polliwog

Nov 26, 2019

Contents

CHAPTER 1

polliwog package

1.1 Subpackages

1.1.1 polliwog.box package

Submodules

polliwog.box.box module

class polliwog.box.box.**Box**(*origin*, *size*)
Bases: object

An axis-aligned cuboid or rectangular prism. It's defined by an origin point, which is its minimum point in each dimension, and non-negative size (length, width, and depth).

Parameters

- **origin** (*np.arraylike*) – The *x*, *y*, and *z* coordinate of the origin, the minimum point in each dimension.
- **size** (*np.arraylike*) – An array containing the width (dx), height (dy), and depth (dz), which must be non-negative.

center_point
The box's geometric center.

depth
The box's depth. Same as *max_z* - *min_z*.

floor_point
The center of the side of the box having the minimum *y* coordinate. This is *center_point* projected to the the level of *min_y*.

classmethod from_points(*points*)
The smallest box which spans the given points.

Parameters **points** (*np.arraylike*) – A *kx3* array of points.

Returns The smallest box which spans the given points.

Return type *Box*

height
The box's height. Same as *max_y - min_y*.

max_x
The box's maximum *x* coordinate.

max_y
The box's maximum *y* coordinate.

max_z
The box's maximum *z* coordinate.

mid_x
The *x* coordinate of the box's center.

mid_y
The *y* coordinate of the box's center.

mid_z
The *z* coordinate of the box's center.

min_x
The box's minimum *x* coordinate.

min_y
The box's minimum *y* coordinate.

min_z
The box's minimum *z* coordinate.

ranges
Ranges for each coordinate axis as a 3x2 *np.ndarray*.

surface_area
The box's surface area.

volume
The box's volume.

width
The box's width. Same as *max_x - min_x*.

1.1.2 polliwog.line package

Submodules

polliwog.line.functions module

polliwog.line.functions.**project_to_line**(*points*, *reference_points_of_lines*, *vectors_along_lines*)

polliwog.line.line module

class polliwog.line.line.**Line**(*point*, *along*, *assume_normalized=False*)
Bases: object

classmethod from_points(*p1*, *p2*)

intersect_line(*other*)
Find the intersection with another line.

project(*points*)
Project a given point (or stack of points) to the plane.

reference_points
Return two reference points on the line.

polliwog.line.line_intersect module

polliwog.line.line_intersect.**line_intersect2**(*p0*, *q0*, *p1*, *q1*)
Intersect two lines: (p0, q0) and (p1, q1). Each should be a 2D point.

Adapted from http://stackoverflow.com/a/26416320/893113

polliwog.line.line_intersect.**line_intersect3**(*p0*, *q0*, *p1*, *q1*)
Intersect two lines in 3d: (p0, q0) and (p1, q1). Each should be a 3D point. See this for a diagram: http://math.stackexchange.com/questions/270767/find-intersection-of-two-3d-lines

1.1.3 polliwog.plane package

Submodules

polliwog.plane.coordinate_planes module

polliwog.plane.functions module

polliwog.plane.functions.**normal_and_offset_from_plane_equations**(*plane_equations*)
Given *A*, *B*, *C*, *D* of the plane equation *Ax + By + Cz + D = 0*, return the plane normal vector which is *[A, B, C]* and the offset *D*.

polliwog.plane.functions.**plane_equation_from_points**(*points*)
Given many sets of three points, return a stack of plane equations [*A*, *B*, *C*, *D*] which satisfy *Ax + By + Cz + D = 0*. Also works on three points to return a single plane equation.

These coefficients can be decomposed into the plane normal vector which is *[A, B, C]* and the offset *D*, either by the caller or by using *normal_and_offset_from_plane_equations()*.

polliwog.plane.functions.**plane_normal_from_points**(*points*, *normalize=True*)
Given a set of three points, compute the normal of the plane which passes through them. Also works on stacked inputs (i.e. many sets of three points).

This is the same as *polliwog.tri.functions.surface_normals*, to which this delegates.

polliwog.plane.functions.**project_point_to_plane**(*points*, *plane_equations*)
Project each point to the corresponding plane.

polliwog.plane.functions.**signed_distance_to_plane**(*points*, *plane_equations*)
Return the signed distances from each point to the corresponding plane.

For convenience, can also be called with a single point and a single plane.

polliwog.plane.intersections module

polliwog.plane.intersections.**intersect_segment_with_plane**(*start_points*, *segment_vectors*, *points_on_plane*, *plane_normals*)

polliwog.plane.plane module

class polliwog.plane.plane.**Plane**(*point_on_plane*, *unit_normal*)
Bases: object

A 2-D plane in 3-space (not a hyperplane).

