

Block Diagram Simulation

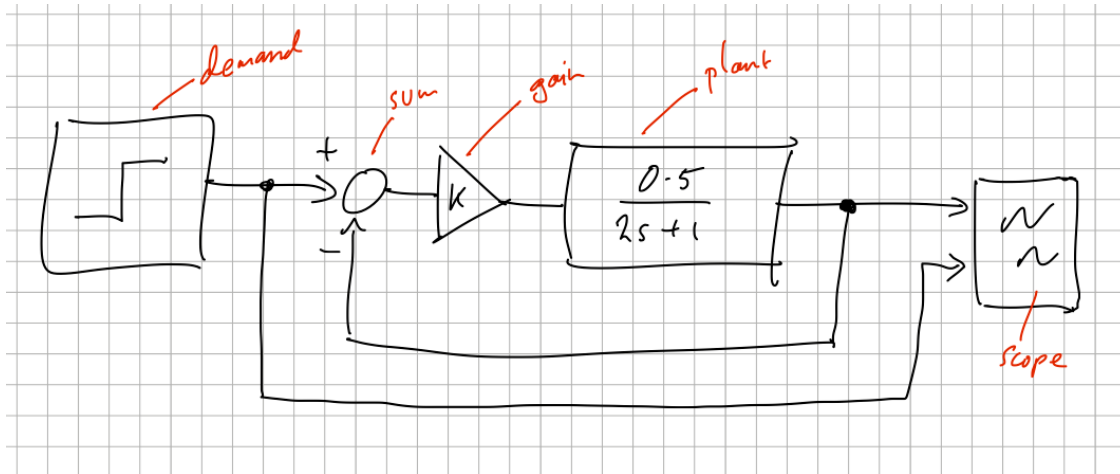
Peter Corke

June 10, 2026

```
[2]: import bdsim
import matplotlib.pyplot as plt
```

1 A first-order system

Here's a hand drawn sketch of a simple first-order system with proportional feedback



We model it using bdsim by writing Python code to represent the model. Python means you can use your favourite IDE, debugger, and version control.

```
[3]: sim = bdsim.BDSim(animation=True) # create simulator
bd = sim.blockdiagram() # create an empty block diagram

# define the blocks
demand = bd.STEP(T=1, name="demand")
sum = bd.SUM("+ -")
gain = bd.GAIN(10)
plant = bd.LTI_SISO(0.5, [2, 1], name="plant")
scope = bd.SCOPE(styles=["k", "r--"], loc="lower right")

# connect the blocks
bd.connect(demand, sum[0], scope[1])
bd.connect(plant, sum[1])
bd.connect(sum, gain)
```

```
bd.connect(gain, plant)
bd.connect(plant, scope[0])
```

Before we can run the code, we need to compile it. That does a bunch of checks and builds the datastructures required for execution.

```
[4]: bd.compile(verbose=True) # check the diagram
```

```
Compiling blockdiagram 'main':
  checking wires and connections...
  checking block parameters...
  checking all stateful blocks...
  connecting wires to blocks...
  checking block inputs/outputs are connected...
  checking for algebraic loops...
  creating execution schedule...
  create slots to transfer data between blocks...
  evaluating network with initial conditions to determine wire datatype...
```

```
[4]: True
```

and this is a succinct representation of the model, showing each block and where it's inputs are sourced from.

```
[5]: bd.report_summary()
```

block	nc	nd	type	inport	source	source type
demand@	0	0	step			
gain.0	0	0	gain	0	sum.0	float
plant	1	0	lti_asiso	0	gain.0	float
scope.0	0	0	scope	0 1	plant demand	float int
sum.0	0	0	sum	0 1	demand plant	int float

