tcnc Documentation

Release 0.2

Claude Zervas

Apr 23, 2018

Table of Contents

CHAPTER 1

Installing Tcnc

For now installation must be done by hand…

Example shell commands should work on MacOS/Linux. For Windows/Cygwin see *Inkscape user extension folder location*.

1. Download the latest version:

```
curl -L -O "https://github.com/utlco/tcnc/archive/master.zip"
```

2. Unzip/extract the downloaded archive file (tcnc-master.zip):

```
unzip tcnc-master.zip
```

3. Copy or move the Inkscape extension description (.inx) files in **tcnc-master/inkinx** to the Inkscape user extension folder:

```
cp tcnc-master/inkinx/\*.inx ~/.config/inkscape/extensions
```

4. Copy or move the entire **tcnc** folder from **tcnc-master** to the Inkscape user extension folder:

```
cp -R tcnc-master/tcnc ~/.config/inkscape/extensions
```

5. Restart Inkscape.

1.1 Inkscape user extension folder location

- MacOS, Linux:

  ~/.config/inkscape/extensions, where ~ is your home directory (i.e. /Users/YourUserName).

- Windows:

  C:\Users\YourUserName.Appdata\Roaming\inkscape\extensions

CHAPTER 2

Tcnc Extension User Documentation

Tcnc is designed as an Inkscape extension plugin but can be also invoked as a standalone program from the command line. It will read and process SVG files.

2.1 As an Inkscape Plugin

When preparing a document to use with Tcnc set the **default units** in **File->Document Properties…** to either **in** or **mm** and use the same unit to specify the document size. This is optional but strongly recommended since it helps to avoid unexpected unit conversion issues. Document units such as **pt** and **pc** should be avoided since this will lead to head scratching.

Select a path in Inkscape or select a layer if you want everything on that layer to be processed. From the **Extensions** menu select **UTLCo -> Tcnc..** and the **Tcnc** extension dialog will appear.

Tcnc only processes paths or basic shapes (circles, arcs, ellipses, boxes, and spirals). Text or other objects must be converted to paths.

2.1.1 The TCnc dialog

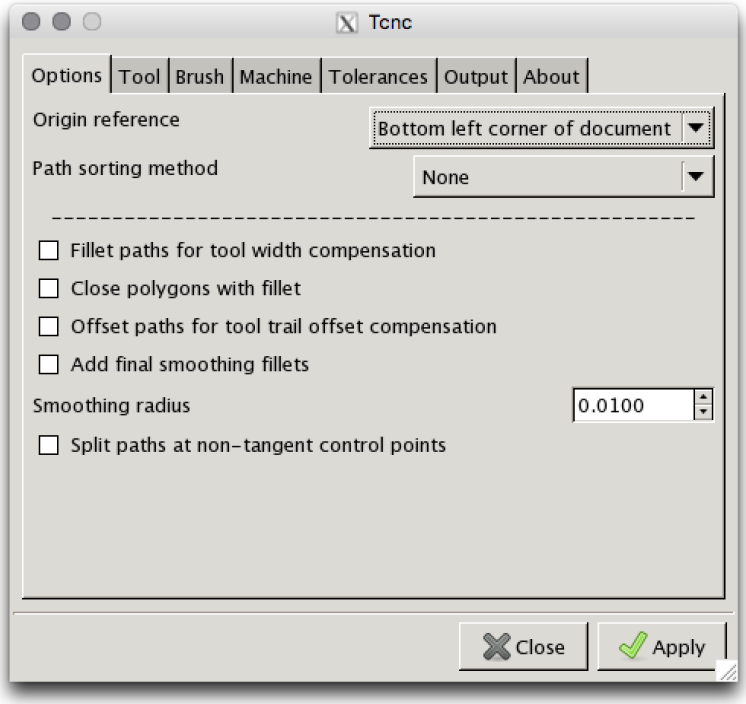
Options

Disable tangent rotation

Normally the A axis is automatically rotated during a feed so that the tool is always tangent to the direction of the XY movement. The A axis is assumed to rotate about the Z axis.

If this option is selected the tangent following behaviour will be disabled. The A axis will rotate to the initial A axis offset (if any) and stay at that angle.

Selecting this option will also disable the path fillet and offset options since they only make sense for a tangent tool of non-zero width and trail.



Path sorting method

- **Flip paths** Use original path order but flip (reverse) paths to shorten rapid moves.
- **Bottom to top, left to right** Sorts the paths from bottom to top and secondarily left to right.

Fillet paths for tool width compensation

For wide brush-like tools. This will modify the path to add fillet arcs at corners to compensate for the tool width. It mitigates rotational artifacts caused by the wider tools.

Close polygons with fillet

Adds a fillet between the end of the last segment of a path and the start of the first segment. Only applies if fillets are enabled.

Offset paths for tool trail offset compensation

Offsets the path segments to compensate for tool trail offset. Tools with a trail offset will follow the orginal path much more closely.

Add final smoothing fillets

This will add small fillets to non-tangential vertices of the path. This can speed up and smooth out tool travel.

Smoothing radius

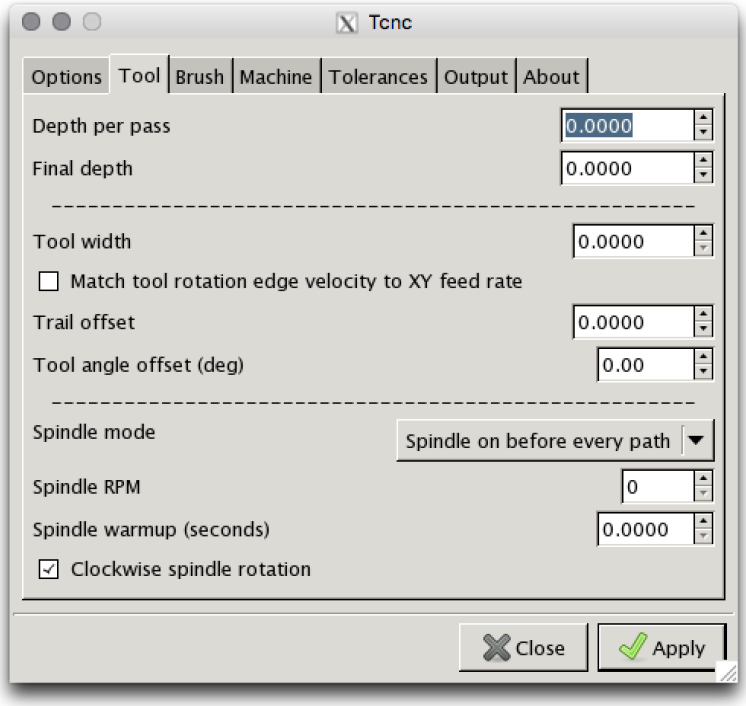
The radius of the smoothing fillets.

Radius of smoothing fillets in document units.

Split paths at non-tangent control points

Split a path at non-tangent (no G1 continuity) control points (vertices). This can avoid weird looking brush rotations at these points if tool offset compensation is turned off or if a corner is too tight for a fillet.

Tool



Depth per pass

Maximum Z depth per pass if multiple passes are needed per path. Ignored if the value is zero.

Final depth

The final tool depth of the last pass. Depth per pass and final depth can be the same if just one pass is required.

Tool width

Width in document units of the tangential tool such as a brush, scraper, or squeegee. This will affect tool path compensation if fillets are enabled.

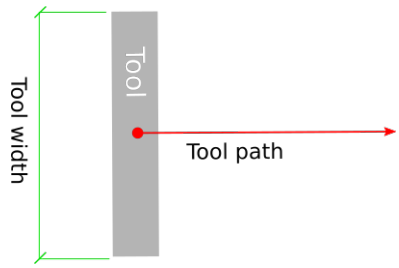


Fig. 2.1: Fig 2. Tool width

Trail offset

Tool trail offset is the distance from the center of rotation to the point of contact with the surface. Flexible tools such as brushes and scrapers will deflect when brought to the work surface. This creates a trailing point of contact and will cause unwanted behavior when making relatively tight turns. Tcnc will try to recalculate the path to compensate for this so that the center of contact follows the original path more closely.

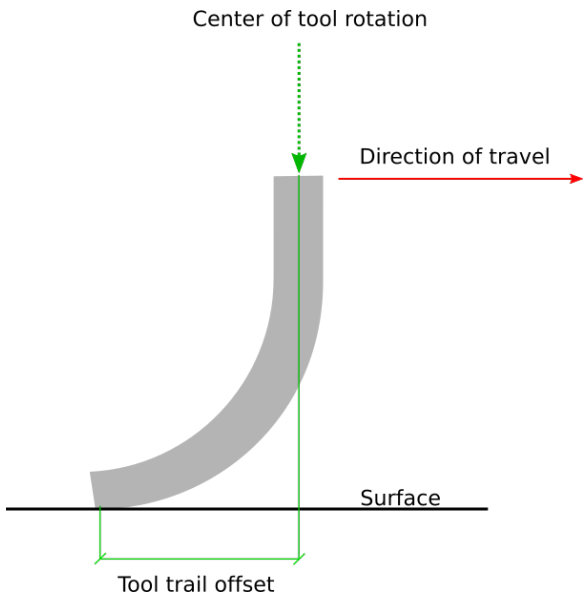


Fig. 2.2: Fig 3. Tool trail offset

Tool angle offset

Offset in degrees of the A axis.

Spindle mode

- No spindle
- Spindle on at start
- Spindle on before every path

Spindle RPM

Spindle speed in revolutions per minute.

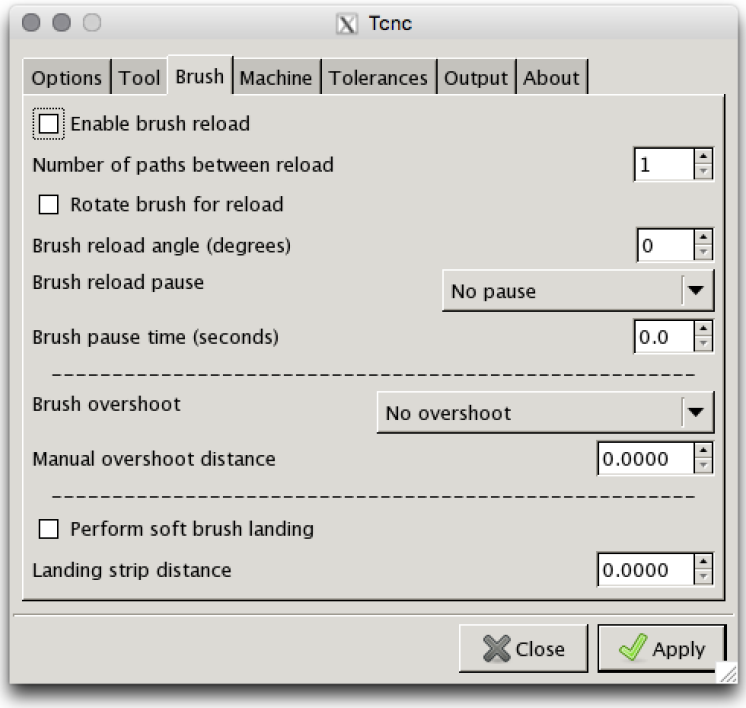
Spindle warmup

Time in seconds to wait for the spindle to get up to speed and warm up.

Clockwise spindle rotation

Direction of spindle rotation. Default is clockwise.

Brush



Enable brush reload

Brushes often require more paint and this enables a brush reload sequence for manual brush reloading. Automated brush reloading is not currently supported since it would be extremely machine dependent.

Number of paths between reload

Number of paths to output before a reload sequence is performed.

Rotate brush for reload

Rotate the brush to the specified reload angle before each path.

Brush reload pause

- **Pause brush until restart** Issue a pause command which will wait until the user starts/unpauses the machine. Usually this is done via Axis or some other UI.
- Pause brush for time:

Brush pause time

Amount of time in seconds that the tool feed will pause to allow manual reloading of the brush.

Brush overshoot

- **Overshoot based on tool width** The overshoot distance will be one half the tool width. Makes closed paths look a little better under some circumstances.
- Manual overshoot distance:

Manual overshoot distance

If *Manual overshoot distance* is selected this determines the overshoot distance.

Perform soft brush landing

A flexible tool such as a brush will develop its trail distance only after touching and pressing into the work surface. This will use the Z depth and trail offset to create a ramp trajectory along the Z and XY axes. See Fig 4.

Landing strip distance

The landing strip is a straight line segment appended to the soft landing trajectory, essentially a mirror of the brush overshoot. This will be prepended to the tool path.

Machine

Units

G code (machine) units. These can be inches or millimeters.

- **Infer from document** Inches or mm inferred from document units. For example if the document is in imperial units then inches will be used, otherwise mm.
- inch
- mm

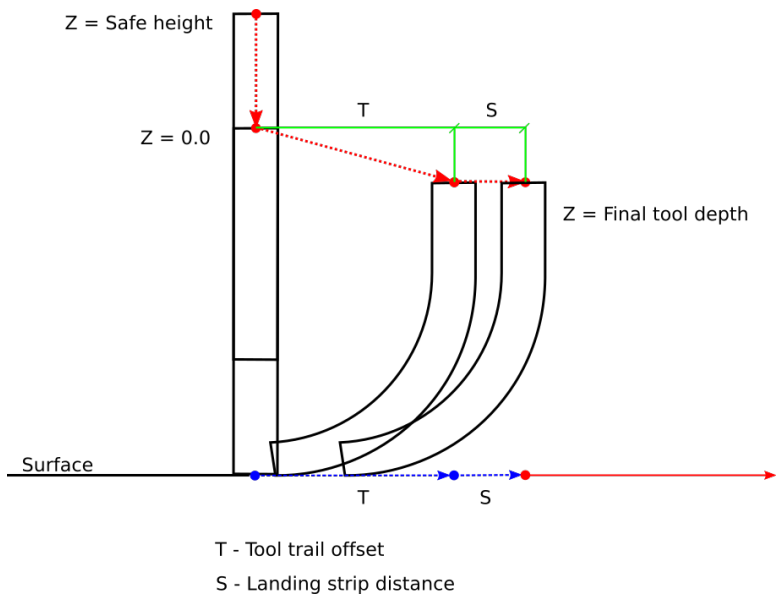


Fig. 2.3: Fig 4. Soft landing Z axis trajectory

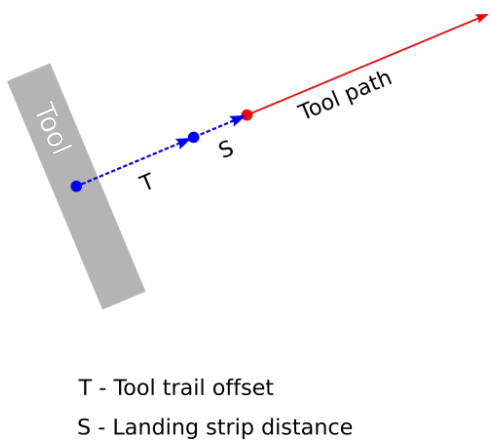
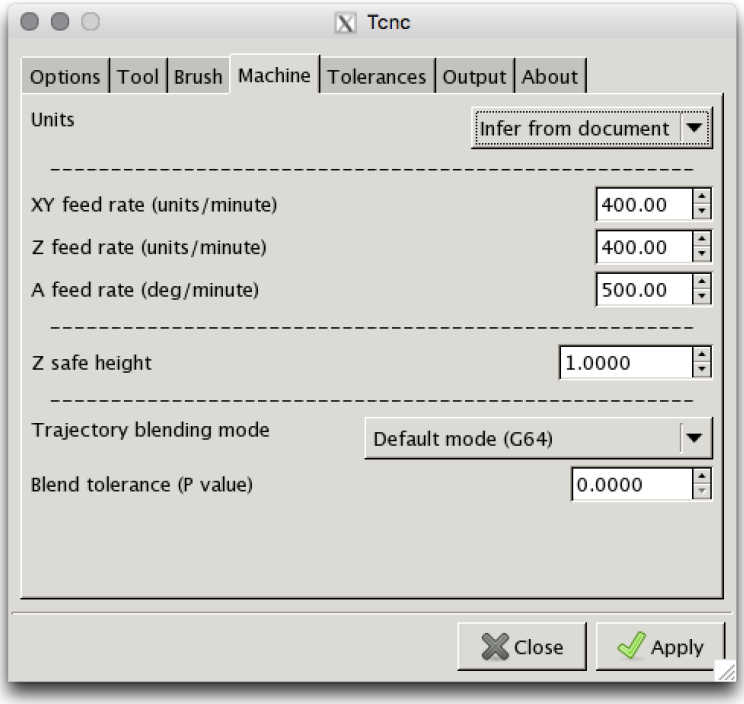


Fig. 2.4: Fig 5. Soft landing XY axis trajectory



XY feed rate

The feed rate of the X and Y axes. In machine units per minute.

Z feed rate

The Z axis (vertical) feed rate. In machine units per minute.

A feed rate

The A (rotational) axis feed rate. In degrees per minute.

Z safe height

The safe height, in machine units, of the Z axis for rapid moves.

