c3dp Documentation

Release 0.2.0

Fahima Islam

Nov 03, 2021

CONTENTS:

CHAPTER ONE

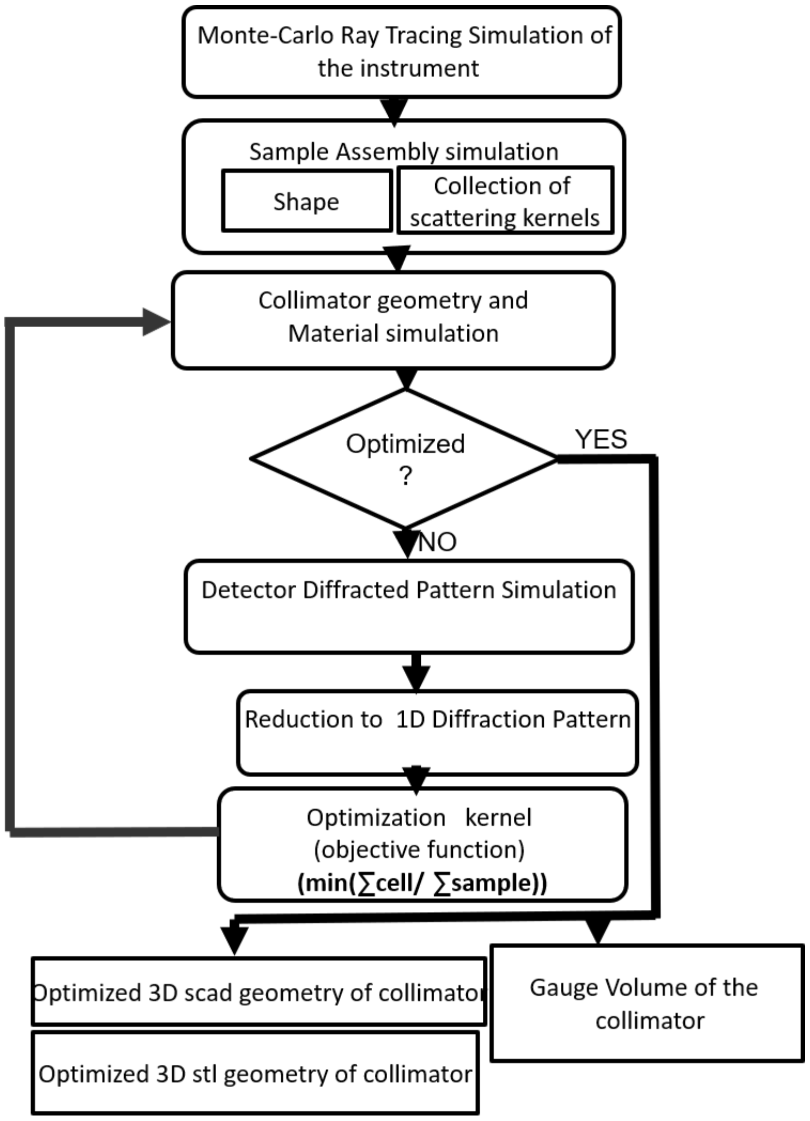
C3DP

1.1 Automated design of 3D printed collimator optimized for high pressure diffraction

1.2 Features

description of the simulation

- Simulation of the the diffractometer
- Simulation of the pressure cells
- Optimization of the collimator for the given pressure cell
- Produced the .stl or .scad file of the collimator to be 3D printed
- Produced the gauge volume of the collimator


Monte-Carlo Ray Tracing Simulation of the instrument
Sample Assembly simulation
Shape
Collection of scattering kernels
Collimator geometry and Material simulation
Optimized ?
YES
NO
Detector Diffracted Pattern Simulation
Reduction to 1D Diffraction Pattern
Optimization kernel (objective function) (min(∑cell/ ∑sample))
Optimized 3D scad geometry of collimator
Optimized 3D stl geometry of collimator
Gauge Volume of the collimator

1.3 Usage

docs/image/with_python.png

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install c3dp, run this command in your terminal:

```
$ conda install -c fi0 c3dp (recommended)
$ pip install c3dp
```

This is the preferred method to install c3dp, as it will always install the most recent stable release.