Params:

- **point_on_plane, plane_normal:** 1 x 3 np.arrays

canonical_point
A canonical point on the plane, the one at which the normal would intersect the plane if drawn from the origin (0, 0, 0).

This is computed by projecting the reference point onto the normal.

This is useful for partitioning the space between two planes, as we do when searching for planar cross sections.

distance(*points*)

equation
Returns parameters A, B, C, D as a 1 x 4 np.array, where

Ax + By + Cz + D = 0

defines the plane.

params:

- **normalized:** Boolean, indicates whether or not the norm of the vector [A, B, C] is 1. Useful when computing the distance from a point to the plane.

classmethod fit_from_points(*points*)
Fits a plane whose normal is orthgonal to the first two principal axes of variation in the data and centered on the points' centroid.

flipped()
Creates a new Plane with an inverted orientation.

classmethod from_points(*p1*, *p2*, *p3*)
If the points are oriented in a counterclockwise direction, the plane's normal extends towards you.

classmethod from_points_and_vector(*p1*, *p2*, *vector*)
Compute a plane which contains two given points and the given vector. Its reference point will be p1.

For example, to find the vertical plane that passes through two landmarks:

from_points_and_normal(p1, p2, vector)

Another way to think about this: identify the plane to which your result plane should be perpendicular, and specify vector as its normal vector.

line_segment_xsection(*a*, *b*)

line_segment_xsections(*a*, *b*)

line_xsection(*pt*, *ray*)

line_xsections(*pts*, *rays*)

normal
Return the plane's normal vector.

points_in_front(*points*, *inverted=False*, *ret_indices=False*)
Given an array of points, return the points which lie either on the plane or in the half-space in front of it (i.e. in the direction of the plane normal).

Parameters

- **points** (*np.arraylikw*) – An array of points.
- **inverted** (*bool*) – When *True*, invert the logic. Return the points that lie behind the plane instead.
- **ret_indices** (*bool*) – When *True*, return the indices instead of the points themselves.

polyline_xsection(*polyline*, *ret_edge_indices=False*)
Deprecated.

project_point(*points*)
Project a given point (or stack of points) to the plane.

reference_point
The point used to create this plane.

sign(*points*)
Given an array of points, return an array with +1 for points in front of the plane (in the direction of the normal), -1 for points behind the plane (away from the normal), and 0 for points on the plane.

signed_distance(*points*)
Returns the signed distances to the given points or the signed distance to a single point.

Params:

- **points:** V x 3 np.array

1.1.4 polliwog.polyline package

Submodules

polliwog.polyline.cut_by_plane module

polliwog.polyline.cut_by_plane.**cut_open_polyline_by_plane**(*vertices*, *plane*)

polliwog.polyline.inflection_points module

polliwog.polyline.inflection_points.**inflection_points**(*points*, *axis*, *span*)

Find the list of vertices that preceed inflection points in a curve. The curve is differentiated with respect to the coordinate system defined by axis and span.

Interestingly, *lambda x: 2*x + 1* should have no inflection points, but almost every point on the line is detected. It's because a zero or zero crossing in the second derivative is necessary but not sufficient to detect an inflection point. You also need a higher derivative of odd order that's non-zero. But that gets ugly to detect reliably using sparse finite differences. Just know that if you've got a straight line this method will go a bit haywire.

axis: A vector representing the vertical axis of the coordinate system. span: A vector representing the the horiztonal axis of the coordinate system.

returns: a list of points in space corresponding to the vertices that immediately preceed inflection points in the curve

polliwog.polyline.polyline module

class polliwog.polyline.polyline.**Polyline**(*v*, *is_closed=False*)

Bases: object

Represent the geometry of a polygonal chain in 3-space. The chain may be open or closed, and there are no constraints on the geometry. For example, the chain may be simple or self-intersecting, and the points need not be unique.

Mutable by setting polyline.v or polyline.closed or calling a method like polyline.partition_by_length().

Note this class is distinct from lace.lines.Lines, which allows arbitrary edges and enables visualization. To convert to a Lines object, use the as_lines() method.

aligned_with(*vector*)

Flip the polyline if necessary, so it's aligned with the given vector rather than against it. Works on open polylines and considers only the two end vertices.

apex(*axis*)

Find the most extreme point in the direction of the axis provided.

axis: A vector, which is an 3x1 np.array.

bounding_box

The bounding box which encompasses the entire polyline.

copy()

Return a copy of this polyline.

e

Return a np.array of edges. Derived automatically from self.v and self.is_closed whenever those values are set.

flipped()

Flip the polyline from end to end. Return a new polyline.

index_of_vertex(*point*, *atol=1e-08*)

Return the index of the vertex with the given point. If there are coincident vertices at that point, return the first one.

intersect_plane(*plane*, *ret_edge_indices=False*)

Returns the points of intersection between the plane and any of the edges of *polyline*, which should be an instance of Polyline.