Trajectory blending mode

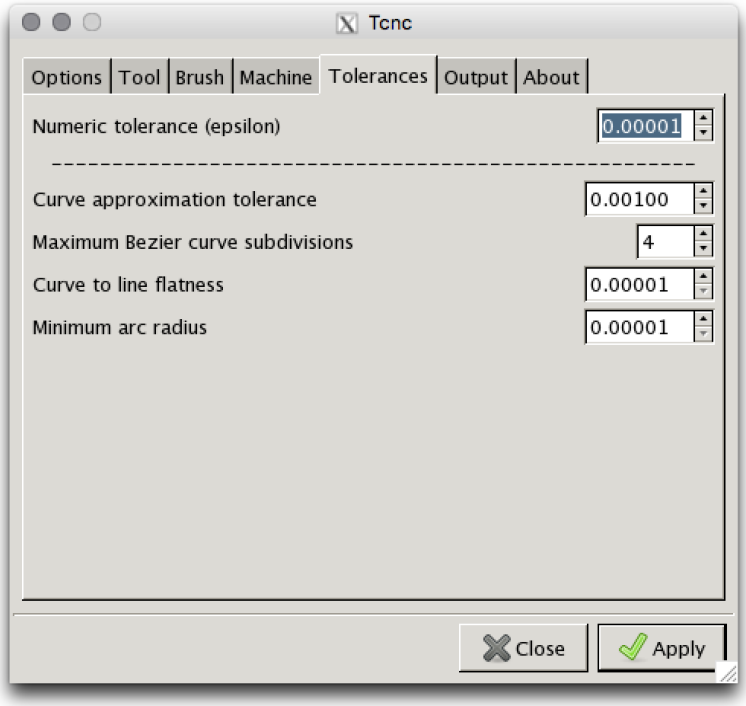
Tool path blending mode used by the machine controller.

- **Default (G64)** The default blending mode which is usually G64 - blending with a default tolerance.
- **Blend with tolerance (G64 P't')** G64 blending with a specified tolerance.
- **Exact path (G61)** Follow the exact path with no blending. Accurate but can be slow. The tool may come to a complete stop at path vertices.

Blend tolerance

The blend tolerance (P) value when using the G64 P‘t‘ blend mode.

Tolerances



Numeric tolerance

This determines the numeric precision of floating point comparisons and the precision (number of digits after the decimal point) of G code output.

Curve approximation tolerance

The maximum distance, in document units, between the approximation and the original curve. Smaller values can result in more accurate approximations but at the expense of slower performance.

Maximum Bezier curve subdivisions

Inkscape paths consist of Bezier curves and to accurately approximate them with circular arcs they may need to be broken down into smaller curves. Larger values can result in more accurate approximations but at the expense of slower performance.

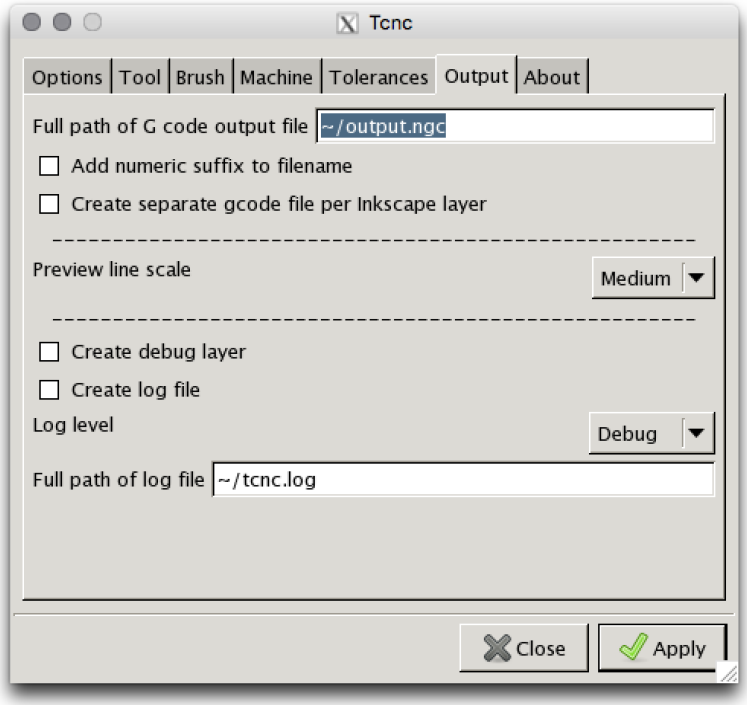
Curve to line flatness

Curves that are flatter than this will be approximated by a straight line. Flatness is the maximum distance from a line between the curve end points and the curve. In document units.

Minimum arc radius

Arcs with a radius smaller than this will be replaced by a straight line. In document units. This can avoid unexpected rotations of the tangential tool when encountering tiny spurious curves that might be in the input path.

Output



Full path of G code output file

Add numeric suffix to filename

Create separate output file per Inkscape layer

Preview line scale

CHAPTER 3

Source Code

Source code can be found at: https://github.com/utlco/tcnc

```
git clone https://github.com/utlco/tcnc.git
```

3.1 Modules

3.1.1 Inkscape extensions

Tcnc

An Inkscape extension that will output G-code from selected paths. The G-code is suitable for a CNC machine that has a tangential tool (ie a knife or a brush).

class tcnc.**Tcnc**

Bases: *inkscape.inkext.InkscapeExtension*

Inkscape plugin that converts selected SVG elements into gcode suitable for a four axis (XYZA) CNC machine with a tangential tool, such as a knife or a brush, that rotates about the Z axis.

OPTIONSPEC = (<ExtOption at 0x7fd1d80e5ab8: --origin-ref>, <ExtOption at 0x7fd1d7ae31

run()

Main entry point for Inkscape plugins.

Quasink

An Inkscape extension to create quasicrystalline/Penrose tesselations.

class quasink.**IdentityProjector**

Bases: object

Identity projection. No distortion.

project(*p*)

class quasink.**SphericalProjector**(*center*, *radius*, *invert=False*)

Bases: *quasink.IdentityProjector*

Project a point on to a sphere.

project(*p*)

class quasink.**QuasiExtension**

Bases: *inkscape.inkext.InkscapeExtension*

Inkscape plugin that creates quasi-crystal-like patterns. Based on quasi.c by Eric Weeks.

run()

Main entry point for Inkscape plugins.

POLYGON_SORT_INSIDE_OUT = 1

POLYGON_SORT_NONE = 0

POLYGON_SORT_OUTSIDE_IN = 2

Voronoi

An Inkscape extension to create a Voronoi diagram from points derived from input SVG geometry.

---

class voronoi.**Voronoi**

Bases: *inkscape.inkext.InkscapeExtension*

Inkscape plugin that creates Voronoi diagrams.

run()

Main entry point for Inkscape plugins.

Polypath

An Inkscape extension to create paths from a collection of vertices.

---

class polypath.**PolyPath**

Bases: *inkscape.inkext.InkscapeExtension*

Inkscape plugin that traces paths on edge connected graphs.

OPTIONSPEC = (<ExtOption at 0x7fd1d66ee290: --epsilon>, <ExtOption at 0x7fd1d66ee2d8:

run()

Main entry point for Inkscape extension.

Polysmooth

Smooth non-G1 path nodes using Bezier curves and draw it as SVG.

Works for polyline/polygons made of line/arc segments.

This can be invoked as an Inkscape extension or from the command line.

class polysmooth.**PolySmooth**

Bases: *inkscape.inkext.InkscapeExtension*

An Inkscape extension that smoothes polygons.

OPTIONSPEC = (<ExtOption at 0x7fd1d64ecbd8: --simplify>, <ExtOption at 0x7fd1d64ecc68

run()

Main entry point for Inkscape extensions.

simplify_polylines(*path*, *tolerance*)

Simplify any polylines in the path.

Sinewave

Approximate a sine wave using Bezier curves and draw it as SVG.

This can be invoked as an Inkscape extension or from the command line.

class sinewave.**SineWave**

Bases: *inkscape.inkext.InkscapeExtension*

An Inkscape extension that draws a sine wave using Bezier curves.

run()

Main entry point for Inkscape extensions.

3.1.2 inkscape - Inkscape extension package

inkscape.inkext

Inkscape extension boilerplate class.

class inkscape.inkext.**ExtOption**(**opts*, ***attrs*)

Subclass of optparse.Option that adds additional type checkers for handling Inkscape-specific types. This should be used in lieu of optparse.Option for Inkscape extensions.

class inkscape.inkext.**InkscapeExtension**

Base class for Inkscape extensions. This does not depend on Inkscape being installed and can be invoked as a stand-alone application. If an input document is not specified a new blank SVG document will be created.

This replaces inkex.Effect which ships with Inkscape.

See also:

inkex.Effect

options = None

Parsed command line option values available to the extension

svg = None

SVG context for this extension

debug_svg = None
Debug SVG context if a debug layer has been created

main(*optionspec=None*, *flip_debug_layer=False*, *debug_layer_name=u'inkext_debug'*)
Main entry point for the extension.

Parameters

- **optionspec** – An optional list of `optarg.Option` objects.
- **flip_debug_layer** – Flip the Y axis of the debug layer. This is useful if the GUI coordinate origin is at the bottom left. Default is False.

run()
Extensions override this method to do the actual work.

get_elements(*selected_only=False*)
Get selected document elements.

Tries to get selected elements first. If nothing is selected and *selected_only* is False then <strike>either the currently selected layer or</strike> the document root is returned. The elements may or may not be visible.

Parameters **selected_only** – Get selected elements only. Default is False.

Returns A (possibly empty) iterable collection of elements.

errormsg(*msg*)
Intended for end-user-visible error messages. Inkscape displays stderr output with an error dialog.

create_log(*log_path=None*, *log_level=u'DEBUG'*)
Create a log file for debug output.

Parameters

- **log_path** – Path to log file. If None or empty the log path name will be the command line invocation name (argv[0]) with a '.log' suffix in the user's home directory.
- **log_level** – Log level: 'DEBUG', 'INFO', 'WARNING', 'ERROR', or 'CRITICAL'. Default is 'DEBUG'.

inkscape.inksvg

A simple library for SVG output - but more Inkscape-centric.

inkscape.inksvg.**inkscape_ns**(*tag*)
Prepend the *inkscape* namespace to an element tag.

inkscape.inksvg.**sodipodi_ns**(*tag*)
Prepend the *sodipodi* namespace to an element tag.

inkscape.inksvg.**utlco_ns**(*tag*)
Prepend the *utlco* namespace to an element tag

class inkscape.inksvg.**InkscapeSVGContext**(*document*, **args*, ***kwargs*)

doc_name = None
Inkscape document name

doc_units = None
Inkscape GUI document units

cliprect = None
Document clipping rectangle

get_document_size()
Return width and height of document in user units as a tuple (W, H).

margin_cliprect(*mtop*, **args*)
Create a clipping rectangle based on document bounds with the specified margins. Margin argument order follows CSS margin property rules.

Parameters

- **mtop** – Top margin (user units)
- **mright** – Right margin (user units)
- **mbottom** – Bottom margin (user units)
- **mleft** – Left margin (user units)

Returns A geom.Box clipping rectangle

get_document_name()
Return the name of this document. Default is 'untitled'.

get_document_units()
Return the Inkscape document unit string ('in', 'mm', etc.). This is the document unit used for the UI (dialogs etc.). It might not be the same as the document user unit.

get_selected_layer()
Get the currently selected Inkscape layer element.

Returns The currently selected layer element or None if no layers are selected.

find_layer(*layer_name*)
Find an Inkscape layer by Inkscape layer name.

If there is more than one layer by that name then just the first one will be returned.

Parameters **layer_name** – The Inkscape layer name to find.

Returns The layer Element node or None.

create_layer(*name*, *opacity=None*, *clear=True*, *incr_suffix=False*, *flipy=False*, *tag=None*)
Create an Inkscape layer or return an existing layer.

Parameters

- **name** – The name of the layer to create.
- **opacity** – Layer opacity (0.0 to 1.0).
- **clear** – If a layer of the same name already exists then erase it first if True otherwise just return it. Default is True.
- **incr_suffix** – If a layer of the same name already exists and it is non-empty then add an auto-incrementing numeric suffix to the name (overrides *clear*).
- **flipy** – Add transform to flip Y axis.
- **tag** (*str*) – A layer tag added as an extended attribute. Uses *utlco* namespace. This can be used to tag layers with a custom label.

Returns A new layer or an existing layer of the same name.

set_layer_name(*layer*, *name*)
Rename an Inkscape layer.

get_layer_name(*layer*)
Return the name of the Inkscape layer.

get_parent_layer(*node*)
Return the layer that the node resides in. Returns None if the node is not in a layer.

layer_is_locked(*layer*)

Returns True if the layer is locked, otherwise False.

find(*path*)
Find an element in the current document.

Parameters **path** – XPath path.

Returns The first matching element or None if not found.

get_visible_layers()
Get a list of visible layers

get_layer_elements(*layer*)
Get document elements by layer.

Returns all the visible child elements of the given layer.

Parameters **layer** – The layer root element.

Returns A (possibly empty) list of visible elements.

is_layer(*node*)
Determine if the element is an Inkscape layer node.

get_shape_elements(*rootnode*, *shapetags=(u'path'*, *u'rect'*, *u'line'*, *u'circle'*, *u'ellipse'*, *u'polyline'*, *u'polygon')*, *parent_transform=None*, *skip_layers=None*, *accumulate_transform=True*)
Traverse a tree of SVG nodes and flatten it to a list of tuples containing an SVG shape element and its accumulated transform.

This does a depth-first traversal of <g> and <use> elements.

Hidden elements are ignored.

Parameters

- **rootnode** – The root of the node tree to traverse and flatten. This can be the document root, a layer, or simply a list of element nodes.
- **shapetags** – List of shape element tags that can be fetched. Default is ('path', 'rect', 'line', 'circle', 'ellipse', 'polyline', 'polygon'). Anything else is ignored.
- **parent_transform** – Transform matrix to add to each node's transforms. If None the node's parent transform is used.
- **skip_layers** – A list of layer names (as regexes) to ignore
- **accumulate_transform** – Apply parent transform(s) to element node if True. Default is True.

Returns A possibly empty list of 2-tuples consisting of SVG element and accumulated transform.

inkscape.inksvg.**create_inkscape_document**(*width*, *height*, *doc_units=u'px'*, *doc_id=None*, *doc_name=None*, *layer_name=None*, *layer_id=u'defaultlayer'*)
Create a minimal Inkscape-compatible SVG document.

Parameters

- **width** – The width of the document in user units.
- **height** – The height of the document in user units.
- **doc_units** – The user unit type (i.e. 'in', 'mm', 'pt', 'em', etc.) By default this will be 'px'.
- **doc_id** – The id attribute of the enclosing svg element. If None (default) then a random id will be generated.
- **doc_name** – The name of the document (i.e. 'MyDrawing.svg').
- **layer_name** – Display name of default layer. By default no default layer will be created.
- **layer_id** – Id attribute value of default layer. Default id is 'defaultlayer'.

Returns An lxml.etree.ElementTree

3.1.3 svg - SVG drawing/utility package

svg.svg

A simple library for SVG output.

svg.svg.**svg_ns**(*tag*)
Shortcut to prepend SVG namespace to *tag*.

svg.svg.**xml_ns**(*tag*)
Shortcut to prepend XML namespace to *tag*.

svg.svg.**xlink_ns**(*tag*)
Shortcut to prepend xlink namespace to *tag*.

svg.svg.**strip_ns**(*tag*)
Strip the namespace part from the tag if any.

class svg.svg.**SVGContext**(*document*, *font_size=None*, *x_height=None*)
SVG document context.

New SVG context.

Parameters

- **document** – An SVG ElementTree. The svg 'width' and 'height' attributes MUST be specified.
- **font_height** – CSS font-height in user units.
- **ex_height** – CSS x-height in user units.

classmethod create_document(*width*, *height*, *doc_id=None*, *doc_units=None*)
Create a minimal SVG document.

Returns An ElementTree

classmethod parse(*filename=None*, *huge_tree=True*)
Parse an SVG file (or stdin) and return an SVGContext.

Parameters **filename** – The SVG file to parse. If this is None stdin will be read by default.

Returns An SVGContext

unit2uu (*value*, *from_unit=u'px'*)
Convert a string/float that specifies a value in some source unit (ie '3mm' or '5in') to a float value in user units. The result will be scaled using the viewBox/viewport ratio. See http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute

Parameters

- **value** – A numeric string value with an optional unit identifier suffix (ie '3mm', '10pt, '5in'), or a float value. If the string does not have a unit suffix then *src_unit* will be used.
- **from_unit** – A string specifying the units for the conversion. Default is 'px'.

Returns A float value or 0.0 if the string can't be parsed.

uu2unit (*value*, *to_unit=u'px'*)
Convert a value in user units to a destination unit.

Parameters

- **value** – A float value in user units.
- **to_unit** – Destination unit (i.e. 'in', 'mm', etc.) Default is 'px'.

Returns The converted value.

unit_convert (*value*, *to_unit=u'px'*, *from_unit=u'px'*)
Convert a string that specifies a scalar value in some source unit (ie '3mm' or '5in') to a float value in a destination unit ('px', or user units, by default). SVG/Inkscape units and viewBoxes are confusing... See http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute and http://www.w3.org/TR/SVG/coords.html#Units and http://wiki.inkscape.org/wiki/index.php/Units_In_Inkscape

Parameters

- **value** – A string scalar with an optional unit identifier suffix. (ie '3mm', '10pt, '5.3in')
- **to_unit** – An SVG unit id of the destination unit conversion. Default is 'px' (user units)
- **from_unit** – Optional default source unit. ('px' if unspecified)

Returns A float value or 0.0 if the string can't be parsed.

write (*filename=None*)
Write the SVG to a file or stdout.

set_precision (*precision*)
Set the output precision.

Parameters **precision** – The number of digits after the decimal point.

float_eq (*a*, *b*)
Compare two floats for equality in the SVG context. The two float are considered equal if the difference between them is less than epsilon. This is based on the current SVG numeric precision.

For a discussion of floating point comparisons see: http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

set_default_parent (*parent*)
Set the current default parent (or layer) to :parent:

get_node_by_id (*node_id*)
Find a node in the current document by id attribute.

