatPhan

- The Python file/module has been renamed to `ct` from `cbct`. E.g.:

```
from pylinac.ct import ...
```

  Most users import directly from pylinac, so this should affect very few people. This was done to generalize the module to make way for other CT/CBCT phantoms that pylinac may support in the future.
- The CBCT module can now support analysis of the CatPhan 600.
- Automatic detection of the phantom is no longer be performed. Previously, it depended on the manufacturer to determine the phantom (Varian->504, Elekta->503), but that did not consider users scanning the CatPhan in their CT scanners, which would give inconsistent results.

- Due to the above, separate classes have been made for the CatPhan models. I.e. flow looks like this now:

```
# old way
from pylinac import CBCT
...

# new way
from pylinac import CatPhan504, CatPhan600
cat504 = CatPhan504('my/folder')
cat600 = CatPhan600.from_zip('my/zip.zip')
```

- A classifier has been generated for each CatPhan. Thus, if loading a 503, a 503 classifier will be used, rather than a general classifier for all phantoms.
- The `use_classifier` parameter has been moved from the `analyze()` method to the class instantiation methods like so:

```
from pylinac import CatPhan504
cat504 = CatPhan504('my/folder', use_classifier=True)
cat504.analyze()  # no classifier argument
```

- MTF is now more consistently calculated. Previously, it would simply look at the first 6 line pair regions. In cases of low mA or very noisy images, finding the last few regions would error out or give inconsistent results. Contrarily, high dose/image quality scans would only give MTF down to ~50% since the resolution was so good. Now, MTF is searched for region-by-region until it cannot find the correct amount of peaks and valleys, meaning it is now lost in the noise. This means high-quality scans will find and calculate MTF over more regions and fewer for low-quality scans. In general, this makes the MTF plot much more consistent and usually always gives the RMTF down to 0-20%.
- Individual modules are now only composed of 1 slice rather than averaging the nearby slices. Previously, for consistency, a given module (e.g. CTP404) would find the correct slice and then average the pixel values of the slices on either side of it to reduce noise and give more consistent results. The drawback of this method is that results that depend on the noise of the image are not accurate, and signal/noise calculations were always higher than reality if only looking at one slice.

6.28.47 v 1.7.2

- Fixed (#78) - Certain CBCT datasets have irregular background values. Additionally, the dead space in the square CT dataset outside the field of view can also be very different from the air background. This fix analyzes the dataset for the air background value and uses that as a baseline value to use as a CatPhan detection threshold.

6.28.48 V 1.7.0

General Changes

- The underlying structure of the watcher script has been changed to use a different framework. This change allows for analysis of existing files within the directory of interest.
- A new module has been introduced: `tg51`, handling several common equations and data processing for things relating to TG-51 absolute dose calibration such as Kq, PDDx, Dref, pion, ptp, etc. It also comes with classes for doing a full TG-51 calculation for photons and electrons with cylindrical chambers.

Log Analyzer

- The log analyzer has changed from having a main class of `MachineLog`, to the two distinct log types: `Dynalog` and `TrajectoryLog`. These classes are used the same way as machinelog, but obviously is meant for one specific type of log. This allows for cleaner source code as the `MachineLog` class had large swaths of if/else clauses for the two log types. But don't worry! If you're unsure of the log type or need to handle both types then a helper function has been made: `load_log`. This function will load a log just like the `MachineLog` did and as the new classes. The difference is it will do automatic log type detection, returning either a Dynalog instance or TrajectoryLog instance. The `MachineLogs` class remains unchanged.
- More specific errors have been introduced; specifically `NogALogError`, `NotADynalogError`, and `DynalogMatchError` which are self-explanatory and more specific than `IOError`.
- Fixed (#74) which was causing Dynalogs with patient names containing a "V" to be classified as Trajectory logs.
- Fixed (#75) which was skewing gamma pass percent values.

Planar Imaging

- The `PipsProQC3` class/phantom has been refactored to correctly reflect its manufacturer to Standard Imaging, thus the class has been renamed to `StandardImagingQC3`.