If you don’t have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for c3dp can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/fahima-islam/c3dp
```

Or download the tarball:

```
$ curl  -OL https://github.com/fahima-islam/c3dp/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER THREE

USAGE

To use c3dp in a project:

```
import c3dp
```

3.1 Tutorials and Examples

optimization of neutron scattering instrument

CHAPTER

FOUR

MODULES

4.1 Peak_detection : Detecting local maxima and minima in a signal

`c3dp.analysis.Peak_Detection.`**`peakdetect`**`(`*y_axis*, *x_axis=None*, *lookahead=200*, *delta=0*`)`

function for detecting local maxima and minima in a signal. Discovers peaks by searching for values which are surrounded by lower or larger values for maxima and minima respectively

keyword arguments: y_axis – A list containing the signal over which to find peaks

x_axis – A x-axis whose values correspond to the y_axis list and is used in the return to specify the position of the peaks. If omitted an index of the y_axis is used. (default: None)

lookahead – distance to look ahead from a peak candidate to determine if it is the actual peak (default: 200) '(samples / period) / f' where '4 >= f >= 1.25' might be a good value

delta – this specifies a minimum difference between a peak and the following points, before a peak may be considered a peak. Useful to hinder the function from picking up false peaks towards to end of the signal. To work well delta should be set to delta >= RMSnoise * 5. (default: 0)

When omitted delta function causes a 20% decrease in speed. When used Correctly it can double the speed of the function

return: two lists [max_peaks, min_peaks] containing the positive and negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

4.2 atomicPercentage_from_weightPercentage : Calculating the atomic percentage from weight percentage

`c3dp.instruments.atomicPercentage_from_weightPencentage.`**`atomic_percentage_from_weight_percentage`**`(`*weight_*
*atomic_*

calculating atomic percentage from weight percentage

Parameters

- **weight_percentage** (*ordered dictionary*) – elements as keys and their weight percentage as values in a component
- **atomic_weight** (*ordered dictionary*) – elements as keys and their atomic weights as values in a component

Returns **atomic percentage** – elements as keys and their atomic percentage as values in a component

Return type ordered dictionary

4.3 cad : converting .xml file to .scad file

4.4 convert2nxs : converting Numpy (.npy) file to Events Nexus (.nxs) file

4.5 from_d_toTOF : conversion to time of flight from d-spacing

`c3dp.reduction.from_d_toTOF.`**`tof_from_d`**`(`*d*, *angle*, *l1=14.699*, *l2=0.3*, *l3=0.5*, *xA=0.0*, *yA=0.0*, *zA=0.0*`)`

unit conversion from d spacing to time of flight

d [float] d spacing in Angstrom.

angle [float] scattering angles in degrees.

l1 [float] from source to guide exit distance in meters

l2 [float] from guide to sample distance in meters

l3 [float] from sample to detector distance in meters

xA [float] pixel position along x-axis with respect to detector center in meters

yA [float] pixel position along y-axis with respect to detector center in meters

zA [float] pixel position along z-axis with respect to detector center in meters

Returns

Return type time of flight in seconds

4.6 gauge_volume : Creating the gauge volume by the collimator

`c3dp.gaugevol.gauge_volume.`**`angle2span`**`(`*Verticle_distance*, *angle*`)`

Parameters

- **Verticle_distance**
- **angle**

`c3dp.gaugevol.gauge_volume.`**`gauge_volume`**`(`*square_theta_phy_sample*, *square_theta_phy_detector*, *sample_points_x_y*`)`

Calculate the non-zero gauge volume for different positions of the sample.