Parameters **node_id** – The node id attribute value.

Returns A node if found otherwise None.

get_element_transform(*node*, *root=None*)
Get the combined transform of the element and it's combined parent transforms.

Parameters

- **node** – The element node.
- **root** – The document root or where to stop searching.

Returns The combined transform matrix or the identity matrix if none found.

get_parent_transform(*node*, *root=None*)
Get the combined transform of the node's parents.

Parameters

- **node** – The child node.
- **root** – The document root or where to stop searching.

Returns The parent transform matrix or the identity matrix if none found.

node_is_visible(*node*, *check_parent=True*, *_recurs=False*)
Return True if the node is visible. CSS visibility trumps SVG visibility attribute.

The node is not considered visible if the *visibility* style is *hidden* or *collapse* or if the *display* style is *none*. If the *visibility* style is *inherit* or *check_parent* is True then the visibility is determined by the parent node.

Parameters

- **node** – An etree.Element node
- **check_parent** – Recursively check parent nodes for visibility

Returns True if the node is visible otherwise False.

parse_transform_attr(*transform_attr*)
Parse an SVG transform attribute.

Parameters **transform_attr** – A string containing the SVG transform list.

Returns A single affine transform matrix.

scale_inline_style(*inline_style*)
For any inline style attribute name that ends with 'width', 'height', or 'size' scale the numeric value with an optional unit id suffix by converting it to user units with no unit id.

styles_from_templates(*style_templates*, *default_map*, *template_map=None*)
Populate a dictionary of styles given a dictionary of templates and mappings. If a template key string ends with 'width', 'height', or 'size' it is assumed that the value is a numeric value with an optional unit id suffix and it will be automatically converted to user units with no unit id.

Parameters

- **style_templates** – A dictionary of style names to inline style template strings.
- **default_map** – A dictionary of template keys to default values. This must contain all template identifiers.
- **template_map** – A dictionary of template keys to values that override the defaults. Default is None.

Returns A dictionary of inline styles.

node_is_group(*node*)
Return True if the node is an SVG group.

create_rect(*position*, *width*, *height*, *style=None*, *parent=None*)
Create an SVG rect element.

create_circle(*center*, *radius*, *style=None*, *parent=None*)
Create an SVG circle element.

create_ellipse(*center*, *rx*, *ry*, *angle*, *style=None*, *parent=None*)
Create an SVG ellipse.

create_line(*p1*, *p2*, *style=None*, *parent=None*, *attrs=None*)
Create an SVG path consisting of one line segment.

create_circular_arc(*startp*, *endp*, *radius*, *sweep_flag*, *style=None*, *parent=None*, *attrs=None*)
Create an SVG circular arc.

create_curve(*startp*, *cp1=None*, *cp2=None*, *p2=None*, *style=None*, *parent=None*, *attrs=None*)
Create an SVG cubic bezier curve.

Parameters

- **startp** – The start point of the curve or an indexable collection of four control points.
- **style** – A CSS style string.
- **parent** – The parent element (or Inkscape layer).
- **attrs** – Dictionary of SVG element attributes.

Returns An SVG path Element node.

create_polygon(*vertices*, *close_polygon=True*, *close_path=False*, *style=None*, *parent=None*, *attrs=None*)
Create an SVG path describing a polygon.

Parameters

- **vertices** – An iterable collection of 2D polygon vertices. A vertice being a tuple containing x,y coordicates.
- **close_polygon** – Close the polygon if it isn't already. Default is True.
- **close_path** – Close and join the the path ends by appending 'Z' to the end of the path ('d') attribute. Default is False.
- **style** – A CSS style string.
- **parent** – The parent element (or Inkscape layer).
- **attrs** – Dictionary of SVG element attributes.

Returns An SVG path Element node, or None if the list of vertices is empty.

create_polypath(*path*, *close_path=False*, *style=None*, *parent=None*, *attrs=None*)
Create an SVG path from a sequence of line and arc segments.

Parameters

- **path** – An iterable sequence of line, circular arc, or cubic Bezier curve segments. A line segment is a 2-tuple containing the endpoints. An arc segment is a 5-tuple containing the start point, end point, radius, angle, and center, respectively. The arc center is ignored. A cubic bezier segment is a 4-tuple containing the first endpoint, the first control point, the second control point, and the second endpoint.
- **close_path** – Close and join the the path ends by appending 'Z' to the end of the path ('d') attribute. Default is False.
- **style** – A CSS style string.

- **parent** – The parent element (i.e. Inkscape layer).
- **attrs** – Dictionary of SVG element attributes.

Returns An SVG path Element node, or None if the path is empty.

create_simple_marker(*marker_id*, *d*, *style*, *transform*, *replace=False*)
Create an SVG line end marker glyph.

The glyph Element is placed under the document root.

create_path(*attrs*, *style=None*, *parent=None*)
Create an SVG path element.

create_text(*text*, *x*, *y*, *line_height=None*, *style=None*, *parent=None*)
Create a text block.

add_elem(*node*, *parent=None*)
Add the element to the parent or the current parent if parent is None.

svg.svg.**path_tokenizer**(*path_data*)
Tokenize SVG path data.

A generator that yields tokens from path data. This will yield a tuple containing a command token or a numeric parameter token followed by a boolean flag that is True if the token is a command and False if the token is a numeric parameter.

Yields A 2-tuple with token and token type hint.

svg.svg.**parse_path**(*path_data*)
Parse an SVG path definition string. Converts relative values to absolute and shorthand commands to canonical (ie. H to L, S to C, etc.) Terminating Z (or z) converts to L.

If path syntax errors are encountered, parsing will simply stop. No exceptions are raised. This is by design so that parsing is relatively forgiving of input.

Implemented as a generator so that memory usage can be minimized for very long paths.

Parameters **path_def** – The 'd' attribute value of a SVG path element.

Yields A path component 2-tuple of the form (cmd, params).

svg.svg.**explode_path**(*path_data*)
Break the path at node points into component segments.

Parameters **path_def** – The 'd' attribute value of a SVG path element.

Returns A list of path 'd' attribute values.

svg.svg.**create_svg_document**(*width*, *height*, *doc_units=u'px'*, *doc_id=None*)
Create a minimal SVG document tree.

The svg element *viewbox* attribute will maintain the size and attribute ratio as specified by width and height.

Parameters

- **width** – The width of the document in user units.
- **height** – The height of the document in user units.
- **doc_units** – The user unit type (i.e. 'in', 'mm', 'pt', 'em', etc.) By default this will be 'px'.
- **doc_id** – The id attribute of the enclosing svg element. If None (default) then a random id will be generated.

Returns An lxml.etree.ElementTree

`svg.svg.random_id`(*prefix=u'_svg'*, *rootnode=None*)
Create a random XML id attribute value.

Parameters

- **prefix** – The prefix prepended to a random number. Default is '_svg'.
- **rootnode** – The root element to search for id name collisions. Default is None in which case no search will be performed.

Returns A random id string that has a fairly low chance of collision with previously generated ids.

`svg.svg.get_node_by_id`(*rootnode*, *node_id*)
Find a node in the current document by id attribute.

Parameters

- **rootnode** – The root element to search.
- **node_id** – The node id attribute value.

Returns A node if found otherwise None.

`svg.svg.transform_attr`(*matrix*)

svg.css

A simple library of functions to parse and format CSS style properties.

`svg.css.CSS_COLORS = {u'indigo': (75, 0, 130), u'gold': (255, 215, 0), u'hotpink': (255,`
CSS color names to hex rgb values.

See: https://www.w3.org/TR/css3-color/#svg-color

`svg.css.inline_style_to_dict`(*inline_style*)
Create a dictionary of style properties from an inline style attribute.

Parameters **inline_style** – A string containing the value of a CSS *style* attribute.

Returns A dictionary of style properties.

`svg.css.dict_to_inline_style`(*style_map*)
Create an inline style attribute string from a dictionary of CSS style properties.

Parameters **style_map** – A dictionary of CSS style properties.

Returns A string containing inline CSS style properties.

`svg.css.csscolor_to_rgb`(*css_color*)
Parse a CSS color property value into an RGB value.

Parameters **css_color** – A CSS color property string. I.e. "#ffc0ee" or "white".

Returns (r, g, b).

Return type A tuple containing the RGB values

See: https://developer.mozilla.org/en-US/docs/Web/CSS/color

`svg.css.csshex_to_rgb`(*hex_color*)
Convert a CSS hex color property to RGB.

Parameters **hex_color** – A CSS hex property string.

Returns (r, g, b). Returns (0, 0, 0) by default if the hex value can't be parsed.

Return type The RGB value as a tuple of three integers

`svg.css.`**`cssrgb_to_rgb`**(*rgb_color*)
Convert a CSS rgb or rgba color property to RGB.

Parameters **`rgb_color`** – A CSS rgb property string: i.e. *rgb(r, g, b)*.

Returns The RGB value as a tuple or list of three integers plus an optional fourth float value if there is an alpha channel. Returns (0, 0, 0) by default if the hex value can't be parsed.

`svg.css.`**`parse_channel_value`**(*value*)
Parse a CSS color channel value.

Parameters **`value`** – A valid CSS color channel value string. Can be an integer number or an integer percentage.

Returns An integer value between 0 and 255. Default is 0 if the value isn't a valid channel value.

`svg.css.`**`csscolor_to_cssrgb`**(*color*)
Returns a CSS color in the form #rrggbb. If *color* is a numeric value then it is converted to a grayscale number. If the number is a floating point value between zero and one then it is scaled up to 0-255 grayscale. If the CSS color can't be parsed then #000000 is returned.

Parameters **`possibly malformed CSS color string or number.`** (A) –

Returns A CSS color in the form #rrggbb.

svg.geomsvg

Methods for converting SVG shape elements to geometry objects.

`svg.geomsvg.`**`svg_to_geometry`**(*svg_elements*, *parent_transform=None*)
Convert the SVG shape elements to Line, Arc, and/or CubicBezier segments, and apply node/parent transforms. The coordinates of the segments will be absolute with respect to the parent container.

Parameters

- **`svg_elements`** – An iterable collection of 2-tuples consisting of SVG Element node and transform matrix.
- **`parent_transform`** – An optional parent transform to apply to all nodes. Default is None.

Returns A list of paths, where a path is a list of one or more segments made of Line, Arc, or CubicBezier objects.

`svg.geomsvg.`**`svg_element_to_geometry`**(*element*, *element_transform=None*, *parent_transform=None*)
Convert the SVG shape element to a list of one or more Line, Arc, and/or CubicBezier segments, and apply node/parent transforms. The coordinates of the segments will be absolute with respect to the parent container.

Parameters

- **`element`** – An SVG Element shape node.
- **`element_transform`** – An optional transform to apply to the element. Default is None.
- **`parent_transform`** – An optional parent transform to apply to the element. Default is None.

Returns A list of zero or more paths. A path being a list of zero or more Line, Arc, EllipticalArc, or CubicBezier objects.

`svg.geomsvg.`**`parse_path_geom`**(*path_data*, *ellipse_to_bezier=False*)
Parse SVG path data and convert to geometry objects.

Parameters

- **`path_data`** – The *d* attribute value of an SVG path element.
- **`ellipse_to_bezier`** – Convert elliptical arcs to bezier curves if True. Default is False.

Returns A list of zero or more subpaths. A subpath being a list of zero or more Line, Arc, EllipticalArc, or CubicBezier objects.

`svg.geomsvg.`**`convert_rect`**(*element*)
Convert an SVG rect shape element to four geom.Line segments.

Parameters **`element`** – An SVG 'rect' element of the form <rect x='X' y='Y' width='W' height='H'/>

Returns A clockwise wound polygon as a list of geom.Line segments.

`svg.geomsvg.`**`convert_line`**(*element*)
Convert an SVG line shape element to a geom.Line.

Parameters **`element`** – An SVG 'line' element of the form: <line x1='X1' y1='Y1' x2='X2' y2='Y2/>

Returns geom.Line((x1, y1), (x2, y2))

Return type A line segment

`svg.geomsvg.`**`convert_circle`**(*element*)
Convert an SVG circle shape element to four circular arc segments.

Parameters **`element`** – An SVG 'circle' element of the form: <circle r='RX' cx='X' cy='Y'/>

Returns A counter-clockwise wound list of four circular geom.Arc segments.

`svg.geomsvg.`**`convert_ellipse`**(*element*)
Convert an SVG ellipse shape element to a geom.Ellipse.

Parameters **`element`** – An SVG 'ellipse' element of the form: <ellipse rx='RX' ry='RY' cx='X' cy='Y'/>

Returns A geom.Ellipse.

`svg.geomsvg.`**`convert_polyline`**(*element*)
Convert an SVG *polyline* shape element to a list of line segments.

Parameters **`element`** – An SVG 'polyline' element of the form: <polyline points='x1,y1 x2,y2 x3,y3 [...]'/>

Returns A list of geom.Line segments.

`svg.geomsvg.`**`convert_polygon`**(*element*)
Convert an SVG *polygon* shape element to a list line segments.

Parameters **`element`** – An SVG 'polygon' element of the form: <polygon points='x1,y1 x2,y2 x3,y3 [...]'/>

Returns A list of geom.Line segments. The polygon will be closed.

3.1.4 geom - Basic 2D geometry package

geom.const

Commonly used constants used in the geometry package.

The following read-only constants are defined:

EPSILON: A small floating point value which is the maximum numeric distance between two floating point numbers for them to be considered equal.

EPSILON_PRECISION: The number of digits after the decimal point of EPSILON.

EPSILON_FLOAT_FMT: A format string for displaying numbers that have PRECISION number of digits after the decimal point.

The tolerance value (EPSILON) *must* be set once (and only once) before using the geometry package. This is enforced.

Useful values are small numbers in the range of 1e-09 to 1e-05. It depends on the application and the size of numbers used with the package.

If set_epsilon() is not called by the user then a default value will be set the first time the EPSILON attribute is accessed. It cannot then be modified.

geom.const.**TAU = 6.283185307179586**
Commonly used constant PI*2

geom.const.**EPSILON = 1e-06**
Tolerance value used for floating point comparisons

geom.const.**EPSILON2 = 1e-12**
Handy for comparing distance**2 when avoiding sqrt

geom.const.**EPSILON_PRECISION = 6**
Number of digits after decimal point

geom.const.**EPSILON_FLOAT_FMT = u'%.6f'**
Format string for float values that matches precision

geom.const.**set_epsilon**(*value*)
Set the global absolute error tolerance value and rounding limit for approximate floating point comparison operations. This value is accessible via the geom.EPSILON global variable.

The default value of 1e-06 is suitable for values that are in the 'countable range'. You may need a larger epsilon when using large absolute values, and a smaller value for very small values close to zero. Otherwise approximate comparison operations may not behave as expected.

In general, this should be called only once before using any modules that refer to EPSILON.

Parameters **value** – Float tolerance value.

geom.const.**float_eq**(*value1*, *value2*)
Compare two floats for equality. The two float are considered equal if the difference between them is less than EPSILON.

For a discussion of floating point comparisons see: http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Parameters

- **value1** – Float value
- **value2** – Float value

Returns True if the two values are approximately equal.

`geom.const.`**`is_zero`**(*value*)
Determine if the float value is essentially zero.

A shortcut for float_eq(n, 0.0).

`geom.const.`**`float_round`**(*value*)
Round the value to a rounding precision corresponding to EPSILON.

geom.transform2d

Basic 2D affine transform matrix operations.

`geom.transform2d.`**`is_identity_transform`**(*m*)
Return True if the matrix is the identity matrix.

`geom.transform2d.`**`compose_transform`**(*m1*, *m2*)
Combine two matrices by multiplying them.

Parameters

- **`m1`** – 2X3 2D transform matrix.
- **`m2`** – 2X3 2D transform matrix.

Note: *m2* is applied before (to) *m1*

`geom.transform2d.`**`matrix_rotate`**(*angle*, *origin=(0.0, 0.0)*)
Create a transform matrix to rotate about the origin.

Parameters

- **`angle`** – Rotation angle in radians.
- **`origin`** – Optional rotation origin. Default is (0,0).

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_translate`**(*x*, *y*)
Create a transform matrix to translate (move).

Parameters

- **`x`** – translation along X axis
- **`y`** – translation along Y axis

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_scale`**(*scale_x*, *scale_y*, *origin=None*)
Create a transform matrix to scale.

Parameters

- **`scale_x`** – X axis scale factor
- **`scale_y`** – Y axis scale factor
- **`origin`** – Optional scale origin. Default is (0,0).

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_scale_translate`**(*scale_x*, *scale_y*, *offset_x*, *offset_y*)
Create a transform matrix to scale and translate.

Parameters

- **`scale_x`** – X axis scale factor
- **`scale_y`** – Y axis scale factor
- **`offset_x`** – translation along X axis
- **`offset_y`** – translation along Y axis

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_skew_x`**(*angle*)
Create a transform matrix to skew along X axis by *angle*.

Parameters **`angle`** – Angle in radians to skew.

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_skew_y`**(*angle*)
Create a transform matrix to skew along Y axis by *angle*.

Parameters **`angle`** – Angle in radians to skew.

Returns A transform matrix as 2x3 tuple

`geom.transform2d.`**`matrix_apply_to_point`**(*matrix*, *p*)
Return a copy of *p* with the transform matrix applied to it.

`geom.transform2d.`**`canonicalize_point`**(*p*, *origin*, *theta*)
Canonicalize the point so that the origin is (0, 0) and axis rotation is zero.

This just rotates then translates the point.

Parameters

- **`p`** – The point to canonicalize (x, y).
- **`origin`** – The origin offset as a 2-tuple (X, Y).
- **`theta`** – The axis rotation angle.

