graphslam Documentation

Release 0.0.17

Jeff Irion

Nov 17, 2023

CONTENTS

CHAPTER

ONE

GRAPHSLAM

1.1 graphslam package

1.1.1 Subpackages

graphslam.edge package

Submodules

graphslam.edge.base_edge module

A base class for edges.

class graphslam.edge.base_edge.**BaseEdge**(*vertex_ids*, *information*, *estimate*, *vertices=None*)

Bases: ABC

A class for representing edges in Graph SLAM.

Parameters

- **vertex_ids** (*list[int]*) – The IDs of all vertices constrained by this edge
- **information** (*np.ndarray*) – The information matrix Ω_j associated with the edge
- **estimate** (BasePose, *np.ndarray*, *float*) – The expected measurement $\mathbf{z}_j$
- **vertices** (*list[graphslam.vertex.Vertex], None*) – A list of the vertices constrained by the edge

estimate

The expected measurement $\mathbf{z}_j$

Type
BasePose, np.ndarray, float

information

The information matrix Ω_j associated with the edge

Type
np.ndarray

vertex_ids

The IDs of all vertices constrained by this edge

Type
list[int]

vertices

A list of the vertices constrained by the edge

Type
list[*graphslam.vertex.Vertex*], None

_NUMERICAL_DIFFERENTIATION_EPSILON = 1e-06

The difference that will be used for numerical differentiation

_abc_impl = <_abc._abc_data object>

_calc_jacobian(*err*, *dim*, *vertex_index*)

Calculate the Jacobian of the edge with respect to the specified vertex's pose.

Parameters

- **err** (*np.ndarray*) – The current error for the edge (see $BaseEdge.calc_error$)
- **dim** (*int*) – The dimensionality of the compact pose representation
- **vertex_index** (*int*) – The index of the vertex (pose) for which we are computing the Jacobian

Returns
The Jacobian of the edge with respect to the specified vertex's pose

Return type
np.ndarray

_is_valid()

Check some basic criteria for the edge.

Returns
Whether the basic validity criteria for the edge are satisfied

Return type
bool

calc_chi2()

Calculate the χ^2 error for the edge.

$$\mathbf{e}_j^T \Omega_j \mathbf{e}_j$$

Returns
The χ^2 error for the edge

Return type
float

calc_chi2_gradient_hessian()

Calculate the edge's contributions to the graph's χ^2 error, gradient ($\mathbf{b}$), and Hessian (H).

Returns

- *float* – The χ^2 error for the edge
- *list[tuple[int, np.ndarray]]* – The edge's contribution(s) to the gradient
- *list[tuple[tuple[int, int], np.ndarray]]* – The edge's contribution(s) to the Hessian

abstract calc_error()

Calculate the error for the edge: $\mathbf{e}_j \in \mathbb{R}^{\bullet}$.

Returns
The error for the edge

Return type
np.ndarray, float

calc_jacobians()

Calculate the Jacobian of the edge's error with respect to each constrained pose.

$$\frac{\partial}{\partial \Delta \mathbf{x}^k} \left[\mathbf{e}_j(\mathbf{x}^k \boxplus \Delta \mathbf{x}^k) \right]$$

Returns
The Jacobian matrices for the edge with respect to each constrained pose

Return type
list[np.ndarray]

equals(*other*, *tol=1e-06*)

Check whether two edges are equal.

Parameters

- **other** (BaseEdge) – The edge to which we are comparing
- **tol** (*float*) – The tolerance

Returns
Whether the two edges are equal

Return type
bool

classmethod from_g2o(*line*, *g2o_params_or_none=None*)

Load an edge from a line in a .g2o file.

Note: Overload this method to support loading from .g2o files.

Parameters

- **line** (*str*) – The line from the .g2o file
- **g2o_params_or_none** (*dict, None*) – A dictionary where the values are *graphslam.g2o_parameters.BaseG2OParameters* objects, or None if there are no such parameters

Returns
The instantiated edge object, or None if line does not correspond to this edge type (or if this edge type does not support loading from g2o)

Return type
BaseEdge, None

abstract is_valid()

Check that the edge is valid.

- The *vertices* attribute is populated, it is the correct length, and the poses are the correct types

- The *estimate* attribute is the correct type and length
- The *information* attribute is the right shape
- Any other checks

Returns
Whether the edge is valid

Return type
bool

`plot`(*color=''*)

Plot the edge.

Note: Overload this method to support plotting the edge.

Parameters
color (*str*) – The color that will be used to plot the edge

`to_g2o`()

Export the edge to the .g2o format.

Note: Overload this method to support writing to .g2o files.

Returns
The edge in .g2o format, or `None` if writing to g2o format is not supported

Return type
str, None

graphslam.edge.edge_landmark module

A class for landmark edges.

`class graphslam.edge.edge_landmark.`**EdgeLandmark**(*vertex_ids*, *information*, *estimate*, *offset*, *offset_id=None*, *vertices=None*)

Bases: *BaseEdge*

A class for representing landmark edges in Graph SLAM.

Parameters

- **vertex_ids** (*list[int]*) – The IDs of all vertices constrained by this edge
- **information** (*np.ndarray*) – The information matrix Ω_j associated with the edge
- **estimate** (BasePose) – The expected measurement $\mathbf{z}_j$; this should be the same type as `self.vertices[1].pose`
- **offset** (BasePose, *None*) – The offset that is applied to the first pose; this should be the same type as `self.vertices[0].pose`
- **offset_id** (*int*, *None*) – The ID of the offset; this is only used for writing to .g2o format

- **vertices** (*list[graphslam.vertex.Vertex], None*) – A list of the vertices constrained by the edge

estimate

The expected measurement $\mathbf{z}_j$; this should be the same type as `self.vertices[1].pose`

Type
BasePose

information

The information matrix Ω_j associated with the edge

Type
np.ndarray

offset

The offset that is applied to the first pose; this should be the same type as `self.vertices[0].pose`