Directory Watching

- The `watch` command line argument now has a sister function, available in a regular Python program: `watch()`. With this command you can run the directory watcher programmatically, perfect for continuous log monitoring.
- A new command line argument is available: `process`. This command is also available in Python as `process()` which can be called on a directory either through the command line or programmatically and will analyze a folder once and then exit, perfect for analyzing a new monthly dataset.
- The structure of querying for files has been changed significantly. Instead of triggering on file changes (e.g. adding a new file to the directory), the watcher now constantly queries for new files at a specified interval. This means that when started, the watcher will analyze existing files in the folder, not just new ones.
- Information given in the email has been modified for logs, which may potentially contain PHI. Instead of the entire log file name given, only the timestamp is given. Additionally, the logs are no longer attached to the email.

6.28.49 V 1.6.0

General Changes

- Changed the default colormap of dicom/grayscale images to be "normal" gray vs the former inverted gray. Brought up in (#70) .
- Added a colormap setting that can be changed. See *Changing Colormaps*
- Added a utility function *clear_data_files()* to clear demo files and classifier files. This may become useful for classifier updates. I.e. the classifier for a given algorithm can be cleared and updated as need be, without the need for a new package release. More information on this will follow as the use of classifiers becomes normal.
- Added a dependency to the pylinac requirements: scikit-learn. This library will allow for machine learning advancements to be used with pylinac. I am aware of the increasing number of dependencies; pylinac has reached a plateau I believe in terms of advancement and I hope that this is the last major dependency to be added.

Winston-Lutz

- (#69) Added EPID position tracking. Now the EPID location will show up in images and will give an output value when printing the summary. Relevant methods like *cax2epid_distance()* and `epid_sag()`, and `plot_epid_sag()` have been added. The summary plot has also been changed to include two sag plots: one for the gantry and one for the EPID.
- Certain properties of WL images have been deprecated. `x_offset` has been replaced by `bb_x_offset()` and respectively for the other axes. Usage of the old properties will raise a deprecation warning and will be removed in v1.7.

Note: The deprecation warnings may not show up, depending on your python version and/or warning settings. See the python docs for more info.

CBCT

- Added a Support Vector Machine classifier option for finding the HU slice. The classifier is faster (~30%) than the brute force method. This option is available as a parameter in the `analyze()` method as `use_classifier`. In the event the classifier does not find any relevant HU slices, it will gracefully fall back to the brute force method with a runtime warning. Because of the fallback feature, the classifier is now used first by default. Using the classifier requires a one-time download to the demo folder, which happens automatically; just make sure you're connected to the internet.

Picket Fence

- An `orientation` keyword argument was added to the *analyze()* method. This defaults to `None`, which does an automatic determination (current behavior). In the event that the determined orientation was wrong, this argument can be utilized.

Watcher Service

- A new option has been added to the `general` section: `use-classifier`. This option tells pylinac whether to use an SVM image classifier to determine the type of image passed. This allows the user not to worry about the file names; the images can be moved to the monitored folder without regard to naming. The use of the classifier does not exclude file naming conventions. If the classifier does not give a good prediction, the algorithm will gracefully fall back to the file name convention.

  The following image types currently support automatic detection:

  - Picket Fence
  - Starshot
  - Leeds TOR
  - PipsPro QC-3

6.28.50 V 1.5.6

- Adds the `dtype` keyword to `DicomImage`'s init method.
- (#66) - Fixed an issue with Winston-Lutz isocenters not calculating correctly.
- (#68) - Fixed the order of the Winston-Lutz images when plotted.
- Many thanks to Michel for noting the WL errors and submitting the first external pull request !
- Fixed several small bugs and runtime errors.

6.28.51 V 1.5.5

- (#65) - Fixed the FlatSym demo file usage.

6.28.52 V 1.5.4

- (#64) - Fixed the Picket Fence offset from CAX value, which previously were all the same value.

6.28.53 V 1.5.1-3

General Changes

- Fixed conda entry points so that the user can use pylinac console scripts.
- Moved demo images outside the package to save space. Files are downloaded when relevant methods are invoked.

