# Taipan Router Documentation

**Carlos Bacigalupo**

**Aug 07, 2018**

# CONTENTS

# CHAPTER ONE

## TAIPAN ROUTER MANUAL

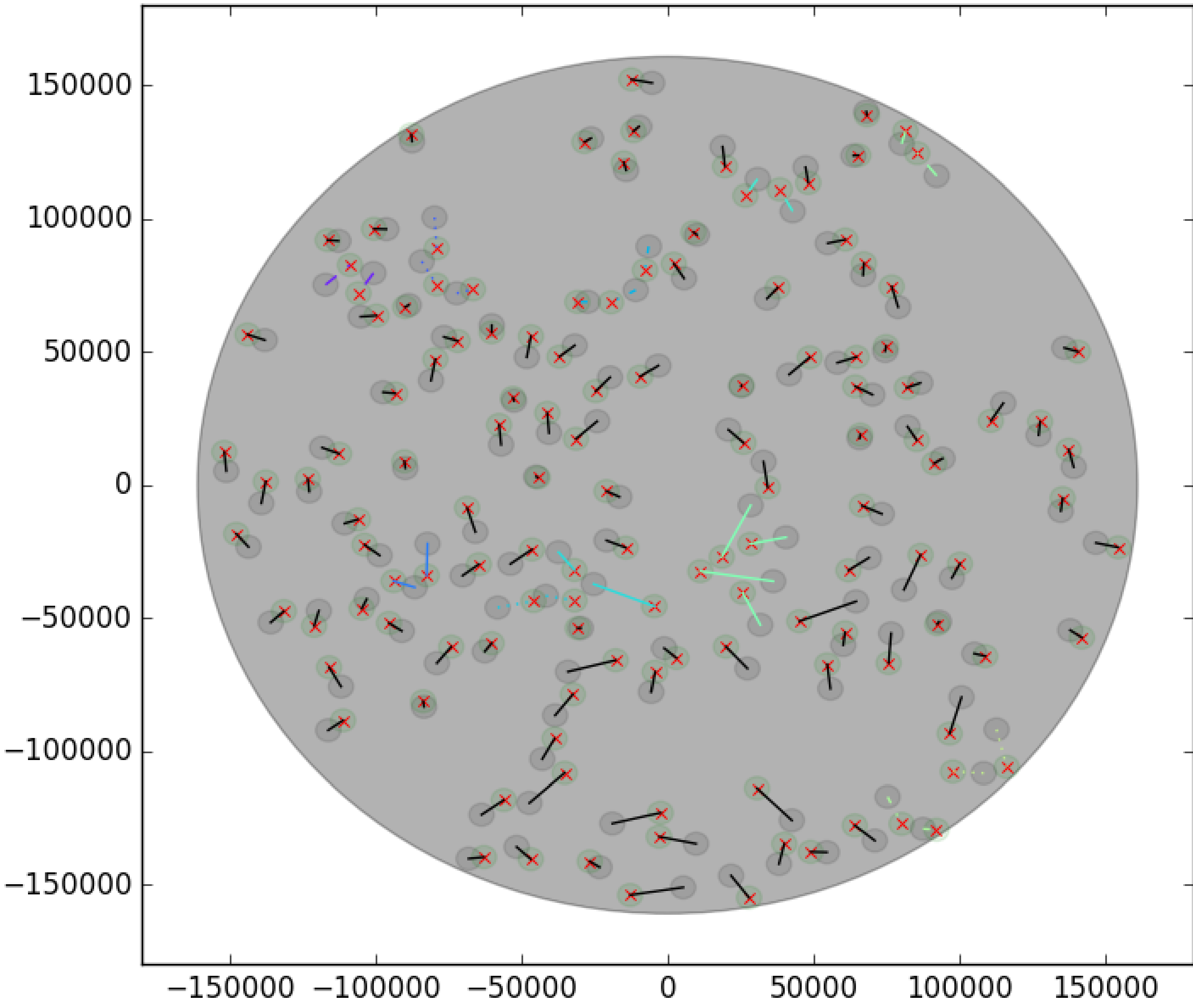
The TAIPAN router calculates the paths that the starbugs need to take to be positioned in a given arrangement.

