Not your grandmother’s toolbox – the Robotics Toolbox reinvented for Python

Peter Corke¹, Jesse Haviland¹

Abstract—For 25 years the Robotics Toolbox for MATLAB® has been used for teaching and research worldwide. This paper describes its successor – the Robotics Toolbox for Python. More than just a port, it takes advantage of popular open-source packages and resources to provide platform portability, fast browser-based 3D graphics, quality documentation, fast numerical and symbolic operations, powerful IDEs, shareable and web-browsable notebooks all powered by GitHub and the open-source community. The new Toolbox provides well-known functionality for spatial mathematics (homogeneous transformations, quaternions, triple angles and twists), trajectories, kinematics (zeroth to second order), dynamics and a rich assortment of robot models. In addition, we’ve taken the opportunity to add new capabilities such as branched mechanisms, collision checking, URDF import, and interfaces to ROS. With familiar, simple yet powerful functions; the clarity of Python syntax; but without the complexity of ROS; users from beginner to advanced will find this a powerful open-source toolset for ongoing robotics education and research.

I. INTRODUCTION

The Robotics Toolbox for MATLAB® (RTB-M) was created in 1991 to support the first author’s PhD research and was first published in 1995-6 [1][2]. It has evolved over 25 years to track changes to MATLAB such as the addition of structures, objects, lists (cell arrays) and strings, new graphics and new tools such as an IDE, debugger, notebooks (LiveScripts), apps and continuous integration. An adverse consequence is that many poor early design decisions hinder ongoing development.

Over time additional functionality was also added, in particular for vision, and two major refactorings led to the current state of three toolboxes: Robotics Toolbox for MATLAB and Machine Vision Toolbox for MATLAB (1999) both of which are built on the Spatial Math Toolbox for MATLAB (SMTB-M) from 2019 [3]. The code was formally open sourced to support its use for the third edition of John Craig’s book [4]. It was hosted on ftp sites, personal web servers, Google code and currently GitHub and maintained under a succession of version control tools including rcs, cvs, svn and git. A support forum on Google Groups was established in 2008 and currently has over 1400 members.

This paper describes the motivation and design of the Robotics Toolbox for Python, and illustrates key features in a tutorial fashion.

II. A PYTHON VERSION

The imperative for a Python version has long existed and the first port was started in 2008 but ultimately failed for

¹Peter Corke and Jesse Haviland are with the Queensland University of Technology Centre for Robotics (QCR), Brisbane, Australia peter.corke@qut.edu.au, j.haviland@qut.edu.au. This research was supported by the QUT Centre for Robotics.

Fig. 1: The toolbox provides several visualizers
lack of ongoing resources to complete a sufficient subset of functionality. Several subsequent attempts met the same fate.

The design goals of this Python version are new functionality:

- A superset of the MATLAB Toolbox functionality.
- Build on the Spatial Math Toolbox for Python which provides objects to represent rotations as matrices in SO(2) and SO(3) as well as unit-quotients, rigid-body motions as matrices SE(2) and SE(3) as well as twists in se(2) and se(3), and spatial vectors [5].
- Allow kinematic models to be expressed using elementary transform sequences [6], [7]. URDF-style rigid-body trees as well as Denavit-Hartenberg notation (standard and modified). Support branched, but not closed-loop or parallel, robots.
- Collision and distance-to-collision checking.

and improved software engineering:

- Use Python 3 (3.6 and greater).
- Utilize WebGL and Javascript graphics technologies.
- Documentation in ReStructured Text using Sphinx and delivered via GitHub pages.
- Hosted on GitHub with continuous integration using GitHub actions.
- High code-quality metrics for test coverage, and automated code review and security analysis.
- As few dependencies as possible, in particular being able to work with ROS but not dependent on ROS. This sidesteps problematic ROS constraints on operating system and Python versions.
- Modular approach to interfacing to different graphics libraries, simulators and physical robots.
- Support Python notebooks which allows publication of static notebooks (for example via GitHub) and interactive online notebooks (via MyBinder.org).
- Use of Unicode characters to make console output richer and easier to read.