Returns A point as 2-tuple

`geom.point`

Basic 2D point/vector.

`class` `geom.point.`**`P`**
Two dimensional immutable point (vector).

Represented as a simple tuple (x, y) so that it is compatible with many other libraries.

If `y` is None `x` is assumed to be a tuple or list containing both x and y.

Parameters

- **`x`** – X coordinate. Type float.
- **`y`** – Y coordinate. Type float.

`x`
The X (horizontal) coordinate.

y
The Y (vertical) coordinate.

static max_point()
Create a point with max X and Y values.

static min_point()
Create a point with min X and Y values.

static from_polar(*r*, *angle*)
Create a Cartesian point from polar coordinates.

See http://en.wikipedia.org/wiki/Polar_coordinate_system

Parameters

- **r** – Magnitude (radius)
- **angle** – Angle in radians

Returns A point.

to_polar()
Return the polar coordinates of this vector as a tuple containing the magnitude (radius) and angle respectively (r, a).

is_zero()
Return True if X and Y are both within EPSILON distance to zero.

almost_equal(*other*, *tolerance=None*)
Compare points for geometric equality.

Parameters

- **other** – Vector (point) being compared. A 2-tuple (x, y).
- **tolerance** – Max distance between the two points. Default is `EPSILON`.

Returns True if distance between the points < *tolerance*.

length()
The length or scalar magnitude of the vector.

Returns Distance from (0, 0).

length2()
The square of the length of the vector.

unit()
The vector scaled to unit length. If the vector length is zero, a null (0, 0) vector is returned.

Returns A copy of this vector scaled to unit length.

Return type P

normal(*left=True*)
Return a vector perpendicular to this one.

Parameters **left** – left of vector if True, otherwise right. Default is True.

mirror()
This vector flipped 180d

dot(*other*)
Compute the dot product with another vector.

Equivalent to |p1|*|p2|*cos(theta) where theta is the angle between the two vectors.

See: http://en.wikipedia.org/wiki/Dot_product

Parameters **other** – The vector with which to compute the dot product. A 2-tuple (x, y).

Returns A scalar dot product.

cross(*other*)

Compute the cross product with another vector. Also called the perp-dot product for 2D vectors. Also called determinant for 2D matrix.

See: http://mathworld.wolfram.com/PerpDotProduct.html http://www.gamedev.net/topic/441590-2d-cross-product/

From Woodward: The cross product generates a new vector perpendicular to the two that are being multiplied, with a length equal to the (ordinary) product of their lengths.

Parameters **other** – The vector with which to compute the cross product.

Returns A scalar cross product.

angle()

The angle of this vector relative to the x axis in radians.

Returns A float value between -pi and pi.

angle2(*p1*, *p2*)

The angle formed by p1->self->p2.

The angle is negative if p1 is to the left of p2.

Parameters

- **p1** – First point as 2-tuple (x, y).
- **p2** – Second point as 2-tuple(x, y).

Returns The angle in radians between -pi and pi. Returns 0 if points are coincident.

ccw_angle2(*p1*, *p2*)

The counterclockwise angle formed by p1->self->p2.

Parameters

- **p1** – First point as 2-tuple (x, y).
- **p2** – Second point as 2-tuple(x, y).

Returns An angle in radians between 0 and 2*math.pi.

bisector(*p1*, *p2*)

The bisector between the angle formed by p1->self->p2.

Parameters

- **p1** – First point as 2-tuple (x, y).
- **p2** – Second point as 2-tuple (x, y).

Returns A unit vector with origin at *self*.

distance(*p*)

Euclidean distance from this point to another point.

Parameters **p** – The other point as a 2-tuple (x, y).

Returns The Euclidean distance.

distance2(*p*)
Euclidean distance squared to other point. This can be used to compare distances without the expense of a sqrt.

distance_to_line(*p1*, *p2*)
Euclidean distance from this point to it's normal projection on a line that intersects the given points.

See: http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/

Parameters

- **p1** – First point on line.
- **p2** – Second point on line.

Returns Normal distance to line.

normal_projection(*p*)
The unit distance from the origin that corresponds to the projection of the specified point on to the line described by this vector.

Parameters **p** – A vector (point) as 2-tuple (x, y).

inside_triangle2D(*A*, *B*, *C*)
Test if this point lies inside the triangle defined by points A, B, and C. Where ABC is clockwise or counter-clockwise.

See http://www.sunshine2k.de/stuff/Java/PointInTriangle/PointInTriangle.html

Parameters

- **A** – First point of triangle as 2-tuple (x, y)
- **B** – Second point of triangle as 2-tuple (x, y)
- **C** – Third point of triangle as 2-tuple (x, y)

Returns True if this point lies within the triangle ABC.

orientation(*p2*, *p3*)
Determine the direction defined by the three points p1->p2->p3. *p1* being this point.

Parameters

- **p2** – Second point as 2-tuple (x, y).
- **p3** – Third point as 2-tuple (x, y).

Returns Positive if self->p2->p3 is clockwise (right), negative if counterclockwise (left), zero if points are colinear.

transform(*matrix*)
Return a copy of this point with the transform matrix applied to it.

rotate(*angle*, *origin=(0.0, 0.0)*)
Return a copy of this point rotated about the origin by *angle*.

mag()
The length or scalar magnitude of the vector.

Returns Distance from (0, 0).

normalized()
The vector scaled to unit length. If the vector length is zero, a null (0, 0) vector is returned.

Returns A copy of this vector scaled to unit length.

Return type P

perpendicular(*left=True*)
Return a vector perpendicular to this one.

Parameters **left** – left of vector if True, otherwise right. Default is True.

geom.line

Basic 2D line/segment geometry.

class geom.line.**Line**
Two dimensional immutable line segment defined by two points.

Parameters

- **p1** – Start point as 2-tuple (x, y).
- **p2** – End point as 2-tuple (x, y).

static from_polar(*startp*, *length*, *angle*)
Create a Line given a start point, magnitude (length), and angle.

p1
The start point of this line segment.

p2
The end point of this line segment.

length()
Return the length of this line segment.

slope()
Return the slope of this line.

slope_intercept()
The slope-intercept equation for this line. Where the equation is of the form: $y = mx + b$, where m is the slope and b is the y intercept.

Returns Slope and intercept as 2-tuple (m, b)

general_equation()
Compute the coefficients of the general equation of this line. Where the equation is of the form $ax + by + c = 0$.

See: http://www.cut-the-knot.org/Curriculum/Calculus/StraightLine.shtml

Returns A 3-tuple (a, b, c)

angle()
The angle of this line segment in radians.

start_tangent_angle()
The direction of this line segment from the start point in radians. This is the same as the angle. For Lines the start and end tangent angle are the same.

end_tangent_angle()
The direction of this line segment from the end point in radians. This is the same as the angle. For Lines the start and end tangent angle are the same.

bounding_box()
Bounding box.

transform(*matrix*)

Returns A copy of this line with the transform matrix applied to it.

midpoint()

Returns The midpoint of this line segment.

bisector()

Returns A line that is perpendicular to and passes through the midpoint of this line. Also called the perpendicular bisector. Essentially this line segment is rotated 90deg about its midpoint.

offset(*distance*)

Offset of this line segment. :param distance: The distance to offset the line by.

Returns A line segment parallel to this one and offset by *distance*. If offset is < 0 the offset line will be to the right of this line, otherwise to the left. If offset is zero or the line segment length is zero then this line is returned.

mu(*p*)

The unit distance from the first end point of this line segment to the specified collinear point. It is assumed that the point is collinear, but this is not checked.

subdivide(*mu*)

Subdivide this line into two lines at the given unit distance from the start point.

Parameters **mu** – location of subdivision, where 0.0 < *mu* < 1.0

Returns A tuple containing two Lines.

point_at(*mu*)

Return the point that is unit distance *mu* from this segment's first point. The segment's first point would be at *mu=0.0* and the second point would be at *mu=1.0*.

Parameters **mu** – Unit distance from p1

Returns The point at *mu*

normal_projection(*p*)

Return the unit distance from this segment's first point that corresponds to the projection of the specified point on to this line.

Parameters **p** – point to project on to line

Returns A value between 0.0 and 1.0 if the projection lies between the segment endpoints. The return value will be < 0.0 if the projection lies south of the first point, and > 1.0 if it lies north of the second point.

normal_projection_point(*p*, *segment=False*)

Return the point on this line segment that corresponds to the projection of the specified point.

Parameters

- **p** – point to project on to line
- **segment** – if True and if the point projection lies outside the two end points that define this line segment then return the closest endpoint. Default is False.

distance_to_point(*p*, *segment=False*)

Return the Euclidian distance from the specified point and its normal projection on to this line or segment.

See http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html http://paulbourke.net/geometry/pointlineplane/

Parameters

- **p** – point to project on to line
- **segment** – if True and if the point projection lies outside the two end points that define this line segment then return the shortest distance to either of the two endpoints. Default is False.

intersection_mu(*other*, *segment=False*, *seg_a=False*, *seg_b=False*)

Line intersection.

http://paulbourke.net/geometry/pointlineplane/ and http://mathworld.wolfram.com/Line-LineIntersection.html

Parameters

- **other** – line to test for intersection. A 4-tuple containing line endpoints.
- **segment** – if True then the intersection point must lie on both segments.
- **seg_a** – If True the intersection point must lie on this line segment.
- **seg_b** – If True the intersection point must lie on the other line segment.

Returns The unit distance from the segment starting point to the point of intersection if they intersect. Otherwise None if the lines or segments do not intersect.

intersection(*other*, *segment=False*, *seg_a=False*, *seg_b=False*)

Return the intersection point (if any) of this line and another line.

See: <http://paulbourke.net/geometry/pointlineplane/> and <http://mathworld.wolfram.com/Line-LineIntersection.html>

Parameters

- **other** – line to test for intersection. A 4-tuple containing line endpoints.
- **segment** – if True then the intersection point must lie on both segments.
- **seg_a** – If True the intersection point must lie on this line segment.
- **seg_b** – If True the intersection point must lie on the other line segment.

Returns A point if they intersect otherwise None.

intersects(*other*, *segment=False*)

Return True if this segment intersects another segment.

is_coincident(*other*, *tolerance=None*)

Return True if this line segment is coincident with another segment.

Parameters

- **other** (*tuple*) – Another line segment as a 2-tuple of end point tuples.
- **tolerance** (*float*) – A floating point comparison tolerance. Default is geom.const.EPSILON.

is_parallel(*other*, *inline=False*)

Determine if this line segment is parallel with another line.

Parameters

- **other** (*tuple*) – The other line as a tuple of two points.
- **inline** (*bool*) – The other line must also be inline with this segment.

Returns

True if the other segment is parallel. If *inline* is True then the other segment must also be inline. Otherwise False.

Return type bool

extend(*amount*, *from_midpoint=False*)

Return a Line segment that is longer (or shorter) than this line by *amount* amount.

Parameters

- **amount** – The distance to extend the line. The line length will be increased by this amount. If *amount* is less than zero the length will be decreased.
- **from_midpoint** – Extend the line an equal amount on both ends relative to the midpoint. The amount length on both ends will be *amount*/2. Default is False.

Returns A new Line.

shift(*amount*)

Shift this segment forward or backwards by *amount*.

Parameters **amount** – The distance to shift the line. If *amount* is less than zero the segment will be shifted backwards.

Returns A copy of this Line shifted by the specified amount.

which_side(*p*, *inline=False*)

Determine which side of this line a point lies.

Parameters

- **p** – Point to test
- **inline** – If True return 0 if the point is inline. Default is False.

Returns 1 if the point lies to the left of this line else -1. If `inline` is True and the point is inline then 0.

which_side_angle(*angle*, *inline=False*)

Determine which side of this line lies a vector from the second end point with the specified direction angle.

Parameters

- **angle** – Angle in radians of the vector
- **inline** – If True return 0 if the point is inline.

Returns 1 if the vector direction is to the left of this line else -1. If `inline` is True and the point is inline then 0.

same_side(*pt1*, *pt2*)

Return True if the given points lie on the same side of this line.

point_on_line(*p*, *segment=False*)

Return True if the point lies on the line defined by this segment.

Parameters

- **p** (*geom.Point*) – the point to test
- **segment** – If True, the point must be collinear AND lie between the two endpoints. Default is False.

reversed()

Return a Line segment with start and end points reversed.

flipped()
Return a Line segment flipped 180deg around the first point.

to_svg_path()
Return a string with the SVG path 'd' attribute that corresponds with this line.

geom.arc

Basic 2D arc geometry.

Arc	Two dimensional immutable circular arc segment.

---

class geom.arc.**Arc**
Two dimensional immutable circular arc segment.

Parameters

- **p1** – Start point.
- **p2** – End point.
- **radius** – Arc radius.
- **angle** – Arc angle described by p1->center->p2.
- **center** – Optional center of arc. Will be computed if not specified. Default is None.

static from_two_points_and_center(*p1*, *p2*, *center*, *large_arc=False*)
Create an Arc given two end points and a center point.

Since this would be ambiguous, a hint must be given as to which way the arc goes.

Parameters

- **p1** – Start point.
- **p2** – End point.
- **center** – Center of arc.
- **large_arc** – If True the Arc will be on the large side (angle > pi). Default is False.

static from_two_points_and_tangent(*p1*, *ptan*, *p2*, *reverse=False*)
Create an Arc given two points and a tangent vector from p1->ptan.

Parameters

- **p1** – Start point.
- **ptan** – Tangent vector with origin at p1.
- **p2** – End point.
- **reverse** – Reverse the resulting arc direction if True. Default is False.

Returns An Arc or None if the arc parameters are degenerate (i.e. if the endpoints are coincident or the start point and tangent vector are coincident.)

static calc_center(*p1*, *p2*, *radius*, *angle*)
Calculate the center point of an arc given two endpoints, the radius, and a central angle. This method is static so that it can be used by __new__.

Parameters

- **p1** – Start point
- **p2** – End point
- **radius** – Radius of arc
- **angle** – The arc's central angle

Returns The center point as a tuple (x, y).

See: Thanks to Christian Blatter for this elegant solution: https://people.math.ethz.ch/~blatter/ http://math.stackexchange.com/questions/27535/how-to-find-center-of-an-arc-given-start-point-end-point-radius-and-arc-direc

p1
The start point of the arc.

p2
The end point of the arc.

radius
The radius of the arc.

angle
The central angle (AKA sweep angle) of this arc. The sign of the angle determines its direction.

center
The center point of this arc.

start_angle()
The angle from the arc center between the x axis and the first point.

length()

Returns The length of this arc segment.

area()

Returns The area inside the central angle between the arc and the center.

segment_area()

Returns The area of the shape limited by the arc and a straight line forming a chord between the two end points.

is_clockwise()

Returns True if arc direction is clockwise from first point to end point.

start_tangent_angle()

Returns The start direction of this arc segment in radians. This is the angle of a tangent vector at the arc segment's first point. Unlike a chord tangent angle this angle is from the x axis. Value is between -PI and PI.

end_tangent_angle()

Returns The end direction of this arc segment in radians. This is the angle of a tangent vector at the arc segment's end point. Value is between -PI and PI.

height()

Returns The distance between the chord midpoint and the arc midpoint. Essentially the Hausdorff distance between the chord and the arc.

transform(*matrix*)

Parameters

- **matrix** – An affine transform matrix. The arc will remain
- **circular.** –

Returns A copy of this arc with the transform matrix applied to it.

offset (*distance*, *preserve_center=True*)

Return a copy of this Arc that is offset by *distance*. If offset is < 0 the offset line will be towards the center otherwise to the other side of this arc. The central angle will be preserved.

Parameters

- **distance** – The distance to offset the line by.
- **preserve_center** – If True the offset arc will have the same center point as this one. Default is True.

Returns An Arc offset by *distance* from this one.

distance_to_point (*p*, *segment=True*)

Parameters

- **p** – The point to measure distance to
- **segment** – The point normal projection must lie on this the arc segment if True. Default is True.

Returns The minimum distance from this arc segment to the specified point, or -1 if *segment* is True and the point normal projection does not lie on this arc segment.

which_side_angle (*angle*, *inline=False*)

Determine which side of a line tangent to the end point of this arc lies a vector from the end point with the specified direction angle.

Parameters

- **angle** – Angle in radians of the vector
- **inline** – If True return 0 if the point is inline.

Returns 1 if the vector direction is to the left of arc tangent else -1. If `inline` is True and the point is inline then 0.

mu (*p*)

The unit distance from the first point of this arc segment to the specified point on the arc segment.

Parameters **p** – A point on this arc segment.

Returns The unit distance *mu* where *mu* >=0 and <= 1.0. If *p* is does not lie on this arc segment *mu* may be < 0 or > 1.

point_at (*mu*)

Parameters **mu** – Unit distance along central arc from first point.

Returns mu: along this arc from the start point.

Return type The point at unit distance

midpoint ()

Returns The point at the middle of the arc segment.

subdivide (*mu*)

Subdivide this arc at unit distance :mu: from the start point.

Parameters **mu** – Unit distance along central arc from first point.

Returns A tuple containing one or two Arc objects. If *mu* is zero or 1 then a tuple containing just this arc is returned.

subdivide_at_angle(*angle*)

Split this arc into two arcs at the point on this arc given by the specified positive arc angle (0-2pi) from the start point.

Parameters **angle** – A central angle the arc start point, in radians.

Returns A tuple containing one or two Arc objects. If the angle is zero or greater than this arc's angle then a tuple containing just this arc will be returned.

subdivide_at_point(*p*)

Split this arc into two arcs at the specified point.

Parameters **p** – A point on this arc.

