Pylinac provides TG-142 quality assurance (QA) tools to Python programmers as well as non-programmers in the field of therapy medical physics.
Pylinac contains high-level modules for automatically analyzing images and data generated by linear accelerators. Most scripts can be utilized with <10 lines of code. The built-in GUI is also straight-forward although programming experience may help identify errors.

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

The major features of the entire package include:

- 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 1

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 2

Installation

Install via pip:

```
$ pip install pylinac
```

See the Installation page for further details.
Below are the high-level tools currently available:

- **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:

```python
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)
```

(continues on next page)
# 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)

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

• 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:

```python
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
```

• **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:

```python
from pylinac import WinstonLutz
wl = WinstonLutz("wl/image/directory")  # images are analyzed upon loading
wl.plot_summary()
print(wl.results())
w1.publish_pdf('my_wl.pdf')
```

- **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:

```python
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
```
• **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:

```python
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
```

• **CT & CBCT QA** - The CBCT module automatically analyzes DICOM images of a CatPhan 504, 503, or 600 acquired when doing CT or CBCT quality assurance. It can load a folder or zip file that the images are in and automatically correct for phantom setup in 6 axes. It can 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.

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:

```python
from pylinac import CatPhan504, CatPhan503, CatPhan600, CatPhan604

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')
```

• **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:

```python
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
# dosage of a 20 cm treatment
# analysis allows you to use
# 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
# publish a PDF report
# publish_pdf()
dlog = load_log("dynalog.dlg")
```

- **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:
```python
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()
```

- **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:

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

Have questions? Ask them on the pylinac discussion forum.
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.

5.1 Pylinac General Overview
5.1.1 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 linac QA. These data usually follow tests outlined by AAPM TG-142. 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 to compute meaningful output related to common physics tests
- Pylinac consumes raw data to present meaningful data types for further exploration
5.1.2 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.

5.1.3 Intended Audience

Pylinac is intended to be used by physicists who know at least a bit of programming.

5.1.4 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.

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

5.1.5 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.
5.1.6 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. Each module gives a starting script in its documentation.

5.2 Installation

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

5.2.1 I know Python already

Great! To get started install via pip:

```bash
$ pip install pylinac
```

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

**Note:** Although pip should work, the dependencies can sometimes give problems. A workaround is to install the dependencies via conda, then install pylinac via pip since pylinac itself is pure python.

5.2.2 Dependencies

Pylinac, as a scientific package, has fairly standard scientific dependencies (>= means at least that version or newer):

```bash
numpy >= 1.16
scipy >= 1.1
pydicom >= 2.0
matplotlib >= 2.0
scikit-image >= 0.17
Pillow >= 4.0
tqdm >= 3.8
reportlab >= 3.3
argue
```

(continues on next page)
pylinac Documentation, Release 3.1.0

(continued from previous page)

5.2.3 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: Minconda, 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.
5.3 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.

5.3.1 Running a Demo

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

```python
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:

```python
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

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.

5.3.2 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:

```python
from pylinac import TrajectoryLog

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

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

```python
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:

```python
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.

5.3.3 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’:

```python
import pylinac
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:

```python
import matplotlib.pyplot as plt
import pylinac
pylinac.settings.ARRAY_COLORMAP = plt.cm.viridis
pylinac.TrajectoryLog.run_demo()
```
5.3. General Tips
5.4 Calibration (TG-51/TRS-398)

5.4.1 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.

5.4.2 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.

5.4.3 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.

5.4.4 TG-51
Equation Definitions

Equation definitions are as follows:

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

\[
\frac{273.2 + T}{273.2 + 22} \times \frac{101.33}{P}
\]

**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 result 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:

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

- **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:

\[
0.9905 + 0.0710e^{-\frac{R_{50}}{R_{ref}}}
\]

- **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:

\[
\begin{cases}
PDD(10) & energy < 10 \\
1.267 * PDD(10) - 20.0 & 75 \leq PDD(10) \leq 89 \\
PDD(10)_{Pb} & lead@50cm, PDD(10)_{Pb} < 73 \\
0.8905 + 0.00150 * PDD(10)_{Pb} * PDD(10)_{Pb} & lead@50cm, PDD(10)_{Pb} \geq 73 \\
PDD(10)_{Pb} & lead@30cm, PDD(10)_{Pb} < 71 \\
0.8116 + 0.00264 * PDD(10)_{Pb} * PDD(10)_{Pb} & lead@30cm, PDD(10)_{Pb} \geq 71
\end{cases}
\]
• M-corrected (corrected chamber reading) - TG-51 Eqn. 8:

\[ P_{ion} \times P_{TP} \times P_{elec} \times P_{pol} \times M_{raw} \]

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

\[ \left\{ \begin{array}{ll}
A + B \times 10^{-3} \times PDD(10)x + C \times 10^{-5} \times (PDD(10)x)^2 & 63 < PDD(10)x < 86
\end{array} \right. \]

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 solely 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:

\[ k_Q = k_{Q,ecal} \times k'_Q \]

where Eqn. 11 states:

\[ k'_Q = a + b \times R_{50}^{-c} \]

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.

• \( D_Q^{\text{w}} \) photon (Dose to water at 10cm from a photon beam of quality Q - TG-51 Eqn. 3):

\[ M \times k_Q \times N^{60Co}_{D,w}(Gy) \]

• \( D_Q^{\text{w}} \) electron (Dose to water at 10cm from an electron beam of quality Q - TG-51 Eqn. 6):

\[ M \times P_{gr}^{Q} \times k'_{R_{50}} \times k_{ecal} \times N^{60Co}_{D,w}(Gy) \]

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.

```python
# 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)
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"

```python
from pylinac.calibration import tg51

ENERGY = 6
TEMP = 22.1
```

---

5.4. Calibration (TG-51/TRS-398) 29
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)

### 5.4.5 TRS-398

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

**Equation Definitions**

- \( K_{tp} \) (Temp/Pressure correction):

\[
\frac{273.2 + T}{273.2 + 22} \times \frac{101.33}{P}
\]

**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):

\[
a_0 + a_1 * \left(\frac{M_1}{M_2}\right) + a_2 * \left(\frac{M_1}{M_2}\right)^2
\]

• 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:

\[
\begin{align*}
1.029 * I_{50} & - 0.06(cm) & 2 \leq I_{50} \leq 10 \\
1.059 * I_{50} & - 0.37(cm) & I_{50} < 10
\end{align*}
\]

• \(D_{w,Q}^{(\text{photon})}\) (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):

\[
M_Q = k_s * k_{TP} * K_{elec} * K_{pol} * M_1
\]

• \(k_{Q,Qo}\) for Photons (cylindrical chamber-specific quality conversion factor): TRS-398 Table 6.III

• \(k_{Q}\) for Electrons (cylindrical chamber-specific quality conversion factor; calibrated in Co-60): TRS-398 Table 7.III

Function-based Use

```python
# 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))
```

(continues on next page)
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 = trs398TRS398Photon(
    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!
```python
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!'], open_file=False)
```

### 5.4.6 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 pg 62 and Followill et al 1998 eqn 1.

**pylinac.calibration.tg51.p_tp** *(*, temp: Union[int, float], press: Union[int, 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: Union[int, float, list, tuple, numpy.ndarray], m_opposite: Union[int, float, list, tuple, numpy.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** BoundError if calculated Ppol is >1% from 1.0.

**pylinac.calibration.tg51.p_ion** *(*, voltage_reference: int, voltage_reduced: int, m_reference: Union[int, float, list, tuple, numpy.ndarray], m_reduced: Union[int, float, list, tuple, numpy.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.

def d_ref(*, i_50: float) -> float
    Calculate the d_ref of an electron beam based on the I50 depth.

Parameters

- `i_50 (float)` – The value of I50 in cm.

def 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.

def kp_r50(*, r_50: float) -> float
    Calculate k’R50 for Farmer-like chambers.

Parameters

- `r_50 (float (2-9))` – The R50 value in cm.

def pq_gr(*, m_dref_plus: Union[int, float, list, tuple, numpy.ndarray], m_dref: Union[int, float, list, tuple, numpy.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.

def m_corrected(*, p_ion: float, p_tp: float, p_elec: float, p_pol: float, m_reference: Union[int, float, list, tuple, numpy.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

Return type `float`

def pddx(*, pdd: float, energy: int, lead_foil: Optional[str] = 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.

Bases: pylinac.calibration.tg51.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).

**pddx**

The photon-only PDD(10) value.

**kq**

The chamber-specific beam quality correction factor.

**dose_mu_10**

cGy/MU at a depth of 10cm.

**dose_mu_dmax**

cGy/MU at a depth of dmax.

**dose_mu_10_adjusted**

The dose/mu at 10cm depth after adjustment.

**dose_mu_dmax_adjusted**

The dose/mu at dmax depth after adjustment.


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: Union[int, float], press: Union[int, 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: Union[int, float, list, tuple, numpy.ndarray], m_opposite: Union[int, float, list, tuple, numpy.ndarray], m_reduced: Union[int, float, list, tuple, numpy.ndarray], m_gradient: Union[int, float, list, tuple, numpy.ndarray], cone: str, mu: int, i_50: float, tissue_correction: float = 1.0, m_reference_adjusted=None)

Bases: pylinac.calibration.tg51.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).

**r_50**
Depth of the 50% dose value.

**dref**
Depth of the reference point.

**pq_gr**
Gradient factor

**kq**
The kQ value using classic TG-51

**dose_mu_dref**
cGy/MU at the depth of Dref.

**dose_mu_dmax**
cGy/MU at the depth of dmax.

**dose_mu_dref_adjusted**
cGy/MU at the depth of Dref.

**dose_mu_dmax_adjusted**
cGy/MU at the depth of dmax.

**publish_pdf** (*filename: str, notes: Optional[list] = None, open_file: bool = False, metadata: Optional[dict] = 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.

Bases: pylinac.calibration.tg51.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.
- **k_elec (float)** – The electrometer correction value given by the calibration laboratory.
- **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).

\[ r_{50} \]
Depth of the 50% dose value.

\[ dref \]
Depth of the reference point.

\[ kq \]
The kQ value using the updated Muir & Rogers values from their 2014 paper, equation 11, or classically if kecal is passed.

\[ \text{dose}_\text{mu}_\text{dref} \]
cGy/MU at the depth of Dref.

\[ \text{dose}_\text{mu}_\text{dmax} \]
cGy/MU at the depth of dmax.

\[ \text{dose}_\text{mu}_\text{dref}_\text{adjusted} \]
cGy/MU at the depth of Dref.

\[ \text{dose}_\text{mu}_\text{dmax}_\text{adjusted} \]
cGy/MU at the depth of dmax.

**publish_pdf** *(filename: str, notes: Optional[list] = None, open_file: bool = False, metadata: Optional[dict] = 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.

### 5.4.7 TRS-398 API Documentation

**pylinac.calibration.trs398.k_s** *(*, voltage_reference: int, voltage_reduced: int, m_reference: Union[int, float, list, tuple, numpy.ndarray], m_reduced: Union[int, float, list, tuple, numpy.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* – The ion recombination factor.

Return type *float*

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* – The calculated kQ given table 6.III

Return type *float*

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* – The calculated kQ from Table 7.III

Return type *float*

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: *array, float*, 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` – The fully corrected chamber reading.

**Return type**  
`float`


Bases: pylinac.calibration.trs398.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.

$k_q$

kQ of the chamber and TPR.

dose_mu_zmax

cGy/MU at a depth of zmax.

dose_mu_zmax_adjusted

The dose/mu at dmax depth after adjustment.


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 = "," physici"
• **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.

**r_50**  
The depth of R50 in cm, derived from I50.

**zref**  
Depth of the reference point.

**kq**  
kQ given the chamber and R50.

**dose_mu_zmax**  
cGy/MU at a depth of zmax.

**dose_mu_zmax_adjusted**  
The dose/mu at dmax depth after adjustment.

**publish_pdf** *(filename: str, notes: Optional[list] = None, open_file: bool = False, metadata: Optional[dict] = 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.

### 5.5 Starshot

#### 5.5.1 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.

#### 5.5.2 Running the Demo

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

```python
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

#### 5.5.3 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.
5.5.4 Typical Use

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

```python
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:

  ```python
  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:

  ```python
  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:

  ```python
  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:

  ```python
  # 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.

  ```python
  # return a dataclass with introspection
  data = mystar.results_data()
  data.tolerance_mm
  data.passed
  ...
  
  # return as a dict
  data_dict = mystar.results_data(as_dict=True)
  data_dict['passed']
  ...
  ```

  Each subplot can be plotted independently as well:

  ```python
  # 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:

  ```python
  mystar.save_analyzed_image('mystar.png')
  ```

  You may also save to PDF:
5.5.5 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.
5.5.6 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.

5.5.7 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:

```python
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()
print(star.results())
star.plot_analyzed_image()
```

with an output of:

```text
Result: PASS
The minimum circle that touches all the star lines has a diameter of 0.045 mm.
```

(continues on next page)
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:

```
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import GaussianFilterLayer, FilteredFieldLayer,
                      AS1200Image, RandomNoiseLayer
                      
# analyze it
star = pylinac.Starshot(star_path)
star.analyze(radius=0.6)  # radius changed
print(star.results())
star.plot_analyzed_image()
```

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.
Analyzed Image

Wobble Circle

5.5. Starshot
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.
5.5.8 API Documentation

class pylinac.starshot.Starshot (filepath: Union[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.

    image
        Type Image
    circle_profile
        Type StarProfile
    lines
        Type LineManager
    wobble
        Type Wobble
    tolerance
        Type 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, **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.
- `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: Union[pylinac.core.geometry.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.

passed

Boolean specifying whether the determined wobble was within tolerance.

results () → str

Return the results of the analysis.

Returns A string with a statement of the minimum circle.

Return type string

results_data (as_dict: bool = False) → Union[pylinac.starshot.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)

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

Parameters show (bool) – Whether to actually show the image.

plot_analyzed_subimage (subimage: str = 'wobble', ax: Optional[matplotlib.axes._axes.Axes] = None, show: bool = True)

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.

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 extension, 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: Union[str, IO], notes: Union[str, List[str], None] = None, open_file: bool = False, metadata: Optional[dict] = 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 ————–

```python
static run_demo()

Demonstrate the Starshot module using the demo image.
```

```python
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: `pylinac.core.utilities.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 = None
circle_diameter_mm = None
circle_radius_mm = None
passed = None
circle_center_x_y = None
```

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

Bases: `pylinac.core.profile.CollapsedCircleProfile`

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

```
def get_peaks(min_peak_height, min_peak_distance=0.02, fwhm=True)
```

Determine the peaks of the profile.

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

Bases: `pylinac.core.geometry.Circle`

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

```
radius_mm
```

Type  The radius of the Circle in mm.

```
diameter_mm
```

Diameter of the wobble in mm.

```python
class pylinac.starshot.LineManager(points: List[pylinac.core.geometry.Point])
```

Bases: `object`

Manages the radiation lines found.

Parameters **points** – The peak points found by the StarProfile

```
def construct_rad_lines(points: List[pylinac.core.geometry.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** – A list of Lines (radiation lines) found.
Return type  list

See also:

Starshot.analyze()

core.profile.CircleProfile.find_FWXM_peaks()  min_peak_distance parameter info.
geometry.Line()  returning object

match_points(points: List[pylinac.core.geometry.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: matplotlib.axes._axes.Axes)

Plot the lines to the axis.

5.6 VMAT

5.6.1 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.

5.6.2 Running the Demos

For this example we will use the DRGS class:

```python
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%
```
5.6.3 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.

5.6.4 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:

  ```python
  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:

```python
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%:

  ```python
  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:

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

PDF reports can also be generated:

```python
myvmat.publish_pdf('drgs.pdf')
```
Chapter 5. Contributing
5.6.5 Customizing the analysis

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

To change the segment size:

```python
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, the class property must be changed. This is different behavior because by default the x-positions are different for the DRGS and DRMLC test:

```python
from pylinac import DRGS, DRMLC

DRGS.SEGMENT_X_POSITIONS_MM = (-100, -80, ...)
# proceed as normal
my_drgs = DRGS(...)

# DRMLC must be addressed separately
DRMLC.SEGMENT_X_POSITIONS_MM = (-40, -10, 10, 40)
my_drmlc = DRMLC(...)
```

5.6.6 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:

```python
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']
```

5.6.7 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:

\[
M_{\text{corr}}(x) = \frac{M_{\text{DRGS}}(x)}{M_{\text{open}}(x)} \times 100
\]

The reading deviation of each segment is calculated as:

\[
M_{\text{deviation}}(x) = \frac{M_{\text{corr}}(x)}{M_{\text{corr}}} \times 100 - 100
\]

where \(M_{\text{corr}}\) 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** – Segment positions are always the same within the image. The x-positions are based on 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.

### 5.6.8 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.

The script will generate the files, but you can also download them here: perfect_open_drmlc.dcm
perfect_dmlc_drmlc.dcm

```python
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%
```

Noisy, Realistic

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

```python
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
```

(continues on next page)
Chapter 5. Contributing
```python
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:

<table>
<thead>
<tr>
<th>Test Results (Tol. +/-1.5%)</th>
<th>Dose Rate &amp; MLC Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>0.0332%</td>
</tr>
<tr>
<td>Absolute Mean Deviation</td>
<td>0.0257%</td>
</tr>
</tbody>
</table>

5.6. VMAT
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.

```python
import random
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),
    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%
```

5.6.9 API Documentation

Main classes

These are the classes a typical user may interface with.

```python
class pylinac.vmat.DRGS (image_paths: Sequence[Union[str, BinaryIO]])
    Bases: pylinac.vmat.VMATBase
```

Class representing a Dose-Rate, Gantry-speed VMAT test. Will accept, analyze, and return the results.
5.6. VMAT
Parameters **image_paths** *(iterable (list, tuple, etc))* – A sequence of paths to the image files.

**static run_demo()**

Run the demo for the Dose Rate & Gantry Speed test.

**analyze**(tolerance: Union[float, int] = 1.5, segment_size_mm: Tuple = (5, 100))

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.

**avg_abs_r_deviation**

Return the average of the absolute R_deviation values.

**avg_r_deviation**

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

**classmethod from_demo_images()**

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)

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.

**max_r_deviation**

Return the value of the maximum R_deviation segment.

**plot_analyzed_image**(show: bool = True)

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.

**publish_pdf**(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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 ————–

**r_devs**

Return the deviations of all segments as an array.
results() → str
   A string of the summary of the analysis results.

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

   Return type  str

results_data(as_dict=False) → Union[pylinac.vmat.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[Union[str, BinaryIO]])
Bases: pylinac.vmat.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.

static run_demo()
   Run the demo for the MLC leaf speed test.

analyze(tolerance: Union[float, int] = 1.5, segment_size_mm: Tuple = (5, 100))
   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.

avg_abs_r_deviation
   Return the average of the absolute R_deviation values.

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

classmethod from_demo_images()
   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)
   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.

max_r_deviation
   Return the value of the maximum R_deviation segment.

plot_analyzed_image(show: bool = True)
   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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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 ————–

**r_devs**
Return the deviations of all segments as an array.

**results() → str**
A string of the summary of the analysis results.

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

**Return type** str

**results_data**(as_dict=False) → Union[pylinac.vmat.VMATResult, dict]
Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

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

**Bases**: pylinac.core.utilities.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** = None

**tolerance_percent** = None

**max_deviation_percent** = None

**abs_mean_deviation** = None

**passed** = None

**segment_data** = None

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

**Bases**: object

An individual segment/ROI result

**passed** = None

**x_position_mm** = None

**r_corr** = None

**r_dev** = None

**center_x_y** = None

**Supporting Classes**

You generally won’t have to interface with these unless you’re doing advanced behavior.
class pylinac.vmat.VMATBase(image_paths: Sequence[Union[str, BinaryIO]])

Bases: object

Parameters

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

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)

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.

classmethod from_demo_images()

Construct a VMAT instance using the demo images.

analyze(tolerance: Union[float, int] = 1.5, segment_size_mm: Tuple = (5, 100))

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.

results() → str

A string of the summary of the analysis results.

Returns

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

Return type

str

results_data(as_dict=False) → Union[pylinac.vmat.VMATResult, dict]

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

r_devs

Return the deviations of all segments as an array.

avg_abs_r_deviation

Return the average of the absolute R_deviation values.

avg_r_deviation

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

max_r_deviation

Return the value of the maximum R_deviation segment.

plot_analyzed_image(show: bool = True)

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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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 ————–


Bases: pylinac.core.geometry.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.

**r_dev**

The reading deviation (R_dev) from the average readings of all the segments. See RTD for equation info.

*Type* float

**r_corr**

The corrected reading (R_corr) of the pixel values. See RTD for explanation and equation info.

*Type* float

**passed**

Specifies where the segment reading deviation was under tolerance.

*Type* boolean

**r_corr**

Return the ratio of the mean pixel values of DMLC/OPEN images.

**bl_corner**

The location of the bottom left corner.

**br_corner**

The location of the bottom right corner.

**passed**

Return whether the segment passed or failed.

**plot2axes**(axes: matplotlib.axes.Axes, edgecolor: str = 'black', angle: float = 0.0, fill: bool = False, alpha: float = 1, facecolor: str = 'g', label=None)

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.

**tl_corner**

The location of the top left corner.

**tr_corner**

The location of the top right corner.
get_bg_color() → str
Get the background color of the segment when plotted, based on the pass/fail status.

5.7 CatPhan/CT

5.7.1 Overview

The CatPhan module automatically analyzes DICOM images of a CatPhan 504, 503, or 600 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.

