polliwog

Oct 06, 2020

Contents

CHAPTER 1

Polygonal chains

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

Represent the geometry of a polygonal chain in 3-space. The chain may be open or closed.

There are no constraints on the geometry. For example, the chain may be simple or self-intersecting, and the points need not be unique.

The methods do not mutate; they create new polylines which exhibit the requested mutation. However, immutability is not enforced. If you wish you can mutate a polyline by updating *polyline.v* or *polyline.is_closed*.

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

an array containing a pair of vertex indices for each edge. This is derived automatically from *self.v* and *self.is_closed* whenever those values are set.

Type Return the edges of the polyline

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 one at the lowest index.

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.

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.

path_centroid
The weighted average of all the points along the edges of the polyline.

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.

sectioned(*section_breakpoints*, *copy_vs=False*)
Section the given open polyline at the given breakpoints, which indicate where one segment ends and the next one starts. Each of the breakpoint vertices is included as an endpoint in one section and a start point in the next section.

Parameters

- **breakpoints** (*np.arraylike*) – The indices of the breakpoints.
- **copy_vs** (*bool*) – When *True*, copy the vertices into the new polylines. When *False*, return polylines with views for vertex arrays.

Returns A list of the sectioned polylines.

Return type list

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.

subdivided_by_length(*max_length*, *edges_to_subdivide=None*, *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*. Returns a new Polyline.

Parameters

- **max_length** (*float*) – The maximum lenth of a segment.
- **edges_to_subdivide** (*np.arraylike*) – An optional boolean mask the same length as the number of edges. Only the edges marked *True* are subdivided. The default is to subdivide all edges longer than *max_length*.
- **ret_indices** (*bool*) – When *True*, also returns the indices of the original vertices.

total_length
The total length of all the segments.

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 Polygonal chain functions

polliwog.polyline.**inflection_points**(*points*, *rise_axis*, *run_axis*)
Find the list of vertices that preceed inflection points in a curve. The curve is differentiated with respect to the coordinate system defined by *rise_axis* and *run_axis*.

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.

rise_axis: A vector representing the vertical axis of the coordinate system. run_axis: 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.**point_of_max_acceleration**(*points*, *rise_axis*, *run_axis*, *subdivide_by_length=None*)
Find the point on a curve where the curve is maximally accelerating in the direction specified by *rise_axis*. *run_axis* is the horizontal axis along which slices are taken.

Parameters

- **points** (*np.arraylike*) – A stack of points, as *kx3*. For best results, trim these to the area of interest before calling.
- **rise_axis** (*np.arraylike*) – The vertical axis, as a 3D vector.
- **run_axis** (*np.arraylike*) – The horizonal axis, as a 3D vector.
- **subdivide_by_length** (*float*) – When provided, the maximum space between each point. The idea is keep the slice width small, however this constraint is applied in 3D space, not along the *run_axis*. For best results pass a value that is small relative to the changes in the geometry. When *None*, the points are used without modification.

CHAPTER 2

Planes

class polliwog.**Plane**(*point_on_plane*, *unit_normal*)

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

Parameters

- **point_on_plane** (*np.arraylike*) – A reference point on the plane, as a NumPy array with three coordinates.
- **unit_normal** (*np.arraylike*) – The plane normal vector, as a NumPy array with three coordinates.

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.

equation

Returns parameters *A*, *B*, *C*, *D* as a 1x4 *np.array*, where

$$Ax + By + Cz + D = 0$$

defines 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 their 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.

mirror_point (*points*)
Mirror a point (or stack of points) to the opposite side of the plane.

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 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*, return the points which lie on or behind the plane instead.
- **ret_indices** (*bool*) – When *True*, return the indices instead of the points themselves.

Note: Use *points_on_or_in_front()* for points which lie either on the plane or in front of it.

---

points_on_or_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*, return the points behind the plane instead.
- **ret_indices** (*bool*) – When *True*, return the indices instead of the points themselves.

Note: Use *points_in_front()* to get points which lie only in front of the plane.

---

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.

Parameters **points** (*np.arraylike*) – A 3D point or a *kx3* stack of points.

Returns

- Given a single 3D point, the distance as a NumPy scalar.
- Given a *kx3* stack of points, an *k* array of distances.

Return type depends

tilted(*new_point*, *coplanar_point*)
Create a new plane, tilted so it passes through *new_point*. Also specify a *coplanar_point* which the old and new planes should have in common.

Parameters

- **new_point** (*np.arraylike*) – A point on the desired plane, with shape *(3,)*.
- **coplanar_point** (*np.arraylike*) – The *(3,)* point which the old and new planes have in common.

