

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

Peter Corke<sup>1</sup>, Jesse Haviland<sup>1</sup>

**Abstract**—For 25 years the Robotics Toolbox for MATLAB<sup>®</sup> 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-browseable 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 CoppeliaSim, ROS and Dynamixel servomotors. 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.

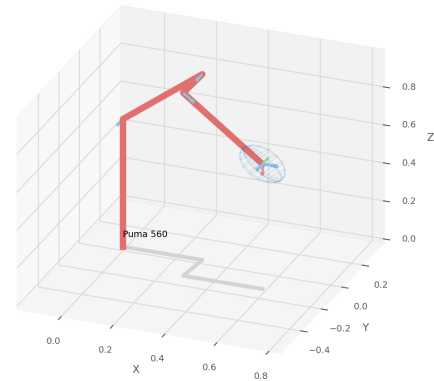
## I. INTRODUCTION

The Robotics Toolbox for MATLAB<sup>®</sup> (RTB-M) was created around 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 and improvements to the MATLAB language and ecosystem, such as the addition of structures, objects, lists (cell arrays) and strings, myriad other improvements to the language, new graphics and new tools such as IDE, debugger, notebooks (LiveScripts), apps and continuous integration. An adverse consequence is that many poor early design decisions hinder development.

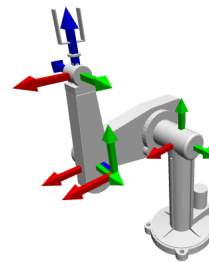
Over time additional functionality was also added, in particular for vision, and two major refactorings led to the current state of three toolboxes: Robotics Toolbox for MATLAB<sup>®</sup> and Machine Vision Toolbox for MATLAB<sup>®</sup> (1999) both of which are built on the Spatial Math Toolbox for MATLAB<sup>®</sup> (SMTB-M) in 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

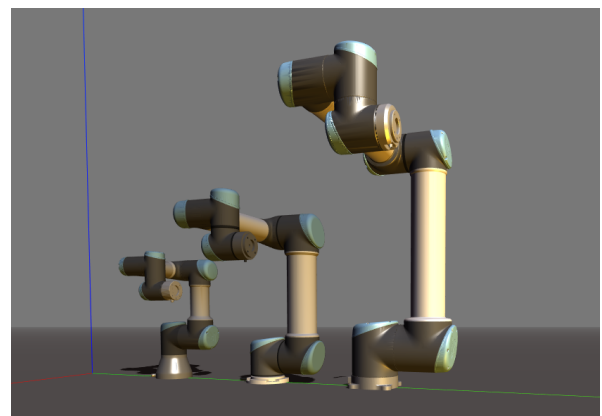
<sup>1</sup>Peter Corke and Jesse Haviland are with the Australian Centre for Robotic Vision (ACRV), Queensland University of Technology Centre for Robotics (QCR), Brisbane, Australia [peter.corke@qut.edu.au](mailto:peter.corke@qut.edu.au), [j.haviland@qut.edu.au](mailto:j.haviland@qut.edu.au). This research was conducted by the Australian Research Council project number CE140100016, and supported by the QUT Centre for Robotics.



(a) Puma560, with a velocity ellipsoid, rendered using the default matplotlib visualizer.



(b) Puma560 rendered using the web-based VPython visualizer.



(c) The Universal Robot family rendered using the Toolbox's Swift visualizer.

Fig. 1: The toolbox provides several visualizers

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 lack of ongoing resources to complete a sufficient subset of functionality. Subsequent attempts have all met the same fate.

