PYTHON BATTERY MATHEMATICAL MODELLING TRAINING WORKSHOP

Oxford University 31st July - 1st August 2019

Exercise 1 – solving ODEs in PyBaMM

Using the examples available in the PyBaMM repository, write a script which solves the following system of ODEs:

$$\frac{dx}{dt} = 4x - 2y, \quad x(0) = 1,$$
$$\frac{dy}{dt} = 3x - y, \quad y(0) = 2.$$

You can try to write the script from scratch, or copy the code in Listing 1 and fill in the blanks.

Listing 1: Solving ODEs script

```
import pybamm
import numpy as np
import matplotlib.pyplot as plt

"Setting up the model"

# 1. Initialise an empty model
model = pybamm.BaseModel()

# 2. Define variables
## DEFINE YOUR VARIABLES HERE ##

# 3. State governing equations
## WRITE THE EQUATIONS HERE ##
model.rhs = {}

# 4. State initial conditions
## ADD INITIAL CONDITIONS HERE ##
model.initial_conditions = {}

# 6. State output variables
## STATE OUTPUT VARIABLES HERE ##
model.variables = {}

"Using the model"

# use default discretisation
disc = pybamm.Discretisation()
```

```
## PROCESS MODEL USING THE GIVEN DISCRETISTAION ##

# solve
solver = pybamm.ScipySolver()
## SOLVE MODEL USING THE GIVEN SOLVER ##

# post-process
## PROCESS THE SOLUTION FOR PLOTTING ##

# plot
## PLOT SOLUTION ##
```

Exercise 2 – solving PDEs in PyBaMM

Write a script to solve the problem of linear diffusion on a unit sphere,

$$\frac{\partial c}{\partial t} = \nabla \cdot (\nabla c),$$

with the following boundary and initial conditions:

$$\left.\frac{\partial c}{\partial r}\right|_{r=0} = 0, \quad \left.\frac{\partial c}{\partial r}\right|_{r=1} = 2, \quad c|_{t=0} = 1.$$

To get started, try looking at the create-model notebook, or fill in the template script below.

Listing 2: Solving PDEs script

```
import pybamm
import numpy as np
import matplotlib.pyplot as plt

"Setting up the model"

# 1. Initialise an empty model
model = pybamm.BaseModel()

# 2. Define variables
## DEFINE YOUR VARIABLES HERE ##

# 3. State governing equations
## WRITE THE EQUATIONS HERE ##
model.rhs = {}

# 4. State boundary conditions
## ADD BOUNDARY CONDITIONS HERE ##
model.boundary_conditions = {}

# 5. State initial conditions
## ADD INITIAL CONDITIONS HERE ##
model.initial_conditions = {}

# 6. State output variables
## STATE OUTPUT VARIABLES HERE ##
model.variables = {}

"Using the model"

# define geometry
r = pybamm.SpatialVariable(
    "r", domain=["negative particle"], coord_sys="spherical polar"
)
geometry = {
```

```
    "negative particle": {
        "primary": {r: {"min": pybamm.Scalar(0), "max": pybamm.Scalar(1)}}
    }
}

# mesh and discretise
submesh_types = {"negative particle": pybamm.Uniform1DSubMesh}
var_pts = {r: 20}
mesh = pybamm.Mesh(geometry, submesh_types, var_pts)

spatial_methods = {"negative particle": pybamm.FiniteVolume}
disc = pybamm.Discretisation(mesh, spatial_methods)
disc.process_model(model)

# solve
solver = pybamm.ScipySolver()
## SOLVE MODEL USING THE GIVEN SOLVER ##

# post-process
## PROCESS THE SOLUTION FOR PLOTTING ##

# plot
## PLOT SOLUTION ##
```

Try solving the model again with different boundary or initial conditions.

Exercise 3 – extending the PDE model

In PyBaMM, parameter objects can be used to define parameters whose value is set during processing of the model. In practice, parameter values can be read in from an external source, such as a .csv file, but they can also be set in a dictionary before model processing. Try to extend your model to include a diffusion coefficient D, i.e. solve

$$\frac{\partial c}{\partial t} = \nabla \cdot (D \nabla c),$$

Listing 3: Adding a parameter and setting its value

```
D = pybamm.Parameter("Diffusion coefficient")
param = pybamm.ParameterValues({"Diffusion coefficient": 0.5})
```

Try adding more parameters to your model, or changing the parameter values.

You can also add additional output variables to your model which can be accessed after the solve. For instance, you may interested in the flux as well as the concentration. Extra output variables are easily added to the model.variables dictionary in PyBaMM.