Returns The adjusted plane.

Return type *Plane*

2.1 Named coordinate planes

polliwog.Plane.**xy = <Plane of [0. 0. 1.] through [0. 0. 0.]>**
The *xy*-plane.

polliwog.Plane.**xz = <Plane of [0. 1. 0.] through [0. 0. 0.]>**
The *xz*-plane.

polliwog.Plane.**yz = <Plane of [1. 0. 0.] through [0. 0. 0.]>**
The *yz*-plane.

2.2 Plane functions

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

polliwog.plane.**mirror_point_across_plane**(*points*, *plane_equations*)
Mirror each point to the corresponding point on the opposite side of the plane.

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

Check for intersections between a line segment and a plane, or pairwise between a stack of line segments and a stack of planes.

CHAPTER 3

Triangles

polliwog.tri.**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.**tri_contains_coplanar_point**(*a*, *b*, *c*, *point*)

Assuming *point* is coplanar with the triangle *ABC*, check if it lies inside it.

polliwog.tri.**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.**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.

CHAPTER 4

Geometric transformations

4.1 High-level API

class polliwog.**CompositeTransform**

Composite transform using homogeneous coordinates.

Example

```
>>> transform = CompositeTransform()
>>> transform.uniform_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*, *discard_z_coord=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_transform (*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.uniform_scale(.01)
```

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

flip (*dim*)
Flip about one of the axes.

Parameters **dim** (*int*) – The axis to flip about: 0 for *x*, 1 for *y*, 2 for *z*.

non_uniform_scale (*x_factor*, *y_factor*, *z_factor*, *allow_flipping=False*)
Scale by the given factors along *x*, *y*, and *z*.

Parameters

- **x_factor** (*float*) – The scale factor to be applied along the *x* axis.
- **y_factor** (*float*) – The scale factor to be applied along the *y* axis.
- **z_factor** (*float*) – The scale factor to be applied along the *z* axis.

See also:

uniform_scale()

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

rotate (*rotation*)
Rotate by the given 3x3 rotation matrix or a Rodrigues vector.

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.

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

uniform_scale (*factor*, *allow_flipping=False*)
Scale by the given factor.

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

See also:

non_uniform_scale()