Parameters

- **square_theta_phy_sample (*tuple*) – the tuple of theta, phi of the four points of the collimator squared openin** sample side where each element of the tuple is the array of theta/phi of four points of the collimator for a particular point of the sample. e.g. (array([theta1, theta2, theta3, theta4]), array([phi1, phi2, phi3, phi4]))

  square_theta_phy_detector [tuple] the tuple of theta, phi of the four points of the collimator squared opening at detector side where each element of the tuple is the array of theta/phi of four points of the collimator for a particular point of the sample. e.g. (array([theta1, theta2, theta3, theta4]), array([phi1, phi2, phi3, phi4]))
- **sample_points_x_y** (`ndarray`) – array of two coordinates of the sample (x,y)

Returns

- *the tuple of positions in the sample (array([y,z])) (where the gaauge volume is non-zero) , corresponding gauge volume (list)*
- *where the position of the sample is an array, i.e. (array([x,y]))*

`c3dp.gaugevol.gauge_volume.`**`make_cylindrical_surface`**(*channel_start_from_sample_center*, *angle*, *height*, *length_misalignment_offset=0.0*, *height_misalignment_offset=0.0*)

create a cylinder which axis is along the vertical axis (z- axis)

Parameters

- **channel_start_from_sample_center** (*float*) – Longitudinal coordinate of the collimator (radius of the cylinder).
- **height** (*float*) – Height of the collimator channel (height of the cylinder).
- **angle** (*degree*) – angular size of collimator channel (curvature of the cylinder)
- **length_misalignment_offset** (*float*) – misalignment offset along the cylinder radius
- **height_misalignment_offset** (*float*) – misalignment offset along the cylinder axis

Returns

Return type the list of the four points of the collimator channel's cylindrical opening

`c3dp.gaugevol.gauge_volume.`**`make_square`**(*x*, *size*, *length_misalignment_offset=0.0*, *misalignment_offset=0.0*)

create a square with the longitudinal coordinate and the height/width of the collimator .

Parameters

- **x** (*float*) – Longitudinal coordinate of the collimator.
- **size** (*float*) – Height or width of the collimator (collimator is square).
- **length_misalignment_offset** (*float*) – misalignment offset along the cylinder radius
- **misalignment_offset** (*float*) – misalignment offset along the vertical and transversal axis

Returns

Return type the list of the four points of the collimator squared opening `c3dp.gaugevol.gauge_volume.`**`making_plot`**`(`*sample_points_x_y_nonZero*, *gauge_volume*, *y_upper_imit*, *y_lower_limit*, *sample_height=10*, *sample_width=5.0*, *min_color=None*, *max_color=None*`)`

Saved the contour of the gauge volume in different positions of the sample in "Figure directory".

Parameters

- **sample_points_x_y_nonZero** (`ndarray`) – array of two coordinates of the sample (x,y) points where gauge volume is non-zero
- **gauge_volume** (*list*) – list of the non zero gauge volumes of different positions of the sample
- **y_upper_imit** (*float*) – the upper limit of Y-axis for the plotting view
- **y_lower_imit** (*float*) – the lower limit of Y-axis for the plotting view

`c3dp.gaugevol.gauge_volume.`**`non_center_channels`**`(`*channel_at_center*`)`

create a square with the longitudinal coordinate and the height/width of the collimator .

Parameters

- **x** (*float*) – Longitudinal coordinate of the collimator.
- **size** (*float*) – Height or width of the collimator (collimator is square).

Returns

Return type the list of the four points of the collimator squared opening

`c3dp.gaugevol.gauge_volume.`**`rotation_around_x_axis`**`(`*vector_point*, *rotation_angle*`)`

create a vector point after rotating a vector around x axis in 3D space anticlockwise

vector_point [list]

list of the three coordinates of a point

rotation_angle [degree] angle to rotate the vector.

the array of the rotated vector consisting of three coordinates of the vector

`c3dp.gaugevol.gauge_volume.`**`rotation_around_y_axis`**`(`*vector_point*, *rotation_angle*`)`

create a vector point after rotating a vector around y axis in 3D space anticlockwise

vector_point [list]

list of the three coordinates of a point

rotation_angle [degree] angle to rotate the vector.