Type
BasePose, None

offset_id

The ID of the offset; this is only used for writing to .g2o format

Type
int, None

vertex_ids

The IDs of all vertices constrained by this edge

Type
list[int]

vertices

A list of the vertices constrained by the edge

Type
list[*graphslam.vertex.Vertex*], None

_abc_impl = <_abc._abc_data object>

calc_error()

Calculate the error for the edge: $\mathbf{e}_j \in \mathbb{R}^{\bullet}$.

$$\mathbf{e}_j = ((p_1 \oplus p_{\text{offset}})^{-1} \oplus p_2) - \mathbf{z}_j$$

$SE(2)$ landmark edges in g2o

- https://github.com/RainerKuemmerle/g2o/blob/c422dcc0a92941a0dfedd8531cb423138c5181bd/g2o/types/slam2d/edge_se2_pointxy.h#L44-L48

$SE(3)$ landmark edges in g2o

- **https://github.com/RainerKuemmerle/g2o/blob/c422dcc0a92941a0dfedd8531cb423138c5181bd/g2o/types/slam3d/edge_se3_pointxyz.cpp#L81-L92**
  - https://github.com/RainerKuemmerle/g2o/blob/c422dcc0a92941a0dfedd8531cb423138c5181bd/g2o/types/slam3d/parameter_se3_offset.h#L76-L82
  - https://github.com/RainerKuemmerle/g2o/blob/c422dcc0a92941a0dfedd8531cb423138c5181bd/g2o/types/slam3d/parameter_se3_offset.cpp#L70

returns
The error for the edge

rtype
np.ndarray

`calc_jacobians()`

Calculate the Jacobian of the edge's error with respect to each constrained pose.

$$\frac{\partial}{\partial \Delta \mathbf{x}^k}\left[\mathbf{e}_j(\mathbf{x}^k \boxplus \Delta \mathbf{x}^k)\right]$$

Returns
The Jacobian matrices for the edge with respect to each constrained pose

Return type
list[np.ndarray]

`equals(`*other*, *tol=1e-06*`)`

Check whether two edges are equal.

Parameters

- **other** (BaseEdge) – The edge to which we are comparing
- **tol** (*float*) – The tolerance

Returns
Whether the two edges are equal

Return type
bool

`classmethod from_g2o(`*line*, *g2o_params_or_none=None*`)`

Load an edge from a line in a .g2o file.

Parameters

- **line** (*str*) – The line from the .g2o file
- **g2o_params_or_none** (*dict, None*) – A dictionary where the values are *graphslam.g2o_parameters.BaseG2OParameters* objects, or `None` if there are no such parameters

Returns
The instantiated edge object, or `None` if `line` does not correspond to a landmark edge

Return type
EdgeLandmark, None

is_valid()

Check that the edge is valid.

Returns
Whether the edge is valid

Return type
bool

plot(*color='g'*)

Plot the edge.

Parameters
color (*str*) – The color that will be used to plot the edge

to_g2o()

Export the edge to the .g2o format.

Returns
The edge in .g2o format

Return type
str

graphslam.edge.edge_odometry module

A class for odometry edges.

class graphslam.edge.edge_odometry.**EdgeOdometry**(*vertex_ids*, *information*, *estimate*, *vertices=None*)

Bases: *BaseEdge*

A class for representing odometry edges in Graph SLAM.

Parameters

- **vertex_ids** (*list[int]*) – The IDs of all vertices constrained by this edge
- **information** (*np.ndarray*) – The information matrix Ω_j associated with the edge
- **estimate** (BasePose) – The expected measurement $\mathbf{z}_j$
- **vertices** (*list[graphslam.vertex.Vertex], None*) – A list of the vertices constrained by the edge

estimate

The expected measurement $\mathbf{z}_j$

Type
BasePose

information

The information matrix Ω_j associated with the edge

Type
np.ndarray

vertex_ids

The IDs of all vertices constrained by this edge

Type
list[int]

`vertices`

A list of the vertices constrained by the edge

Type
list[*graphslam.vertex.Vertex*], None

`_abc_impl = <_abc._abc_data object>`

`calc_error()`

Calculate the error for the edge: $\mathbf{e}_j \in \mathbb{R}^{\bullet}$.

$$\mathbf{e}_j = \mathbf{z}_j - (p_2 \ominus p_1)$$

Returns
The error for the edge

Return type
np.ndarray

`calc_jacobians()`

Calculate the Jacobian of the edge's error with respect to each constrained pose.

$$\frac{\partial}{\partial \Delta \mathbf{x}^k} \left[\mathbf{e}_j(\mathbf{x}^k \boxplus \Delta \mathbf{x}^k) \right]$$

Returns
The Jacobian matrices for the edge with respect to each constrained pose

Return type
list[np.ndarray]

`classmethod from_g2o(`*line*, *g2o_params_or_none=None*`)`

Load an edge from a line in a .g2o file.

Parameters

- **line** (*str*) – The line from the .g2o file
- **g2o_params_or_none** (*dict, None*) – A dictionary where the values are *graphslam.g2o_parameters.BaseG2OParameters* objects, or `None` if there are no such parameters

Returns
The instantiated edge object, or `None` if `line` does not correspond to an odometry edge

Return type
EdgeOdometry, None

`is_valid()`

Check that the edge is valid.

Returns
Whether the edge is valid

Return type
bool

`plot(`*color='b'*`)`

Plot the edge.

Parameters
color (*str*) – The color that will be used to plot the edge

`to_g2o()`

Export the edge to the .g2o format.

Returns
The edge in .g2o format

Return type
str

Module contents

graphslam.pose package

Submodules

graphslam.pose.base_pose module

A base class for poses.

`class graphslam.pose.base_pose.BasePose`

Bases: `ndarray`

A base class for poses.

`copy()`

Return a copy of the pose.

Returns
A copy of the pose

Return type
BasePose

`equals(other, tol=1e-06)`

Check whether two poses are equal.