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.

5.7.2 Running the Demo

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

```python
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 -
    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:

```python
cbct.plot_analyzed_subimage('rmtf')
```

Or generate a PDF report:
5.7. CatPhan/CT

HU linearity

<table>
<thead>
<tr>
<th>Nominal Values</th>
<th>HU Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>750</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

HU Delta values range from -40 to 40.
5.7.3 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.

```python
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:

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

  or load a zip file of the images:

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

You can also use the demo images provided:

```python
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:

  ```python
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:

  ```python
  # 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.
  ```

5.7.4 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:

```python
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:

```python
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):

```python
cr = ...  # load a dataset like normal
cr.analyze()
cr.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
    }

c = 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 nominal HU values 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, 'distance': roi_dist_mm, 'radius': roi_radius_mm},  # changed 'angle' from -90
        'PMP': {'value': -196, 'angle': -120, '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
c = CustomCP504(...)
5.7.5 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.

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):

  Nominal material formulation and specific gravity

<table>
<thead>
<tr>
<th>Material</th>
<th>Formula</th>
<th>$Z_{eff}$</th>
<th>Specific Gravity</th>
<th>HU range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air</td>
<td>.78N, .21O, .01Ar</td>
<td>8.00</td>
<td>0.00</td>
<td>(-1046, -988)</td>
</tr>
<tr>
<td>PMP</td>
<td>[C$<em>6$H$</em>{12}$(CH$_2$)]</td>
<td>5.44</td>
<td>0.83</td>
<td>(-220, -172)</td>
</tr>
<tr>
<td>LDPE</td>
<td>[C$_2$H$_4$]</td>
<td>5.44</td>
<td>0.92</td>
<td>(-121, -87)</td>
</tr>
<tr>
<td>Polystyrene</td>
<td>[C$_8$H$_8$]</td>
<td>5.70</td>
<td>1.03</td>
<td>(-65, -29)</td>
</tr>
<tr>
<td>Acrylic</td>
<td>[C$_5$H$_9$O$_2$]</td>
<td>6.47</td>
<td>1.18</td>
<td>(92, 137)</td>
</tr>
<tr>
<td>Bone 20%</td>
<td>.61C, .06Ca, .06H,</td>
<td>9.09</td>
<td>1.14</td>
<td>(211, 263)</td>
</tr>
<tr>
<td></td>
<td>.06N, .30O, .03P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dehrin®</td>
<td>Proprietary</td>
<td>6.95</td>
<td>1.42</td>
<td>(344, 387)</td>
</tr>
<tr>
<td>Bone 50%</td>
<td>.35C, .14Ca, .04H,</td>
<td>11.46</td>
<td>1.40</td>
<td>(667, 783)</td>
</tr>
<tr>
<td></td>
<td>.06N, .34O, .06P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teflon®</td>
<td>[CP$_2$]</td>
<td>8.43</td>
<td>2.16</td>
<td>(941, 1060)</td>
</tr>
</tbody>
</table>

- **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.

- **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. Based on testing, the FWHM is not always perfectly detected and may not “catch” the profile, giving an undervalued representation. Thus, the two longest profiles are averaged and the value is converted from pixels to mm and multiplied by 0.42.

**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.

**5.7.6 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.

5.7.7 API Documentation

Main classes

These are the classes a typical user may interface with.

class pylinac.ct.CatPhan504 (folderpath: Union[str, Sequence[T_co]], check_uid: bool = True)
Bases: pylinac.ct.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 or list of strings) – 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.

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.


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*) – Deprecated since version 3.0: Use visibility parameter instead.

  The threshold for “detecting” low-contrast image. See RTD for calculation info.

- **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.

**catphan_size**

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** The middle slice of the HU linearity module.

**Return type** *int*

**find_phantom_roll** () → *float*

Determine the “roll” of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

**Returns** *float*

**Return type** 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: Union[str, zipfile.ZipFile, BinaryIO], check_uid: bool = True*)

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.

**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

**mm_per_pixel**
The millimeters per pixel of the DICOM images.

**num_images**
The number of images loaded.

**plot_analyzed_image**(show: bool = True) → None
Plot the images used in the calculate and summary data.

**Parameters**

• **show**(bool) – Whether to plot the image or not.

**plot_analyzed_subimage**(subimage: str = ‘hu’, delta: bool = True, show: bool = True) → None
Plot a specific component of the CBCT analysis.

**Parameters**

• **subimage**(‘hu’, ‘un’, ‘sp’, ‘lc’, ‘mtf’, ‘lin’, ‘prof’) – 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.
  – **mtf** draws the RMTF plot.
  – **lin** draws the HU linearity values. Used with **delta**.
  – **prof** draws the HU uniformity profiles.
• **delta**(bool) – Only for use with **lin**. Whether to plot the HU delta or actual values.
• **show**(bool) – Whether to actually show the plot.

**publish_pdf**(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[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 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 ————–

**results**() → str
Return the results of the analysis as a string. Use with print().
Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.

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 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.

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 or list of strings) – 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.

Raises

- NotADirectoryError – If folder str passed is not a valid directory.
- FileNotFoundError – If no CT images are found in the folder

Run the CBCT demo using high-quality head protocol images.

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*) – Deprecated since version 3.0: Use visibility parameter instead.

  The threshold for “detecting” low-contrast image. See RTD for calculation info.

• **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*.

**catphan_size**

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** The middle slice of the HU linearity module.

  **Return type** *int*

**find_phantom_roll()** → *float*

Determine the “roll” of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

  **Returns** *float*

  **Return type** 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: Union[str, zipfile.ZipFile, BinaryIO], check_uid: bool = True*)

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.

  **Raises**

  • FileExistsError : If zip_file passed was not a legitimate zip file.
• FileNotFoundError: If no CT images are found in the folder

```python
localize() → None
```

Find the slice number of the catphan’s HU linearity module and roll angle

```python
mm_per_pixel
```

The millimeters per pixel of the DICOM images.

```python
num_images
```

The number of images loaded.

```python
plot_analyzed_image(show: bool = True) → None
```

Plot the images used in the calculate and summary data.

**Parameters**

- `show (bool)`: Whether to plot the image or not.

```python
plot_analyzed_subimage(subimage: str = 'hu', delta: bool = True, show: bool = True) → None
```

Plot a specific component of the CBCT analysis.

**Parameters**

- `subimage (str)`: 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.
  - `mtf`: draws the RMTF plot.
  - `lin`: draws the HU linearity values. Used with `delta`.
  - `prof`: draws the HU uniformity profiles.
- `delta (bool)`: Only for use with `lin`. Whether to plot the HU delta or actual values.
- `show (bool)`: Whether to actually show the plot.

```python
publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[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 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 ————–

```python
results() → str
```

Return the results of the analysis as a string. Use with `print()`.

```python
results_data(as_dict=False) → Union[pylinac.ct.CatphanResult, dict]
```

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

```python
save_analyzed_image(filename: str, **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: Union[str, BinaryIO], subimage: str = 'hu', **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.

**class** `pylinac.ct.CatPhan600` *(folderpath: Union[str, Sequence[T_co]], check_uid: bool = True)*

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 or list of strings)* – 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.

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.


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)* – Deprecated since version 3.0: Use visibility parameter instead.
The threshold for “detecting” low-contrast image. See RTD for calculation info.

- **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*.

**catphan_size**
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** The middle slice of the HU linearity module.

**Return type** int

**find_phantom_roll () → float**
Determine the “roll” of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

**Returns** float

**Return type** 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: Union[str, zipfile.ZipFile, BinaryIO], check_uid: bool = True)**
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.

**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
**mm_per_pixel**
The millimeters per pixel of the DICOM images.

**num_images**
The number of images loaded.

**plot_analyzed_image** *(show: bool = True) → None*
Plot the images used in the calculate and summary data.

**Parameters**
- **show** *(bool)* – Whether to plot the image or not.

**plot_analyzed_subimage** *(subimage: str = 'hu', delta: bool = True, show: bool = True) → None*
Plot a specific component of the CBCT analysis.

**Parameters**
- **subimage** *({'hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof'})* – 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.
  - **mtf** draws the RMTF plot.
  - **lin** draws the HU linearity values. Used with delta.
  - **prof** draws the HU uniformity profiles.
- **delta** *(bool)* – Only for use with lin. Whether to plot the HU delta or actual values.
- **show** *(bool)* – Whether to actually show the plot.

**publish_pdf** *(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[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 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 ————–

**results() → str**
Return the results of the analysis as a string. Use with print().

**results_data** *(as_dict=True) → Union[pylinac.ct.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, **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: Union[str, BinaryIO], subimage: str = 'hu', **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.

class pylinac.ct.CatPhan604(folderpath: Union[str, Sequence[T_co]], check_uid: bool = True)
    Bases: pylinac.ct.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 or list of strings) – 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.

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.

    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) – Deprecated since version 3.0: Use visibility parameter instead.

The threshold for “detecting” low-contrast image. See RTD for calculation info.
- **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*.

**catphan_size**

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** The middle slice of the HU linearity module.

**Return type** int

**find_phantom_roll()** → float

Determine the “roll” of the phantom.

This algorithm uses the two air bubbles in the HU slice and the resulting angle between them.

**Returns** float

**Return type** 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: Union[str, zipfile.ZipFile, BinaryIO], check_uid: bool = True)**

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.

**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

**mm_per_pixel**

The millimeters per pixel of the DICOM images.

**num_images**

The number of images loaded.
plot_analyzed_image (show: bool = True) → None
Plot the images used in the calculate and summary data.

Parameters
- **show** (bool) – Whether to plot the image or not.

plot_analyzed_subimage (subimage: str = 'hu', delta: bool = True, show: bool = True) → None
Plot a specific component of the CBCT analysis.

Parameters
- **subimage** ('hu', 'un', 'sp', 'lc', 'mtf', 'lin', 'prof') – 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.
  - mtf draws the RMTF plot.
  - lin draws the HU linearity values. Used with delta.
  - prof draws the HU uniformity profiles.
- **delta** (bool) – Only for use with lin. Whether to plot the HU delta or actual values.
- **show** (bool) – Whether to actually show the plot.

publish_pdf (filename: str, notes: str = None, open_file: bool = False, metadata: Optional[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 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 ————–

results () → str
Return the results of the analysis as a string. Use with print().

results_data (as_dict=False) → Union[pylinac.ct.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, **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: Union[str, BinaryIO], subimage: str = 'hu', **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.


    Bases: pylinac.core.utilities.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 = None
    catphan_roll_deg = None
    origin_slice = None
    num_images = None
    ctp404 = None
    ctp486 = None
    ctp528 = None
    ctp515 = 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[str, pylinac.ct.ROIResult])

    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 = None
    low_contrast_visibility = None
    thickness_passed = None
    measured_slice_thickness_mm = None
    thickness_num_slices_combined = None
    geometry_passed = None
    avg_line_distance_mm = None
    line_distances_mm = None
    hu_linearity_passed = None
    hu_tolerance = None
    hu_rois = None
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 = None
mtf_lp_mm = None
roi_settings = None

class pylinac.ct.CTP515Result (cnr_threshold: float, num_rois_seen: int, 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.

cnr_threshold = None
num_rois_seen = None
roi_settings = None

class pylinac.ct.CTP486Result (uniformity_index: float, integral_non_uniformity: float, 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.

uniformity_index = None
integral_non_uniformity = None
passed = None

class pylinac.ct.ROIResult (name: str, value: float, difference: float, nominal_value: float, 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.

name = None
value = None
difference = None
nominal_value = None
passed = None

Module classes (CTP404, etc)

class pylinac.ct.Slice (catphan, slice_num: Optional[int] = None, combine: bool = True, combine_method: str = 'mean', num_slices: int = 0)
Bases: object
Base class for analyzing specific slices of a CBCT dicom set.

**Parameters**

- `catphan (~pylinac.cbct.CatPhanBase 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` (from `{'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.

**phan_center**

Determine the location of the center 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

**Raises** `ValueError` – If any of the above conditions are not met.

class pylinac.ct.CatPhanModule (catphan, tolerance: Optional[float], offset: int = 0)

Bases: `pylinac.ct.Slice`

Base class for a CTP module.

roi_dist_mm

alias of `builtins.float`

roi_radius_mm

alias of `builtins.float`

preprocess (catphan)

A preprocessing step before analyzing the CTP module.

**Parameters** `catphan (~pylinac.cbct.CatPhanBase instance.)` –

slice_num

The slice number of the spatial resolution module.

**Returns**

**Return type** `float`

plot_rois (axis: matplotlib.axes._axes.Axes) → None

Plot the ROIs to the axis.

class pylinac.ct.CTP404CP503 (catphan, offset: int, hu_tolerance: float, thickness_tolerance: float, scaling_tolerance: float)

Bases: `pylinac.ct.CTP404CP504`

Alias for namespace consistency

**Parameters**

- `catphan (~pylinac.cbct.CatPhanBase instance.)` –
- `offset (int)` –
- `hu_tolerance (float)` –
• **thickness_tolerance** *(float)* –
• **scaling_tolerance** *(float)* –

class **pylinac.ct.CTP404CP504** *(catphan, offset: int, hu_tolerance: float, thickness_tolerance: float, scaling_tolerance: float)*

**Bases:** **pylinac.ct.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)* –

**preprocess** *(catphan)* → None

A preprocessing step before analyzing the CTP module.

**Parameters**

- **catphan** *(~pylinac.cbct.CatPhanBase instance.)* –

**lcv**

The low-contrast visibility

**plot_linearity** *(axis: Optional[matplotlib.axes._axes.Axes] = 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).

**passed_hu**

Boolean specifying whether all the ROIs passed within tolerance.

**plot_rois** *(axis: matplotlib.axes._axes.Axes)* → None

Plot the ROIs onto the image, as well as the background ROIs

**passed_thickness**

Whether the slice thickness was within tolerance from nominal.

**meas_slice_thickness**

The average slice thickness for the 4 wire measurements in mm.

**passed_geometry**

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)*

**Bases:** **pylinac.ct.CTP404CP504**

**Parameters**

• **catphan** *(~pylinac.cbct.CatPhanBase instance.)* –
• **offset** *(int)* –

5.7. CatPhan/CT
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.

**radius2linepairs_mm**

The radius in mm to the line pairs.

**mtf**

The Relative MTF of the line pairs, normalized to the first region.

**plot_rois** *(axis: matplotlib.axes._axes.Axes) → None*

Plot the circles where the profile was taken within.

**circle_profile**

Calculate the median profile of the Line Pair region.

**plot_mtf** *(axis: Optional[matplotlib.axes._axes.Axes] = None) → Tuple*

Plot the Relative MTF.

**Parameters**

- **radius2linepairs_mm** *(float)*
- **mtf** *(float)*
class pylinac.ct.CTP528CP604 (catphan, tolerance: Optional[float], offset: int = 0)
Bases: pylinac.ct.CTP528CP504
Alias for namespace consistency.

Bases: pylinac.ct.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”.

rois_visible
The number of ROIs “visible”.

lower_window
Lower bound of CT window/leveling to show on the plotted image. Improves apparent contrast.

upper_window
Upper bound of CT window/leveling to show on the plotted image. Improves apparent contrast

class pylinac.ct.CTP486 (catphan, tolerance: Optional[float], offset: int = 0)
Bases: pylinac.ct.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.

plot_profiles (axis: matplotlib.axes._axes.Axes = 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.

overall_passed
Boolean specifying whether all the ROIs passed within tolerance.

uniformity_index
The Uniformity Index

integral_non_uniformity
The Integral Non-Uniformity

ROI Objects

Bases: pylinac.core.roi.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.
value_diff
The difference in HU between measured and nominal.

passed
Boolean specifying if ROI pixel value was within tolerance of the nominal value.

plot_color
Return one of two colors depending on if ROI passed.

class pylinac.ct.ThicknessROI(array, width, height, angle, dist_from_center, phantom_center)
Bases: pylinac.core.roi.RectangleROI
A rectangular ROI that measures the angled wire rod in the HU linearity slice which determines slice thickness.

long_profile
The profile along the axis perpendicular to ramped wire.

wire_fwhm
The FWHM of the wire in pixels.

plot_color
The plot color.

class pylinac.ct.GeometricLine(geo_roi1: pylinac.core.geometry.Point, geo_roi2: pylinac.core.geometry.Point, mm_per_pixel: float, tolerance: Union[int, float])
Bases: pylinac.core.geometry.Line
Represents a line connecting two nodes/ROIs on the Geometry Slice.

nominal_length_mm
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.

passed
Whether the line passed tolerance.

pass_fail_color
Plot color for the line, based on pass/fail status.

length_mm
Return the length of the line in mm.

Helper Functions

pylinac.ct.combine_surrounding_slices(dicomstack: pylinac.core.image.DicomImageStack, nominal_slice_num: int, slices_plusminus: int = 1, mode: str = 'mean') → numpy.ndarray

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** – The combined array of the DICOM stack slices.

Return type **numpy.array**

```python
def get_regions(slice_or_arr: Union[pylinac.ct.Slice, numpy.ndarray], fill_holes: bool = False, clear_borders: bool = True, threshold: str = 'otsu') -> Tuple[numpy.ndarray, list, int]
```

Get the skimage regions of a black & white image.

### 5.8 Log Analyzer

#### 5.8.1 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.

#### 5.8.2 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 numpay arrays along with a difference array if applicable. Each of these arrays can be quickly plotted for visual analysis. See Axis for more info.

### 5.8.3 Running the Demos

As usual, the module comes with demo files and methods:

```python
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
```

(continues on next page)
95th percentile error: 0.088 cm
Number of beam holdoffs: 20
Gamma pass %: 18.65
Gamma average: 0.468

Your file location will be different, but the values should be the same. The same can be done using the demo Trajectory log:

```python
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:
5.8.4 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.

```python
from pylinac import load_log

log_path = "C:/path/to/tlog.bin"
l = load_log(log_path)
```

In addition, a folder, ZIP archive, or URL can also be passed:

```python
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.

```python
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.

```python
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')
```
5.8.5 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:

```python
>>> 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 an `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`
• `previous_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:

```python
>>> 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:
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:

```python
>>> 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**

**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:

```python
>>> 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:

```python
>>> 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
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:

```python
import pylinac

log = pylinac.TrajectoryLog.from_demo()
log.axis_data.mu.plot_actual()
```

Plot the Gantry difference:

```python
log.axis_data.gantry.plot_difference()
```

Axis plots are just as easily saved:

```python
log.axis_data.gantry.save_plot_difference(filename='gantry diff.png')
```

Now, lets plot the actual fluence:

```python
log.fluence.actual.calc_map()
log.fluence.actual.plot_map()
```
5.8. Log Analyzer
And the fluence gamma. But note we must calculate the gamma first, passing in any DoseTA or DistTA parameters:

```python
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:

```python
log.subbeams[0].fluence.gamma.calc_map()
log.subbeams[0].fluence.actual.plot_map()
```

### 5.8.6 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:

```python
log = TrajectoryLog.from_demo()
log.to_csv()
```
5.8.7 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:

```python
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:

```python
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
```

5.8.8 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.
5.8.9 API Documentation

Main classes

These are the classes a typical user may interface with.

```python
```

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** `MachineLogs`.

**Return type** One of `Dynalog`, `TrajectoryLog`, `MachineLogs`.

```python
pylinac.log_analyzer.anonymize(source: str, inplace: bool = False, destination: bool = 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.

```python
class pylinac.log_analyzer.Dynalog(filename, exclude_beam_off: bool = True)
Bases: pylinac.log_analyzer.LogBase
```

Class for loading, analyzing, and plotting data within a Dynalog file.

**header**

Type `DynalogHeader`

**axis_data**

Type `DynalogAxisData`

**fluence**
Type `FluenceStruct`

```python
da_logfile
Path of the A* dynalog file.

b_logfile
Path of the B* dynalog file.

def num_beamholds
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.

def run_demo()
Run the Dynalog demo.

def publish_pdf(filename: str, notes: str = None, metadata: dict = None, open_file: bool = False)
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 ————–

def 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

The absolute file path to the corresponding dynalog file.

Return type str

class pylinac.log_analyzer.TrajectoryLog(filename: Union[str, BinaryIO], exclude_beam_off: bool = True)
Bases: pylinac.log_analyzer.LogBase

A class for loading and analyzing the data of a Trajectory log.

header

Type ~pylinac.log_analyzer.TrajectoryLogHeader, which has the following attributes:

axis_data

Type ~pylinac.log_analyzer.TrajectoryLogAxisData

fluence

Type ~pylinac.log_analyzer.FluenceStruct

subbeams

5.8. Log Analyzer
Type

- pylinac.log_analyzer.SubbeamManager

**txt_filename**
The name of the associated .txt file for the .bin file. The file may or may not be available.

classmethod from_demo(exclude_beam_off: bool = True)
Load and instantiate from the demo trajectory log file included with the package.

static run_demo()
Run the Trajectory log demo.

to_csv(filename: Optional[str] = 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
The full filename of the newly created CSV file.

Return type
str

publish_pdf(filename: Union[str, BinaryIO], metadata: dict = None, notes: Union[str, list] = None, open_file: bool = False)
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 ————–

**num_beamholds**
Return the number of times the beam was held.

**is_hdmlc**
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:

```python
>>> log_folder = r'C:\path\log\directory'
>>> logs = MachineLogs(log_folder)
```
Batch methods include determining the average gamma and average gamma pass value:

```python
>>> 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.