the array of the rotated vector consisting of three coordinates of the vector

`c3dp.gaugevol.gauge_volume.`**`rotation_around_z_axis`**`(`*vector_point*, *rotation_angle*`)` create a vector point after rotating a vector around z axis in 3D space anticlockwise

vector_point [list]

list of the three coordinates of a point

rotation_angle [degree] angle to rotate the vector.

the array of the rotated vector consisting of three coordinates of the vector

`c3dp.gaugevol.gauge_volume.`**`span2angle`**`(`*distance*, *distance_fr_sample*`)`

Parameters

- **distance**
- **distance_fr_sample**

`c3dp.gaugevol.gauge_volume.`**`theta_phi`**`(`*Collimator_square*, *sample_point*`)`

Calculate the spherical coordinate(theta and phi) of the four points of the square collimator from the sample.

Parameters

- **Collimator_square** (*list*) – List of the four points of the collimator openning cross-section .
- **sample_point** (`ndarray`) – array of three coordinates of the sample (x,y,z).T

Returns

- *the tuple of theta, phi of the four points of the collimator squared opening*
- *where each element of the tuple is the array of theta/phi of four points of the*
- *collimator for a particular point of the sample*

4.7 normalization_by_area : normalizing the curve by integrated area

`c3dp.analysis.normalization_by_area.`**`area_under_curve`**`(`*y*, *dx*, *method=None*`)`
Calculate the area under the curve

Parameters **y** (*array_like*) – Input array to integrate

dx [scalar] The spacing between sample points

Returns **area** – Integrated area

Return type float

`c3dp.analysis.normalization_by_area.`**`normalization`**`(`*y*, *area*`)`
normalizing the curve by integrated area

Parameters **y** (*array_like*) – Input array to integrate

area: float Integrated area

Returns **normalized_data** – normalized curve by area

Return type array_like

4.8 sampleassembly_program : Creating the template for sample assembly

`c3dp.instruments.sampleassembly_program.`**`makeSAXML`**(*sampleassembly_fileName*, *pathTosave*, *scatterers={('collimator', 'shapeColl', 'coll_geometry', 'B4C', 'B4C.cif', 'cif'), ('inner-sleeve', 'shapeCu', 'inner-sleeve-geom', 'Cu', 'Cu.xyz', 'xyz'), ('outer-body', 'shapeAl', 'outer-body-geom', 'Al', 'Al.xyz', 'xyz'), ('sample', 'shapeSample', 'sample_geom', 'Si', 'Si.xyz', 'xyz')}*)

creating the sample assembly file on which mcvine simulation will run

Parameters

- **sampleassembly_fileName** (*string*) – file name of the sampleassembly
- **pathTosave** (*string*) – the path where the sample assembly file will be saved
- **scatterers** (*set*) – the set of scatterer information : scatterer name, shape name, shape file name, chemical formula, structure file path, structure file extension

4.9 scattering_kernal_program : Creating the template for scattering kernel

`c3dp.instruments.scatkernel.scattering_kernal_program.`**`makeSKXML`**(*kernel_type*, *path_ToSave_ScatteringKernel*, *scatterer_type_name*, *scatterer=None*, *absorption=0*, *scattering=1*, *transmission=3*, *Dd_over_d=1e-05*, *DebyeWaller_factor=1*, *E=None*, *S_Q_E=None*, *Qmin=None*, *Qmax=None*)

making scattering kernel xml file

kernel_type: string if the 'elastic or inelastic' kernel to specify

scatterer: string peaks location of the scatterer

path_ToSave_ScatteringKernel [string] path where the scattering kernel file would be saved

scatterer_type_name: string name of the scatterer type (sample/ cell)

absorption: int weight for absorption

transmission: int weight for transmission

Dd_over_d: float d spacing resolution in Angstrom

DebyeWaller_factor: int Debye Waller factor

4.10 section : properties of different sections of the collimator

class `c3dp.instruments.collimator.section.`**`CollimatorSection`**(*thickness*, *sample_distance*, *aperture*)