Note: @ = event source

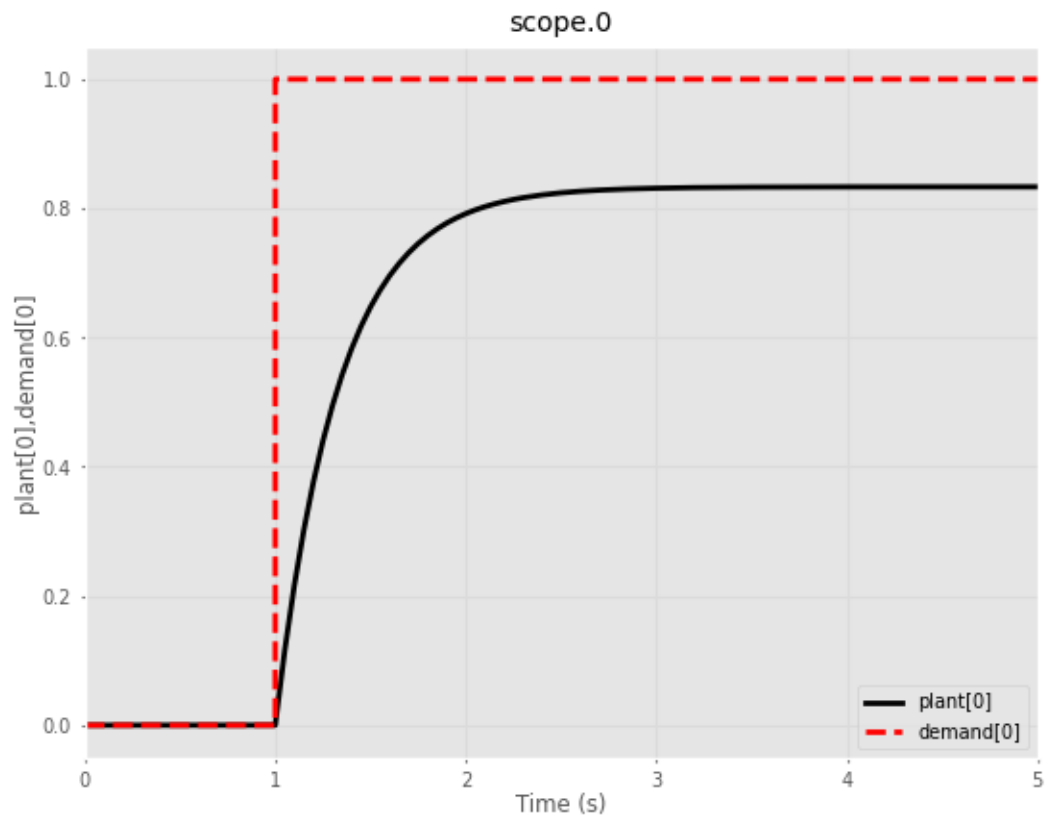
Now we can run the model. In this case for 5 seconds, and we'll see the output of the SCOPE block. We will also record the outputs of the demand and sum blocks.

```
[6]: out = sim.run(bd, T=5, watch=[demand, sum]) # simulate for 5s
```

```
>>> Start simulation: T = 5, dt = None, max_step = 0.05
```

```
Hybrid system solver: 1 continuous state variable, 0 discrete state variables
```

```
x0 = [0.]
```



```
<<< Simulation complete

block diagram evaluations: 1782

  of which ydot calls:      1562  (remaining are post-integration replay + sink
passes)

bd.evaluate() mean time:    59.7 us/call  (total 106.4 ms)

clocktime speedup:         0.82x  (simulated 5s in 6072.3 ms)

integration time points:    220

scheduled event intervals: 100  (each bounded by a clock tick, scheduled
event, or terminal marker)

integrator wall time:       0.133 s  (solve_ivp only; excludes post-
integration replay)

zero-crossing events:       0
```

```
[7]: print(out)
```

```
t      = ndarray:float64 (220,)
x      = ndarray:float64 (220, 1)
xnames = ['plant:x_0'] (list)
y      = ndarray:float64 (220, 2)
ynames = ['demand', 'sum.0'] (list)
```

```
[8]: from IPython.display import Markdown, display
```

```
mermaid = bd.graph(format="mermaid") # export a graph in Mermaid format
display(Markdown(f"```` mermaid\n{mermaid}\n````")) # Render dynamically via
↳ Jupyter's built-in Mermaid engine
```

```
graph LR
    b0["demand"]
    b1["sum.0"]
    b2["gain.0"]
    b3["plant"]
    b4["scope.0"]
    b0 --> "+" b1
    b0 --> b4
    b3 --> "-" b1
    b1 --> b2
    b2 --> b3
    b3 --> b4
```