The design goals of this version can be summarised as new functionality:

- A superset of the MATLAB<sup>®</sup> Toolbox functionality
- Build on the Spatial Math Toolbox for Python which provides objects to represent rotations as matrices in  $\mathbf{SO}(2)$  and  $\mathbf{SO}(3)$  as well as unit-quaternions, rigid-body motions as matrices  $\mathbf{SE}(2)$  and  $\mathbf{SE}(3)$  as well as twists in  $\mathfrak{se}(2)$  and  $\mathfrak{se}(3)$ , and Featherstone’s spatial vectors [5].
- Support models expressed using Denavit-Hartenberg notation (standard and modified), elementary transform sequences [6], [7], and URDF-style rigid-body trees. 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 be dependent on ROS. This sidesteps 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 (MyBinder.org).
- Use of Unicode characters to make console output 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 *pythonic* development-driven applications. 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 a RTB-M clone or conversion, 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  $\in \mathbf{SE}(n)$  or twists  $\in \mathfrak{se}(n)$ ), rotations (matrices  $\in \mathbf{SO}(n)$  or  $\mathfrak{so}(n)$ , Euler or roll-pitch-yaw angles, or unit quaternions  $\in S^3$ ). 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)<sup>1</sup>. For example

```
>>> from spatialmath.base import *
>>> T = transl(0.5, 0.0, 0.0)
      @ rpy2tr(0.1, 0.2, 0.3, order='xyz')
      @ trotx(-90, 'deg')
>>> print(T)
[[ 0.97517033 -0.19866933 -0.0978434  0.5   ]
 [ 0.153792   0.28962948  0.94470249  0.   ]
 [-0.15934508 -0.93629336  0.31299183  0.   ]
 [ 0.         0.         0.         1.   ]]
```

There is strong similarity to the equivalent MATLAB<sup>®</sup> 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 movies, 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<sup>®</sup> matrices or, as in this case, NumPy arrays. Firstly, arrays are not typed and for example

<sup>1</sup><https://github.com/petercorke/spatialmath-python>

a  $3 \times 3$  array could be an element of  $\mathbf{SE}(2)$  or  $\mathbf{SO}(3)$  or an arbitrary matrix.

Secondly, the operators we need for poses are a subset of those available for matrices, and some operators may need to be redefined in a specific way. For example,  $\mathbf{SE}(3) * \mathbf{SE}(3) \rightarrow \mathbf{SE}(3)$  but  $\mathbf{SE}(3) + \mathbf{SE}(3) \rightarrow \mathbb{R}^{4 \times 4}$ , and equality testing for a unit-quaternion has to respect the double mapping.

Thirdly, in robotics we often need to represent time sequences of poses. 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 `SE3`, `Twist3`, `SO3`, `UnitQuaternion`, `SE2`, `Twist2` and `SO2`. For example, the previous example could be written as

```
>>> from spatialmath import *
>>> T = SE3(0.5, 0.0, 0.0)
      * SE3.RPY([0.1, 0.2, 0.3], order='xyz')
      * SE3.Rx(-90, unit='deg')
>>> print(T)
0.97517    -0.198669   -0.0978434   0.5
0.153792    0.289629    0.944702    0
-0.159345   -0.936293    0.312992    0
0           0           0           1
```

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

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

```
>>> T.eul()
array([ 95.91307498,  71.76037536, -80.34153447])
```

the rotation matrix can be easily extracted

```
>>> 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

