scanpointgenerator Documentation

Release 2.1.1

Tom Cobb

May 04, 2018

Overview

CHAPTER 1

Scan Point Generator

Scan point generator contains a number of python iterators that are used in GDA and malcolm to determine the motor demand positions and dataset indexes that various scan types will produce

1.1 Installation

To install the latest release, type:

```
pip install scanpointgenerator
```

To install the latest code directly from source, type:

```
pip install git+git://github.com/dls-controls/scanpointgenerator.git
```

1.2 Changelog

See CHANGELOG

1.3 Contributing

See CONTRIBUTING

1.4 License

APACHE License. (see LICENSE)

1.5 Documentation

Full documentation is available at http://scanpointgenerator.readthedocs.org

CHAPTER 2

Architecture

Scan points are produced by a *Compound Generator* that wraps base generators, *Excluders* and *Mutators*.

All Generators inherit from the Generator baseclass, which provides the following API:

class scanpointgenerator.**Generator**
Base class for all malcolm scan point generators

Variables

- **units** (*dict*) – Dict of str position_name -> str position_unit for each scannable dimension. E.g. {"x": "mm", "y": "mm"}
- ***axes*** (*list*) – List of scannable names, used in GDA to reconstruct Point in CompoundGenerators

prepare_arrays(*index_array*)
Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

to_dict()
Abstract method to convert object attributes into a dictionary

classmethod from_dict(*d*)
Abstract method to create a ScanPointGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

Returns New ScanPointGenerator instance

Return type *Generator*

classmethod register_subclass(*generator_type*)
Register a subclass so from_dict() works

Parameters **generator_type** (Generator) – Subclass to register

Each point produced by the iterator represents a scan point, with the following API:

class scanpointgenerator.**Point**

Contains information about for each scan point

Variables

- **positions** (*dict*) – Dict of str position_name -> float position for each scannable dimension. E.g. {"x": 0.1, "y": 2.2}
- ***lower*** (*dict*) – Dict of str position_name -> float lower_bound for each scannable dimension. E.g. {"x": 0.95, "y": 2.15}
- ***upper*** (*dict*) – Dict of str position_name -> float upper_bound for each scannable dimension. E.g. {"x": 1.05, "y": 2.25}
- **indexes** (*list*) – List of int indexes for each dataset dimension, fastest changing last. E.g. [15]
- **duration** (*int*) – Int or None for duration of the point exposure

2.1 Using the API

A basic use case that uses two generators looks like this:

```
cgen = CompoundGenerator([outer_generator, inner_generator], [], [])
cgen.prepare()
for point in cgen.iterator():
    for mname, mpos in point.positions():
        motors[mname].move(mpos)
    det.write_data_to_index(point.indexes)
```

CHAPTER 3

Line Generator

class scanpointgenerator.**LineGenerator**(*axes*, *units*, *start*, *stop*, *size*, *alternate=False*)

Generate a line of equally spaced N-dimensional points

Parameters

- **axes** (*str/list(str)*) – The scannable axes E.g. “x” or [“x”, “y”]
- **units** (*str/list(str)*) – The scannable units. E.g. “mm” or [“mm”, “mm”]
- **start** (*float/list(float)*) – The first position to be generated. e.g. 1.0 or [1.0, 2.0]
- **stop** (*float or list(float)*) – The final position to be generated. e.g. 5.0 or [5.0, 10.0]
- **size** (*int*) – The number of points to generate. E.g. 5
- **alternate** (*bool*) – Specifier to reverse direction if generator is nested

prepare_arrays(*index_array*)

Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

to_dict()

Convert object attributes into a dictionary

classmethod from_dict(*d*)

Create a LineGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

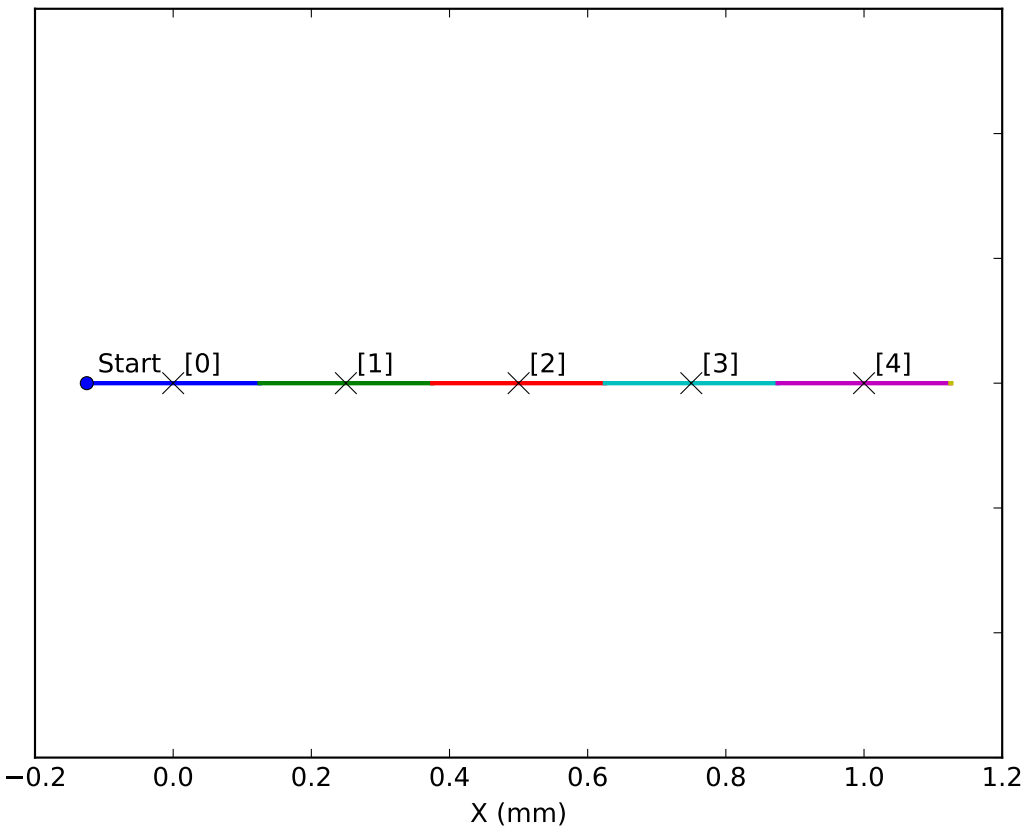
Returns New LineGenerator instance

Return type *LineGenerator*

3.1 Examples

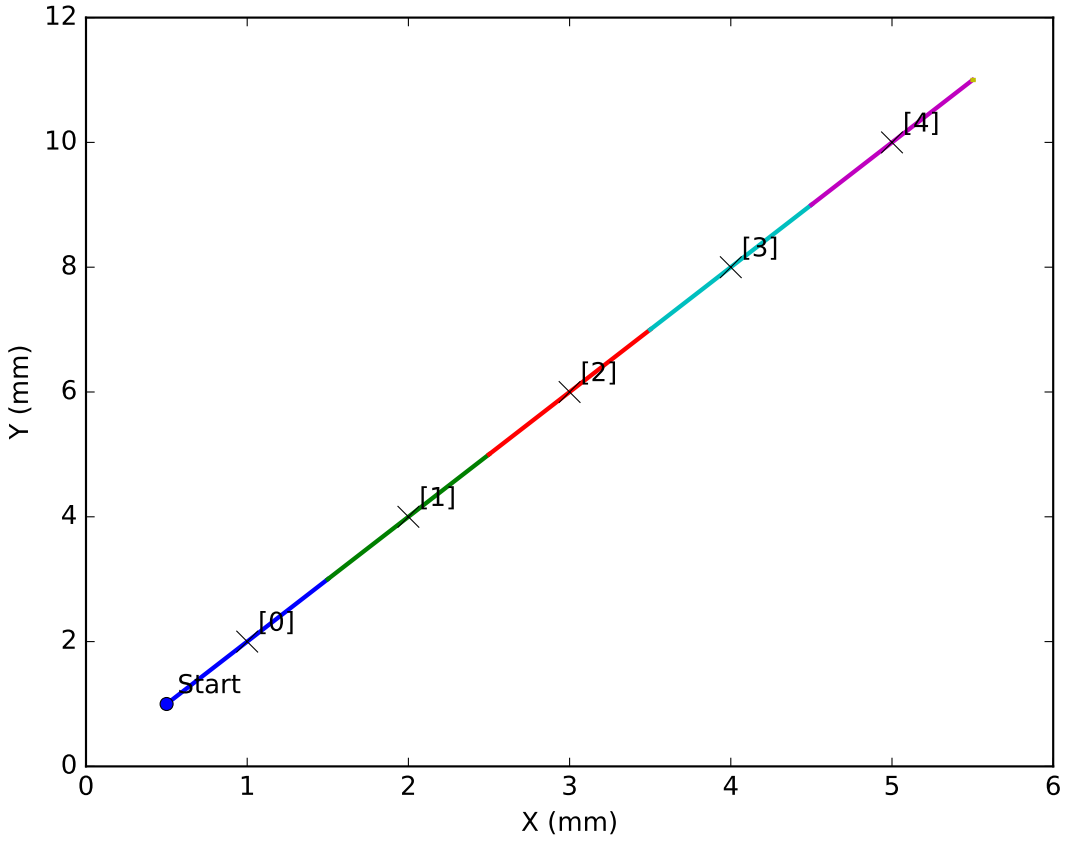
This example defines a motor "x" with engineering units "mm" which is being scanned from 0mm to 1mm with 5 scan points inclusive of the start. Note that the capture points are as given, so the bounds will be +-0.5*step of each capture point.

```
from scanpointgenerator import LineGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = LineGenerator("x", "mm", 0.0, 1.0, 5)
plot_generator(gen)
```



LineGenerator is N dimensional; just pass in ND lists for name, start and stop.

```
from scanpointgenerator import LineGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = LineGenerator(["x", "y"], ["mm", "mm"], [1.0, 2.0], [5.0, 10.0], 5)
plot_generator(gen)
```


12
10
8
6
4
2
0
Y (mm)
0
1
2
3
4
5
6
X (mm)
Start
[0]
[1]
[2]
[3]
[4]

CHAPTER 4

Spiral Generator

class scanpointgenerator.**SpiralGenerator**(*axes*, *units*, *centre*, *radius*, *scale=1.0*, *alternate=False*)