The starting coordinates are the individual park positions as provided by the locationProperties.json (S5). This file also contains the information on bug availability allowing the identification of missing bugs.

The target coordinates are specified in the XY tiles (S2). These tiles are created from the Tiles (S1) that are generated by the tiler code.

The step to convert S1 -> S2 is done by an external program creatively named S12S2. This program uses the telescope model to convert RA/Dec coordinates to X/Y positions in the focal plane.

The output of the code is a Routed Tile (S3) with the necessary steps to move the bugs into position without colliding.


150000
100000
50000
0
−50000
−100000
−150000
−150000
−100000
−50000
0
50000
100000
150000

# CHAPTER TWO

# TAIPAN ROUTER REFERENCE MANUAL

**class** taipanPyRouter.**CGroupSolver**(*CGroup*, *tickArray*, *bugStatus*)

Class to hold the crossing group solving funcitons.

**calcCoeffs**(*A*, *B*)

Calculates the collision coefficients.

- Given 2 ESegments, it returns the coefficients of the collision points.
- If the ESegments are parallel, it returns NaNs

**Parameters**

- **A** (`ESegment`) – First segment to compare.
- **B** (`ESegment`) – Second segment to compare.

**Returns** Pair of coefficients

**Return type** Tuple

**constructCMatrix**()

Creates a distance matrix for all segments in the crossing group

**constructESegments**(*XYs*)

Constructs a list of all ESegments in the CGroup.

**Parameters XYs** (*np.ndarray*) – List of coordinates of the end points of each ESegment

**Returns** List of ESegments

**Return type** list

**findMovingSequence**()

Analyses the cost matrix to find the sequence of motion that doesn't crash.

**findMovingSequenceBF**()

**class** taipanPyRouter.**ESegment**(*XYs*)

Bases: `shapely.geometry.linestring.LineString`

Class to extend the existing Linestring class into E(xtended)Segments.

ESegments provide extra elements that apply to route solving

**almost_equals**(*other*, *decimal=6*)

Returns True if geometries are equal at all coordinates to a specified decimal place

Refers to approximate coordinate equality, which requires coordinates be approximately equal and in the same order for all components of a geometry.

**area**
Unitless area of the geometry (float)

**array_interface**()
Provide the Numpy array protocol.

**array_interface_base**

**boundary**
Returns a lower dimension geometry that bounds the object

The boundary of a polygon is a line, the boundary of a line is a collection of points. The boundary of a point is an empty (null) collection.

**bounds**
Returns minimum bounding region (minx, miny, maxx, maxy)

**buffer**(*distance*, *resolution=16*, *quadsegs=None*, *cap_style=1*, *join_style=1*, *mitre_limit=5.0*)
Returns a geometry with an envelope at a distance from the object's envelope

A negative distance has a "shrink" effect. A zero distance may be used to "tidy" a polygon. The resolution of the buffer around each vertex of the object increases by increasing the resolution keyword parameter or second positional parameter. Note: the use of a *quadsegs* parameter is deprecated and will be gone from the next major release.

The styles of caps are: CAP_STYLE.round (1), CAP_STYLE.flat (2), and CAP_STYLE.square (3).

The styles of joins between offset segments are: JOIN_STYLE.round (1), JOIN_STYLE.mitre (2), and JOIN_STYLE.bevel (3).