```
>>> T.plot(color='red', label='2')
```

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

Rotation can also be represented by a unit quaternion

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

which again demonstrates several alternative constructors. The classes are somewhat polymorphic and have the same constructors for canonic rotations, Euler and roll-pitch-yaw angles, angle-vector, as well as a random value.  $\mathbf{SE}(n)$  and  $\mathbf{SO}(n)$  also support a matrix exponential constructor where the argument is the corresponding Lie algebra element.

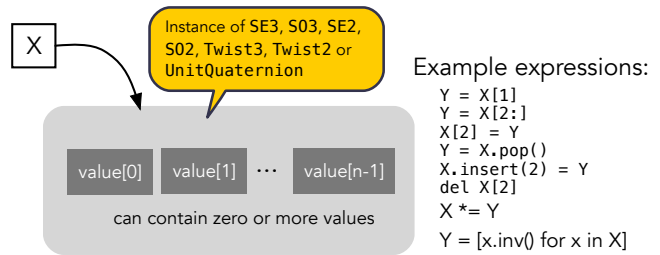


Fig. 2: Any of the SMTB-P pose classes can contain a list of values

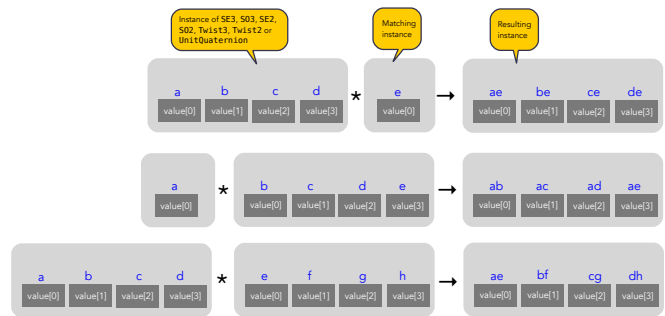


Fig. 3: Overloaded operators support broadcasting.

To support trajectories each of these types inherits list properties from `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

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

where the instance `R` contains a sequence of 100 rotation matrices in  $\mathbf{SE}(3)$ . 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 `.inv()` and support composition with the inverse using the `/` operator and integer exponentiation (repeated composition) using the `**` operator. Other overloaded operators include `*`, `*=`, `**`, `**=`, `/`, `/=`, `==`, `!=", +, -`. Supporting classes include `Quaternion` and `Plucker` (for lines in 3D).

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, ie. a NumPy array can contain symbolic elements. Many the Toolbox methods and functions contain extra logic to ensure that symbolic operations work as expected. While this adds to the overhead it means that for the user, working with

Function/method	Execution time
<code>base.rotx()</code>	4.07 $\mu$ s
<code>base.trotx()</code>	5.79 $\mu$ s
<code>SE3.Rx()</code>	12.3 $\mu$ s
<code>SE3 * SE3</code>	4.69 $\mu$ s
<code>4x4 @</code>	0.986 $\mu$ s
<code>SE3.inv()</code>	7.62 $\mu$ s
<code>base.trinv()</code>	4.19 $\mu$ s
<code>np.linalg.inv()</code>	4.49 $\mu$ s

TABLE I: Spatial math execution performance

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 ETS string [6], as a rigid-body tree, or from a URDF file.

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

```
>>> 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 need to be 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`. Other parameters such as mass, CoG, link inertia, motor inertia, viscous friction, Coulomb friction, and joint limits can also be specified using additional keyword arguments.

We can easily perform standard kinematic operations

```
>>> puma = rtb.models.DH.Puma560()
>>> T = puma.fkine([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
>>> q, *_ = puma.ikine(T)
>>> puma.plot(q)
```

`ikine()` is a generalised iterative inverse kinematic (IK) solution [18] based on Levenberg-Marquadt minimization, and additional status results are also returned. The default plot interface, using matplotlib, produces a “noodle robot” plot like that shown in Fig 1a.

The starting point for the solution may be specified, but defaults to zero, and affects both the search time and the solution found, since in general a manipulator may have several joint configurations which result in the same end-effector pose. For a redundant manipulator there is no explicit control over the null-space. For a manipulator with  $n < 6$  DOF an additional argument is required to indicate which of the  $6 - n$  task-space DOF are to be unconstrained in the solution.

The IK procedure for any specific robot can be derived symbolically [19] and commonly an efficient closed-form solution can be obtained. Some robot classes have an analytical solution coded, for example

```
>>> 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

```
>>> e = ET.tz(11) * ET.rz() * ET.ty(12) * ET.ry()
      * ET.tz(13) * ET.tx(16) * ET.ty(14) * ET.ry()
      * ET.tz(15) * ET.rz() * ET.ry() * ET.rz()
```

```
>>> robot = SerialLink(e)
```

and can represent single-branched robots with any combination of revolute and prismatic joints.

3) *ERobot + URDF import*: The final approach is to import a URDF file. The Toolbox includes a parser with built-in xacro processor which makes all models from the ROS *universe* accessible.

More complex models such as for Panda or Puma are defined by classes but built this way

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

and comprise a list or tree of `ELink` objects. Standard operations such as forward and inverse kinematics, and plotting function just like the example above. If the URDF model includes Collada graphics models, then Swift is able to produce a 3-D visualization like that shown in Figure 1a. The `ERobot` class supports branched robots, with multiple end-effectors, so the name of the 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