Returns A tuple containing one or two Arc objects.

point_at_angle(*angle*, *segment=False*)

Get a point on this arc given an angle.

Parameters

- **angle** – A central angle from start point.
- **segment** – The point must lie on the arc segment if True. Default is False.

Returns The point on this arc given the specified angle from the start point of the arc segment. If `segment` is True and the point would lie outside the segment then None. Otherwise, if *angle* is negative return the first point, or if *angle* is greater than the central angle then return the end point.

point_on_arc(*p*)

Determine if a point lies on this arc.

Parameters **p** – Point to test.

Returns True if the point lies on this arc, otherwise False.

intersect_line(*line*, *on_arc=False*, *on_line=False*)

Find the intersection (if any) of this Arc and a Line.

See <http://mathworld.wolfram.com/Circle-LineIntersection.html>

Parameters

- **line** – A line defined by two points (as a 2-tuple of 2-tuples).
- **on_arc** – If True the intersection(s) must lie on the arc between the two end points. Default is False.
- **on_line** – If True the intersection(s) must lie on the line segment between its two end points. Default is False.

Returns A list containing zero, one, or two intersections as point (x, y) tuples.

intersect_arc(*arc*, *on_arc=False*)

The intersection (if any) of this Arc and another Arc.

See: <http://mathworld.wolfram.com/Circle-CircleIntersection.html>

Parameters

- **arc** – An Arc.
- **on_arc** – If True the intersection(s) must lie on both arc segments, otherwise the arcs are treated as circles for purposes of computing the intersections. Default is False.

Returns A list containing zero, one, or two intersections.

reversed()

Returns A copy of this Arc with direction reversed.

to_svg_path()

Returns A string with the SVG path 'd' attribute value that corresponds to this arc.

geom.ellipse

Two dimensional ellipse and elliptical arc.

class geom.ellipse.**Ellipse**(*center*, *rx*, *ry=None*, *phi=0.0*)

Two dimensional ellipse.

For the parametric function the parameter *t* is the parametric angle (aka eccentric anomaly) from the semi-major axis of the ellipse before stretch and rotation (i.e. as if this ellipse were a circle.)

The ellipse will be normalized so that the semi-major axis is aligned with the X axis (i.e. *rx* >= *ry*.) The ellipse rotation (phi) will be adjusted 90deg to compensate if necessary.

See: https://en.wikipedia.org/wiki/Ellipse http://www.spaceroots.org/documents/ellipse/ http://www.w3.org/TR/SVG11/implnote.html#ArcImplementationNotes https://en.wikipedia.org/wiki/Eccentric_anomaly

Parameters

- **center** – The center of the ellipse.
- **rx** – Semi-major axis length.
- **ry** – Semi-minor axis length. Default is *rx* if None.
- **phi** – Rotation angle of the ellipse.

is_circle()

theta2t(*theta*)

Compute parametric angle from geometric angle.

Parameters **theta** – The geometrical angle from the semi-major axis and a point on the ellipse.

Returns *t* - the parametric angle - 0 < *t* < 2*PI.

pointt(*p*)

Compute *t* given a point on the ellipse.

Parameters **p** – A point on the ellipse.

Returns *t* - the parametric angle - 0 < *t* < 2*PI.

point_at(*t*)

Return the point on the ellipse at *t*.

This is the parametric function for this ellipse.

Parameters **t** – Parametric angle - 0 < t < 2*PI.

Returns A point at *t*

point_inside(*p*)
Test if point is inside ellipse or not.

Parameters **p** – Point (x, y) to test.

Returns True if the point is inside the ellipse, otherwise False.

all_points_inside(*points*)
Return True if all the given points are inside this circle.

focus()
The focus of this ellipse.

Returns Distance from center to focus points.

focus_points()
Return the two focus points.

Returns A tuple of two focus points along major axis.

area()
The area of this ellipse.

eccentricity()
The eccentricity *e* of this ellipse.

curvature(*p*)
The curvature at a given point.

derivative(*t*, *d=1*)
First and second derivatives of the parametric ellipse function.

Parameters

- **t** – Parametric angle - 0 < t < 2*PI.
- **d** – 1 => First derivative, 2 => Second derivative. Default is 1.

Returns (dx, dy)

Return type A 2-tuple

class geom.ellipse.**EllipticalArc**(*center*, *p1*, *p2*, *rx*, *ry*, *start_angle*, *sweep_angle*, *large_arc*, *sweep_flag*, *phi=0.0*)
Two dimensional elliptical arc. A section of an ellipse.

See: http://www.w3.org/TR/SVG11/implnote.html#ArcImplementationNotes

Create an elliptical arc. If only center parameters or endpoint parameters are known the static factory methods can be used instead.

Parameters

- **p1** – The start point of the arc.
- **p2** – The end point of the arc.
- **rx** – Semi-major axis length.
- **ry** – Semi-minor axis length.
- **start_angle** – Parametric start angle of the arc.
- **sweep_angle** – Parametric sweep angle of the arc.
- **large_arc** – The large arc flag.
- **sweep_flag** – The sweep flag.

- **phi** – Rotation angle, in radians, of the ellipse. Default is 0.

static from_center(*center*, *rx*, *ry*, *start_angle*, *sweep_angle*, *phi=0.0*)
Create an elliptical arc from center parameters.

Parameters

- **center** – The center point of the arc.
- **rx** – Semi-major axis length.
- **ry** – Semi-minor axis length.
- **start_angle** – Start angle of the arc.
- **sweep_angle** – Sweep angle of the arc.
- **phi** – The angle from the X axis to the semi-major axis of the ellipse.

Returns An EllipticalArc

static from_endpoints(*p1*, *p2*, *rx*, *ry*, *large_arc*, *sweep_flag*, *phi=0.0*)
Create an elliptical arc from SVG-style endpoint parameters. This will correct out of range parameters as per SVG spec. The center, start angle, and sweep angle will also be calculated.

See: https://www.w3.org/TR/SVG11/implnote.html#ArcSyntax https://www.w3.org/TR/SVG11/implnote.html#ArcOutOfRangeParameters

Parameters

- **p1** – The start point of the arc.
- **p2** – The end poin of the arc.
- **rx** – Semi-major axis length.
- **ry** – Semi-minor axis length.
- **large_arc** – The large arc flag.
- **sweep_flag** – The sweep flag.
- **phi** – The angle in radians from the X axis to the semi-major axis of the ellipse. Default is 0.

Returns An EllipticalArc or None if *rx* or *ry* is 0.

transform(*matrix*)
Transform this using the specified affine transform matrix.

geom.ellipse.**ellipse_in_parallelogram**(*vertices*, *eccentricity=1.0*)
Inscribe a parallelogram with an ellipse.

See: Horwitz 2008, http://arxiv.org/abs/0808.0297

Vertices The four vertices of a parellelogram as a list of 2-tuples.

Eccentricity The eccentricity of the ellipse. Where 0.0 >= *eccentricity* <= 1.0. If *eccentricity* == 1.0 then a special eccentricity value will be calculated to produce an ellipse of maximal area. The minimum eccentricity of 0.0 will produce a circle.

Returns A tuple containing the semi-major and semi-minor axes respectively.

geom.ellipse.**intersect_circle**(*c1_center*, *c1_radius*, *c2_center*, *c2_radius*)
The intersection (if any) of two circles.

See: <http://mathworld.wolfram.com/Circle-CircleIntersection.html>

Parameters

- **c1_center** – Center of first circle.
- **c1_radius** – Radius of first circle.
- **c2_center** – Center of second circle.
- **c2_radius** – Radius of second circle.

Returns A tuple containing two intersection points if the circles intersect. A tuple containing a single point if the circles are only tangentially connected. An empty tuple if the circles do not intersect or if they are coincident (infinite intersections).

geom.bezier

Cubic bezier curve.

Includes biarc approximation.

class geom.bezier.**CubicBezier**

Two dimensional immutable cubic bezier curve.

For information about Bezier curves see: https://pomax.github.io/bezierinfo

Parameters

- **p1** – Start point as 2-tuple (x, y).
- **c1** – First control point as 2-tuple (x, y).
- **c2** – Second control point as 2-tuple (x, y).
- **p2** – End point as 2-tuple (x, y).

static from_quadratic(*qp1*, *qp2*, *qp3*)

Create a CubicBezier from the control points of a quadratic Bazier curve.

Parameters

- **qp1** – Start point as 2-tuple (x, y).
- **qp2** – Control point as 2-tuple (x, y).
- **qp3** – End point as 2-tuple (x, y).

p1

The start point of curve.

c1

The first control point of curve.

c2

The second control point of curve.

p2

The end point of curve.

transform(*matrix*)

Return a copy of this curve with the transform matrix applied to it.

start_tangent_angle()

Return the tangent direction of this curve in radians at the start (first) point. This would normally be the same as the angle of the first control point vector, unless the control point is coincident with the first point. -PI < angle < PI.

end_tangent_angle()
Return the tangent direction of this curve in radians at the end (second) point. -PI < angle < PI.

point_at(*t*)
A point on the curve corresponding to <t>.

This is the parametric function Bezier(t).

Returns A point as 2-tuple (x, y).

tangent(*t*)
The tangent unit vector at the point on the curve corresponding to *t*.

normal(*t*)
Normal unit vector at *t*.

flatness()
Return the flatness of this curve.

The maximum distance between the control points and the line segment defined by the start and end points of the curve. This is known as convex hull flatness and is robust regarding degenerate curves.

subdivide(*t*)
Subdivide this curve at the point corresponding to <t> into two cubic bezier curves, where 0<=t<=1. Uses De Casteljaus's algorithm.

Returns A tuple of one or two CubicBezier objects.

subdivide_inflections()
Subdivide this curve at the inflection points, if any.

Returns A list containing one to three curves depending on whether there are no inflections, one inflection, or two inflections.

find_inflections(*imaginary=False*)
Find (t1, t2) where the curve changes direction, has a cusp, or a loop. There may be none, one, or two inflections on the curve. A loop will have two inflections.

These inflection points can be used to subdivide the curve.

See http://www.caffeineowl.com/graphics/2d/vectorial/cubic-inflexion.html

Parameters **imaginary** – If True find *imaginary* inflection points. These are useful for subdividing curves with loops. Default is False.

Returns A tuple containing the parametric locations of the inflections, if any. The location values will be 0 if no inflection.

find_extrema_align(*calc_bbox=True*)
Find the extremities of the curve as if a chord connecting the end points is parallel to the X axis. This can be used to find the height of the curve if the curve has no inflections..

This also returns the bounding box since it needs to be rotated to match the curve alignment.

Parameters **calc_bbox** – Calculate an aligned bounding box. This can be performed slightly more efficiently here since the alignment rotation is known.

Returns A tuple where the first item is a list of zero to four points and the second is the bounding box (as a list of four points) or None if no extrema can be found.

find_extrema_points()
Find the extremities of this curve.

See: https://pomax.github.io/bezierinfo/#extremities

Returns A list of zero to four points.

find_extrema()

Find the extremities of this curve.

See: https://pomax.github.io/bezierinfo/#extremities

Returns A list of zero to four parametric (t) values.

controlpoints_at(*t*)

Get the point on this curve corresponding to *t* plus control points.

Useful for subdividing the curve at *t*.

Parameters **t** – location on curve. A value between 0.0 and 1.0

Returns A tuple of the form (C0, C1, P, C2, C3) where C1 and C2 are the control points tangent to P and C0 and C3 would be the new control points of the endpoints where this curve to be subdivided at P.

derivative1(*t*)

Calculate the 1st derivative of this curve at *t*.

Returns The first derivative at *t* as 2-tuple (dx, dy).

derivative2(*t*)

Calculate the 2nd derivative of this curve at *t*.

Returns The second derivative at *t* as 2-tuple (dx, dy).

derivative3()

Calculate the 3rd derivative of this curve.

Returns The third derivative as 2-tuple (dx, dy).

curvature_at(*t*)

Calculate the curvature at *t*.

See http://www.spaceroots.org/documents/ellipse/node6.html

Returns A scalar value *K* representing the curvature at *t*. Negative if curving to the right or positive if curving to the left when *t* increases.

length(*tolerance=None*)

Calculate the approximate arc length of this curve within the specified tolerance. The resulting computed arc length will be cached so that subsequent calls are not expensive.

Uses a simple and clever numerical algorithm described/invented by Jens Gravesen.

See: http://steve.hollasch.net/cgindex/curves/cbezarclen.html

Parameters **tolerance** – The approximation tolerance. Default is const.EPSILON.

Returns The approximate arc length of this curve.

biarc_approximation(*tolerance=0.001*, *max_depth=4*, *line_flatness=0.001*, *_recurs_depth=0*)

Approximate this curve using biarcs.

This will recursively subdivide the curve into a series of G1 (tangential continuity) connected arcs or lines until the Hausdorff distance between the approximation and this bezier curve is within the specified tolerance.

Parameters

- **tolerance** – Approximation tolerance. A lower value increases accuracy at the cost of time and number of generated biarc segments.
- **max_depth** – Maximum recursion depth. This limits how many times the Bezier curve can be subdivided.
- **line_flatness** – Segments flatter than this value will be converted to straight line segments instead of arcs with huge radii. Generally this should be a small value (say <= 0.01) to avoid path distortions.

Returns A list of Arc and/or Line objects. The list will be empty if the curve is degenerate (i.e. if the end points are coincident).

reversed()
Return a CubicBezier with control points (direction) reversed.

to_svg_path()
Return a string with the SVG path 'd' attribute value that corresponds with this curve.

geom.bezier.**bezier_circle**(*center=(0, 0)*, *radius=1.0*)
Create an approximation of a circle with a cubic Bezier curve.

Parameters

- **center** (*tuple*) – The center point of the circle. Default is (0,0).
- **radius** (*float*) – The radius of the circle. Default is 1.

Returns A tuple with four bezier curves for each circle quadrant. Circle will be counterclockwise from the positive x axis relative to the center point.

Return type tuple

See: https://pomax.github.io/bezierinfo/#circles_cubic

geom.bezier.**bezier_circular_arc**(*arc*)
Create a cubic Bezier approximation of a circular arc. The central arc must be less than PI/2 radians (90deg).

Parameters **arc** (geom.Arc) – A circular arc.

Returns A bezier curve.

Return type *CubicBezier*

geom.bezier.**bezier_ellipse**(*ellipse*)
Approximate this elliptical arc segment with Bezier curves.

If the sweep angle is greater than PI/2 the arc will be subdivided so that no segment has a sweep angle larger than PI/2.

Parameters **ellipse** – An Ellipse or EllipticalArc

Returns A list containing one to four BezierCurves.

geom.bezier.**bezier_elliptical_arc**(*ellipse*, *t1*, *t2*)
Compute a BezierCurve that can approximate the elliptical arc between *t1* and *t2*.

This does not subdivide the arc to reduce errors so if t2-t1 > PI/2 then the results may be less than ideal.

Parameters

- **ellipse** – An Ellipse
- **t1** – Parametric angle from semi-major axis to first location.
- **t2** – Parametric angle from semi-major axis to second location.

Returns A cubic bezier curve (as CubicBezier).

See: http://www.spaceroots.org/documents/ellipse/node22.html

`geom.bezier.`**`bezier_sine_wave`**(*amplitude*, *wavelength*, *cycles=1*, *origin=(0.0, 0.0)*)

Create an approximation of a sine wave using a cubic Bezier curve.

Parameters

- **amplitude** (*float*) – The amplitude (vertical scale) of the sine wave. This is one half the vertical distance from the trough to the peak.
- **wavelength** (*float*) – The horizontal length of one complete cycle.
- **cycles** (*int*) – The number of cycles. Default is one.
- **origin** (*tuple*) – Location of start point as a tuple (x,y). Default is (0, 0).

Returns A list of BezierCurve instances that describe the sine wave. Each curve will be one quarter of a sine wave, so one cycle will return a list four BezierCurves, two cycle will be eight, etc…

Return type list

`geom.bezier.`**`smoothing_curve`**(*seg1*, *seg2*, *cp1=None*, *smoothness=0.5*, *match_arcs=True*)

Create a smoothing Bezier curve between two segments that are not currently G1 continuous. The resulting Bezier curve will connect the two endpoints of the first segment.

Parameters

- **seg1** – First path segment containing first and second points. Can be a geom.Line or geom.Arc.
- **seg2** – Second path segment containing second and third points. Can be a geom.Line or geom.Arc.
- **cp1** (*tuple*) – First control point computed from previous invocation. If cp1 is None then the first endpoint of the first segment will be used as the initial control point. Default is None.
- **smoothness** (*float*) – Affects the magnitude of the smoothing curve control points. A value between 0 and 1. Default is 0.5
- **match_arcs** (*bool*) – Try to better match arc connections.

Returns A tuple containing CubicBezier and the control point for the next curve.

Return type tuple

See: Maxim Shemanarev http://www.antigrain.com/agg_research/bezier_interpolation.html http://hansmuller-flex.blogspot.com/2011/04/approximating-circular-arc-with-cubic.html

`geom.bezier.`**`smooth_path`**(*path*, *smoothness=0.5*)

Create a smooth approximation of the path using Bezier curves.

Parameters

- **path** (*list*) – A list of Line/Arc segments.
- **smoothness** (*float*) – Smoothness value (usually between 0 and 1). .5 is a reasonable default.

Returns A list of CubicBezier segments.

Return type list

geom.box

Basic 2D bounding box geometry.

class geom.box.**Box**

Two dimensional immutable rectangle defined by two points, the lower left corner and the upper right corner respectively.

The sides are always assumed to be aligned with the X and Y axes.

Useful as clipping rectangle or bounding box.

static from_points(*points*)

Create a Box from the bounding box of the given points.

