Python Dynamics Simulations : Part 1 — Setting Up Simulations

From some years now, it has become popular to teach university subjects with the help of Matlab and Simulink. It is justified as it is a plug and play environment which allows to launch all kind of simulations with minor efforts and to focus in the actual content under study. Nice. The problem it originates is the dependence of this property software. Many people cannot simulate dynamics (e.g. a simple DC motor) *without using Matlab*.

> *I worked in a project for which engineers were struggling to embed Simulink in a test-bed system as they were not able to simulate the digital twin otherwise… That does not make sense.*

For this reason I have planned to write some stories, a tutorial, to show how I deal with the dynamics simulations using Python, NumPy and SciPy (mainly) which can be executed even with a Raspberry Pi. Here you have the complete index of the tutorial.

In this **Part 1** I will show some easy ways to create system "blocks" and how to interconnect them for the simulation.

> ***Note: This post** is the most basic in terms of code. It **is introductory**, written for people who is not comfortable dealing with OOP. If it is too easy for you, I encourage you to take a look of the index and to jump to the stuff you want.*

1. Steps

The following is my workflow recommendation, it is not the only way to do it so feel free to make suggestions.

1. Draw a **block diagram**. As we do not have a visual framework to setup the system blocks, it is easier if you draw them first with reference to all the signals.
2. Write, at least, **one function** for **each block**. The code is more readable when there is a function for the DC motor model and another for its controller. This gets more important as systems become more complex.
3. Write an **interconnection function**.
   - The functions must begin with the extraction of the previous simulation step arguments, i.e. for time interval k, the variable x_{k} will be equal to x_{k+1} from the previous step.
   - With the help of the block diagram and (important!) suitable nomenclature, write outputs variables as inputs for the subsequent blocks.
   - Return the values which will be the updating arguments for the following interval and also all the data you may want to store or plot later.
4. Create a **initial values list** with the same length of the interconnection function return list, one value per variable.
5. Create **time vector** and (if necessary) **reference values** for the controller.
6. Do not forget to **plot your results** or to store them after simulating.

2. Non-Linear Motor Control

To show it in a simple way, we will see the non-linear control (**input-output linearization** method) of a DC motor as an example of those steps and to show what can we achieve simulating dynamics with Python. If you only want to see how to implement your equations, **you can skip the maths** stuff.

2.1 The Mathematics

Suppose the motor dynamics following the expression [1]:

$$\dot{\tau} = -ak\,\tau + ku$$

Equation 1. Motor model

Where *tau* is the torque, ***a*** (empirically determined) weights the inertia due to the current torque, ***k*** (empirically determined too) weights the effort through the current torque and the input to modify the change rate of the torque itself, and ***u*** is the control input to the motor, the voltage. Then, choosing the control law (how ***u*** is calculated) as follows:

$$u = a\,\tau + \left(\dot{\tau}_{ref} - k_{c1}\left(\tau - \tau_{ref}\right)\right)/k$$

Equation 2. Control law

Leads the output torque to:

$$\dot{\tau} = \dot{\tau}_{ref} - k_{c1}\left(\tau - \tau_{ref}\right) \rightarrow \dot{e} + k_{c1}e = 0$$

Equation 3. Error dynamics

Thus the term which depends of tau gets compensated by the first control term, linearizing the relation input-output and making the error to exponentially tend to zero.

2.2 Code Implementation

1. Blocks diagram:

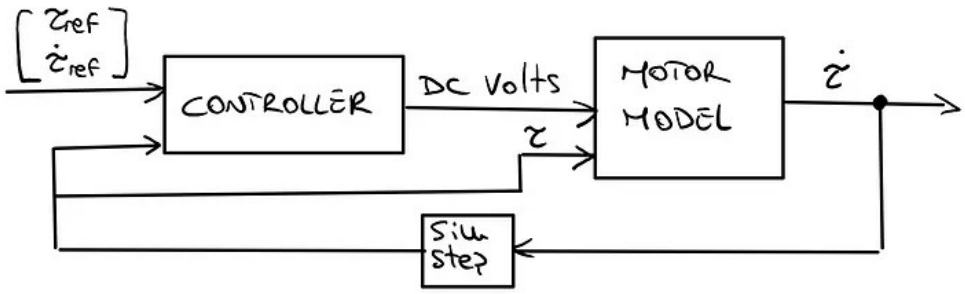

Figure 1. Blocks diagram of motor model with controller

Note. The simulation step block only serves for us to clearly see how outputs are propagated in time, but it is not something you have to create.

