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:

$$\boldsymbol{k}_1 = \Delta t \boldsymbol{f}(\boldsymbol{y}_i, t_i) \tag{1.5}$$

$$\boldsymbol{k}_2 = \Delta t \boldsymbol{f}(\boldsymbol{y}_i + \beta \boldsymbol{k}_1, t_i + \alpha \Delta t) \tag{1.6}$$

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + a\boldsymbol{k}_1 + b\boldsymbol{k}_2 \tag{1.7}$$

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

$$\frac{df_{N-1}}{dx} + \sum_i^{N-1} f_i = 0, \tag{1.22}$$

$$\frac{df}{dx} = f_1 \tag{1.23}$$

$$\frac{df_1}{dx} = f_2 \tag{1.24}$$

$$.... \tag{1.25}$$

$$\frac{df_{N-2}}{=} f_{N-1}. \tag{1.26}$$

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

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

```
#!/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

```
#!/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.

```
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,

```
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.

```

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

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

3 Partial Differential Equations

Partial Differential equations (PDEs) are the heart of many physical systems that we are interested in. We will study three main classes of PDEs

1. Hyperbolic
2. Elliptic
3. Parabolic

Hyperbolic: Hyperbolic PDEs characterized by real distinct propagation speeds. As such their usual physical intepretation involves a state that evolves in time in accordance to a known signal speed. An example of this is the wave equation:

$$\frac{\partial^2 f}{\partial t^2} - \frac{\partial^2 f}{\partial x^2} = 0 \tag{3.1}$$

To solve these equations they require boundary conditions in space and initial conditions in time. The most typical example of hyperbolic equations are the compressible fluid equations.

Elliptic: Elliptic PDEs characterized by effectively infinite propagation speeds. As such they require boundary conditions everywhere as their solution relies on the BCs. Their solutions are also smooth. An example of this is the Poisson equation:

$$\frac{\partial^2 f}{\partial t^2} + \frac{\partial^2 f}{\partial x^2} = g \tag{3.2}$$

In addition to Poisson, other examples of elliptic equation are electrostatics, (Newtonian) gravity, etc. Incompressible fluid flow also has an elliptic nature as the incomoressibility conditions is elliptic.

Parabolic: Parabolic PDEs somewhat between hyperbolic and elliptic equations. They do propagrate in time so only require boundary conditions on the spatial part and while they can allows sharp solutions, it likes to smooth it out. An example is the diffusion equation"

$$\frac{\partial f}{\partial t} - \frac{\partial^2 f}{\partial x^2} = 0 \tag{3.3}$$

The origin of the name comes from a classification of conic sections. For instance for a general PDE:

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + f = g \tag{3.4}$$

It is hyperbolic if $b^2 - 4ac > 0$, elliptic if $b^2 - 4ac < 0$ and parabolic if $b^2 - 4ac = 0$.

Now we already know how to solve ODE problems so our goal here is to convert PDEs to ODEs and use the standard techniques to solve them. There is no general way of converting an arbitrary PDEs to an ODE though this can be done for certain problems by defining a new variable that mixes two or more of the independent variables, e.g., self-similar methods. However, this works for a very special subset of problems and don't work generally.

As a result, the usual method for solving (hyperbolic and parabolic) PDEs is to descretized space and approximate the spatial derivatives on that space and use ODE solvers to advance the solution in time. There are a number of possible ways to do this.

1. finite difference: values of a function are stored at discrete points – replace derivatives with
2. finite volume: the values of a function are average over the volume centered around a grid point. Because of this, the methods here involve replacing differentiation with integration of a flux over a the boundary of the volume.
3. finite element: kinda like spectral methods, but with compact basis functions.
4. spectral methods: decompose the values of a function on space to Fourier components. Solve for the evolution of the Fourier components. Amazing for smooth flows - exponential convergence.
5. particle methods: break up space into discrete sampled points the evolve at some velocity – only really useful for hyperbolic equations.

3.1 Advection and Hyperbolic Problems

We will begin first with the linear advection problem as many hyperbolic problems can be rewritten in a manner similar to advection.

$$\frac{\partial f(t,x)}{\partial t} + v\frac{\partial f(t,x)}{\partial x} = 0, \tag{3.5}$$

where v is some propagating speed. We will assume some initial condition $f(0,x)$ and lets assume periodic boundary conditions $f(t,0) = f(t,L)$. The solution to this problem is trivial: $f(t,x) = g(x-vt)$ for any arbitrary function g. Because such a simple analytic solution exists, these is an ideal test case to test the error of whatever method we bring to bear.

3.1.1 Finite Difference

The first way we will look at this is via finite difference. Lets try a very simple discretization:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} - v\frac{f_{i+1}^n - f_{i-1}^n}{2\Delta x} = 0 \tag{3.6}$$

This is centered differencing in space and first order differencing in time or FTCS. For a constant v, we can write this as

$$f_i^{n+1} = f_i^n - \frac{v\Delta t}{2\Delta x}\left(f_{i+1}^n - f_{i-1}^n\right) = f_i^n - \frac{\alpha_{\text{CFL}}}{2}\left(f_{i+1}^n - f_{i-1}^n\right), \tag{3.7}$$

where $\alpha_{\text{CFL}} = v\Delta t/\Delta x$ is called the Courant-Friedrichs-Levy (CFL) number. By setting this number, we force what Δt will be. This is the timestep. For stability, $\alpha_{\text{CFL}} < 1$ and can

be much smaller than unity. This is the same statement as information does not propagate more then one cell at a time.

We write a simple code to evolve the advection equations here. It evolves a simple gaussian or tophat profile across a period box of length $L = 1$ at a velocity of $v = 1$. The CFL number is 0.8. We will present the code later on, but the evolution part looks like this:

```
def ftcs(f, dt, dx, cfl=cfl) :
    pass
```

If we evolve this, we see that the code appears stable for a little while before it falls apart. For the tophat profile the situation is dire immediately.

Now the issue is the FTCS is not stable and, in fact, it is what we call unconditionally unstable, which is a bad state of affairs. The reason for this is that information in flows moves from upstream to downstream, but the ftcs scheme above allows information to flow from downstream to upstream. So like the future affecting the past, this is now allowed and results in the destruction of the computational universe.

To resolve this lets define two first order spatial derivatives

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} - v\frac{f_{i+1}^n - f_i^n}{\Delta x} = 0 \qquad \text{for } v < 0 \tag{3.8}$$

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} - v\frac{f_i^n - f_{i-1}^n}{\Delta x} = 0 \qquad \text{for } v > 0 \tag{3.9}$$

where we pick the upwind difference, e.g., we look at which way the flow is coming.

Lets look at the complete code now.

```
import numpy as np
import matplotlib.pyplot as pl
import math

cfl = 0.8
v = 1
L = 1

FTCS = 0
UPWIND = 1

TOPHAT = 0
GAUSSIAN = 1
PROFILE = GAUSSIAN

MODE = UPWIND
MOVIE = False

def ftcs(f, dt, dx, cfl=cfl) :
  newf = f
  newf[1:-1] -= v*(f[2:] - f[0:-2])/(2*dx)*dt
  newf[0] -= v*(f[1] - f[-1])/(2*dx)*dt
  newf[-1] -= v*(f[0] - f[-2])/(2*dx)*dt
  return newf
```

```
25
26 def upwind(f, dt, dx, cfl=cfl) :
27   newf = f
28   newf[1:] -= v*(f[1:] - f[0:-1])/dx*dt
29   newf[0] -= v*(f[0] - f[-1])/dx*dt
30   return newf
31
32 def initial_conditions(x) :
33   N = x.size
34
35   f = np.ones(N)*0.5
36
37   # top hat
38   if( PROFILE == TOPHAT) :
39     f[int(0.25*N):int(0.75*N)] = 1 # top hat profile
40   elif( PROFILE == GAUSSIAN) :
41     # gaussian profile
42     sigma = 0.125*L
43     f += np.exp(-(x-x.mean())**2/sigma**2)
44
45   return f
46
47 def error(f, ftrue) :
48   return np.sqrt(np.average((f - ftrue)**2))
49
50 def run_model(MODE, N=100) :
51
52   dx = L/N
53
54   tend = 1
55   t = 0
56   x = np.arange(0,L,dx)
57
58   f = initial_conditions(x)
59   i = 0
60
61   while( t < tend) :
62     dt = min(tend-t,math.fabs(cfl*dx/v))
63     if( MODE == FTCS) :
64       f = ftcs(f, dt, dx, cfl=cfl)
65     else :
66       f = upwind(f, dt, dx, cfl=cfl)
67     t += dt
68
69     if MOVIE :
70       i += 1
71       pl.clf()
72       pl.plot( x, f)
73       pl.ylim(0.,2.0)
74       print("Writing frame {0}".format(i))
75       pl.savefig("movie/frame{0:04d}.png".format(i))
76
77   if MOVIE:
78     pl.clf()
```