2 The “bouncing ball” problem

This is a classic hybrid simulation problem with continuous dynamics (the ball flight) and discrete dynamics (the impact).

```
[9]: import bdsim

# define constants
g = -9.81 # gravity m/s2
e = 0.8 # coefficient of restitution
h0 = 10 # initial height m

sim = bdsim.BDSim(animation=True) # create simulator
bd = sim.blockdiagram() # create an empty block diagram

# define the blocks
gravity = bd.CONSTANT(g, name="gravity")

# chain of integrators for ball flight dynamics
velocity = bd.INTEGRATOR(name="velocity", x0=0) # initial height is initial
    ↳ velocity
position = bd.INTEGRATOR(name="position", x0=h0)

scope = bd.SCOPE(styles=["k", "r--"], loc="lower right")

# ----- the impact event detection and response -----
    ↳ #
# detect when the ball hits the ground (position=0) and trigger an event
def impact(event_block, state_map):
    state_map[velocity] *= -e # reverse velocity and apply restitution

# connect the impact function to the event triggered when position crosses zero
    ↳ from above
ground = bd.EVENT("-", impact)

# -----
    ↳ #

# connect the blocks together
bd.connect(gravity, velocity)
bd.connect(velocity, position)
bd.connect(velocity, scope[0])
bd.connect(position, scope[1])
bd.connect(position, ground)

bd.compile() # check the diagram
bd.report_summary()
```

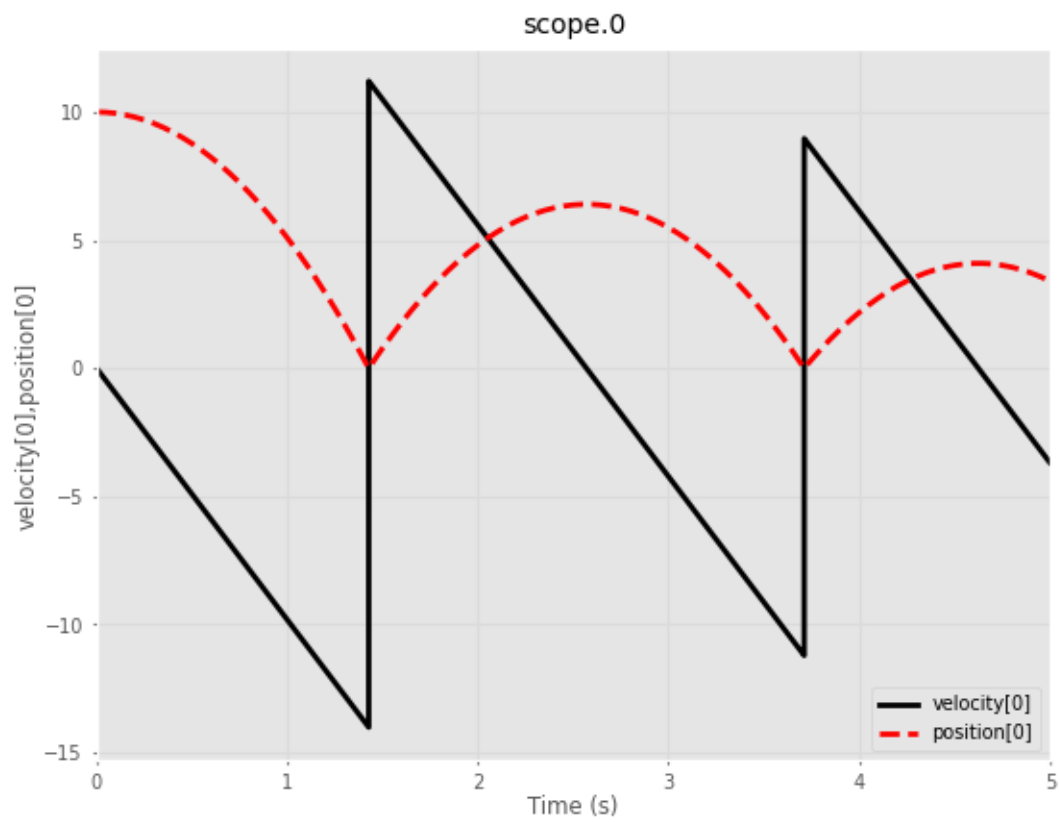
block	nc	nd	type	inport	source	source type
event.0	0	0	event	0	position	float
gravity	0	0	constant			
position	1	0	integrator	0	velocity	float
scope.0	0	0	scope	0	velocity	float
				1	position	float
velocity	1	0	integrator	0	gravity	float

```
[10]: out = sim.run(bd, T=5) # , watch=[demand, sum]) # simulate for 5s
```

```
>>> Start simulation: T = 5, dt = None, max_step = 0.05
```

```
Hybrid system solver: 2 continuous state variables, 0 discrete state variables
```

```
x0 = [ 0. 10.]
```