```
>>> traj = rtb.jtraj(puma.qz, puma.qr, 100)
>>> rtb.qplot(traj.j)
```

where `puma.qr` is an example of a named joint configuration. `traj` is a named tuple with elements  $\mathbf{q} = \mathbf{q}_k$ ,  $\mathbf{q}\dot{\mathbf{d}} = \dot{\mathbf{q}}_k$  and  $\mathbf{q}\ddot{\mathbf{d}} = \ddot{\mathbf{q}}_k$ . Each element is an array with one row per time step, and each row a joint coordinate vector. The trajectory is a fifth order polynomial which has continuous jerk. By default, the initial and final velocities are zero, but these may be specified by additional arguments.

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

```
>>> t = np.arange(0, 2, 0.010)
>>> T0 = SE3(0.6, -0.5, 0.0)
>>> T1 = SE3(0.4, 0.5, 0.2)
>>> 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

```
>>> qs, *_ = puma.ikine(Ts)
>>> qs.shape
(200, 6)
```



where `qs` is an array of joint coordinates, one row per timestep. In this case the starting joint coordinates for each inverse kinematic solution is taken as the result of the previous solution.

### C. Symbolic manipulation

As mentioned earlier, the Toolbox supports SymPy. For example:

```
>>> import roboticstoolbox.base.symbolics as sym
>>> phi, theta, psi = sym.symbol('phi, theta, psi')
>>> rpy2r(phi, theta, psi)
array([[cos(psi)*cos(theta),
        sin(phi)*sin(theta)*cos(psi) - sin(psi)*cos(phi),
        sin(phi)*sin(psi) + sin(theta)*cos(phi)*cos(psi)],
        ...])
```

This capability readily extends to forward kinematics

```
>>> q = sym.symbol("q:6") # q = (q_1, q_2, ... q_5)
>>> T = puma.fkine(q)
```

however the expression is more complex than it should be, because the  $\sin \alpha_j$  and  $\cos \alpha_j$  are not precisely 0, -1 or +1 due to finite precision of the value `math.pi`. If the  $\alpha_j$  values are set to the symbolic value of  $\pi$  (`sympy.S.Pi`) then the expressions are greatly simplified

```
>>> puma = rtb.models.DH.Puma560(symbolic=True)
>>> T = puma.fkine(q)
```

If we display the value of `puma` we see that the  $\alpha_j$  values are now displayed in red to indicate that they are symbolic constants. The x-coordinate of the end-effector is

```
>>> T.t[0]
0.15005*sin(q_0) - 0.0203*sin(q_1)*sin(q_2)*cos(q_0)
- 0.4318*sin(q_1)*cos(q_0)*cos(q_2)
- 0.4318*sin(q_2)*cos(q_0)*cos(q_1)
+ 0.0203*cos(q_0)*cos(q_1)*cos(q_2)
+ 0.4318*cos(q_0)*cos(q_1)
```

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

### D. Differential kinematics

The Toolbox computes Jacobians

```
>>> J = puma.jacob0(q)
>>> J = puma.jacobe(q)
```

in the base or end-effector frames respectively, as NumPy arrays. At a singular configuration

```
>>> J = puma.jacob0(puma.qr)
>>> np.linalg.matrix_rank(J)
5
>>> jsingu(J)
joint 5 is dependent on joint 3
```

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

For `ERobot` instances we can also compute the Hessians

```
>>> H = puma.hessian0(q)
>>> H = puma.hessiane(q)
```

in the base or end-effector frames respectively, as 3D NumPy arrays in  $\mathbb{R}^{6 \times n \times n}$ .