class polliwog.**CoordinateManager**

Example

```
>>> coordinate_manager = CoordinateManager()
>>> coordinate_manager.tag_as('source')
>>> coordinate_manager.translate(-cube.floor_point)
>>> coordinate_manager.uniform_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.

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

4.2 Transform functions

polliwog.transform.**apply_transform**(*transform*)
Wrap the given transformation matrix with a function which conveniently can be invoked with either points or a single point, returning the same. It applies the transformation to those points using homogeneous coordinates.

Parameters **points** (*np.ndarray*) – The point *(3,)* or points *kx3* to transform.

Returns A function which accepts an *np.ndarray* containing a point *(3,)* or points *kx3* to transform, and returns an *ndarray* of the same shape. Also accepts two kwargs. The first is *discard_z_coord*. When *True*, discard the z coordinate of the result. This is useful when applying viewport transformations. The second is *treat_input_as_vectors* which does not use the homogeneous coordinate, and therefore ignores translation.

Return type func

polliwog.transform.**euler**(*xyz*, *order='xyz'*, *units='deg'*)
Convert a Euler angle representation of 3D rotations to a 3x3 rotation matrix.

Euler angles are a way of representing 3D rotations as a sequence of rotations about the axes. Conceptually, think of *euler([10, 20, 30])* as "Rotate 10 degrees around the x axis, then 20 degrees around the y axis, then 30 degrees around the z axis" (that ordering can be changed with the *order* argument, and the units can be given in degrees or radians by setting *units* to *'deg'* or *'rad'*).

Euler angles are a problematic representation of rotation for numerical methods, as there are multiple possible representations for a given rotation. But they are a very intuitive and readable way to initialize a rotation matrix.

See also:

- https://en.wikipedia.org/wiki/Euler_angles

polliwog.transform.**rodrigues_vector_to_rotation_matrix**(*r*, *calculate_jacobian=False*)
Convert a 3x1 or 1x3 Rodrigues vector to a 3x3 rotation matrix.

A Rodrigues vector is a 3 element vector representing a 3D rotation. Its direction represents the axis about which to rotate and its magnitude represents the amount to rotate by.

All of SO3 (that is, all 3D rotations) can be uniquely represented by a Rodrigues vector, and it does not suffer from the multiple representation and gimbal locking problems that Euler angle representations do.

If *calculate_jacobian* is passed, then the derivative of the rotation is also computed. Note that the derivative is undefined for a Rodrigues vector of *[0,0,0]* (that is, no rotation).

See also:

- https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula

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

Convert a 3x3 rotation matrix to a 3x1 or 1x3 Rodrigues vector.

A Rodrigues vector is a 3 element vector representing a 3D rotation. Its direction represents the axis about which to rotate and its magnitude represents the amount to rotate by.

All of SO3 (that is, all 3D rotations) can be uniquely represented by a Rodrigues vector, and it does not suffer from the multiple representation and gimbal locking problems that Euler angle representations do.

If *calculate_jacobian* is passed, then the derivative of the rotation is also computed. Note that the derivative is undefined for a Rodrigues vector of *[0,0,0]* (that is, no rotation).

See also:

- https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula

polliwog.transform.**cv2_rodrigues**(*r*, *calculate_jacobian=False*)

cv2_rodrigues is a wrapped function designed to be API compatible with OpenCV's *cv2.Rodrigues*.

If it is given a rotation matrix, it returns a Rodrigues vector.

If it is given a Rodrigues vector, it returns a rotation matrix.

To make your code clearer, call *rodrigues_vector_to_rotation_matrix* or *rotation_matrix_to_rodrigues_vector* directly, which makes the intent of your code clearer.

polliwog.transform.**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 $+z$.

polliwog.transform.**world_to_view**(*position*, *target*, *up=array([0., 1., 0.])*, *inverse=False*)

Create a transform matrix which sends world-space coordinates to view-space coordinates.

Parameters

- **position** (*np.ndarray*) – The camera's position in world coordinates.
- **target** (*np.ndarray*) – The camera's target in world coordinates. *target - position* is the "look at" vector.
- **up** (*np.ndarray*) – The approximate up direction, in world coordinates.
- **inverse** (*bool*) – When *True*, return the inverse transform instead.

Returns The *4x4* transformation matrix, which can be used with *polliwog.transform.apply_transform()*.

Return type np.ndarray

See also:

https://cseweb.ucsd.edu/classes/wi18/cse167-a/lec4.pdf http://www.songho.ca/opengl/gl_camera.html

polliwog.transform.**view_to_orthographic_projection**(*width*, *height*, *near=0.1*, *far=2000*, *inverse=False*)

Create an orthographic projection matrix with the given parameters, which maps points from world space to coordinates in the normalized view volume. These coordinates range from -1 to 1 in x, y, and z with *(-1, -1, -1)* at the bottom-left of the near clipping plane, and *(1, 1, 1)* at the top-right of the far clipping plane.

Parameters

- **width** (*float*) – Width of the window, in pixels. (FIXME: Is this really correct?)
- **height** (*float*) – Height of the window, in pixels. (FIXME: Is this really correct?)
- **near** (*float*) – Near clipping plane. (FIXME: Clarify!)
- **far** (*float*) – Far clipping plane. (FIXME: Clarify!)