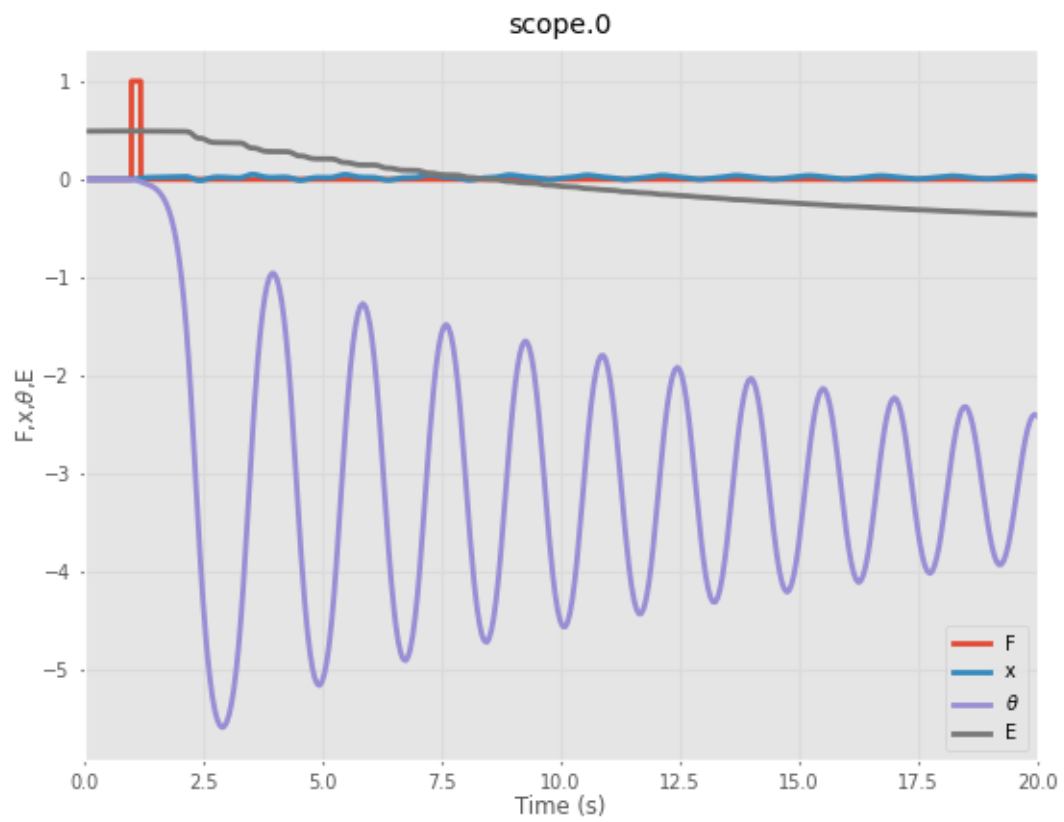


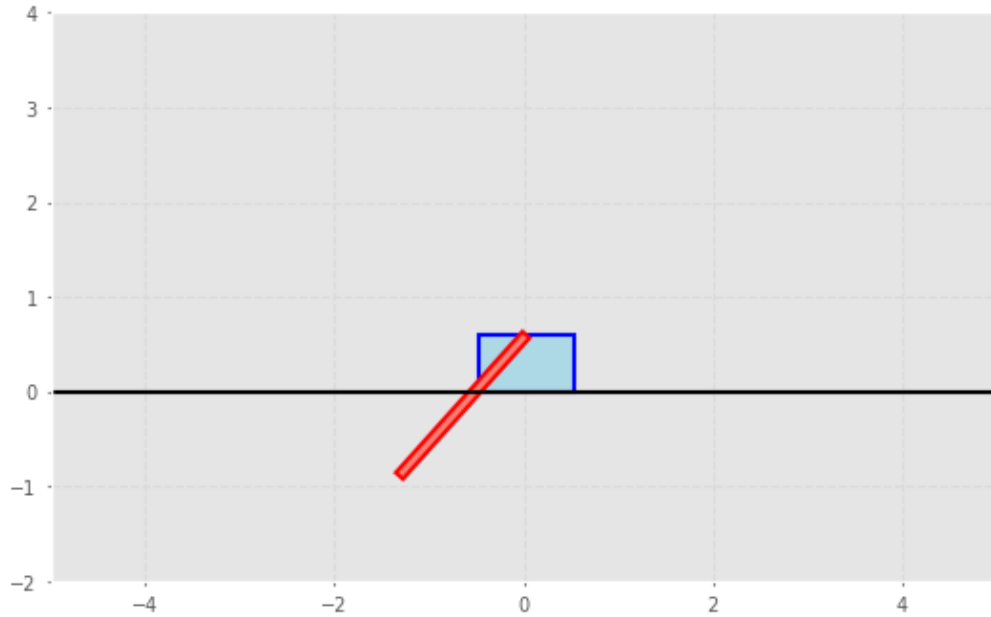
```
<<< Simulation complete
  block diagram evaluations: 1256
    of which ydot calls:      906  (remaining are post-integration replay + sink
passes)
  bd.evaluate() mean time:    47.1 us/call  (total 59.1 ms)
  clocktime speedup:         0.81x  (simulated 5s in 6161.4 ms)
  integration time points:    117
  scheduled event intervals: 102  (each bounded by a clock tick, scheduled
event, or terminal marker)
  integrator wall time:       0.100 s  (solve_ivp only; excludes post-
integration replay)
  zero-crossing events:       2
    event.0: 2
```

3 Cart-pole system

This is another classical simulation problem, a non-linear system comprising two coupled dynamical systems.

```
[11]: %run support/cartpole.py -q
```





t=20.000

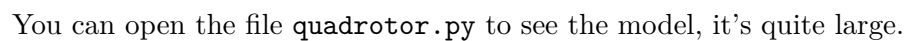
```
t      = ndarray:float64 (752,)
x      = ndarray:float64 (752, 4)
xnames = ['x_dot:x_0', 'x:x_0', 'theta_dot:x_0', 'theta:x_0'] (list)
y      = ndarray:float64 (752, 3)
ynames = ['_sum.0', 'x', 'theta'] (list)
```

4 Quadrotor flight controller

This block diagram is a complete controller for a quadrotor and drives it to follow a circular path. The controller has nested loops, from the inside going outwards:

- roll/pitch attitude control
- CoM velocity control
- CoM position control

and a separate yaw controller, independent of the CoM motion

 $t=6.283$

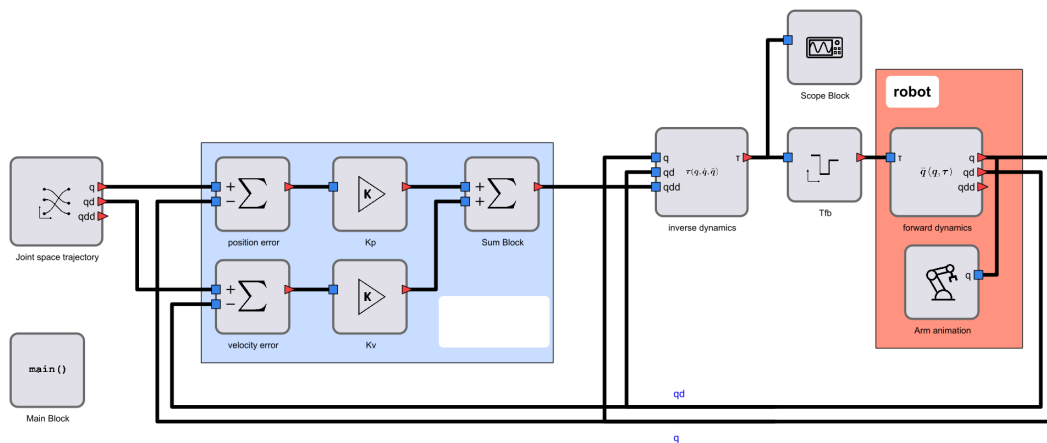
```

t      = ndarray:float64 (339,)
x      = ndarray:float64 (339, 12)
xnames = ['quadrotor:x_0', 'quadrotor:x_1', 'quadrotor:x_2', 'quadrotor:x_3',
'quadrotor:x_4', 'quadrotor:x_5', 'quadrotor:x_6', 'quadrotor:x_7',
'quadrotor:x_8', 'quadrotor:x_9', 'quadrotor:x_10', 'quadrotor:x_11'] (list)

```

5 The block diagram editor

The nice looking block diagrams above were drawn with `bdedit` which is a Qt-based editor included with `bdsim`. It's run from the command-line, and can't be run in a notebook. It provides a fairly complete block diagram drawing environment and the results are saved in JSON files with a `.bd` extension.



`.bd` files can be parsed to build a `bdim` block diagram which can then be simulated. For example

```

[13]: from bdsim import BDSim, bdload
from roboticstoolbox.models.DH import Puma560

sim = BDSim(animation=True, quiet=True)
bd = sim.blockdiagram()

# create objects the loaded model will need
robot = Puma560().nofriction()
clock_fb = bd.clock(50, unit="Hz")

# load the model
bd = bdload(bd, "support/computed-torque.bd", globalvars=globals())

# compile and run
bd.compile()
sim.report(bd, "summary")
out = sim.run(bd, 5)

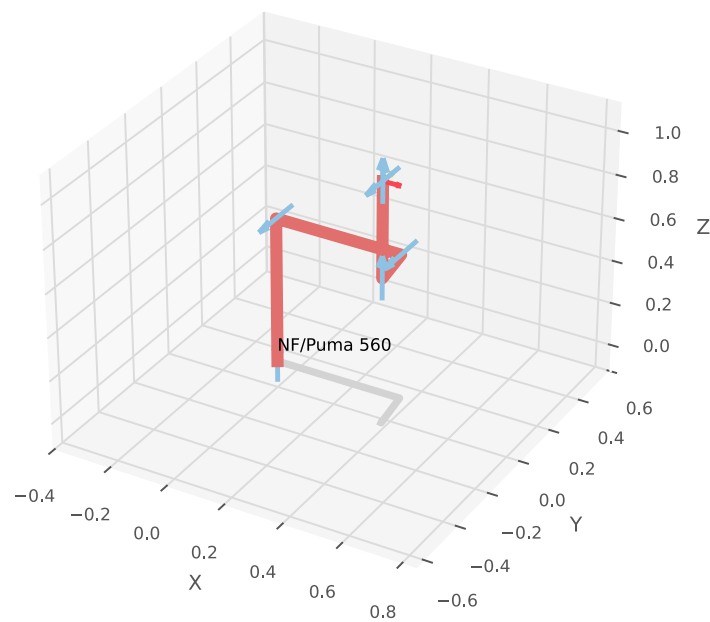
```