The mitre limit ratio is used for very sharp corners. The mitre ratio is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

### Example

```
>>> from shapely.wkt import loads
>>> g = loads('POINT (0.0 0.0)')
>>> g.buffer(1.0).area        # 16-gon approx of a unit radius circle
3.1365484905459389
>>> g.buffer(1.0, 128).area   # 128-gon approximation
3.1415138011443009
>>> g.buffer(1.0, 3).area     # triangle approximation
3.0
>>> list(g.buffer(1.0, cap_style='square').exterior.coords)
[(1.0, 1.0), (1.0, -1.0), (-1.0, -1.0), (-1.0, 1.0), (1.0, 1.0)]
>>> g.buffer(1.0, cap_style='square').area
4.0
```

**centroid**
Returns the geometric center of the object

**contains**(*other*)
Returns True if the geometry contains the other, else False

**convex_hull**
*Imagine an elastic band stretched around the geometry* – that's a convex hull, more or less

The convex hull of a three member multipoint, for example, is a triangular polygon.

**coords**
Access to geometry's coordinates (CoordinateSequence)

**covers** (*other*)
Returns True if the geometry covers the other, else False

**crosses** (*other*)
Returns True if the geometries cross, else False

**ctypes**

**difference** (*other*)
Returns the difference of the geometries

**disjoint** (*other*)
Returns True if geometries are disjoint, else False

**distance** (*other*)
Unitless distance to other geometry (float)

**empty** (*val=4460832448*)

**envelope**
A figure that envelopes the geometry

**equals** (*other*)
Returns True if geometries are equal, else False

Refers to point-set equality (or topological equality), and is equivalent to (self.within(other) & self.contains(other))

**equals_exact** (*other*, *tolerance*)
Returns True if geometries are equal to within a specified tolerance

Refers to coordinate equality, which requires coordinates to be equal and in the same order for all components of a geometry

**geom_type**
Name of the geometry's type, such as 'Point'

**geometryType** ()

**has_z**
True if the geometry's coordinate sequence(s) have z values (are 3-dimensional)

**hausdorff_distance** (*other*)
Unitless hausdorff distance to other geometry (float)

**impl = <GEOSImpl object: GEOS C API version (1, 8, 0)>**

**interpolate** (***kwargs*)
Return a point at the specified distance along a linear geometry

If the normalized arg is True, the distance will be interpreted as a fraction of the geometry's length.

**intersection** (*other*)
Returns the intersection of the geometries

**intersects** (*other*)
Returns True if geometries intersect, else False

**is_closed**
True if the geometry is closed, else False

Applicable only to 1-D geometries.

**is_empty**
True if the set of points in this geometry is empty, else False

**is_ring**
True if the geometry is a closed ring, else False

**is_simple**
True if the geometry is simple, meaning that any self-intersections are only at boundary points, else False

**is_valid**
True if the geometry is valid (definition depends on sub-class), else False

**length**
Unitless length of the geometry (float)

**minimum_rotated_rectangle**
Returns the general minimum bounding rectangle of the geometry. Can possibly be rotated. If the convex hull of the object is a degenerate (line or point) this same degenerate is returned.

**overlaps**(*other*)
Returns True if geometries overlap, else False

**parallel_offset**(*distance*, *side='right'*, *resolution=16*, *join_style=1*, *mitre_limit=5.0*)
Returns a LineString or MultiLineString geometry at a distance from the object on its right or its left side.

The side parameter may be 'left' or 'right' (default is 'right'). The resolution of the buffer around each vertex of the object increases by increasing the resolution keyword parameter or third positional parameter. Vertices of right hand offset lines will be ordered in reverse.

The join style is for outside corners between line segments. Accepted values are JOIN_STYLE.round (1), JOIN_STYLE.mitre (2), and JOIN_STYLE.bevel (3).

The mitre ratio limit is used for very sharp corners. It is the ratio of the distance from the corner to the end of the mitred offset corner. When two line segments meet at a sharp angle, a miter join will extend far beyond the original geometry. To prevent unreasonable geometry, the mitre limit allows controlling the maximum length of the join corner. Corners with a ratio which exceed the limit will be beveled.

**pointFromCoeff**(*coeff*)
Calculates a projected point from a given coefficient.

The resulting point is in the position coeff*length, where length is the length of the ESegment.

