SCIENTIFIC COMPUTING USING PYTHON
1. PYTHON + SCIENTIFIC COMPUTING, PROJECT

Silvin Willemsen

June 2021

1 Problem Statement

This projects implements the Lorenz attractor, devised by Edward Lorenz in his 1963 paper [1] in the Python programming language. The Lorenz attractor is a mathematical model for atmospheric convection and is described by the following non-linearly coupled ordinary differential equations (ODEs):

$$\frac{dx}{dt} = \sigma(y - x), \tag{1a}$$
$$\frac{dy}{dt} = x(\rho - z) - y, \tag{1b}$$
$$\frac{dz}{dt} = xy - \beta z, \tag{1c}$$

where independent variables $x = x(t)$, $y = y(t)$, and $z = z(t)$, can be interpreted as coordinates in three-dimensional space (in m) and are all dependent on time t (in s). The system is parameterised by σ, ρ and β, which govern the eventual behaviour of the system.

Discretisation

In order to implement the continuous-time ODE described in (1), they need to be approximated in a process called *discretisation*. Time t can be discretised using $t = nt_\delta$. Here, the temporal index $n \in \{0, \dots, N\}$ is an integer with desired number of iterations N and t_δ is the time step between two consecutive indices (in s). The spatial coordinates can then be be approximated using $x(t) \approxeq x[n]$, $y(t) \approxeq y[n]$ and $z(t) \approxeq z[n]$.

To approximate the derivatives in (1), we can use Euler's method. A derivative of a continuous function $f(t)$ can be approximated according to

$$\frac{df}{dt} \approxeq \frac{f(t + t_\delta) - f(t)}{t_\delta}. \tag{2}$$

In other words, the derivative is approximated using difference between two consecutive function-values divided by the difference in t (t_δ) between them. Other (more accurate) approximations exist, but for simplicity, we continue with the above.

We can then apply Eq. (2) to the system of ODEs in (1) to get an approximation. Together with substituting the discrete variables $x[n]$, $y[n]$ and $z[n]$, this yields

$$\frac{x[n+1] - x[n]}{t_\delta} = \sigma(y[n] - x[n]), \tag{3a}$$
$$\frac{y[n+1] - y[n]}{t_\delta} = x[n](\rho - z[n]) - y[n], \tag{3b}$$
$$\frac{z[n+1] - z[n]}{t_\delta} = x[n]y[n] - \beta z[n]. \tag{3c}$$

The last step before these equations can be implemented in code, is to write them in terms of the respective coordinate at the next time step $n+1$. This yields the following update equations

$$x[n+1] = \sigma(y[n] - x[n])t_\delta + x[n], \tag{4a}$$
$$y[n+1] = \Big(x[n](\rho - z[n]) - y[n]\Big)t_\delta + y[n], \tag{4b}$$
$$z[n+1] = (x[n]y[n] - \beta z[n])t_\delta + z[n]. \tag{4c}$$

which can be implemented in Python. A for-loop needs to be created so Eqs. (4) can be iteratively calculated until a user-specified iteration number N.

1.1 Cases

The project description states 5 different cases that need to be tested. The cases contain different values for the parameters σ, β and ρ according to

1. $\sigma = 10$, $\beta = 8/3$, and $\rho = 6$
2. $\sigma = 10$, $\beta = 8/3$, and $\rho = 16$
3. $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$
4. $\sigma = 14$, $\beta = 8/3$, and $\rho = 28$
5. $\sigma = 14$, $\beta = 13/3$, and $\rho = 28$

2 Implementation

Algorithm 1 contains pseudocode showing the core algorithm implementing the Lorenz attractor. The implementation can be found online[1].

```
Initialise the following:
- $x[0]$, $y[0]$, and $z[0]$
- free parameters $\sigma$, $\beta$ and $\rho$
- time step $t_\delta$
- no. of iterations $N$
- vectors to store values for $x$, $y$ and $z$.
for $n = 0 : N-1$ do
    Calculate $x[n+1]$, $y[n+1]$, and $z[n+1]$ using Eqs. (4).
    Save these values to vectors initialised before
end
Plot the saved vectors $x$, $y$ and $z$.
```

Algorithm 1: *Pseudocode showing the order of calculations.*

The initial conditions used for the cases $x[0] = 1.0$, $y[0] = 1.0$ and $z[0] = 1.0$. Furthermore, the time step $t_\delta = 0.002$ and number of iterations $N = 10000$ for all cases. A script is made where the simulation can be easily run using other initial conditions, and parameters. This will be introduced below.

[1]https://github.com/SilvinWillemsen/lorenz

3 Design Considerations

The goal of the project is to implement Eqs. (4) in Python. The variables needed are the ones found in Eqs. (4). The output of the simulation needs to be three vectors (/arrays) that have the values of x, y and z stored over time. These can then be plotted using the `matplotlib.pyplot` module.

Below, the file and folder structure can be found:

```
/
├── lorenz
│   ├── __init__.py
│   ├── run.py
│   ├── solver.py
│   ├── finiteDifferences.py
│   └── plotting.py
├── cases
│   ├── run_case.py
│   ├── generate_data_and_figures.py
│   ├── generate_cases_CSV.py
│   ├── scheme_variables.csv
│   └── output_files
│       ├── case1_output
│       │   ├── simulation_result.hdf5
│       │   ├── xyPlot.pdf
│       │   ├── xzPlot.pdf
│       │   ├── yzPlot.pdf
│       │   └── xyzPlot.pdf
│       ├── case2_output
│       │   └── ...
│       ├── case3_output
│       ├── case4_output
│       └── case5_output
├── test
│   ├── __init__.py
│   └── test_lorenz.py
├── doc
│   └── ...
├── LICENSE
└── README.txt
```

Below, some descriptions of the various files are given. A much more detailed description of the modules and scripts, however, can be found in the files themselves and via the sphinx-generated documentation in `doc/_build/html/index.html`.

The `lorenz` package (/folder) contains the following files:

- `__init__.py`: needed to make `lorenz` a package
- `run.py`: This script can be run to simulate the Lorenz attractor.
- `solver.py`: Solves the system and returns vectors of data with x, y and z over time.
- `finiteDifferences.py`: Used by `solver.py` to solve the equations using Eqs. (4).
- `plotting.py`: Plots the results.

The `run.py` file runs the `solve` function in `solver.py` and plots the results (vectors of data with x, y and z over time) using the `plotLorenz` function in `plotting.py` (without generating files). The initial conditions, parameters (σ, β, and ρ), time step t_δ and number of iterations N can be changed in the file. I decided to have a separate file `finiteDifferences.py` that calculates the finite difference schemes, and is used by the solver to have a better overview of what is happening.

The `plotting.py` file plots the values of x, y and z over time returned by the `solve` function in `solver.py`. Three of the plots are 2D from an (x, y), (x, z) and (y, z) perspective, and one is a 3D plot. Apart from the data output from `solver.solve`, `plotting.plotLorenz` takes in some additional arguments. The first is the `step_size` that allows to skip some of the indices of the x, y and z vectors as it might get 'too crowded' in the plot for a smaller value of t_δ. Furthermore, the `scatter_size` argument determines the size of the scattered dots. The first data-entry is plotted in red and denotes the initial condition. The rest of the data is colour-coded based on the current velocity of the system state (again, calculated using Euler's method) and a colourbar is added as a legend.

The `solver.solve` and `plotting.plotLorenz` functions contain timers that print how long the functions took to run. This is to get an idea about the parts of the code that are most time-consuming.

The `cases` folder contains the following files:

- `run_case.py`: Script that uses `generate_data_and_figures.py` and generates the data and plots for a case specified in the code.
- `generate_data_and_figures.py`: Called from `run_case.py` and generates the data and plots for the case specified. It uses the `scheme_variables.csv` file to retrieve the cases data.
- `generate_cases_CSV.py`: Used to generate the `scheme_variables.csv` file.
- `scheme_variables.csv`: Contining the cases data from Section 1.1 generated by the `generate _cases_CSV.py` file/.
- `output_files`: Contains a folder for each case storing the plots and the data generated by `run_case.py`

I decided to not have separate files to generate each of the cases (i.e., `case1.py`, `case2.py`, etc.) but instead have one script `run_case.py` where the case number can be changed in the script. This to avoid copy-pasting the same file 5 times. The `run_case.py` file acts the same way as `run.py` but has the values of σ, β, and ρ determined by the `scheme_variables.csv` file rather than being user-determined. This `.csv` file contains the parameter-values of the 5 different cases presented in Section 1.1. An `action` variable can be changed from `'simulate'` to `'load'` to load the data after it has been generated. If there is no data available, an exception is thrown and the user is told to change `action` to `'simulate'` and generate the data first. Furthermore, the data and figures are generated and are saved in the `output_files` folder in a(nother) folder called `case<casenumber>_output`. So `case1_output` for case 1, `case2_output` for case 2, etc. The data generated (the output from `solver.solve`) is saved in an `.hdf5` file called `simulation_result.hdf5`. I decided on this file-type, as it is a great way to store large amounts of data and work with in Python. Furthermore, the 4 plots generated by `plotting.py` are saved in `.pdf` format as it allows for vector images and is cross-platform.

4 Testing

The main function that I want to test is `solver.solve` as this is where the ODE simulation is carried out. I chose to use `pytest` as the strategy for testing the code and decided against using `doctest`. This is because the function output is 'complicated' – large `numpy` arrays. I figured that a test class formatted for `pytest` can be written more elaborately than using the docstring and `doctest`.

Once I got plots that looked like the ones I found online, I knew that the implementation of the algorithm could be considered correct. In the future, to confirm this hunch, a comparison to the output of an ODE solver such as the one in `scipy` could be used to test whether the output of the approximation is close enough to the more accurate solution.

I decided against using a high number of iterations for testing and instead only used 2 iterations. This way I would know that the simulation had ran a few times and that it would go the right way and not return odd values.

The first test I wanted to implement was one that checked whether if the the initial conditions are set to 0 and the parameters are set to 0, the solver should return 0.

Then I tested with different initial conditions and parameter values, ran the solve function for 2 iterations and saved the output in the `test_lorenz.py` file. This could then be used for testing the `solve` function at a later point and confirm that any changes to the algorithm would not affect the final result.

As a wrong choice of initial conditions could make the scheme unstable, a final test should be implemented to check whether after a larger number of iterations (one that would normally be used to plot the results – e.g. $N = 10000$), the values are `nan` (not-a-number). Unfortunately, I have not been able to implement this and this is left for future work.

References

[1] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of Atmospheric Sciences*, vol. 20, pp. 130–141, 1963.