For all robot classes we can compute manipulability

```
>>> m = puma.manipulability(q)
>>> m = puma.manipulability(q, "asada")
```

for the Yoshikawa [20] and Asada [21] measures, and

```
>>> m = puma.manipulability(q, 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

```
>>> Jm = puma.manipm(q, J, H)
```

such that  $\dot{m} = J_m(q)\dot{q}$ .

### 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<sup>2</sup>. For example, the inverse dynamics

```
>>> tau = puma.rne(puma.qn, np.zeros((6,)), np.zeros((6,)))
```

is the gravity torque for the robot in the configuration `qn`.

Inertia, Coriolis/centripetal and gravity terms are computed by

```
>>> J = puma.inertia(q)
>>> C = puma.coriolis(q, qd)
>>> g = puma.gravload(q)
```

respectively, using the method of [23] from the inverse dynamics. These values include the effect of motor inertia and friction.

Forward dynamics are given by

```
>>> qdd = puma.accel(q, tau, qd)
```

which we can integrate over time

```
>>> q = puma.fdyn(5, q0, mycontrol, ...)
```

uses an RK45 numerical integration from the SciPy package to solve for the joint trajectory `q` given the joint-space control function called as

```
tau = mycontrol(robot, t, q, qd, **args)
```

The fast C implementation is not capable of symbolic operation so a Python version of RNE `rne_python` has been implemented as well. 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 the best strategy is to eliminate the least significant terms and this typically gets the expression for the first joint to a hundred or so terms which is quite manageable. This is an area of active 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  $23 \mu\text{s}$  while the Python version takes 1.5 ms ( $65 \times$  slower). With symbolic operands it takes 170 ms ( $113 \times$  slower) to produce the unsimplified torque expressions.

For all robots there is also an implementation of Featherstone's spatial vector method, `rne_spatial`, and SMTB-P provides a set of classes for spatial velocity, acceleration, momentum, force and inertia.

## V. NEW CAPABILITY

There are several areas of innovation compared to the MATLAB<sup>®</sup> version of the Toolbox.

### A. Branched mechanisms

The RTB-M `SerialLink` class had no option to express branching. In RTB-P the equivalent 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

<sup>2</sup>The same code as used by RTB-M is called directly from Python, and does not use NumPy.

to the root and leaf `ELinks`. This structure closely mirrors the URDF representation, allowing for easy import of URDF models.

### 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` has 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  $1 \times 1 \times 1$  box centered at  $(1, 0, 0)$  can be tested against all, or just one link, of the robot by

```
>>> panda = rtb.models.Panda()
>>> obstacle = rtb.Box([1, 1, 1], SE3(1, 0, 0))
>>> iscollision = panda.collided(obstacle) # boolean
>>> iscollision = panda.links[0].collided(obstacle)
```

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

```
>>> d, p1, p2 = panda.closest_point(obstacle)
>>> d, p1, p2 = 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.

### C. Interfaces

RTB-M could only animate a robot in a figure, and there was limited but not-well-supported ability to 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 [24] 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<sup>®</sup> version with a plot method

```
>>> puma.plot(q)
```

which will plot the robot at the specified joint configuration, or animate it if `q` is an  $m \times 6$  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

```
>>> pyplot = roboticstoolbox.backends.PyPlot()
>>> pyplot.launch()
>>> pyplot.add(puma)
>>> puma.q = q
>>> puma.step()
```

This makes it possible to animate multiple robots in the one graphical window, or the one robot in various environments either graphical or real.

The `VPython` backend, see Fig. 1b, provides browser-based 3D graphics based on `WebGL`. This is advantageous for displaying on mobile devices. `Swift`, see Fig. 1c, is an Electron app that uses `three.js` to provide high-quality 3D animations. It can produce vivid 3D effects using anaglyphs viewed with colored glasses, and we also adapted it to work

with a Looking Glass light-field (holographic) display<sup>3</sup> 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  $\geq 3.6$  and all code is hosted on GitHub and unit-testing is 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-P, and its dependencies, can be installed simply by

```
$ pip install roboticstoolbox-python
```

which includes basic visualization using `matplotlib`. Options such as `vpython` can be used to specify additional dependencies to be installed. The Toolbox adopts a “when needed” approach to many dependencies and will only attempt to import them if the user attempts to exploit a particular functionality. If the dependency is not installed a warning provides instructions on how to install them using `pip` and/or `npm`. More details are given on the project home page.<sup>4</sup> This 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, Windows and Linux. The code 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`, `Dynamixel` servo chains, and `ROS`; symbolic dynamics, simplification and code generation; mobile robotics motion models, planners, EKF localization, map making and SLAM; and a minimalist block-diagram simulation tool<sup>5</sup>.