Returns A geom.Box or None if there are zero points.

p1

The lower left corner of the box rectangle.

p2

The upper right corner of the box rectangle.

topleft

The upper left corner of the box recangle.

bottomright

The bottom right corner of the box recangle.

xmin

Minimum X value of bounding box.

xmax

Maximum X value of bounding box.

ymin

Minimum Y value of bounding box.

ymax

Maximum X value of bounding box.

vertices()

Get the four vertices of the box as a tuple of four points.

center()

Return the center point of this rectangle.

height()

Height of rectangle. (along Y axis)

width()

Width of rectangle. (along X axis)

diagonal()

Length of diagonal

point_inside(*p*)

Return True if the point is inside this rectangle.

line_inside(*line*)

Return True if the line segment is inside this rectangle.

all_points_inside(*points*)

Return True if the given set of points lie inside this rectangle.

buffered(*distance*)
Return a copy of this box with it's boundaries expanded or shrunk by the specified distance. Also known as buffering.

Parameters **distance** – The distance to offset. The box will shrink if the distance is negative.

transform(*matrix*)
Return a copy of this box with the transform matrix applied to it.

Note: rotations just scale since a Box is always aligned to the X and Y axes.

clip_line(*line*)
If the given line segment is clipped by this rectangle then return a new line segment with clipped end-points.

If the line segment is entirely within the rectangle this returns the same (unclipped) line segment.

If the line segment is entirely outside the rectangle this returns None.

Uses the Liang-Barsky line clipping algorithm. Translated C++ code from: http://hinjang.com/articles/04.html

Parameters **line** – The line segment to clip.

Returns A new clipped line segment or None if the segment is outside this clipping rectangle.

clip_arc(*arc*)
If the given circular arc is clipped by this rectangle then return a new arc with clipped end-points.

This only returns a single clipped arc even if the arc could be clipped into two sub-arcs... For now this is considered a pathological condition.

Parameters **arc** – The arc segment to clip.

Returns A new clipped arc or None if the arc segment is entirely outside this clipping rectangle. If the arc segment is entirely within the rectangle this returns the same (unclipped) arc segment.

start_tangent_angle()
Return the angle in radians of a line tangent to this shape beginning at the first point. It's pretty obvious this will always be PI/2...

This is just to provide an orthogonal interface for geometric shapes...

The corner point order for rectangles is clockwise from lower left.

bounding_box()
Bounding box - self.

intersection(*other*)
Return a Box that is the intersection of this rectangle and another.

Returns None if the rectangles do not intersect.

union(*other*)
Return a Box that is the union of this rectangle and another.

geom.polygon

Some handy polygon tools such as convex hull, area, and centroid calculations.

Some references: http://paulbourke.net/geometry/ http://geomalgorithms.com/index.html

`geom.polygon.turn`(*p*, *q*, *r*)
Returns -1, 0, 1 if p,q,r forms a right, straight, or left turn.

Parameters

- **p** – Point from which initial direction is determined. A 2-tuple (x, y) point.
- **q** – Point from which turn is determined. A 2-tuple (x, y) point.
- **r** – End point which determines turn direction. A 2-tuple (x, y) point.

`geom.polygon.convex_hull`(*points*)
Returns points on convex hull of an array of points in CCW order.

Uses the Graham Scan algorithm.

Parameters **points** – a list of 2-tuple (x, y) points.

Returns The convex hull as a list of 2-tuple (x, y) points.

`geom.polygon.convex_hull_chan`(*points*)
Returns the points on the convex hull of points in CCW order.

Uses Chan's algorithm. May be faster than Graham scan on large point collections.

See http://tomswitzer.net/2010/12/2d-convex-hulls-chans-algorithm/

Parameters **points** – a list of 2-tuple (x, y) points.

Returns The convex hull as a list of 2-tuple (x, y) points.

`geom.polygon.bounding_box`(*points*)
Simple bounding box of a collection of points.

Parameters **points** – an iterable collection of point 2-tuples (x,y).

`geom.polygon.area`(*vertices*)
Return the area of a simple polygon.

Parameters **vertices** – the polygon vertices. A list of 2-tuple (x, y) points.

Returns (float): The area of the polygon. The area will be negative if the vertices are ordered clockwise.

`geom.polygon.area_triangle`(*a*, *b=None*, *c=None*)
Area of a triangle.

This is just a slightly more efficient specialization of the more general polygon area.

Parameters

- **a** – The first vertex of a triangle or an iterable of three vertices.
- **b** – The second vertex or None if *a* is iterable.
- **c** – The third vertex or None if *a* is iterable.

Returns (float): The area of the triangle.

`geom.polygon.centroid`(*vertices*)
Return the centroid of a simple polygon.

See http://paulbourke.net/geometry/polygonmesh/

Parameters **vertices** – The polygon vertices. A list of 2-tuple (x, y) points.

Returns The centroid point as a 2-tuple (x, y)

`geom.polygon.`**`point_inside`**(*vertices*, *p*)

Return True if point *p* is inside the polygon defined by *vertices*.

See: http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html Also: http://paulbourke.net/geometry/polygonmesh/ Also: http://geomalgorithms.com/a03-_inclusion.html

Parameters

- **vertices** – polygon vertices. A list of 2-tuple (x, y) points.
- **p** – Point to test.

Returns True if the point lies inside the polygon, else False.

`geom.polygon.`**`intersect_line`**(*vertices*, *line*)

Compute the intersection(s) of a polygon and a line segment.

Parameters

- **vertices** – the polygon vertices. An iterable of 2-tuple (x, y) points.
- **line** – a line possibly intersecting the polygon. A 2-tuple of line end points, each a 2-tuple `((x1, y1), (x2, y2))`.

Returns a list of one or more line segments that intersect the polygon or that lie completely within the polygon. Returns an empty list if there are no intersections.

`geom.polygon.`**`is_closed`**(*vertices*)

Return True if the polygon is closed. I.e. if the first vertice matches the last vertice.

`geom.polygon.`**`offset_polygons`**(*poly*, *offset*, *jointype=0*, *limit=0.0*)

Offset a polygon by *offset* amount. This is also called polygon buffering.

See: http://www.angusj.com/delphi/clipper.php

Parameters

- **poly** – A polygon as a list of 2-tuple vertices.
- **offset** – The amount to offset (can be negative).
- **jointype** – The type of joins for offset vertices.
- **limit** – The max distance to a offset vertice before it will be squared off.

Returns Zero or more offset polygons as a list of 2-tuple vertices. If the specified offset cannot be performed for the input polygon an empty list will be retured.

`geom.polygon.`**`poly2clipper`**(*poly*)

Convert a polygon (as a list of float 2-tuple vertices) to a Clipper polygon (a list of integer 2-tuples).

`geom.polygon.`**`clipper2poly`**(*clipper_poly*)

Convert a Clipper polygon (a list of integer 2-tuples) to a polygon (as a list of float 2-tuple vertices).

`geom.polygon.`**`simplify_polyline_rdp`**(*points*, *tolerance*)

Simplify a polyline (a list of line segments given as a list of points).

Uses Ramer-Douglas-Peucker algorithm.

See: https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm

Parameters

- **points** (*list*) – A list of segment endpoints
- **tolerance** (*float*) – Line flatness tolerance

Returns A list of points defining the vertices of the simplified polyline.

`geom.polygon.`**`simplify_polyline_vw`**(*points*, *tolerance*)

Simplify a polyline (a list of line segments given as a list of points).

Uses Visvalingam-Whyatt algorithm.

See: Visvalingam, M., and Whyatt, J.D. (1992) "Line Generalisation by Repeated Elimination of Points", Cartographic J., 30 (1), 46 - 51

Parameters

- **points** (*list*) – A list of segment endpoints
- **tolerance** (*float*) – Line flatness tolerance

Returns A list of points defining the vertices of the simplified polyline.

geom.planargraph

Simple planar graph data structure.

class `geom.planargraph.`**`GraphNode`**(*vertex*, *edge_nodes=None*)

Graph node.

A node has a vertex and a list of outgoing nodes that define the outgoing edges that connect to other nodes in the graph.

Graph node.

Parameters **vertex** – The vertex associated with this node.

degree()

Number of incident graph edges. I.e. number of edges that share this node's vertex.

See: http://mathworld.wolfram.com/VertexOrder.html

add_edge_node(*edge_node*)

Add an outgoing edge node.

sort_edges()

Sort outgoing edges in CCW order.

ccw_edge_node(*ref_node*, *skip_spikes=True*)

The most CCW edge node from the reference edge defined by ref_node->this_node.

Parameters

- **ref_node** – The edge node reference.
- **skip_spikes** – Skip over edges that connect to nodes of order one.

Returns The counter-clockwise edge node closest to the reference node by angular distance. If all edges nodes are dead ends the reference node will be returned.

class `geom.planargraph.`**`Graph`**(*edges=None*)

Simple connected undirected 2D planar graph.

Parameters **edges** – An iterable collection of line segments that define the graph edges. Each edge connects two nodes. An edge being a 2-tuple of endpoints of the form: ((x1, y1), (x2, y2)).

edges = None

Set of graph edges

nodemap = None
Map of vertex points to graph nodes.

add_edge(*edge*)

Parameters **edge** – A line segment that defines a graph edge. An edge being a 2-tuple of endpoints of the form: ((x1, y1), (x2, y2)).

remove_edge(*edge*)
Remove and unlink the specified edge from the graph.

Parameters **edge** – A line segment that defines a graph edge connecting two nodes. An edge being a 2-tuple of endpoints of the form: ((x1, y1), (x2, y2)).

add_poly(*vertices*, *close_poly=True*)
Add edges from the line segments defined by the vertices of a polyline/polygon.

Parameters

- **vertices** – A list of polyline/polygon vertices as 2-tuples (x, y).
- **close_poly** – If True a closing segment will be automatically added if absent. Default is True.

order()
Number of graph nodes (vertices.)

size()
Number of edges.

vertices()
A collection view of node vertex points.

boundary_polygon()
A polygon defining the outer edges of this segment graph.

peel_boundary_polygon(*boundary_polygon*)
Similar to convex hull peeling but with non-convex boundary polygons.

Parameters **boundary_polygon** – The initial graph polygon hull to peel.

Returns A list of peeled inner polygons. Possibly empty.

cull_open_edges()
Remove edges that have one or two disconnected endpoints.

get_face_polygons()
Graph face polygons.

Returns A list of face polygons.

class geom.planargraph.**GraphPathBuilder**(*graph*)
Given a Graph, build a set of paths made of connected graph edges.

build_paths(*start_edge=None*, *path_strategy=0*)
Given the starting edge, find the set of edge paths that completely fill the graph…

Parameters

- **start_edge** – The graph edge that starts the path.
- **path_strategy** – How paths will be constructed. Possible path strategies are:

  PATH_STRAIGHTEST, PATH_SQUIGGLY, PATH_RANDOM, and PATH_RANDOM2

Returns A list of paths sorted by descending order of path length.

build_longest_paths(*path_strategy=0*)
Find the longest paths in this graph.

PATH_RANDOM = 2

PATH_RANDOM2 = 3

PATH_SQUIGGLY = 1

PATH_STRAIGHTEST = 0

class geom.planargraph.**MarkedEdge**(*edge*)
A graph edge that is used to keep track of graph traversal direction.

Parameters **edge** – A graph edge (a Line segment).

visited_p1 = None
True if traversed in direction p2->p1, CCW winding

visited_p2 = None
True if traversed in direction p1->p2, CCW winding

visited_left(*dest_vertex*)
True if this edge has been visited with a CCW winding. The edge will be marked as visited.

Parameters **dest_vertex** – The destination vertex. Determines which side of the edge has been visited (i.e. direction).

Returns True if this edge has been visited during a counter-clockwise traversal. I.e. the left side given the direction. Otherwise False.

class geom.planargraph.**MarkedEdgeMap**(*edges*)

lookup(*p1*, *p2*)

mark_edge(*p1*, *p2*)

geom.planargraph.**make_face_polygons**(*edges*, *nodemap*)
Given a graph, make polygons from graph faces delineated by edges.

Parameters **nodemap** – A mapping of edges to nodes.

geom.planargraph.**make_face**(*edgemap*, *start_node*, *next_node*)

geom.planargraph.**find_free_edge_node**(*edgemap*, *start_node*)

geom.voronoi

Voronoi diagram / Delaunay triangulation.

Compute a Voronoi diagram and optional Delaunay triangulation for a set of 2D input points.

Based on Steve Fortune's original code: http://ect.bell-labs.com/who/sjf/

Derek Bradley's fixes for memory leaks: http://zurich.disneyresearch.com/derekbradley/voronoi.html

Shane O'Sullivan's updates: http://mapviewer.skynet.ie/voronoi.html

Translated to Python by Bill Simons September, 2005: (not sure where this original translation can be found anymore)

Nicely refactored version by Manfred Moitzi at: https://bitbucket.org/mozman/geoalg

This version was based on the Bill Simons version and refactored with some of Moitzi's cleanups.