A. Related work

There are a number of Python-based packages for robotics, each reflecting different design approaches or requirements and with various levels of finish in terms of documentation, examples, and continuous integration.

PythonRobotics [8] and Klampt [9] offer a focus on autonomous navigation and planning. While differing in scope to this work, PythonRobotics features excellent documentation and code quality, making it an exemplar.

Many packages have a focus on dynamics including iDynTree [10], Siconos [11], PyDy [12], DART [13], and PyBullet [14]. PyBullet and DART both feature graphical simulations, physics simulation, dynamical modelling, and collision detection making them useful robotics toolkits. These two packages are written in C++ with Python bindings making them fast and efficient, while still being usable in a Python environment. However, although both packages are feature-rich, they lack the ease of use and intuitive interfaces provided by 

Siconos, iDynTree, and PyDy focus on multibody dynamics for model specification, simulation and benchmarking.

The pybotics [15] robotics toolkit focusses on robot kinematics but is limited to modified Denavit–Hartenberg notation. A similar work, python-robotics [16], was created due to the author’s inspiration by RTB-M and dissatisfaction with Python-based alternatives [17]. The package, while far from an RTB-M clone or port, contains useful functionality for robotics education.

Our reinvented toolbox: The Robotics Toolbox for Python, promises to encapsulate an extensive scope of robotics, from low-level spatial-mathematics to robot arm kinematics and dynamics (regardless of model notation), and mobile robots, provide interfaces to graphical simulators and real robots, while being pythonic, well documented, and well maintained with applications in research, education, and industry.

III. Spatial Mathematics

Robotics and computer vision require us to describe position, orientation and pose in 3D space. Mobile robotics has the same requirement, but generally for 2D space. We therefore need tools to represent quantities such as rigid-body transformations (matrices ∈ SE(n) or twists ∈ se(n)), rotations (matrices ∈ SO(n) or so(n)), Euler or roll-pitch-yaw angles, or unit quaternions ∈ S3. Such capability is amongst the oldest in RTB-M and the equivalent functionality in RTB-P makes use of the Spatial Maths Toolbox for Python (SMTB-P)\(^1\).

For example

```python
>>> from spatialmath.base import *
>>> T = transl(0.5, 0.0, 0.0)
>>> rpy2tr(0.1, 0.2, 0.3, order='xyz')
>>> trotx(-90, 'deg')
```

```python
>>> print(T)
[[ 0.97517033 -0.19866933 -0.0978434 0.5 ]
 [ 0.153792 -0.93629336 0.31299183 0. ]
 [ 0.15934508 -0.93629336 0.31299183 0. ]
 [ 0. 0. 0. 1. ]]
```

There is strong similarity to the equivalent MATLAB case apart from the use of the @ operator, the use of keyword arguments instead of keyword-value pairs, and the format of the printed array. All the “classic” RTB-M functions are provided in the `spatialmath.base` package as well as additional functions for quaternions, vectors, twists and argument handling. There are also functions to perform interpolation, plot and animate coordinate frames, and create animations, using matplotlib. The underlying datatypes in all cases are 1D and 2D NumPy arrays. For a user transitioning from MATLAB the most significant difference is the use of 1D arrays – all MATLAB arrays have two dimensions, even if one of them is equal to one.

However some challenges arise when using arrays, whether native MATLAB matrices or, as in this case, NumPy arrays. Firstly, arrays are not typed by the group they represent – for example a 3 x 3 array could be an element of SE(2) or SO(3) or an arbitrary matrix.

Secondly, the operators we need for poses are a subset of those available for matrices, and some operators

\(^1\)https://github.com/petercorke/spatialmath-python
may need to be redefined in a specific way. For example, 
\( \text{SE}(3) \ast \text{SE}(3) \rightarrow \text{SE}(3) \) but \( \text{SE}(3) + \text{SE}(3) \rightarrow \mathbb{R}^{4 \times 4} \), and equality testing for a unit-quaternion has to respect the double mapping.