Parameters

- **other** (BasePose) – The pose to which we are comparing
- **tol** (*float*) – The tolerance

Returns
Whether the two poses are equal

Return type
bool

`classmethod identity()`

Return the identity pose.

Returns
The identity pose

Return type
BasePose

property inverse

Return the pose's inverse.

Returns
The pose's inverse

Return type
BasePose

jacobian_boxplus()

Compute the Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$.

Let

```
# The dimensionality of :math:`\Delta \mathbf{x}`, which should be the same as
# the compact dimensionality of `self`
n_dx = self.COMPACT_DIMENSIONALITY

# The dimensionality of :math:`p_1 \boxplus \Delta \mathbf{x}`, which should be
# the same as the dimensionality of `self`
n_boxplus = len(self.to_array())
```

Then the shape of the Jacobian will be `n_boxplus x n_dx`.

Returns
The Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$

Return type
np.ndarray

jacobian_inverse()

Compute the Jacobian of p^{-1}.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())
```

Then the shape of the Jacobian will be `n_self x n_self`.

Returns
The Jacobian of p^{-1}

Return type
np.ndarray

jacobian_self_ominus_other_wrt_other(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Let

```
# The dimensionality of `other`
n_other = len(other.to_array())

# The dimensionality of `self - other`
n_ominus = len((self - other).to_array())
```

Then the shape of the Jacobian will be `n_ominus x n_other`.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Return type
np.ndarray

jacobian_self_ominus_other_wrt_other_compact(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Let

```
# The dimensionality of `other`
n_other = len(other.to_array())

# The compact dimensionality of `self - other`
n_compact = (self - other).COMPACT_DIMENSIONALITY
```

Then the shape of the Jacobian will be `n_compact x n_other`.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Return type
np.ndarray

jacobian_self_ominus_other_wrt_self(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())

# The dimensionality of `self - other`
n_ominus = len((self - other).to_array())
```

Then the shape of the Jacobian will be `n_ominus x n_self`.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Return type
np.ndarray

jacobian_self_ominus_other_wrt_self_compact(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())

# The compact dimensionality of `self - other`
n_compact = (self - other).COMPACT_DIMENSIONALITY
```

Then the shape of the Jacobian will be `n_compact x n_self`.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Return type
np.ndarray

jacobian_self_oplus_other_wrt_other(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Let

```
# The dimensionality of `other`
n_other = len(other.to_array())

# The dimensionality of `self + other`
n_oplus = len((self + other).to_array())
```

Then the shape of the Jacobian will be `n_oplus x n_other`.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Return type
np.ndarray

jacobian_self_oplus_other_wrt_other_compact(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Let

```
# The dimensionality of `other`
n_other = len(other.to_array())

# The compact dimensionality of `self + other`
n_compact = (self + other).COMPACT_DIMENSIONALITY
```

Then the shape of the Jacobian will be `n_compact x n_other`.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Return type
np.ndarray

jacobian_self_oplus_other_wrt_self(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())

# The dimensionality of `self + other`
n_oplus = len((self + other).to_array())
```

Then the shape of the Jacobian will be `n_oplus x n_self`.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Return type
np.ndarray

jacobian_self_oplus_other_wrt_self_compact(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())

# The compact dimensionality of `self + other`
n_compact = (self + other).COMPACT_DIMENSIONALITY
```

Then the shape of the Jacobian will be `n_compact x n_self`.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Return type
np.ndarray

jacobian_self_oplus_point_wrt_point(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2, where p_2 is a point.

Let

```
# The dimensionality of `point`
n_point = len(point.to_array())

# The dimensionality of `self + point`
n_oplus = len((self + point).to_array())
```

Then the shape of the Jacobian will be `n_oplus x n_point`.

Parameters
point (BasePose) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_self`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1, where p_2 is a point.

Let

```
# The dimensionality of `self`
n_self = len(self.to_array())

