# pybotics

Aug 30, 2023

# Contents

# CHAPTER 1

# Overview

*Pybotics* is an open-source Python toolbox for robot kinematics and calibration. It was designed to provide a simple, clear, and concise interface to quickly simulate and evaluate common robot concepts, such as kinematics, dynamics, trajectory generations, and calibration. The toolbox is specifically designed for use with the Modified Denavit–Hartenberg parameters convention.

Please see the README for more information and examples.

CHAPTER 2

# Installation

## 2.1 pybotics

### 2.1.1 pybotics package

#### 2.1.1.1 Submodules

#### pybotics.errors module

Pybotics errors.

**exception** pybotics.errors.**PyboticsError**(*message: str = 'Pybotics error'*)
Bases: Exception

Base class for Pybotics errors.

**args**

**with_traceback**()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

#### pybotics.geometry module

Geometry functions and utilities.

**class** pybotics.geometry.**OrientationConvention**
Bases: enum.Enum

Orientation of a body with respect to a fixed coordinate system.

**EULER_XYX = 'xyx'**

**EULER_XYZ = 'xyz'**

**EULER_XZX = 'xzx'**

**EULER_XZY = 'xzy'**

**EULER_YXY = 'yxy'**

**EULER_YXZ = 'yxz'**

**EULER_YZX = 'yzx'**

**EULER_YZY = 'yzy'**

**EULER_ZXY = 'zxy'**

**EULER_ZXZ = 'zxz'**

**EULER_ZYX = 'zyx'**

**EULER_ZYZ = 'zyz'**

**FIXED_XYX = 'xyx'**

**FIXED_XYZ = 'zyx'**

**FIXED_XZX = 'xzx'**

**FIXED_XZY = 'yzx'**

**FIXED_YXY = 'yxy'**

**FIXED_YXZ = 'zxy'**

**FIXED_YZX = 'xzy'**

**FIXED_YZY = 'yzy'**

**FIXED_ZXY = 'yxz'**

**FIXED_ZXZ = 'zxz'**

**FIXED_ZYX = 'xyz'**

**FIXED_ZYZ = 'zyz'**

pybotics.geometry.**matrix_2_vector**(*matrix: numpy.ndarray[Any, numpy.dtype[numpy.float64]], convention: pybotics.geometry.OrientationConvention = <OrientationConvention.EULER_ZYX: 'zyx'>*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Convert 4x4 matrix to a vector.

pybotics.geometry.**position_from_matrix**(*matrix: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get the position values from a 4x4 transform matrix.

pybotics.geometry.**rotation_matrix_x**(*angle: float*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Generate a basic 4x4 rotation matrix about the X axis.

pybotics.geometry.**rotation_matrix_y**(*angle: float*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Generate a basic 4x4 rotation matrix about the Y axis.

pybotics.geometry.**rotation_matrix_z**(*angle: float*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Generate a basic 4x4 rotation matrix about the Z axis.

pybotics.geometry.**translation_matrix**(*xyz: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Generate a basic 4x4 translation matrix.

pybotics.geometry.**vector_2_matrix**(*vector: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *convention: Union[pybotics.geometry.OrientationConvention, str] = <OrientationConvention.EULER_ZYX: 'zyx'>*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Calculate the pose from the position and euler angles.

**Parameters**

- **convention** –
- **vector** – transform vector

**Returns** 4x4 transform matrix

pybotics.geometry.**wrap_2_pi**(*angle: float*) → float

Wrap given angle to +/- PI.

**Parameters** **angle** – angle to wrap

**Returns** wrapped angle

## pybotics.json_encoder module

JSON Encoder for Pybotics classes.

**class** pybotics.json_encoder.**JSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

Pybotics JSON Encoder class.

**default**(*o: Any*) → Any

Return serializable robot objects.

**encode**(*o*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

**item_separator = ', '**

**iterencode**(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

**key_separator = ': '**

### pybotics.kinematic_chain module

Kinematic chain module.

**class** pybotics.kinematic_chain.**KinematicChain**
Bases: collections.abc.Sized, typing.Generic

An assembly of rigid bodies connected by joints.

Provides constrained (or desired) motion that is the mathematical model for a mechanical system.