6.28.54 V 1.5.0

General Changes

- The pylinac directory watcher service got a nice overhaul. Now, rather than running the watcher script file directly, you can use it via the console like so:

```
$ pylinac watch "path/to/dir"
```

  This is accomplished through the use of console scripts in the Python setup file. Once you upgrade to v1.5, this console command immediately becomes available. See the updated docs on Directory Watching. Previously, customizing behavior required changing the watcher script directly. Now, a YAML file can be generated that contains all the analysis configurations. Create and customize your own to change tolerances and even to trigger emails on analyses.

- You can now anonymize logs via console scripts:

```
$ pylinac anonymize "path/to/log/dir"
```

  This script is a simple wrapper for the log analyzer's anonymize function.

- Pylinac is now on anaconda.org – i.e. you can install via `conda` and forget about dependency & installation issues. This is the recommended way to install pylinac now. To install, add the proper channel to the conda configuration settings.

```
$ conda config --add channels jrkerns
```

Then, installation and upgrading is as simple as:

```
$ conda install pylinac
```

The advantage of saving the channel is that upgrading or installing in other environments is always as easy as `conda install pylinac`.

- Pylinac's core modules (`image`, `io`, etc) are now available via the root package level.

```
# old way
from pylinac.core import image

# new way
from pylinac import image
```

Starshot

- Relative analysis is no longer allowed. I.e. you can no longer pass images that do not have a DPI or SID. If the image does not have these values inherently (e.g. jpg), you must pass it explicitly to the Starshot constructor. No changes are required for EPID images since those tags are in the image file.
- Added a `.from_zip()` class method. This can contain a single image (to save space) or a set of images that will be combined.

Log Analyzer

- The anonymize function received an optimization that boosted anonymization speed by ~3x for Trajectory logs and ~2x for Dynalogs. This function is *very* fast.
- Trajectory log subbeam fluences are now available. This works the same way as for the entire log:

```
log = MachineLog.from_demo_dynalog()
# calculate & view total actual fluence
log.fluence.actual.calc_map()
log.fluence.actual.plot_map()
# calculate & view the fluence from the first subbeam
log.subbeams[0].fluence.actual.calc_map()
log.subbeams[0].fluence.actual.plot_map()
```

- The gamma calculation has been refactored to use the image.gamma() method. Because of this, all `threshold` parameters have been changed to fractions:

```
log = MachineLog.from_demo_trajectorylog()
# old way
log.fluence.gamma.calc_map(threshold=10)  # <- this indicates 10% threshold
# new way
log.fluence.gamma.calc_map(threshold=0.1)  # <- this also indicates 10% threshold
```

The gamma threshold parameter requires the value to be between 0 and 1, so any explicit thresholds will raise an error that should be addressed.

- The `.pixel_map` attribute of the actual, expected, and gamma fluence structures have been renamed to `array` since they are numpy arrays. This attribute is not normally directly accessed so few users should be affected.

Bug Fixes

- Fixed a bug that would not cause certain imaging machine logs (CBCT setup, kV setups) to be of the "Imaging" treatment type.

6.28.55 V 1.4.1

- (#56) - Fixes a starshot issue where if the SID wasn't 100 it was corrected for twice.
- (#57) - CR images sometimes have an RTImageSID tag, but isn't numeric; this caused SID calculation errors.

6.28.56 V 1.4.0

General Changes

- Nearly all instance-based loading methods (e.g. `Starshot().load('myfile')`) have been deprecated. Essentially, you can no longer do empty constructor calls (`PicketFence()`). The only way to load data is through the existing class-based methods (e.g. `Starshot('myfile')`, `Starshot.from_url('http...')`, etc). The class-based methods have existed for several versions, and they are now the preferred and only way as there is no use case for an empty instance.
- Since v1.2 most URLs were downloaded and then the local (but temporary) files were loaded. This practice has now been standardized for all modules. I.e. any `from_url()`-style call downloads a temporary file and loads that. Because the downloads are to a temporary directory, then are removed upon exit.
- Loading images using the `Image` class has been deprecated (but still works) in favor of the new functions in the same module with the same name. Where previously one would do:

```
from pylinac.core.image import Image

img = Image.load('my/file.dcm')
```

One should now do:

```
from pylinac.core.image import load

img = load('my/file.dcm')
```

Functionality is exactly the same, but supports a better abstraction (there is no reason for a class for just behaviors). The same change applies for the other loading methods of the Image class: `load_url` and `load_multiples`. The `Image` class is still available but will be removed in v1.5.