**num_logs**  
The number of logs currently loaded.

**num_tlogs**  
The number of Trajectory logs currently loaded.

**num_dlogs**  
The number of Dynalog 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: Union[int, float] = 1, distTA: Union[int, float] = 1, threshold: Union[int, float] = 0.1, resolution: Union[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: Union[int, float] = 1, distTA: Union[int, float] = 1, threshold: Union[int, float] = 0.1, resolution: Union[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** A list of all the filenames of the newly created CSV files.

**Return type** list

anonymousize (inplace: bool = False, suffix: Optional[str] = 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** A list containing the paths to the newly written files.

**Return type** list

class pylinac.log_analyzer.Graph

Bases: enum.Enum

An enumeration.

GAMMA = 'gamma'
HISTOGRAM = 'histogram'
RMS = 'rms'

class pylinac.log_analyzer.MLCBank

Bases: enum.Enum

An enumeration.

A = 'A'
B = 'B'
BOTH = 'both'
class pylinac.log_analyzer.Fluence
    Bases: enum.Enum
    An enumeration.
    ACTUAL = 'actual'
    EXPECTED = 'expected'
    GAMMA = 'gamma'

class pylinac.log_analyzer.TreatmentType
    Bases: enum.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: numpy.ndarray, expected: Optional[numpy.ndarray] = None)
    Bases: object
    Represents an ‘Axis’ of a Trajectory log or dynalog file, holding actual and potentially expected and difference values.

    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.

difference
    Return an array of the difference between actual and expected positions.

    Returns Array the same length as actual/expected.

    Return type  numpy.ndarray

    plot_actual() → None
    Plot the actual positions as a matplotlib figure.

5.8. Log Analyzer
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: Optional[numpy.ndarray] = 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)
• 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 (Jaw_Struct)
• 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).

leaf_axes
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.

Type  dict containing Axis

classmethod from_dlog(dlog, jaws, snapshot_data: numpy.ndarray, snapshot_idx: Union[list, numpy.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.

num_pairs
Return the number of MLC pairs.

num_leaves
Return the number of MLC leaves.

num_snapshots
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() 

num_moving_leaves
Return the number of leaves that moved.

moving_leaves
Return an array of the leaves that moved during treatment.

add_leaf_axis(leaf_axis: pylinac.log_analyzer.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.

### leafMoved

```
def 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.

### pairMoved

```
def 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.

### getRMSAvg

```
def get_RMS_avg(bank: pylinac.log_analyzer.MLCBank = <MLCBank.BOTH: '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**

**Return type** `float`

### getRMSMax

```
def get_RMS_max(bank: pylinac.log_analyzer.MLCBank = <MLCBank.BOTH: 'both'>) -> float
```

Return the overall maximum RMS of given leaves.

**Parameters**

- **bank** – Specifies which bank(s) is desired.

**Returns**

**Return type** `float`

### getRMSPercentile

```
def get_RMS_percentile(percentile: Union[int, float] = 95, bank: pylinac.log_analyzer.MLCBank = <MLCBank.BOTH: 'both'>, only_moving_leaves: bool = False)
```

Return the n-th percentile value of RMS for the given leaves.

**5.8. Log Analyzer**
Get 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 An array for the given leaves containing the RMS error.

Return type numpy.ndarray

Get 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 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.

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 An array of size leaves-x-num_snapshots

Return type `numpy.ndarray`

### create_RMS_array

`create_RMS_array(leaves: Sequence[int]) → numpy.ndarray`

Create an RMS array of only the leaves specified.

**Parameters**

- **leaves** (*sequence*) – Leaves desired.

**Returns** An array of size leaves-x-num_snapshots

**Return type** `numpy.ndarray`

### leaf_under_y_jaw

`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

`get_snapshot_values(bank_or_leaf: Union[pylinac.log_analyzer.MLCBank, Iterable[T_co]] = <MLCBank.BOTH: 'both'>, dtype: str = 'actual') → numpy.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** 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).

**Return type** `ndarray`

### plot_mlc_error_hist

`plot_mlc_error_hist(show: bool = True) → None`

Plot an MLC error histogram.

### save_mlc_error_hist

`save_mlc_error_hist(filename: str, **kwargs) → None`

Save the MLC error histogram to file.

### plot_rms_by_leaf

`plot_rms_by_leaf(show: bool = True) → None`

Plot RMSs by leaf.

### save_rms_by_leaf

`save_rms_by_leaf(filename: str, **kwargs) → None`

Save the RMS-leaf to file.

### class pylinac.log_analyzer.DynalogHeader

`__init__(dlogdata)`

Bases: `pylinac.core.utilities.Structure`

**version**

The Dynalog version letter.

- **Type** `str`

### patient_name

Patient information.

- **Type** `str`

### plan_filename

Filename if using standalone. If using Treat <=6.5 will produce PlanUID, Beam Number. Not yet implemented for this yet.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolerance</td>
<td>str</td>
<td>Plan tolerance.</td>
</tr>
<tr>
<td>num_mlc_leaves</td>
<td>int</td>
<td>Number of MLC leaves.</td>
</tr>
<tr>
<td>clinac_scale</td>
<td>int</td>
<td>Clinac scale; 0 -&gt; Varian scale, 1 -&gt; IEC 60601-2-1 scale</td>
</tr>
<tr>
<td>num_snapshots</td>
<td>int</td>
<td>Number of snapshots recorded.</td>
</tr>
<tr>
<td>mu</td>
<td>Axis</td>
<td>Current dose fraction</td>
</tr>
</tbody>
</table>

**Note:** This can be gantry rotation under certain conditions. See Dynalog file specs.
collimator
    Collimator data in degrees.
    Type Axis

jaws
    Jaw data structure. Data in cm.
    Type Jaw_Struct

carriage_A
    Carriage A data. Data in cm.
    Type Axis

carriage_B
    Carriage B data. Data in cm.
    Type Axis

mlc
    MLC data structure. Data in cm.
    Type MLC

Read the dynalog axis data.

class pylinac.log_analyzer.TrajectoryLogHeader (file: BinaryIO)
Bases: object
    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
    collimator [Axis] Collimator data in degrees.
    couch [Couch_Struct] 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_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.

c**ontrol_point**

Internally-defined marker that defines where the plan is currently executing.

  **Type** int

mu_delivered

Dose delivered in units of MU.

  **Type** float

rad_time

Radiation time in seconds.

  **Type** float

sequence_num

Sequence number of the subbeam.

  **Type** int

beam_name

Name of the subbeam.

  **Type** str

gantry_angle

Median gantry angle of the subbeam.

collimator_angle

Median collimator angle of the subbeam.

jaw_x1

Median X1 position of the subbeam.

jaw_x2

Median X2 position of the subbeam.
jaw_y1
Median Y1 position of the subbeam.

jaw_y2
Median Y2 position of the subbeam.

class pylinac.log_analyzer.FluenceStruct (mlc_struct=None, 
                                           pylinac.log_analyzer.Axis = None, 
                                           jaw_struct=None)

Bases: object

Structure for data and methods having to do with fluences.

actual
The actual fluence delivered.

  Type FluenceBase

expected
The expected, or planned, fluence.

  Type FluenceBase

gamma
The gamma structure regarding the actual and expected fluences.

  Type GammaFluence

class pylinac.log_analyzer.FluenceBase (mlc_struct=None, 
                                         pylinac.log_analyzer.Axis = None, 
                                         jaw_struct=None)

Bases: object

An abstract base class to be used for the actual and expected fluences.

array
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.

  Type numpy.ndarray

resolution
The resolution of the fluence calculation; -1 means calculation has not been done yet.

  Type int, float

Parameters

  • mlc_struct (MLC_Struct) -
  • mu_axis (BeamAxis) -
  • jaw_struct (Jaw_Struct) -

is_map_calced (raise_error: bool = False) \rightarrow bool
Return a boolean specifying whether the fluence has been calculated.

calc_map (resolution: float = 0.1, equal_aspect: bool = False) \rightarrow numpy.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** 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.

**rtype** numpy.ndarray

**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, pylinac.log_analyzer.Axis = None, jaw_struct=None*)

**Bases:** pylinac.log_analyzer.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, pylinac.log_analyzer.Axis = None, jaw_struct=None*)

**Bases:** pylinac.log_analyzer.FluenceBase

The expected fluence object.

**Parameters**

- **mlc_struct** (*MLC_Struct*) –

- **mu_axis** (*BeamAxis*) –

- **jaw_struct** (*Jaw_Struct*) –


**Bases:** pylinac.log_analyzer.FluenceBase

Gamma object, including pixel maps of gamma, binary pass/fail pixel map, and others.

**array**

The gamma map. Only available after calling calc_map()

**Type** numpy.ndarray

**passfail_array**

The gamma pass/fail map; pixels that pass (<1.0) are set to 0, while failing pixels (>=1.0) are set to 1.

**Type** numpy.ndarray
distTA
The distance to agreement value used in gamma calculation.

Type int, float
doseTA
The dose to agreement value used in gamma calculation.

Type int, float

threshold
The threshold percent dose value, below which gamma was not evaluated.

Type int, float

pass_prctnt
The percent of pixels passing gamma (<1.0).

Type float

avg_gamma
The average gamma value.

Type float

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: Union[int, float] = 1, distTA: Union[int, float] = 1, threshold: Union[int, float] = 0.1, resolution: Union[int, float] = 0.1, calc_individual_maps: bool = False) → numpy.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 A num_mlc_leaves-x-400/resolution numpy array.

Return type numpy.ndarray

plot_map (show: bool = True)
Plot the fluence; the fluence (pixel map) must have been calculated first.

histogram (bins: Optional[list] = None) → Tuple[numpy.ndarray, numpy.ndarray]
Return a histogram array and bin edge array of the gamma map values.
```python
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: Optional[list] = 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: Optional[list] = 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.


Bases: object
Jaw Axes data structure.

x1
Type Axis

y1
Type Axis

x2
Type Axis

y2
Type Axis

class pylinac.log_analyzer.CouchStruct(vert: pylinac.log_analyzer.CouchAxis, lon:

Bases: object
Couch Axes data structure.

vert
Type Axis

long
```
Type Axis

lat1

Type Axis

rotn

Type Axis

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 & vice 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.

5.9 Picket Fence

5.9.1 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.

5.9.2 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.

### 5.9.3 Running the Demo

To run the picketfence demo, create a script or start in interpreter and input:

```python
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:

```python
pf = PicketFence.from_demo()
pf.analyze()
pf.publish_pdf(filename='PF_Oct-2018.pdf')
```

### 5.9.4 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.

### 5.9.5 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:

```python
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:

```python
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):

```python
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:

```python
pf = PicketFence.from_demo_image()
```

You can also change the MLC type:

```python
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...

• 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:

```python
# print results to the console
print(pf.results())

# view analyzed image
pf.plot_analyzed_image()
```

which results in:

![Matplotlib Image](image.png)

The plot is also able to be saved to PNG:

```python
pf.save_analyzed_image('mypf.png')
```

Or you may save to PDF:

```python
pf.publish_pdf('mypf.pdf')
```

5.9.6 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(...)
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.

### 5.9.7 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/rt/t individual analysis vs combined.

- **One leaf error:** When one single leaf has an error. This is the quintessential example for PF.

![Combined analysis](image)

*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.

### 5.9.8 Plotting a histogram

As of v3.0, you may plot a histogram of the error data like so:

```python
from pylinac import PicketFence
pf = PicketFence.from_demo_image()
```

(continues on next page)


Fig. 6: Separate analysis

(continued from previous page)

```python
pf.analyze()
pf.plot_histogram()
```

### 5.9.9 Plotting a leaf profile

As of v3.0, you may plot an individual leaf profile like so:

```python
from pylinac import PicketFence
pf = PicketFence.from_demo_image()
pf.analyze()
pf.plot_leaf_profile(leaf=15, picket=2)
```

### 5.9.10 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:

```python
from pylinac import PicketFence

pf = PicketFence('my/pf.dcm', log='my/pf_log.bin')
...
```

Everything else is the same except the measurements are **absolute**.
5.9. Picket Fence

MLC profile Leaf: 15, Picket: 2

- Fitted picket location
- Measured MLC position
- Guard rail
- Guard rail
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:

![Graph showing average error in mm for different positions.]

5.9.11 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:

```python
from pylinac.picketfence import PicketFence, MLCs

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:

```python
from pylinac.picketfence import PicketFence, MLCArrangement
mlc_setup = MLCArrangement(leaf_arrangement=\[(10, 10), (40, 5), (10, 10)\])
mlc_setup_offset = MLCArrangement(leaf_arrangement=\..., offset=2.5)  # offset is in mm
pf = PicketFence('path/to/img', mlc=mlc_setup)
```

5.9.12 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.

5.9.13 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:
```python
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:

```python
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:

```python
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:
5.9.14 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`.

```python
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,
)
```

(continues on next page)
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.

```python
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),
],
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()
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(
    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`

```python
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`

```python
from scipy import ndimage

import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,
PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "rotated_up_down.dcm"
generate_picketfence(
    simulator=AS1200Image(sid=1500),
    # (continues on next page)
```

5.9. Picket Fence
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
import pylinac
from pylinac.core.image_generator import generate_picketfence, GaussianFilterLayer,
    PerfectFieldLayer, RandomNoiseLayer, AS1200Image
from pylinac.picketfence import Orientation

pf_file = "offset_picket.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,
    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.

```python
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
    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
        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()
5.9.15 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

<table>
<thead>
<tr>
<th>Warning: Analysis can fail or give unreliable results if any Restriction is violated.</th>
</tr>
</thead>
</table>

- 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.
5.9.16 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:

  ```python
  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:

  ```python
  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:

  ![Streak Artifacts Example](image.png)

  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:

```python
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:

```python
pf = PicketFence(r'my/pf.dcm')
pf.image.crop(pixels=3)
pf.analyze()
```

### 5.9.17 API Documentation

**Main classes**

These are the classes a typical user may interface with.

```python
```

**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)
Instantiate from a URL.

classmethod from_demo_image(filter: int = None)
Construct a PicketFence instance using the demo image.

classmethod from_multiple_images(path_list: Iterable[Union[str, pathlib.Path]], dtype: numpy.dtype = <class 'numpy.uint16'>, **kwargs)
Load and superimpose multiple images and instantiate a Starshot object.

Parameters
  • path_list (iterable) – An iterable of path locations to the files to be loaded/combined.
  • kwargs – Passed to load_multiples().

passed
Boolean specifying if all MLC positions were within tolerance.

percent_passing
Return the percentage of MLC positions under tolerance.

max_error
Return the maximum error found.

max_error_picket
Return the picket number where the maximum error occurred.

max_error_leaf
Return the leaf that had the maximum error.

failed_leaves() → Union[List[int], List[str]]
A list of the failed leaves. Either the leaf number or the bank+leaf number if using separate leaves.

abs_median_error
Return the median error found.

num_pickets
Return the number of pickets determined.

mean_picket_spacing
The average distance between pickets in mm.

plot_leaf_profile(leaf: Union[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: Union[str, pathlib.Path, BinaryIO], leaf: Union[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
Run the Picket Fence demo using the demo image. See analyze() for parameter info.

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.

```python
plot_analyzed_image(guard_rails: bool = True, mlc_peaks: bool = True, overlay: bool = True, leaf_error_subplot: bool = True, show: bool = True, figure_size: Union[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.

```python
save_analyzed_image(filename: Union[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.

```python
results() → str
```

Return results of analysis. Use with print().

```python
results_data(as_dict=False) → Union[pylinac.picketfence.PFResult, dict]
```

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

```python
```

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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

```python
def 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.
```

```python
def plot_histogram(bins: int = 10, show: bool = True) -> None
    Plot a histogram of the leaf errors
```

```python
def save_histogram(filename: [<class 'str'>, <class 'pathlib.Path'>, <class 'typing.BinaryIO'>], bins: int = 10, **kwargs) -> None
    Save a histogram of the leaf errors
```

**orientation**

The orientation of the image, either Up-Down or Left-Right.

```python
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.)* –

```python
class pylinac.picketfence.Orientation
Bases: enum.Enum

Possible orientations of the image

UP_DOWN = 'Up-Down'

LEFT_RIGHT = 'Left-Right'
```

```python
class pylinac.picketfence.MLC
Bases: enum.Enum

The pre-built MLC types

MILLENNIUM = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name': 'Millennium'}

HD_MILLENNIUM = {'arrangement': <pylinac.picketfence.MLCArrangement object>, 'name': 'HD Millennium'}

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, mean_picket_spacing_mm: float, offsets_from_cax_mm: List[float], passed: bool, failed_leaves: Union[List[str], List[int]], mlc_skew: float)

Bases: pylinac.core.utilities.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 = None
action_tolerance_mm = None
percent_leaves_passing = None
number_of_pickets = None
absolute_median_error_mm = None
max_error_mm = None
mean_picket_spacing_mm = None
offsets_from_cax_mm = None
passed = None
failed_leaves = None
mlc_skew = None

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: pylinac.core.image.LinacDicomImage

A subclass of a DICOM image that checks for noise and inversion when instantiated. Can also adjust for EPID sag.

adjust_for_sag(sag: int, orientation: Union[str, pylinac.picketfence.Orientation]) → None

Roll the image to adjust for EPID sag.


Bases: object

Holds picket information in a Picket Fence test.

get_fit() → numpy.poly1d

The fit of a polynomial to the MLC measurements.

dist2cax

The distance from the CAX to the picket, in mm.
**left_guard_separated**
The line representing the left-sided guard rails. When not doing separate analysis, the left and right rails will overlap.

**right_guard_separated**
The line representing the right-sided guard rails.

**add_guards_to_axes** *(axis: matplotlib.axes.Axes, color: str = 'g') → None*
Plot guard rails to the axis.

```python
```

Bases: object

Representation of an MLC kiss or of each MLC about a kiss.

**full_leaf_nums**
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: matplotlib.axes.Axes, width: Union[float, int] = 1) → None*
Plot the measurement to the axes.

**passed**
Whether the MLC kiss or leaf was within tolerance.

**passed_action**
Whether the MLC kiss or leaf was within the action tolerance.

**bg_color**
The color of the measurement when the PF image is plotted, based on pass/fail status.

**picket_positions**
The position(s) of the pickets in mm

**error**
The error (difference) of the MLC measurement and the picket fit. If using individual leaf analysis, returns both errors otherwise return one.

**max_abs_error**
The maximum absolute error

**marker_lines**
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
5.10 Winston-Lutz

5.10.1 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”.

5.10.2 Running the Demo

To run the Winston-Lutz demo, create a script or start an interpreter session and input:

```python
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
```

5.10.3 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
Gantry positions superimposed
Gantry iso size: 1.09mm
Gantry wobble

Collimator positions superimposed
Collimator iso size: 1.08mm
Collimator wobble

Couch positions superimposed
Couch iso size: 2.35mm
Couch wobble
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.

#### 5.10.4 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.

#### 5.10.5 Typical Use

Analyzing a Winston-Lutz test is simple. First, let’s import the class:

```python
from pylinac import WinstonLutz
```

From here, you can load a directory:

```python
my_directory = 'path/to/wl_images'
w1 = WinstonLutz(my_directory)
```

You can also load a ZIP archive with the images in it:

```python
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
```

### 5.10.6 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']
...
```

### 5.10.7 Accessing individual images

Each image can be plotted and otherwise accessed easily:
5.10.8 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`.

```python
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.

5.10.9 Using File Names

If your linac EPID images do not include axis information (such as Elekta) 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: ‘<*>gantry<*>coll<*>couch<*>’. 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.

5.10.10 Changing BB detection size

To change the size of BB pylinac is expecting you can pass the size to the analyze method:

```python
class my_directory = 'path/to/wl_images'
wl = WinstonLutz(my_directory, use_filenames=True)
wl.analyze(bb_size_mm=3)
```

5.10.11 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.

### 5.10.12 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 be found by “hackprojecting” 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. 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. The gantry is also reversed in Varian scale, but this can be compensated by simply reversing the sign of the Z axis.

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 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, the BB is considered found and the pixel-weighted center of mass of the BB is considered the BB location.

**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.

### 5.10.13 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:

```python
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)
wplot_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
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:

```python
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)
wll.plot_images()

G=0, B=0, P=0
CAX to BB: 1.01mm
wl_dir/WL G=0, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=1, in=0, up=0; Gantry
tilt=0, Gantry sag=0.dcm

G=90, B=0, P=0
CAX to BB: 0.00mm
wl_dir/WL G=90, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=1, in=0, up=0; Gantry
tilt=0, Gantry sag=0.dcm

G=180, B=0, P=0
CAX to BB: 1.01mm
wl_dir/WL G=180, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=1, in=0, up=0; Gantry
tilt=0, Gantry sag=0.dcm

G=270, B=0, P=0
CAX to BB: 0.00mm
wl_dir/WL G=270, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=1, in=0, up=0; Gantry
tilt=0, Gantry sag=0.dcm

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:

```python
import pylinac
from pylinac.core.image_generator import (GaussianFilterLayer,
FilteredFieldLayer,
AS1200Image,
RandomNoiseLayer,
```

(continued on next page)
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,
)

wl = pylinac.WinstonLutz(wl_dir)
wl.analyze(bb_size_mm=4)
wl.plot_images()

G=0, B=0, P=0
CAX to BB: 0.90mm
wl_dir/WL G=0, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=0, in=0, up=0; Gantry
tilt=1, Gantry sag=0.dcm

G=90, B=0, P=0
CAX to BB: 0.00mm
wl_dir/WL G=90, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=0, in=0, up=0; Gantry
tilt=1, Gantry sag=0.dcm

G=180, B=0, P=0
CAX to BB: 0.90mm
wl_dir/WL G=180, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=0, in=0, up=0; Gantry
tilt=1, Gantry sag=0.dcm

G=270, B=0, P=0
CAX to BB: 0.00mm
wl_dir/WL G=270, C=0, P=0;
Field=(40, 40)mm; BB=4mm @
left=0, in=0, up=0; Gantry
tilt=1, Gantry sag=0.dcm

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'
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)
w1.analyze(bb_size_mm=4)
w1.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,
)
w1 = pylinac.WinstonLutz(w1_dir)
w1.analyze(bb_size_mm=4)
w1.plot_images()

5.10.14 API Documentation

class pylinac.winston_lutz.WinstonLutz(directory: str, use_filenames: bool = False)
Bases: object

Class for performing a Winston-Lutz test of the radiation isocenter.