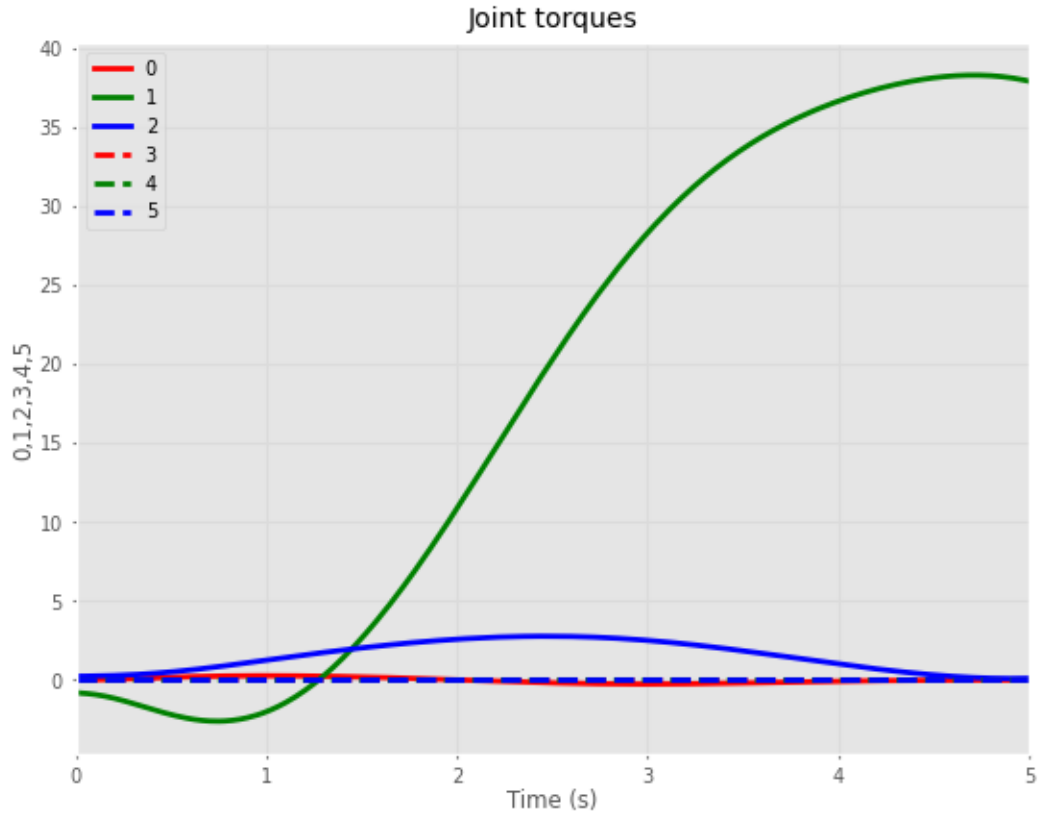
```
print(out)
```

INFORMATION: [Block.continuous('forward dynamics', type=fdyn, nin=1, nout=3, nstates=12, inames=['\tau'], onames=['q', 'qd', 'qdd'])] output 2 is not connected

INFORMATION: [Block.source('Joint space trajectory', type=jtraj, nout=3, onames=['q', 'qd', 'qdd'])] output 2 is not connected

t = 5.03





```
clock0::
    X      = ndarray:float64 (250, 6)
    Xnames = ['Tfb:X_0', 'Tfb:X_1', 'Tfb:X_2', 'Tfb:X_3', 'Tfb:X_4',
'Tfb:X_5'] (list)
    t      = ndarray:float64 (250,)
t      = ndarray:float64 (302,)
x      = ndarray:float64 (302, 12)
xnames = ['forward dynamics:x_0', 'forward dynamics:x_1', 'forward
dynamics:x_2', 'forward dynamics:x_3', 'forward dynamics:x_4', 'forward
dynamics:x_5', 'forward dynamics:x_6', 'forward dynamics:x_7', 'forward
dynamics:x_8', 'forward dynamics:x_9', 'forward dynamics:x_10', 'forward
dynamics:x_11'] (list)
```

6 What next for bdsim?

Not yet releasable but these things are under development:

- web-based editor to replace Qt-based `bdedit`
- soft real-time version, running on RaspberryPi and host + Arduino/Firmata
- C++ code generation