Thirdly, in robotics we often need to represent sets or sequences of poses, for example, all link frames on a robot or a trajectory. We could add an extra dimension to the matrices representing rigid-body transformations or unit-quaternions, or place them in a list. The first approach is cumbersome and reduces code clarity, while the second cannot ensure that all elements of the list have the same type.

We use classes and data encapsulation to address all these issues. SMTB-P provides abstraction classes \texttt{SE3, Twist3, SO3, UnitQuaternion, SE2, Twist2 and SO2}. For example, the previous example could be written as

```
>>> \texttt{from spatialmath import }*
>>> \texttt{T = SE3(0.5, 0.0, 0.0)}
>>> \texttt{* SE3.RPY([0.1, 0.2, 0.3], order='xyz')}
>>> \texttt{SE3.Rx(-90, unit='deg')}
```

where composition is denoted by the \( \ast \) operator and the matrix is printed more elegantly (and elements are color coded at the console or in ipython). \texttt{SE3.RPY()} is a class method that acts like a constructor, creating an \texttt{SE3} instance from a set of roll-pitch-yaw angles, and \texttt{SE3.Rx()} creates an \texttt{SE3} instance describing pure rotation about the \( x \)-axis. Attempts to compose with a non \texttt{SE3} instance results in a TypeError.

The orientation of the new coordinate frame may be expressed in terms of Euler angles

```
>>> \texttt{T.eul()}
array([ 95.91307496, 71.76037536, -80.34153447])
```

the rotation matrix can be easily extracted

```
>>> \texttt{T.R}
array([ 0.97517033, -0.19866933, -0.0978434 ,
       0.153792 , 0.28962948, 0.94470249,
       -0.15934508, -0.93629336, 0.31299183])
```

and we can plot the coordinate frame

```
>>> \texttt{T.plot(color='red', frame='2')}
```

Similar constructors create objects with orientation expressed in terms of an angle-vector pair, Euler vector, or orientation and approach vectors.

Rotation can also be represented by a unit quaternion

```
>>> \texttt{UnitQuaternion.Rx(0.3)}
0.988771 << 0.149438, 0.000000, 0.000000
>>> \texttt{UnitQuaternion.AngVec(0.3, [1, 0, 0])}
0.988771 << 0.149438, 0.000000, 0.000000
```

which again demonstrates several alternative constructors. The classes are generally polymorphic and have the same constructors for canonical rotations, Euler and roll-pitch-yaw angles, angle-vector, as well as a random value, \texttt{SEn.Exp()} and \texttt{SON.Exp()} is the matrix exponential constructor whose argument is the corresponding Lie algebra element.

To support trajectories each of these types inherits list properties from \texttt{collections.UserList} as shown in Figure 2. We can index the values, iterate over the values, assign to values. Some constructors take an array-like argument allowing creation of multi-valued pose objects, for example

```
>>> \texttt{R = SE3.Rx(np.linspace(0, \pi/2, num=100))}
>>> \texttt{len(R)}
100
```

where the instance \( R \) contains a sequence of 100 rotation matrices in \( \text{SE}(3) \). Composition with a single-valued (scalar) pose instance broadcasts the scalar across the sequence as shown in Figure 3.

The types all have an inverse method \texttt{.inv()} and support composition with the inverse using the / operator and integer exponentiation (repeated composition) using the \( \ast \) operator. Other overloaded operators include \( \ast, \ast\ast, \ast\ast\ast, /, /=, //= \), !=, +, -. The operator \( \ast \) acts like \( \ast \) but normalizes the result.

Supporting classes include Quaternions and Plucker (for lines in 3D). Most constructors accept a Python list or a NumPy array.

All of this allows for concise and readable code. The use of classes ensures type safety and that the matrices abstracted by the class are always valid members of the group. Operations such as addition, which are not group operations, yield a NumPy array rather than a class instance.