<sup>3</sup><https://lookingglassfactory.com>

<sup>4</sup><https://github.com/petercorke/robotics-toolbox-python>

<sup>5</sup><https://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] —, "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=MqMeAQAAIAAJ>
- [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/krishhauser/Klampt>
- [10] F. Nori, S. Traversaro, J. Eljaik, F. Romano, A. Del Prete, and D. Pucci, "iCub whole-body control through force regulation on rigid noncoplanar contacts," *Frontiers in Robotics and AI*, vol. 2, no. 6, 2015. [Online]. Available: [http://www.frontiersin.org/humanoid\\_robotics/10.3389/frobt.2015.00006/abstract](http://www.frontiersin.org/humanoid_robotics/10.3389/frobt.2015.00006/abstract)
- [11] "Siconos." [Online]. Available: <https://github.com/siconos/siconos>
- [12] G. Gede, D. L. Peterson, A. S. Nanjangud, J. K. Moore, and M. Hubbard, "Constrained multibody dynamics with Python: From symbolic equation generation to publication," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 55973. American Society of Mechanical Engineers, 2013, p. V07BT10A051.
- [13] J. Lee, M. X. Grey, S. Ha, T. Kunz, S. Jain, Y. Ye, S. S. Srinivasa, M. Stilman, and C. K. Liu, "DART: Dynamic animation and robotics toolkit," *The Journal of Open Source Software*, vol. 3, no. 22, p. 500, Feb 2018. [Online]. Available: <https://doi.org/10.21105/joss.00500>
- [14] E. Coumans and Y. Bai, "PyBullet, a Python module for physics simulation for games, robotics and machine learning," <http://pybullet.org>, 2016–2019.
- [15] N. Nadeau, "Pybotics: Python toolbox for robotics," *Journal of Open Source Software*, vol. 4, no. 41, p. 1738, Sep. 2019. [Online]. Available: <https://doi.org/10.21105/joss.01738>
- [16] R. Fraanje, "Python Robotics module." [Online]. Available: <https://github.com/prfraanje/python-robotics>
- [17] R. Fraanje, T. Koreneef, A. Le Mair, and S. de Jong, "Python in robotics and mechatronics education," in *2016 11th France-Japan & 9th Europe-Asia Congress on Mechatronics (MECATRONICS)/17th International Conference on Research and Education in Mechatronics (REM)*. IEEE, 2016, pp. 014–019.
- [18] S. Chiaverini, L. Sciavicco, and B. Siciliano, "Control of robotic systems through singularities," in *Advanced Robot Control, Proceedings of the International Workshop on Nonlinear and Adaptive Control: Issues in Robotics*, ser. Lecture Notes in Control and Information Sciences, vol. 162. Grenoble: Springer, 1991, pp. 285–295.
- [19] R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.
- [20] T. Yoshikawa, "Analysis and control of robot manipulators with redundancy," in *Robotics Research: The First International Symposium*, M. Brady and R. Paul, Eds. The MIT press, 1984, pp. 735–747.
- [21] H. Asada, "A geometrical representation of manipulator dynamics and its application to arm design," *Journal of Dynamic Systems, Measurement, and Control*, vol. 105, p. 131, 1983.
- [22] W. Armstrong, "Recursive solution to the equations of motion of an n-link manipulator," in *Proc. 5th World Congress on Theory of Machines and Mechanisms*, Montreal, Jul. 1979, pp. 1343–1346.
- [23] 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.
- [24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *arXiv preprint arXiv:1606.01540*, 2016.