Parameters

- **directory**(str) – Path to the directory of the Winston-Lutz EPID images.
- **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.

images = None

classmethod from_demo_images()

Instantiate using the demo images.

classmethod from_zip(zfile: Union[str, BinaryIO], use_filenames: bool = False)

Instantiate from a zip file rather than a directory.

Parameters

- **zfile** – Path to the archive file.
- **use_filenames**(bool) – Whether to interpret axis angles using the filenames. Set to true for Elekta machines where the gantry/coll/couch data is not in the DICOM metadata.

classmethod from_url(url: str, use_filenames: bool = False)

Instantiate from a URL.

Parameters

- **url**(str) – URL that points to a zip archive of the DICOM images.
• **use_filenames** *(bool)* – Whether to interpret axis angles using the filenames. Set to true for Elekta machines where the gantry/coll/couch data is not in the DICOM metadata.

```python
static run_demo()  
Run the Winston-Lutz demo, which loads the demo files, prints results, and plots a summary image.
```

```python
analyze(bb_size_mm: float = 5)  
Analyze the WL images.
```

**Parameters**

**bb_size_mm** – The expected size of the BB in mm. The actual size of the BB can be +/-2mm from the passed value.

```python
gantry_iso_size  
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.
```

```python
gantry_coll_iso_size  
The diameter of the 3D gantry isocenter size in mm including collimator and gantry/coll combo images. Images where the couch!=0 are excluded.
```

```python
collimator_iso_size  
The 2D collimator isocenter size (diameter) in mm. The iso size is in the plane normal to the gantry.
```

```python
couch_iso_size  
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.
```

```python
bb_shift_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.

```python
bb_shift_instructions(couch_vrt: Optional[float] = None, couch_lng: Optional[float] = None,  
couch_lat: Optional[float] = 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.

```python
axis_rms_deviation(axis: pylinac.winston_lutz.Axis = <Axis.GANTRY: 'Gantry'>, value: str =  
'all') → Union[Iterable[T_co], 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'.

```python
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'))* – 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**: `str = 'max', 'median'` – The metric of distance to use.


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_images** *(axis: pylinac.winston_lutz.Axis = <Axis.GANTRY: 'Gantry'>, show: bool = True) → None*

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'}`
- **show**: `bool` – Whether to show the image.

**save_images** *(filename: Union[str, BinaryIO], axis: pylinac.winston_lutz.Axis = <Axis.GANTRY: '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**: `None` – The axis to save.

**plot_summary** *(show: bool = True, fig_size: Optional[tuple] = None) → None*

Plot a summary figure showing the gantry sag and wobble plots of the three axes.

**save_summary** *(filename: Union[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=False) → Union[pylinac.winston_lutz.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: Union[str, List[str], None] = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

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, max_2d_cax_to_epid_mm: float, median_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)

Bases: pylinac.core.utilities.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.

```
neg_num_gantry_images = None
neg_num_gantry_coll_images = None
neg_num_coll_images = None
neg_num_couch_images = None
neg_num_total_images = None
neg_max_2d_cax_to_bb_mm = None
neg_median_2d_cax_to_bb_mm = None
neg_max_2d_cax_to_epid_mm = None
neg_median_2d_cax_to_epid_mm = None
neg_gantry_3d_iso_diameter_mm = None
neg_max_gantry_rms_deviation_mm = None
neg_max_epid_rms_deviation_mm = None
neg_gantry_coll_3d_iso_diameter_mm = None
neg_coll_2d_iso_diameter_mm = None
neg_max_coll_rms_deviation_mm = None
neg_couch_2d_iso_diameter_mm = None
```
max_couch_rms_deviation_mm = None

class pylinac.winston_lutz.ImageManager(directory: str, use_filenames: bool = False):
    Bases: list

    Manages the images of a Winston-Lutz test.

    Parameters

    • directory (str) – The path to the images.

    • 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.

class pylinac.winston_lutz.WinstonLutz2D(file: Union[str, BinaryIO], use_filenames: bool = False):
    Bases: pylinac.core.image.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) → None
    Analyze the image.

    epid
    Center of the EPID panel

    cax_line_projection
    The projection of the field CAX through space around the area of the BB. Used for determining gantry isocenter size.

    Returns: The virtual line in space made by the beam CAX.

    Return type: Line

couch_angle_varian_scale
    The couch angle converted from IEC 61217 scale to “Varian” scale. Note that any new Varian machine uses 61217.

cax2bb_vector
    The vector in mm from the CAX to the BB.

cax2bb_distance
    The scalar distance in mm from the CAX to the BB.

cax2epid_vector
    The vector in mm from the CAX to the EPID center pixel

cax2epid_distance
    The scalar distance in mm from the CAX to the EPID center pixel

    plot(ax: Optional[matplotlib.axes._axes.Axes] = None, show: bool = True, clear_fig: bool = False):
    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.

**variable_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=False) \(\rightarrow\) Union[pylinac.winston_lutz.WinstonLutz2DResult, dict]

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

## 5.11 Planar Imaging

### 5.11.1 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
• 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.

### 5.11.2 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:

```python
from pylinac import LeedsTOR  # or LasVegas, DoselabMC2kV, etc
LeedsTOR.run_demo()
```

A figure showing the phantom, low contrast plot, and RMTF will be generated:

![Leeds Phantom Analysis](image)

Typical Use

Import the class:

```python
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:

  ```python
  leeds = LeedsTOR('my/leeds.dcm')
  ```

  Alternatively, a URL can be passed:
You may also use the demo image:

```python
leeds = LeedsTOR.from_demo_image()
```

- **Analyze the images** – Analyze the image using the `analyze()` method. The low and high contrast thresholds can be specified:

```python
leeds.analyze(low_contrast_threshold=0.01, high_contrast_threshold=0.5)
```

Additionally, you may specify the SSD of the phantom if it is not at iso (e.g. sitting on the panel):

```python
leeds.analyze(..., ssd=1400)
```

- **View the results** – The results of analysis can be viewed with the `plot_analyzed_image()` method.

```python
leeds.plot_analyzed_image()
```

Note that each subimage can be turned on or off.

```python
# don't show the low contrast plot
leeds.plot_analyzed_image(low_contrast=False)
```

The figure can also be saved:
A PDF report can also be generated:

```python
leeds.publish_pdf('leeds_october16.pdf')
```

## 5.11.3 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.

### 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 (MTF)*.

**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`:
5.11.4 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 (MTF).

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.

5.11.5 Las Vegas Phantom

The Las Vegas phantom is for MV image quality testing and includes low contrast regions of varying contrast and size.
Image Acquisition

The Las Vegas phantom has a recommended position as stated on the phantom. Pylinac will however account for angles, shifts, and inversions. Best practices for the Las Vegas phantom:

- Keep the phantom 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 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.

5.11.6 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 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).

5.11.7 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:

   ```python
   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.

   ```python
   class CustomPhantom(ImagePhantomBase):
       common_name = 'Custom Phantom v2.0'
   ```

3. If the phantom has a high-contrast measurement object, define the ROI locations.

   ```python
   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.

```python
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},
        # 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:

```python
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
```

```python
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.

```python
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.
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).
    return Point.

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.
    return float.

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.

### 5.11.8 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.

```
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):
    high_contrast_roi_settings = {
```
Calculate a specific MTF

To calculate a specific MTF value, i.e. the frequency at a given MTF%:

```python
dl = DoselabMC2kV(...) dl.analyze(...) print(dl.mtf.relative_resolution(x=50))  # 50% rMTF```

Get/View the contrast of a low-contrast ROI

```python
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:

```python
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(...)```

5.11.9 Wrong phantom angle

It may sometimes be the case 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.

### 5.11.10 API Documentation

**class** pylinac.planar_imaging.LeedsTOR(filepath: Union[str, BinaryIO], normalize: bool = True)

* Bases:* pylinac.planar_imaging.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.

* 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: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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 counter-clockwise.

• **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.

• **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.

**phantom_bbox_size_px**
The phantom bounding box size in pixels^2.

**phantom_ski_region**
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) → Tuple[List[matplotlib.figure.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.

**publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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’: ‘True-Beam’}` would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

**results()** → str
Return the results of the analysis.

**save_analyzed_image** (filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], None]
Save the analyzed image to disk or to stream. Kwars 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.StandardImagingQC3 (filepath: Union[str, BinaryIO], normalize: bool = True)
**Bases:** pylinac.planar_imaging.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.

**classmethod from_demo_image** ()
Instantiate and load the demo image.

**static run_demo** () → None
Run the Standard Imaging QC-3 phantom analysis demonstration.

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.

• **low_contrast_method** – The equation to use for calculating low contrast.

• **visibility_threshold** – The threshold for whether an ROI is “seen”.

```python
classmethod from_url(url: str)
```

**Parameters**

url *(str)* – The URL to the image.

```python
phantom_bbox_size_px
```

The phantom bounding box size in pixels^2.

```python
phantom_ski_region
```

The skimage region of the phantom outline.

```python
plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False) → Tuple[List[matplotlib.figure.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.

**publish_pdf** *(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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’: ‘True-Beam’}` would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

**results** *(*) → str

Return the results of the analysis.

**save_analyzed_image** *(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], 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.StandardImagingQCkV** *(filepath: Union[str, BinaryIO], normalize: bool = True)*

Bases: **pylinac.planar_imaging.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.

**static run_demo** *(*) → None

Run the Standard Imaging QC-3 phantom analysis demonstration.


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.

- **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.

### phantom_bbox_size_px

The phantom bounding box size in pixels^2.

### phantom_ski_region

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)* → Tuple[List[matplotlib.figure.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.

**publish_pdf** *(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

**results** () → str

Return the results of the analysis.

**save_analyzed_image** *(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs)* → Union[List[str], Dict[str, BinaryIO], 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.LasVegas** *(filepath: Union[str, BinaryIO], normalize: bool = True)*

**Bases:** `pylinac.planar_imaging.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.

**static run_demo** ()

Run the Las Vegas phantom analysis demonstration.
analyze (low_contrast_threshold: float = 0.05, high_contrast_threshold: float = 0.5, invert: bool = False, angle_override: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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.
- **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.

**phantom_bbox_size_px**

The phantom bounding box size in pixels^2.

**phantom_ski_region**

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) \rightarrow Tuple[List[matplotlib.figure.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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

results() \rightarrow str

Return the results of the analysis.

save_analyzed_image(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) \rightarrow Union[List[str], Dict[str, BinaryIO], None]

Save the analyzed image to disk or to stream. Kwarg is 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.DoselabMC2MV(filepath: Union[str, BinaryIO], normalize: bool = True)

Bases: pylinac.planar_imaging.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.

```python
static run_demo() → None
Run the Doselab MC2 MV-area phantom analysis demonstration.
```

```python
analyze(low_contrast_threshold: float = 0.05, high_contrast_threshold: float = 0.5, invert: bool = False, angle_override: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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.

- **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:** 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.

- **low_contrast_method** – The equation to use for calculating low contrast.

- **visibility_threshold** – The threshold for whether an ROI is “seen”.

```python
classmethod from_demo_image()
Instantiate and load the demo image.
```

```python
classmethod from_url(url: str)
Parameters url (str) – The URL to the image.
```
phantom_bbox_size_px
The phantom bounding box size in pixels^2.

phantom_ski_region
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) → Tuple[List[matplotlib.figure.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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

results() → str
Return the results of the analysis.

save_analyzed_image(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], 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.DoselabMC2kV(filepath: Union[str, BinaryIO], normalize: bool = True)
Bases: pylinac.planar_imaging.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.

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: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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.
- **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.

phantom_bbox_size_px
    The phantom bounding box size in pixels^2.

phantom_ski_region
    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) → Tuple[List[matplotlib.figure.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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: Author: James Unit: True-Beam

results() → str
    Return the results of the analysis.

save_analyzed_image(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], None]
    Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig()
• **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.

```python
class pylinac.planar_imaging.SNCMV(filepath: Union[str, BinaryIO], normalize: bool = True)

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.

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: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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.

• **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.
• **low_contrast_method** – The equation to use for calculating low contrast.
• **visibility_threshold** – The threshold for whether an ROI is “seen”.

```python
classmethod from_demo_image()
    Instantiate and load the demo image.

classmethod from_url(url: str)
    Parameters url (str) – The URL to the image.
```

```python
phantom_bbox_size_px
    The phantom bounding box size in pixels^2.

phantom_ski_region
    The skimage region of the phantom outline.
```

```python
plot_analyzed_image(image: bool = True, low_contrast: bool = True, high_contrast: bool = True, show: bool = True, split_plots: bool = False) → Tuple[List[matplotlib.figure.Figure], List[str]]
    Plot the analyzed image.
```

```python
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.
```

```python
publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = None)
    Publish (print) a PDF containing the analysis, images, and quantitative results.
```

```python
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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

results() → str
    Return the results of the analysis.
```

```python
save_analyzed_image(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], None]
    Save the analyzed image to disk or to stream. Kwargs are passed to plt.savefig() 
```

```python
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.

```python
class pylinac.planar_imaging.SNCkV(filepath: Union[str, BinaryIO], normalize: bool = True)

Bases: pylinac.planar_imaging.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.

```staticmethod
run_demo() → None
```

Run the Sun Nuclear kV-QA phantom analysis demonstration.

```python
def analyze(low_contrast_threshold: float = 0.05, high_contrast_threshold: float = 0.5, invert: bool = False, angle_override: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = pylinac.core.roi.Contrast.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.

• **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:** 0 is pointing from the center toward the right and positive values go counterclockwise.

---
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.
- **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.

**phantom_bbox_size_px**

The phantom bounding box size in pixels^2.

**phantom_ski_region**

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) → Tuple[List[matplotlib.figure.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.

**publish_pdf** *(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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': 'True-Beam'} would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

**results () → str**

Return the results of the analysis.
save_analyzed_image(filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], 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.PTWEPIDQC(filepath: Union[str, BinaryIO], normalize: bool = True)

Bases: pylinac.planar_imaging.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.

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: Optional[float] = None, center_override: Optional[tuple] = None, size_override: Optional[float] = None, ssd: float = 1000, low_contrast_method: pylinac.core.roi.Contrast = <Contrast.MICHELSON: '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.
- **low_contrast_method** – The equation to use for calculating low contrast.
- **visibility_threshold** – The threshold for whether an ROI is “seen”.

```python
classmethod from_demo_image()
Instantiate and load the demo image.

classmethod from_url(url: str)
Parameters url (str) – The URL to the image.

phantom_bbox_size_px
The phantom bounding box size in pixels^2.

phantom_ski_region
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) → Tuple[List[matplotlib.figure.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.

publish_pdf(filename: str, notes: str = None, open_file: bool = False, metadata: Optional[dict] = 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’: ‘True-Beam’}` would result in text in the PDF like: ————– Author: James Unit: True-Beam ————–

`results () → str`  
Return the results of the analysis.

`save_analyzed_image (filename: Union[None, str, BinaryIO] = None, split_plots: bool = False, to_streams: bool = False, **kwargs) → Union[List[str], Dict[str, BinaryIO], 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.

## 5.12 Field Analysis

### 5.12.1 Overview

The field analysis module (`pylinac.field_analysis`) allows a physicist to analyze metrics from an EPID or 2D device array 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 create 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 contains two main classes `FieldAnalysis` which is used for EPID images and `DeviceFieldAnalysis` which is used for 2D device arrays.

### 5.12.2 Running the Demo

To run the demo, import the main class and run the demo method:

```bash
from pylinac import FieldAnalysis
FieldAnalysis.run_demo()
```
Field Profile Analysis

Horizontal Profile

Vertical Profile

Profile
Penumbra region
Field edge
Symmetry max
Flatness region

flatsym_demo.dcm
Profile Extraction Area

Normalized Response

pixels

Symmetry (%)

mm

Normalized Response

mm

Symmetry (%)

pixels
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
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%

5.12.3 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:

```python
from pylinac import FieldAnalysis
```

Loading images is easy and just like any other module:

```python
# 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 = DeviceFieldAnalysis.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:

```python
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
```

## 5.12.4 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`.

```python
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.

```python
from pylinac import FieldAnalysis, Centering

fa = FieldAnalysis(...)  
fa.analyze(..., centering=Centering.MANUAL)  # default is the middle of the image
```
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.

```python
typ from pylinac import FieldAnalysis, Centering
fa = FieldAnalysis(...)  
fa.analyze(...)  # nothing special needed as it's the default  
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.

```python
typ from pylinac import FieldAnalysis, Centering
fa = FieldAnalysis(...)  
fa.analyze(...)  # nothing special needed as it's the default  
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.

```python
typ 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.

```python
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:

\[ f(x) = A + \frac{B - A}{1 + C^D} \]

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.

```python
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.

5.12.5 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.

```python
from pylinac import FieldAnalysis, Normalization
fa = FieldAnalysis(...)
fa.analyze(..., normalization_method=Normalization.BEAM_CENTER)
```

5.12.6 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.

```python
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.

```python
from pylinac import FieldAnalysis, Interpolation
fa = FieldAnalysis(...)
fa.analyze(..., interpolation=Interpolation.SPLINE, interpolation_resolution_mm=0.1)
```
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.

**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.

```python
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.

\[
\text{flatness} = 100 \times \frac{|D_{\text{max}} - D_{\text{min}}|}{(D_{\text{max}} + D_{\text{min}})}
\]

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:

\[
\text{symmetry} = 100 \times \frac{\max(|L_{\text{pt}} - R_{\text{pt}}|)}{D_{\text{CAX}}}
\]

where \(L_{\text{pt}}\) and \(R_{\text{pt}}\) 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`.

```python
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.

\[ \text{flatness} = 100 \times \frac{D_{max}}{D_{min}} \]

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):

\[ \text{symmetry} = 100 \times \max(|L_{pt}/R_{pt}|, |R_{pt}/L_{pt}|) \]

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()`.

```python
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.

\[ \text{flatness} = 100 \times \frac{|D_{max} - D_{min}|}{(D_{max} + D_{min})} \]

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:

\[ \text{symmetry} = 100 \times \frac{(A_{left} - A_{right})}{(A_{left} + A_{right})} \]

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.

5.12.8 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:

```python
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, **kwars) -> float:
    # do whatever. Must return a float. `profile` will be called twice, once for the vertical profile and horizontal profile.
    # the kwars are passed to `analyze` and can be used here for your own purposes (e.g. fitting parameters)
    my_special_value = kwars.get("funkilicious")
    flatness = profile...
    return flatness

# custom plot function for the above flatness function
# This is OPTIONAL
# 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() call
    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': ...
    }
    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:

```python
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.

### 5.12.9 Loading Device Data

The field analysis module can load data from a 2D device array. Currently, only the SNC Profiler is supported, but this will expand over time. The `DeviceFieldAnalysis` class and associated `Device` class will handle the loading and parsing of the file as well as spatial resolution information.

To load a Profiler dataset:

```python
from pylinac import DeviceFieldAnalysis, Device
fa = DeviceFieldAnalysis(path='my/path/to/6x.prs', device=Device.PROFILER)
# proceed as normal
fa.analyze(...)
```

**Note:** While largely the same as the `FieldAnalysis` class, the `DeviceFieldAnalysis`'s `analyze` signature is different. Specifically, you cannot pass a vert/horiz position or width (since it’s not an image) nor can you pass a centering option. The table below shows the differences. Any parameter not shown is the same between the two.

<table>
<thead>
<tr>
<th>parameter</th>
<th>FieldAnalysis</th>
<th>DeviceFieldAnalysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>vert_position</td>
<td>0.5</td>
<td>N/A</td>
</tr>
<tr>
<td>horiz_position</td>
<td>0.5</td>
<td>N/A</td>
</tr>
<tr>
<td>vert_width</td>
<td>0.5</td>
<td>N/A</td>
</tr>
<tr>
<td>horiz_width</td>
<td>0.5</td>
<td>N/A</td>
</tr>
<tr>
<td>centering</td>
<td>Beam center</td>
<td>N/A</td>
</tr>
<tr>
<td>interpolation</td>
<td>Linear</td>
<td>None</td>
</tr>
<tr>
<td>normalization_method</td>
<td>Beam center</td>
<td>Geometric center</td>
</tr>
<tr>
<td>edge_detection_method</td>
<td>Inflection derivative</td>
<td>Inflection Hill</td>
</tr>
</tbody>
</table>

### 5.12.10 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:

```python
from pylinac import DeviceFieldAnalysis, Protocol
fa = DeviceFieldAnalysis.from_demo_image()
fa.analyze(protocol=Protocol.VARIAN, is_FFF=True)
fa.plot_analyzed_image()
```

Field Profile Analysis

```
<table>
<thead>
<tr>
<th>Horizontal Profile</th>
<th>Vertical Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td>detector</td>
<td>detector</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

Normalized Response

- Profile
- Penumbra region
- Hill fit
- Field edge
- "top" polynomial fit
- "top" position
- in-field slope
- Symmetry max
- Flatness region

"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
```
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
```

5.12.11 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`.