Generate the points of an Archimedean spiral

Parameters

- **axes** (*list(str)*) – The scannable axes e.g. ["x", "y"]
- **units** (*list(str)*) – The scannable units e.g. ["mm", "mm"]
- **centre** (*list*) – List of two coordinates of centre point of spiral
- **radius** (*float*) – Maximum radius of spiral
- **scale** (*float*) – Gap between spiral arcs; higher scale gives fewer points for same radius
- **alternate** (*bool*) – Specifier to reverse direction if generator is nested

prepare_arrays(*index_array*)

Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

to_dict()

Convert object attributes into a dictionary

classmethod from_dict(*d*)

Create a SpiralGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

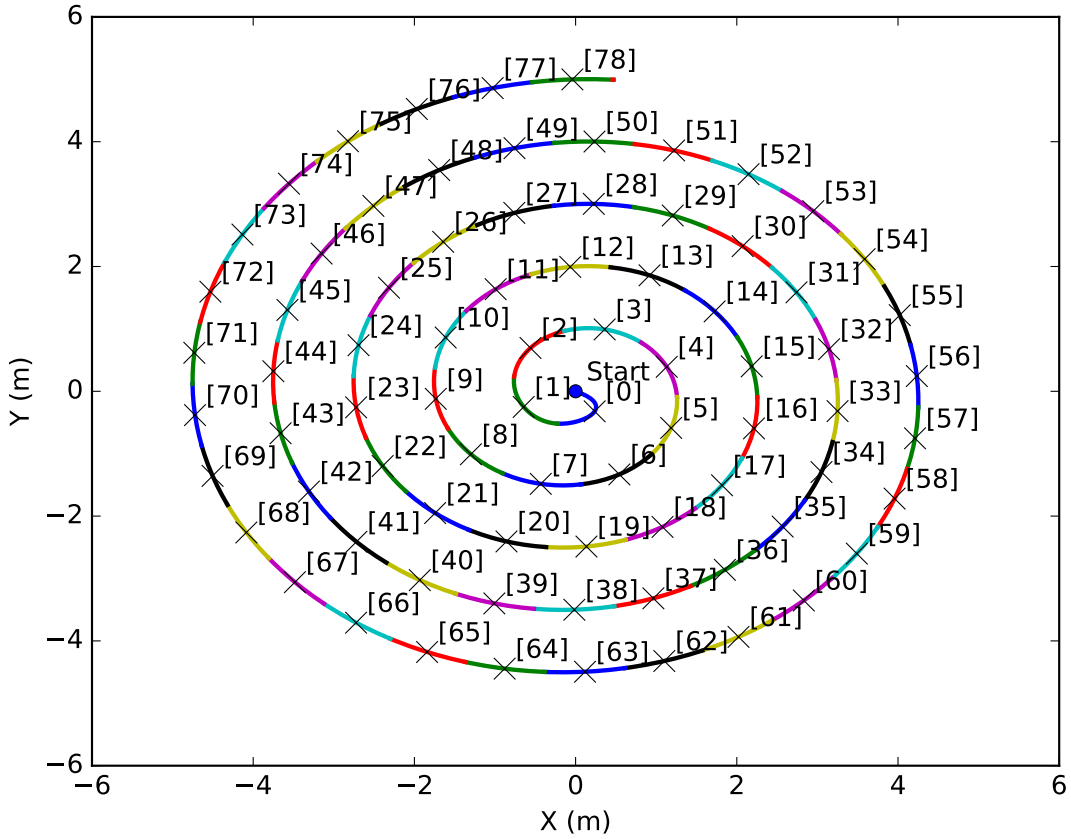
Returns New SpiralGenerator instance

Return type *SpiralGenerator*

4.1 Examples

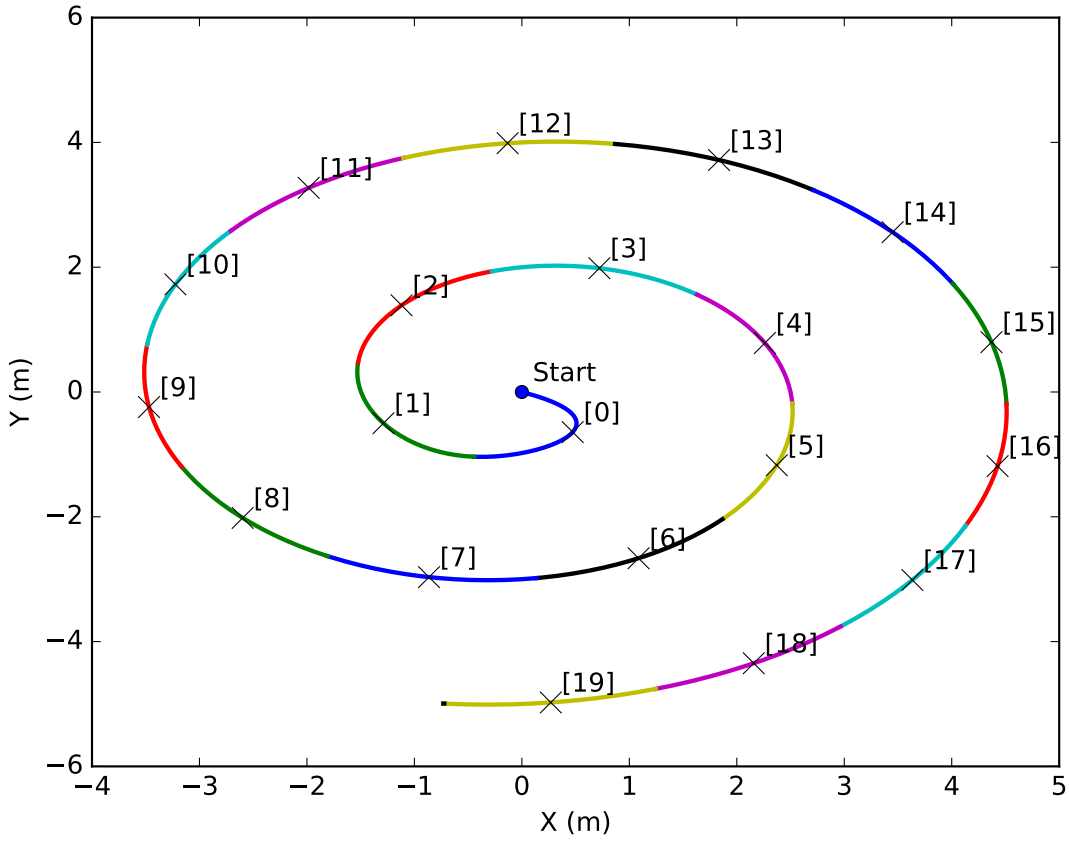
This example defines motors “x” and “y” with engineering units “mm” which will be scanned in a spiral filling a circle of radius 5mm.

```
from scanpointgenerator import SpiralGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = SpiralGenerator(["x", "y"], "mm", [0.0, 0.0], 5.0)
plot_generator(gen)
```



In this example the spiral is scaled to be more sparse.

```
from scanpointgenerator import SpiralGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = SpiralGenerator(["x", "y"], "mm", [0.0, 0.0], 5.0, scale=2.0)
plot_generator(gen)
```


Start
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
X (m)
Y (m)

CHAPTER 5

Lissajous Generator

class scanpointgenerator.**LissajousGenerator**(*axes*, *units*, *centre*, *span*, *lobes*, *size=None*, *alternate=False*)

Generate the points of a Lissajous curve

Parameters

- **axes** (*list(str)*) – The scannable axes e.g. ["x", "y"]
- **units** (*list(str)*) – The scannable units e.g. ["mm", "mm"]
- **centre** (*list(float)*) – The centre of the lissajous curve
- **span** (*list(float)*) – The [height, width] of the curve
- **num** (*int*) – Number of x-direction lobes for curve; will have lobes+1 y-direction lobes
- **size** (*int*) – The number of points to fill the Lissajous curve. Default is 250 * lobes

prepare_arrays(*index_array*)

Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

to_dict()

Convert object attributes into a dictionary

classmethod from_dict(*d*)