These benefits come at a price in terms of execution time due to the overhead of constructors and methods which wrap base functions, and type checking. The Toolbox supports SymPy which provides powerful symbolic support for Python and it works well in conjunction with NumPy, i.e. a NumPy array can contain symbolic elements. Many Toolbox methods and functions contain extra logic to ensure that symbolic operations work as expected. While this also adds to the overhead it means that for the user, working with symbols is as easy as working with numbers. Performance on a 3.6 GHz Intel Core i9 is shown in Table I.
### IV. Robotics Toolbox

#### A. Robot models

The Toolbox ships with over 30 robot models, most are purely kinematic but some have inertial and frictional parameters or graphical models. Kinematic models can be specified in a variety of ways: standard or modified Denavit-Hartenberg (DH, MDH) notation, as an elementary transform sequence [6], as a rigid-body tree, or from a URDF file.

1) Denavit-Hartenberg parameters: To create a kinematic model using DH notation, we pass a list of link objects to the DHRobot constructor. For example, an IRB140 is simply

```python
>>> from roboticstoolbox import *
>>> robot = DHRobot(
    [RevoluteDH(d=d1, a=a1, alpha=-pi/2),
     RevoluteDH(a=a2),
     RevoluteDH(alpha=pi/2),
     ...
    ], name="my IRB140")
```

where only the non-zero parameters are specified. In this case we used RevoluteDH objects for a revolute joint described using standard DH conventions. Other classes available are PrismaticDH, RevoluteMDH and PrismaticMDH. Parameters such as mass, CoG, link inertia, motor inertia, viscous friction, Coulomb friction, and joint limits can be specified using additional keyword arguments.

We can easily perform standard kinematic operations

```python
>>> puma = models.DH.Puma560()
>>> T = puma.fkine([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
>>> sol = puma.ikine_LM(T)
IKSolution(q-array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6]),
success=True, reason=None, iterations=12,
residual=8.515092622288915e-13)
>>> puma.plot(sol.q)
```

where T is an SE3 instance, and ikine_LM() is one of several generalised iterative inverse kinematic (IK) solutions [18] provided, this one based on Levenberg-Marquardt minimization. Additional status results are also returned. By default plot uses matplotlib to produce a “noodle robot” plot like that shown in Fig 1a.

The starting point for the solution may be specified, but defaults to zero, and this affects both the search time and the solution found. For a redundant manipulator there is no explicit control over the null-space. For a manipulator with \( n < 6 \) DOF an additional argument indicates which of the \( 6-n \) task-space DOF are to be unconstrained in the solution. Some robot classes have an efficient closed-form analytical solution coded, for example [19]

```python
>>> q = puma.ikine_a(T, config="lun")
```

2) ERobot + ETS notation: A Puma robot can also be specified in ETS format [6] as a sequence of simple rigid-body transformations, pure translation or pure rotation, with a constant parameter or a free parameter which is a joint variable

```python
>>> e = ET.tz(l1) * ET.rz() * ET.ty(l2) * ET.ry()
+ ET.tz(l3) * ET.tx(l4) * ET.ty(l5)
+ ET.ry() * ET.tz(l6) * ET.rrz() * ET.ry()
+ ET.rrz()
```

where l1-l6 are robot dimensions. Note there is no constraint on the joint variables being about or along the z-axis. We can promote the ETS to a robot class by

```python
>>> robot = ERobot(e)
```

An ERobot is a generalization of the robot that represents a directed-graph of ELink objects, allowing for support of branched robots.

3) ERobot + URDF import: The Toolbox includes a parser with built-in xacro processor which makes all models from the ROS universe accessible. The import is wrapped by an ERobot subclass

```python
>>> panda = models.URDF.Panda()
```

which results in an object with 3D meshes, collision objects, kinematic and dynamic data. All standard operations already introduced above are provided, but since the robot can potentially have multiple end-effectors, the name of the end-effector link must be provided to the kinematic methods.

#### B. Trajectories

A joint-space trajectory for the Puma robot from the zero-angle to the upright (or READY) joint configuration in 100 steps is

```python
>>> traj = jtraj(puma.qz, puma.qr, 100)
>>> qplot(traj.q)
```

where puma.qz is an example of a named joint configuration. traj is a trajectory object with elements \( q = q_k, \dot{q} = \dot{q}_k \) and \( \ddot{q} = \ddot{q}_k \). Each element is an array with one row per time step. The trajectory is a fifth order polynomial which has continuous jerk. Non-zero initial and final velocities may be specified.