5.12. Field Analysis
5.12.12 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.

5.12.13 API Documentation

Main classes

These are the classes a typical user may interface with.

```python
class pylinac.field_analysis.FieldAnalysis(path: Union[str, BinaryIO], filter: Optional[int] = 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 = None
vert_profile = None
horiz_profile = None

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 the image to determine parameters such as field edges, penumbra, and/or flatness & symmetry.

Parameters


- **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 use 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) → Union[pylinac.field_analysis.FieldResult, dict]
Present the results data and metadata as a dataclass or dict. The default return type is a dataclass.
```

results_data.as_dict() → dict
Convert the results to a dict.
```
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 ————–

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.

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: pylinac.field_analysis.Protocol
    protocol_results: dict
    centering_method: pylinac.field_analysis.Centering
    normalization_method: pylinac.core.profile.Normalization
    interpolation_method: pylinac.core.profile.Interpolation
    edge_detection_method: pylinac.core.profile.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
    bottom_penumbra_percent_mm: float
    left_penumbra_percent_mm: float
    right_penumbra_percent_mm: float

    Bases: pylinac.core.utilities.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.

    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.

    protocol = None
    protocol_results = None
    centering_method = None
    normalization_method = None
    interpolation_method = None
    edge_detection_method = None
    top_penumbra_mm = None
```python
bottom_penumbra_mm = None
left_penumbra_mm = None
right_penumbra_mm = None
geometric_center_index_x_y = None
beam_center_index_x_y = None
field_size_vertical_mm = None
field_size_horizontal_mm = None
beam_center_to_top_mm = None
beam_center_to_bottom_mm = None
beam_center_to_left_mm = None
beam_center_to_right_mm = None
cax_to_top_mm = None
cax_to_bottom_mm = None
cax_to_left_mm = None
cax_to_right_mm = None
top_position_index_x_y = None
top_horizontal_distance_from_cax_mm = None
top_vertical_distance_from_cax_mm = None
top_horizontal_distance_from_beam_center_mm = None
top_vertical_distance_from_beam_center_mm = None
left_slope_percent_mm = None
right_slope_percent_mm = None
top_slope_percent_mm = None
bottom_slope_percent_mm = None
bottom_penumbra_percent_mm = 0
left_penumbra_percent_mm = 0
right_penumbra_percent_mm = 0

class pylinac.field_analysis.DeviceFieldAnalysis(path: str, device: pylinac.field_analysis.Device)
    Bases: pylinac.field_analysis.FieldAnalysis

Field analysis using a device array.

Parameters

• path – Path to the file of the device output
• device – The array device. Currently, the Profiler is supported. See Loading Device Data.

device = None
```

5.12. Field Analysis
classmethod from_demo_image()

Load the demo image into an instance.

static run_demo() → None

Run the Field analysis demo by loading the demo device dataset, print results, and plot the profiles.

analyze(protocol: pylinac.field_analysis.Protocol = <Protocol.VARIAN: {'symmetry': {'calc': <function symmetry_point_difference>, 'unit': '%', 'plot': <function plot_symmetry_point_difference>}, 'flatness': {'calc': <function flatness_dose_difference>, 'unit': '%', 'plot': <function plot_flatness>}}, in_field_ratio: float = 0.8, slope_exclusion_ratio: float = 0.3, is_FFF: bool = False, penumbra: tuple = (20, 80), interpolation: pylinac.core.profile.Interpolation = <Interpolation.NONE: None>, interpolation_resolution_mm: float = 0.1, ground: bool = True, normalization_method: pylinac.core.profile.Normalization = <Normalization.GEOMETRIC_CENTER: 'Geometric center'>, edge_detection_method: pylinac.core.profile.Edge = <Edge.INFLECTION_HILL: 'Inflection Hill'>, edge_smoothing_ratio: float = 0.003, hill_window_ratio: float = 0.15, **kwargs) → None

Analyze the device profiles to determine parameters such as field edges, penumbra, and/or flatness & symmetry.

Parameters

- **protocol** – The analysis protocol. See Protocol Definitions for equations and options.
- **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.

- **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.

- **interpolation** – Interpolation technique to use. Must be one of the enum options of Interpolation.
- **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. 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.

• **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** – 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.

• **kwarg**s – Use these to pass parameters to custom protocol functions. See Creating & Using Custom Protocols.

class pylinac.field_analysis.Device
    Bases: enum.Enum

    2D array device Enum.

    PROFILER = {'detector spacing (mm)': 5, 'device': <class 'pylinac.core.io.SNCProfiler'>}

class pylinac.field_analysis.Protocol
    Bases: enum.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
    Bases: enum.Enum

    See Centering

    MANUAL = 'Manual'

    BEAM_CENTER = 'Beam center'

    GEOMETRIC_CENTER = 'Geometric center'

class pylinac.field_analysis.Interpolation
    Bases: enum.Enum

    Interpolation Enum

    NONE = None

    LINEAR = 'Linear'

    SPLINE = 'Spline'

5.12. Field Analysis 239
class pylinac.field_analysis.Normalization
    Bases: enum.Enum
    
    Normalization method Enum
    
    NONE = None
    GEOMETRIC_CENTER = 'Geometric center'
    BEAM_CENTER = 'Beam center'
    MAX = 'Max'

class pylinac.field_analysis.Edge
    Bases: enum.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: pylinac.core.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: pylinac.core.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: pylinac.core.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: pylinac.core.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: pylinac.core.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.

5.13 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.
5.13.1 Image Module

This module holds classes for image loading and manipulation.

```python
```

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)
- `image2` (ArrayImage) – The returns are new instances of Images.

```python
pylinac.core.image.is_image(path: Union[str, _io.BytesIO, DicomImage, ArrayImage, FileImage, LinacDicomImage, numpy.ndarray]) -> bool
```

Determine whether the path is a valid image file.

**Returns**

- `Return type` bool

```python
pylinac.core.image.retrieve_image_files(path: str) -> List[str]
```

Retrieve the file names of all the valid image files in the path.

**Returns**

- Contains strings pointing to valid image paths.

- `Return type` list

```python
```

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

Return type depends on input image.

Return type `FileImage`, `ArrayImage`, or `DicomImage`

Examples

Load an image from a file and then apply a filter:

```python
>>> 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:

```python
>>> arr = np.arange(36).reshape(6, 6)
>>> img = load(arr)  # returns an ArrayImage
```

**pylinac.core.image.load_url**

```
```

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.

**Examples**

Load multiple images:

```python
pylinac.core.image.load_multiples(image_file_list: Sequence[T_co], method: str = 'mean', stretch_each: bool = True, **kwargs) → Union[pylinac.core.image.DicomImage, pylinac.core.image.ArrayImage, pylinac.core.image.FileImage, pylinac.core.image.LinacDicomImage]
```

Combine multiple image files into one superimposed image.

**Parameters**

- `image_file_list (list)` – A list of the files to be superimposed.

- `method (str)` – 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:
```python
>>> from pylinac.core.image import load_multiples
>>> paths = ['starshot1.tif', 'starshot2.tif']
>>> superimposed_img = load_multiples(paths)
```


Bases: object

Base class for the Image classes.

**path**
The path to the image file.

**array**
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)**

Load an instance from multiple image items. See `load_multiples()`.

**center**
Return the center position of the image array as a Point.

**physical_shape**
The physical size of the image in mm.

**plot(ax: matplotlib.axes._axes.Axes = None, show: bool = True, clear_fig: bool = False, **kwargs)**

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.

**filter(size: Union[float, int] = 0.05, kind: str = 'median') → None**

Filter the profile.

**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.

**remove_edges** *(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.

**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.

**threshold** *(threshold: int, 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.


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

**Return type** *ArrayImage*

**dist2edge_min** *(point: Union[pylinac.core.geometry.Point, Tuple]) → float*

Calculates minimum distance from given point to image edges.

Parameters **point** (*geometry.Point, tuple*) –

Returns
Return type float

`ground()` → float

Ground the profile 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 The amount subtracted from the image.

Return type float

`normalize(norm_val: Union[str, int, float] = 'max')` → None

Normalize the image values to the given value.

**Parameters**

- `norm_val` *(str, number)* – If a string, must be ‘max’, which normalizes the values to the maximum value. If a number, normalizes all values to that number.

`check_inversion(box_size: int = 20, position: Sequence[T_co] = (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=(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).


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** – The calculated gamma map.

**Return type**  
numpy.ndarray

**See also:**
equate_images()

dtype=None, dpi: Union[int, float] = None, sid: Union[int, float] = None)

Bases: pylinac.core.image.BaseImage

An image from a DICOM RTImage file.

**metadata**
The dataset of the file as returned by pydicom without pixel data.

**Type**  
pydicom Dataset

**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.

**save**(filename: str) → str

Save the image instance back out to a .dcm file.

**Returns**

**Return type**  
A string pointing to the new filename.

**sid**
The Source-to-Image in mm.

**dpi**
The dots-per-inch of the image, defined at isocenter.

**dpmm**
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.

**cax**
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, use_filenames: bool = False)
Bases: pylinac.core.image.DicomImage

DICOM image taken on a linac. Also allows passing of gantry/coll/couch values via the filename.

gantry_angle
Gantry angle of the irradiation.

collimator_angle
Collimator angle of the irradiation.

couch_angle
Couch angle of the irradiation.

class pylinac.core.image.FileImage(path: str, *, dpi: Union[int, float] = None, sid: Union[int, float] = None, dtype=None)
Bases: pylinac.core.image.BaseImage

An image from a “regular” file (.tif, .jpg, .bmp).

info
The info dictionary as generated by Pillow.

Type dict

sid
The SID value as passed in upon construction.

Type float

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.

dpi
The dots-per-inch of the image, defined at isocenter.

dpmm
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: numpy.array, *, dpi: Union[int, float] = None, sid: Union[int, float] = None, dtype=none)
Bases: pylinac.core.image.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.

**dpmm**

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.

**dpi**

The dots-per-inch of the image, defined at isocenter.

**class** pylinac.core.image.DicomImageStack *(folder: str, dtype=None, min_number: int = 39, check_uid: bool = True)*

**Bases:** object

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.

**images**

Holds instances of DicomImage. Can be accessed via index; i.e. self[0] == self.images[0].

**Type** list

**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, dtype=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.

**static** is_CT_slice *(file: str) → bool*

Test if the file is a CT Image storage DICOM file.
plot (slice: int = 0) → None
Plot a slice of the DICOM dataset.

Parameters slice (int) – The slice to plot.

metadata
The metadata of the first image; shortcut attribute. Only attributes that are common throughout the stack should be used, otherwise the individual image metadata should be used.

5.13.2 Geometry Module

Module for classes that represent common geometric objects or patterns.

pylinac.core.geometry.tan (degrees: Union[int, float]) → float
Calculate the tangent of the given degrees.

pylinac.core.geometry.cos (degrees: Union[int, float]) → float
Calculate the cosine of the given degrees.

pylinac.core.geometry.sin (degrees: Union[int, float]) → float
Calculate the sine of the given degrees.

class pylinac.core.geometry.Point (x: Union[int, float, tuple, Point] = 0, y: Union[int, float] = 0, z: Union[int, float] = 0, idx: Optional[int] = None, value: Union[int, 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: Union[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) → numpy.array
Return the point as a numpy array.

class pylinac.core.geometry.Circle (center_point: Union[pylinac.core.geometry.Point, Iterable[T_co]] = (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.

`diameter`

Get the diameter of the circle.

`plot2axes` (axes, edgecolor: str = 'black', fill: bool = False) → 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.

```python
class pylinac.core.geometry.Vector:
    Bases: object
    A vector with x, y, and z coordinates.
    
    as_scalar() → float
    Return the scalar equivalent of the vector.
    
    distance_to (thing: pylinac.core.geometry.Circle, pylinac.core.geometry.Point) → float
    Calculate the distance to the given point.
    
    Parameters
    thing (Circle, Point, 2 element iterable) – The other point to calculate distance to.

distance_to = pylinac.core.geometry.vector_is_close

    vector_is_close(vector1: pylinac.core.geometry.Vector, vector2: pylinac.core.geometry.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:
    Bases: object
    A line that is represented by two points or by m*x+b.

    m
    Return the slope of the line.
    m = (y1 - y2)/(x1 - x2)
    From: http://www.purplemath.com/modules/slope.htm

    b
    Return the y-intercept of the line.
    b = y - m*x
```

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
\[ y(x) \rightarrow \text{float} \]
Return y-value along line at position x.

\[ x(y) \rightarrow \text{float} \]
Return x-value along line at position y.

center
Return the center of the line as a Point.

length
Return length of the line, if finite.

distance_to \( (point) \rightarrow \text{float} \)
Calculate the minimum distance from the line to a point.
Equations are from here: http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html #14

Parameters
\begin{itemize}
\item \textbf{point (Point, iterable)} – The point to calculate distance to.
\end{itemize}

plot2axes \( (axes: \text{matplotlib.axes.Axes}, \text{width: } \text{Union[int, float]} = 1, \text{color: } \text{str} = \text{'w'}) \rightarrow \text{None} \)
Plot the line to an axes.

Parameters
\begin{itemize}
\item \textbf{axes (matplotlib.axes.Axes)} – An MPL axes to plot to.
\item \textbf{color (str)} – The color of the line.
\end{itemize}

class \texttt{pylinac.core.geometry.Rectangle} \( (width: \text{float}, \text{height: } \text{float}, \text{center: } \text{Union[pylinac.core.geometry.Point, Tuple], as_int: bool = False}) \)
Bases: \texttt{object}
A rectangle with width, height, center Point, top-left corner Point, and bottom-left corner Point.

Parameters
\begin{itemize}
\item \textbf{width (number)} – Width of the rectangle.
\item \textbf{height (number)} – Height of the rectangle.
\item \textbf{center (Point, iterable, optional)} – Center point of rectangle.
\item \textbf{as_int (bool)} – If False (default), inputs are left as-is. If True, all inputs are converted to integers.
\end{itemize}

\texttt{br\_corner}
The location of the bottom right corner.

\texttt{bl\_corner}
The location of the bottom left corner.

\texttt{tl\_corner}
The location of the top left corner.

\texttt{tr\_corner}
The location of the top right corner.

\texttt{plot2axes \( (axes: \text{matplotlib.axes.Axes}, \text{edgecolor: } \text{str} = \text{'black'}, \text{angle: } \text{float} = 0.0, \text{fill: } \text{bool} = \text{False}, \text{alpha: } \text{float} = 1, \text{facecolor: } \text{str} = \text{'g'}, \text{label=None}) \)}
Plot the Rectangle to the axes.

Parameters
\begin{itemize}
\item \textbf{axes (matplotlib.axes.Axes)} – An MPL axes to plot to.
\item \textbf{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.

### 5.13.3 Profile Module

Module of objects that resemble or contain a profile, i.e. a 1 or 2-D f(x) representation.

```python
pylinac.core.profile.stretch(array: numpy.ndarray, min: int = 0, max: int = 1, fill_dtype: Optional[numpy.dtype] = None) → numpy.ndarray

'Stretch' the profile to fit a new min and max value and interpolate in between. From: http://www.labri.fr/perso/nrougier/teaching/numpy.100/exercise #17
```

**Parameters**

• `array (numpy.ndarray)` – The numpy array to stretch.

• `min (number)` – The new minimum of the values.

• `max (number)` – The new maximum value.

• `fill_dtype (numpy data type)` – If None (default), the array will be stretched to the passed min and max. If a numpy data type (e.g. np.int16), the array will be stretched to fit the full range of values of that data type. If a value is given for this parameter, it overrides `min` and `max`.

```python
class pylinac.core.profile.ProfileMixin
Bases: object

A mixin to provide various manipulations of 1D profile data.

invert() → None
Invert (imcomplement) the profile.

normalize(norm_val: Union[str, int, float] = 'max') → None
Normalize the profile to the given value.

Parameters

• `norm_val (str, number)` – If a string, must be ‘max’, which normalizes the values to the maximum value. If a number, normalizes all values to that number.

stretch(min: Union[int, float] = 0, max: Union[int, 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.

ground() → float
Ground the profile such that the lowest value is 0.

Returns The minimum value that was used as the grounding value.

Return type float

filter(size: Union[int, 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.

```python
class pylinac.core.profile.Interpolation
    Bases: enum.Enum

    Interpolation Enum

    NONE = None
    LINEAR = 'Linear'
    SPLINE = 'Spline'

class pylinac.core.profile.Normalization
    Bases: enum.Enum

    Normalization method Enum

    NONE = None
    GEOMETRIC_CENTER = 'Geometric center'
    BEAM_CENTER = 'Beam center'
    MAX = 'Max'

class pylinac.core.profile.Edge
    Bases: enum.Enum

    Edge detection Enum

    FWHM = 'FWHM'
    INFLECTION_DERIVATIVE = 'Inflection Derivative'
    INFLECTION_HILL = 'Inflection Hill'

```

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.**

```python
generic_center() → dict

The geometric center (i.e. the device center)
```

```python
beam_center() → dict

The center of the detected beam. This can account for asymmetries in the beam position (e.g. offset jaws)
```

```python
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.
```

```python
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**.

**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') → Union[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.

**plot() → None**

Plot the profile.

**class pylinac.core.profile.MultiProfile(values: Union[numpy.ndarray, Sequence[T_co]])**

**Bases:** pylinac.core.profile.ProfileMixin

A class for analyzing 1-D profiles that contain multiple signals. Methods are mostly for finding & filtering the signals, peaks, valleys, etc. Profiles with a single peak (e.g. radiation beam profiles) are better suited by the SingleProfile class.

**values**

The array of values passed in on instantiation.

**Type** ndarray

**peaks**

List of Points, containing value and index information.

**Type** list

**valleys**

Same as peaks, but for valleys.

**Type** list

**Parameters values(iterable) – Array of profile values.**

**plot(ax: Optional[matplotlib.axes._axes.Axes] = None) → None**

Plot the profile.

**Parameters ax(plt.Axes) – An axis to plot onto. Optional.**
find_peaks (threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple = (0.0, 1.0), peak_sort='prominences') \rightarrow Tuple[numpy.ndarray, numpy.ndarray]

Find the peaks of the profile using a simple maximum value search. This also sets the peaks attribute.

Parameters

- **threshold (int, float)** – The value the peak must be above to be considered a peak. This removes “peaks” that are in a low-value region. If passed an int, the actual value is the threshold. E.g. when passed 15, any peak less with a value <15 is removed. If passed a float, it will threshold as a percent. Must be between 0 and 1. E.g. when passed 0.4, any peak <40% of the maximum value will be removed.

- **min_distance (int, float)** – If passed an int, parameter is the number of elements apart a peak must be from neighboring peaks. If passed a float, must be between 0 and 1 and represents the ratio of the profile to exclude. E.g. if passed 0.05 with a 1000-element profile, the minimum peak width will be 0.05*1000 = 50 elements.

- **max_number (int, None)** – Specify up to how many peaks will be returned. E.g. if 3 is passed in and 5 peaks are found, only the 3 largest peaks will be returned. If None, no limit will be applied.

- **search_region (tuple of ints, floats, or both)** – The region within the profile to search. The tuple specifies the (left, right) edges to search. This allows exclusion of edges from the search. If a value is an int, it is taken as is. If a float, must be between 0 and 1 and is the ratio of the profile length. The left value must be less than the right.

Returns **indices** – The indices and values of the peaks.

Return type **ndarray, values, ndarray**

find_valleys (threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple = (0.0, 1.0)) \rightarrow Tuple[numpy.ndarray, numpy.ndarray]

Find the valleys (minimums) of the profile using a simple minimum value search.

Returns **indices** – The indices and values of the valleys.

Return type **ndarray, values, ndarray**

See also:

find_peaks() : Further parameter info.

find_fwxm_peaks (threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple = (0.0, 1.0), peak_sort: str = 'prominences', required_prominence=None) \rightarrow Tuple[numpy.ndarray, numpy.ndarray]

Find peaks using the center of the FWXM (rather than by max value).

Parameters **x (int, float)** – The Full-Width-X-Maximum desired. E.g. 0.7 will return the FW70%M. Values must be between 0 and 100.

See also:

find_peaks() Further parameter info
class pylinac.core.profile.CircleProfile(center: pylinac.core.geometry.Point, radius: Union[int, float], image_array: numpy.ndarray, start_angle: Union[float, int] = 0, ccw: bool = True, sampling_ratio: float = 1.0)

Bases: pylinac.core.profile.MultiProfile, pylinac.core.geometry.Circle

A profile in the shape of a circle.

image_array
The 2D image array.

Type ndarray

start_angle
Starting position of the profile in radians; 0 is right (0 on unit circle).

Type int, float

ccw
How the profile is/was taken; clockwise or counter-clockwise.

Type bool

Parameters

- **image_array**(ndarray) – The 2D image array.
- **start_angle**(int, float) – Starting position of the profile in radians; 0 is right (0 on unit circle).
- **ccw**(bool) – If True (default), the profile will proceed counter-clockwise (the direction on the unit circle). If False, will proceed clockwise.
- **sampling_ratio**(float) – The ratio of pixel sampling to real pixels. E.g. if 1.0, the profile will have approximately the same number of elements as was encountered in the profile. A value of 2.0 will sample the profile at 2x the number of elements.