Bases: `object`

thickness: float Distance from upstream and downstream collimator faces, in mili-meters

sample_distance: float Distance from the sample to the middle of the section, in mili-meters

aperture: float Angle subtended by the section to the sample, in degrees

`blade_blade_angle_distances`(*blade_angles*)

In the boundary between two collimator sections, the interior blades of each collimator section may overlap on top of each other. Here we look at the upstream face of the collimator section, that is the boundary between this section and the collimator section immediately nearer to the sample. For each interior blade of the collimator section, we calculate the angle difference to each of the blades of the neighbor collimator section, retaining only the minimum of these differences. In the end, we obtain an angle difference between each blade of the collimator section and the set of blades from the neighbor collimator section.

Parameters **blade_angles** (*numpy.ndarray*) – Blade angle positions for the neighbor collimator section

Returns Angle differences, in degrees

Return type numpy.ndarray

`blade_blade_distances`(*blade_angles*)

In the boundary between two collimator sections, the interior blades of each collimator section may overlap on top of each other. Here we look at the upstream face of the collimator section, that is the boundary between this section and the collimator section immediately nearer to the sample. For each interior blade of the collimator section, we calculate the distance to each of the blades of the neighbor collimator section, retaining only the minimum of these distances. In the end, we obtain a distance between each blade of the collimator section and the set of blades from the neighbor collimator section.

Parameters **blade_angles** (*numpy.ndarray*) – Blade angle positions for the neighbor collimator section

Returns Distances between neighboring blades, in mili-meters

Return type numpy.ndarray

`chanel_thickness`(*n_blades=None*)

Chanel thickness given a number of interior blades. Blade thickness is also taken into account.

Calculations using the upstream face of the collimator section.

Parameters **n_blades** (*int*) – Number of interior blades. If none, the number of blade angles is used

Returns

Return type float

`minimal_blade_blade_distance`(*blade_angles*)

Minimal distance between the set of interior blades of the collimator section and the set of interior blades of the collimator section immediately nearer to the sample.

Parameters **blade_angles** (*numpy.ndarray*) – Blade angle positions for the neighbor collimator section

Returns Minimal distance, in mili-meters

Return type float

`property n_blades`

`n_blades_from_thickness`(*channel_thickness*)

Number of interior blades given a channel thickness. Blade thickness is also taken into account.

Calculations using the upstream face of the collimator section

Parameters **channel_thickness** (*float*)

Returns

Return type int

`property sample_upstream_distance`

Distance from sample to upstream collimator face

`set_blade_angles`(*n_blades*)

Angle positions of the blades. Origin of angles at the top surface.

Parameters **n_blades** (*int*) – number of interior blades

Returns

Return type numpy.ndarray

`class c3dp.instruments.collimator.section.Triad`(*blades*, *d*, *widths*)

Bases: `tuple`

`blades`

Alias for field number 0

`d`

Alias for field number 1

`widths`

Alias for field number 2

`c3dp.instruments.collimator.section.blade_configurations`(*upstream_distance*, *collimator_thickness*, *aperture*, *minimum_channel_width=3.0*, *blade_thickness=1.0*, *blade_gap=1.1*)

List of blade configurations. Each configuratio avoids blade overlap between adjacent collimator sections.