Straight line (Cartesian) paths can be generated in a similar way between two poses in \( SE(3) \) by

```python
>>> t = np.arange(0, 2, 0.01) # time
>>> T0 = SE3(0.6, -0.5, 0.3) # initial pose
>>> T1 = SE3(0.4, 0.5, 0.2) # final pose
>>> Ts = ctraj(T0, T1, t)
>>> len(Ts)
200
```

The resulting trajectory, \( Ts \), is an SE3 instance with 200 values. For both trajectory types, the number of steps is given by an integer argument or the length of a passed time vector. Inverse kinematics can then be applied to determine the corresponding joint angle motion using

```python
>>> sol = puma.ikine_LM(Ts)
>>> sol.q.shape
(200, 6)
```

where sol is an IK solution tuple whose elements are a 2D-array of joint coordinates (one row per timestep) with the other status as either a 1D-array or list. In this case the starting joint coordinates for each inverse kinematic solution is taken as the result of the previous solution.
C. Symbolic manipulation

The Toolbox supports SymPy, for example:
```python
>>> import spatialmath.base.symbolic as sym
>>> phi, theta, psi = sym.symbol('phi theta psi')
>>> T = puma.fkine(q)
```

however the expression is more complex than it should be, because the sin α_j and cos α_j are not precisely 0, -1 or +1 due to finite precision of the value math.pi. If the α_j values are set to the symbolic value of π (sympy.S.Pi) then the expressions are greatly simplified
```python
>>> T = puma.fkine(q)
```

SymPy allows any expression to be converted to LTEX or runnable code in a variety of languages including C, Python, Octave/MATLAB.

D. Differential kinematics

The Toolbox computes Jacobians
```python
>>> J = puma.jacob0(puma.qn)
>>> J = puma.jacobe(puma.qn)
```
in the base or end-effector frames respectively, as NumPy arrays. At a singular configuration
```python
>>> J = puma.jacob0(puma.qr)
>>> np.linalg.matrix_rank(J)
5
```

Jacobians can also be computed using symbolic joint variables as for forward kinematics above.

For ERobot instances we can also compute the Hessian tensor
```python
>>> H = panda.hessian0(panda.qn)
```
in the base frame as a 3D NumPy array in \( \mathbb{R}^{6 \times n \times n} \).

For all robot classes we can compute manipulability \( m \)
```python
>>> puma.manipulability(puma.qn)
>>> puma.manipulability(puma.qn, method="asada")
```
for the Yoshikawa [20] and Asada [21] measures, and
```python
>>> puma.manipulability(puma.qn, axes="trans")
```
is the Yoshikawa measure computed for just the task-space translational degrees of freedom. For ERobot instances we can also compute the manipulability Jacobian
```python
>>> Jm = panda.jacobm(panda.qr)
```
such that \( \dot{m} = \dot{J}_m(q)q \).

E. Dynamics

The new Toolbox supports several approaches to computing dynamics. For models defined using standard or modified DH notation we use a classical version of the recursive Newton-Euler [22] algorithm implemented in Python or C.

For example, the inverse dynamics
```python
>>> tau = puma.rne(puma.qn, puma.qz, puma.qz)
```
is the gravity torque for the robot in the configuration qn.

Inertia, Coriolis/centripetal and gravity terms are computed respectively by
```python
>>> I = puma.inertial(puma.qn)
>>> C = puma.coriolis(puma.qn, qd)
>>> g = puma.gravity(puma.qn)
```
from the inverse dynamics using the method of [23]. Forward dynamics are given by
```python
>>> qdd = puma.accel(q, tau, qd)
```
which we can integrate over time
```python
>>> q = puma.fdyn(5, q0, mycontrol, ...)
```
uses an RK45 numerical integration from the SciPy package to solve for the joint trajectory \( \dot{q} \) given the joint-space control function called with the signature
```python
tau = mycontrol(robot, t, q, qd, **args)
```

The fast C implementation is not capable of symbolic operation but the slower Python version rne.py is. For a 6- or 7-DoF manipulator the torque expressions have thousands of terms yet are computed in less than a second. However, subsequent expression manipulation is slow, and this is an area of future work, as is the automatic generation of efficient run-time code for manipulator dynamics.

