With modifications by A.Viceré

Full phugoid model

This is the third IPython Notebook of the series on the *phugoid model of glider flight*, from the course "Practical Numerical Methods with Python". In the first notebook, we described the physics of the trajectories known as phugoids obtained from an exchange of potential and kinetic energy in an idealized motion with no drag. We gave you a neat little code to play with and plot various phugoid curves.

In the second notebook, we looked at the equation representing small perturbations on the straight-line phugoid, resulting in simple harmonic motion. This is a second-order ordinary differential equation, and we solved it numerically using **Euler's method**: the simplest numerical method of all. We learned about convergence and calculated the error of the numerical solution, comparing with an analytical solution. That is a good foundation!

Now, let's go back to the dynamical model, and take away the idealization of no-drag. Let's remind ourselves of the forces affecting an aircraft, considering now that it may be accelerating, with an instantaneous upward trajectory. We use the designation θ for the angle, and consider it positive upwards.

Figure 1. Forces with a positive trajectory angle.

In Figure 1, L is the lift, W is the weight, D is the drag, and θ the positive angle of the trajectory, instantaneously.

In lesson 1, we wrote the force balance in the directions perpendicular and parallel to the trajectory for a glider in *equilibrium*. What if the forces are *not* in balance? Well, there will be acceleration terms in the equations of motion, and we would have in that case:

$$m\frac{dv}{dt} = -W\sin\theta - D \tag{1}$$

$$mv\frac{d\theta}{dt} = -W\cos\theta + L \tag{2}$$

We can use a few little tricks to make these equations more pleasing. First, use primes to denote the time derivatives and divide through by the weight:

$$\frac{v'}{g} = -\sin\theta - D/W \tag{3}$$

$$\frac{v}{g}\theta' = -\cos\theta + L/W \tag{4}$$

Recall, from our first lesson, that the ratio of lift to weight is known from the trim conditions
—
$L/$ W
— and also from the definitions of lift and drag,

$$L = C_L S \frac{1}{2}\rho v^2 \tag{5}$$

$$D = C_D S \frac{1}{2}\rho v^2 \tag{6}$$

we see that
$L/$ L
. The system of equations can be re-written:

$$v' = -g\,\sin\theta - \frac{C_D}{C_L}\frac{g}{v_t^2}v^2 \tag{7}$$

$$\theta' = -\frac{g}{v}\cos\theta + \frac{g}{v_t^2}\,v \tag{8}$$

It is very interesting that the first equation has the factor
$C_D/$
, which is the inverse of a measure of the aerodynamic efficiency of the aircraft. It turns out, this is the term that contributes damping to the phugoid model: if drag is zero, there is no damping. Drag is never zero in real life, but as engineers design more aerodynamically efficient aircraft, they make the phugoid mode more weakly damped. At altitude, this is nothing but a slight bother, but vertical oscillations are unsafe during final approach to land, so this is something to watch out for!

The initial value problem

If we want to visualize the flight trajectories predicted by this model, we are going to need to integrate the spatial coordinates, which depend on both the forward velocity (tangential to the trajectory) and the trajectory angle. The position of the glider on a vertical plane will be designated by coordinates
(x, y)
with respect to an inertial frame of reference, and are obtained from:

$$x'(t) = v\cos(\theta) \tag{9}$$

$$y'(t) = v\sin(\theta). \tag{10}$$

Augmenting our original two differential equations by the two equations above, we have a system of four first-order differential equations to solve. We will use a time-stepping approach, like in the previous lesson. To do so, we do need *initial values* for every unknown:

$$v(0) = v_0 \quad \text{and} \quad \theta(0) = \theta_0 x(0) = x_0 \quad \text{and} \quad y(0) = y_0$$

Solve with Euler's method

We know how to apply Euler's method from the previous lesson. We replace each of the time derivatives by an approximation of the form:

$$v'(t) \approx \frac{v^{n+1} - v^n}{\Delta t},$$

where we are now using a superscript n to indicate the n-th value in the time iterations. The first differential equation, for example, gives:

$$v' = -g \sin\theta - \frac{C_D}{C_L}\frac{g}{v_t^2} v^2$$

$$\frac{v^{n+1} - v^n}{\Delta t} = -g \sin\theta^n - \frac{C_D}{C_L}\frac{g}{v_t^2} (v^n)^2$$

Alright, we know where this is going. At each time iteration t^n, we want to evaluate all the known data of our system to obtain the state at t^{n+1}—the next time step. We say that we are *stepping in time* or *time marching*.

The full system of equations discretized with Euler's method is:

$$v^{n+1} = v^n + \Delta t \left(-g \sin\theta^n - \frac{C_D}{C_L}\frac{g}{v_t^2} (v^n)^2 \right) \tag{11}$$

$$\theta^{n+1} = \theta^n + \Delta t \left(-\frac{g}{v^n} \cos\theta^n + \frac{g}{v_t^2} v^n \right) \tag{12}$$

$$x^{n+1} = x^n + \Delta t \, v^n \cos\theta^n \tag{13}$$

$$y^{n+1} = y^n + \Delta t \, v^n \sin\theta^n. \tag{14}$$

As we've learned before, the system of differential equations can also be written as a vector equation:

$$u'(t) = f(u)$$

where

$$u = \begin{pmatrix} v \\ \theta \\ x \\ y \end{pmatrix} \qquad f(u) = \begin{pmatrix} -g\sin\theta - \frac{C_D}{C_L}\frac{g}{v_t^2}v^2 \\ -\frac{g}{v}\cos\theta + \frac{g}{v_t^2}v \\ v\cos\theta \\ v\sin\theta \end{pmatrix}. \tag{15}$$

It's a bit tricky to code the solution using a `NumPy` array holding all your independent variables. But if you do, a function for the Euler step can be written that takes any number of simultaneous equations. It simply steps in time using the same line of code:

```
def euler_step(u, f, dt):
    return u + dt * f(u)
```

This function can take a NumPy array `u` with any number of components. All we need to do is create an appropriate function `f(u)` describing our system of differential equations. Notice how we are passing a *function* as part of the arguments list to `euler_step()` . Neat!

And solve!

As always, we start by loading the modules and libraries that we need for this problem. We'll need a few transcendental functions, including the
$\log$
for a convergence study later on. And remember: the line `%matplotlib inline` is a magic function that tells Matplotlib to give us the plots in the notebook (the default behavior of Matplotlib is to open a pop-up window). In addition, we are importing the module `rcParams` to define notebook-wide plotting parameters: font family and size. Here we go!

In [ ]:
```
from math import sin, cos, log, ceil
import numpy
from matplotlib import pyplot
%matplotlib inline
from matplotlib import rcParams
rcParams['font.family'] = 'serif'
rcParams['font.size'] = 16
```

Next, we need to set things up to start our numerical solution: the parameter values and the *initial values*. You know what the acceleration of gravity is: 9.81 m/s
^{2}
, but what are good values for
$C_D/$
, the inverse of the aerodynamic efficiency? Some possible values are given on a table in the Wikipedia entry for lift-to-drag ratio: a modern sailplane can have
$L/$
of 40 to 60, depending on span (and, in case you're interested, a flying squirrel has
$L/$
close to 2).

For the *trim velocity*, the speed range for typical sailplanes is between 65 and 280 km/hr,

according to Wikipedia (it must be right!). Let's convert that to meters per second: 18 to 78 m/s. We'll pick a value somewhere in the middle of this range.

Here's a possible set of parameters for the simulation, but be sure to come back and change some of these, and see what happens!

In [ ]:
```
# model parameters:
g = 9.81     # gravity in m s^{-2}
v_t = 30.0   # trim velocity in m s^{-1}
C_D = 1./40.  # drag coefficient --- or D/L if C_L=1
C_L = 1.   # for convenience, use C_L = 1

### set initial conditions ###
v0 = v_t     # start at the trim velocity (or add a delta)
theta0 = 0. # initial angle of trajectory
x0 = 0.      # horizontal position is arbitrary
y0 = 1000.  # initial altitude
```

We'll define a function `f()` to match the right-hand side of Equation (15), the full differential system in vector form. This function assumes that we have available the parameters defined above. If you re-execute the cell above with different parameter values, you can just run the solution without re-executing the function definition.

In [ ]:
```
def f(u):
    """Returns the right-hand side of the phugoid system of equations.

    Parameters
    ----------
    u : array of float
        array containing the solution at time n.

    Returns
    -------
    dudt : array of float
        array containing the RHS given u.
    """

    v = u[0]
    theta = u[1]
    x = u[2]
    y = u[3]
    return numpy.array([-g*sin(theta) - C_D/C_L*g/v_t**2*v**2,
                        -g*cos(theta)/v + g/v_t**2*v,
                        v*cos(theta),
                        v*sin(theta)])
```

Compare the code defining function `f(u)` with the differential equations, and convince yourself that it's right!

$$u = \begin{pmatrix} v \\ \theta \\ x \\ y \end{pmatrix} \qquad f(u) = \begin{pmatrix} -g\sin\theta - \frac{C_D}{C_L}\frac{g}{v_t^2}v^2 \\ -\frac{g}{v}\cos\theta + \frac{g}{v_t^2}v \\ v\cos\theta \\ v\sin\theta \end{pmatrix}$$

Now, Euler's method is implemented in a simple function `euler_step()`:

In [ ]:
```
def euler_step(u, f, dt):
    """Returns the solution at the next time-step using Euler's method.

    Parameters
    ----------
    u : array of float
        solution at the previous time-step.
    f : function
        function to compute the right hand-side of the system of equations.
    dt : float
        time-increment.

    Returns
    -------
    u_n_plus_1 : array of float
        approximate solution at the next time step.
    """

    return u + dt * f(u)
```

After defining a final time for the solution, and the time step Δt, we can construct the grid in time using the NumPy function `linspace()`. Make sure you study the decisions we made here to build the time grid.

Look at the code below, and make sure you understand the following aspects of it.

- The NumPy array `u` contains the solution at every time-step, consisting of the velocity, angle and location of the glider.
- The first element of the array `u` is set to contain the initial conditions.
- In the `for`-loop, the function `euler_step()` is called to get the solution at time-step $n+1$.

In [ ]:
```
T = 100                          # final time
dt = 0.1                         # time increment
N = int(T/dt)                    # number of time-steps
t = numpy.linspace(0, T, N)      # time discretization

# initialize the array containing the solution for each time-step
u = numpy.empty((N, 4))
u[0] = numpy.array([v0, theta0, x0, y0])# fill 1st element with initial values
```

```
# time loop - Euler method
# n cycles from 0 to N-2 included
for n in range(N-1):
    u[n+1] = euler_step(u[n], f, dt)
```

Plot the trajectory

In order to plot the path of the glider, we need the location (`x` , `y`) with respect to time. That information is already contained in our NumPy array containing the solution; we just need to pluck it out.

Make sure you understand the indices to `u` , below, and the use of the colon notation. If any of it is confusing, read the Python documentation on Indexing.

In [ ]:
```
# get the glider's position with respect to the time
x = u[:,2]
y = u[:,3]
```

Time to plot the path of the glider and get the distance travelled!

In [ ]:
```
# visualization of the path
pyplot.figure(figsize=(8,6))
pyplot.grid(True)
pyplot.xlabel(r'x', fontsize=18)
pyplot.ylabel(r'y', fontsize=18)
pyplot.title('Glider trajectory, flight time = %.2f' % T, fontsize=18)
pyplot.plot(x,y, 'r-', lw=2);
```

Grid convergence

Let's study the convergence of Euler's method for the phugoid model. In the previous lesson, when we studied the straight-line phugoid under a small perturbation, we looked at convergence by comparing the numerical solution with the exact solution. Unfortunately, most problems don't have an exact solution (that's why we compute in the first place!). But here's a neat thing: we can use numerical solutions computed on different grids to study the convergence of the method, even without an analytical solution.

We need to be careful, though, and make sure that the fine-grid solution is resolving all of the features in the mathematical model. How can we know this? We'll have a look at that in a bit. Let's see how this works first.

You need a sequence of numerical solutions of the same problem, each with a different number of time grid points.

Let's create a NumPy array called `dt_values` that contains the time-increment of each grid to be solved on. For each element `dt_values[i]`, we will compute the solution `u_values[i]` of the glider model using Euler's method. If we want to use five different values of
Δt
, we'll have five solutions: we put them in an array ... but each one is also an array! We'll have

an array of arrays. How meta is that?

We have one more trick up our sleeve: `enumerate()`. To get all the numerical solutions—each with its value of Δt—done in one fell swoop, we will loop over the elements of the array `dt_values`. Within the loop, we need to access both `dt_values[i]` and the index `i`. It turns out, `enumerate()` is a built-in Python function that will give us consecutive `index, value` pairs just like we need.

Read the code below carefully, and remember: you can get a help pane on any function by entering a question mark followed by the function name. For example, add a new code cell below and type: `?numpy.empty_like`.

In [ ]:
```
dt_values = numpy.array([0.1, 0.05, 0.01, 0.005, 0.001])

u_values = numpy.empty_like(dt_values, dtype=numpy.ndarray)

for i, dt in enumerate(dt_values):

    N = int(T/dt)

    ### discretize the time t ###
    t = numpy.linspace(0.0, T, N)

    # initialize the array containing the solution for each time-step
    u = numpy.empty((N, 4))
    u[0] = numpy.array([v0, theta0, x0, y0])

    # time loop
    for n in range(N-1):

        u[n+1] = euler_step(u[n], f, dt)   ### call euler_step() ###

    # store the value of u related to one grid
    u_values[i] = u
```

In Lesson 2, we compared our numerical result to an analytical solution, but now we will instead compare numerical results from different grids.

For each solution, we'll compute the difference relative to the finest grid. You will be tempted to call this an "*error*", but be careful: the solution at the finest grid is *not the exact* solution, it is just a reference value that we can use to estimate grid convergence.

To calculate the difference between one solution `u_current` and the solution at the finest grid, `u_finest`, we'll use the L_1-norm, but any norm will do.

There is a small problem with this, though. The coarsest grid, where $\Delta t = 0.1$

, has 1000 grid points, while the finest grid, with
$\Delta t = 0.001$
has 100000 grid points. How do we know which grid points correspond to the same location in two numerical solutions, in order to compare them?

If we had time grids of 10 and 100 steps, respectively, this would be relatively simple to calculate. Each element in our 10-step grid would span ten elements in our 100-step grid.

Calculating the *ratio* of the two grid sizes will tell us how many elements in our fine-grid will span over one element in our coarser grid.

Recall that we can *slice* a NumPy array and grab a subset of values from it. The syntax for that is

```
my_array[3:8]
```

An additional slicing trick that we can take advantage of is the "slice step size." We add an additional `:` to the slice range and then specify how many steps to take between elements. For example, this code

```
my_array[3:8:2]
```

will return the values of `my_array[3]`, `my_array[5]` and `my_array[7]`

With that, we can write a function to obtain the differences between coarser and finest grids. Here we go ...

In [ ]:
```
def get_diffgrid(u_current, u_fine, dt):
    """Returns the difference between one grid and the fine one using L-1 norm.

    Parameters
    ----------
    u_current : array of float
        solution on the current grid.
    u_finest : array of float
        solution on the fine grid.
    dt : float
        time-increment on the current grid.

    Returns
    -------
    diffgrid : float
        difference computed in the L-1 norm.
    """

    N_current = len(u_current[:,0])
    N_fine = len(u_fine[:,0])

    grid_size_ratio = int(ceil(N_fine/N_current))

    diffgrid = dt * numpy.sum( numpy.abs(\
            u_current[:,3]- u_fine[::grid_size_ratio,3]))

    return diffgrid
```

Now that the function has been defined, let's compute the grid differences for each solution, relative to the fine-grid solution. Call the function `get_diffgrid()` with two solutions, one of which is always the one at the finest grid. Here's a neat Python trick: you can use negative indexing in Python! If you have an array called `my_array` you access the *first* element with

```
my_array[0]
```

But you can also access the *last* element with

```
my_array[-1]
```

and the next to last element with

```
my_array[-2]
```

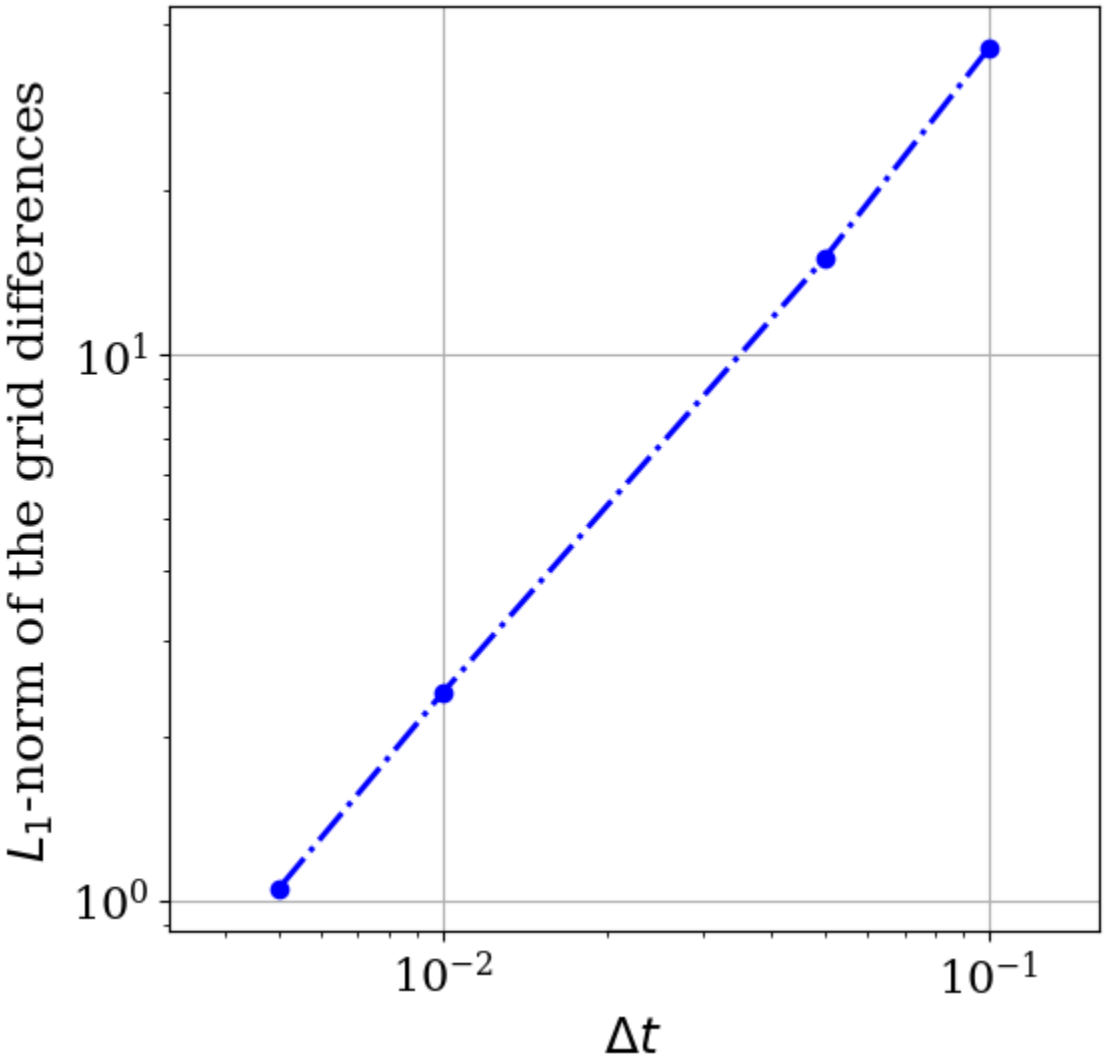
and so on.

In [ ]:
```
# compute difference between one grid solution and the finest one
diffgrid = numpy.empty_like(dt_values)

for i, dt in enumerate(dt_values):
    print('dt = {}'.format(dt))

    ### call the function get_diffgrid() ###
    diffgrid[i] = get_diffgrid(u_values[i], u_values[-1], dt)
```

```
dt = 0.1
dt = 0.05
dt = 0.01
dt = 0.005
dt = 0.001
```

Time to create a plot of the results! We'll create a *log-log* plot with the Matplotlib function `loglog()` . Remember to skip the difference of the finest-grid solution with itself, which is zero.

In [ ]:
```
# log-log plot of the grid differences
pyplot.figure(figsize=(6,6))
pyplot.grid(True)
pyplot.xlabel('$\Delta t$', fontsize=18)
pyplot.ylabel('$L_1$-norm of the grid differences', fontsize=18)
pyplot.axis('equal')
pyplot.loglog(dt_values[:-1], diffgrid[:-1], color='b', ls='-.', lw=2, marker='o');
```



Order of convergence

The order of convergence is the rate at which the numerical solution approaches the exact

one as the mesh is refined. Considering that we're not comparing with an exact solution, we use 3 grid resolutions that are refined at a constant ratio r to find the *observed order of convergence* (p), which is given by:

$$p = \frac{\log\left(\frac{\|f_3 - f_2\|}{\|f_2 - f_1\|}\right)}{\log(r)} \tag{16}$$

where f_1 is the finest mesh solution, f_3 the coarsest and $\|f_3 - f_2\|$ is the distance in norm L_1 between two successive solutions.

The idea is that refining the mesh by a factor r one expects to reduce the observed error dividing by a factor r^p, so the formula extracts this exponent p.

In [ ]:
```
r = 4
h = 0.001

dt_values2 = numpy.array([h, r*h, r**2*h])

u_values2 = numpy.empty_like(dt_values2, dtype=numpy.ndarray)

diffgrid2 = numpy.empty(2)

for i, dt in enumerate(dt_values2):

    N = int(T/dt)   # number of time-steps

    ### discretize the time t ###
    t = numpy.linspace(0.0, T, N)

    # initialize the array containing the solution for each time-step
    u = numpy.empty((N, 4))
    u[0] = numpy.array([v0, theta0, x0, y0])

    # time loop
    for n in range(N-1):

        u[n+1] = euler_step(u[n], f, dt)         ### call euler_step() ###
```

```
    # store the value of u related to one grid
    u_values2[i] = u


#calculate f2 - f1
diffgrid2[0] = get_diffgrid(u_values2[1], u_values2[0], dt_values2[1])

#calculate f3 - f2
diffgrid2[1] = get_diffgrid(u_values2[2], u_values2[1], dt_values2[2])

# calculate the order of convergence
# p = log(diffgrid2[1]/diffgrid2[0]) / log(r)
p = (log(diffgrid2[1]) - log(diffgrid2[0])) / log(r)

print('The order of convergence is p = {:.3f}'.format(p));
```

```
The order of convergence is p = 1.035
```

See how the observed order of convergence is close to 1? This means that the rate at which the grid differences decrease match the mesh-refinement ratio. We say that Euler's method is of *first order*, and this result is a consequence of that.

The error is
$O(\Delta t)^p$
, which means that if
$\Delta t \rightarrow \frac{\Delta t}{10}$
, one has
$error \rightarrow \frac{error}{10^p}$

Paper airplane challenge

Suppose you wanted to participate in a paper-airplane competition, and you want to use what you know about the phugoid model to improve your chances. For a given value of $L/$ L that you can obtain in your design, you want to know what is the best initial velocity and launch angle to fly the longest distance from a given height.

Using the phugoid model, write a new code to analyze the flight of a paper airplane, with the following conditions:

- Assume $L/$ L of 5.0 (a value close to measurements in Feng et al. 2009)
- For the trim velocity, let's take an average value of 4.9 m/s.
- Find a combination of launch angle and velocity that gives the best distance.
- Think about how you will know when the flight needs to stop ... this will influence how you organize the code.
- How can you check if your answer is realistic?

References

- Feng, N. B. et al. "*On the aerodynamics of paper airplanes*", AIAA paper 2009-3958, 27th AIAA Applied Aerodynamics Conference, San Antonio, TX. PDF
- Simanca, S. R. and Sutherland, S. "*Mathematical problem-solving with computers,*" 2002 course notes, Stony Brook University, chapter 3: The Art of Phugoid. (Note that there is an error in the figure: sine and cosine are switched.)