- **inverse** (*bool*) – When *True*, return the inverse transform instead.

Returns The *4x4* transformation matrix, which can be used with *polliwog.transform.apply_transform()*.

Return type np.ndarray

See also:

https://cseweb.ucsd.edu/classes/wi18/cse167-a/lec4.pdf http://www.songho.ca/opengl/gl_projectionmatrix.html http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/projection/orthographicprojection.html

polliwog.transform.**viewport_transform**(*x_right*, *y_bottom*, *x_left=0*, *y_top=0*, *inverse=False*)

Create a matrix which transforms from the normalized view volume to screen coordinates, with a depth value ranging from 0 in front to 1 in back.

No clipping is performed.

Parameters

- **x_right** (*int*) – The *x* coordinate of the right of the viewport. (usually the width).
- **y_bottom** (*int*) – The *y* coordinate of the bottom of the viewport (usually the height).
- **x_left** (*int*) – The *x* coordinate of the left of the viewport (usually zero).
- **y_top** (*int*) – The *y* coordinate of the top of the viewport (usually zero).
- **inverse** (*bool*) – When *True*, return the inverse transform instead.

Returns The *4x4* transformation matrix, which can be used with *polliwog.transform.apply_transform()*.

Return type np.ndarray

See also:

https://cseweb.ucsd.edu/classes/wi18/cse167-a/lec4.pdf http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/projection/viewport_transformation.html

polliwog.transform.**world_to_canvas_orthographic_projection**(*width*, *height*, *position*, *target*, *zoom=1*, *inverse=False*)

Create a transformation matrix which composes camera, orthographic projection, and viewport transformations into a single operation.

Parameters

- **width** (*float*) – Width of the window, in pixels. (FIXME: Is this really correct?)
- **height** (*float*) – Height of the window, in pixels. (FIXME: Is this really correct?)

- **position** (*np.ndarray*) – The camera's position in world coordinates.
- **target** (*np.ndarray*) – The camera's target in world coordinates. *target - position* is the "look at" vector.
- **inverse** (*bool*) – When *True*, return the inverse transform instead.

Returns The *4x4* transformation matrix, which can be used with *polliwog.transform.apply_transform()*.

Return type np.ndarray

polliwog.transform.**transform_matrix_for_non_uniform_scale**(*x_factor*, *y_factor*, *z_factor*, *allow_flipping=False*, *ret_inverse_matrix=False*)

Create a transformation matrix that scales by the given factors along *x*, *y*, and *z*.

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

- **x_factor** (*float*) – The scale factor to be applied along the *x* axis, which should be positive.
- **y_factor** (*float*) – The scale factor to be applied along the *y* axis, which should be positive.
- **z_factor** (*float*) – The scale factor to be applied along the *z* axis, which should be positive.
- **allow_flipping** (*bool*) – When *True*, allows scale factors to be positive or negative, though not zero.
- **ret_inverse_matrix** (*bool*) – When *True*, also returns a matrix which provides the inverse transform.

polliwog.transform.**transform_matrix_for_rotation**(*rotation*, *ret_inverse_matrix=False*)

Create a transformation matrix from the given 3x3 rotation matrix or a Rodrigues vector.

With *ret_inverse_matrix=True*, also returns a matrix which provides the reverse transform.

polliwog.transform.**transform_matrix_for_translation**(*translation*, *ret_inverse_matrix=False*)

Create a transformation matrix which translates by the provided displacement vector.

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.**transform_matrix_for_uniform_scale**(*scale_factor*, *allow_flipping=False*, *ret_inverse_matrix=False*)

Create a transformation matrix that scales 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.
- **ret_inverse_matrix** (*bool*) – When *True*, also returns a matrix which provides the inverse transform.

CHAPTER 5

Lines

class polliwog.**Line**(*point*, *along*, *assume_normalized=False*)

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.**intersect_lines**(*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

polliwog.line.**intersect_2d_lines**(*p0*, *q0*, *p1*, *q1*)
Intersect two lines: (p0, q0) and (p1, q1). Each should be a 2D point.

polliwog.line.**project_point_to_line**(*points*, *reference_points_of_lines*, *vectors_along_lines*)
Project a point to a line, or pairwise project a stack of points to a stack of lines.

polliwog.line.**coplanar_points_are_on_same_side_of_line**(*a*, *b*, *p1*, *p2*)
Test if the given points are on the same side of the given line.

Parameters

- **a** (*np.arraylike*) – The first 3D point of interest.
- **b** (*np.arraylike*) – The second 3D point of interest.
- **p1** (*np.arraylike*) – A first point which lies on the line of interest.
- **p2** (*np.arraylike*) – A second point which lies on the line of interest.

Returns *True* when *a* and *b* are on the same side of the line defined by *p1* and *p2*.

Return type bool

CHAPTER 6

Line segments

polliwog.segment.**closest_point_of_line_segment**(*points*, *start_points*, *segment_vectors*)
Compute pairwise the point on each line segment that is nearest to the corresponding query point.

polliwog.segment.**subdivide_segment**(*p1*, *p2*, *num_points*, *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.**subdivide_segments**(*v*, *num_subdivisions=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.

CHAPTER 7

Boxes

class polliwog.**Box**(*origin*, *size*)

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.

contains(*point*, *atol=None*)

Test whether the box contains the given point. When *atol* is provided, returns *True* for points inside the box and points whose coordinates are all within *atol* of the box boundary.