For the Puma 560 robot, the C version of inverse dynamics takes 23 \( \mu \)s while the Python version takes 1.5 ms (65x slower). With symbolic operands it takes 170 ms (113x slower) to produce the unsimplified torque expressions.

SMTB-P provides a set of classes for spatial velocity, acceleration, momentum, force and inertia to support Featherstone’s spatial vector approach to robot dynamics [5].

V. New capability

There are several areas of innovation compared to the MATLAB version of the Toolbox.

A. Branched mechanisms

The RTB-M SerialLink class had no option to express branching. The equivalent RTB-P class DHRobot is similarly limited, but a new class ERobot is more general and allows for branching (but not closed kinematic loops). The robot is described by a set of ELink objects, each of which points to its parent link. The ERobot has references to the root and leaf ELinks. This structure closely mirrors the URDF representation, allowing easy import of URDF models.

2The same code as used by RTB-M is called directly from Python, and does not use NumPy.
B. Collision checking

RTB-M had a simple, contributed but unsupported, collision checking capability. This is dramatically improved in the Python version using PyBullet [14] which supports primitive shapes such as Cylinders, Spheres and Boxes as well as mesh objects. Every robot ELink can have a collision shape in addition to the shape used for rendering. We can conveniently perform collision checks between links as well as between whole robots, discrete links, and objects in the world. For example a 1x1x1 box centered at (1,0,0) can be tested against all, or just one link, of the robot by

```python
>>> obstacle = rtb.Box([1, 1, 1], SE3(1, 0, 0))
>>> collision = panda.collided(obstacle)
```

Additionally, we can compute the minimum Euclidean distance between whole robots, discrete links, or objects

```python
>>> d, p1, p2 = panda.closest_point(obstacle)
>>> d, p1, p2 = \n    panda.links[0].closest_point(obstacle)
```

which returns the shortest line segment expressed as a length and two endpoints – the coordinates of the closest points on each of the two bodies. As an example, the work in [24] uses this feature for reactive collision avoidance for manipulators.

C. Interfaces

RTB-M could only animate a robot in a figure, and there was a limited interface to V-REP and a physical robot. The Python version supports a simple, but universal API to a robot inspired by the simplicity and expressiveness of the OpenAI Gym API [25] which was designed as a toolkit for developing and comparing reinforcement learning algorithms. Whether simulating a robot or controlling a real physical robot, the API operates in the same manner.

By default the Toolbox behaves like the MATLAB version with a plot method

```python
>>> panda.plot(panda.qr)
```

which will plot the robot at the specified joint configuration, or animate it if q is a multi-row matrix. This uses the default PyPlot backend which draws a “noodle robot” using matplotlib similar to that shown in Figure 1a.

The more general solution, and what is implemented inside plot in the example above, is

```python
>>> from roboticstoolbox.backends.Swift \n    import Swift
>>> backend = Swift()
>>> backend.launch() # create graphical scene
>>> backend.add(panda) # add robot to the scene
>>> panda.q = panda.qr # update the robot
>>> backend.step() # display the scene
```

which will open a new browser tab and render the robot. This interface makes it possible to animate multiple robots in the one graphical window, or the one robot in various environments either graphical or real.

The Swift, see Fig. 1c, and VPython, see Fig. 1b, backends provides browser-based 3D graphics based on WebGL. This is advantageous for displaying on mobile devices or from a headless control computer. Swift uses three.js to provide high-quality 3D animations and can produce vivid 3D effects using anaglyphs and a Looking Glass light-field (holographic) display3 for glasses-free 3D viewing. Animations can be recorded as MP4 files or animated GIF files (which are useful for inclusion in GitHub markdown documents).