Create a LissajousGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

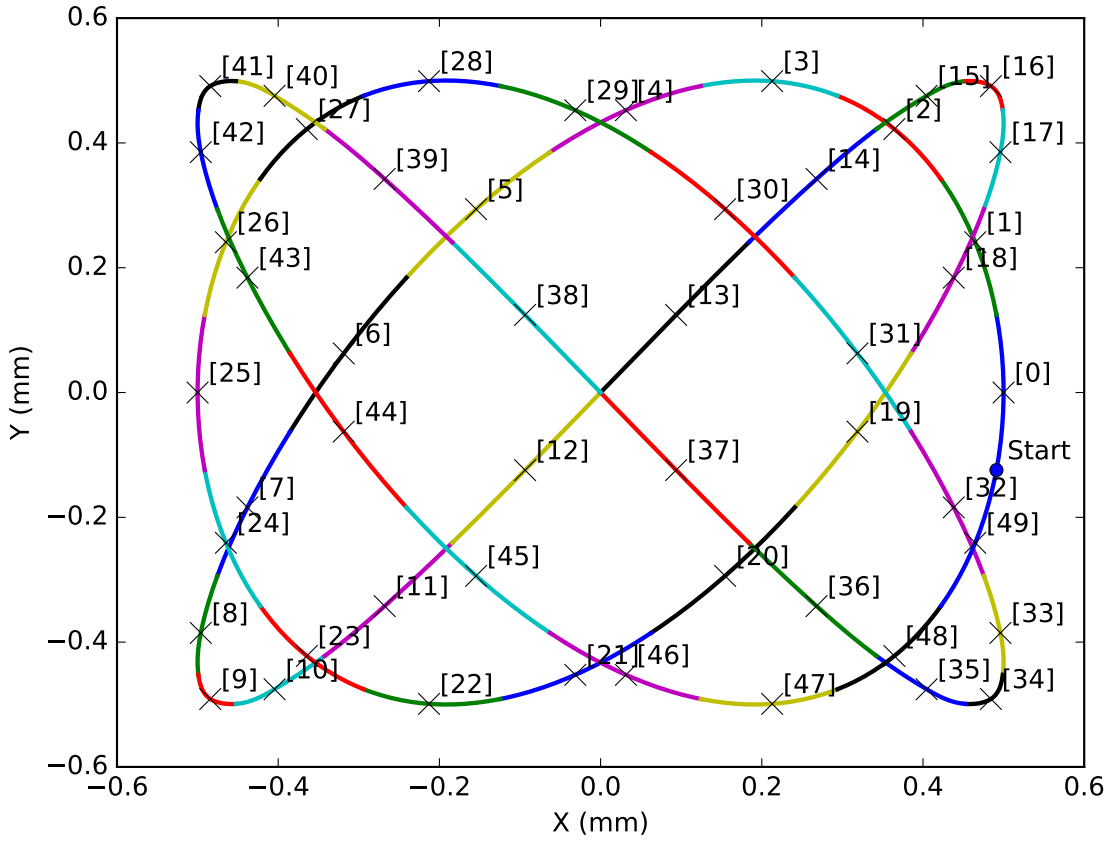
Returns New LissajousGenerator instance

Return type *LissajousGenerator*

5.1 Examples

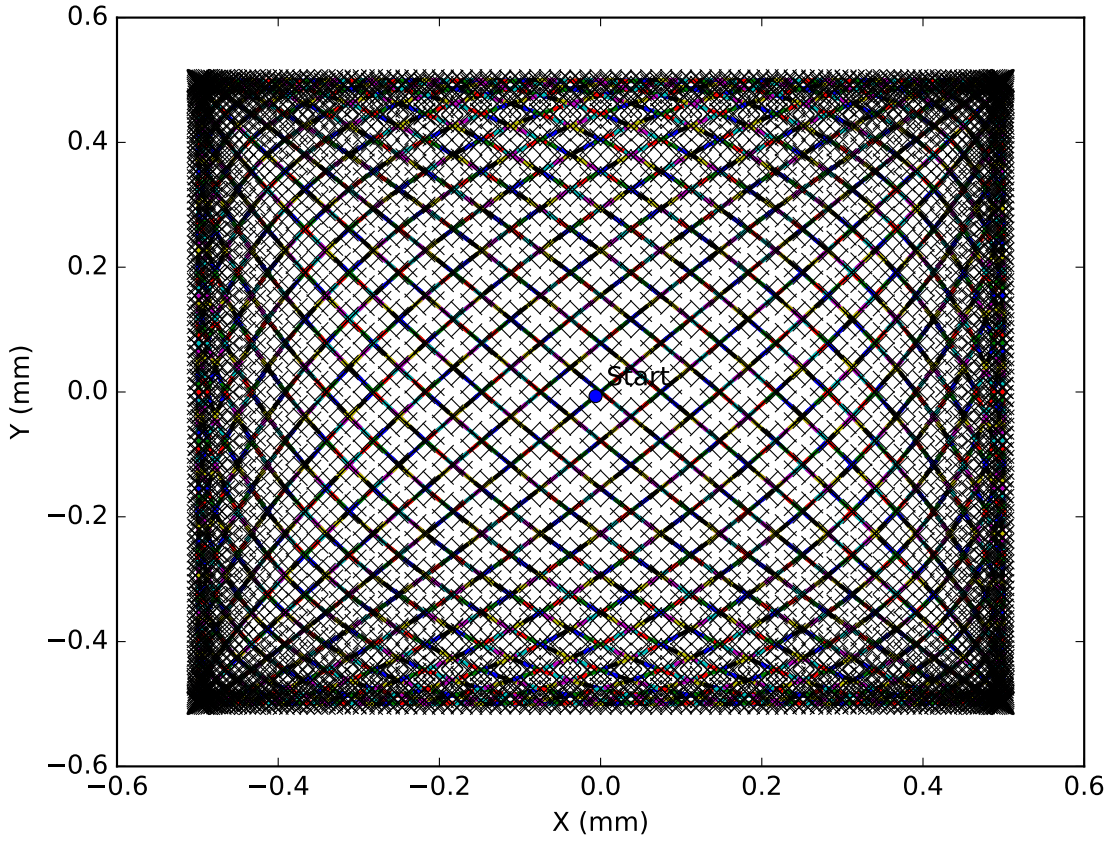
This example defines motors “x” and “y” with engineering units “mm” which will be scanned over a 3x4 lobe Lissajous curve with filling a 1x1mm rectangle.

```
from scanpointgenerator import LissajousGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = LissajousGenerator(['x', 'y'], ["mm", "mm"], [0.0, 0.0], [1.0, 1.0],
        lobes=3, size=50)
plot_generator(gen)
```



The number of points has been lowered from the default to make the plot more visible. The following plot is for 10x11 lobes with the default number of points.

```
from scanpointgenerator import LissajousGenerator
from scanpointgenerator.plotgenerator import plot_generator

gen = LissajousGenerator(['x', 'y'], ["mm", "mm"], [0.0, 0.0], [1.0, 1.0],
        lobes=20)
plot_generator(gen, show_indexes=False)
```


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Y (mm)
Start
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
X (mm)

CHAPTER 6

Array Generator

class scanpointgenerator.**ArrayGenerator**(*axis*, *units*, *points*, *alternate=False*)

Generate points fron a given list of positions

Parameters

- **axis** (*str*) – The scannable axis name
- **units** (*str*) – The scannable units.
- **points** (*list(double)*) – array positions
- **alternate** (*bool*) – Alternate directions

prepare_arrays(*index_array*)

Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

to_dict()

Serialize ArrayGenerator to dictionary

classmethod from_dict(*d*)

Create a ArrayGenerator from serialized form.

Parameters **d** (*dict*) – Serialized generator

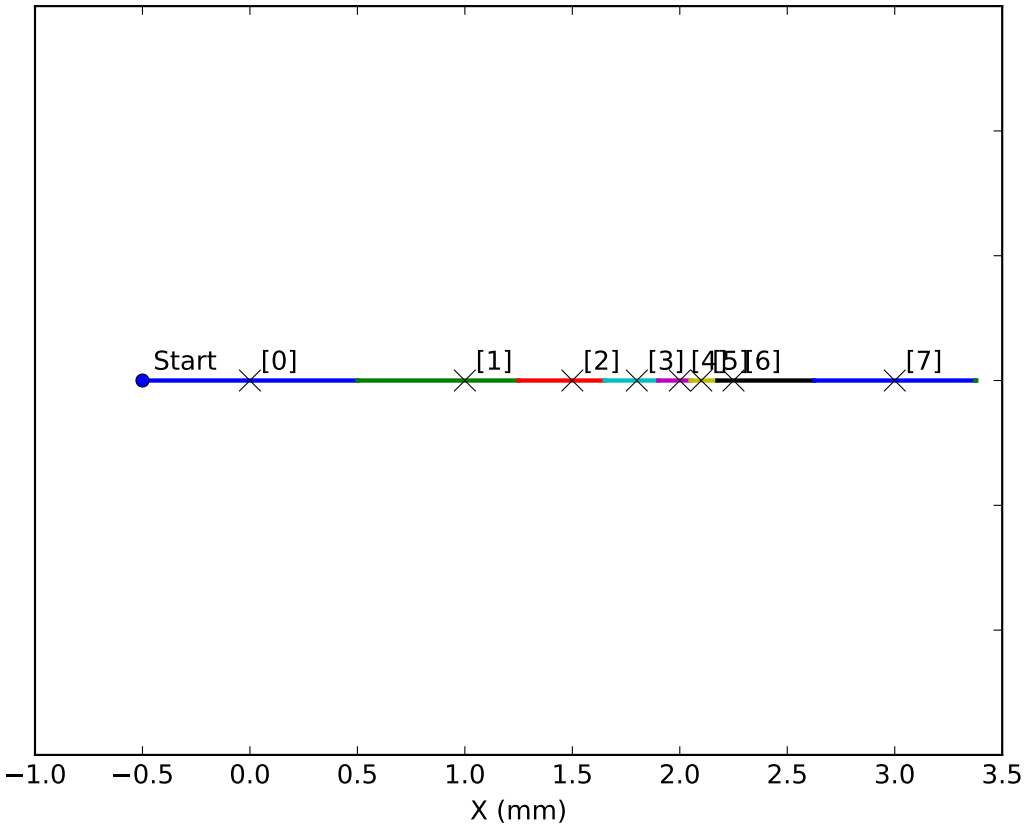
Returns New ArrayGenerator instance

Return type *ArrayGenerator*

6.1 Examples

This example defines a motor “x” with units “mm” which is being scanned over the series of positions [0, 1, 1.5, 1.8, 2, 2.1, 2.25, 3]

```
from scanpointgenerator import ArrayGenerator
from scanpointgenerator.plotgenerator import plot_generator

positions = [0, 1, 1.5, 1.8, 2, 2.1, 2.25, 3]
gen = ArrayGenerator("x", "mm", positions)
plot_generator(gen)
```



CHAPTER 7

Static Point Generator

class scanpointgenerator.**StaticPointGenerator**(*size*)

Generate 'empty' points with no axis information

to_dict()

Abstract method to convert object attributes into a dictionary

prepare_arrays(*index_array*)

Abstract method to create position or bounds array from provided index array. index_array will be np.arange(self.size) for positions and np.arange(self.size + 1) - 0.5 for bounds.

Parameters **index_array** (*np.array*) – Index array to produce parameterised points

Returns Dictionary of axis names to position/bounds arrays

Return type Positions

classmethod from_dict(*d*)

Abstract method to create a ScanPointGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

Returns New ScanPointGenerator instance

Return type *Generator*

7.1 Examples

Produce empty points to "multiply" existing generators within a *Compound Generator*, adding an extra dimension.

```
>>> from scanpointgenerator import StaticPointGenerator, LineGenerator,␣
↪CompoundGenerator
>>> line_gen = LineGenerator("x", "mm", 0.0, 1.0, 3)
>>> nullpoint_gen = StaticPointGenerator(2)
>>> gen = CompoundGenerator([nullpoint_gen, line_gen], [], [])
>>> gen.prepare()
```

(continues on next page)

(continued from previous page)

```
>>> [point.positions for point in gen.iterator()]
[{'x': 0.0}, {'x': 0.5}, {'x': 1.0}, {'x': 0.0}, {'x': 0.5}, {'x': 1.0}]
```

Using a StaticPointGenerator on its own in a compound generator is also allowed.

```
>>> from scanpointgenerator import StaticPointGenerator, CompoundGenerator
>>> nullpoint_gen = StaticPointGenerator(3)
>>> gen = CompoundGenerator([nullpoint_gen], [], [])
>>> gen.prepare()
>>> [point.positions for point in gen.iterator()]
[{}, {}, {}]
```

CHAPTER 8

Compound Generator

class scanpointgenerator.**CompoundGenerator**(*generators*, *excluders*, *mutators*, *duration=-1*, *continuous=True*)

Nest N generators, apply exclusion regions to relevant generator pairs and apply any mutators before yielding points