**links**
Get links.

**matrix**
Convert chain to matrix of link parameters.

Rows = links Columns = parameters

**ndof**
Get number of degrees of freedom.

**Returns** number of degrees of freedom

**num_parameters**
Get number of parameters of all links.

**to_json**() → str
Encode model as JSON.

**transforms**(*q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → Sequence[numpy.ndarray[Any, numpy.dtype[numpy.float64]]]
Generate a sequence of spatial transforms.

The sequence represents the given position of the kinematic chain. :param q: given position of kinematic chain :return: sequence of transforms

**vector**
Get the vector representation of the kinematic chain.

**Returns** vectorized kinematic chain

**class** pybotics.kinematic_chain.**MDHKinematicChain**(*links: Sequence[pybotics.link.MDHLink]*)
Bases: *pybotics.kinematic_chain.KinematicChain*

Kinematic Chain of MDH links.

**classmethod from_parameters**(*parameters: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → Any
Construct Kinematic Chain from parameters.

**links**
Get links.

**matrix**
Convert chain to matrix of link parameters.

Rows = links Columns = parameters

**ndof**
Get number of degrees of freedom.

**Returns** number of degrees of freedom

**num_parameters**
Get number of parameters of all links.

**to_json**() → str
Encode model as JSON.

**transforms**(*q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → Sequence[numpy.ndarray[Any, numpy.dtype[numpy.float64]]]
Get sequence of 4x4 transforms.

**vector**
Get parameters of all links.

### pybotics.link module

Link module.

**class** pybotics.link.**Link**
Bases: collections.abc.Sized

Links: connected joints allowing relative motion of neighboring link.

**displace**(*q: float*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a vector of the new link state given a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**size**
Get number of parameters.

**to_json**() → str
Encode model as JSON.

**transform**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a 4x4 transform matrix given a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**vector**
Return the vector representation of the link.

**Returns** vectorized kinematic chain

**class** pybotics.link.**MDHLink**(*alpha: float = 0*, *a: float = 0*, *theta: float = 0*, *d: float = 0*)
Bases: *pybotics.link.Link*

Link class that uses Modified DH parameters.

https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

**displace**(*q: float*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a vector of the new link state given a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**size**
Get number of parameters.

**to_json**() → str
Encode model as JSON.

**transform**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a 4x4 transform matrix with a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**vector**
Return the vector representation of the link.

**Returns** vectorized kinematic chain

**class** pybotics.link.**PrismaticMDHLink**(*alpha: float = 0*, *a: float = 0*, *theta: float = 0*, *d: float = 0*)
Bases: *pybotics.link.MDHLink*

Link class that uses Modified DH parameters for a revolute joint.

https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

**displace**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a vector of the new link state given a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**size**
Get number of parameters.

**to_json**() → str
Encode model as JSON.

**transform**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a 4x4 transform matrix with a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**vector**
Return the vector representation of the link.

**Returns** vectorized kinematic chain

**class** pybotics.link.**RevoluteMDHLink**(*alpha: float = 0*, *a: float = 0*, *theta: float = 0*, *d: float = 0*)
Bases: *pybotics.link.MDHLink*

Link class that uses Modified DH parameters for a revolute joint.

https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

**displace**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a vector of the new link state given a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**size**
Get number of parameters.

**to_json**() → str
Encode model as JSON.

**transform**(*q: float = 0*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate a 4x4 transform matrix with a displacement.

**Parameters** **q** – given displacement

:return vector of new displacement state

**vector**
Return the vector representation of the link.

**Returns** vectorized kinematic chain

### pybotics.optimization module

Optimization module.

**class** pybotics.optimization.**OptimizationHandler**(*robot: pybotics.robot.Robot*, *kinematic_chain_mask: MutableSequence[bool] = False*, *tool_mask: MutableSequence[bool] = False*, *world_mask: MutableSequence[bool] = False*)

Bases: object

Handler for optimization tasks.

**apply_optimization_vector**(*vector: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → None
Apply vector.

**generate_optimization_vector**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Generate vector.

pybotics.optimization.**compute_absolute_error**(*q: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *position: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *robot: pybotics.robot.Robot*) → float
Compute the absolute error of a given position.

pybotics.optimization.**compute_absolute_errors**(*qs: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *positions: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *robot: pybotics.robot.Robot*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
Compute the absolute errors of a given set of positions.