**Parameters** **coeff** (`float`) – Distance to the requested point in ESegment lengths. Can be negative.

**Returns** Position of the point.

**Return type** Tuple

**project**(***kwargs*)
Returns the distance along this geometry to a point nearest the specified point

If the normalized arg is True, return the distance normalized to the length of the linear geometry.

**relate**(*other*)
Returns the DE-9IM intersection matrix for the two geometries (string)

**relate_pattern**(*other*, *pattern*)
Returns True if the DE-9IM string code for the relationship between the geometries satisfies the pattern, else False

**representative_point**(***kwargs*)
Returns a point guaranteed to be within the object, cheaply.

**simplify**(***kwargs*)
Returns a simplified geometry produced by the Douglas-Peucker algorithm

Coordinates of the simplified geometry will be no more than the tolerance distance from the original. Unless the topology preserving option is used, the algorithm may produce self-intersecting or otherwise invalid geometries.

**svg**(*scale_factor=1.0*, *stroke_color=None*)
Returns SVG polyline element for the LineString geometry.

**Parameters**

- **scale_factor** (*float*) – Multiplication factor for the SVG stroke-width. Default is 1.
- **stroke_color** (*str, optional*) – Hex string for stroke color. Default is to use "#66cc99" if geometry is valid, and "#ff3333" if invalid.

**symmetric_difference**(*other*)
Returns the symmetric difference of the geometries (Shapely geometry)

**to_wkb**()

**to_wkt**()

**touches**(*other*)
Returns True if geometries touch, else False

**type**

**union**(*other*)
Returns the union of the geometries (Shapely geometry)

**within**(*other*)
Returns True if geometry is within the other, else False

**wkb**
WKB representation of the geometry

**wkb_hex**
WKB hex representation of the geometry

**wkt**
WKT representation of the geometry

**xy**
Separate arrays of X and Y coordinate values

### Example

```
>>> x, y = LineString(((0, 0), (1, 1))).xy
>>> list(x)
[0.0, 1.0]
>>> list(y)
[0.0, 1.0]
```

taipanPyRouter.**checkForCollisions**(*tickArray*, *bugStatus*, *booPrint=True*)
Looks for collisions in the created tick array.

- Steps through the tickArray 1 tick at the time.
- Tries to re-create the crossing groups to look for crossings

**Parameters**

- **`tickArray`** (*np.ndarray*) – Array with the routing solution
- **`bugStatus`** (*np.ndarray*) – Status of the bugs

**Returns** True if collisions found.

**Return type** boolean

`taipanPyRouter.`**`checkValidGFPandCrash`**(*bugsTargetXY*, *bugTargetTypes*)

**Checks that targets are:**

- inside GFP
- far enough from eachother

Removes

**Parameters**

- **`bugsTargetXY`** – array with the target points
- **`bugsTargetStatus`** – array with the end points

`taipanPyRouter.`**`checkValidTargets`**(*parkPos*, *bugsTargetXY*, *bugStatus*, *bugRouting*)

**Checks that targets are:**

- inside patrol radius
- far from static bugs

Updates bugStatus accordingly

**Parameters**

- **`parkPos`** – array with the starting points
- **`bugsTargetXY`** – array with the end points

`taipanPyRouter.`**`checkValidTargetsPR`**(*parkPos*, *bugsTargetXY*, *bugStatus*, *bugRouting*)

**Checks that targets are:**

- inside patrol radius
- far from static bugs

Updates bugStatus accordingly

**Parameters**

- **`parkPos`** – array with the starting points
- **`bugsTargetXY`** – array with the end points

`taipanPyRouter.`**`consolidateCGroups`**(*crossingBugs*, *tickArray*)
Takes all pairs that collide and groups them assigning a unique ID.