Parameters

- **generators** (*list(Generator)*) – List of Generators to nest
- **excluders** (*list(Excluder)*) – List of Excluders to filter points by
- **mutators** (*list(Mutator)*) – List of Mutators to apply to each point
- **duration** (*double*) – Point durations in seconds (-1 for variable)
- **continuous** (*boolean*) – Make points continuous (set upper/lower bounds)

size = None
int – Final number of points to be generated - valid only after calling prepare

shape = None
tuple(int) – Final shape of the scan - valid only after calling prepare

dimensions = None
list(Dimension) – Dimension instances - valid only after calling prepare

prepare()
Prepare data structures required for point generation and initialize size, shape, and dimensions attributes. Must be called before get_point or iterator are called.

iterator()
Iterator yielding generator positions at each scan point

Yields *Point* – The next point

get_point(*n*)
Retrieve the desired point from the generator

Parameters **n** (*int*) – point to be generated

Returns The requested point

Return type *Point*

to_dict()
Convert object attributes into a dictionary

classmethod from_dict(*d*)
Create a CompoundGenerator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

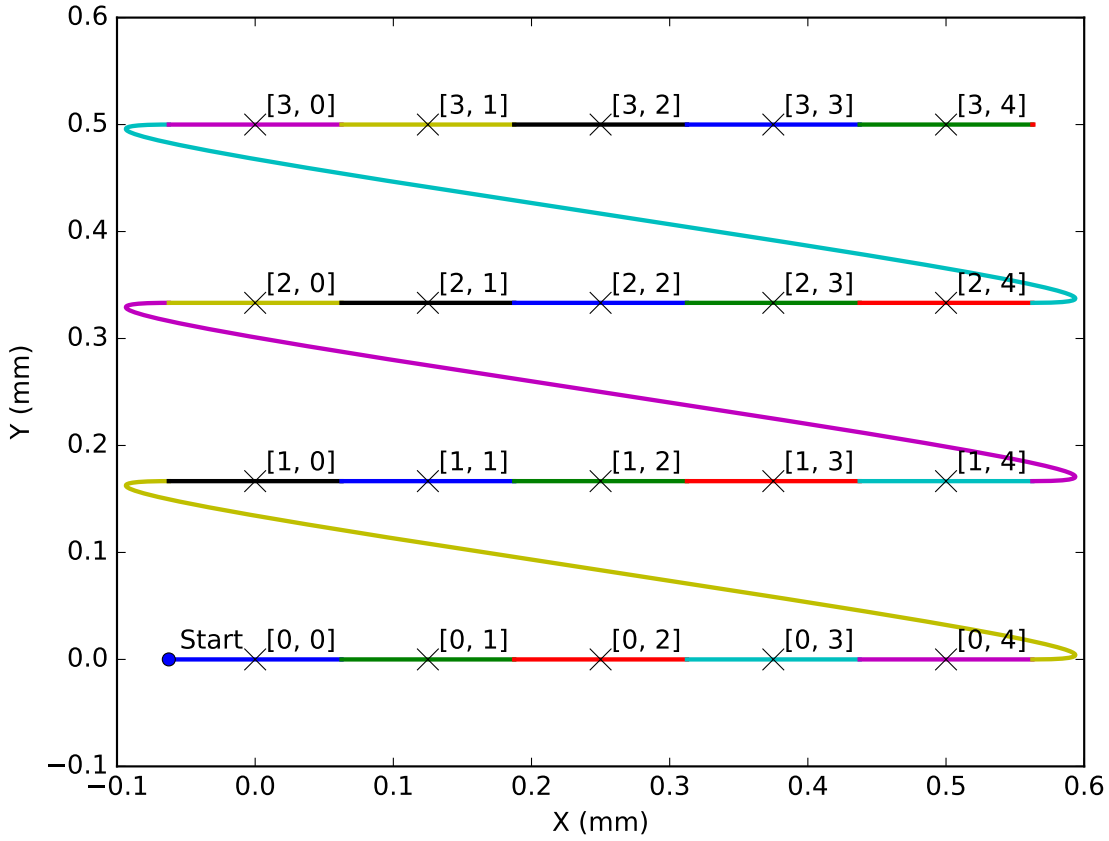
Returns New CompoundGenerator instance

Return type *CompoundGenerator*

8.1 Raster Scan Example

This scan will create an outer "y" line scan with 4 points, then nest an "x" line scan inside it with 5 points.

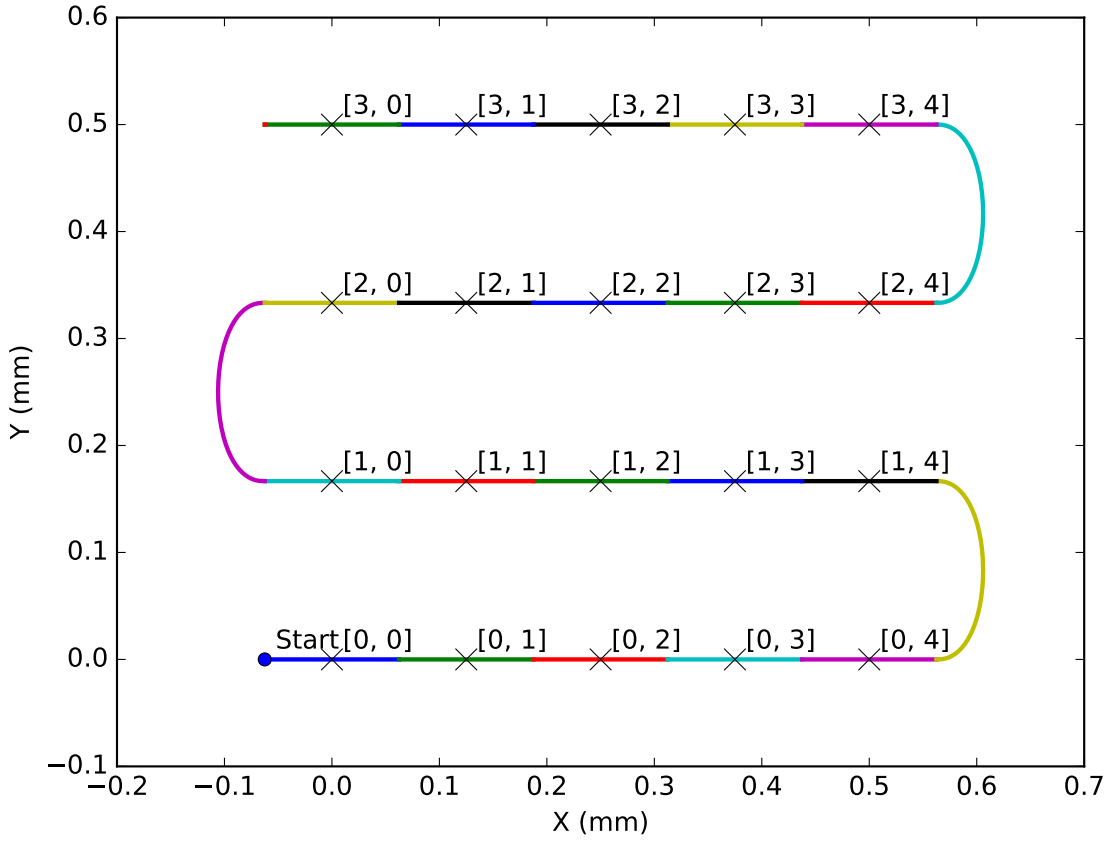
```
from scanpointgenerator import LineGenerator, CompoundGenerator
from scanpointgenerator.plotgenerator import plot_generator

xs = LineGenerator("x", "mm", 0.0, 0.5, 5, alternate=False)
ys = LineGenerator("y", "mm", 0.0, 0.5, 4)
gen = CompoundGenerator([ys, xs], [], [])
plot_generator(gen)
```



8.2 Snake Scan Example

This scan will create an outer "y" line scan with 4 points, then nest an "x" line scan inside it with 5 points. On every second row, the "x" line scan will be run in reverse to give a snake scan.

```
from scanpointgenerator import LineGenerator, CompoundGenerator
from scanpointgenerator.plotgenerator import plot_generator

xs = LineGenerator("x", "mm", 0.0, 0.5, 5, alternate=True)
ys = LineGenerator("y", "mm", 0.0, 0.5, 4)
gen = CompoundGenerator([ys, xs], [], [])
plot_generator(gen)
```



8.3 Restrictions

Generators with axes filtered by an excluder or between any such generators must have a common `alternate` setting. An exception is made for the outermost generator as it is not repeated.

CHAPTER 9

Dimension

class scanpointgenerator.**Dimension**(*generator*)

An unrolled set of generators joined by excluders. Represents a single dimension within a scan.

axes = None

list(int) – Unrolled axes within the dimension

size = None

int – Size of the dimension

upper = None

list(float) – Upper bound for the dimension

lower = None

list(float) – Lower bound for the dimension

get_positions(*axis*)

Retrieve the positions for a given axis within the dimension.

Parameters **axis** (*str*) – axis to get positions for

Returns Array of positions

Return type Positions (np.array)

CHAPTER 10

Excluders

Excluders are used to filter points in a generator based on a pair of coordinates and some attribute of the point, for example its position or duration.

CHAPTER 11

ROIExcluders

ROIExcluders filter points that fall outside of a given a region of interest.

class scanpointgenerator.**ROIExcluder**(*rois*, *axes*)

A class to exclude points outside of regions of interest.

Parameters

- **rois** (*list(ROI)*) – List of regions of interest
- **axes** (*list(str)*) – Names of axes to exclude points from

create_mask(**point_arrays*)

Create a boolean array specifying the points to exclude.

The resulting mask is created from the union of all ROIs.

Parameters ***point_arrays** (*numpy.array(float)*) – Array of points for each axis

Returns Array of points to exclude

Return type np.array(int8)

to_dict()

Construct dictionary from attributes.

classmethod from_dict(*d*)

Create a ROIExcluder from a serialised dictionary.

