Brief Primer on Statistical and Numerical Methods in Python

1 Ordinary Differential Equations

1.1 Euler's Method

Consider the following first order ordinary differential equation.

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{f}(\boldsymbol{y}, t), \tag{1.1}$$

where $\boldsymbol{y}$ is a vector of variables, t is the independent variable, and $\boldsymbol{f}$ is some arbitrary vector function of $\boldsymbol{y}$ and t. We can use our definition of derivative to write:

$$\frac{d\boldsymbol{y}}{dt} \approx \frac{\boldsymbol{y}_{i+1} - \boldsymbol{y}_i}{\Delta t} = \boldsymbol{f}(\boldsymbol{y}_i, t_i), \tag{1.2}$$

where $\Delta t = t_{i+1} - t_i$. This is not the only choice that could have been made, it is also possible to write it as

$$\frac{\boldsymbol{y}_{i+1} - \boldsymbol{y}_i}{\Delta t} = \boldsymbol{f}(\boldsymbol{y}_{i+1}, t_{i+1}). \tag{1.3}$$

The difference between these two is the choice of either t_i or t_{i+1} on the right hand side. Equation (1.3) gives rise to implicit methods which are harder to code up, but offers potentially greater stability and speed. Instead, we will focus on equation (1.2).

If we know the value at $\boldsymbol{y}(t_i)$, we can solve for $\boldsymbol{y}(t_{i+1})$ to be

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \boldsymbol{f}(\boldsymbol{y}_i, t_i)\Delta t. \tag{1.4}$$

This method is known as Euler's method. As an aside, it we take $\boldsymbol{f}$ to be a scalar function of just t, then it just becomes an ordinary integral. Hence, the methodology we describe here is directly relevant for numerical integration so we don't need a separate discussion for it.

Equation (1.4) gives the correct answer for $\Delta t \to 0$, but it should never be used solving any equations you code up on a computer as superior methods abound. But it is easy and simple to code up and introduces the idea of generic algorithms.

1.2 Second Order Runge-Kutta Method

Runge-Kutta methods are an example of predictor-corrector methods. That is, it "predicts" the value at $\boldsymbol{y}_{i+1}$ from the current solution at $\boldsymbol{y}_i$. Using this predicted value, it performs a "corrector" step to increase the accuracy of the solution. The generic two-step Runge-Kutta method is as follows:

$$\begin{aligned} \boldsymbol{k}_1 &= \Delta t \boldsymbol{f}(\boldsymbol{y}_i, t_i) & (1.5)\\ \boldsymbol{k}_2 &= \Delta t \boldsymbol{f}(\boldsymbol{y}_i + \beta \boldsymbol{k}_1, t_i + \alpha \Delta t) & (1.6)\\ \boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + a\boldsymbol{k}_1 + b\boldsymbol{k}_2 & (1.7) \end{aligned}$$

where $\boldsymbol{k}_1$ is the "predictor" and is the same as an Euler step, $\boldsymbol{k}_2$ is the "corrector", and the $i+1$ step is some linear combination of the two. The constants, α, β, a, and b are chosen to make the entire algoritm accurate to $\mathcal{O}(\Delta t^3)$. To determine these unknown constants, let perform a Taylor expansion of $\boldsymbol{y}_{i+1}$

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \frac{d\boldsymbol{y}}{dt}(t_i)\Delta t + \frac{1}{2}\frac{d^2\boldsymbol{y}}{dt^2}(t_i)\Delta t^2 \tag{1.8}$$

Now

$$\begin{aligned} \frac{d^2\boldsymbol{y}}{dt^2}(t_i) &= \frac{d\boldsymbol{f}(\boldsymbol{y},t)}{dt} = \frac{\partial \boldsymbol{f}(\boldsymbol{y},t)}{\partial t} + \frac{d\boldsymbol{y}}{dt}\cdot\boldsymbol{\nabla}_{\boldsymbol{y}}\boldsymbol{f} & (1.9)\\ &= \frac{\partial \boldsymbol{f}(\boldsymbol{y},t)}{\partial t} + \boldsymbol{f}\cdot\boldsymbol{\nabla}_{\boldsymbol{y}}\boldsymbol{f} & (1.10)\end{aligned}$$

Thus we have

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \boldsymbol{f}(\boldsymbol{y}_i,t_i)\Delta t + \frac{1}{2}\left(\frac{\partial \boldsymbol{f}(\boldsymbol{y}_i,t_i)}{\partial t} + \boldsymbol{f}(\boldsymbol{y}_i,t_i)\cdot\boldsymbol{\nabla}_{\boldsymbol{y}}\boldsymbol{f}(\boldsymbol{y}_i,t_i)\right)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{1.11}$$

Now we Taylor expand out $\boldsymbol{k}_2$ to find

$$\begin{aligned} \boldsymbol{k}_2 &= \Delta t \boldsymbol{f}(\boldsymbol{y}_i + \beta\boldsymbol{k}_1, t_i + \alpha\Delta t) & (1.12)\\ &= \Delta t\left(\boldsymbol{f}(\boldsymbol{y}_i,t_i) + \alpha\Delta t\frac{\partial \boldsymbol{f}(\boldsymbol{y}_i,t_i)}{\partial t} + \beta\Delta t\boldsymbol{f}\cdot\boldsymbol{\nabla}_{\boldsymbol{y}}\boldsymbol{f}(\boldsymbol{y}_i,t_i)\right) & (1.13)\end{aligned}$$

Putting this all together, we have

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + (a+b)\Delta t\boldsymbol{f}(\boldsymbol{y}_i,t_i) + b\Delta t^2\left(\alpha\frac{\partial \boldsymbol{f}(\boldsymbol{y}_i,t_i)}{\partial t} + \beta\boldsymbol{f}\cdot\boldsymbol{\nabla}_{\boldsymbol{y}}\boldsymbol{f}(\boldsymbol{y}_i,t_i)\right) + \mathcal{O}(\Delta t^3) \tag{1.14}$$

Comparing Equations (1.11) and (1.14), we get the following conditions:

$$a+b=1 \qquad b\alpha = \frac{1}{2} \qquad b\beta = \frac{1}{2}, \tag{1.15}$$

or 3 equation for 4 unknowns. So that mean there exist a infinite number of second order schemes that are possible, e.g., error per step that goes likes Δt^3, so the total error over an interval goes like Δt^2. So using α as a parameter, we have

$$\beta = \alpha \qquad b = \frac{1}{2\alpha} \qquad a = 1 - \frac{1}{2\alpha} \tag{1.16}$$

So a generic second order Runge-Kutta scheme is then

$$\begin{aligned} \boldsymbol{k}_1 &= \Delta t\boldsymbol{f}(\boldsymbol{y}_i,t_i) & (1.17)\\ \boldsymbol{k}_2 &= \Delta t\boldsymbol{f}(\boldsymbol{y}_i + \alpha\boldsymbol{k}_1, t_i + \alpha\Delta t) & (1.18)\\ \boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + \left(1 - \frac{1}{2\alpha}\right)\boldsymbol{k}_1 + \frac{\boldsymbol{k}_2}{2\alpha} & (1.19)\end{aligned}$$

A few famous examples are

- Midpoint method: $\alpha = 1/2$ Estimate the values of y at the midpoint and solve for the derivative at the midpoint. Use this midpoint derivative to complete the integration. Note that prefactor in front of $\boldsymbol{k}_1$ in this case is zero.
- Heun's Method: $\alpha = 1$ Estimate the values of y at the endpoint and give equal weight to both starting and endpoints to compute the derivative.

It turns out that for the most part this is all you really need. We should use the generic ode solvers that come with scipy generally.

1.3 Higher Order ODEs

Thus far we have discussed the case of first order odes. What about higher order ODEs. It turns out that there is a very simple extension to arbitrary high order ODEs. The trick is it identify higher order derivatives as variables in themselves. Consider the ODE

$$\sum_{n}^{N} \frac{d^n f}{dx^n} = 0 \tag{1.20}$$

We can write this as a sum first order ODEs by the identification of

$$f_i = \frac{df_{i-1}}{dx} \qquad \text{and} \qquad f_0 = f \tag{1.21}$$

Thus we have

$$\begin{align}
\frac{df_{N-1}}{dx} + \sum_{i}^{N-1} f_i = 0, \tag{1.22}\\
\frac{df}{dx} = f_1 \tag{1.23}\\
\frac{df_1}{dx} = f_2 \tag{1.24}\\
.... \tag{1.25}\\
\frac{df_{N-2}}{=} f_{N-1}. \tag{1.26}
\end{align}$$

So this converts a Nth order ODE to N first order ODEs, which we can solve.

1.4 Timestepping

One thing that we have not discussed is the choice for Δt. For an interval between t_0 and t_1, a larger Δt results in fewer computational steps, which makes things faster. However, a smaller Δt results in greater accuracy. There is a limit with higher order methods on how accurate you can make a solution.

But there is another subtle issue that can happen. In many instances the right hand side of an ODE can take on large (positive or negative) values for a limited set of circumstances. In these cases, it is useful to have a variable Δt – small when things change quickly and large when things change slowly. How can we estimate when these occurs.

Suppose you have an ODE of the form

$$\frac{\partial y}{\partial t} = f(y,t) \tag{1.27}$$

Then according to Euler's method we have

$$y_{n+1} - y_n = f(y_n, t_n)\Delta t \tag{1.28}$$

Now suppose we want the change in $\Delta y = |y_{n+1} - y_n| < \alpha|y_n|$. This means that

$$\alpha|y_n| = |f(y_n, t_n)|\Delta t_{\max} \tag{1.29}$$

This allows us to solve for $\Delta t_{\max}$ to be

$$\Delta t_{\max} = \alpha \left| \frac{y_n}{f(y_n, t_n)} \right| \tag{1.30}$$

Typically, you don't want α to be too large nor too small. I have found values between 0.01 and 0.1 to work well.

At the same time, we don't want to miss something if $\Delta t_{\max}$ is too large that it totally misses a change. In this case, I like to pick a $\Delta t_{\max,0} = (t_1 - t_0)/N_0$, where N_0 is a number between 10 or 100, but this can change as well.

So a selection for Δt at a time t would be

$$\Delta t = min(t_1 - t, min(\Delta t_{\max,0}, \Delta t_{\max})) \tag{1.31}$$

Now lets try an example of this.

2 Optimization and Parallelization

2.1 Optimization

There are two kinds of optimization. There is optimizing human time and effort and optimizing machine time and effort. Generally human time and effort is far more valuable than machine time and effort which is why we have use python in this class even though it is thousands of times slower than C or fortran. The reason is that it is far cleaner and easier to use and link up to libraries than C or fortran.

However, this is not always the case and so we should discuss a few ways to do optimization and parallelization of code. In this section we will discuss python optimization. The key fact about python optimization is several-fold. But before optimizing you should consider the following questions

1. Is your code correct?
2. Do you need to optimize?
3. Do you really need to optimize?
4. Optimize is not parallelization – usually do this last.
5. Optimization involves tradeoffs. Be careful what you wish for.

There are a few steps to optimization:

1. profile
2. profile again.
3. check the hotspots.
4. payoff in optimization: modify your use case, use better algorithms, use builtin functions, use numba, pre-compiled code

So at this point, you have decide to optimize. We will take the N-body problem we discussed earlier as a starting point. Jupyter notebooks has a really useful magic function for profiling called %prun. Lets see this in action.

Go to Jupyter notebook.

Now we will optimize this function in several ways. These are in order:

1. Writing optimal python – using numpy functions whenever possible – for this case I got a speedup of 4-5x
2. Using Numba – this works well with numpy code. It use a decorator @jit or @njit which using a just-in-time compiler for great speedups. This is super easy, but also you have no control over what it does, so the result can be good or horrible.

3. Using cython – this requires some knowledge of c and data types that is native to computers.

4. Using fortran and f2py – I got a 600x speedup with this.

For fortran and cython, it is important to keep in mind that c and to some extent fortran has been the dominant language/scheme of computing over the last 50 years and so to some degree processors are designed to work with these languages. At one point, there was designs to build cpus optimized for lisp (scary thought). Let think about how a cpu works.

A cpu consists of a integer unit and a floating point unit. Early cpus prior to the pentium are mostly an integer unit and floating point was slow. But these days integer and floating point computations are similarly fast. So the native data types that give great speed are int and float (or double). So you want to map it to these things whenever you can.

So the key idea for cython is to take your python code and judiously use “cdef int”, “cdef double”, or “cdef double []” in the correct places to greatly speed up the code.

2.2 Parallelization

The choice of python for this course was unfortunate in one crucial aspect and that is the ability of python to be parallelized easily on one machine. Modern cpus have >4 cores and thus the ability to use more than one core for computation are a real boon. Python is especially limited in this respect, but there are parallelization methods that we can discuss for python.

The simpliest parallelization strategy that you will encounter and probably the most common is “embarassingly” parallel. This occurs is many situations such as large data analysis or parameter studies. Generally the paradigm that one should think about for “embarassingly” parallel problems is something called MapReduce. Here we map a computation to a large number of computers/cores and then reduce the information to a simpler data set at the end. To see how this works, let us consider the counting of the numbers of words in “War and Peace”. You could do this by sitting down and counting the entire book, or you can assign each page to a different person. Each person will count the words on the page and you can just add up all the counts together. The assignment of a person to a task is the map and the collation of information returned is the reduce part.

There are several ways of doing map-reduce. Here are a few off the top of my head:

- shell-script
- gnu-parallel
- python multiprocessing pool
- python MPI
- condor/open science grid

Lets discuss a few in turn.

Here is a example shell script

```bash
#!/bin/bash

do-job-1 &
do-job-2 &
wait
```

Pretty simple. We do 2 jobs and we wait until it is finish. Very simple, but it can be tedious to code especially for large number of jobs.

Here is an example for gnu-parallel

```bash
#!/bin/bash

seq 1 1 100 | parallel -j 8 python job-name {}
```

seq counts from 1 to 100 in steps of 1 and parallel execute up to 8 jobs at once wher the label in {} is the number labeled by seq. Great way to doing an arbitrary number of jobs on a cluster. gnu-parallel is usually used on one node, but multiple nodes can use used if you know how.

There are a few ways of doing multiprocessing on python – having a single python instance launch multiple instances and run different things on them. Here I will introduce multiprocessing pools as I have experience with them. Ideally, you should use python executors as this much more future proof, but the future proof part is fairly far away.

```python
def func( x) :
    print(x)

import multiprocessing as mp
with mp.Pool(processes=4) as pool :
    result = pool.map(func, np.arange(100))
```

This is awesome, map and reduce in essentially one one. It returns the result as a list in the same order of the original map and thus is can be reduces as soon as one prefers.

Finally lets talk a bit about mpi. The structure of MPI is as follows. At startup N programs launch and are able to communicate with each other. On startup you will get a comm object,

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

The rank is the identifier of your process. rank==0 is the first program and usually acts as an overseer for everything else. So for instance suppose you want to do a map reduce in MPI.

```python

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
```

Here comm.scatter recognized that the data that is to be mapped comes from the rank == 0 process. All other processes (including root) will then get a subset of data to play with. To reduce, we must do a gather

```python
    data = comm.gather(data, root=0)
    if rank == 0:
        # do something with the data
        pass
```