Picket Fence

- `PicketFence` can now load a machine log along with the image to use the expected fluence to determine error. This means if an MLC bank is systematically shifted it is now detectable, unlike when the pickets are fitted to the MLC peaks. Usage is one extra parameter:

```
pf = PicketFence('my/pf.dcm', log='my/pf_log.bin')
```

Winston-Lutz

- A `from_url()` method has been added.
- Upon loading, all files are searched within the directory, not just the root level. This allows for nested files to be included.

CBCT

- The `from_zip_file()` class constructor method has been renamed to `from_zip()` to be consistent with the rest of pylinac's similar constructors.

Log Analyzer

- A new `treatment_type` has been added for CBCT and kV logs: `Imaging`.
- A new function has been added to the module: `anonymize()`. This function is similar to the `.anonymize()` method, but doesn't require you to load the logs manually. The function is also threaded so it's very fast for mass anonymization:

```
from pylinac.log_analyzer import anonymize

anonymize('my/log/folder')
anonymize('mylog.bin')
```

Starshot

- The starshot minimization algorithm has been changed from differential evolution to the more predictable minimize. Previously, results would *often* be predictable, but would occasionally give really good or really bad results even though no input was changed. This was due to the algorithm; now that a stable algorithm is being used, results are reproducible.

VMAT

- The VMAT loading scheme got a few changes. The Naming Convention is still the same, but images are always loaded upon instantiation (see General Changes). Also, if the naming convention isn't used, image delivery types can be passed in during construction; e.g.:

```
VMAT(images=(img1, img2), delivery_types=['open', 'dmlc']
```

- Loading from a URL has been renamed from `from_urls()` to `from_url()` and assumes it points to a ZIP archive with the images inside.

Bug Fixes

- (#47) - Fixes the trajectory log number of beam holds calculation. Thanks, Anthony.
- (#50) - Fixes RMS calculations for "imaging" trajectory logs. Previously, the RMS calculation would return `nan`, but now returns 0.
- (#51) - Results of the starshot wobble were sometimes extremely high or low. This has been fixed by using a more stable minimization function.
- (#52) - The starshot wobble diameter was incorrect. A recent change of the point-to-line algorithm from 2D to 3D caused this issue and has been fixed.
- (#53) - The Winston-Lutz BB-finding algorithm would sometimes pick up noise, mis-locating the BB. A size criteria has been added to avoid detecting specks of noise.
- (#54) - Imaging Trajectory logs, besides having no RMS calculation, was producing warnings when calculating the fluence. Since there is no fluence for kV imaging logs, the fluence now simply returns an 0'd fluence array.
- (#55) - Dead pixels outside the field were throwing off the thresholding algorithm and not detecting the field and/or BB.

6.28.57 V 1.3.1

- (#46) - Fixes CBCT analysis where there is a ring artifact outside the phantom. Incidentally, analysis is sped up by ~10%.

6.28.58 V 1.3.0

General Changes

- A new dependency has been added: scikit-image. Given that pylinac is largely an image processing library, this is actually overdue. Several extremely helpful functions exist that are made use of in both the new modules and will slowly be incorporated into the old modules as needed. The package is easily installed via pip (`pip install scikit-image`) or via conda (`conda install scikit-image`) if using the Anaconda distribution. Finally, if simply upgrading pylinac scikit-image will automatically install via pip. For the sake of installation speed I'd recommend conda.
- ROI sampling for CBCT and Leeds classes have been sped up ~10x, making analysis moderately to much faster.
- All user-interface dialog functions/methods have been deprecated. E.g. `PicketFence.from_UI()` is no longer a valid method. To retain similar functionality use Tk to open your own dialog box and then pass in the file name. Specifically, this applies to the VMAT, Starshot, PicketFence, MachineLog(s), FlatSym, and CBCT classes. The original goal of pylinac was to be used for a standalone desktop application. The assuranceqa.com web interface is the successor to that idea and does not need those UI methods.

Planar Imaging

- A new planar imaging class has been added: PipsProQC3. This class analyzes the PipsPro QC-3 MV imaging phantom. The class locates and analyzes low and high contrast ROIs.
- The Leeds phantom utilizes the scikit-image library to do a canny edge search to find the phantom. This will bring more stability for this class.