Parameters **d** (*dict*) – Dictionary of attributes

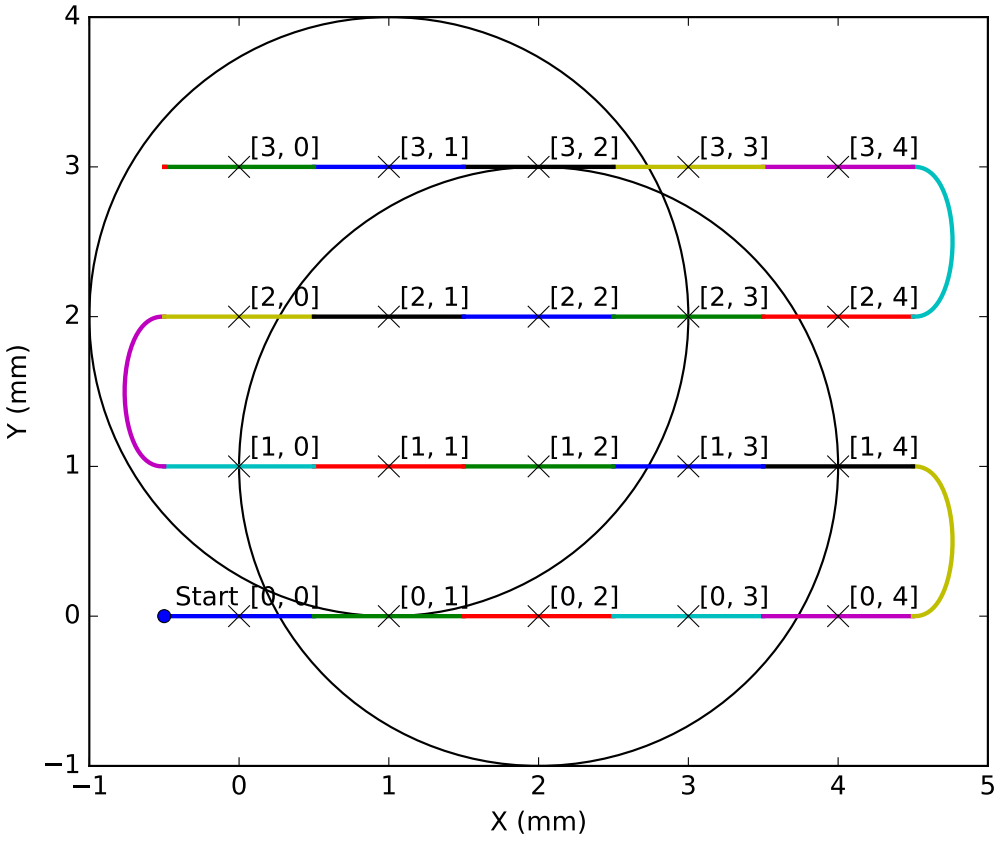
Returns New instance of ROIExcluder

Return type *ROIExcluder*

11.1 CircularROI Example

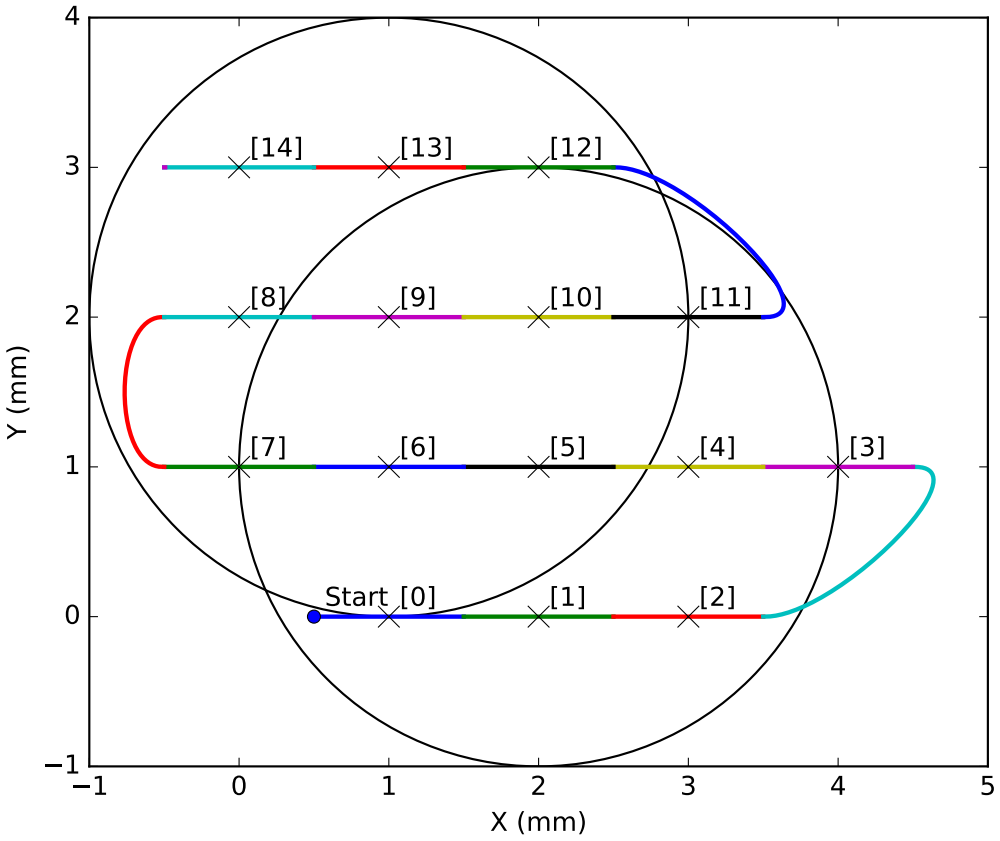
Here we use CircularROIs to filter the points of a snake scan

```
from scanpointgenerator import LineGenerator, CompoundGenerator, \
ROIExcluder, CircularROI
from scanpointgenerator.plotgenerator import plot_generator

x = LineGenerator("x", "mm", 0.0, 4.0, 5, alternate=True)
y = LineGenerator("y", "mm", 0.0, 3.0, 4)
circles = ROIExcluder([CircularROI([1.0, 2.0], 2.0),
                       CircularROI([2.0, 1.0], 2.0)], ["x", "y"])
gen = CompoundGenerator([y, x], [], [])
plot_generator(gen, circles)
```



And with the excluder applied

```
from scanpointgenerator import LineGenerator, CompoundGenerator, \
ROIExcluder, CircularROI
from scanpointgenerator.plotgenerator import plot_generator

x = LineGenerator("x", "mm", 0.0, 4.0, 5, alternate=True)
y = LineGenerator("y", "mm", 0.0, 3.0, 4)
circles = ROIExcluder([CircularROI([1.0, 2.0], 2.0),
                       CircularROI([2.0, 1.0], 2.0)], ["x", "y"])
gen = CompoundGenerator([y, x], [circles], [])
plot_generator(gen, circles)
```


[14]
[13]
[12]
[8]
[9]
[10]
[11]
[7]
[6]
[5]
[4]
[3]
Start
[0]
[1]
[2]
Y (mm)
X (mm)
4
3
2
1
0
−1
−1
0
1
2
3
4
5

CHAPTER 12

Mutators

Mutators are used for post processing points after they have been generated and filtered by any regions of interest.

class scanpointgenerator.**Mutator**
Abstract class to apply a mutation to the points of an ND ScanPointGenerator

mutate(*point*, *index*)
Abstract method to take a point, apply a mutation and then return the new point

Parameters

- **Point** – point to mutate
- **Index** – one-dimensional linear index of point

Returns Mutated point

Return type *Point*

to_dict()
Abstract method to convert object attributes into a dictionary

classmethod from_dict(*d*)
Abstract method to create a Mutator instance from a serialised dictionary

Parameters **d** (*dict*) – Dictionary of attributes

Returns New Mutator instance

Return type *Mutator*

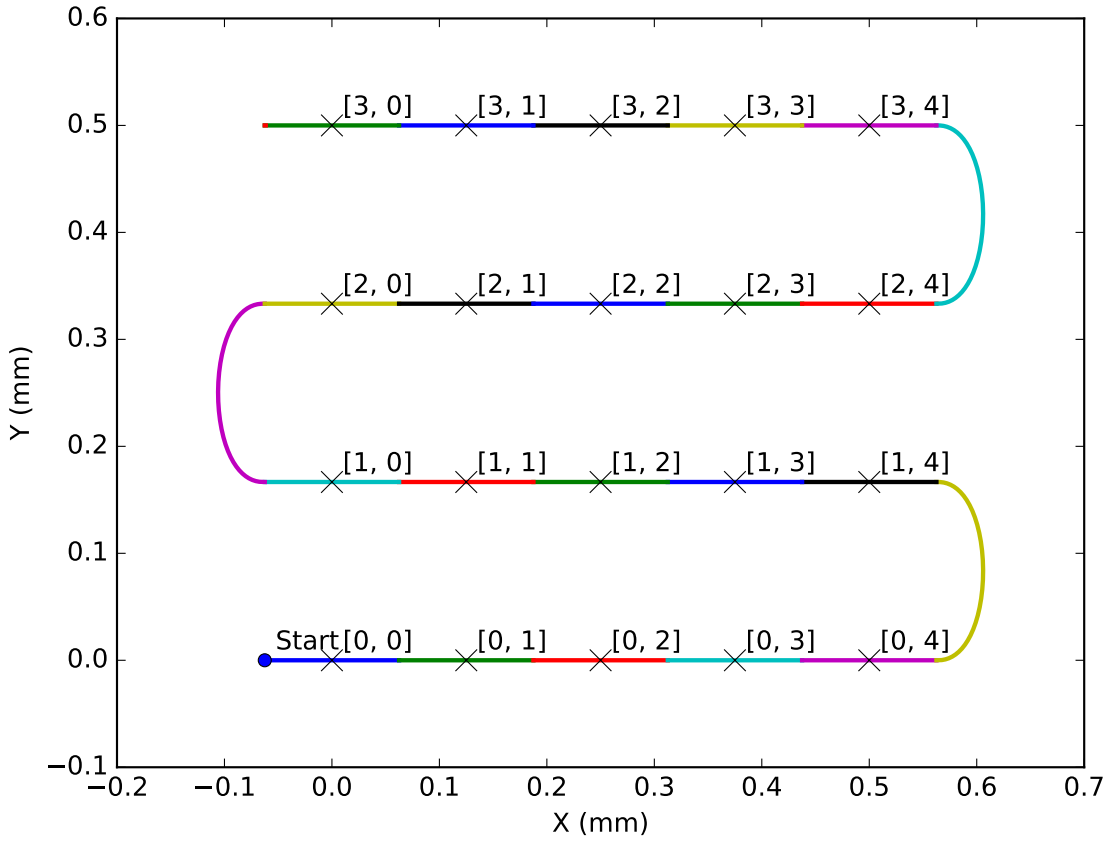
classmethod register_subclass(*mutator_type*)
Register a subclass so from_dict() works