See also:

**Circle** : Further parameter info.

size
The elemental size of the profile.

x_locations
The x-locations of the profile values.

y_locations
The x-locations of the profile values.

find_peaks(threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple[float, float] = (0.0, 1.0)) → Tuple[numpy.ndarray, numpy.ndarray]

Overloads Profile to also map peak locations to the image.

find_valleys(threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple[float, float] = (0.0, 1.0)) → Tuple[numpy.ndarray, numpy.ndarray]

Overload Profile to also map valley locations to the image.

find_fwxm_peaks(threshold: Union[float, int] = 0.3, min_distance: Union[float, int] = 0.05, max_number: int = None, search_region: Tuple[float, float] = (0.0, 1.0)) → Tuple[numpy.ndarray, numpy.ndarray]

Overloads Profile to also map the peak locations to the image.
**roll** *(amount: int) → None*

Roll the profile and x and y coordinates.

**plot2axes** *(axes: matplotlib.axes.Axes, None, edgecolor: str = 'black', fill: bool = False, plot_peaks: bool = True) → None*

Plot the circle to an axes.

**Parameters**

- **axes** *(matplotlib.Axes, None) – The axes to plot on. If None, will create a new figure of the image array.*
- **edgecolor** *(str) – Color of the Circle; must be a valid matplotlib color.*
- **fill** *(bool) – Whether to fill the circle. matplotlib keyword.*
- **plot_peaks** *(bool) – If True, plots the found peaks as well.*

**class** *pylinac.core.profile.CollapsedCircleProfile*

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.

**size**

The elemental size of the profile.

**plot2axes** *(axes: matplotlib.axes.Axes, 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.

**pylinac.core.profile.find_peaks** *(values: numpy.ndarray, threshold: Union[float, int] = -inf, peak_separation: Union[float, int] = 0, max_number: int = None, fwxm_height: float = 0.5, min_width: int = 0, search_region: Tuple[float, float] = (0.0, 1.0), peak_sort='prominences', required_prominence=None) → Tuple[numpy.ndarray, dict]*

Find the peaks of a 1D signal. Heavily relies on the scipy implementation.

**Parameters**
• values (array-like) – Signal values to search for peaks within.
• threshold (int, float) – The value the peak must be above to be considered a peak. This removes “peaks” that are in a low-value region. If passed an int, the actual value is the threshold. E.g. when passed 15, any peak less with a value <15 is removed. If passed a float, it will threshold as a percent. Must be between 0 and 1. E.g. when passed 0.4, any peak <40% of the maximum value will be removed.
• peak_separation (int, float) – If passed an int, parameter is the number of elements apart a peak must be from neighboring peaks. If passed a float, must be between 0 and 1 and represents the ratio of the profile to exclude. E.g. if passed 0.05 with a 1000-element profile, the minimum peak width will be 0.05*1000 = 50 elements.
• max_number (int, None) – Specify up to how many peaks will be returned. E.g. if 3 is passed in and 5 peaks are found, only the 3 largest peaks will be returned.
• fwxm_height (float) – The relative height at which a FWXM calculation is performed. Although this function finds simple max values, the underlying function can provide fwxm information as well.
• min_width (int) – The minimum width of the peak.
• search_region (tuple) – The search region to use within the values. Using between 0 and 1 will convert to a ratio of the indices. E.g. to search the middle half of the passed values, use (0.25, 0.75). Using ints above 1 will use the indices directly. E.g. (33, 71) will search between those two indices.

Returns

• peak_idxs (numpy.array) – The indices of the peaks found.
• peak_props (dict) – A dict containing contextual peak data.

5.13.4 I/O Module

I/O helper functions for pylinac.

pylinac.core.io.is_dicom (file: str) → 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) → 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) → pydicom.dataset.FileDataset

Read and return the DICOM dataset.

Parameters file (str) – The path to the file.
pylinac.core.io.is_zipfile(file: str) → bool
Wrapper function for detecting if file is a true ZIP archive

class pylinac.core.io.TemporaryZipDirectory(zfile)
    Bases: tempfile.TemporaryDirectory
    Creates a temporary directory that unpacks a ZIP archive.
    Parameters zfile (str) – String that points to a ZIP archive.

pylinac.core.io.retrieve_filenames(directory: str, func: Callable = 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(url: str, force: bool = False) → str
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 url (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
    Return type  bool

pylinac.core.io.get_url(url: str, destination: str = 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 – The location of the downloaded file.
    Return type  str

Notes
Progress bar use/example adapted from tqdm documentation: https://github.com/tqdm/tqdm
class pylinac.core.io.SNCProfiler(path: str, 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** –
- **bias_row** –
- **calibration_row** –
- **data_row** –
- **data_columns** – The range of columns that the data is in. Usually, there are some columns before and after the real data.

```
pylinac.core.profile.SingleProfile, pylinac.core.profile.SingleProfile, pylinac.core.profile.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.

5.13.5 ROI Module

```
pylinac.core.roi.bbox_center(region: skimage.measure._regionprops.RegionProperties) → pylinac.core.geometry.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**

Return type: **Point**

```py
class pylinac.core.roi.Contrast
Bases: enum.Enum
```

Contrast calculation technique. See Visibility

- **MICHELSON** = 'Michelson'
- **WEBER** = 'Weber'
- **RATIO** = 'Ratio'

```py
class pylinac.core.roi.DiskROI(array: numpy.ndarray, angle: Union[float, int], roi_radius: Union[float, int], dist_from_center: Union[float, int], phantom_center: Union[Tuple, pylinac.core.geometry.Point])
```

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.

**pixel_value**
The median pixel value of the ROI.

**std**
The standard deviation of the pixel values.

**circle_mask** () → numpy.ndarray
Return a mask of the image, only showing the circular ROI.

**plot2axes** (*axes=None, edgecolor: str = 'black', fill: bool = False*) → 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.


**Bases**: pylinac.core.roi.DiskROI

A class for analyzing the low-contrast disks.

**Parameters**

• **contrast_threshold** (*float, int*) – The threshold for considering a bubble to be “seen”.

• **signal_to_noise**
The signal to noise ratio.

• **contrast_to_noise**
The contrast to noise ratio of the ROI

• **contrast**

**Type**
The contrast of the bubble. Uses the contrast method passed in the constructor. See https
cnr_constant
The contrast-to-noise value times the bubble diameter.

• **visibility**
Importantly, the Rose model is not applicable for high-contrast use cases.

**Type** The visual perception of CNR. Uses the model from A Rose

**contrast_constant**
The contrast value times the bubble diameter.

**passed**
Whether the disk ROI contrast passed.

**passed_visibility**
Whether the disk ROI’s visibility passed.

**passed_contrast_constant**
Boolean specifying if ROI pixel value was within tolerance of the nominal value.

**passed_cnr_constant**
Boolean specifying if ROI pixel value was within tolerance of the nominal value.

**plot_color**
Return one of two colors depending on if ROI passed.

**plot_color_constant**
Return one of two colors depending on if ROI passed.

**plot_color_cnr**
Return one of two colors depending on if ROI passed.

```python
class pylinac.core.roi.HighContrastDiskROI(array: numpy.ndarray, angle: float, roi_radius: float, dist_from_center: float, phantom_center: Union[tuple, pylinac.core.geometry.Point], contrast_threshold: float)
```

**Parameters**
- **contrast_threshold** (*float, int*) – The threshold for considering a bubble to be “seen”.

**max**
The max pixel value of the ROI.

**min**
The min pixel value of the ROI.

```python
class pylinac.core.roi.RectangleROI(array, width, height, angle, dist_from_center, phantom_center)
```

**Bases:** *pylinac.core.roi.DiskROI*

**Class that represents a rectangular ROI.**

**pixel_array**
The pixel array within the ROI.

## 5.13.6 Mask Module

Module for processing “masked” arrays, i.e. binary images.
Get the bounding box values of an ROI in a 2D array.

Return the ratio of filled pixels to empty pixels in the ROI bounding box.

For example a solid square would be 1.0, while a sold circle would be ~0.785.

Determine the width/height ratio of the ROI

5.13.7 Utilities Module

Utility functions for pylinac.

Convert a value to an enum representation from an enum value if needed

class pylinac.core.utilities.ResultBase
    Bases: object

    pylinac_version = None
    date_of_analysis = None

Delete all demo files, image classifiers, etc from the demo folder

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.

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

Return type bool

Round a number to the given number of decimals. Fixes small floating number errors.

Check whether the passed object is numeric in any sense.
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: Union[Type[int], Type[float], Type[str]], num_values: int = 1, cursor_shift: int = 0) → Union[int, float, str, numpy.ndarray]
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.

pylinac.core.utilities.open_path(path: str) → None
Open the specified path in the system default viewer.

pylinac.core.utilities.file_exists(filename: str) → str
Check if the file exists and if it does add a timestamp

5.13.8 Decorators Module

pylinac.core.decorators.timethis(func)
Report execution time of function.

pylinac.core.decorators.lru_cache(*lru_args, **lru_kwargs)
https://stackoverflow.com/a/33672499

5.14 Image Generator

5.14.1 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.

### 5.14.2 Quick Start

The basics to get started are to import the image simulators and layers from pylinac and add the layers as desired.

```python
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
plt.imshow(as1000.image)
```
5.14.3 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.

5.14.4 Extending Layers & Simulators

This module was 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:

```python
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:

```python
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())
...```

5.14.5 Examples

Let’s make some images!

Simple Open Field

```python
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

def 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

```python
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))
as1200.add_layer(PerfectBBLayer(bb_size_mm=5))
as1200.add_layer(GaussianFilterLayer(sigma_mm=2))
as1200.add_layer(RandomNoiseLayer(sigma=2000))
# plot the generated image
plt.imshow(as1200.image)
```

VMAT DRMLC

```python
from matplotlib import pyplot as plt
from pylinac.core.image_generator import AS1200Image
from pylinac.core.image_generator.layers import FilteredFieldLayer,
   GaussianFilterLayer

as1200 = AS1200Image()
```

(continues on next page)
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()

% matplotlib inline

import matplotlib.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())

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.

```python
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.add_layer(GaussianFilterLayer())
plt.imshow(as1200.image)
plt.show()

5.14. Image Generator

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:

```python
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'

# 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]])

5.14.6 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.

5.14.7 API Documentation

Layers

class pylinac.core.image_generator.layers.PerfectConeLayer(cone_size_mm: float = 10, cax_offset_mm: Tuple[float, float] = (0, 0), alpha: float = 1.0)

Bases: pylinac.core.image_generator.layers.Layer

A cone without flattening filter effects
apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.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: Tuple[float, float] = (0, 0), 
alpha: float = 1.0, filter_magnitude: float = 0.4, 
filter_sigma_mm: float = 80, 
penumbra_mm: float = 2)

Bases: pylinac.core.image_generator.layers.PerfectConeLayer
A cone with flattening filter effects.

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.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: Tuple[float, float] = (10, 10), 
cax_offset_mm: Tuple[float, float] = (0, 0), 
alpha: float = 1.0)

Bases: pylinac.core.image_generator.layers.Layer
A square field without flattening filter effects

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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: Tuple[float, float] = (10, 10), 
cax_offset_mm: Tuple[float, float] = (0, 0), 
alpha: float = 1.0, gaussian_height: float = 0.03, 
gaussian_sigma_mm: float = 32)

Bases: pylinac.core.image_generator.layers.PerfectFieldLayer
A square field with flattening filter effects

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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: Tuple[Union[float, int], Union[float, int]] = (10, 10),
cax_offset_mm: Tuple[Union[float, int], Union[float, int]] = (0, 0),
alpha: float = 1.0, gaussian_height: float = 0.4, gaussian_sigma_mm: float = 80)
Bases: pylinac.core.image_generator.layers.FilteredFieldLayer
A square field with flattening filter free (FFF) effects

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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: Tuple[Union[float, int], Union[float, int]] = (0, 0),
alpha: float = -0.5)
Bases: pylinac.core.image_generator.layers.PerfectConeLayer
A BB-like layer. Like a cone, but with lower alpha (i.e. higher opacity)

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.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: pylinac.core.image_generator.layers.Layer
A Gaussian filter. Simulates the effects of scatter on the field

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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 = 0.01)
Bases: pylinac.core.image_generator.layers.Layer
A salt and pepper noise, simulating dark current

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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: pylinac.core.image_generator.layers.Layer
A constant layer. Can be used to simulate scatter or background.

apply (image: numpy.ndarray, pixel_size: float, mag_factor: float) → numpy.ndarray
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: pylinac.core.image_generator.simulators.Simulator
Simulates an AS500 EPID image.

**Parameters**

- **sid** – Source to image distance in mm.

**generate_dicom**

```python
(file_out_name: str, gantry_angle: float = 0.0, coll_angle: float = 0.0, table_angle: float = 0.0) → None
```

Generate a DICOM file with the constructed image (via add_layer)

**add_layer**

```python
(layer: pylinac.core.image_generator.layers.Layer) → None
```

Add a layer to the image

**class**

```python
pylinac.core.image_generator.simulators.AS1000Image(sid: float = 1500)
```

Bases:

```python
pylinac.core.image_generator.simulators.Simulator
```

Simulates an AS1000 EPID image.

**Parameters**

- **sid** – Source to image distance in mm.

**generate_dicom**

```python
(file_out_name: str, gantry_angle: float = 0.0, coll_angle: float = 0.0, table_angle: float = 0.0) → None
```

Generate a DICOM file with the constructed image (via add_layer)

**add_layer**

```python
(layer: pylinac.core.image_generator.layers.Layer) → None
```

Add a layer to the image

**class**

```python
pylinac.core.image_generator.simulators.AS1200Image(sid: float = 1500)
```

Bases:

```python
pylinac.core.image_generator.simulators.Simulator
```

Simulates an AS1200 EPID image.

**Parameters**

- **sid** – Source to image distance in mm.

**generate_dicom**

```python
(file_out_name: str, gantry_angle: float = 0.0, coll_angle: float = 0.0, table_angle: float = 0.0) → None
```

Generate a DICOM file with the constructed image (via add_layer)

**add_layer**

```python
(layer: pylinac.core.image_generator.layers.Layer) → None
```

Add a layer to the image

**Helpers**

```python
pylinac.core.image_generator.utils.generate_picketfence(simulator:
pylinac.core.image_generator.simulators.Simulator,
field_layer:
Type[Union[pylinac.core.image_generator.layers.FilterFreeFieldLayer,
pylinac.core.image_generator.layers.FilteredFieldLayer,
pylinac.core.image_generator.layers.PerfectFieldLayer]],
file_out: str, final_layers: List[pylinac.core.image_generator.layers.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.UP_DOWN: 'Up-Down'>, picket_offset_error: Optional[Sequence[T_co]] = 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.

```python
pylinac.core.image_generator.utils.generate_winstonlutz(
simulator: pylinac.core.image_generator.simulators.Simulator,
field_layer: Type[pylinac.core.image_generator.layers.Layer],
dir_out: str,
field_size_mm: Tuple[float, float] = (30, 30), final_layers: Optional[List[pylinac.core.image_generator.layers.Layer]] = None, bb_size_mm: float = 5, offset_mm_left: float = 0, offset_mm_up: float = 0, offset_mm_in: float = 0, image_axes: List[Tuple[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)
```

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.

```python
generate_winstonlutz_cone(simulator: pylinac.core.image_generator.simulators.Simulator, cone_layer: Union[Type[pylinac.core.image_generator.layers.FilterFreeConeLayer], Type[pylinac.core.image_generator.layers.PerfectConeLayer]], dir_out: str, cone_size_mm: float = 17.5, final_layers: Optional[List[pylinac.core.image_generator.layers.Layer]] = None, bb_size_mm: float = 5, offset_mm_left: float = 0, offset_mm_up: float = 0, offset_mm_in: float = 0, image_axes: List[Tuple[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 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.

## 5.15 Topics

### 5.15.1 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.

**How data is loaded**

Pylinac uses the excellent pydicom library to load DICOM images. The pydicom dataset is actually stored in pylinac images under the `metadata` attribute, so if want to access them, they’re there.

**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

Assigned pixel values now have the following logic:

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: 

\[ P_{corrected} = \text{Sign} \times \text{Slope} \times P_{raw} + \text{Intercept} \]

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:

\[ P_{corrected} = \text{Slope} \times P_{raw} + \text{Intercept}. \]

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: 

\[ \text{newarray} = -\text{oldarray} + \max(\text{oldarray}) + \min(\text{oldarray}) \]

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 incomplimented, 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.

5.15.2 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.

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 contrast, aka visibility. See here for more comparisons: https://en.wikipedia.org/wiki/Display_contrast

\[
\frac{I_{\text{max}} - I_{\text{min}}}{I_{\text{max}} + I_{\text{min}}} \quad \frac{\max (I_{\text{max}} - I_{\text{min}})}{I_{\text{max}} + I_{\text{min}}}
\]

where \( I = 1, \ldots, n \) line pair ROIs.

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.

Note: The combination of low contrast and ROI size 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 (default; good choice)

\[
\frac{I_{\text{mean}} - R_{\text{mean}}}{I_{\text{mean}} + R_{\text{mean}}}
\]

Weber

\[
\frac{I_{\text{mean}} - R_{\text{mean}}}{I_{\text{mean}}}
\]
Ratio

\[ \frac{I_{\text{mean}}}{R_{\text{mean}}} \]

Visibility

Visibility is the ability for humans to detect signal against noise. Visibility is a component of low contrast detectability. Typically, 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 in pylinac is a derivation of the Rose model, defined here as:

\[ V_{\text{isibility}}(I) = \frac{\text{Contrast}(I)}{\sqrt{\text{Area}(I) \times DQE(I)}} = \frac{\text{Contrast}(I)}{\sqrt{\pi \times I^2_{\text{radius}}}} \]

where contrast is an option from the low contrast methods.

Note: What is meant by “noise” is unclear in the literature. Technically, it was meant to be the detective quantum efficiency. For simplicity and easy of understanding, the standard deviation works.

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 tolerance in the algorithm. Thus, visibility is a very specific number that depends on the size of the sampling ROI.

Contrast-to-noise ratio

The contrast to noise ratio (CNR) is defined as follows:

\[ \text{CNR}(I) = \frac{\text{Contrast}(I)}{\text{noise}(I)} = \frac{\text{Contrast}(I)}{\text{std}(I)} \]

where contrast is an option from the low contrast methods.

5.15.3 Modulation Transfer Function (MTF)

The MTF is used in CBCT and planar imaging metrics to describe high-contrast characteristics of the imaging system. An excellent introduction is here: https://www.edmundoptics.com/knowledge-center/application-notes/optics/introduction-to-modulation-transfer-function/ In pylinac, MTF is calculated using equation 3 of the above reference:

\[ \text{contrast} = \frac{I_{\text{max}} - I_{\text{min}}}{I_{\text{max}} + I_{\text{min}}} \]

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 \( I_{\text{max}} \) and \( I_{\text{min}} \) 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.
5.16 Troubleshooting

5.16.1 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, email me, or file an issue on Github - You may have found a bug and it needs fixing!
  - Forum
  - Email
  - Issue tracker

5.16.2 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.

5.17 Contributing

There are several ways you can contribute to the pylinac project no matter your skill level. Read on for more info.

5.17.1 Submitting bugs

The easiest way to contribute is to report bugs. Submit bugs via a Github issue here.

5.17.2 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.
5.17.3 Suggesting ideas

Ideas are always welcome (though they might not get implemented). You can submit new ideas here.

5.17.4 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).
- Evaluate the work involved. This includes reviewing any existing modules or 3rd party tools that can help solve the problem.
- Start a new branch for the feature.
- 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.
- Write the test framework. Same as the regular framework but defines the parameters and endpoints for the module.
- Once the framework is agreed upon the code can start flowing. Get to it!
- If you’re unsure, just ask.

5.18 Changelog

5.18.1 v 3.1.0

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

- 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.

5.18.2 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.

```python
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:

```python
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.
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)
...

cf = CatPhan504(...)  
cf.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 Topics.

- 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. See [Loading Device Data](#).
- 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.SNCPprofiler. 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.

```python
from pylinac.core.io import SNCPprofiler
snc = SNCPprofiler("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 `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”.

5.18. Changelog
Machine logs

• #161 Trajectory logs v4.0 are now supported

5.18.3 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.
5.18.4 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:

```python
# 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 `rois`.

- Due to the above refactor, you may notice small differences in the contrast constant value and thus the ROIs “seen”.

5.18. Changelog
- 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.

#### 5.18.5 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.

#### 5.18.6 v 2.3.1

### Bug Fixes

- #281 The ct module had a wrong usage of the new MTF module that caused a break.
5.18.7 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.

### 5.18.8 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*.

**5.18.9 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 formula 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 slightly 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.

5.18.10 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. Behavior was able to be changed internally to handle this case without an API change.

5.18.11 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 unknowingly affected. Behavior has been reverted to pre-2.2.2 behavior and documentation changed.

5.18.12 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.

5.18.13 V 2.2.3

Bug Fixes

• #158 Catphan roll determination algorithm has slightly widened the air bubble-finding criterion.
5.18.14 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.

5.18.15 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.

5.18.16 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: flatsym_module.
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.

5.18.17 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 \texttt{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 \texttt{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 \texttt{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.

```python
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.

### 5.18.18 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:

```python
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:

  ```python
  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 Accessing data 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 \textit{kq} function now accepts an \textit{r\_50} parameter to calculate kQ based on the above data.

**Core Modules**

- The \textit{Image} class has been fully deprecated and is no longer available. Use the functions available in the \texttt{pylinac.core.image} module instead. See the version 1.4.0 release notes for further details.
- The \textit{remove_edges} method has been deprecated and is now an alias for \textit{crop}. The \textit{crop} method should be used instead. Parameters are exactly the same.