# The dimensionality of `self + point`
n_oplus = len((self + point).to_array())
```

Then the shape of the Jacobian will be `n_oplus x n_self`.

Parameters
point (BasePose) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Return type
np.ndarray

`property orientation`

Return the pose's orientation.

Returns
The pose's orientation

Return type
float, np.ndarray

`property position`

Return the pose's position.

Returns
The pose's position

Return type
np.ndarray

`to_array`()

Return the pose as a numpy array.

Returns
The pose as a numpy array

Return type
np.ndarray

`to_compact`()

Return the pose as a compact numpy array.

Returns
The pose as a compact numpy array

Return type
np.ndarray

graphslam.pose.r2 module

Representation of a point in $\mathbb{R}^2$.

class graphslam.pose.r2.**PoseR2**(*position*)

Bases: *BasePose*

A representation of a 2-D point.

Parameters
position (*np.ndarray, list*) – The position in $\mathbb{R}^2$

COMPACT_DIMENSIONALITY = 2

The compact dimensionality

copy()

Return a copy of the pose.

Returns
A copy of the pose

Return type
PoseR2

classmethod identity()

Return the identity pose.

Returns
The identity pose

Return type
PoseR2

property inverse

Return the pose's inverse.

Returns
The pose's inverse

Return type
PoseR2

jacobian_boxplus()

Compute the Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$.

Returns
The Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$ (shape: 2 x 2)

Return type
np.ndarray

jacobian_inverse()

Compute the Jacobian of p^{-1}.

Returns
The Jacobian of p^{-1} (shape: 2 x 2)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `2 x 2`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `2 x 2`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

jacobian_self_oplus_other_wrt_other_compact(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

jacobian_self_oplus_other_wrt_self(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `2 x 2`)

Return type
np.ndarray

jacobian_self_oplus_other_wrt_self_compact(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `2 x 2`)

Return type
np.ndarray

jacobian_self_oplus_point_wrt_point(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2, where p_2 is a point.

Parameters
point (PoseR2) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

jacobian_self_oplus_point_wrt_self(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1, where p_2 is a point.

Parameters
point (PoseR2) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `2 x 2`)

Return type
np.ndarray

property orientation

Return the pose's orientation.

Returns
A PoseR2 object has no orientation, so this will always return 0.

Return type
float

property position

Return the pose's position.

Returns
The position portion of the pose

Return type
np.ndarray

to_array()

Return the pose as a numpy array.

Returns
The pose as a numpy array

Return type
np.ndarray

to_compact()

Return the pose as a compact numpy array.

Returns
The pose as a compact numpy array

Return type
np.ndarray

graphslam.pose.r3 module

Representation of a point in $\mathbb{R}^3$.

class graphslam.pose.r3.**PoseR3**(*position*)

Bases: *BasePose*

A representation of a 3-D point.

Parameters
position (*np.ndarray, list*) – The position in $\mathbb{R}^3$

COMPACT_DIMENSIONALITY = 3

The compact dimensionality

copy()

Return a copy of the pose.

Returns
A copy of the pose

Return type
PoseR3

classmethod identity()

Return the identity pose.

Returns
The identity pose

Return type
PoseR3

property inverse

Return the pose's inverse.

Returns
The pose's inverse

Return type
PoseR3

jacobian_boxplus()

Compute the Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$.

Returns
The Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$ (shape: `3 x 3`)

Return type
np.ndarray

jacobian_inverse()

Compute the Jacobian of p^{-1}.

Returns
The Jacobian of p^{-1} (shape: `3 x 3`)

Return type
np.ndarray

jacobian_self_ominus_other_wrt_other(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

jacobian_self_ominus_other_wrt_other_compact(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_point`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2, where p_2 is a point.

Parameters
point (PoseR3) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_self`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1, where p_2 is a point.

Parameters
point (PoseR3) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`property orientation`

Return the pose's orientation.

Returns
A `PoseR3` object has no orientation, so this will always return 0.

Return type
float

`property position`

Return the pose's position.

Returns
The position portion of the pose

Return type
np.ndarray

`to_array`()

Return the pose as a numpy array.

Returns
The pose as a numpy array

Return type
np.ndarray

`to_compact()`

Return the pose as a compact numpy array.

Returns
The pose as a compact numpy array

Return type
np.ndarray

graphslam.pose.se2 module

Representation of a pose in $SE(2)$.

`class graphslam.pose.se2.PoseSE2(position, orientation)`

Bases: *BasePose*

A representation of a pose in $SE(2)$.

Parameters

- **position** (*np.ndarray, list*) – The position in $\mathbb{R}^2$
- **orientation** (*float*) – The angle of the pose (in radians)

`COMPACT_DIMENSIONALITY = 3`

The compact dimensionality

`copy()`

Return a copy of the pose.

Returns
A copy of the pose

Return type
PoseSE2

`classmethod from_matrix(matrix)`

Return the pose as an $SE(2)$ matrix.

Parameters
matrix (*np.ndarray*) – The $SE(2)$ matrix that will be converted to a *PoseSE2* instance

Returns
The matrix as a *PoseSE2* object

Return type
PoseSE2

`classmethod identity()`

Return the identity pose.

Returns
The identity pose

Return type
PoseSE2

`property inverse`

Return the pose's inverse.

Returns
The pose's inverse

Return type
PoseSE2

`jacobian_boxplus()`

Compute the Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$.

Returns
The Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$ (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_inverse()`

Compute the Jacobian of p^{-1}.

Returns
The Jacobian of p^{-1} (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other(`*other*`)`

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (`BasePose`) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other_compact(`*other*`)`

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (`BasePose`) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self(`*other*`)`

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (`BasePose`) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_point`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2, where p_2 is a point.

Parameters