Parameters **mutator_type** (Mutator) – Subclass to register

12.1 RandomOffsetMutator

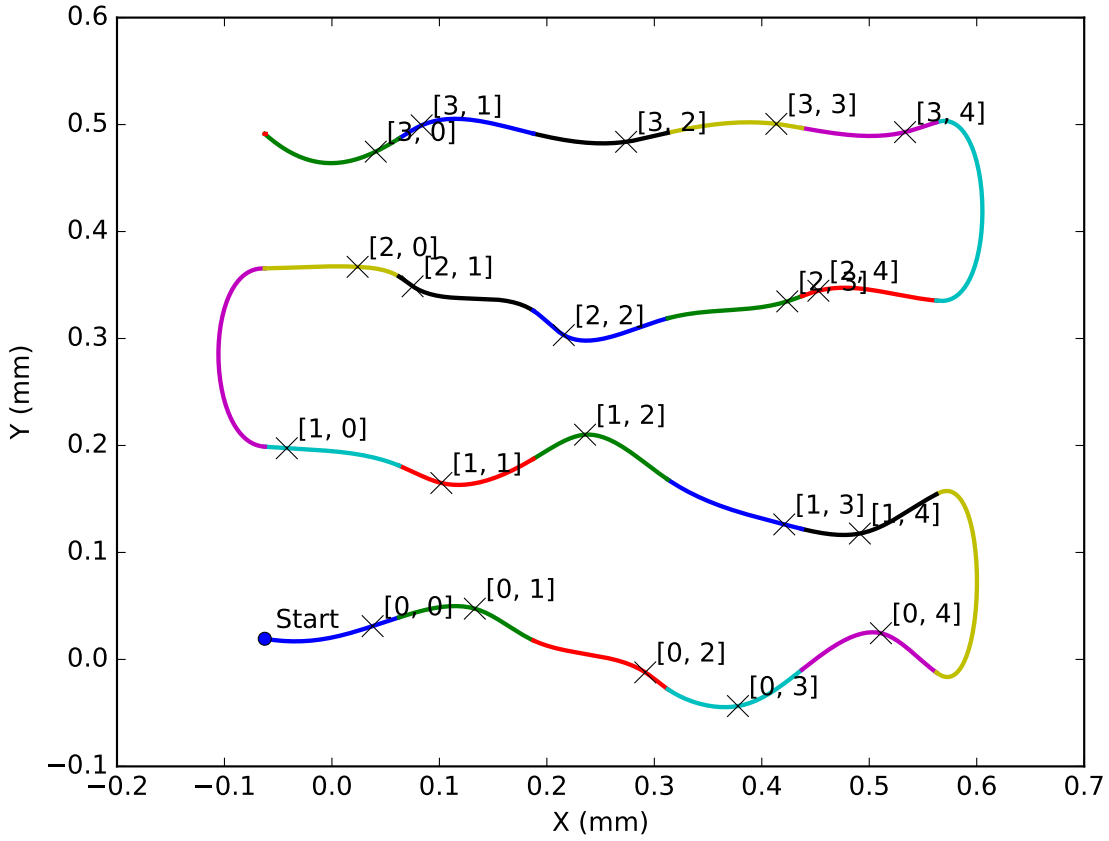
This is used to apply a random offset to each point in an iterator. Here we apply it to a snake scan

```
from scanpointgenerator import LineGenerator, CompoundGenerator
from scanpointgenerator.plotgenerator import plot_generator

xs = LineGenerator("x", "mm", 0.0, 0.5, 5, alternate=True)
ys = LineGenerator("y", "mm", 0.0, 0.5, 4)
gen = CompoundGenerator([ys, xs], [], [])
plot_generator(gen)
```



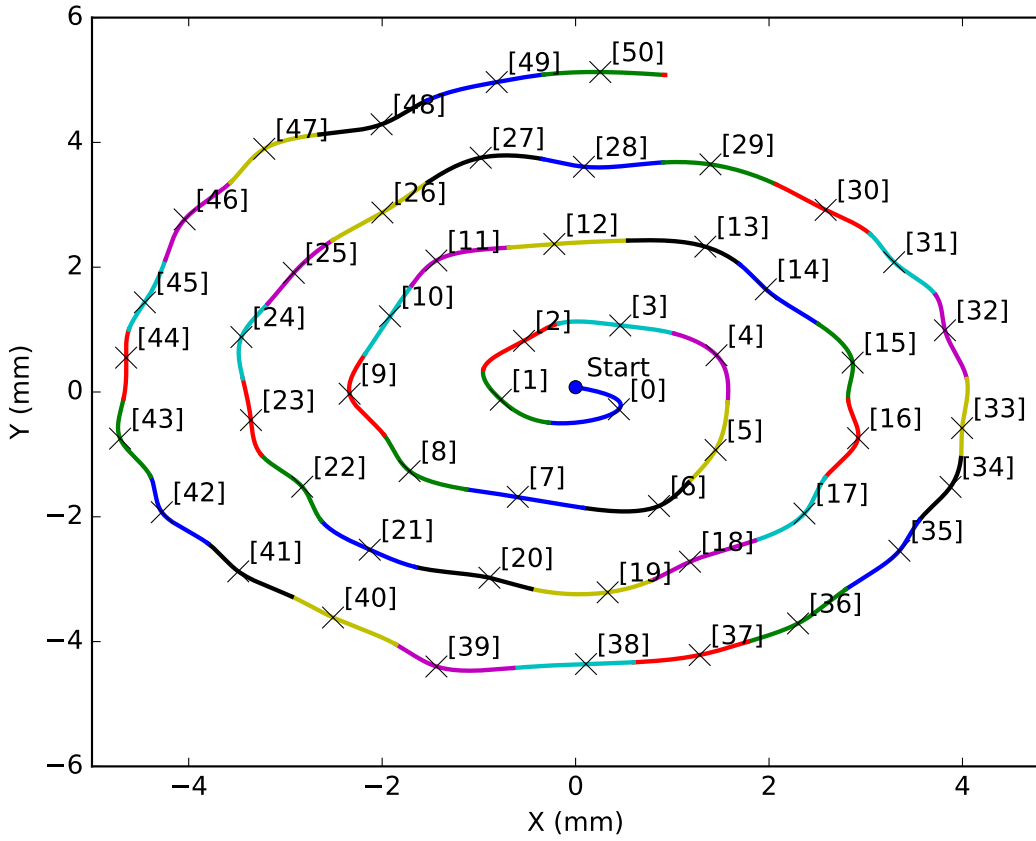
And with the random offset

```
from scanpointgenerator import LineGenerator, CompoundGenerator, RandomOffsetMutator
from scanpointgenerator.plotgenerator import plot_generator

xs = LineGenerator("x", "mm", 0.0, 0.5, 5, alternate=True)
ys = LineGenerator("y", "mm", 0.0, 0.5, 4)
random_offset = RandomOffsetMutator(seed=12345, axes = ["x", "y"], max_
↪offset=dict(x=0.05, y=0.05))
gen = CompoundGenerator([ys, xs], [], [random_offset])
plot_generator(gen)
```



Example with a spiral

```
from scanpointgenerator import SpiralGenerator, CompoundGenerator, RandomOffsetMutator
from scanpointgenerator.plotgenerator import plot_generator

spiral = SpiralGenerator(["x", "y"], ["mm", "mm"], [0., 0.], 5.0, 1.25)
random_offset = RandomOffsetMutator(seed=12345, axes = ["x", "y"], max_
↪offset=dict(x=0.2, y=0.2))
gen = CompoundGenerator([spiral], [], [random_offset])
plot_generator(gen)
```


Start
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
[28]
[29]
[30]
[31]
[32]
[33]
[34]
[35]
[36]
[37]
[38]
[39]
[40]
[41]
[42]
[43]
[44]
[45]
[46]
[47]
[48]
[49]
[50]
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
Y (mm)
X (mm)

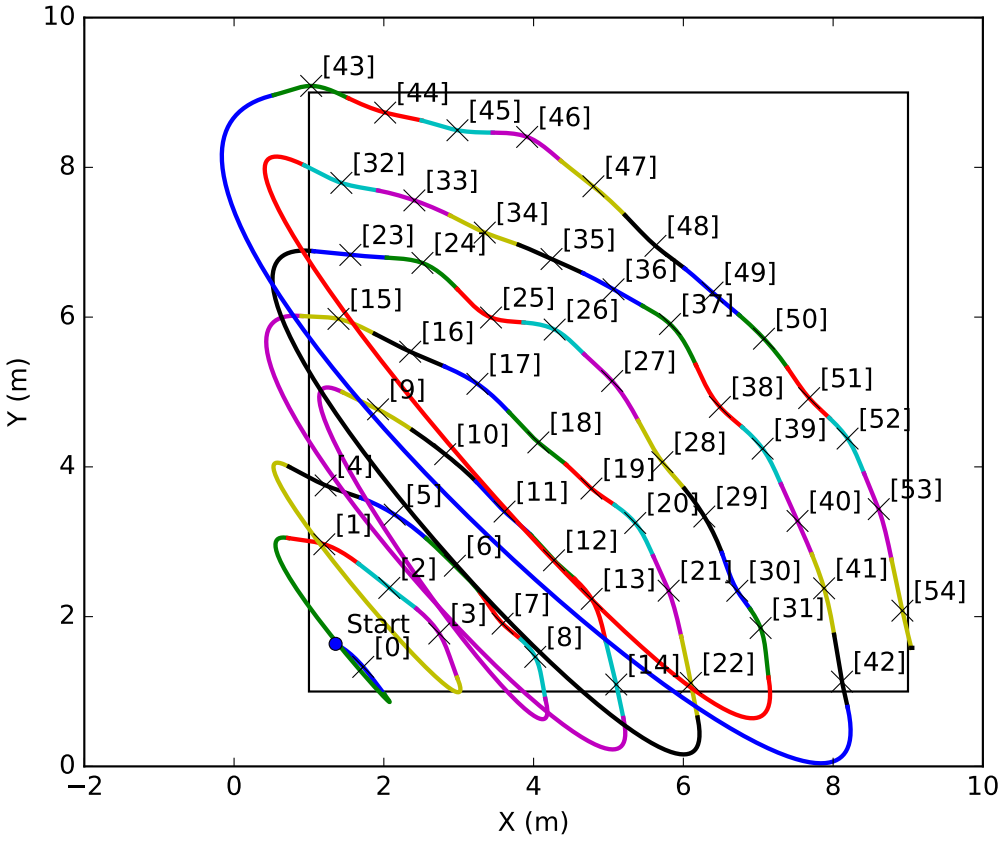
CHAPTER 13

Creating a Generator

The idea of CompoundGenerator is that you can combine generators, excluders and mutators arbitrarily. The following will show some more extensive examples to show the capabilities of scanpointgenerator. Remember to account for the restrictions specified in *Restrictions*.