TODO: This doesn't correctly handle vertices which lie on the plane.

is_closed

classmethod join(**polylines*, *is_closed=False*)
Join together a stack of open polylines end-to-end into one contiguous polyline. The last vertex of the first polyline is connected to the first vertex of the second polyline, and so on.

nearest(*points*, *ret_segment_indices=False*)
For the given query point or points, return the nearest point on the polyline. With *ret_segment_indices=True*, also return the segment indices of those points.

num_e
Return the number of segments in the polyline.

num_v
Return the number of vertices in the polyline.

partition_by_length(*max_length*, *ret_indices=False*)
Subdivide each line segment longer than max_length with equal-length segments, such that none of the new segments are longer than max_length.

ret_indices: If True, return the indices of the original vertices. Otherwise return self for chaining.

rolled(*index*, *ret_edge_mapping=False*)
Return a new Polyline which reindexes the callee polyline, which much be closed, so the vertex with the given index becomes vertex 0.

ret_edge_mapping: if True, return an array that maps from old edge indices to new.

segment_lengths
The length of each of the segments.

segment_vectors
Vectors spanning each segment.

segments
Coordinate pairs for each segment.

sliced_at_indices(*start*, *stop*)
Take an slice of the given polyline starting at the *start* vertex index and ending just befeor reaching the *stop* vertex index. Always returns an open polyline.

When called on a closed polyline, the indies can wrap around the end.

sliced_at_points(*start_point*, *end_point*, *atol=1e-08*)
Take a slice of the given polyline at the given start and end points. These are expected to be on a vertex or on a segment. If on a segment (or near to but not directly on a segment) a new point is inserted at exactly the given point.

sliced_by_plane(*plane*)
Return a new Polyline which keeps only the part that is in front of the given plane.

For open polylines, the plane must intersect the polyline exactly once.

For closed polylines, the plane must intersect the polyline exactly twice, leaving a single contiguous segment in front.

to_dict(*decimals=3*)

total_length
The total length of all the segments.

v

with_insertions(*points*, *indices*, *ret_new_indices=False*)
Return a new polyline with the given points inserted before the given indices.

With *ret_new_indices=True*, also returns the new indices of the original vertices and the new indices of the inserted points.

with_segments_bisected(*segment_indices*, *ret_new_indices=False*)
Return a new polyline with the given segments cut in half.

With *ret_new_indices=True*, also returns the new indices of the original vertices and the new indices of the inserted points.

1.1.5 polliwog.segment package

Submodules

polliwog.segment.segment module

polliwog.segment.segment.**closest_point_of_line_segment**(*points*, *start_points*, *segment_vectors*)

polliwog.segment.segment.**partition**(*v*, *partition_size=5*)

params:

v: V x N np.array of points in N-space

partition_size: how many partitions intervals for each segment?

Fill in the line segments determined by v with equally spaced points - the space for each segment is determined by the length of the segment and the supplied partition size.

polliwog.segment.segment.**partition_segment**(*p1*, *p2*, *n_samples*, *endpoint=True*)
For two points in n-space, return an np.ndarray of equidistant partition points along the segment determined by p1 & p2.

The total number of points returned will be n_samples. When n_samples is 2, returns the original points.

When endpoint is True, p2 is the last point. When false, p2 is excluded.

Partition order is oriented from p1 to p2.

Parameters

- **p2** (*p1,*) – 1 x N vectors
- **partition_size** – size of partition. should be >= 2.

polliwog.segment.segment.**partition_segment_old**(*p1*, *p2*, *partition_size=5*)
Deprecated. Please use partition_segment.

For two points in n-space, return an np.ndarray of partition points at equal widths determined by 'partition_size' on the interior of the segment determined by p1 & p2.

Accomplished by partitioning the segment into 'partition_size' sub-intervals.

Partition order is oriented from p1 to p2.

Parameters

- **p2** (*p1,*) – 1 x N vectors
- **partition_size** – size of partition. should be > 1.

1.1.6 polliwog.transform package

Submodules

polliwog.transform.affine_transform module

polliwog.transform.affine_transform.**apply_affine_transform**(*points*, *transform_matrix*)

Apply the given transformation matrix to the points using homogenous coordinates.

(This works on any transformation matrix, whether or not it is affine.)

polliwog.transform.composite module

class polliwog.transform.composite.**CompositeTransform**

Bases: `object`

Composite transform using homogeneous coordinates.