`point` (PoseR2) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `2 x 2`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_self`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1, where p_2 is a point.

Parameters
`point` (PoseR2) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `2 x 3`)

Return type
np.ndarray

`property orientation`

Return the pose's orientation.

Returns
The angle of the pose

Return type
float

`property position`

Return the pose's position.

Returns
The position portion of the pose

Return type
np.ndarray

`to_array`()

Return the pose as a numpy array.

Returns
The pose as a numpy array

Return type
np.ndarray

`to_compact`()

Return the pose as a compact numpy array.

Returns
The pose as a compact numpy array

Return type
np.ndarray

to_matrix()

Return the pose as an $SE(2)$ matrix.

Returns
The pose as an $SE(2)$ matrix

Return type
np.ndarray

graphslam.pose.se3 module

Representation of a pose in $SE(3)$.

class graphslam.pose.se3.**PoseSE3**(*position*, *orientation*)

Bases: *BasePose*

A representation of a pose in $SE(3)$.

Parameters

- **position** (*np.ndarray, list*) – The position in $\mathbb{R}^3$
- **orientation** (*np.ndarray, list*) – The orientation of the pose as a unit quaternion: $[q_x, q_y, q_z, q_w]$

COMPACT_DIMENSIONALITY = 6

The compact dimensionality

copy()

Return a copy of the pose.

Returns
A copy of the pose

Return type
PoseSE3

classmethod identity()

Return the identity pose.

Returns
The identity pose

Return type
PoseSE3

property inverse

Return the pose's inverse.

Returns
The pose's inverse

Return type
PoseSE3

jacobian_boxplus()

Compute the Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$.

Returns
The Jacobian of $p_1 \boxplus \Delta\mathbf{x}$ w.r.t. $\Delta\mathbf{x}$ evaluated at $\Delta\mathbf{x} = \mathbf{0}$ (shape: 7 x 6)

Return type
np.ndarray

`jacobian_inverse`()

Compute the Jacobian of p^{-1}.

Returns
The Jacobian of p^{-1} (shape: `7 x 7`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `7 x 7`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_other_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_2 (shape: `6 x 7`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `7 x 7`)

Return type
np.ndarray

`jacobian_self_ominus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \ominus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being subtracted from `self`

Returns
The Jacobian of $p_1 \ominus p_2$ w.r.t. p_1 (shape: `6 x 7`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – (Unused) The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `7 x 7`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_other_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2.

Parameters
other (BasePose) – (Unused) The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `6 x 7`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `7 x 7`)

Return type
np.ndarray

`jacobian_self_oplus_other_wrt_self_compact`(*other*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1.

Parameters
other (BasePose) – The pose that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `6 x 7`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_point`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_2, where p_2 is a point.

Parameters
point (PoseR3) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_2 (shape: `3 x 3`)

Return type
np.ndarray

`jacobian_self_oplus_point_wrt_self`(*point*)

Compute the Jacobian of $p_1 \oplus p_2$ w.r.t. p_1, where p_2 is a point.

Parameters
point (PoseR3) – The point that is being added to `self`

Returns
The Jacobian of $p_1 \oplus p_2$ w.r.t. p_1 (shape: `3 x 7`)

Return type
np.ndarray

`normalize`()

Normalize the quaternion portion of the pose.

`property orientation`

Return the pose's orientation.

Returns
The pose's quaternion

Return type
float

`property position`

Return the pose's position.

Returns
The position portion of the pose

Return type
np.ndarray

`to_array`()

Return the pose as a numpy array.

Returns
The pose as a numpy array

Return type
np.ndarray

`to_compact`()

Return the pose as a compact numpy array.

Returns
The pose as a compact numpy array

Return type
np.ndarray

`to_matrix`()

Return the pose as an $SE(3)$ matrix.

Returns
The pose as an $SE(3)$ matrix

Return type
np.ndarray

Module contents

1.1.2 Submodules

graphslam.g2o_parameters module

Classes for storing parameters from .g2o files.

class graphslam.g2o_parameters.**BaseG2OParameter**(*key*, *value*)

Bases: ABC

A base class for representing parameters from .g2o files.

Parameters

- **key** – A key that will be used to lookup this parameter in a dictionary
- **value** – This parameter's value

key

A key that will be used to lookup this parameter in a dictionary

value

This parameter's value

_abc_impl = <_abc._abc_data object>

abstract classmethod from_g2o(*line*)

Load a parameter from a line in a .g2o file.

Parameters
line (*str*) – The line from the .g2o file

Returns
The instantiated parameter object, or None if line does not correspond to this parameter type

Return type
BaseG2OParameter, None

abstract to_g2o()

Export the parameter to the .g2o format.

Returns
The parameter in .g2o format

Return type
str, None

class graphslam.g2o_parameters.**G2OParameterSE2Offset**(*key*, *value*)

Bases: *BaseG2OParameter*

A class for storing a g2o $SE(2)$ offset parameter.

key

A tuple of the form ("PARAMS_SE2OFFSET", id)

Type
tuple[str, int]

value

The offset

Type
graphslam.pose.se2.PoseSE2

_abc_impl = <_abc._abc_data object>

classmethod from_g2o(*line*)

Load an $SE(2)$ offset parameter from a line in a .g2o file.

Parameters
line (*str*) – The line from the .g2o file

Returns
The instantiated parameter object, or `None` if `line` does not correspond to an $SE(2)$ offset parameter

Return type
G2OParameterSE2Offset, None

to_g2o()

Export the $SE(2)$ offset parameter to the .g2o format.

Returns
The parameter in .g2o format

Return type
str

class graphslam.g2o_parameters.**G2OParameterSE3Offset**(*key*, *value*)

Bases: *BaseG2OParameter*

A class for storing a g2o $SE(3)$ offset parameter.

key

A tuple of the form `("PARAMS_SE3OFFSET", id)`

Type
tuple[str, int]

value

The offset

Type
graphslam.pose.se3.PoseSE3

_abc_impl = <_abc._abc_data object>

classmethod from_g2o(*line*)

Load an $SE(3)$ offset parameter from a line in a .g2o file.

Parameters
line (*str*) – The line from the .g2o file

Returns
The instantiated parameter object, or `None` if `line` does not correspond to an $SE(3)$ offset parameter

Return type
G2OParameterSE3Offset, None

`to_g2o()`

Export the $SE(3)$ offset parameter to the .g2o format.

Returns
The parameter in .g2o format

Return type
str

graphslam.graph module