6.28.59 V 1.2.2

- (#45) Fixes various crashes of Leeds analysis.

6.28.60 V 1.2.1

- (#44) Fixed a stale wheel build causing `pip install` to install v1.1.

6.28.61 V 1.2.0

General Changes

- CatPhan 503 (Elekta) analysis is now supported.
- A new planar imaging module has been added for 2D phantom analysis; currently the Leeds TOR phantom is available.
- The `requests` package is no longer needed for downloading URLs; the urllib stdlib module is now used instead.
- Requirements were fixed in the docs and setup.py; a numpy function was being used that was introduced in v1.9 even though v1.8 was stated as the minimum; the new requirement is v1.9.
- Demonstration methods for the main classes have been fully converted to static methods. This means, for example, the following are equivalent: `CBCT().run_demo()` and `CBCT.run_demo()`.

Core Modules

- A tutorial on the use of the core modules is now available.
- A new `mask` core module was created for binary array operations.
- (#42) The Image classes now have a `gamma` method available.
- The Image classes' `median_filter()` method has been renamed to `filter()`, which allows for different types of filters to be passed in.
- The Image class can now load directly from a URL: `load_url()`.

CBCT

- CatPhan 503 (Elekta) is now supported. Usage is exactly the same except for the low-contrast module, which is not present in the 503.
- The low contrast measurements now use two background bubbles on either side of each contrast ROI. The default contrast threshold has been bumped to 15, which is still arbitrary but fits most eyeball values.

Starshot

- (#43) Keyword arguments can be passed to the init and class methods regarding the image info. For example, if a .tif file is loaded but the DPI is not in the image header it can be passed in like so:

```
star = Starshot("mystar.tif", dpi=100, sid=1000)
```

Planar Imaging

- 2D analysis of the Leeds TOR phantom is available. Tests low and high contrast. A new *Planar Imaging* doc page has been created.

Winston-Lutz

- A *save_summary()* method has been added for saving the plot to file.

6.28.62 V 1.1.1

- Winston-Lutz demo images were not included in the pypi package.

6.28.63 V 1.1.0

General Changes

- This release debuts the new Winston-Lutz module, which easily loads any number of EPID images, finds the field CAX and the BB, and can plot various metrics.

Log Analyzer

- Logs can now be anonymized using the `.anonymize()` method for both MachineLog and MachineLogs.
- The `.to_csv()` methods for MachineLog and MachineLogs returns a list of the newly created files.
- MachineLogs can now load from a zip archive using `.from_zip()`.

6.28.64 V 1.0.3

- Fixes #39. MachineLog fluence was inverted in the left-right direction.
- Fixes #40. MachineLog fluence calculations from dynalogs were dependent on the load order (A-file vs. B-file).

6.28.65 V 1.0.2

- Fixes #38. MachineLog fluence calculations would crash if there was no beam-on snapshots (e.g. kV images).

6.28.66 V 1.0.1

- Fixes #37. Reading in a trajectory log txt file with a blank line caused a crash.

6.28.67 V 1.0.0

General Changes

- This release debuts the new interactive plotting for certain figures. Quickly, matplotlib line/bar plots (althouth not yet images/arrays) can be plotted and saved in HTML using the MPLD3 library. This is less of interest to users doing interactive work, but this adds the ability to embed HTML plots in web pages.
- Several numpy array indexing calls were converted to ints from floats to avoid the new 1.9 numpy type-casting warnings. This also speeds up indexing calls slightly.

Picket Fence

- The analyzed image now has the option of showing a leaf error subplot beside the image. The image is aligned to the image such that the leaves align with the image.

Starshot

- Plotting the analyzed starshot image now shows both the zoomed-out image and a second, zoomed-in view of the wobble.
- Each subplot can be plotted and saved individually.

VMAT

- Plotting the analyzed image now shows the open and dmlc images and the segment outlines as well as a profile comparison between the two images. Each subplot can also be plotted and saved individually.
- `MLCS` is no longer a test option; `DRMLC` should be used instead.

6.28.68 V 0.9.1

- Fixed a bug with the log analyzer treatment type property.