```
    pl.plot( x, f)
    pl.ylim(-0.5,1.5)
    pl.savefig("advect.pdf")
  return f

if __name__ == "__main__" :
  import argparse
  parser = argparse.ArgumentParser(description='Run advection')
  parser.add_argument('--run_error', action='store_true', default=False,
                      help='make a graph of E vs T')

  parser.add_argument('--ftcs', action='store_true', default=False,
                      help='make a graph of E vs T')
  parser.add_argument('--movie', action='store_true', default=False,
                      help='make a graph of E vs T')

  args = parser.parse_args()

  if args.ftcs :
    MODE = FTCS
  else :
    MODE = UPWIND

  if not args.run_error :
    MOVIE = args.movie
    run_model(MODE)
  else :
    lgNs = np.arange(1.3,3.3,0.25)
    errors = []
    for lgN in lgNs :
      N = int(1e1**lgN)
      dx = L/N

      x = np.arange(0,L,dx)

      ftrue = initial_conditions(x)
      f = run_model(MODE,N=N)
      errors.append( error(f,ftrue))

    pl.clf()
    pl.loglog(1e1**lgNs,errors)
    pl.savefig("advect_conv.pdf")
```

There are other methods to do the solutions other than upwinding. These include Lax-Friedrichs and Lax-Wendroff, but we will move onto finite volume methods.

Before doing so, we should discuss convergence. Convergence is the measure of how accurately a numerical method reproduces the exact solution. In the case of ODEs, we care about the accuracy of

$$\text{Err} = 2\left|\frac{f_{\text{num}}(t_{\text{end}}) - f_{\text{analytic}}(t_{\text{end}})}{f_{\text{num}}(t_{\text{end}}) + f_{\text{analytic}}(t_{\text{end}})}\right| \tag{3.10}$$

So the error is taken only at the end point. For hyperbolic equations, we can think of this

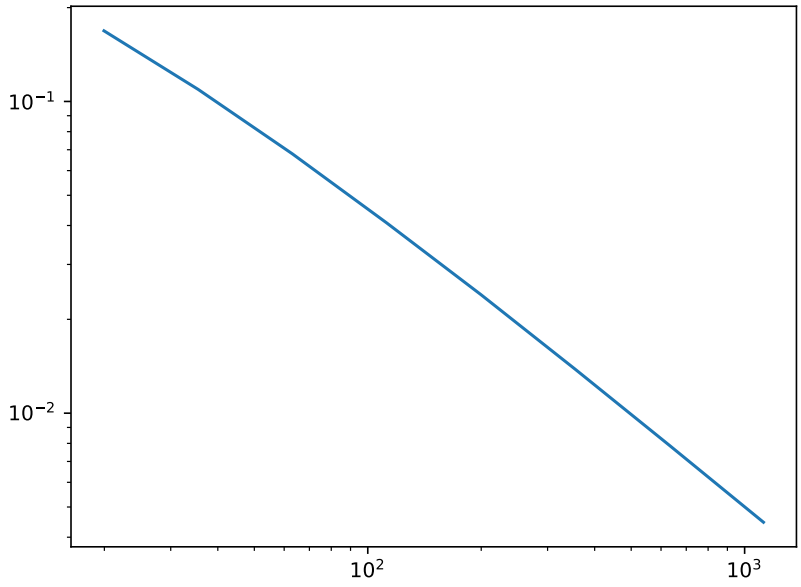

Figure 1

as a set of ODEs and so we the error is

$$\text{Err} = 2N^{-1} \sum_i \left| \frac{f_{i,\text{num}}(t_{\text{end}}) - f_{i,\text{analytic}}(t_{\text{end}})}{f_{i,\text{num}}(t_{\text{end}}) + f_{i,\text{analytic}}(t_{\text{end}})} \right| \tag{3.11}$$

This is actually called the L1 norm. Another version if the L2 norm, which is

$$\text{Err} = 2\sqrt{N^{-1} \sum_i \left(\frac{f_{i,\text{num}}(t_{\text{end}}) - f_{i,\text{analytic}}(t_{\text{end}})}{f_{i,\text{num}}(t_{\text{end}}) + f_{i,\text{analytic}}(t_{\text{end}})} \right)^2} \tag{3.12}$$

I don't have any advice on which one to choose, but for this version we will look at the L2 norm. This gives a error as a function of N that goes N^{-1}

3.2 Elliptic Problems

For elliptic problems, the speed of the propagation is infinite and thus responds instantly to any change the source or boundary conditions. For these sorts of problems, relaxation methods work well, but techniques like multigrid can speed things up.

Lets consider a problem that arises in astrophysics and electrostatics, the Poisson equation

$$\nabla^2 \Phi = f \tag{3.13}$$

Let's consider the 1-d version first

$$\frac{\partial^2 \Phi}{\partial x^2} = f \tag{3.14}$$

with boundary conditions $\Phi(0) = \Phi(1) = 0$. If we pick $f = \sin(x)$, then we have an analytic solution $\Phi = -\sin(x) + x\sin(1)$. Given this, how can we solve for this numerically.

Fortunately, this is an ODE and so the first technique that we will try is using one of our ODE integrators – say rk2. Now if we use rk2, we start our and $x = 0$ and integrate to $x = 1$. Since $\Phi(x = 0) = 0$, we have one initial condition already, but we will need another initial condition for $\Phi'(x = 0)$. We can set this to be a free value, say α and vary it until we get the second boundary condition.

In other works, lets define a function $g(\alpha)$ such that

$$g(\alpha) = \Phi(x = 1; \alpha), \tag{3.15}$$

where Φ is computed by numerical integration to $x = 1$ using $\Phi'(x = 0) = \alpha$. Since we want $g(\alpha) = 0$, this reduces to a root-finding problem for α. So once we define $g(\alpha)$, we can use root finding routines to find the appropriate value of α. For this, we will use a standard python package. This technique is known as shooting as you are essential shooting until you hit a target.