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_x_plane
The plane facing the inside of the box, aligned with its maximum *x* coordinate.

max_y
The box's maximum *y* coordinate.

max_y_plane
The plane facing the inside of the box, aligned with its maximum *y* coordinate.

max_z
The box's maximum *z* coordinate.

max_z_plane
The plane facing the inside of the box, aligned with its 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_x_plane
The plane facing the inside of the box, aligned with its minimum *x* coordinate.

min_y
The box's minimum *y* coordinate.

min_y_plane
The plane facing the inside of the box, aligned with its minimum *y* coordinate.

min_z
The box's minimum *z* coordinate.

min_z_plane
The plane facing the inside of the box, aligned with its minimum *z* coordinate.

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

surface_area
The box's surface area.

v
Corners of the box as an *8x3* array of coordinates.

volume
The box's volume.

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

CHAPTER 8

Point clouds

Functions for working with point clouds (i.e. unstructured sets of 3D points).

polliwog.pointcloud.**extent**(*points*, *ret_indices=False*)

Find the distance between the two farthest-most points.

Parameters

- **points** (*np.arraylike*) – A *kx3* stack of points.
- **ret_indices** (*bool*) – When *True*, return the indices along with the distance.

Returns With *ret_indices=False*, the distance; with *ret_indices=True* a tuple *(distance, first_index, second_index)*.

Return type object

Note: This is implemented using a brute-force method.

polliwog.pointcloud.**percentile**(*points*, *axis*, *percentile*)

Given a cloud of points and an axis, find a point along that axis from the centroid at the given percentile.

Parameters

- **points** (*np.arraylike*) – A *kx3* stack of points.
- **axis** (*np.arraylike*) – A 3D vector specifying the direction of interest.
- **percentile** (*float*) – The desired percentile.

Returns A 3D point at the requested percentile.

Return type np.ndarray

CHAPTER 9

Tesselated shapes

Functions for creating sets of triangles to model 3D shapes.

These functions have two possible return types:

- When *ret_unique_vertices_and_faces=True*, they return a vertex array (with each vertex listed once) and a face array (i.e. an array of triples of vertex indices). This is ideal when using with a mesh library like Lace (https://github.com/lace/lace/) or Trimesh (https://trimsh.org/) or when you care about the topology.
- When *ret_unique_vertices_and_faces=False*, they return a flattened array of triangle coordinates with each vertex repeated. This is useful for computation that use flattened triangle coordinates, such as the functions in *polliwog.tri*.

See also:

https://en.wikipedia.org/wiki/Tessellation_(computer_graphics)

`polliwog.shapes.`**`rectangular_prism`**(*origin*, *size*, *ret_unique_vertices_and_faces=False*)

Tesselate an axis-aligned rectangular prism. One vertex is *origin*. The diametrically opposite vertex is *origin* + *size*.

Parameters

- **`origin`** (*`np.ndarray`*) – A 3D point vector containing the point on the prism with the minimum x, y, and z coords.
- **`size`** (*`np.ndarray`*) – A 3D vector specifying the prism's length, width, and height, which should be positive.
- **`ret_unique_vertices_and_faces`** (*`bool`*) – When *True* return a vertex array containing the unique vertices and an array of faces (i.e. vertex indices). When *False*, return a flattened array of triangle coordinates.

Returns

- With *ret_unique_vertices_and_faces=True*: a tuple containing an *8x3* array of vertices and a *12x3* array of triangle faces.
- With *ret_unique_vertices_and_faces=False*: a *12x3x3* matrix of flattened triangle coordinates.

Return type object

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

Tesselate an axis-aligned cube. One vertex is *origin*. The diametrically opposite vertex is *size* units along *+x*, *+y*, and *+z*.

Parameters

- **origin** (*np.ndarray*) – A 3D point vector containing the point on the prism with the minimum x, y, and z coords.
- **size** (*float*) – The length, width, and height of the cube, which should be positive.
- **ret_unique_vertices_and_faces** (*bool*) – When *True* return a vertex array containing the unique vertices and an array of faces (i.e. vertex indices). When *False*, return a flattened array of triangle coordinates.

Returns

- With *ret_unique_vertices_and_faces=True*: a tuple containing an *8x3* array of vertices and a *12x3* array of triangle faces.
- With *ret_unique_vertices_and_faces=False*: a *12x3x3* matrix of flattened triangle coordinates.

Return type object

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

Tesselate 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.

Parameters

- **p1** (*np.ndarray*) – A 3D point on the base of the prism.
- **p2** (*np.ndarray*) – A 3D point on the base of the prism.
- **p3** (*np.ndarray*) – A 3D point on the base of the prism.
- **height** (*float*) – The height of the prism, which should be positive.
- **ret_unique_vertices_and_faces** (*bool*) – When *True* return a vertex array containing the unique vertices and an array of faces (i.e. vertex indices). When *False*, return a flattened array of triangle coordinates.

Returns

- With *ret_unique_vertices_and_faces=True*: a tuple containing an *6x3* array of vertices and a *8x3* array of triangle faces.
- With *ret_unique_vertices_and_faces=False*: a *8x3x3* matrix of flattened triangle coordinates.

Return type object

CHAPTER 10

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

U

V

W

X

Y