**Parameters**

- **`crossingBugs`** – np.ndarray Collection of pairs of crossing bugs.
- **`tickArray`** – np.ndarray [lemoID-1, tick, coords]

`taipanPyRouter.`**`createWorkingFolder`**(*base='.'*)
Creates a folder to drop files using the next available name.

`taipanPyRouter.`**`doRoutes`**(*args*)

`taipanPyRouter.`**`findCrossingGroups`**(*tickArray*, *bugStatus*)
Identify crossing groups within the direct paths

**Parameters** **`tickArray`** – array with the starting and ending points

**Returns** np.ndarray Collection of pairs of crossing bugs.

**Return type** crossingBugs

`taipanPyRouter.`**`initialiseTickArray`**(*parkPos*, *bugsTargetXY*)
Creates a new tick array with 2 only ticks (direct path)

**Parameters**

- **`parkPos`** – array with the starting points
- **`bugsTargetXY`** – array with the end points

**Returns** 3D np.array [lemoID-1, tick, coords]

**Return type** tickArray

`taipanPyRouter.`**`loadParkPosJSON`**(*filename='locationProperties.json'*, *folder='.'*)
Reads the park position file

**Parameters**

- **`filename`** (`str`) – The name of the input file
- **`folder`** (`str`) – The location of the input file

**Returns** 2D - [[x],[y]] parked positions coordinates indexed by LemoId-1 bugStatus (np.ndarray) : 1D [status] bugStatus indexed by LemoId-1 bugTypes (np.ndarray) : 1D [types] bugTypes indexed by LemoId-1

**Return type** parkPos (np.ndarray)

**Note:** Currently hardcoded array size to 309 bugs This function should read from the database

`taipanPyRouter.`**`openS2JSONTile`**(*fileName*, *folder='./jsonTiles_s2'*)
Reads the JSON file containing the target information (S2)

**Parameters**

- **`filename`** (`str`) – The name of the input file
- **`folder`** (`str`) – The location of the input file

**Returns** Bugs requested position [[x,y]] indexed by LemoId-1

**Return type** bugsXY (np.ndarray)

**Note:**

- Currently hardcoded array size to 309,2

`taipanPyRouter.`**`optimiseAllocation`**(*parkPos*, *bugStatus*, *bugTypes*, *bugsTargetXY*, *bugTargetTypes*)
Minimise the cost matrix of distances between a set of parked positions and set of targets.

- It assigns the best target positions based on minimum combined distance

- Only the positions in parkPos that can be allocated have actual values, the rest of the values are NaNs.
- This process allows the code to segment the allocation by fibre type

**Parameters**

- **parkPos** (*np.ndarray*) – Park position for all starbugs
- **bugsTargetXY** (*np.ndarray*) – Target positions to be allocated

**Returns** Array of same shape of parkPos with the new allocations

**Return type** newTargetsAlloc (np.ndarray)

`taipanPyRouter.`**`shiftTickArray`**(*movSeq*, *tickArray*)

Given a sequence of bugIds, it creates the ticks to move bugs sequentially

- It reshapes tickArray as needed. [nBugs, len(movSeq)+1, 2]
- It cascades the motion of the corresponding bugs sequentially as per movSeq order
- It completes preceding ticks with initial tick pos for movSeq bugs
- It adds target position to all tick following motion

**Parameters**

- **movSeq** (*np.ndarray*) – List of bugIds to be moved in sequential order.
- **tickArray** (*np.ndarray*) – Tick array to be appended with sequential motion

**Returns** Shifted TickArray.

**Return type** np.ndarray

`taipanPyRouter.`**`writeOuputFile`**(*S2FileName*, *S3FileName*, *tickArray*)

Writes an RTile (S3) from a tickArray

**Parameters**

- **S2FileName** (*string*) – Input XYTile
- **S3FileName** (*string*) – Output RTile
- **tickArray** (*np.ndarray*) – routing ticks array

- genindex

# PYTHON MODULE INDEX

## t

# INDEX

## O



## P



## R



## S



## T



## U



## W



## X