6.28.69 V 0.9.0

General Changes

- This release has a few new features for the CBCT class, but is mostly an internal improvement. If you only use the main classes (CBCT, PicketFence, Starshot, etc), there should be no changes needed.

CBCT

- The CBCT analysis now examines low contrast ROIs and slice thickness.
- CBCT components have been renamed. E.g. the HU linearity attr has been renamed `hu` from `HU`.

Starshot

- Fixes #32 which was causing FWHM peaks on starshots to sometimes be erroneous for uint8/uint16 images.

PicketFence

- Adds #31, a method for loading multiple images into PicketFence.

Log Analyzer

- Fixes a bug which sometimes caused the parsing of the associated .txt log file for trajectory logs to crash.

6.28.70 V 0.8.2

- Fixed a bug with the picket fence overlay for left-right picket patterns.
- Plots for starshot, vmat, and picketfence now have a larger DPI, which should mean some more detail for saved images.

6.28.71 V 0.8.1

- Fixed an import bug

6.28.72 V 0.8.0

General Changes

- An upgrade for the robustness of the package. A LOT of test images were added for the Starshot, CBCT, PicketFence, and VMAT modules and numerous bugs were caught and fixed in the process.
- The debut of the "directory watcher". Run this script to tell pylinac to watch a directory; if a file with certain keywords is placed in the directory, pylinac will analyze the image and output the analyzed image and text file of results in the same directory.
- A generic troubleshooting section has been added to the documentation, and several modules have specific troubleshooting sections to help identify common errors and how to fix them.

VMAT

- Added a `from_zip()` and `load_zip()` method to load a set of images that are in a zip file.
- Added an `x_offset` parameter to `analyze()` to make shifting segments easier.

PicketFence

- Fixed #30, which wasn't catching errors on one side of the pickets, due to a signed error that should've been absolute.
- Two new parameters have been added to `analyze()`: `num_pickets` and `sag_adjustment`, which are somewhat self-explanatory. Consult the docs for more info.

Starshot

- Fixed #29, which was causing analysis to fail for images with a pin prick.

CBCT

- Fixed #28, which was applying the phantom roll adjustment the wrong direction.

6.28.73 V 0.7.1

General Changes

- Added `.from_url()` class method and `.load_url()` methods to most modules.

PicketFence

- Fixed #23, which was not properly detecting pickets for picket patterns that covered less than half the image.
- Fixed #24, which was failing analysis from small but very large noise. A small median filter is now applied to images upon loading.

6.28.74 V 0.7.0

General Changes

- The scipy dependency has been bumped to v0.15 to accommodate the new differential evolution function using in the Starshot module.

CBCT

- Whereas v0.6 attempted to fix an issue where if the phantom was not centered in the scan it would error out by adding a z-offset, v0.7 is a move away from this idea. If the offset given was not correct then analysis would error disgracefully. It is the point of automation to automatically detect things like where the phantom is in the dataset. Thus, v0.7 is a move towards this goal. Briefly, upon loading all the images are scanned and the HU linearity slice is searched for. Of the detected slices, the median value is taken. Other slices are known relative to this position.
- As per above, the z-offset idea is no longer used or allowed.
- Plots are now all shown in grayscale.
- If the phantom was not completely scanned (at least the 4 modules of analysis) analysis will now error out more gracefully.

6.28.75 V 0.6.0

General Changes

- Pylinac now has a wheel variation. Installation should thus be quicker for users with Python 3.4.
- Most main module classes now have a save method to save the image that is plotted by the plot method.

Class-based Constructors

- This release presents a normalized and new way of loading and initializing classes for the PicketFence, Starshot, VMAT and CBCT classes. Those classes all now accept the image path (folder path for CBCT) in the initialization method. Loading other types of data should be delegated to class-based constructors (e.g. to load a zip file into the CBCT class, one would use `cbct = CBCT.from_zip_file('zfiles.zip')`). This allows the user to both initialize and load the images/data in one step. Also prevents user from using methods before initialization (i.e. safer). See ReadTheDocs page for more info.

Dependencies

- Because the VMAT module was reworked and is now based on Varian specs, the pandas package will no longer be required. FutureWarnings have been removed.