A `Graph` class that stores the edges and vertices required for Graph SLAM.

Given:

- A set of M edges (i.e., constraints) $\mathcal{E}$
  - $e_j \in \mathcal{E}$ is an edge
  - $\mathbf{e}_j \in \mathbb{R}^{\bullet}$ is the error associated with that edge, where $\bullet$ is a scalar that depends on the type of edge
  - Ω_j is the $\bullet \times \bullet$ information matrix associated with edge e_j
- A set of N vertices $\mathcal{V}$
  - $v_i \in \mathcal{V}$ is a vertex
  - $\mathbf{x}_i \in \mathbb{R}^c$ is the compact pose associated with v_i
  - $\boxplus$ is the pose composition operator that yields a (non-compact) pose that lies in (a subspace of) $\mathbb{R}^d$

We want to optimize

$$\chi^2 = \sum_{e_j \in \mathcal{E}} \mathbf{e}_j^T \Omega_j \mathbf{e}_j.$$

Let

$$\mathbf{x} := \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} \in \mathbb{R}^{cN}.$$

We will solve this optimization problem iteratively. Let

$$\mathbf{x}^{k+1} := \mathbf{x}^k \boxplus \Delta\mathbf{x}^k.$$

The χ^2 error at iteration $k+1$ is

$$\chi^2_{k+1} = \sum_{e_j \in \mathcal{E}} \underbrace{\left[\mathbf{e}_j(\mathbf{x}^{k+1})\right]^T}_{1\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\mathbf{e}_j(\mathbf{x}^{k+1})}_{\bullet\times 1}.$$

We will linearize the errors as:

$$\begin{aligned} \mathbf{e}_j(\mathbf{x}^{k+1}) &= \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k) \\ &\approx \mathbf{e}_j(\mathbf{x}^k) + \frac{\partial}{\partial \Delta\mathbf{x}^k}\left[\mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)\right]\Delta\mathbf{x}^k \\ &= \mathbf{e}_j(\mathbf{x}^k) + \left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}}\right)\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial\Delta\mathbf{x}^k}\Delta\mathbf{x}^k. \end{aligned}$$

Plugging this into the formula for χ^2, we get:

$$
\begin{aligned}
\chi^2_{k+1} \approx \quad & \sum_{e_j \in \mathcal{E}} \underbrace{[\mathbf{e}_j(\mathbf{x}^k)]^T}_{1\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\mathbf{e}_j(\mathbf{x}^k)}_{\bullet\times 1} \\
& + \sum_{e_j \in \mathcal{E}} \underbrace{[\mathbf{e}_j(\mathbf{x}^k)]^T}_{1\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)}_{\bullet\times dN} \underbrace{\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k}}_{dN\times cN} \underbrace{\Delta\mathbf{x}^k}_{cN\times 1} \\
& + \sum_{e_j \in \mathcal{E}} \underbrace{(\Delta\mathbf{x}^k)^T}_{1\times cN} \underbrace{\left(\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k} \right)^T}_{cN\times dN} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)^T}_{dN\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)}_{\bullet\times dN} \underbrace{\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k}}_{dN\times cN} \underbrace{\Delta\mathbf{x}^k}_{cN\times 1} \\
= \chi^2_k + 2\mathbf{b}^T \Delta\mathbf{x}^k + & (\Delta\mathbf{x}^k)^T H \Delta\mathbf{x}^k,
\end{aligned}
$$

where

$$
\begin{aligned}
\mathbf{b}^T &= \sum_{e_j \in \mathcal{E}} \underbrace{[\mathbf{e}_j(\mathbf{x}^k)]^T}_{1\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)}_{\bullet\times dN} \underbrace{\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k}}_{dN\times cN} \\
H &= \sum_{e_j \in \mathcal{E}} \underbrace{\left(\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k} \right)^T}_{cN\times dN} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)^T}_{dN\times\bullet} \underbrace{\Omega_j}_{\bullet\times\bullet} \underbrace{\left(\frac{\partial \mathbf{e}_j(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=\mathbf{0}} \right)}_{\bullet\times dN} \underbrace{\frac{\partial(\mathbf{x}^k \boxplus \Delta\mathbf{x}^k)}{\partial \Delta\mathbf{x}^k}}_{dN\times cN}.
\end{aligned}
$$

Using this notation, we obtain the optimal update as

$$\Delta\mathbf{x}^k = -H^{-1}\mathbf{b}.$$

We apply this update to the poses and repeat until convergence.

class graphslam.graph.**Graph**(*edges*, *vertices*)

Bases: `object`

A graph that will be optimized via Graph SLAM.

Parameters

- **edges** (*list[graphslam.edge.base_edge.BaseEdge]*) – A list of the vertices in the graph
- **vertices** (*list[graphslam.vertex.Vertex]*) – A list of the vertices in the graph

_chi2

The current χ^2 error, or `None` if it has not yet been computed

Type
float, None

_edges

A list of the edges (i.e., constraints) in the graph

Type
list[*graphslam.edge.base_edge.BaseEdge*]

_fixed_gradient_indices

The set of gradient indices (i.e., *Vertex.gradient_index*) for vertices that are fixed

Type
set[int]

_g2o_params

A dictionary where the values are *BaseG2OParameter* objects

Type
dict, None

_gradient

The gradient $\mathbf{b}$ of the χ^2 error, or `None` if it has not yet been computed

Type
numpy.ndarray, None

_hessian

The Hessian matrix H, or `None` if it has not yet been computed

Type
scipy.sparse.lil_matrix, None

_len_gradient

The length of the gradient vector (and the Hessian matrix)

Type
int, None

_vertices

A list of the vertices in the graph

Type
list[*graphslam.vertex.Vertex*]

_calc_chi2_gradient_hessian()

Calculate the χ^2 error, the gradient $\mathbf{b}$, and the Hessian H.

_initialize()

Fill in the `vertices` attributes for the graph's edges, and other necessary preparations.

calc_chi2()

Calculate the χ^2 error for the `Graph`.

Returns
The χ^2 error

Return type
float

equals(*other*, *tol=1e-06*)

Check whether two graphs are equal.

Parameters

- **other** (Graph) – The graph to which we are comparing
- **tol** (*float*) – The tolerance

Returns
Whether the two graphs are equal

Return type
bool

classmethod from_g2o(*infile*, *custom_edge_types=None*)

Load a graph from a .g2o file.

Parameters

- **infile** (*str*) – The path to the .g2o file
- **custom_edge_types** (*list[type], None*) – A list of custom edge types, which must be subclasses of `BaseEdge`

Returns
The loaded graph

Return type
Graph

optimize(*tol=0.0001*, *max_iter=20*, *fix_first_pose=True*, *verbose=True*)

Optimize the χ^2 error for the `Graph`.

Parameters

- **tol** (*float*) – If the relative decrease in the χ^2 error between iterations is less than `tol`, we will stop
- **max_iter** (*int*) – The maximum number of iterations
- **fix_first_pose** (*bool*) – If `True`, we will fix the first pose
- **verbose** (*bool*) – Whether to print information about the optimization

Returns
ret – Information about this optimization

Return type
OptimizationResult

plot(*vertex_color='r'*, *vertex_marker='o'*, *vertex_markersize=3*, *edge_color='b'*, *title=None*)

Plot the graph.

Parameters

- **vertex_color** (*str*) – The color that will be used to plot the vertices
- **vertex_marker** (*str*) – The marker that will be used to plot the vertices