**5.18.19 V 1.9.0**

**General Changes**

- This release introduces PDF reports for most major modules. All classes with this functionality have been given a \texttt{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. \texttt{pylinac.starshot.Starshot.publish_pdf()} or \texttt{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 task\_scheduler 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 \texttt{zip\_after} argument in the \texttt{analyze} method.

**Watcher/Processor**

- The \texttt{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, \texttt{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 \texttt{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 \texttt{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.
5.18.20 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:

```yaml
# other sections
...

catphan:  # not cbct:
  ...
...
```

CBCT/CatPhan

- The Python file/module has been renamed to ct from cbct. E.g.:

```python
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:

```python
# 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:

```python
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.

5.18.21 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.

5.18.22 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.

5.18.23 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

5.18.24 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.

5.18.25 V 1.5.5

• (#65) - Fixed the FlatSym demo file usage.

5.18.26 V 1.5.4

• (#64) - Fixed the Picket Fence offset from CAX value, which previously were all the same value.
5.18.27 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.

5.18.28 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:

  ```bash
  $ 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:

  ```bash
  $ 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.

  ```bash
  $ conda config --add channels jrkerns
  ```

  Then, installation and upgrading is as simple as:

  ```bash
  $ 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.

  ```python
  # 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:

```python
log = MachineLog.from_demo_dynalog()
# calculate & view total actual fluence
logfluence.actual.calc_map()
logfluence.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:

```python
log = MachineLog.from_demo_trajectorylog()
# old way
logfluence.gamma.calc_map(threshold=10)  # <- this indicates 10% threshold
# new way
logfluence.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.

**5.18.29 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.

**5.18.30 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:

```python
from pylinac.core.image import Image
img = Image.load('my/file.dcm')
```

One should now do:

```python
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:

```python
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:

```python
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.:

```python
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.

5.18.31 V 1.3.1

- (#46) - Fixes CBCT analysis where there is a ring artifact outside the phantom. Incidentally, analysis is sped up by ~10%.

5.18.32 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.

**5.18.33 V 1.2.2**

- (#45) Fixes various crashes of Leeds analysis.

**5.18.34 V 1.2.1**

- (#44) Fixed a stale wheel build causing pip install to install v1.1.

**5.18.35 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:

```python
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.

5.18.36 V 1.1.1

• Winston-Lutz demo images were not included in the pypi package.

5.18.37 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()`.
5.18.38 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).

5.18.39 V 1.0.2

- Fixes #38. MachineLog fluence calculations would crash if there was no beam-on snapshots (e.g. kV images).

5.18.40 V 1.0.1

- Fixes #37. Reading in a trajectory log txt file with a blank line caused a crash.

5.18.41 V 1.0.0

General Changes

- This release debuts the new interactive plotting for certain figures. Quickly, matplotlib line/bar plots (although 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.

5.18.42 V 0.9.1

- Fixed a bug with the log analyzer treatment type property.
5.18.43 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.

5.18.44 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.

5.18.45 V 0.8.1

- Fixed an import bug

5.18.46 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.

5.18.47 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.

5.18.48 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.

5.18.49 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 \texttt{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 \texttt{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.

5.18.50 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`

5.18.51 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.

5.18.52 V 0.4.1

- Batch processing of logs added via a new class.
- ~4x speedup of fluence calculations.
5.18.53 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.

5.18.54 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

5.18.55 V 0.2.1

- Demo files were not included when installed from pip

5.18.56 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.

5.18.57 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

5.18.58 V 0.1.2

A PyPI setup.py bug was not properly installing pylinac nor including demo files. Both of these have been fixed.
5.18.59 V 0.1.1

Several small bugs were fixed and small optimizations made. A few methods were refactored for similarity between modules.

5.18.60 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.
CHAPTER 6

Indices and tables

• genindex
• modindex
• search
p
pylinac.core.decorators, 265
pylinac.core.geometry, 249
pylinac.core.image, 241
pylinac.core.io, 259
pylinac.core.mask, 263
pylinac.core.profile, 252
pylinac.core.roi, 261
pylinac.core.utilities, 264
pylinac.ct, 75
pylinac.log_analyzer, 103
pylinac.picketfence, 135
pylinac.planar_imaging, 184
pylinac.starshot, 46
pylinac.vmat, 59
pylinac.winston_lutz, 167
Index

A

A (pylinac.log_analyzer.MLCBank attribute), 122
a_logfile (pylinac.log_analyzer.Dynalog attribute), 119
abs_mean_deviation (pylinac.vmat.VMATResult attribute), 72
abs_median_error (pylinac.picketfence.PicketFence attribute), 161
absolute_median_error_mm (pylinac.picketfence.PFResult attribute), 165
action_tolerance_mm (pylinac.picketfence.PFResult attribute), 165
ACTUAL (pylinac.log_analyzer.Fluence attribute), 123
actual (pylinac.log_analyzer.FluenceStruct attribute), 131
ActualFluence (class in pylinac.log_analyzer), 132
add_guards_to_axes() (pylinac.picketfence.Picket method), 166
add_layer() (pylinac.core.image_generator.simulators.AS1000Image method), 277
add_layer() (pylinac.core.image_generator.simulators.AS1200Image method), 277
add_layer() (pylinac.core.image_generator.simulators.AS500Image method), 277
add_leaf_axis() (pylinac.log_analyzer.MLC method), 124
adjust_for_sag() (pylinac.picketfence.PFDicomImage method), 165
analyze() (pylinac.ct.CatPhan503 method), 87
analyze() (pylinac.ct.CatPhan504 method), 84
analyze() (pylinac.ct.CatPhan600 method), 90
analyze() (pylinac.ct.CatPhan604 method), 93
analyze() (pylinac.field_analysis.DeviceFieldAnalysis method), 238
analyze() (pylinac.field_analysis.FieldAnalysis method), 232
analyze() (pylinac.picketfence.PicketFence method), 161
analyze() (pylinac.planar_imaging.DoselabMC2kV method), 211
analyze() (pylinac.planar_imaging.DoselabMC2MV method), 209
analyze() (pylinac.planar_imaging.LasVegas method), 206
analyze() (pylinac.planar_imaging.LeedsTOR method), 200
analyze() (pylinac.planar_imaging.PTWEPIDQC method), 217
analyze() (pylinac.planar_imaging.SNCkV method), 215
analyze() (pylinac.planar_imaging.SNCMV method), 213
analyze() (pylinac.planar_imaging.StandardImagingQC3 method), 202
analyze() (pylinac.planar_imaging.StandardImagingQCkV method), 204
analyze() (pylinac.starshot.Starshot method), 56
AS1000Image (pylinac.vmat.DRGS method), 70
AS1000Image (pylinac.vmat.DRMLC method), 71
AS1200Image (pylinac.vmat.VMATBase method), 73
AS500Image (pylinac.winston_lutz.WinstonLutz2D method), 183
anonymize() (in module pylinac.log_analyzer), 118
anonymize() (pylinac.log_analyzer.MachineLogs method), 122
append() (pylinac.log_analyzer.MachineLogs method), 121
apply() (pylinac.core.image_generator.layers.ConstantLayer method), 276
apply() (pylinac.core.image_generator.layers.FilteredFieldLayer method), 275
apply() (pylinac.core.image_generator.layers.FilterFreeConeLayer method), 275
apply() (pylinac.core.image_generator.layers.FilterFreeFieldLayer method), 276
apply() (pylinac.core.image_generator.layers.GaussianFilterLayer method), 276
apply() (pylinac.core.image_generator.layers.PerfectBBLayer method), 276
apply() (pylinac.core.image_generator.layers.PerfectConeLayer method), 274
apply() (pylinac.core.image_generator.layers.PerfectFieldLayer method), 275
apply() (pylinac.core.image_generator.layers.RandomNoiseLayer method), 276
array (pylinac.core.image.BaseImage attribute), 243
array (pylinac.log_analyzer.Fluence attribute), 131
array (pylinac.log_analyzer.Gamma attribute), 132
ArrayImage (class in pylinac.core.image), 247
AS1000Image (class in pylinac.core.image_generator.simulators), 277
AS1200Image (class in pylinac.core.image_generator.simulators), 277
AS500Image (class in pylinac.core.image_generator.simulators), 276
as_array() (pylinac.core.geometry.Point method), 249
as_binary() (pylinac.core.image.BaseImage method), 244
as_scalar() (pylinac.core.geometry.Vector method), 250
assign2machine() (in module pylinac.core.utilities), 264
avg_abs_r_deviation (pylinac.vmat.DRGS attribute), 70
avg_abs_r_deviation (pylinac.vmat.DRMLC attribute), 71
avg_abs_r_deviation (pylinac.vmat.VMAT attribute), 73
avg_gamma (pylinac.log_analyzer.Gamma attribute), 133
avg_gamma() (pylinac.log_analyzer.MachineLogs method), 121
avg_gamma_pct() (pylinac.log_analyzer.MachineLogs method), 121
avg_line_distance_mm (pylinac.ct.CTP404Result attribute), 96
avg_r_deviation (pylinac.vmat.DRGS attribute), 70
avg_r_deviation (pylinac.vmat.DRMLC attribute), 71
avg_r_deviation (pylinac.vmat.VMAT attribute), 73
Axis (class in pylinac.log_analyzer), 123
axis_data (pylinac.log_analyzer.Dynalog attribute), 119
axis_rms_deviation() (pylinac.winston_lutz.WinstonLutz method), 180
B (pylinac.log_analyzer.MLCBank attribute), 122
bbox_center() (in module pylinac.core.roi), 261
BEAM_CENTER (pylinac.core.profile.Normalization attribute), 253
BEAM_CENTER (pylinac.field_analysis.Centering attribute), 239
BEAM_CENTER (pylinac.field_analysis.Normalization attribute), 240
beam_center() (pylinac.core.profile.SingleProfile method), 254
beam_center_index_x_y (pylinac.field_analysis.FieldResult attribute), 237
beam_center_to_bottom_mm (pylinac.field_analysis.FieldResult attribute), 237
beam_center_to_left_mm (pylinac.field_analysis.FieldResult attribute), 237
beam_center_to_right_mm (pylinac.field_analysis.FieldResult attribute), 237
beam_center_to_top_mm (pylinac.field_analysis.FieldResult attribute), 237
beam_hold (pylinac.log_analyzer.DynalogAxisData attribute), 128
beam_name (pylinac.log_analyzer.Subbeam attribute), 130
beam_on (pylinac.log_analyzer.DynalogAxisData attribute), 128
bg_color (pylinac.picketfence.MLCValue attribute), 166
bl_corner (pylinac.core.geometry.Rectangle attribute), 251
BOTH (pylinac.log_analyzer.MLCBank attribute), 122
pylinac Documentation, Release 3.1.0

attribute), 247
collimator_angle (pylinac.log_analyzer.Subbeam
attribute), 130
collimator_iso_size
(pylinac.winston_lutz.WinstonLutz attribute), 180
combine_surrounding_slices() (in module
pylinac.ct), 102
ConstantLayer (class in
pylinac.core.image_generator.layers), 276
construct_rad_lines()
(pylinac.starshot.LineManager method), 58
Contrast (class in pylinac.core.roi), 261
contrast (pylinac.core.roi.LowContrastDiskROI
attribute), 262
contrast_constant
(pylinac.core.roi.LowContrastDiskROI
attribute), 263
contrast_to_noise
(pylinac.core.roi.LowContrastDiskROI
attribute), 262
control_point (pylinac.log_analyzer.Subbeam
attribute), 130
convert_to_enum() (in module
pylinac.core.utilities), 264
cos() (in module pylinac.core.geometry), 249
couch_2d_iso_diameter_mm
(pylinac.winston_lutz.WinstonLutzResult
attribute), 182
couch_angle (pylinac.core.image.LinacDicomImage
attribute), 247
couch_angle_varian_scale
(pylinac.winston_lutz.WinstonLutz2D
attribute), 183
couch_iso_size
(pylinac.winston_lutz.WinstonLutz
attribute), 180
CouchStruct (class in pylinac.log_analyzer), 134
create_error_array()
(pylinac.log_analyzer.MLC method), 126
create_RMS_array()
(pylinac.log_analyzer.MLC method), 127
crop() (pylinac.core.image.BaseImage method), 243
ctp404 (pylinac.ct.CatphanResult attribute), 96
CTP404CP503 (class in pylinac.ct), 98
CTP404CP504 (class in pylinac.ct), 99
CTP404CP600 (class in pylinac.ct), 99
CTP404CP604 (class in pylinac.ct), 100
CTP404Result (class in pylinac.ct), 96
CTP486 (class in pylinac.ct), 101
ctp486 (pylinac.ct.CatphanResult attribute), 96
CTP486Result (class in pylinac.ct), 97
CTP515 (class in pylinac.ct), 101
ctp515 (pylinac.ct.CatphanResult attribute), 96
CTP515Result (class in pylinac.ct), 97
ctp528 (pylinac.ct.CatphanResult attribute), 96
CTP528CP503 (class in pylinac.ct), 100
CTP528CP504 (class in pylinac.ct), 100
CTP528CP600 (class in pylinac.ct), 100
CTP528CP604 (class in pylinac.ct), 100
CTP528Result (class in pylinac.ct), 96
CTP528Result (class in pylinac.ct), 96
ctp528 (pylinac.ct.CatphanResult attribute), 96
d_ref() (in module pylinac.calibration.tg51), 34
date_of_analysis (pylinac.core.utilities.ResultBase
attribute), 264
decode_binary() (in module pylinac.core.utilities), 265
Device (class in pylinac.field_analysis), 239
device (pylinac.field_analysis.DeviceFieldAnalysis
attribute), 237
DeviceFieldAnalysis (class in
pylinac.field_analysis), 237
diameter (pylinac.core.geometry.Circle attribute), 250
diameter_mm (pylinac.starshot.Wobble attribute), 58
DicomImage (class in pylinac.core.image), 246
DicomImageStack (class in pylinac.core.image), 248
difference (pylinac.ct.ROIResult attribute), 97
difference (pylinac.log_analyzer.Axis attribute), 123
DiskROI (class in pylinac.core.roi), 261
dist2cax (pylinac.picketfence.Picket attribute), 165
dist2edge_min() (pylinac.core.image/BaseImage
method), 244
distance_to() (pylinac.core.geometry.Line method), 251
distance_to() (pylinac.core.geometry.Point
method), 249
distance_to() (pylinac.core.geometry.Vector
method), 250
distTA (pylinac.log_analyzer.GammaFluence attribute), 132
dose_mu_10 (pylinac.calibration.tg51.TG51Photon
attribute), 37
dose_mu_10_adjusted
(pylinac.calibration.tg51.TG51Photon
attribute), 37
dose_mu_dmax (pylinac.calibration.tg51.TG51ElectronLegacy
attribute), 39
dose_mu_dmax (pylinac.calibration.tg51.TG51ElectronModern
attribute), 41
dose_mu_dmax (pylinac.calibration.tg51.TG51Photon
attribute), 37
dose_mu_dmax_adjusted
(pylinac.calibration.tg51.TG51ElectronLegacy
attribute), 39
dose_mu_dmax_adjusted
(pylinac.calibration.tg51.TG51ElectronModern
attribute), 41
dose_mu_dmax_adjusted
(pylinac.calibration.tg51.TG51Photon attribute), 37

dose_mu_dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39

dose_mu_dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

dose_mu_dref_adjusted (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39

dose_mu_dref_adjusted (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

dose_mu_zmax (pylinac.calibration.trs398.TRS398Electron attribute), 45

dose_mu_zmax (pylinac.calibration.trs398.TRS398Photon attribute), 44

dose_mu_zmax_adjusted (pylinac.calibration.trs398.TRS398Electron attribute), 45

dose_mu_zmax_adjusted (pylinac.calibration.trs398.TRS398Photon attribute), 44
DoselabMC2kV (class in pylinac.planar_imaging), 210
DoselabMC2MV (class in pylinac.planar_imaging), 208
doseTA (pylinac.log_analyzer.GammaFluence attribute), 133
dpi (pylinac.core.image.ArrayImage attribute), 248
dpi (pylinac.core.image.DicomImage attribute), 246
dpi (pylinac.core.image.FileImage attribute), 247
dpmm (pylinac.core.image.ArrayImage attribute), 248
dpmm (pylinac.core.image.DicomImage attribute), 246
dpmm (pylinac.core.image.FileImage attribute), 247
dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

DRGS (class in pylinac.vmat), 68

DRMLC (class in pylinac.vmat), 71

Dynalog (class in pylinac.log_analyzer), 118
DynalogAxisData (class in pylinac.log_analyzer), 128
DynalogHeader (class in pylinac.log_analyzer), 127
DynalogMatchError (class in pylinac.log_analyzer), 135

DYNAMIC_IMRT (pylinac.log_analyzer.TreatmentType attribute), 123

E

Edge (class in pylinac.core.profile), 253

Edge (class in pylinac.field_analysis), 240

detect method (pylinac.field_analysis.FieldResult attribute), 236

dose_mu_dmax_adjusted (pylinac.calibration.tg51.TG51Photon attribute), 37
dose_mu_dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dose_mu_dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41
dose_mu_dref_adjusted (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dose_mu_dref_adjusted (pylinac.calibration.tg51.TG51ElectronModern attribute), 41
dose_mu_zmax (pylinac.calibration.trs398.TRS398Electron attribute), 45
dose_mu_zmax (pylinac.calibration.trs398.TRS398Photon attribute), 44
dose_mu_zmax_adjusted (pylinac.calibration.trs398.TRS398Electron attribute), 45
dose_mu_zmax_adjusted (pylinac.calibration.trs398.TRS398Photon attribute), 44
DoselabMC2kV (class in pylinac.planar_imaging), 210
DoselabMC2MV (class in pylinac.planar_imaging), 208
doseTA (pylinac.log_analyzer.GammaFluence attribute), 133
dpi (pylinac.core.image.ArrayImage attribute), 248
dpi (pylinac.core.image.DicomImage attribute), 246
dpi (pylinac.core.image.FileImage attribute), 247
dpmm (pylinac.core.image.ArrayImage attribute), 248
dpmm (pylinac.core.image.DicomImage attribute), 246
dpmm (pylinac.core.image.FileImage attribute), 247
dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

DRGS (class in pylinac.vmat), 68

DRMLC (class in pylinac.vmat), 71

Dynalog (class in pylinac.log_analyzer), 118
DynalogAxisData (class in pylinac.log_analyzer), 128
DynalogHeader (class in pylinac.log_analyzer), 127
DynalogMatchError (class in pylinac.log_analyzer), 135

DYNAMIC_IMRT (pylinac.log_analyzer.TreatmentType attribute), 123

E

Edge (class in pylinac.core.profile), 253

Edge (class in pylinac.field_analysis), 240

detect method (pylinac.field_analysis.FieldResult attribute), 236

ELEKTA (pylinac.field_analysis.Protocol attribute), 239
epid (pylinac.winston_lutz.WinstonLutz2D attribute), 183
epid (pylinac.winston_lutz.WinstonLutz2D attribute), 183
equal_images() (in module pylinac.core.image), 241

DoselabMC2kV (class in pylinac.planar_imaging), 210
DoselabMC2MV (class in pylinac.planar_imaging), 208
doseTA (pylinac.log_analyzer.GammaFluence attribute), 133
dpi (pylinac.core.image.ArrayImage attribute), 248
dpi (pylinac.core.image.DicomImage attribute), 246
dpi (pylinac.core.image.FileImage attribute), 247
dpmm (pylinac.core.image.ArrayImage attribute), 248
dpmm (pylinac.core.image.DicomImage attribute), 246
dpmm (pylinac.core.image.FileImage attribute), 247
dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

DRGS (class in pylinac.vmat), 68

DRMLC (class in pylinac.vmat), 71

Dynalog (class in pylinac.log_analyzer), 118
DynalogAxisData (class in pylinac.log_analyzer), 128
DynalogHeader (class in pylinac.log_analyzer), 127
DynalogMatchError (class in pylinac.log_analyzer), 135

DYNAMIC_IMRT (pylinac.log_analyzer.TreatmentType attribute), 123

equal_images() (in module pylinac.core.image), 241

DoselabMC2kV (class in pylinac.planar_imaging), 210
DoselabMC2MV (class in pylinac.planar_imaging), 208
doseTA (pylinac.log_analyzer.GammaFluence attribute), 133
dpi (pylinac.core.image.ArrayImage attribute), 248
dpi (pylinac.core.image.DicomImage attribute), 246
dpi (pylinac.core.image.FileImage attribute), 247
dpmm (pylinac.core.image.ArrayImage attribute), 248
dpmm (pylinac.core.image.DicomImage attribute), 246
dpmm (pylinac.core.image.FileImage attribute), 247
dref (pylinac.calibration.tg51.TG51ElectronLegacy attribute), 39
dref (pylinac.calibration.tg51.TG51ElectronModern attribute), 41

DRGS (class in pylinac.vmat), 68

DRMLC (class in pylinac.vmat), 71

Dynalog (class in pylinac.log_analyzer), 118
DynalogAxisData (class in pylinac.log_analyzer), 128
DynalogHeader (class in pylinac.log_analyzer), 127
DynalogMatchError (class in pylinac.log_analyzer), 135