Example

```
>>> transform = CompositeTransform()
>>> transform.scale(10)
>>> transform.reorient(up=[0, 1, 0], look=[-1, 0, 0])
>>> transform.translate([0, -2.5, 0])
>>> transformed_scan = transform(scan_v)
>>> # ... register the scan here ...
>>> untransformed_alignment = transform(alignment_v, reverse=True)
```

See also:

- *Computer Graphics: Principles and Practice*, Hughes, van Dam, McGuire, Sklar, Foley
- http://gamedev.stackexchange.com/questions/72044/why-do-we-use-4x4-matrices-to-transform-things-in-3d

__call__(*points*, *from_range=None*, *reverse=False*)

Parameters

- **points** (*np.arraylike*) – Points to transform, as a 3xn array.
- **from_range** (*tuple*) – The indices of the subset of the transformations to apply. e.g. *(0, 2)*, *(2, 4)*. When *None*, which is the default, apply them all.
- **reverse** (*bool*) – When *True* applies the selected transformations in reverse. This has no effect on how range is interpreted, only whether the selected transformations apply in the forward or reverse mode.

append_transform3(*forward*, *reverse=None*)

Append an arbitrary transformation, defined by 3x3 forward and reverse matrices.

The new transformation is added to the end. Return its index.

append_transform4(*forward*, *reverse=None*)

Append an arbitrary transformation, defined by 4x4 forward and reverse matrices.

The new transformation is added to the end. Return its index.

convert_units(*from_units*, *to_units*)
Convert the mesh from one set of units to another.

These calls are equivalent:

```
>>> composite.convert_units(from_units='cm', to_units='m')
>>> composite.scale(.01)
```

Supports the length units from Ounce: https://github.com/lace/ounce/blob/master/ounce/core.py#L26

reorient(*up*, *look*)
Reorient using up and look.

rotate(*rotation*)
Rotate by either an explicit matrix or a rodrigues vector

scale(*factor*)
Scale by the given factor.

Forward: [[s_0, 0, 0, 0], [0, s_1, 0, 0], [0, 0, s_2, 0], [0, 0, 0, 1]]

Reverse: [[1/s_0, 0, 0, 0], [0, 1/s_1, 0, 0], [0, 0, 1/s_2, 0], [0, 0, 0, 1]]

Parameters **factor** (*float*) – The scale factor.

transform_matrix_for(*from_range=None*, *reverse=False*)
Return a 4x4 transformation matrix representation.

range: The min and max indices of the subset of the transformations to apply. e.g. (0, 2), (2, 4). Inclusive of the min value, exclusive of the max value. The default is to apply them all.

reverse: When *True* returns a matrix for the inverse transform. This has no effect on how range is interpreted, only whether the forward or reverse matrices are used.

translate(*translation*)
Translate by the vector provided.

Forward:

[[1, 0, 0, v_0], [0, 1, 0, v_1], [0, 0, 1, v_2], [0, 0, 0, 1]]

Reverse:

[[1, 0, 0, -v_0], [0, 1, 0, -v_1], [0, 0, 1, -v_2], [0, 0, 0, 1]]

Parameters **vector** (*np.arraylike*) – A 3x1 vector.

polliwog.transform.coordinate_manager module

class polliwog.transform.coordinate_manager.**CoordinateManager**
Bases: object

Here's the idea:

coordinate_manager = CoordinateManager() coordinate_manager.tag_as('source') coordinate_manager.translate(-cube.floor_point) coordinate_manager.scale(2) coordinate_manager.tag_as('floored_and_scaled') coordinate_manager.translate(np.array([0., -4., 0.])) coordinate_manager.tag_as('centered_at_origin')

coordinate_manager.source = cube centered_mesh = coordinate_manager.centered_at_origin

__setattr__(*name*, *points*)
value: An nx3 array of points or an instance of Mesh.

append_transform3(**args*, ***kwargs*)

append_transform4(**args*, ***kwargs*)

convert_units(**args*, ***kwargs*)

do_transform(*points*, *from_tag*, *to_tag*)

reorient(**args*, ***kwargs*)

rotate(**args*, ***kwargs*)

scale(**args*, ***kwargs*)

tag_as(*name*)
Give a name to the current state.

translate(**args*, ***kwargs*)

polliwog.transform.rigid_transform module

polliwog.transform.rigid_transform.**find_rigid_rotation**(*a*, *b*, *allow_scaling=False*)

Parameters

- **a** – a 3xN array of vertex locations
- **b** – a 3xN array of vertex locations