Parameters

- **upstream_distance** (*float*) – Distance from sample to the upstream face of the collimator.
- **thickness** (*float*) – Distance of each collimator section. Assumed all three same thickness.
- **aperture** (*float*) – Collimator aperture angle, in degrees
- **minimum_channel_width** (*float*) – Units in mili meters
- **blade_thickness** (*float*) – Units in mili meters
- **blade_gap** (*float*) – Minimal distance between blades from adjacent sections

Returns List of Triad objects, that can later be filtered and sorted

Return type list

`c3dp.instruments.collimator.section.sort_filter_confs`(*confs*, *min_n_blades=None*, *max_channel_width=None*, *sort_by_total_blades=False*, *sort_max_section_blades=None*)

Sort or filter according to different criteria :Parameters: * **confs** (*list*) – List of Triad objects

- **min_n_blades** (*int*) – Discard Triads having one collimator section with a number of blades smaller than this number.
- **max_channel_width** (*float*) – Discard Triads having one collimator section with a channel width bigger than this value.
- **sort_by_total_blades** (*Bool*) – Sort by decreasing total number of blades in the collimator.
- **sort_max_section_blades** (*str*) – One of 'first', 'middle', 'last'. Sort list of Triads according to decreasing number of blades in either the 'first', 'middle', or 'last' collimator section.

Returns sorted or filtered list of Triad objects

Return type list

`c3dp.instruments.collimator.section.`**`valid_blade_pair_configurations`**(*upstream_section*, *max_upstream_blades*, *downstream_section*, *max_downstream_blades*, *tolerance=1.0*)

List of pairs, each pair containing a number of interior blades for the upstream collimator section and a number of interior blades for the downstream collimator section. For each valid pair, it is guaranteed that no blade from the upstream collimator is closer than the tolerance distance to any blade from the downstream collimator. Also for each pair, the minimal blade-to-blade distance is reported

Parameters

- **upstream_section** (*CollimatorSection*) – Upstream collimator section
- **max_upstream_blades** (*int*) – Maximum number of interior blades for the upstream collimator section
- **downstream_section** (*CollimatorSection*) – Downstream collimator section
- **max_downstream_blades** (*int*) – Maximum number of interior blades for the downstream collimator section
- **tolerance** (*float*) – Minimal blade-to-blade distance between adjacent collimator sections.

Returns Tuple containing a list of valid pairs and a list of minimal distances

Return type tuple

`c3dp.instruments.collimator.section.`**`valid_blade_triad_configurations`**(*sections*, *max_blades*, *tolerance=1.0*)

List of triads, each triad containing number of interior blades for each collimator sections. For each triad, it is guaranteed that no blade from a collimator section is closer than the tolerance distance to any blade from the adjacent collimator(s).

Parameters

- **sections** (*list*) – List of collimator sections, beginning with the section closer to the sample
- **max_blades** (*list*) – Maximum number of interior blades per collimator section.
- **tolerance** (*float*) – Minimal blade-to-blade distance between adjacent collimator sections.

Returns a list of Triad objects

Return type list

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/fahima-islam/c3dp/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

c3dp could always use more documentation, whether as part of the official c3dp docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/fahima-islam/c3dp/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *c3dp* for local development.

1. Fork the *c3dp* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/c3dp.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv c3dp
   $ cd c3dp/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 c3dp tests
   $ python setup.py test or py.test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/fahima-islam/c3dp/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_c3dp
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SIX

CREDITS

6.1 Development Lead

- Fahima islam <ffiqnf.2017@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER SEVEN

HISTORY

7.1 0.1.0 (2019-04-22)

- First release on PyPI.

The Python ecosystem is an ideal environment for developing full-circle applications for merging collimator design, experimental planning and optimization, and 3D printing for neutron scattering instruments. We present a Python package, c3dp, that uses numpy, scipy, h5py, shapely, and Matplotlib to design, simulate, optimize and visualize a collimator's performance quickly, accurately, and finally convert the optimized configuration straight to a format ready for 3D printing for diamond anvil or clamp pressure cells used in neutron diffraction experiments on the SNAP beamline. The package includes Monte Carlo ray tracing of the SNAP instrument, the collimator geometry, and simulates the neutron interaction with the collimator and the optimization of the collimator geometry to produce the best configuration. A differential evolution algorithm from the SciPy library was used for optimization with the objective of minimizing the simulated background, and a Jupyter notebook is used to integrate each of the steps of the package into a design and optimization work flow.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX