PyODESolver

Python package for solving ODEs

Quickstart guide:

To get started with using PyODESolver include a solver method. In this example we will use the explicit euler method to solve the differential equation y'(t) = -2 * y(t), y(0) = pi. This differential equation has the exact solution

```
y(t) = pi * exp(-2 * t).
```

The right hand side in the expression

```
y'(t) = -2*y(t),
y(0) = pi
```

is called a RHSFunction. This example has been implemented in ExampleFunc01 in the file rhs_function.py. To use the explicit euler solver for this ODE we first import the required files

```
from method_explicit_euler import ExplicitEuler
from rhs_function import ExampleFunc01
import numpy as np
```

we can than construct a method object in our main function and call generate on it to get the solver object.

```
if __name__ == "__main__":
    N = 10**3
    t = np.linspace(0, 1, num=N)
    y0 = np.pi
    ee_solver = ExplicitEuler(N, y0, [0, 1], ExampleFunc01())
    solution = ee_solver.generate()
```

The generator object is useful in this instance as it very lean in term of required memory and puts full control to the user at it allows to generate new time steps as required. For example one thread can be used for updating a plot and a different thread can call next(solution) to match the framerate of the plot.

To extract the whole solution we can use:

```
numericSol = []
for (time, val) in solution:
    numericSol.append(val)
```

To plot the solution in this example we can use

```
import matplotlib.pyplot as plt
plt.plot(t, np.array(numericSol))
plt.show()
```

Solving arbitrary ODEs

To solve a customised ode the only two components required are the new RHSFunction and the Jacobian of this function. In cases where the Jacobian is unknown explicit methods will still function.

To implement a new RHSFunction the user can inherit from RHSFunction and define a new subclass

```
class NewFunction(RHSFunction):
...
```

RHSFunction has exposes three methods that raise NotImplementedErrors. The user can then implement these methods by providing

```
def eval(self, y_vec, time):
    ...
```

and

```
def jacobian(self, y_vec, time):
    ...
```

to provide everything required for explicit and implicit methods. The solution can be obtained as described in the Quick-start-guide.