VI. Code engineering

The code is implemented in Python ≥ 3.6 and hosted on GitHub with unit-testing performed using GitHub-actions. Test coverage is uploaded to codecov.io for visualization and trending, and we use lgtm.com to perform automated code review. The code is documented with ReStructured Text format docstrings which provides powerful markup including cross-referencing, equations, class inheritance diagrams and figures – all of which is converted to HTML documentation whenever a change is pushed, and this is accessible via GitHub pages. Issues can be reported via GitHub issues or patches submitted as pull requests.

RTB-P4, and its dependencies, can be installed by either

```
$ pip install roboticstoolbox-python
$ conda install -c conda-forge \n    roboticstoolbox-python
```

which includes basic visualization using matplotlib. Options such as swift.vpython can be used to specify additional dependencies to be installed.

The Toolbox adopts a “when needed” approach to many dependencies. This reduces Toolbox import time, since it will only attempt to import a dependency if the user exploits a functionality that requires it. If a required dependency is not installed a warning provides instructions on how to install it. This approach applies to the visualizers VPython and Swift, as well as pybullet and ROS. The Toolbox provides capability to import URDF-xacro files without ROS. The backend architecture allows a user to connect to a ROS environment if required, and only then does ROS have to be installed.

VII. Conclusion

This paper has introduced and demonstrated in tutorial form the principle features of the Robotics Toolbox for Python which runs on Mac, Linux and Windows. The code examples are provided with the Toolbox as icra2021.ipynb. The Toolbox is free and open, and released under the MIT licence. It provides many of the essential tools necessary for robotic manipulator modelling, simulation and control which is essential for robotics education and research. It is familiar yet new, and we hope it will serve the community well for the next 25 years.

Currently under development are backend interfaces for CoppeliaSim and Dynamixel servo chains; symbolic dynamics; simplification and code generation; spatial vector dynamics; mobile robotics motion models, planners, EKF localization, map making and SLAM; and a minimalistic block-diagram simulation tool5.

3https://lookingglassfactory.com
4https://github.com/petercorke/robotics-toolbox-python
5https://github.com/petercorke/bdsim
REFERENCES

[1] P. Corke, “A computer tool for simulation and analysis: the Robotics Toolbox for MATLAB,” in Proc. National Conf. Australian Robot Association, Melbourne, Jul. 1995, pp. 319–330.

[2] P. Corke, “A robotics toolbox for MATLAB,” IEEE Robotics and Automation Magazine, vol. 3, no. 1, pp. 24–32, Sep. 1996.

[3] P. I. Corke, Robotics, Vision & Control: Fundamental Algorithms in MATLAB, 2nd ed. Springer, 2017, ISBN 978-3-319-54412-0.

[4] J. Craig, Introduction to Robotics: Mechanics and Control, ser. Addison-Wesley series in electrical and computer engineering: control engineering. Pearson/Prentice Hall, 2005. [Online]. Available: https://books.google.com.au/books?id=MqMeAQAIAAJ

[5] R. Featherstone, Robot Dynamics Algorithms. Kluwer Academic, 1987.

[6] P. I. Corke, “A simple and systematic approach to assigning Denavit-Hartenberg parameters,” IEEE transactions on robotics, vol. 23, no. 3, pp. 590–594, 2007.

[7] J. Haviland and P. Corke, “A systematic approach to computing the manipulator Jacobian and Hessian using the elementary transform sequence,” arXiv preprint, 2020.

[8] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques, “PythonRobotics: a Python code collection of robotics algorithms,” arXiv preprint arXiv:1808.10703, 2018.

[9] K. Hauser, “Klampt module.” [Online]. Available: https://github.com/krisinhauser/Klampt

[10] M. W. Walker and D. E. Orin, “Efficient dynamic computer simulation of robotic mechanisms,” ASME Journal of Dynamic Systems, Measurement and Control, vol. 104, no. 3, pp. 205–211, 1982.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI gym,” arXiv preprint arXiv:1606.01540, 2016.