A spiral scan with an offset rectangular roi overlay and randomly offset points in the y direction

```
from scanpointgenerator import LineGenerator, SpiralGenerator, \
CompoundGenerator, ROIExcluder, RandomOffsetMutator, RectangularROI
from scanpointgenerator.plotgenerator import plot_generator

spiral = SpiralGenerator(["x", "y"], "mm", [0.0, 0.0], 10.0,
                         alternate=True)
rectangle = ROIExcluder([RectangularROI([1.0, 1.0], 8.0, 8.0)], ["x", "y"])
mutator = RandomOffsetMutator(2, ["x", "y"], dict(x=0.0, y=0.25))
gen = CompoundGenerator([spiral], [rectangle], [mutator])

plot_generator(gen, rectangle)
```



A spiral scan at each point of a line scan with alternating direction

```
from scanpointgenerator import LineGenerator, SpiralGenerator, \
CompoundGenerator

line = LineGenerator("z", "mm", 0.0, 20.0, 3)
spiral = SpiralGenerator(["x", "y"], "mm", [0.0, 0.0], 1.2,
                         alternate=True)
gen = CompoundGenerator([line, spiral], [], [])
gen.prepare()

for point in gen.iterator():
    for axis, value in point.positions.items():
        point.positions[axis] = round(value, 3)
    print(point.positions)
```

```
{'y': -0.321, 'x': 0.237, 'z': 0.0}
{'y': -0.25, 'x': -0.644, 'z': 0.0}
{'y': 0.695, 'x': -0.56, 'z': 0.0}
{'y': 0.992, 'x': 0.361, 'z': 0.0}
{'y': 0.992, 'x': 0.361, 'z': 10.0}
{'y': 0.695, 'x': -0.56, 'z': 10.0}
{'y': -0.25, 'x': -0.644, 'z': 10.0}
{'y': -0.321, 'x': 0.237, 'z': 10.0}
{'y': -0.321, 'x': 0.237, 'z': 20.0}
```

(continues on next page)

(continued from previous page)

```
{'y': -0.25, 'x': -0.644, 'z': 20.0}
{'y': 0.695, 'x': -0.56, 'z': 20.0}
{'y': 0.992, 'x': 0.361, 'z': 20.0}
```

Three nested line scans with an excluder operating on the two innermost axes

```
from scanpointgenerator import LineGenerator, CompoundGenerator, \
ROIExcluder, CircularROI

line1 = LineGenerator("x", "mm", 0.0, 2.0, 3)
line2 = LineGenerator("y", "mm", 0.0, 1.0, 2)
line3 = LineGenerator("z", "mm", 0.0, 1.0, 2)
circle = ROIExcluder([CircularROI([1.0, 1.0], 1.0)], ["x", "y"])
gen = CompoundGenerator([line3, line2, line1], [circle], [])
gen.prepare()

for point in gen.iterator():
    print(point.positions)
```

```
{'y': 0.0, 'x': 1.0, 'z': 0.0}
{'y': 0.0, 'x': 1.0, 'z': 1.0}
{'y': 1.0, 'x': 0.0, 'z': 0.0}
{'y': 1.0, 'x': 1.0, 'z': 0.0}
{'y': 1.0, 'x': 2.0, 'z': 0.0}
{'y': 1.0, 'x': 0.0, 'z': 1.0}
{'y': 1.0, 'x': 1.0, 'z': 1.0}
{'y': 1.0, 'x': 2.0, 'z': 1.0}
```

CHAPTER 14

Serialisation

These generators are designed to be serialised and sent over json. The model for the CompoundGenerator is as follows:

```
{
    typeid: "scanpointgenerator:generator/CompoundGenerator:1.0",
    generators: [
        {
            typeid: "scanpointgenerator:generator/LineGenerator:1.0"
            axes: "y"
            units: "mm"
            start: 0.0
            stop: 1.0
            size: 5
            alternate = False
        },
        {
            typeid: "scanpointgenerator:generator/LineGenerator:1.0"
            axes: "x"
            units: "mm"
            start: 0.0
            stop: 5.0
            size: 5
            alternate = True
        }
    ],
    excluders: [
        {
            roi: {
                typeid: "scanpointgenerator:roi/CircularROI:1.0"
                centre: [0.0, 0.0]
                radius: 0.5
            }
            scannables: ["x", "y"]
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    mutators: [
        {
            typeid: "scanpointgenerator:mutator/RandomOffsetMutator:1.0"
            seed: 10
            axes: ["x", "y"]
            max_offset: {
                    x: 0.1
                    y: 0.2
            }
        }
    duration: -1.0
    ]
}
```

The models for each base generator are:

LineGenerator (axes, start and stop can be N-dimensional to create and ND scan):

```
{
    typeid: "scanpointgenerator:generator/LineGenerator:1.0"
    axes: "x" or ["x", "y"]
    units: "mm" or ["mm", "mm"]
    start: 0.0 or [0.0, 0.0]
    size: 5
    alternate = True
}
```

LissajousGenerator:

```
{
    typeid: "scanpointgenerator:generator/LissajousGenerator:1.0"
    axes: ["x", "y"]
    units: ["mm", "mm"]
    centre: [0.0, 0.0]
    span: [10.0, 10.0]
    lobes: 20
    size: 1000
    alternate = False
}
```

SpiralGenerator:

```
{
    typeid: "scanpointgenerator:generator/SpiralGenerator:1.0"
    axes: ["x", "y"]
    units: ["mm", "mm"]
    centre: [0.0, 0.0]
    radius: 5.0
    scale: 2.0
    alternate = True
}
```

ArrayGenerator:

```
{
    typeid: "scanpointgenerator:generator/ArrayGenerator:1.0"
    axis: "x"
```

(continues on next page)

(continued from previous page)

```
    units: "mm"
    points: [0., 1., 1.5, 2.]
    alternate = True
}
```

And for the mutators:

RandomOffsetMutator:

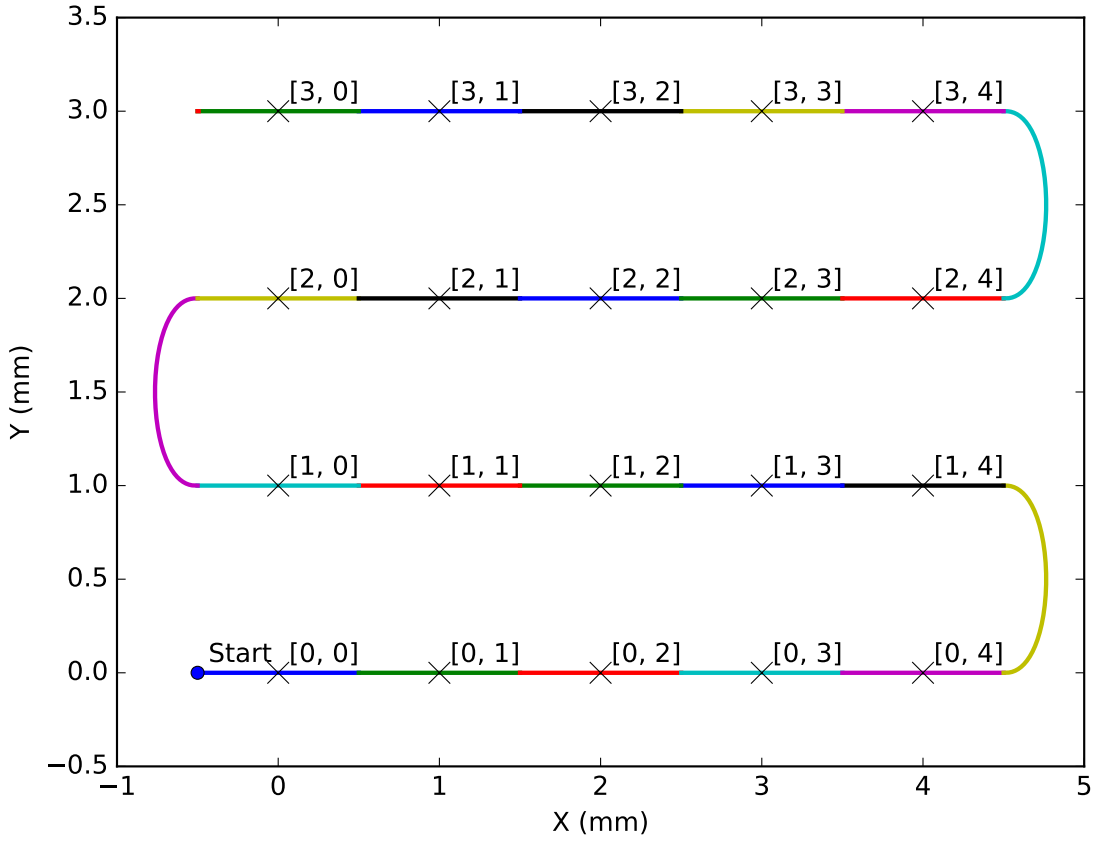
```
{
    typeid: "scanpointgenerator:mutator/RandomOffsetMutator:1.0"
    seed: 10
    axes: ["x", "y"]
    max_offset: {
        x: 0.1
        y: 0.2
    }
}
```

And the excluders:

To be added...