Listing 4: Adding extra output variables

```
# define the flux
N = -D * pybamm.grad(c)
model.variables = {"Concentration": c, "Flux": N}
```

Exercise 4 – the negative particle problem

Now it is time to solve a real-life battery problem! Adapt your linear diffusion model to solve the problem of diffusion in the negative electrode particle within the single particle model. That is,

$$\frac{\partial c}{\partial t} = \nabla \cdot (D \nabla c),$$

with the following boundary and initial conditions:

$$\left.\frac{\partial c}{\partial r}\right|_{r=0} = 0, \quad \left.\frac{\partial c}{\partial r}\right|_{r=R} = -\frac{j}{FD}, \quad c|_{t=0} = c_0,$$

where c is the concentration, r the radial coordinate, t time, R the particle radius, D the diffusion coefficient, j the interfacial current density, F Faraday's constant, and c_0 the initial concentration. Use the parameters from Table 1. You will need to add the parameter values to the the `pybamm.ParameterValues` dictionary, as in Exercise 3.

Symbol	Units	Value
R	m	10×10^{-6}
D	$\mathrm{m^2\,s^{-1}}$	3.9×10^{-14}
j	$\mathrm{A\,m^{-2}}$	1.4
F	$\mathrm{C\,mol^{-1}}$	96485
c_0	$\mathrm{mol\,m^{-3}}$	2.5×10^4

Table 1: Parameter values for use in Exercise 4.

Exercise 5 – making a model class

Now that you have created a script which solves the negative particle problem you have all of the ingredients you need to create a model. PyBaMM is designed to be used in a modular fashion: users create their own models or submodels which may be combined to capture different physical phenomena. Within this framework, it is easy to add new physics to an existing model, without the need to write a new code from scratch. As well as being able to include different models, users can also quickly change between different spatial discretisations and solvers, so that solution strategies may be easily compared within the same modelling framework.

Create a new file called `my_spherical_diffusion.py` and save it in `pybamm/models`. Before starting work on you model, you will need to add it to the file `PyBaMM/__init__.py` so that it is imported as part of pybamm (see line 178 of Listing 5).

Listing 5: Adding you new model to the init file.

```
172 # Battery models
173 from .models.full_battery_models.base_battery_model import ↩
       BaseBatteryModel
174 from .models.full_battery_models import lead_acid
175 from .models.full_battery_models import lithium_ion
176
177 # User models
178 from .models.my_spherical_diffusion import MySphericalDiffusion
```

Models in PyBaMM are classes, which can inherit features from a number of base classes. The simplest is `pybamm.BaseModel`. Create a new class called `MySphericalDiffusion` which is an instance of `pybamm.BaseModel`, and takes an argument `param` so that different parameters can be passed to the model. To get started take a look at the code in Listing 6. You can then add the variables, equations, initial and boundary conditions from your script into the class file.

Listing 6: Creating you new model.

```
1  import pybamm
2
3
4  class MySphericalDiffusion(pybamm.BaseModel):
5      """A model for diffusion in a sphere.
6
7      **Extends:** :class:`pybamm.BaseModel`
8      """
9      def __init__(self, param, name="Spherical Diffusion"):
10         # Initialise base class
11         super().__init__(name)
12
13         # Add parameters as an attribute of the model
```

```
14          self.param = param
15
16          ## ADD THE DETAILS OF THE MODEL HERE ##
```

You will often want to reuse the same parameters across a number of models, so it can be useful to store them in a standard parameters file instead of redefining them each time you would like to use them. Create a file `my_parameters.py` and save it in `pybamm/parameters`. Here you can add the definitions the parameters you used in Exercise 4. Add your parameters file to `PyBaMM/__init__.py` so that it is imported as part of pybamm (see line 207 of Listing 7).

Listing 7: Adding your parameters to the init file.

```
197 #
198 # Parameters class and methods
199 #
200 from .parameters.parameter_values import ParameterValues
201 from .parameters import standard_current_functions
202 from .parameters import geometric_parameters
203 from .parameters import electrical_parameters
204 from .parameters import thermal_parameters
205 from .parameters import standard_parameters_lithium_ion, ↩
       standard_parameters_lead_acid
206 from .parameters.print_parameters import print_parameters, ↩
       print_evaluated_parameters
207 from .parameters import parameter_sets, my_parameters
```

Now that you have defined your parameters and created your model, they can be easily call up for use in a script (see Listing 8). Once the model is loaded, you can define the geometry, mesh and discretisation, and solve as before.

Listing 8: Using your model.

```
1  import pybamm
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  "Setting up the model"
6
7  # 1. Initialise model
8  param = pybamm.my_parameters
9  model = pybamm.MySphericalDiffusion(param)
10
11 "Using the model"
12
13 ## DEFINE GEOMETRY, MESH, ETC. HERE, AND SOLVE
```

Congratulations, you have now created your first PyBaMM model!

Try extending the model to include default settings for the geometry, parameters, mesh and solver. You can also pass more arguments to the model. For instance, you may want to pass the domain (“negative" or “positive") so that the model can alter the boundary conditions depending on which electrode you are solving in.