**Parameters**

- **qs** – Array of joints, shape=(n-poses, n-dof) [rad]
- **positions** – Array of Cartesian positions, shape=(n-poses, 3)
- **robot** – Robot model

pybotics.optimization.**compute_relative_error**(*q_a: numpy.ndarray[Any, numpy.dtype[numpy.float64]], q_b: numpy.ndarray[Any, numpy.dtype[numpy.float64]], distance: float, robot: pybotics.robot.Robot*) → float

Compute the relative error of a given position combination.

pybotics.optimization.**compute_relative_errors**(*qs_a: numpy.ndarray[Any, numpy.dtype[numpy.float64]], qs_b: numpy.ndarray[Any, numpy.dtype[numpy.float64]], distances: numpy.ndarray[Any, numpy.dtype[numpy.float64]], robot: pybotics.robot.Robot*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the relative errors of a given set of position combinations.

pybotics.optimization.**optimize_accuracy**(*optimization_vector: numpy.ndarray[Any, numpy.dtype[numpy.float64]], handler: pybotics.optimization.OptimizationHandler, qs: numpy.ndarray[Any, numpy.dtype[numpy.float64]], positions: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Fitness function for accuracy optimization.

### pybotics.predefined_models module

Predefined robot models.

These models correspond to the Modified Denavit–Hartenberg parameters: https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

pybotics.predefined_models.**abb_irb120**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get ABB irb120 MDH model.

pybotics.predefined_models.**kuka_lbr_iiwa_7**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get KUKA LBR iiwa 7 MDH model.

pybotics.predefined_models.**mecademic_meca500**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get Meca500 MDH model.

pybotics.predefined_models.**puma560**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get PUMA560 MDH model.

pybotics.predefined_models.**ur10**() → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Get UR10 MDH model.

### pybotics.robot module

Robot module.

**class** pybotics.robot.**Robot**(*kinematic_chain: pybotics.kinematic_chain.KinematicChain*, *tool: pybotics.tool.Tool = NOTHING*, *world_frame: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING*, *random_state: numpy.random.mtrand.RandomState = NOTHING*, *home_position: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING*, *joints: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING*, *joint_limits: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING*)

Bases: collections.abc.Sized, typing.Generic

Robot manipulator class.

**clamp_joints**(*q: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Limit joints to joint limits.

**compute_joint_torques**(*wrench: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Calculate the joint torques due to external flange force.

Method from: 5.9 STATIC FORCES IN MANIPULATORS Craig, John J. Introduction to robotics: mechanics and control. Vol. 3. Upper Saddle River: Pearson Prentice Hall, 2005. :param wrench: :param q: :return:

**fk**(*q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Compute the forward kinematics of a given position.

Uses the current position if None is given. :param q: :return: 4x4 transform matrix of the FK pose

**classmethod from_parameters**(*parameters: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*) → pybotics.robot.Robot

Construct Robot from Kinematic Chain parameters.

**ik**(*pose: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]]

Solve the inverse kinematics.

**jacobian_flange**(*q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Calculate the Jacobian wrt the flange frame.

**jacobian_world**(*q: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]

Calculate the Jacobian wrt the world frame.

**joint_limits**

Limits of the robot position (e.g., joint limits).

**Returns** limits with shape (2,num_dof) where first row is upper limits

**joints**

Get the robot configuration (e.g., joint positions for serial robot).

**Returns** robot position

**ndof**

Get the number of degrees of freedom.

**Returns** number of degrees of freedom

**random_joints**(*in_place: bool = False*) → Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]]
Generate random joints within limits.

**to_json**() → str
Encode robot model as JSON.

### pybotics.tool module

Tool module.

isort:skip_file

**class** pybotics.tool.**Tool**(*matrix: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING, mass: float = 0, cg: numpy.ndarray[Any, numpy.dtype[numpy.float64]] = NOTHING*)
Bases: object

Tool class.

**position**
Get the position XYZ of the frame.

**Returns**

**vector**
Return the vector representation of the frame as EULER ZYX.

**Returns** vectorized frame

#### 2.1.1.2 Module contents

Pybotics modules.

CHAPTER 3

# Indices

- genindex
- modindex
- search

# Python Module Index

## p

# Index

## S



## T



## U



## V



## W