CBCT

- Bug #18 is fixed. This bug did not account for slice thickness when determining the slice positions of the relevant slices.
- Bug #19 is fixed. This bug allowed the loading of images that did not belong to the same study. An error is now raised if such behavior is observed.
- Demo files are now read from the zipfile, rather than being extracted and then potentially cleaning up afterward. Behavior is now quicker and cleaner.
- Individual plots of certain module/slices can now be done. Additionally, the MTF can be plotted.
- The user can now adjust the relative position of the slice locations in the event the phantom is not set up to calibration conditions.

Log Analyzer

- Keys in the `txt` attr dict weren't stripped and could have trailing spaces. Keys are now stripped.

VMAT

- **Ability to offset the segments has been added.**
  Complete overhaul to conform to new Varian RapidArc QA specs. This includes the following:
- Rather than individual samples, 4 or 7 segments are created, 5x100mm each.
- Deviation is now calculated for each segment, based on the average segment value.
- The `DRMLC` test has changed name to `MLCS`. E.g. passing a test should be: `myvmat.analyze('mlcs')`, not `myvmat.analyze('drmlc')`; the latter will still work but raises a future warning.

Starshot

- Fixed a bug where an image that did not have pixels/mm information would error out.
- Added a tolerance parameter to the analyze method.

6.28.76 V 0.5.1

Log Analyzer

- Axis limits are now tightened to the data when plotting log_analyzer.Axis data.
- Gamma map plot luminescence is now normalized to 1 and a colorbar was added.
- Bug #14 fixed, where Tlogs v3 were not loading couch information properly.
- Trajectory log .txt files now also load along with the .bin file if one is around.

Starshot

- Multiple images can now be superimposed to form one image for analysis.

VMAT

- `load_demo_image()` parameter changed from `test_type` to `type`

6.28.77 V 0.5.0

- A new flatness & symmetry module allows for film and EPID image analysis.
- The `log_analyzer` module now supports writing trajectory logs to CSV.
- A FutureWarning that pandas will be a dependency in later versions if it's not installed.

6.28.78 V 0.4.1

- Batch processing of logs added via a new class.
- ~4x speedup of fluence calculations.

6.28.79 V 0.4.0

- A Varian MLC picket fence analysis module was added; this will analyze EPID PF images of any size and either orientation.

6.28.80 V 0.3.0

- Log Analyzer module added; this module reads Dynalogs and Trajectory logs from Varian linear accelerators.

Starshot

- The profile circle now aligns with the lines found.
- Recursive option added to analyze for recursive searching of a reasonable wobble.
- Image now has a cleaner interface and properties

6.28.81 V 0.2.1

- Demo files were not included when installed from pip

6.28.82 V 0.2.0

- Python 2.7 support dropped. Python 3 has a number of features that Python 2 does not, and because this project is just getting started, I didn't want to support Python 2, and then eventually drop it as Python 3 becomes more and more mainstream.
- Internal overhaul. Modules are now in the root folder. A core module with specialized submodules was created with a number of various tools.
- Demo files were assimilated into one directory with respective subdirectories.
- VMAT module can now handle HDMLC images.
- CBCT module was restructured and is much more reliable now.
- method names normalized, specifically the `return_results` method, which had different names in different modules.
- Lots of tests added; coverage increased dramatically.

6.28.83 V 0.1.3

Overall

A module for analyzing CBCT DICOM acquisitions of a CatPhan 504 (Varian) has been added. The starshot demo files have been compressed to zip files to save space. A value decorator was added for certain functions to enforce, e.g., ranges of values that are acceptable. The "Files" directory was moved outside the source directory. -Starshot now reports the diameter instead of radius

6.28.84 V 0.1.2

A PyPI setup.py bug was not properly installing pylinac nor including demo files. Both of these have been fixed.

6.28.85 V 0.1.1

Several small bugs were fixed and small optimizations made. A few methods were refactored for similarity between modules.

6.28.86 V 0.1.0

This is the initial release of Pylinac. It includes two modules for doing TG-142-related tasks: Starshot & VMAT QA

Versioning mostly follows standard semantic revisioning. However, each new module will result in a bump in minor release, while bug fixes will bump patch number.

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z