```
import numpy as np
import matplotlib.pyplot as pl
import math
import rk2
import newton_raphson as nr

N = 100
L = 1

def derivatives(x, y) :
  Phi = y[0]
  dPhidx = y[1]
  dydx = np.zeros(y.size)
  dydx[0] = dPhidx
  dydx[1] = math.sin(x)
  return dydx

def g(alpha, output=False) :
  x = 0
  dx = L/N
  y = np.zeros(2)
  y[1] = alpha
  yout = []
  xout = np.arange(0,1,dx)
  for x in xout:
    y = rk2.rk2(x, x+dx, dx, y, derivatives)
    if( output) :
      yout.append(y[0])
  if( output) :
    return xout, yout
  return y[0]


if __name__ == "__main__" :
```

```
  alpha, error, iterations = nr.newton_raphson(g, 1.)
  x, y = g(alpha, output=True)
  pl.plot(x,y, lw=2)
  pl.show()
```

Shooting doesn't always work especially with stiff equations which are extremely sensitive to initial conditions. Examples of this in astrophysics include hydrostatic balance for stars. In this case, we need to try something different. Lets discretize the equation to be

$$\frac{\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}}{\Delta x^2} = f_i, \tag{3.16}$$

where we pick $\Phi_0 = \Phi_N = 0$. We then have a bunch of algebraic equations:

$$\Phi_i = 0.5\left(\Phi_{i+1} + \Phi_{i-1} - \Delta x^2 f_i\right). \tag{3.17}$$

In principle, we can solve with matrix inversion, but it is easier to solve using relaxation starting with some initial guess Φ_i^0 This can use either

1. Jacobi iteration: $\Phi_i^{k+1} = 0.5\left(\Phi_{i+1}^k + \Phi_{i-1}^k - \Delta x^2 f_i\right)$.
2. Gauss-Seidel iteration: Use the new $k+1$ values as they appear.

In either case, you need to keep track of the error to ensure that the error goes down below some threshold.

Lets look at the code here.

```
import numpy as np
import matplotlib.pyplot as pl
import math

N = 100
L = 1
TINY = 1e-10
MAXERR = 1e-7

def Jacobi(Phi, f) :
  dx = L/(Phi.size-1)
  Phi[1:-1] = 0.5*(Phi[:-2] + Phi[2:] - dx*dx*f[1:-1])
  return Phi

def redBlackGaussSeidel(Phi, f) :
  dx = L/(Phi.size-1)
  Phi[2:-1:2] = 0.5*(Phi[1:-2:2] + Phi[3::2] - dx*dx*f[2:-1:2])
  Phi[1:-1:2] = 0.5*(Phi[:-2:2] + Phi[2::2] - dx*dx*f[1:-1:2])
  return Phi

def error(Phi1, Phi2):
  return np.abs((Phi2-Phi1)[1:-1]/(0.5*np.abs(Phi1+Phi2)+TINY)[1:-1]).sum
    ()

def init() :
```

```
  Phi = np.zeros(N+1)
  dx = L/(Phi.size-1)
  x = np.arange(Phi.size)*dx
  f = np.sin(x)
  return x, Phi, f

if __name__ == "__main__" :
  x, Phi, f = init()

  err = 1
  iterations = 0
  while err > MAXERR :
    iterations += 1
    Phi2 = Phi.copy()
    #Phi2 = Jacobi(Phi2, f)
    Phi2 = redBlackGaussSeidel(Phi2, f)

    #print(Phi2)
    err = error(Phi,Phi2)
    if( iterations % 1000 == 0) :
      print("Iteration: {0} {1:.3e}".format(iterations, err))
    Phi = Phi2

  print("Total iteration: {0} {1:.3e}".format(iterations, err))
  pl.scatter(x, Phi,s=1,label="numerical")
  pl.plot(x,-np.sin(x)+x*np.sin(1), label="analytic")
  pl.legend(loc="best")
  pl.savefig("relax.pdf")
```

which yields the following compared to the analytic result:

Running this you can see that theres are quite a few iteration needs to solve this equation. This can be accelerated by coarsening and then refining the grid in a technique known as multigrid. We won't cover this in this as in astrophysics there are other ways to solve this equation (FFTs for instance).

3.3 Parabolic Problems

As stated above, the parabolic problem has properties of both the hyperbolic and elliptic problem. In particular, we will see that it can have very large propagation speeds like the elliptic problem, but can be solve in a flux-conservative way as in the hyperbolic problem. Lets start with a prototypical problem

$$\frac{\partial \Phi}{\partial t} = \frac{\partial^2 \Phi}{\partial x^2} \tag{3.18}$$

An identification of the flux to the $F = -\partial\Phi/\partial x$ allows us to write this like the flux-conservative equation

$$\frac{\partial \Phi}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{3.19}$$

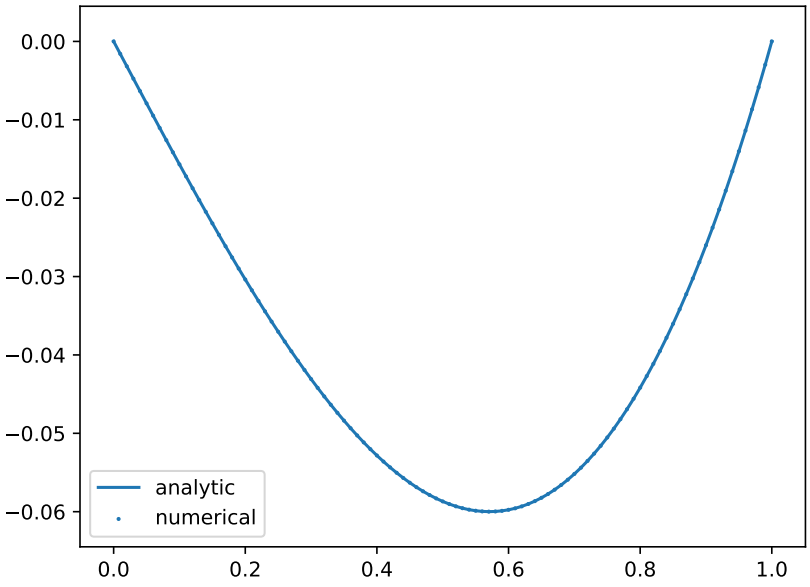

Figure 2

If we presume the analytic ansatz

$$\Phi(t,x) = \frac{\exp -x^2/4t}{t^{1/2}} + \Phi_0, \tag{3.20}$$

we see that it solve the diffusion equation (3.18). Lets try to solve this problem numerically. Lets use a standard centered expression for the second derivative in Space

$$\frac{\Phi_i^{n+1} - \Phi_i^n}{\Delta t} = \frac{\Phi_{i+1}^n - 2\Phi_i^n + \Phi_{i-1}^n}{\Delta x^2} \tag{3.21}$$

We can examine its stability in space with a single Fourier mode $\Phi_i^n = A^n \exp(-i2\pi x_i/L)$. This gives

$$\left|\frac{A^{n+1}}{A^n}\right| = \left|1 + 2\frac{\Delta t}{\Delta x^2}\left(\cos(2\pi\Delta x/L) - 1\right)\right| < 1 \tag{3.22}$$

which implies that $\Delta t \propto \Delta x^2/2$, e.g., the equivalent "Courant" number is $\alpha_{\rm CFL} = 2\Delta t/\Delta x^2$ So the timestep rapidly comes down as Δx (higher resolution) shrinks.

We can numerically solve this once we specific the boundary conditions over a finite domain. However, we will note that the analytic solution spreads from $x = (-\infty, \infty)$. However, if we put the boundaries far enough away, it will not "pollute" the solution too much in the region of interest. Here the fact that the timescale goes like $t \sim L^2$ helps.

Now lets examine the code in detail:

```
1 import numpy as np
2 import matplotlib.pyplot as pl
```

```
import math

N = 1000
xmax = 10
TINY = 1e-10
MAXERR = 1e-3

def applyBC( Phi) :
  Phi[0] = Phi[-2]
  Phi[-1] = Phi[1]

def evolve(Phi, t1, t2, dt, dx) :
  t = t1
  while(t < t2) :
    dt = min(dt, t2-t)
    newPhi = Phi.copy()
    dx2 = dx*dx
    newPhi[1:-1] += dt/dx2*(Phi[0:-2] - 2*Phi[1:-1] + Phi[2:])
    applyBC(newPhi)
    Phi = newPhi
    t += dt

  return Phi

def init(N=N,t0=1e-2,Phi0=0.1) :
  Phi = np.zeros(N+2)
  dx = 2*xmax/N
  x = np.arange(-xmax+dx/2, xmax, dx)
  Phi[1:-1] = np.exp(-np.minimum(x*x/(4*t0),100))/t0**0.5 + Phi0
  applyBC(Phi)
  return x, dx, Phi

if __name__ == "__main__" :
  x, dx, Phi = init()

  iframe = 0
  cfl = 0.8
  dt = cfl*dx**2*0.5
  tstep = 0.1
  t = 0
  tend = 10
  while( t < tend) :
    Phi2 = evolve(Phi, t, t+tstep, dt, dx)
    Phi = Phi2
    t += tstep
    print("iframe: {0} t={1:.3f}".format(iframe,t))
    pl.clf()
    pl.plot(x, Phi[1:-1],lw=2)
    pl.xlim(-xmax, xmax)
    pl.ylim(0,4.)
    pl.savefig("movie/frame{0:04d}.png".format(iframe))
    iframe += 1
```

4 Statistical Methods

We will discuss statistical methods here. Generally, statistical methods underly questions about statistical inference which is the ability to draw conclusions from data. There are generally three questions that we may seek to answer:

1. Parameter Estimation: What is the best estimate for a parameter, θ, given some data.
2. Confidence Estimation: How confident should be be in our parameter estimation.
3. Hypothesis testing: How consistent is a given hypothesis with the available data.

A few of you in astronomy and physics deal with this all the time. A few more of you deal with this as part of your job as a TA or lecturer, after all, what is a grade but a estimate of the amount of student acquired knowledge/skills.

In this context there are two paradigms, the classical or frequentist view and the Bayesian view. In short, the classical view is based on

1. Probabilities are related to the relative frequency of events in the world.
2. Parameters are fixed constants – they don't fluctuate.
3. All things converge in the long-run. A 95% confidence interval means that over the long term, the measure parameter should fall within the 95% confidence interval, 95% of the time.

On the other hand, the Bayesian view is based on.

1. Probability describes the degree of subjective belief – so parameters and models can have probabilities associated with them.
2. Inference is done by looking at the probability distribution. Probability distributions quantify the limits of our knowledge.

So the battle between these two paradigms can be though of as a battle between confidence and distributions. Because distributions are functions and are more difficult to calculate, Bayesian statistics are beset by large computational requirements, something is a familiar to some of you.

4.1 Maximum Likelihood

The first things we will discuss is maximum likelihood estimation. For that we need to come up with a likelihood function:

$$L = p(D|M(\theta)) \tag{4.1}$$

where L is the likelihood, D is the data and $M(\theta)$ is a models with parameters θ. A good likelihood estimator is a Gaussian estimator:

$$p(D|M(\theta)) = \prod_{\text{data}} \frac{1}{\sqrt{\pi}\sigma_i} \exp\left(-\left(\frac{x_i - \mu_i(\theta)}{\sigma_i}\right)^2\right) \tag{4.2}$$

Because of the nature of exponentials, it is easier to deal with the log likelihood which is

$$\log(L) = -\sum_{\text{data}} \left(\frac{x_i - \mu_i(\theta)}{\sigma_i}\right)^2, \tag{4.3}$$

where we have thrown away an additive term which is constant in θ.

The log-likelihood is the function that we have to deal with when fitting functions to data. Here we will take two examples. In the first case, lets assume that we have a constant error $\sigma_i = \sigma$ and lets try to fit an polynomial to it.

Now the polynomial is already baked into numpy as it is used all the time. However, let's do something else. We know that the underlying function is a sine function with noise, so lets fit a sine function to it.

In doing so we will need to compute the error. Here the error or residuals r_i are the unsquared part of the likelihood function:

$$r_i = \frac{x_i - \mu_i(\theta)}{\sigma_i} \tag{4.4}$$

Note that in this case that r_i can be positive and negative.

4.2 χ^2 statistic

Sometimes we want to see how good a model fits the data. In this case, we can define a χ^2 statistic, which is

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{x_i - \mu_i(\theta)}{\sigma_i}\right)^2 \tag{4.5}$$

where N denotes the number of data points. Now χ^2 get larger and larger as the number of data points increase. So we need to define a "normalized" version of χ^2, which is χ^2dof of χ^2 per degree of freedom. For a model with p parameters and N data points the number of degrees of freedom is $N - p$. So

$$\chi^2_{\text{dof}} = \frac{1}{N-k} \sum_{i=1}^{N} \left(\frac{x_i - \mu_i(\theta)}{\sigma_i}\right)^2 \tag{4.6}$$

Now if the error is distributed normally, then for a good model χ_{dof} is about unity. Much larger and this means that model likely sucks given the error. Much smaller might indicate the we don't understand our errors or their are overestimated. Now generally, a order unity χ^2_{dof} does not mean

Go ahead and now compute the χ^2_{dof}. Is it order unity? You should find out that it is much smaller than unity. What does this tell you?

4.3 Markov-Chain Monte-Carlo

Suppose you are given some distribution of the form P and you want to compute the expectation value of y. Then the formal term is

$$\langle y \rangle = \int y dp \tag{4.7}$$

Or in other words

$$\langle y \rangle = \int y P dV, \tag{4.8}$$

where V represents some volume in phase (state) space and $P = dp/dV$ is the probability density function. Now lets consider the case when the state space is descrete like in quantum mechanics

$$\langle y \rangle = \frac{\sum_i y_i P(V_i)}{\sum_i P(V_i)}, \tag{4.9}$$

where the denominator is just ensure that the sums are properly normalized. To make this more concrete, lets consider the case where the state space is energy $V = E$ and P follows a Maxwell-Boltzman distribution. In this case we have

$$\langle y \rangle = \frac{\sum_i y_i \exp(-E_i/k_B T)}{\sum_i \exp(-E_i/k_B T)}, \tag{4.10}$$

where you recognized the denominator as $Z = \sum_i \exp(-E_i/k_B T)$ is the partition function from statistical mechanics.

Suppose now we want to do this sum using monte carlo methods. Here we cannot sum over infinite number of states but over N states so we have.

$$\langle y \rangle = \frac{\sum_{i=1}^{N} y_i \exp(-E_i/k_B T)}{\sum_{i=1}^{N} \exp(-E_i/k_B T)}, \tag{4.11}$$

The issue is that the space of E_i is huge, but for most values of E_i, the contribution is small e.g. $\propto \exp(-E_i/k_B T)$. So we want to pick random values of E_i following a distribution that *maximizes the contribution to the sum.* To do this let consider a expectation value over a weight function w

$$\langle f \rangle_w = \frac{\sum_j f_j w_j}{\sum_j w_j} \tag{4.12}$$

Now lets define the expectation value

$$\left\langle \frac{yP(E)}{w} \right\rangle_w = \frac{\sum_j y_j P(E_j) w_j / w_j}{\sum_j w_j} = \frac{\langle y \rangle Z}{\sum_j w_j} \tag{4.13}$$

or in other words

$$\langle y \rangle = \left\langle \frac{yP(E)}{w} \right\rangle_w Z^{-1} \sum_j w_j \tag{4.14}$$

Now the trick is that I will pick $w = P(E)/Z$ so that I have (when I integrate monte carlo style)

$$\langle y \rangle = N^{-1} \sum_{i=1}^{N} y_i \tag{4.15}$$

which is simply an average over my random points, but the difficulty is that I have to pick from a distribution that follows a PDF that is defined by the weight function $w = P(E)/Z$. I do not necessarily know how to compute the partition function as it involves as integral over all phase space.

At the point, we will use a Markov chain, which allows us to do this integral without exact knowledge of the complete PDF. The way is works is as follows.

1. Starting a state i, make a small change to state j.
2. compute the transition probability ΔP
3. draw a random number r in $[0, 1]$ and accept the move if $r < T_{ij}$; otherwise do nothing

In the case of the weight function that follows a Maxwell-Boltzmann distribution, the transition probability can be computed from detailed balance where