- **vertex_markersize** (*int*) – The size of the plotted vertices
- **edge_color** (*str*) – The color that will be used to plot the edges
- **title** (*str, None*) – The title that will be used for the plot

to_g2o(*outfile*)

Save the graph in .g2o format.

Parameters
outfile (*str*) – The path where the graph will be saved

class graphslam.graph.**OptimizationResult**

Bases: `object`

A class for storing information about a graph optimization; see *Graph.optimize*.

`converged`

Whether the optimization converged

Type
bool

`duration_s`

The total time for the optimization (in seconds)

Type
float, None

`final_chi2`

The final χ^2 error

Type
float, None

`initial_chi2`

The initial χ^2 error

Type
float, None

`iteration_results`

Information about each iteration

Type
list[*IterationResult*]

`num_iterations`

The number of iterations that were performed

Type
int, None

`class IterationResult`

Bases: `object`

A class for storing information about a single graph optimization iteration; see *Graph.optimize*.

`calc_chi2_gradient_hessian_duration_s`

The time to compute χ^2, the gradient, and the Hessian (in seconds); see *Graph._calc_chi2_gradient_hessian*

Type
float, None

`chi2`

The χ^2 of the graph after performing this iteration's update

Type
float, None

`duration_s`

The total time for this iteration (in seconds)

Type
float, None

`rel_diff`

The relative difference in the χ^2 as a result of this iteration

Type
float, None

`solve_duration_s`

The time to solve $H\Delta\mathbf{x}^k = -\mathbf{b}$ (in seconds)

Type
float, None

`update_duration_s`

The time to update the poses (in seconds)

Type
float, None

`is_complete_iteration()`

Whether this was a full iteration.

At iteration `i`, we compute the χ^2 error for iteration `i-1` (see *Graph.optimize*). If this meets the convergence criteria, then we do not solve the linear system and update the poses, and so this is not a complete iteration.

Returns
Whether this was a complete iteration (i.e., we solve the linear system and updated the poses)

Return type
bool

`class graphslam.graph._Chi2GradientHessian`

Bases: `object`

A class that is used to aggregate the χ^2 error, gradient, and Hessian.

`chi2`

The χ^2 error

Type
float

`gradient`

The contributions to the gradient vector

Type
defaultdict

`hessian`

The contributions to the Hessian matrix

Type
defaultdict

`class DefaultArray`

Bases: `object`

A class for use in a *defaultdict*.

`static update(chi2_grad_hess, incoming)`

Update the χ^2 error and the gradient and Hessian dictionaries.

Parameters

- **chi2_grad_hess** (_Chi2GradientHessian) – The `_Chi2GradientHessian` that will be updated

- **incoming** (*tuple*) – The return value from *BaseEdge.calc_chi2_gradient_hessian*

graphslam.load module

Functions for loading graphs.

graphslam.load.**load_g2o**(*infile*)

Load a graph from a .g2o file.

Parameters
infile (*str*) – The path to the .g2o file

Returns
The loaded graph

Return type
Graph

graphslam.load.**load_g2o_r2**(*infile*)

Load an $\mathbb{R}^2$ graph from a .g2o file.

Parameters
infile (*str*) – The path to the .g2o file

Returns
The loaded graph

Return type
Graph

graphslam.load.**load_g2o_r3**(*infile*)

Load an $\mathbb{R}^3$ graph from a .g2o file.

Parameters
infile (*str*) – The path to the .g2o file

Returns
The loaded graph

Return type
Graph

graphslam.load.**load_g2o_se2**(*infile*)

Load an $SE(2)$ graph from a .g2o file.

Parameters
infile (*str*) – The path to the .g2o file

Returns
The loaded graph

Return type
Graph

graphslam.load.**load_g2o_se3**(*infile*)

Load an $SE(3)$ graph from a .g2o file.

Parameters
infile (*str*) – The path to the .g2o file

Returns
The loaded graph

Return type
Graph

graphslam.util module

Utility functions used throughout the package.

graphslam.util.**neg_pi_to_pi**(*angle*)

Normalize `angle` to be in $[-\pi, \pi)$.

Parameters
angle (*float*) – An angle (in radians)

Returns
The angle normalized to $[-\pi, \pi)$

Return type
float

graphslam.util.**solve_for_edge_dimensionality**(*n*)

Solve for the dimensionality of an edge.

In a .g2o file, an edge is specified as `<estimate> <information matrix>`, where only the upper triangular portion of the matrix is provided.

This solves the problem:

$$d + \frac{d(d+1)}{2} = n$$

Returns
The dimensionality of the edge

Return type
int

graphslam.util.**upper_triangular_matrix_to_full_matrix**(*arr*, *n*)

Given an upper triangular matrix, return the full matrix.

Parameters

- **arr** (*np.ndarray*) – The upper triangular portion of the matrix
- **n** (*int*) – The size of the matrix

Returns
mat – The full matrix

Return type
np.ndarray

graphslam.vertex module

A `Vertex` class.

class graphslam.vertex.**Vertex**(*vertex_id*, *pose*, *fixed=False*)

Bases: `object`

A class for representing a vertex in Graph SLAM.

Parameters

- **vertex_id** (*int*) – The vertex's unique ID
- **pose** (graphslam.pose.base_pose.BasePose) – The pose associated with the vertex
- **fixed** (*bool*) – Whether this vertex should be fixed

gradient_index

The index of the first entry in the gradient vector to which this vertex corresponds (and similarly for the Hessian matrix)

Type
int, None

id

The vertex's unique ID

Type
int

pose

The pose associated with the vertex

Type
graphslam.pose.base_pose.BasePose

fixed

Whether this vertex should be fixed

Type
bool

equals(*other*, *tol=1e-06*)

Check whether two vertices are equal.

Parameters

- **other** (Vertex) – The vertex to which we are comparing
- **tol** (*float*) – The tolerance

Returns
Whether the two vertices are equal

Return type
bool

classmethod from_g2o(*line*)

Load a vertex from a line in a .g2o file.

Parameters
line (*str*) – The line from the .g2o file

Returns
The instantiated vertex object, or `None` if `line` does not correspond to a vertex

Return type
Vertex, None

`plot`(*color='r'*, *marker='o'*, *markersize=3*)

Plot the vertex.

Parameters

- **color** (*str*) – The color that will be used to plot the vertex
- **marker** (*str*) – The marker that will be used to plot the vertex
- **markersize** (*int*) – The size of the plotted vertex

`to_g2o`()

Export the vertex to the .g2o format.

Returns
The vertex in .g2o format

Return type
str

1.1.3 Module contents

Graph SLAM solver in Python.

Documentation for this package can be found at https://python-graphslam.readthedocs.io/.

This package implements a Graph SLAM solver in Python.

CHAPTER

TWO

FEATURES

- Optimize $\mathbb{R}^2$, $\mathbb{R}^3$, $SE(2)$, and $SE(3)$ datasets
- Analytic Jacobians
- Supports odometry and landmark edges
- Supports custom edge types (see tests/test_custom_edge.py for an example)
- Import and export .g2o files

CHAPTER

THREE

INSTALLATION