Returns: R such that R.dot(a) ~= b

See link: http://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem

polliwog.transform.rigid_transform.**find_rigid_transform**(*a*, *b*, *visualize=False*)

Parameters

- **a** – a 3xN array of vertex locations
- **b** – a 3xN array of vertex locations

Returns: (R,T) such that R.dot(a)+T ~= b Based on Arun et al, "Least-squares fitting of two 3-D point sets," 1987. See also Eggert et al, "Estimating 3-D rigid body transformations: a comparison of four major algorithms," 1997.

polliwog.transform.rodrigues module

polliwog.transform.rodrigues.**as_rotation_matrix**(*r*)

polliwog.transform.rodrigues.**rodrigues**(*r*, *calculate_jacobian=False*)

polliwog.transform.rotation module

polliwog.transform.rotation.**estimate_normal**(*planar_points*)

polliwog.transform.rotation.**euler**(*xyz*, *order='xyz'*, *units='deg'*)

polliwog.transform.rotation.**rotation_from_up_and_look**(*up*, *look*)
Rotation matrix to rotate a mesh into a canonical reference frame. The result is a rotation matrix that will make up along +y and look along +z (i.e. facing towards a default opengl camera).

up: The direction you want to become $+y$. look: The direction you want to become $+z$.

1.1.7 polliwog.tri package

Submodules

polliwog.tri.functions module

polliwog.tri.functions.**barycentric_coordinates_of_points**(*vertices_of_tris*, *points*)
Compute barycentric coordinates for the projection of a set of points to a given set of triangles specfied by their vertices.

These barycentric coordinates can refer to points outside the triangle. This happens when one of the coordinates is negative. However they can't specify points outside the triangle's plane. (That requires tetrahedral coordinates.)

The returned coordinates supply a linear combination which, applied to the vertices, returns the projection of the original point the plane of the triangle.

Parameters

- **vertices_of_tris** (*np.arraylike*) – A set of triangle vertices as *kx3x3*.
- **points** (*np.arraylike*) – Coordinates of points as *kx3*.

Returns Barycentric coordinates as *kx3*

Return type np.ndarray

See also:

- https://en.wikipedia.org/wiki/Barycentric_coordinate_system
- Heidrich, "Computing the Barycentric Coordinates of a Projected Point," JGT 05 (http://www.cs.ubc.ca/~heidrich/Papers/JGT.05.pdf)

polliwog.tri.functions.**contains_coplanar_point**(*a*, *b*, *c*, *point*, *atol=1e-08*)
Assuming *point* is coplanar with the triangle *ABC*, check if it lies inside it.

polliwog.tri.functions.**coplanar_points_are_on_same_side_of_line**(*a*, *b*, *p1*, *p2*, *atol=1e-08*)

polliwog.tri.functions.**surface_normals**(*points*, *normalize=True*)
Compute the surface normal of a triangle. The direction of the normal follows conventional counter-clockwise winding and the right-hand rule.

Also works on stacked inputs (i.e. many sets of three points).

polliwog.tri.quad_faces module

polliwog.tri.quad_faces.**quads_to_tris**(*quads*, *ret_mapping=False*)
Convert quad faces to triangular faces.

quads: An nx4 array. ret_mapping: A bool.

When *ret_mapping* is *True*, return a 2nx3 array of new triangles and a 2nx3 array mapping old quad indices to new trangle indices.

When *ret_mapping* is *False*, return the 2nx3 array of triangles.

polliwog.tri.shapes module

polliwog.tri.shapes.**create_cube**(*origin*, *size*, *ret_unique_vertices_and_faces=False*)

Return vertices (or unique verties and faces) with an axis-aligned cube. One vertex is *origin*; the diametrically opposite vertex is *size* units along +x, +y, and +z.

size: int or float.

polliwog.tri.shapes.**create_horizontal_plane**(*ret_unique_vertices_and_faces=False*)

Creates a horizontal plane.

polliwog.tri.shapes.**create_rectangular_prism**(*origin*, *size*, *ret_unique_vertices_and_faces=False*)

Return vertices (or unique verties and faces) of an axis-aligned rectangular prism. One vertex is *origin*; the diametrically opposite vertex is *origin* + *size*.

size: 3x1 array.

polliwog.tri.shapes.**create_triangular_prism**(*p1*, *p2*, *p3*, *height*, *ret_unique_vertices_and_faces=False*)

Return vertices (or unique verties and faces) of a triangular prism whose base is the triangle p1, p2, p3. If the vertices are oriented in a counterclockwise direction, the prism extends from behind them.

Imported from lace.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

H

I

J

L

M

N

P

Q

R

S

T

V

W