DYNAMIC_IMRT (pylinac.log_analyzer.TreatmentType attribute), 123

E

Edge (class in pylinac.core.profile), 253

Edge (class in pylinac.field_analysis), 240

detect method (pylinac.field_analysis.FieldResult attribute), 236

fahrenheit2celsius() (in module pylinac.calibration.tg51), 33

failed_leaves (pylinac.picketfence.PFResult attribute), 165

failed_leaves() (pylinac.picketfence.PicketFence method), 161

field_calculation() (pylinac.core.profile.SingleProfile method), 255

field_data() (pylinac.core.profile.SingleProfile method), 254

field_size_horizontal_mm (pylinac.field_analysis.FieldResult attribute), 237

field_size_vertical_mm (pylinac.field_analysis.FieldResult attribute), 237

FieldAnalysis (class in pylinac.field_analysis), 232
FieldResult (class in pylinac.field_analysis), 235

file_exists() (in module pylinac.core.utilities), 265

FileImage (class in pylinac.core.image), 247

filled_area_ratio() (in module pylinac.core.mask), 264

filter() (pylinac.core.image.BaseImage method), 243

filter() (pylinac.core.profile.ProfileMixin method), 252

FilteredFieldLayer (class in pylinac.core.image_generator.layers), 275

FilterFreeConeLayer (class in pylinac.core.image_generator.layers), 275

FilterFreeFieldLayer (class in pylinac.core.image_generator.layers), 275

find_fwxm_peaks() (pylinac.core.profile.CircleProfile method), 257

find_fwxm_peaks() (pylinac.core.profile.MultiProfile method), 256
find_origin_slice() (pylinac.ct.CatPhan503 method), 88
find_origin_slice() (pylinac.ct.CatPhan504 method), 85
find_origin_slice() (pylinac.ct.CatPhan600 method), 91
find_origin_slice() (pylinac.ct.CatPhan604 method), 94
find_peaks() (in module pylinac.core.profile), 258
find_peaks() (pylinac.core.profile.CircleProfile method), 257
find_peaks() (pylinac.core.profile.MultiProfile method), 257
find_phantom_roll() (pylinac.ct.CatPhan503 method), 88
find_phantom_roll() (pylinac.ct.CatPhan504 method), 85
find_phantom_roll() (pylinac.ct.CatPhan600 method), 91
find_phantom_roll() (pylinac.ct.CatPhan604 method), 94
find_valleys() (pylinac.core.profile.CircleProfile method), 257
find_valleys() (pylinac.core.profile.MultiProfile method), 257

flatness_dose_difference() (in module pylinac.field_analysis), 240
flatness_dose_ratio() (in module pylinac.field_analysis), 240
fliplr() (pylinac.core.image.BaseImage method), 244
flipud() (pylinac.core.image.BaseImage method), 244

Fluence (class in pylinac.log_analyzer), 122
fluence (pylinac.log_analyzer.Dynalog attribute), 118
fluence (pylinac.log_analyzer.TrajectoryLog attribute), 119
FluenceBase (class in pylinac.log_analyzer), 131
FluenceStruct (class in pylinac.log_analyzer), 131
from_demo() (pylinac.log_analyzer.Dynalog class method), 119
from_demo() (pylinac.log_analyzer.TrajectoryLog class method), 120

from_demo_image() (pylinac.planar_imaging.DoselabMC2kV class method), 209
from_demo_image() (pylinac.planar_imaging.DoselabMC2MV class method), 205
from_demo_image() (pylinac.planar_imaging.LasVegas class method), 207
from_demo_image() (pylinac.planar_imaging.LeedsTOR class method), 201
from_demo_image() (pylinac.planar_imaging.PTWEPIDQC class method), 218
from_demo_image() (pylinac.planar_imaging.StandardImagingQC3 class method), 202
from_demo_image() (pylinac.planar_imaging.StandardImagingQCkV class method), 205
from_demo_images() (pylinac.ct.CatPhan503 class method), 88
from_demo_images() (pylinac.ct.CatPhan504 class method), 85
from_demo_images() (pylinac.ct.CatPhan600 class method), 91
from_demo_images() (pylinac.ct.CatPhan604 class method), 94
from_demo_images() (pylinac.vmat.DRGS class method), 70
from_demo_images() (pylinac.vmat.DRMLC class method), 71
from_demo_images() (pylinac.vmat.VMATBase class method), 73
from_demo_images() (pylinac.winston_lutz.WinstonLutz class method), 179
from_diog() (pylinac.log_analyzer.MLC class method), 124
from_multiple_images() (pylinac.picketfence.PicketFence class method), 161
from_multiple_images() (pylinac.starshot.Starshot class method), 55
from_multiples() (pylinac.core.image.BaseImage class method), 243
INDEX

get_bg_color() (pynlinac.vmat.Segment method), 74
get_error_percentile() (pynlinac.log_analyzer.MLC method), 126
get_fit() (pynlinac.picketfence.Picket method), 165
get_leaves() (pynlinac.log_analyzer.MLC method), 126
get_peaks() (pynlinac.starshot.StarProfile method), 58
get_regions() (in module pynlinac.ct), 103
get_RMS() (pynlinac.log_analyzer.MLC method), 126
get_RMS_avg() (pynlinac.log_analyzer.MLC method), 125
get_RMS_max() (pynlinac.log_analyzer.MLC method), 125
get_RMS_percentile() (pynlinac.log_analyzer.MLC method), 125
get_snapshot_values() (pynlinac.log_analyzer.MLC method), 127
get_url() (in module pynlinac.core.io), 260
Graph (class in pynlinac.log_analyzer), 122
ground() (pynlinac.core.image.BaseImage method), 245
ground() (pynlinac.core.profile.ProfileMixin method), 252

H
HALCYON_DISTAL (pynlinac.picketfence.MLC attribute), 164
HALCYON_PROXIMAL (pynlinac.picketfence.MLC attribute), 164
HD_MILLENNIUM (pynlinac.picketfence.MLC attribute), 164
header (pynlinac.log_analyzer.Dynalog attribute), 118
header (pynlinac.log_analyzer.TrajectoryLog attribute), 119
HighContrastDiskROI (class in pynlinac.core.roi), 263
HISTOGRAM (pynlinac.log_analyzer.Graph attribute), 122
histogram() (pynlinac.log_analyzer.GammaFluence method), 133
horiz_profile (pynlinac.field_analysis.FieldAnalysis attribute), 232
hu_linearity_passed (pynlinac.ct.CTP404Result attribute), 96
hu_rois (pynlinac.ct.CTP404Result attribute), 96
hu_tolerance (pynlinac.ct.CTP404Result attribute), 96
HUDiskROI (class in pynlinac.ct), 101

I
identify_other_file() (pynlinac.log_analyzer.Dynalog static method),

119
image (pynlinac.field_analysis.FieldAnalysis attribute), 232
image (pynlinac.starshot.Starshot attribute), 55
image_array (pynlinac.core.profile.CircleProfile attribute), 257
ImageManager (class in pynlinac.winston_lutz), 183
images (pynlinac.core.image.DicomImageStack attribute), 248
images (pynlinac.winston_lutz.WinstonLutz attribute), 179
IMAGING (pynlinac.winston_lutz.TreatmentType attribute), 123
inflection_data() (pynlinac.core.profile.SingleProfile method), 254
INFECTION_DERIVATIVE (pynlinac.core.profile.Edge attribute), 253
INFECTION_DERIVATIVE (pynlinac.field_analysis.Edge attribute), 240
INFECTION_HILL (pynlinac.core.profile.Edge attribute), 253
INFECTION_HILL (pynlinac.field_analysis.Edge attribute), 240
info (pynlinac.core.image.FileImage attribute), 247
integral_non_uniformity (pynlinac.ct.CTP486 attribute), 101
integral_non_uniformity (pynlinac.ct.CTP486Result attribute), 97
Interpolation (class in pynlinac.core.profile), 253
Interpolation (class in pynlinac.field_analysis), 239
interpolation_method (pynlinac.field_analysis.FieldResult attribute), 236
invert() (pynlinac.core.image.BaseImage method), 244
invert() (pynlinac.core.profile.ProfileMixin method), 252
is_close() (in module pynlinac.core.utilities), 264
is_CT_slice() (pynlinac.core.image.DicomImageStack static method), 248
is_dicom() (in module pynlinac.core.io), 259
is_dicom_image() (in module pynlinac.core.io), 259
is_hdmic (pynlinac.log_analyzer.TrajectoryLog attribute), 120
is_image() (in module pynlinac.core.image), 241
is_iterable() (in module pynlinac.core.utilities), 264
is_map_calced() (pynlinac.log_analyzer.FluenceBase method), 131
is_url() (in module pynlinac.core.io), 260
is_zipfile() (in module pynlinac.core.io), 259
isnumeric() (in module pynlinac.core.utilities), 264
Index 335

offset (pylinac.ct.CTP404Result attribute), 96
offsets_from_cax_mm (pylinac.picketfence.PFResult attribute), 165
open_path() (in module pylinac.core.utilities), 265
Orientation (class in pylinac.picketfence), 164
orientation (pylinac.picketfence.PicketFence attribute), 164
origin_slice (pylinac.ct.CatphanResult attribute), 96
overall_passed (pylinac.ct.CTP486 attribute), 101
p_ion() (in module pylinac.calibration.tg51), 33
p_tps() (in module pylinac.calibration.tg51), 33
pair_moved() (pylinac.log_analyzer.MLC method), 125
pass_fail_color (pylinac.ct.GeometricLine attribute), 102
pass_prctnt (pylinac.log_analyzer.GammaFluence attribute), 133
passed (pylinac.core.roi.LowContrastDiskROI attribute), 263
passed (pylinac.ct.CTP486Result attribute), 97
passed (pylinac.ct.GeometricLine attribute), 102
passed (pylinac.ct.HUDiskROI attribute), 102
passed (pylinac.ct.ORIResult attribute), 97
passed (pylinac.picketfence.MLCValue attribute), 166
passed (pylinac.picketfence.PFResult attribute), 165
passed (pylinac.picketfence.PicketFence attribute), 161
passed (pylinac.starshot.Starshot attribute), 57
passed (pylinac.starshot.StarshotResults attribute), 58
passed (pylinac.vmat.Segment attribute), 74
passed (pylinac.vmat.SegmentResult attribute), 72
passed_action (pylinac.picketfence.MLCValue attribute), 166
passed_cnr_constant (pylinac.core.roi.LowContrastDiskROI attribute), 263
passed_contrast_constant (pylinac.core.roi.LowContrastDiskROI attribute), 263
passed_geometry (pylinac.ct.CTP404CP504 attribute), 99
passed_hu (pylinac.ct.CTP404CP504 attribute), 99
passed_thickness (pylinac.ct.CTP404CP504 attribute), 99
passed_visibility (pylinac.core.roi.LowContrastDiskROI attribute), 263
passfail_array (pylinac.log_analyzer.GammaFluence attribute), 132
path (pylinac.core.image.BaseImage attribute), 243
patient_name (pylinac.log_analyzer.DynalogHeader attribute), 127
pddx (pylinac.calibration.tg51.TG51Photon attribute), 37
pddx() (in module pylinac.calibration.tg51), 34
peaks (pylinac.core.profile.MultiProfile attribute), 255

pylinac Documentation, Release 3.1.0
plot_analyzed_image() (pylinac.ct.CatPhan504 method), 86
plot_analyzed_image() (pylinac.ct.CatPhan600 method), 92
plot_analyzed_image() (pylinac.ct.CatPhan604 method), 94
plot_analyzed_image() (pylinac.field_analysis.FieldAnalysis method), 235
plot_analyzed_image() (pylinac.picketfence.PicketFence method), 163
plot_analyzed_image() (pylinac.planar_imaging.DoselabMC2kV method), 212
plot_analyzed_image() (pylinac.planar_imaging.DoselabMC2MV method), 210
plot_analyzed_image() (pylinac.planar_imaging.LasVegas method), 207
plot_analyzed_image() (pylinac.planar_imaging.LeedsTOR method), 201
plot_analyzed_image() (pylinac.planar_imaging.PTWEPIDQC method), 218
plot_analyzed_image() (pylinac.planar_imaging.SNCKV method), 216
plot_analyzed_image() (pylinac.planar_imaging.SNCMV method), 214
plot_analyzed_image() (pylinac.planar_imaging.StandardImagingQC3 method), 203
plot_analyzed_image() (pylinac.planar_imaging.StandardImagingQCkV method), 205
plot_analyzed_image() (pylinac.starshot.Starshot method), 57
plot_analyzed_image() (pylinac.vmat.DRGS method), 70
plot_analyzed_image() (pylinac.vmat.DRMLC method), 71
plot_analyzed_image() (pylinac.vmat.VMATBase method), 73
plot_analyzed_subimage() (pylinac.ct.CatPhan503 method), 89
plot_analyzed_subimage() (pylinac.ct.CatPhan504 method), 86
plot_analyzed_subimage() (pylinac.ct.CatPhan600 method), 92
plot_analyzed_subimage() (pylinac.ct.CatPhan604 method), 95
plot_analyzed_subimage() (pylinac.starshot.Starshot method), 57
plot_axis_images() (pylinac.winston_lutz.WinstonLutz method), 181
plot_color (pylinac.core.roi.LowContrastDiskROI attribute), 263
plot_color (pylinac.ct.HUDiskROI attribute), 102
plot_color (pylinac.ct.ThicknessROI attribute), 102
plot_color_cnr (pylinac.core.roi.LowContrastDiskROI attribute), 263
plot_color_constant (pylinac.core.roi.LowContrastDiskROI attribute), 263
plot_difference() (pylinac.log_analyzer.Axis method), 124
plot_expected() (pylinac.log_analyzer.Axis method), 124
plot_histogram() (pylinac.log_analyzer.GammaFluence method), 134
plot_histogram() (pylinac.picketfence.PicketFence method), 164
plot_images() (pylinac.winston_lutz.WinstonLutz method), 181
plot_leaf_profile() (pylinac.picketfence.PicketFence method), 161
plot_linearity() (pylinac.ct.CTP404CP504 method), 99
plot_map() (pylinac.log_analyzer.FluenceBase method), 132
plot_map() (pylinac.log_analyzer.GammaFluence method), 133
plot_mlc_error_hist() (pylinac.log_analyzer.MLC method), 127
plot_mtf() (pylinac.ct.CTP528CP504 method), 100
plot_overlay2axes() (pylinac.picketfence.MLCValue method), 166
plot_passfail_map() (pylinac.log_analyzer.GammaFluence method), 134
plot_profiles() (pylinac.ct.CTP486 method), 101
plot_rms_by_leaf() (pylinac.log_analyzer.MLC method), 127
plot_rois() (pylinac.ct.CatPhanModule method), 98
plot_rois() (pylinac.ct.CTP404CP504 method), 99
plot_rois() (pylinac.ct.CTP528CP504 method), 100
plot_summary() (pylinac.winston_lutz.WinstonLutz method), 181
Point (class in pylinac.core.geometry), 249
post_hoc_metadata() (pylinac.log_analyzer.SubbeamManager method), 337
run_demo() (pylinac.field_analysis.DeviceFieldAnalysis static method), 238
run_demo() (pylinac.field_analysis.FieldAnalysis static method), 232
run_demo() (pylinac.log_analyzer.Dynalog static method), 119
run_demo() (pylinac.log_analyzer.TrajectoryLog static method), 120
run_demo() (pylinac.picketfence.PicketFence static method), 161
run_demo() (pylinac.planar_imaging.DoselabMC2kV static method), 211
run_demo() (pylinac.planar_imaging.DoselabMC2MV static method), 209
run_demo() (pylinac.planar_imaging.LasVegas static method), 206
run_demo() (pylinac.planar_imaging.LeedsTOR static method), 200
run_demo() (pylinac.planar_imaging.PTWEPIDQC static method), 217
run_demo() (pylinac.planar_imaging.PTWEpidQC method), 219
run_demo() (pylinac.planar_imaging.StandardImagingQC3 method), 204
run_demo() (pylinac.planar_imaging.StandardImagingQCkV method), 206
run_demo() (pylinac.planar_imaging.StandardImagingQCkV method), 214
run_demo() (pylinac.planar_imaging.StandardImagingQCMe method), 202
run_demo() (pylinac.planar_imaging.Starshot static method), 57
run_demo() (pylinac.planar_imaging.Starshot static method), 58
run_demo() (pylinac.vmat.DRGS static method), 70
run_demo() (pylinac.vmat.DRMLC static method), 71
run_demo() (pylinac.winston_lutz.WinstonLutz static method), 180

S
save() (pylinac.core.image.DicomImage method), 246
save_analyzed_image() (pylinac.planar_imaging.DoselabMC2kV method), 212
save_analyzed_image() (pylinac.planar_imaging.DoselabMC2kV method), 210
save_analyzed_image() (pylinac.planar_imaging.DoselabMC2MV method), 206
save_analyzed_image() (pylinac.planar_imaging.LasVegas method), 208
save_analyzed_image() (pylinac.planar_imaging.LeedsTOR method), 202
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQC3 method), 204
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQCkV method), 216
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQCkV method), 214
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQCMe method), 211
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQCMe method), 209
save_analyzed_image() (pylinac.planar_imaging.StandardImagingQCMe method), 206
save_analyzed_subimage() (pylinac.planar_imaging.StandardImagingQCMe method), 204
save_analyzed_subimage() (pylinac.ct.CatPhan503 method), 90
save_analyzed_subimage() (pylinac.ct.CatPhan504 method), 87
save_analyzed_subimage() (pylinac.ct.CatPhan504 method), 87
save_analyzed_subimage() (pylinac.ct.CatPhan504 method), 87
save_analyzed_subimage() (pylinac.ct.CatPhan504 method), 87
save_analyzed_subimage() (pylinac.ct.CatPhan504 method), 87
save_plot() (pylinac.winston_lutz.WinstonLutz2D method), 184
save_plot() (pylinac.winston_lutz.WinstonLutz2D method), 184
save_rms_by_leaf() (pylinac.log_analyzer.MLC method), 127
save_histogram() (pylinac.log_analyzer.GammaFluence method), 134
save_histogram() (pylinac.picketfence.PicketFence method), 164
save_images() (pylinac.planar_imaging.PTWEPIDQC method), 181
save_leaf_profile() (pylinac.picketfence.PicketFence method), 161
save_map() (pylinac.log_analyzer.FluenceBase method), 132
save_mlc_error_hist() (pylinac.log_analyzer.MLC method), 127
save_mlc_error_hist() (pylinac.log_analyzer.MLC method), 127
top_position_index_x_y (pylinac.field_analysis.FieldResult attribute), 237

top_slope_percent_mm (pylinac.field_analysis.FieldResult attribute), 237

top_vertical_distance_from_beam_center_mm (pylinac.field_analysis.FieldResult attribute), 237

top_vertical_distance_from_cax_mm (pylinac.field_analysis.FieldResult attribute), 237

tpr2010_from_pdd2010() (in module pylinac.calibration.tg51), 33

tr_corner (pylinac.core.geometry.Rectangle attribute), 251

tr_corner (pylinac.vmat.Segment attribute), 74

TrajectoryLog (class in pylinac.log_analyzer), 119

TrajectoryLogAxisData (class in pylinac.log_analyzer), 129

TrajectoryLogHeader (class in pylinac.log_analyzer), 129

TreatmentType (class in pylinac.log_analyzer), 123

TRS398Electron (class in pylinac.calibration.trs398), 44

TRS398Photon (class in pylinac.calibration.trs398), 43

txt_filename (pylinac.log_analyzer.TrajectoryLog attribute), 120

uniformity_index (pylinac.ct.CTP486 attribute), 101

uniformity_index (pylinac.ct.CTP486Result attribute), 97

UP_DOWN (pylinac.picketfence.Orientation attribute), 164

upper_window (pylinac.ct.CTP515 attribute), 101

valleys (pylinac.core.profile.MultiProfile attribute), 255

value (pylinac.ct.ROIResult attribute), 97

value_diff (pylinac.ct.HUDiskROI attribute), 101

values (pylinac.core.profile.MultiProfile attribute), 255

variable_axis (pylinac.winston_lutz.WinstonLutz2D attribute), 184

VARIAN (pylinac.field_analysis.Protocol attribute), 239

Vector (class in pylinac.core.geometry), 250

vector_is_close() (in module pylinac.core.geometry), 250

version (pylinac.log_analyzer.DynalogHeader attribute), 127

vert (pylinac.log_analyzer.CouchStruct attribute), 134

vert_profile (pylinac.field_analysis.FieldAnalysis attribute), 232

visibility (pylinac.core.roi.LowContrastDiskROI attribute), 262

VMAT (pylinac.log_analyzer.TreatmentType attribute), 123

VMATResult (class in pylinac.vmat), 72

W

WEBER (pylinac.core.roi.Contrast attribute), 261

WinstonLutz (class in pylinac.winston_lutz), 179

WinstonLutz2D (class in pylinac.winston_lutz), 182

wire_fwhm (pylinac.ct.ThicknessROI attribute), 102

wobble (class in pylinac.starshot), 58

wobble (pylinac.starshot.Starshot attribute), 55

X

x() (pylinac.core.geometry.Line method), 251

x1 (pylinac.log_analyzer.JawStruct attribute), 134

x2 (pylinac.log_analyzer.JawStruct attribute), 134

x_locations (pylinac.core.profile.CircleProfile attribute), 257

x_position_mm (pylinac.vmat.SegmentResult attribute), 72

Y

y() (pylinac.core.geometry.Line method), 250

y1 (pylinac.log_analyzer.JawStruct attribute), 134

y2 (pylinac.log_analyzer.JawStruct attribute), 134

y_locations (pylinac.core.profile.CircleProfile attribute), 257

Z

zref (pylinac.calibration.trs398.TRS398Electron attribute), 45