2. Functions for each module: Here the controller and the motor model.

```python
def dc_motor_model(x1_m, u):
    # DC motor model:
    # taup + a*k*tau = k*u
    # With the change: x1_m = tau (x1_motor)
    dx1_m = -a*k*x1_m + k*u
    y1_m = x1_m
    return dx1_m
```

dc_motor_model.py hosted with ♥ by GitHub view raw

Motor model code

```python
def motor_controller(tau, tau_ref, taup_ref):
    # Non-Linear control for DC Motor following Dyn ecs: taup + a*k*tau = k*u
    # The controller returns dc_volts
    v = taup_ref - k_c1*(tau - tau_ref)
    return (a+a_model_error)*tau + v/(k+k_model_error)
```

nonlinear_motor_controller.py hosted with ♥ by GitHub view raw

Controller code

3. Interconnection function: This function also must have a special arguments in order to work as ODE Integrator expects to, i.e. integrated signals (states), time steps and external arguments like the references for the controller.

```python
def connected_systems_model(states, t, tau_ref, taup_ref):
    # Input values. Check this with the out_states list
    x1_m, _ = states

    # Compute motor controller
    dc_volts = motor_controller(x1_m, tau_ref, taup_ref)
    # Compute motor torque
    tau, taup = dc_motor_model(x1_m, dc_volts)

    # Output
    out_states = [tau, dc_volts]
    return out_states
```

interconnection_function.py hosted with ♥ by GitHub view raw

Interconnection code

4. Initial values, time vector and ODE simulation: In this example, the ODE integration is made for every step, loading the corresponding references.

```python
# Initial conditions
states0 = [0, 0]
n = int((1 / (ts_ms / 1000.0))*tf + 1) # number of time points

time_vector = np.linspace(0,tf,n)
t_sim_step = time_vector[1] - time_vector[0]

# Reference to be followed by the system
torque_ref = np.sin(time_vector)
torquep_ref = np.cos(time_vector)

for i in range(n-1):
    out_states = odeint(connected_systems_model,states0,[0.0, tf/n],
                        args=(torque_ref[i],torquep_ref[i]))
    # Replace initial states with the last outputs for the nex sim step
    states0 = out_states[-1,:]
    # Store all simulation outputs
    states[i] = out_states[-1,:]
```

scipy_ode_use.py hosted with ♥ by GitHub view raw

ODE Integration (simulation)

Note: It may be easier to run the simulation once instead of taking every step separately in a for loop, but this way you can add logic and other stuff

without breaking the integrator. Feel free to share any considerations here!

3. Results

The resulting controller for this simple system makes perfect results, so I **added some modelling error** to the controller embedded model, with respect to the real motor model, in order to evaluate how it works with some **uncertainties**. For values: a = 1, k = 1, a_model_error = 0.1 k_model_error = 0.3, the results are something like this (in red, real output, dashed the reference):

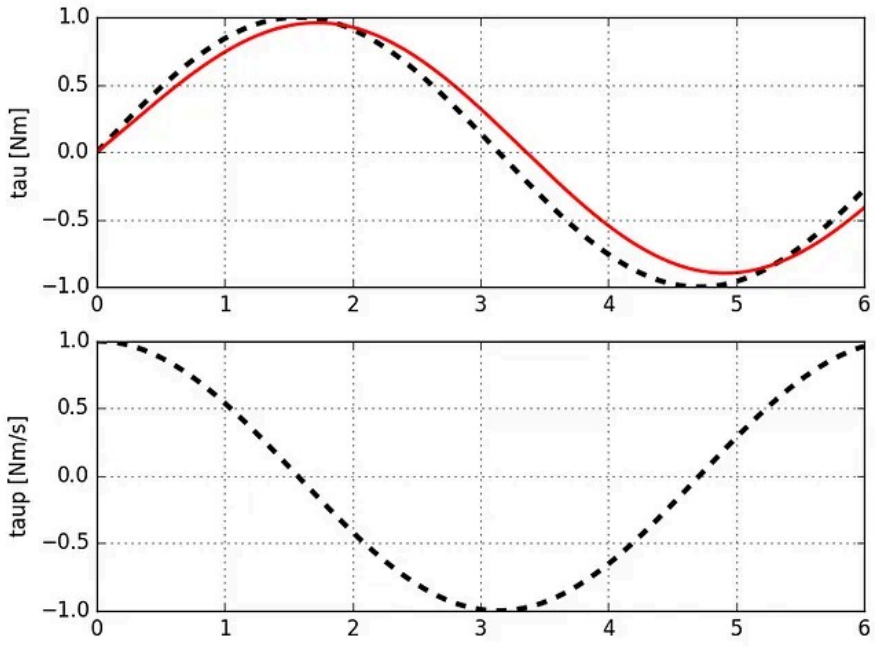

Figure 2. Simulation results with non-perfect modelling of the motor.

Pretty good even with injected errors in the modelling of 10% and 30% respectively for each model parameter.

That's all ! Here you can find the **complete code on Github** (also with the plots functions). To learn how to do these simulations, I read this Marat Kopytjuk story and also this web owned by John D. Hedengren. They have good material, so take a look of it.

I hope you enjoyed this tutorial and, if you have any thoughts, recommendations or petitions for future tutorials on Dynamic Simulations with Python, I would really appreciate your comments!

Now you can check the **second part** of this tutorial to learn how to test C/C++ controllers using what you know of Python Dynamics Simulations.

4. References