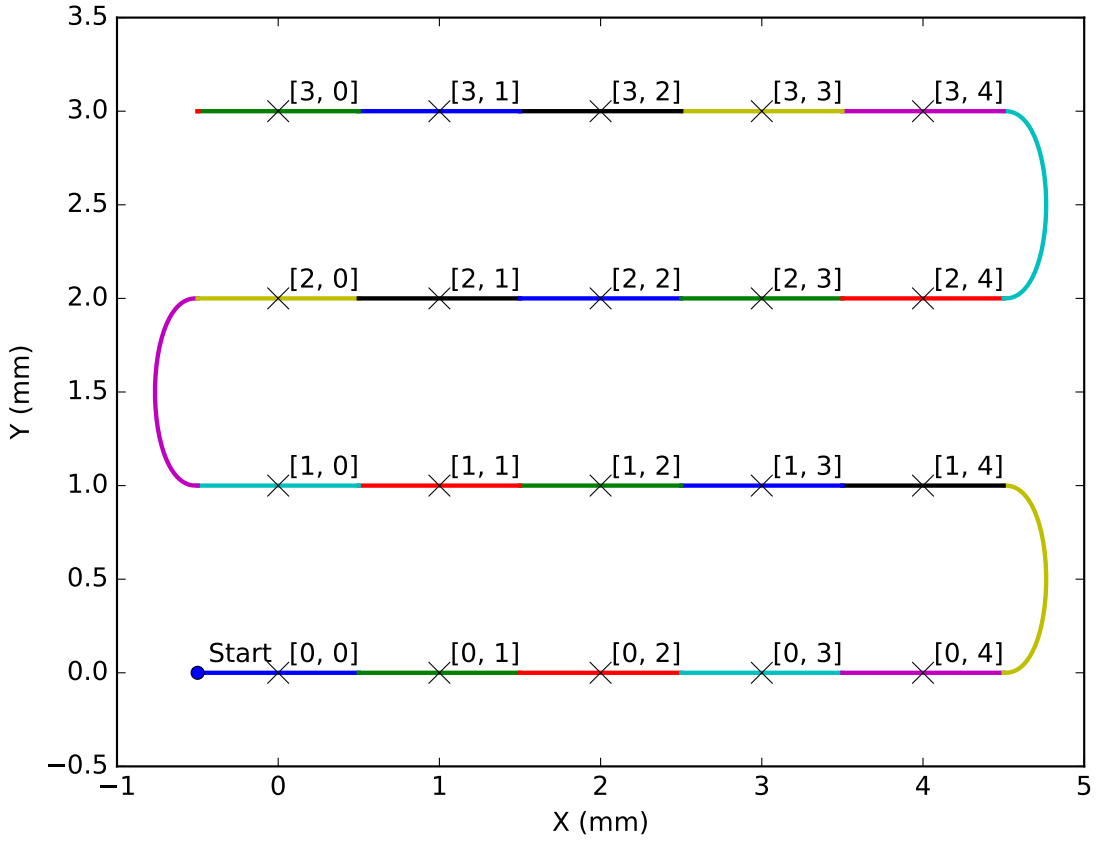
As an example of serialising, here is a simple snake scan.

```
from scanpointgenerator import LineGenerator, CompoundGenerator
from scanpointgenerator.plotgenerator import plot_generator

x = LineGenerator("x", "mm", 0.0, 4.0, 5, alternate=True)
y = LineGenerator("y", "mm", 0.0, 3.0, 4)
gen = CompoundGenerator([y, x], [], [])

plot_generator(gen)
```



It is the same after being serialised and deserialised.

```
from scanpointgenerator import LineGenerator, CompoundGenerator
from scanpointgenerator.plotgenerator import plot_generator

x = LineGenerator("x", "mm", 0.0, 4.0, 5, alternate=True)
y = LineGenerator("y", "mm", 0.0, 3.0, 4)
gen = CompoundGenerator([y, x], [], [])

gen_dict = gen.to_dict()
new_gen = CompoundGenerator.from_dict(gen_dict)

plot_generator(new_gen)
```


[3, 0]
[3, 1]
[3, 2]
[3, 3]
[3, 4]
[2, 0]
[2, 1]
[2, 2]
[2, 3]
[2, 4]
[1, 0]
[1, 1]
[1, 2]
[1, 3]
[1, 4]
Start
[0, 0]
[0, 1]
[0, 2]
[0, 3]
[0, 4]
Y (mm)
X (mm)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1
0
1
2
3
4
5

CHAPTER 15

Writing new scan point generators

Let's walk through the simplest generator, *LineGenerator*, and see how it is written.

```
###
# Copyright (c) 2016, 2017 Diamond Light Source Ltd.
```

We import the baseclass *Generator* and the compatibility wrappers around the Python `range()` function and the `numpy` module

```
#    Charles Mita - initial API and implementation and/or initial documentation
#
###
```

Our new subclass includes a docstring giving a short explanation of what it does and registers itself as a subclass of Generator for deserialization purposes.

```
    def __init__(self, axes, units, start, stop, size, alternate=False):
        """
        Args:
            axes (str/list(str)): The scannable axes E.g. "x" or ["x", "y"]
            units (str/list(str)): The scannable units. E.g. "mm" or ["mm", "mm"]
            start (float/list(float)): The first position to be generated.
                e.g. 1.0 or [1.0, 2.0]
            stop (float or list(float)): The final position to be generated.
                e.g. 5.0 or [5.0, 10.0]
            size (int): The number of points to generate. E.g. 5
            alternate(bool): Specifier to reverse direction if
                generator is nested
        """

        self.axes = to_list(axes)
        self.start = to_list(start)
        self.stop = to_list(stop)
        self.alternate = alternate
        self.units = {d:u for (d, u) in zip(self.axes, to_list(units))}
```

(continues on next page)

(continued from previous page)

```
        if len(self.axes) != len(set(self.axes)):
            raise ValueError("Axis names cannot be duplicated; given %s" %
                             axes)

        if len(self.axes) != len(self.start) or \
           len(self.axes) != len(self.stop):
            raise ValueError(
                "Dimensions of axes, start and stop do not match")

        self.size = size

        self.step = []
        if self.size < 2:
            self.step = [0]*len(self.start)
        else:
            for axis in range_(len(self.start)):
                self.step.append(
                    (self.stop[axis] - self.start[axis])/(self.size - 1))
```

The initializer performs some basic validation on the parameters and stores them. The units get stored as a dictionary attribute of axis->unit:

```
    def prepare_arrays(self, index_array):
        arrays = {}
        for axis, start, stop in zip(self.axes, self.start, self.stop):
            d = stop - start
            step = float(d)
            # if self.size == 1 then single point case
            if self.size > 1:
                step /= (self.size - 1)
            f = lambda t: (t * step) + start
            arrays[axis] = f(index_array)
        return arrays
```

This is used by CompoundGenerator to create the points for this generator. This method should create, for each axis the generator defines, an array of positions by transforming the input index array. The index array will be the numpy array [0, 1, 2, . . . , n-1, n] for normal positions, and [-0.5, 0.5, 1.5, . . . , n-0.5, n+0.5] when used to calculate boundary positions.

The arrays are returned as a dictionary of {axis_name : numpy float array}

CHAPTER 16

Contributing

Contributions and issues are most welcome! All issues and pull requests are handled through github on the dls_controls repository. Also, please check for any existing issues before filing a new one. If you have a great idea but it involves big changes, please file a ticket before making a pull request! We want to make sure you don't spend your time coding something that might not fit the scope of the project.

16.1 Running the tests

To get the source source code and run the unit tests, run:

```
$ git clone git://github.com/dls_controls/scanpointgenerator.git
$ cd scanpointgenerator
$ virtualenv env
$ . env/bin/activate
$ pip install nose
$ python setup.py install
$ python setup.py nosetests
```

While 100% code coverage does not make a library bug-free, it significantly reduces the number of easily caught bugs! Please make sure coverage is at 100% before submitting a pull request!

16.2 Code Quality

Landscape.io will test code quality when you create a pull request. Please follow PEP8.

16.3 Code Styling

Please arrange imports with the following style

```
# Standard library imports
import os

# Third party package imports
from mock import patch

# Local package imports
from scanpointgenerator.version import __version__
```

Please follow Google's python style guide wherever possible.

16.4 Building the docs

When in the project directory:

```
$ pip install -r requirements/docs.txt
$ python setup.py build_sphinx
$ open docs/html/index.html
```

16.5 Release Checklist

Before a new release, please go through the following checklist:

- Bump version in scanpointgenerator/version.py
- Add a release note and diff URL in CHANGELOG.rst
- Git tag the version
- Upload to pypi:

  ```
  make publish
  ```

CHAPTER 17

Change Log

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

17.1 Unreleased

Added:

- Nothing yet

17.2 2-1-1 - 2018-04-30

Added:

- Tags now cause Travis to deploy to PyPi
- Added StaticPointGenerator
- Allow ROI to span multiple Dimensions
- Add continuous property to CompoundGenerator

Fixed:

- Fixed plotgenerator to interpolate turnarounds properly

17.3 2-1-0 - 2017-04-18

Fixed:

- Fixed incorrect comparison PolygonalROI mask_points that resulted in an incorrect mask
- Point bounds are now giben for a grid scan in a rectangular region

Changed:

- Use numpy import when running in Jython instead of "numjy"

17.4 2-0-0 - 2017-03-17

Added:

- Adds dependency on numpy
- Added Dimension class, providing points along a dataset dimension
- Add dimensions attribute to CompoundGenerator
- Add shape attribute to CompoundGenerator
- Jython builds using a numpy emulator are tested
- Add ROIExcluder, replacing previous use of Excluder (now a generic base class)

Changed:

- Rewrite Generator mechanisms to use vectorised operations for point calculation
- Generators now only usable through CompoundGenerator
- CompoundGenerator requires call to prepare before use
- CompoundGenerator now takes a duration argument, replacing FixedDurationMutator (removed)
- Rename name/names to axes in Generators
- Rename scannables to axes in Excluders
- Generators take an array for units with the same size as axes
- Rename num_points to size and num_lobes to lobes in LissajousGenerator
- PolygonalROI takes separate x,y arrays for its vertices
- Bounds are only applied to the innermost axis/axes
- Remove index_names and index_dims from Generators
- License changed to Eclipse Public License v1.0

17.5 1-6-1 - 2016-10-27

Fixed:

- Add workaround for GDA not working with threading

17.6 1-6 - 2016-10-18

Fixed:

- CompoundGenerator to set the right number of points if excluders are used

Changed:

- Refactored internal structure of modules

17.7 1-5 - 2016-10-07

Added:

- Add full ROI set and FixedDurationMutator

17.8 1-4 - 2016-09-22

Added:

- Caching of points to CompoundGenerator

17.9 1-3-1 - 2016-09-13

Added:

- Serialisation for ROIs
- Change type to typeid to match with Malcolm

17.10 1-3 - 2016-08-31

Added:

- Remove OrderedDict entirely for 2.5 back-compatibility

Changed:

- type is now typeid to make it compatible with malcolm

17.11 1-2-1 - 2016-08-17

Fixed:

- Refactor RandomOffsetMutator to be consistent in Jython and Python without OrderedDict in Point

17.12 1-2 - 2016-08-17

Added:

- Remove OrderedDict from Point and speed up LineGenerator

17.13 1-1 - 2016-08-16

Added:

- Small tweaks for GDA and script to push changes to daq-eclipse on release

17.14 1-0 - 2016-07-18

Added:

- Initial requirements for GDA and Malcolm

17.15 0-5 - 2016-06-20

Added:

- Additions to work with GDA and Malcolm

17.16 0-4 - 2016-04-15

Added:

- MANIFEST.in file to allow install in travis builds

17.17 0-3 - 2016-03-03

Added:

- Documentation on writing new generators

17.18 0-2 - 2016-02-29

Added:

- Documentation
- Indexes to plots

17.19 0-1 - 2016-02-26

Added:

- Initial structure with Line and Nested generators

Python Module Index

S

Index

S

T

U