Derived from code bearing the following notice:

```
The author of this software is Steven Fortune. Copyright (c) 1994 by AT&T
Bell Laboratories.

Permission to use, copy, modify, and distribute this software for any
purpose without fee is hereby granted, provided that this entire notice
is included in all copies of any software which is or includes a copy
or modification of this software and in all copies of the supporting
documentation for such software.
THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
WARRANTY.  IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T MAKE ANY
REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

Comments were incorporated from Shane O'Sullivan's translation of the original code into C++:

> http://mapviewer.skynet.ie/voronoi.html

This module has no dependencies besides standard Python libraries.

`geom.voronoi.`**`EPSILON = 1e-09`**

Tolerance for floating point comparisons

class `geom.voronoi.`**`VoronoiEdge`**

A Voronoi edge. The dual of a corresponding This is a line segment that bisects a line between nearest neighbor sites.

If one end point of the edge is None it means the line extends to infinity. If the first end point is None the edge extends to the left. If the second end point is None the edge extends to the right.

`p1`

First point of edge segment.

`p2`

Second point of edge segment.

`equation`

The line equation for this segment in the form $a*x + b*y = c$ as a 3-tuple (a, b, c)

`delaunay_edge`

The dual of this Voronoi edge.

class `geom.voronoi.`**`DelaunayEdge`**

A Delaunay edge. The dual of a corresponding Voronoi segment that bisects this Delaunay segment. This is a line segment between nearest neighbor sites.

`p1`

First point of edge segment.

`p2`

Second point of edge segment.

class `geom.voronoi.`**`DelaunayTriangle`**

A Delaunay triangle. This a 3-tuple of 2-tuple (x, y) points.

`p1`

First point of triangle.

p2
Second point of triangle.

p3
Third point of triangle.

class geom.voronoi.**VoronoiDiagram**(*input_points*, *do_delaunay=False*, *jiggle_points=False*)
Voronoi diagram and Delaunay triangulation.

Parameters

- **input_points** – An indexable collection of points as (x, y) 2-tuples
- **do_delaunay** – True if Delaunay edges and triangles are to be generated. Default is False.
- **jiggle_points** – Jiggle the input points by a small random distance to mitigate problems caused by degenerate point sets (such as collinear or coincident points). Default is False.

vertices
List of the Voronoi diagram vertices as 2-tuple (x, y) coordinates.

edges
List of VoronoiEdges.

triangles
List of DelaunayTriangles.

delaunay_edges
List of DelaunayEdges.

geom.voronoi.**jiggle**(*point*)
Move a point in a random direction by a small random distance.

Useful for when input is degenerate (i.e. when points are collinear.)

Parameters **point** – The point as a 2-tuple of the form (x, y)

Returns A new jiggled point as a 2-tuple

geom.voronoiclip

geom.voronoiclip.**clip_voronoi_segments**(*self*, *diagram*, *clip_rect*)
Clip a voronoi diagram to a clipping rectangle.

Parameters

- **diagram** – A VoronoiDiagram.
- **A Box. Clipping rectangle.**(*clip_rect.*) –

Returns A list of (possibly) clipped voronoi segments.

geom.voronoiclip.**clip_voronoi_segments_poly**(*self*, *voronoi_segments*, *clip_polygon*)
Clip voronoi segments to a polygon.

Parameters **voronoi_segments** –

geom.voronoiclip.**clipped_delaunay_segments**(*self*, *voronoi_diagram*, *clip_polygon*)

geom.voronoiclip.**line_inside_hull**(*self*, *points*, *line*, *allow_hull=False*)
Test if line is inside or on the polygon defined by *points*.

This is a special case.... basically the line segment will lie on the hull, have one endpoint on the hull, or lie completely within the hull, or be completely outside the hull. It will not intersect. This works for the Delaunay triangles and polygon segments...

Parameters

- **points** – polygon vertices. A list of 2-tuple (x, y) points.
- **line** – line segment to test.
- **allow_hull** – allow line segment to lie on hull

Returns True if line is inside or on the polygon defined by *points*. Otherwise False.

geom.quasi

Python port of quasi.c which was originally written by Eric Weeks weeks@physics.emory.edu

See: http://www.physics.emory.edu/~weeks/software/quasic.html

This algorithm implements the "Generalized Dual Method" or GDM. See [Socolar, Steinhardt, Levine] 1985.

Mostly unchanged except to make it a little more pythonic and:

- Removed Postscript output and main()
- Fixed divide by zero exception for even symmetries
- Added segment connection to vertices options
- Removed coloring - should be done in plotter

---

class geom.quasi.**QuasiPlotter**

Quasi plotter base class. Subclass this to produce output.

Does nothing by default.

plot_polygon(*vertices*, *color*)

Draw a polygon.

Parameters

- **vertices** – A list of tuples containing the (x,y) coordinates of the polygon vertices.
- **color** – Fill color. A value between 0.0 and 1.0 or None if no fill.

Returns True if the polygon is not clipped, otherwise False.

plot_segment(*p1*, *p2*)

Draw a line segment.

Parameters

- **p1** – Segment start point as tuple containing (x,y) coordinates.
- **p2** – Segment end point as tuple containing (x,y) coordinates.

plot_segpoly(*vertices*, *color*)

Draw a line segment box.

Parameters

- **vertices** – A list of tuples containing the (x,y) coordinates of the polygon vertices.
- **color** – Fill color. A value between 0.0 and 1.0 or None if no fill.

class geom.quasi.**Quasi**(*symmetry=5*, *segtype_skinny=0*, *segtype_fat=0*, *plotter=None*, *tolerance=1e-08*)

Parameters

- **symmetry** – Degrees of symmetry. Must be at least two.
- **segtype_skinny** – Segment connection type for skinny rhombuses. Default is SEG_NONE.
- **segtype_fat** – Segment connection type for fat rhombuses. Default is SEG_NONE.
- **plotter** – Plotter to draw output. Default is None.
- **tolerance** – Tolerance for floating point comparisons. Default is 1e-08.

SEG_NONE = 0
No segment connection.

SEG_MIDP_ACUTE = 1
Connect midpoints of polygon edges that meet at an acute angle.

SEG_MIDP_OBTUSE = 2
Connect midpoints of polygon edges that meet at an obtuse angle.

SEG_MIDP_CROSS = 3
Connect midpoints of polygon edges to form a cross.

SEG_MIDP_RECT = 4
Connect midpoints of polygon edges to form a rectangle.

SEG_VERT_ACUTE = 5
Connect polygon vertices whose edges form an acute angle.

SEG_VERT_OBTUSE = 6
Connect polygon vertices whose edges form an obtuse angle.

SEG_VERT_CROSS = 7
Connect polygon vertices to form a cross.

tolerance = None
Tolerance for floating point comparisons

plotter = None
Plotter to draw output.

symmetry = None
Degrees of quasi symmetry.

segment_split_cross = None
Split crossed segments

offset_salt_x = None
Random-ish offset to avoid more than two lines intersecting.

offset_salt_y = None
Random-ish offset to avoid more than two lines intersecting.

segment_ratio = None
Ratio that determines edge midpoint.

skinnyfat_ratio = None
Ratio that determines whether a rhombus is fat or skinny

numlines = None
Number of lines. A larger number enables more tiles to be generated.

color_fill = None
Color fill polygons. Default is False.

color_by_polytype = None
Fill color by rhombus type.

quasi()
Draw tiling.

geom.fillet

Connect Line/Arc segments with a fillet arc.

geom.fillet.**fillet_path**(*path*, *radius*, *fillet_close=True*)
Attempt to insert a circular arc of the specified radius to connect adjacent path segments.

Parameters

- **path** – A list of Line or Arc segments.
- **radius** – The radius of the fillet arc.
- **fillet_close** – If True and the path is closed then add a terminating fillet. Default is False.

Returns A new path with fillet arcs. If no fillets are created then the original path will be returned.

geom.fillet.**fillet_polygon**(*poly*, *radius*, *fillet_close=True*)
Attempt to insert a circular arc of the specified radius connecting adjacent polygon segments.

Parameters

- **poly** – A list of polygon vertices.
- **radius** – The radius of the fillet arc.
- **fillet_close** – If True and the path is closed then add a terminating fillet. Default is True.

Returns A new path with fillet arcs as a list of Line and Arc segments. If no fillets are created then the original path will be returned.

geom.fillet.**insert_fillet**(*seg1*, *seg2*, *radius*)
Try to create a fillet between two segments. Any GCode rendering hints attached to the segments will be preserved.

Parameters

- **seg1** – First segment, an Arc or a Line.
- **seg2** – Second segment, an Arc or a Line.
- **radius** – Fillet radius.

Returns (seg1, fillet_arc, seg2) Returns an empty tuple if the segments cannot be connected with a fillet arc (either they are too small or somehow degenerate.)

Return type A tuple containing the adjusted segments and fillet arc

geom.fillet.**connect_fillet**(*seg1*, *farc*, *seg2*)
Connect two segments with a fillet arc. This will adjust the lengths of the segments to accommodate the fillet.

geom.fillet.**create_fillet_arc**(*seg1*, *seg2*, *radius*)
Try to create a fillet between two segments.

Parameters

- **seg1** – First segment, an Arc or a Line.
- **seg2** – Second segment, an Arc or a Line.
- **radius** – Fillet radius.

Returns A fillet arc or None if the segments cannot be connected with a fillet arc (either they are too small, already G1 continuous, or are somehow degenerate.)

`geom.fillet.`**`fillet_line_line`**(*line1*, *line2*, *fillet_radius*)

Create a fillet arc between two line segments.

Parameters

- **line1** – A Line.
- **line2** – A Line connected to line1.
- **fillet_radius** – The radius of the fillet.

Returns An Arc, or None if the fillet radius is too big to fit or if the two segments are not connected.

`geom.fillet.`**`fillet_arc_arc`**(*arc1*, *arc2*, *fillet_radius*)

Create a fillet arc between two connected arcs.

Parameters

- **arc1** – First arc.
- **arc2** – Second arc.
- **fillet_radius** – The radius of the fillet.

Returns An Arc, or None if the fillet radius is too big to fit or if the two segments are not connected.

`geom.fillet.`**`fillet_line_arc`**(*line*, *arc*, *fillet_radius*)

Create a fillet arc between a line segment and a connected arc. The fillet arc end point order will match the line-arc order.

Parameters

- **line** – A Line.
- **arc** – An Arc.
- **fillet_radius** – The radius of the fillet.

Returns An Arc, or None if the fillet radius is too big to fit or if the two segments are not connected.

geom.util

Basic 2D utility functions.

`geom.util.`**`normalize_angle`**(*angle*, *center=3.141592653589793*)

Normalize `angle` (in radians) about a 2*PI interval centered at `center`.

For angle between 0 and 2*PI (default): normalize_angle(angle, center=math.pi)

For angle between -PI and PI: normalize_angle(angle, center=0.0)

Parameters

- **angle** – Angle to normalize
- **angle** – float

- **center** – Center value about which to normalize. Default is math.pi.
- **center** – float

geom.util.**calc_rotation**(*start_angle*, *end_angle*)
Calculate the amount of rotation required to get from *start_angle* to *end_angle*.

Parameters

- **start_angle** – Start angle in radians.
- **end_angle** – End angle in radians.

Returns Rotation amount in radians where -PI <= rotation <= PI.

geom.util.**segments_are_g1**(*seg1*, *seg2*, *tolerance=None*)
Determine if two segments have G1 continuity (are tangentially connected.) G1 implies G0 continuity.

Parameters

- **seg1** – First segment. Can be geom.Line, geom.Arc, geom.CubicBezier.
- **seg2** – Second segment. Can be geom.Line, geom.Arc, geom.CubicBezier.
- **tolerance** – G0/G1 tolerance. Default is geom.const.EPSILON.

Returns True if the two segments have G1 continuity within the specified tolerance. Otherwise False.

geom.util.**reverse_path**(*path*)
Reverse in place the order and direction of path segments.

geom.debug

Debug output support for geometry package.

geom.debug.**svg_context**()
The SVG context used for debug output.

geom.debug.**set_svg_context**(*svg_context*)
Initialize this module with an SVGContext that can be used for debug output by draw...() methods.

geom.debug.**draw_point**(*point*, *radius=3*, *color=u'#000000'*, *parent=None*)
Draw a dot. Useful for debugging and testing.

geom.debug.**draw_line**(*line*, *color=u'#c00000'*, *width=u'1px'*, *opacity=1*, *verbose=False*, *parent=None*)
Draw an SVG line segment for debugging/testing

geom.debug.**draw_poly**(*vertices*, *color=u'#c00000'*, *width=u'1px'*, *verbose=False*, *parent=None*, *close_poly=True*)
Draw an SVG polygon.

geom.debug.**draw_arc**(*arc*, *color=u'#cccc99'*, *width=u'1px'*, *verbose=False*, *parent=None*)
Draw an SVG arc for debugging/testing

geom.debug.**draw_circle**(*center*, *radius*, *color=u'#cccc99'*, *width=u'1px'*, *verbose=False*, *parent=None*)
Draw an SVG circle.

geom.debug.**draw_ellipse**(*ellipse*, *color=u'#cccc99'*, *width=u'1px'*, *verbose=False*, *parent=None*)
Draw an SVG arc for debugging/testing

`geom.debug.draw_bezier`(*curve*, *color=u'#cccc99'*, *verbose=False*, *parent=None*)
Draw an SVG version of this curve for debugging/testing. Include control points, inflection points, and tangent lines.

3.1.5 `cam` - CAM/GCode package

`cam`

Tool path library.

`cam.gcode`

A G-code generator that is suitable for a four axis (or 3.5 axis) machine with X, Y, and Z axes along with an angular A axis that rotates about the Z axis. It is more general but that's the machine I have and the code might reflect that.

The generated G code is currently intended for a LinuxCNC interpreter, but probably works fine for others as well.

---

class `cam.gcode.PreviewPlotter`
Base interface that can be implemented by users of the GCode class to provide a graphical preview of the G-code output.

See `cam.gcodesvg` for an example of an SVG implemention.

plot_move(*endp*)
Plot G00 - rapid move from current tool location to to `endp`.

Parameters **endp** – Endpoint of move as a 4-tuple (x, y, z, a).

plot_feed(*endp*)
Plot G01 - linear feed from current tool location to `endp`.

Parameters **endp** – Endpoint of feed as a 4-tuple (x, y, z, a).

plot_arc(*center*, *endp*, *clockwise*)
Plot G02/G03 - arc feed from current tool location to to `endp`.

Parameters

- **center** – Center of arc as a 2-tuple (x, y)
- **endp** – Endpoint of feed as a 4-tuple (x, y, z, a).
- **clockwise** – True if the arc moves in a clockwise direction.

plot_tool_down()
Plot the beginning of a tool path.

plot_tool_up()
Plot the end of a tool path.

exception `cam.gcode.GCodeException`
Exception raised by gcode generator.

class `cam.gcode.GCodeGenerator`(*xyfeed*, *zsafe*, *zfeed=None*, *afeed=None*, *output=None*, *plotter=None*)
GCode generator class that describes a basic two axis (XY), three axis (XYZ), or four axis (XYZA) machine. The G code output is compatible with LinuxCNC.

Angles are always specified in radians but output as degrees.

Axis values are always specified in user/world coordinates and output as machine units (ie inches or millimeters) using `GCode.unit_scale` as the scaling factor.

Parameters

- **xyfeed** – Default feed rate along X and Y axes, in machine units per minute.
- **zsafe** – The safe height of the Z axis for rapid XY moves.
- **zfeed** – Feed rate along Z axis in machine units per minute. (Defaults to *xyfeed*.)
- **afeed** – Feed rate along A axis in degrees per minute. (Defaults to *xyfeed*.)
- **output** – Output stream for generated G code. Must implement `write()` method. Defaults to a StringIO if None (default).
- **plotter** – Preview plotter. Should be a subclass of `gcode.PreviewPlotter`.

TARGET = u'linuxcnc'
Current machine target

xyfeed = None
Feed rate along X and Y axes

zsafe = None
Z axis safe height for rapid moves

line_number = None
Current line number

unit_scale = None
User to machine unit scale

tolerance = None
Tolerance for float comparisons

angle_tolerance = None
Tolerance for angle comparisons

tool_wait_down = None
Delay time in millis for tool-down

tool_wait_up = None
Delay time in millis for tool-up

alt_tool_up = None
Alternate G code for Tool Up

alt_tool_down = None
Alternate G code for Too Down

spindle_wait_on = None
Default delay time in milliseconds after spindle is turned on.

spindle_wait_off = None
Default delay time in milliseconds after spindle is shut off.

spindle_clockwise = None
Spindle direction flag

spindle_speed = None
Default spindle speed

spindle_auto = None
Turn spindle on/off automatically on tool_up/tool_down

wrap_angles = None
Angles < 360 ?

show_comments = None
Show comments if True

show_line_numbers = None
Show line numbers if True

header_comments = None
Extra header comments

blend_mode = None
Blend mode. Can be None, 'blend', or 'exact'.

blend_tolerance = None
Blend tolerance. P value for G64 blend directive.

blend_qtolerance = None
Naive cam detector tolerance value. Q value for G64 blend directive.

verbose = None
Output code comments

zfeed = None
Z axis feed rate

afeed = None
Angular axis feed rate

output = None
The G code output stream

preview_plotter = None
The current preview plotter

units
GCode output units. Can be 'in' or 'mm'.

X
The current X axis value or none if unknown.

Y
The current Y axis value or none if unknown.

Z
The current Z axis value or none if unknown.

A
The current A axis value or none if unknown.

set_tolerance(*tolerance*, *angle_tolerance=None*)
Set tolerance (epsilon) for floating point comparisons.

Parameters

- **tolerance** – The tolerance for scalar floating point comparisons except angular values.
- **angle_tolerance** – The tolerance for comparing angle values. Set to `tolerance` if None (default).

set_output_precision(*precision*)
Set numeric output precision. This determines the number of digits after the decimal point.

This can be different from the precision implied by the *tolerance* value. The default is derived from the *tolerance* value.

Parameters **precision** – The number of digits after the decimal point.

set_units(*units*, *unit_scale=1.0*)
Set G code units and unit scale factor.

Note: Linear axis values are specified in user/world coordinates and output as machine units (ie inches or millimeters) using `unit_scale` as the scaling factor to scale from user/world units to G-code units.

Parameters

- **units** – Unit specifier. Must be *'in'* or *'mm'*.
- **unit_scale** – Scale factor to apply to linear axis values. Default is 1.0.

set_spindle_defaults(*speed*, *clockwise=True*, *wait_on=0*, *wait_off=0*, *auto=False*)
Set spindle parameter defaults.

Parameters

- **speed** – Spindle speed in RPM
- **clockwise** – Spindle direction. True if clockwise (default).
- **wait_on** – Number of milliseconds to wait for the spindle to reach full speed.
- **wait_off** – the number of milliseconds to wait for the spindle to stop.
- **auto** – Turn on/off spindle automatically on `tool_up()`/`tool_down()`. Default is False.

set_path_blending(*mode*, *tolerance=None*, *qtolerance=None*)
Set path trajectory blending mode and optional tolerance.

Parameters

- **mode** – Path blending mode. Can be 'exact' or 'blend'. Uses *G64* in 'blend' mode and *G61* in 'exact' mode.
- **tolerance** – Blending tolerance. Only used in 'blend' mode. This is the value for the G64 *P* parameter. Default is None.
- **qtolerance** – Naive cam detector tolerance value. This is the value for the G64 *Q* parameter. Default is None.

set_axis_offset(***kwargs*)
Set the offset for the specified axes.

Axis offsets are always specified in *machine units*. Angular offsets are always in *degrees*.

This is a 'soft' offset, not a G92 offset. The offset value will be added to the current axis value when a move is performed.

Example:

```
gcode_gen = gcode.GCodeGenerator(...)
gcode_gen.set_axis_offset(x=10, y=10)
```

Parameters

- **x** – X axis offset value (optional)
- **y** – Y axis offset value (optional)
- **z** – Z axis offset value (optional)
- **a** – A axis offset value (optional)

set_axis_scale(***kwargs*)

Set the scaling factors for the specified axes. The scaling is applied before the world/machine unit scaling.

Example:

```
gcode_gen = gcode.GCodeGenerator(...)
gcode_gen.set_axis_scale(x=10, y=10)
```

Parameters

- **x** – X axis scale value (optional)
- **y** – Y axis scale value (optional)
- **z** – Z axis scale value (optional)
- **a** – A axis scale value (optional)

map_axis(*canonical_name*, *output_name*)

Map canonical axis names to G code output names.

Mapping can be used to accommodate machines that expect different axis names (ie. using C instead of A or UVW instead of XYZ).

Parameters

- **canonical_name** – Canonical axis name. (ie 'X', 'Y', 'Z', or 'A')
- **output_name** – Output name. (ie 'U', 'V', 'W', or 'C')

add_header_comment(*comment*)

Append a comment to the header section.

Parameters **comment** – A comment or list of comments.

comment(*comment=None*, *use_semi=True*)

Write a G code comment line.

Outputs a newline if the comment string is None (default).

Parameters

- **comment** – A comment string or a list (or tuple) of comment strings. In the case of multiple comments, each one will be on a separate line.
- **use_semi** – Use the semicolon as a comment start character if True. Otherwise enclose the comment in parentheses.

header(*comment=None*)

Output a pretty standard G code file header.

Parameters **comment** – A header comment or a list of comments (optional).

footer()

Output a generic G code file footer.

feed_rate (*feed_rate*)
Set the specified feed rate. Outputs the *F* G code directive if the feed rate has changed since the last feed value.

Parameters **feed_rate** – The feed rate in machine units per minute.

pause (*conditional=False*, *comment=u'Pause'*)
Pause the G code interpreter.

Outputs *M1* or *M0* G code.

Note: Normally, pressing the start button in LinuxCNC/Axis will restart the interpreter after a pause.

Parameters

- **conditional** – use conditional stop if True.
- **comment** – Optional comment string.

dwell (*milliseconds*, *comment=None*)
Output a dwell command which pauses the tool for the specified number of milliseconds.

Parameters

- **milliseconds** – Number of milliseconds to pause.
- **comment** – Optional comment string.

tool_up (*rapid=True*, *wait=None*, *comment=None*)
Moves tool to a safe Z axis height. This should be called before performing a rapid move.

The spindle will also be automatically shut off if `Gcode.spindle_auto` is True.

Parameters

- **rapid** – Use G0 to move Z axis, otherwise G1 at current feed rate. Default is True.
- **wait** – the number of milliseconds to wait for the tool to retract. Uses GCode.tool_wait_up value by default if None specified. This parameter is mainly useful for pneumatically controlled up/down axes where the actuator may take a few milliseconds to extend/retract.
- **comment** – Optional comment string.

tool_down (*z*, *feed=None*, *wait=None*, *comment=None*)
Moves tool on Z axis down to specified height. Outputs a *G1* move command using the current feed rate for the Z axis.

The spindle will be automatically turned on first if `Gcode.spindle_auto` is True.

Parameters

- **z** – Height of Z axis to move to.
- **feed** – Feed rate (optional - default Z axis feed rate used if None.)
- **wait** – the number of milliseconds to wait for the tool to actually get to the specified depth. Uses *GCode.tool_wait_down* value by default if None specified. This parameter is mainly useful for pneumatically controlled up/down axes where the actuator may take a few milliseconds to extend/retract.
- **comment** – Optional comment string.

spindle_on(*speed=None*, *clockwise=None*, *wait=None*, *comment=None*)
Turn on the spindle.

Parameters

- **speed** – Spindle speed in RPM. If None use default speed.
- **clockwise** – Spindle turns clockwise if True. If None use default value.
- **wait** – Number of milliseconds to wait for the spindle to reach full speed. Uses `GCode.spindle_wait_on` value by default.
- **comment** – Optional comment string.

spindle_off(*wait=None*, *comment=None*)
Turn off the spindle.

Parameters

- **wait** – the number of milliseconds to wait for the spindle to stop. Uses `GCode.spindle_wait_off` value by default.
- **comment** – Optional comment string.

normalize_axis_angle(*axis=u'A'*)
Unwrap (normalize) a rotational axis. If the current angular position of the axis is > 360 this will reset the rotary axis origin so that 0 < angle < 360.

Useful when cutting large spirals with a tangent knife to minimize long unwinding moves between paths.

Parameters **axis** – Name of axis to unwrap. Default is 'A'.

rapid_move(*x=None*, *y=None*, *z=None*, *a=None*, *comment=None*)
Perform a rapid *G0* move to the specified location.

At least one axis should be specified. If the tool is below the safe 'Z' height it will be raised before the rapid move is performed.

Parameters

- **x** – X axis value (optional)
- **y** – Y axis value (optional)
- **z** – Z axis value (optional)
- **a** – A axis value (optional)
- **comment** – Optional comment string.

feed(*x=None*, *y=None*, *z=None*, *a=None*, *feed=None*, *comment=None*)
Perform a *G1* linear tool feed to the specified location.

At least one axis should be specified.

Parameters

- **x** – X axis value (optional)
- **y** – Y axis value (optional)
- **z** – Z axis value (optional)
- **a** – A axis value (optional)
- **feed** – Feed rate (optional - default feed rate used if None)
- **comment** – Optional comment string.

feed_arc(*clockwise*, *x*, *y*, *arc_x*, *arc_y*, *a=None*, *z=None*, *feed=None*, *comment=None*)
Perform a *G2*/*G3* arc feed.

This will raise a GCodeException if the beginning and ending arc radii do not match, ie if one of the end points does not lie on the arc.

Parameters

- **clockwise** – True if the arc moves in a clockwise direction.
- **x** – X value of arc end point
- **y** – Y value of arc end point
- **arc_x** – Center of arc relative to `x`
- **arc_y** – Center of arc relative to `y`
- **a** – A axis value at endpoint (in radians)
- **feed** – Feed rate (optional - default feed rate used if None)
- **comment** – Optional comment string.

get_current_position_xy()
The last known tool position on the XY plane.

Returns A 2-tuple containing coordinates of X and Y axes of the form (X, Y). An axis value will be None if the position is unknown.

get_current_position()
The last known tool position.

Returns A 4-tuple containing coordinates of all four axes of the form (X, Y, Z, A). An axis value will be None if the position is unknown.

position(*axis*)
The current position of the specified axis.

Parameters **axis** – The axis name - i.e. 'X', 'Y', 'Z', 'A', etc.

Returns The current position of the named axis as a float value.

gcode_command(*command*, ***kwargs*)
Output a line of gcode.

This is mainly for internal use and should be used with extreme caution. Use the higher level methods if at all possible.

Parameters

- **command** – G code command. Required.
- **params** – Parameter string that will be output as is. This *must not* be used with commands that that may change the position of the machine. Optional.
- **X** – The X axis value. Optional.
- **Y** – The Y axis value. Optional.
- **Z** – The Z axis value. Optional.
- **I** – Center (x) of arc relative to X,Y. Optional.
- **J** – Center (y) of arc relative to X,Y. Optional.
- **R** – Arc radius. Optional.

- **A** – The A axis value in radians. Optional.
- **force_value** – A string containing the modal parameter names whose values will be output regardless of whether their values have changed. By default if the specified value of a modal parameter has not changed since its last value then it will not be output.
- **comment** – Optional inline comment string.

Raises *GCodeException*

float_eq(*a*, *b*)
Compare two floats for approximate equality within the tolerance specified for the GCodeGenerator class.

cam.simplecam

Simple G code generation from basic 2D geometry.

exception cam.simplecam.**CAMException**

class cam.simplecam.**SimpleCAM**(*gcode_gen*)
Simple 2D CAM library that converts line/arc path geometry into G code suitable for a straightforward 2.5 axis machine with an optional fourth angular axis (A) that rotates about the Z axis. The fourth axis position is always tangential to the movement along the X and Y axes. This is usually called a tangential tool (ie a knife or a brush).

Since the path geometry is two dimensional the Z and A axes are calculated automatically. By default the Z axis value is determined by the current plunge depth and the A axis value is the tangent normal of the current segment.

These defaults can be overridden by assigning extra attributes to the segment.

Segment attributes:

- *inline_end_z*: The Z axis value at the end of the segment.
- *inline_start_a*: The A axis value at the start of the segment.
- *inline_end_a*: The A axis value at the start of the segment.
- ***inline_ignore_a*: Boolean. True if the A axis is not to be** rotated for the length of the segment.

Parameters **gcode_gen** – a *cam.gcode.GCodeGenerator* instance

generate_gcode(*path_list*)
Generate G code from tool paths.

Parameters **path_list** – A list of drawing paths. Where a drawing path is a sequential collection of bezier.CubicBezier, geom.Line, or geom.Arc segments. Other shape types will be silently ignored…

preprocess_paths(*path_list*)
Preprocess paths. Sort order will be maintained.

Parameters **path_list** – A list of drawing paths. Where a drawing path is a sequential collection of bezier.CubicBezier, geom.Line, or geom.Arc objects.

Returns A new list of tool paths.

path_biarc_approximation(*path*)
Convert all cubic bezier curves in the drawing path to biarcs (tangentially connected circular arcs).

Parameters

- **path** – A drawing path; an iterable collection of
- **geom.Line, or geom.Arc objects.** (bezier.CubicBezier,) –

Returns A new drawing path containing only geom.Line and geom.Arc objects.

Raises *CAMException* – If the path contains anything other than CubicBezier, Line, or Arc segments.

generate_header(*path_list*)
Output header boilerplate and comments.

plunge(*depth*, *path*)
Bring the tool down to the current working depth.

This can be subclassed to generate custom plunge profiles.

generate_rapid_move(*path*)
Generate G code for a rapid move to the beginning of the tool path.

sort_paths(*path_list*, *sort_method=u'optimize'*)
Sort the tool paths to minimize tool movements. This will try to sort the tool paths to minimize tool travel between the end of one path and the start of the next path.

Parameters

- **path_list** – A list of tool paths.
- **sort_method** – Sorting strategy.

Returns A sorted list of paths.

generate_segment_gcode(*segment*, *depth*)
Generate G code for Line and Arc path segments.

flip_tool()
Offset tangential tool rotation by 180deg. This useful for brush-type or double sided tools to even out wear.

cam.toolpath

exception cam.toolpath.**ToolpathException**

class cam.toolpath.**Toolpath**
A Toolpath is an ordered list of Line and Arc segments.

biarc_approximation(*path*, *tolerance*, *max_depth*, *line_flatness*)
Append the path while converting all cubic bezier curves to biarcs (tangentially connected circular arcs).

Parameters

- **path** – An iterable collection of bezier.CubicBezier, geom.Line, or geom.Arc objects.
- **tolerance** – Approximation tolerance. A lower value increases accuracy at the cost of time and number of generated biarc segments.
- **max_depth** – Maximum recursion depth. This limits how many times the Bezier curve can be subdivided.
- **line_flatness** – Segments flatter than this value will be converted to straight line segments instead of arcs with huge radii. Generally this should be a small value (say <= 0.01) to avoid path distortions.

Raises *ToolpathException* – If the path contains anything other than CubicBezier, Line, or Arc segments.

verify_continuity()
Verify that this path has point continuity (C0/G0).

is_closed()
Return True if this path forms a closed polygon.

cam.util

cam.util.**split_path_g1**(*path*)
Split the path at path vertices that connect non-tangential segments.

Parameters **path** – The path to split.

Returns A list of one or more paths.

cam.util.**inline_hint_attrs**(*segment*)
Generator to get hint attribute names.

cam.util.**copy_segment_attrs**(*seg1*, *seg2*)
Copy inline GCode rendering hints from seg1 to seg2.

cam.util.**seg_start_angle**(*segment*)
The tangent angle of this segment at the first end point. If there is a cam segment hint attribute ('inline_start_angle') its value will be returned instead.

cam.util.**seg_end_angle**(*segment*)
The tangent angle of this segment at the last end point. If there is a cam segment hint attribute ('inline_end_angle') its value will be returned instead.

cam.fillet

Connect Line/Arc segments with a fillet arc.

cam.fillet.**fillet_path**(*path*, *radius*, *fillet_close=True*, *adjust_rotation=False*, *mark_fillet=False*)
Attempt to insert a circular arc of the specified radius to blend adjacent path segments that have C0 or G0 continuity.

Parameters

- **path** – a list of geom.Line or geom.Arc segments.
- **fillet_close** – If True and the path is closed then add a terminating fillet. Default is False.
- **adjust_rotation** – If True adjust the A axis rotation hints to compensate for the offset caused by the fillet.
- **mark_fillets** – If True add an attribute to the fillet arc to mark it to ignore G1. Default is False.

Returns A new path with fillet arcs. If no fillets are created then the original path will be returned.

cam.fillet.**create_adjusted_fillet**(*seg1*, *seg2*, *radius*, *adjust_rotation=False*, *mark_fillet=False*)
Try to create a fillet between two segments. Any GCode rendering hints attached to the segments will be preserved.

Parameters

- **seg1** – First segment, an Arc or a Line.
- **seg2** – Second segment, an Arc or a Line.

- **radius** – Fillet radius.
- **adjust_rotation** – If True adjust the A axis rotation hints to compensate for the offset caused by the fillet.
- **mark_fillets** – If True add an attribute to the fillet arc to mark it to ignore G1. Default is False.

Returns A tuple containing the adjusted segments and fillet arc (seg1, fillet_arc, seg2), or an empty tuple if the segments cannot be connected with a fillet arc (either they are too small, already G1 continuous, or are somehow degenerate.)

cam.offset

Offset Line/Arc segments in a tool path to compensate for tool trail offset.

`cam.offset.`**`offset_path`**(*path*, *offset*, *min_arc_dist*, *g1_tolerance=None*)

Recalculate path to compensate for a trailing tangential offset. This will shift all of the segments by *offset* amount. Arcs will be recalculated to correct for the shift offset.

Parameters

- **path** – The path to recalculate.
- **offset** – The amount of tangential tool trail.
- **min_arc_dist** – The minimum distance between two connected segment end points that can be bridged with an arc. A line will be used if the distance is less than this.
- **g1_tolerance** – The angle tolerance to determine if two segments are g1 continuous.

Returns A new path

Raises `cam.toolpath.ToolpathException` – if the path contains segment types other than Line or Arc.

`cam.offset.`**`offset_arc`**(*arc*, *offset*)

Offset the arc by the specified offset.

`cam.offset.`**`fix_G1_path`**(*path*, *tolerance*, *line_flatness*)

`cam.offset.`**`smoothing_arcs`**(*seg1*, *seg2*, *cp1=None*, *tolerance=0.0001*, *line_flatness=0.0001*, *max_depth=1*, *match_arcs=True*)

Create circular smoothing biarcs between two segments that are not currently G1 continuous.

Parameters

- **seg1** – First path segment containing first and second points. Can be a geom.Line or geom.Arc.
- **seg2** – Second path segment containing second and third points. Can be a geom.Line or geom.Arc.
- **cp1** – Control point computed from previous invocation.
- **tolerance** – Biarc matching tolerance.
- **line_flatness** – Curve to line tolerance.
- **max_depth** – Max Bezier subdivision recursion depth.
- **match_arcs** – Attempt to more closely match existing arc segments. Default is True.

Returns A tuple containing a list of biarc segments and the control point for the next curve.

- modindex

3.2 Notes

While performance is important, maintainability and clarity has been more of a priority.

My memory is too unreliable to understand some opaque code I wrote two months ago, so I've tried to mitigate this somewhat with comments and avoiding magical python.

3.2.1 Why

You may be wondering why I didn't just use or extend **gcodetools**, which is currently packaged with Inkscape. Well, to be honest I found the author's code difficult to follow and it was very time consuming to customize for my needs. I had a hard time understanding the author's biarc curve approximation algorithm so I went to primary sources and wrote my own version that works pretty well (maybe even a bit better).

Like any programmer I've a mild case of NIH syndrome and I also just wanted code that I understood well and could modify easily and quickly for project-specific requirements.

Lots of people use gcodetools and it has many more features (such as pocketing and raster fills). I highly recommend checking it out.

3.2.2 Unit handling

Unit handling in SVG and Inkscape can be confusing.

The problem mainly stems from SVG's viewbox/viewport handling and Inkscape's concept of a 'document unit'. In addition, the SVG *viewbox* attribute can specify a box with a different aspect ratio than its parent - which is really only relevant in web browser contexts. Inkscape calls document units 'default units' in the UI, which further confuses things. Weirdly, the default document unit is pixels even when using a document template specified in inches or mm.

Anyway, for the purposes of using tcnc as an Inkscape extension, the simplest way to deal with this is in *File->Document Properties...*, set the 'default units' to the same value as the units used to specify the document size, and to use either inches or mm for both.

3.2.3 Inkscape extensions

If you are wondering how to build an Inkscape extension then it might be helpful to look at this source code. There are some reusable components as well that make writing extensions a little easier.

I suggest also taking a look at clearly written and well documented extensions such as the Eggbot extension. Documentation about Inkscape extensions is fairly poor and the best way to learn how to write one is to look at previous attempts.

3.2.4 Reusable packages

There are also some handy libraries, such as the SVG and geometry packages, that are more generally useful. I rewrote the Inkscape extension classes partly because Inkscape version .91 broke my extensions (mainly because of unit handling) and I just wanted to hoist more of the biolerplate involved with writing extensions.

I added a **docunits** option checker to convert UI values to current document units automatically. The creation of a debug layer and logging output file is done by the extension base class as well.

If you find any of this useful, great!

3.2.5 Etc.

I've been slowly converting doc strings to the Google python style since it's more readable than the standard reST style. There's currently a mix of the two styles… Sphinx handles both just fine.

Testing has been artisinal.

Most modules produce some pylint warnings. But IMO pylint is overly nit-picky about some things.

An attempt has been made to start migrating the code to full python3 compatibility, but this has not been tested. Inkscape still depends on Python 2.6+…

Emails, pull requests, feature requests, and issues are infrequently examined and may be left ignored for an uncomfortably long period of time… Sorry about that. Bug reports are welcome in any case.

- genindex

CHAPTER 4

Introduction

Tcnc is an Inkscape extension that generates G-code targeted for a four (or 3.5) axis CNC machine controlled by LinuxCNC v2.4+. The fourth axis (A) is angular and is assumed to rotate about a vertical Z axis. The tool path is calculated so that the A axis is kept tangent to movement along the X and Y axis. This is designed to move a tangential tool such as a brush, scraper, or knife centered along the path.

Tcnc will calculate tool paths to compensate for a trailing tool offset (such as a flexible brush whose contact with the surface trails behind the Z axis center) and can also perform automatic filleting to compensate for distortions caused by tool width. See *Trail offset* and *Tool width*.

There is an optional feature that will smooth the path by adding very small arc fillets at non-tangent path vertices, which can speed up feed rates. This is probably not necessary if LinuxCNC version 2.8+ is used since it has built-in toolpath blending.

Bezier curves are converted to circular arcs using a biarc approximation method. Compared to using line segment approximation this results in much smaller G code files (by orders of magnitude) and faster machine operation. Accuracy is very good.

Tcnc is currently used to produce G code for a painting apparatus based on a modified Fletcher 6100 CNC mat cutter controlled by LinuxCNC. A stepper controlled Z axis was added. The original pneumatic tool pusher was left on and is triggered by **spindle_on**. This allows for fast brush lifts along with very fine Z axis control. I haven't tested this with anything else so YMMV.

Tcnc does not perform tool path buffering to compensate for kerf created by cutting tools such as router bits, lasers, plasma cutters, or water jets. If kerf is not an issue or the user is willing to manually compensate for it by adjusting the input paths then this might work just fine for these applications.

Tcnc is an ongoing project that is mainly designed for my personal use and some of the features may seem weirdly specific.

There is absolutely no warranty for any purpose whatsoever. Use at your own risk.

Python Module Index

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z