```
pip install graphslam
```

CHAPTER

FOUR

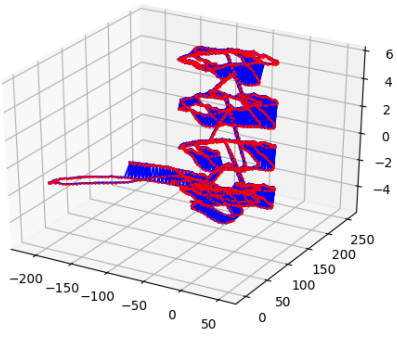
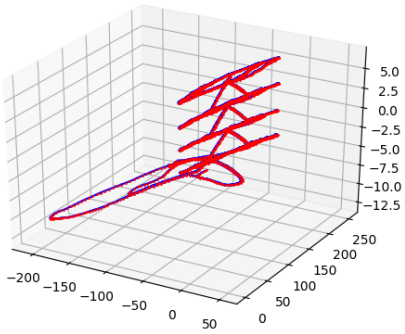
EXAMPLE USAGE

4.1 SE(3) Dataset

```
>>> from graphslam.graph import Graph

>>> g = Graph.from_g2o("data/parking-garage.g2o")  # https://lucacarlone.mit.edu/
↪datasets/

>>> g.plot(vertex_markersize=1)

>>> g.calc_chi2()

16720.02100546733

>>> g.optimize()

>>> g.plot(vertex_markersize=1)
```

Output:

```
Iteration                chi^2        rel. change
---------                -----        -----------
        0          16720.0210
        1             45.6644          -0.997269
        2              1.2936          -0.971671
        3              1.2387          -0.042457
        4              1.2387          -0.000001
```

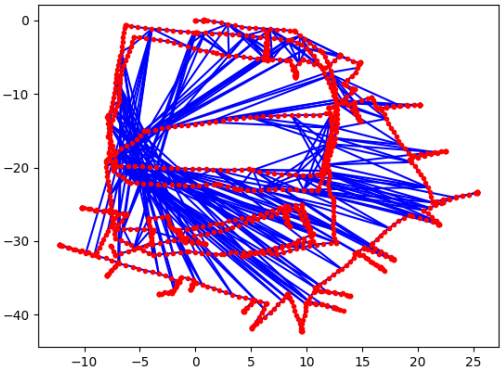
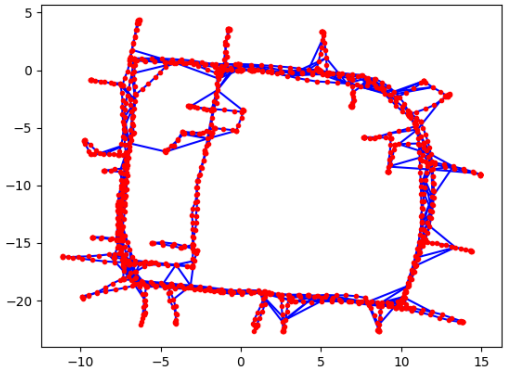
Original	Optimized
	

4.2 SE(2) Dataset

```
>>> from graphslam.graph import Graph

>>> g = Graph.from_g2o("data/input_INTEL.g2o")  # https://lucacarlone.mit.edu/datasets/

>>> g.plot()

>>> g.calc_chi2()

7191686.382493544

>>> g.optimize()

>>> g.plot()
```

Output:

```
Iteration                chi^2        rel. change
---------                -----        -----------
        0         7191686.3825
        1       319950425.6477          43.488929
        2       124950341.8035          -0.609470
        3          338165.0770          -0.997294
        4             734.7343          -0.997827
        5             215.8405          -0.706233
        6             215.8405          -0.000000
```

Original	Optimized
	

CHAPTER

FIVE

REFERENCES AND ACKNOWLEDGMENTS

1. Grisetti, G., Kummerle, R., Stachniss, C. and Burgard, W., 2010. A tutorial on graph-based SLAM. IEEE Intelligent Transportation Systems Magazine, 2(4), pp.31-43.
2. Blanco, J.L., 2010. A tutorial on SE(3) transformation parameterizations and on-manifold optimization. University of Malaga, Tech. Rep, 3.
3. Carlone, L., Tron, R., Daniilidis, K. and Dellaert, F., 2015, May. Initialization techniques for 3D SLAM: a survey on rotation estimation and its use in pose graph optimization. In 2015 IEEE international conference on robotics and automation (ICRA) (pp. 4597-4604). IEEE.
4. Carlone, L. and Censi, A., 2014. From angular manifolds to the integer lattice: Guaranteed orientation estimation with application to pose graph optimization. IEEE Transactions on Robotics, 30(2), pp.475-492.

Thanks to Luca Larlone for allowing inclusion of the Intel and parking garage datasets in this repo.

CHAPTER

SIX

LIVE CODING GRAPH SLAM IN PYTHON

If you're interested, you can watch as I coded this up.

1. Live coding Graph SLAM in Python (Part 1)
2. Live coding Graph SLAM in Python (Part 2)
3. Live coding Graph SLAM in Python (Part 3)
4. Live coding Graph SLAM in Python (Part 4)
5. Live coding Graph SLAM in Python (Part 5)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

B

C

D

E

F

G

H

I

J

K

L

M

N

O