$$P(E_i)R_{ij} = P(E_j)R_{ji} \rightarrow \Delta P = \frac{R_{ij}}{R_{ji}} = \frac{P(E_j)}{P(E_i)} = \exp\left(-\frac{\Delta E_{ij}}{k_B T}\right) \tag{4.16}$$

where R_{ij} are the rates of going from the i-th state to the j-th state

To encode all this, let us introduce the Metropolis-Hasting algorithm.

1. Starting with some state i
2. Generate a set of possible moves.
3. Pick a random move j from the generate set
4. Accept the move with probability

$$P = \min\left(1, \exp(-\frac{\Delta E_{ij}}{k_B T}\right) \tag{4.17}$$

4.4 Bayesian Statistics

The biggest use of MCMC is its application to model fitting especially in the context of Bayesian modeling. Usually in this case, we are concerned about the computation of the posterior distribution function $\pi(\boldsymbol{\theta}; \boldsymbol{x})$ of parameters $\boldsymbol{\theta}$ given some data $\boldsymbol{x}$ and a prior $\pi(\boldsymbol{\theta})$. We do this via Bayes' theorem

$$\pi(\boldsymbol{\theta}; \boldsymbol{x}) = \frac{\pi(\boldsymbol{\theta}) p(\boldsymbol{x}; \boldsymbol{\theta})}{p(\boldsymbol{x})} \tag{4.18}$$

where $p(\boldsymbol{x};\boldsymbol{\theta})$ is a likelihood function and $p(\boldsymbol{x}) = \int \pi(\boldsymbol{\theta})p(\boldsymbol{x};\boldsymbol{\theta})d\boldsymbol{\theta}$ is essentially the normalization. In essence, we want to compute

$$p(\boldsymbol{x}) = \int \pi(\boldsymbol{\theta})p(\boldsymbol{x};\boldsymbol{\theta})d\boldsymbol{\theta} \tag{4.19}$$

The key is that we only want to compute over the values of θ that really contribute. So we can define a Metropolis algorithm that goes for this as

1. Begin with some parameter set $\boldsymbol{\theta}_1$
2. Compute the joint likelihood $L_1 = \pi(\boldsymbol{\theta}_1)p(\boldsymbol{x};\boldsymbol{\theta}_1)$
3. Pick a random proposal $\boldsymbol{\theta}_2$ from a distribution based on the priors.
4. Compute the likelihood $L_2 = \pi(\boldsymbol{\theta}_2)p(\boldsymbol{x};\boldsymbol{\theta}_2)$
5. Compute the transition probability $R = L_2/L_1$
6. Accept the proposal with probability R

Lets consider the computation of the expectation values of the parameters $\boldsymbol{\theta}$ over the distribution given by the posterior distribution, $\pi(\boldsymbol{\theta};\boldsymbol{x})$ or

$$\langle\boldsymbol{\theta}\rangle_\pi = \int \boldsymbol{\theta}\pi(\boldsymbol{\theta};\boldsymbol{x})d\boldsymbol{\theta} \tag{4.20}$$

If I compute this value, I get the expected value of the parameters. If I plot out $\pi(\boldsymbol{\theta})$, I get the distribution of relevant θ's

Thus I can use MCMC for this. A subtle, but important point is that when I take the average over the entire Markov chain, then the values of the parameters should reduce to the expected value. However, if I take the distribution of the parameters, i.e., take a histogram of the values of the parameters in the chain, the distribution of the parameters is the posterior distribution. This is a subtle, but important point and in case you ever wondered how these parameters distributions for various models where ever generated, now you know.

Lets solidify our understanding with a "simple" example. Let us consider our noisy sin wave as before, which we reproduce for clarity

```
def generate_data() :
    A = 5
    phi = np.pi/4
    sigma = 1.0

    theta = np.arange(-2*np.pi, 2*np.pi, 0.025*np.pi)
    y = A*np.sin(theta + phi) + np.random.normal(0, sigma, theta.size)

    return theta, y, np.array([A, phi])
```

Now we will define a log-likelihood function

$$\log p(\boldsymbol{y};\boldsymbol{\theta}) = -\sum_i \left(\frac{y_i - \theta_0 \sin(t_i + \theta_1)}{\sigma} \right)^2, \tag{4.21}$$

where we set $\sigma = 1$ and t is an independent variable. We need to set some priors. Lets set θ_0 to be a flat distribution between 1 and 10 and θ_1 to be a flat distribution between 0 and 2π.

```
def priors( x) :
    if(x[0] < 1 or x[0] > 10) :
      return 0
    if(x[1] < 0 or x[1] > 2*np.pi) :
      return 0
    return 1
```

Here is the rest of the specification. For your HW, you will implement the heart of the algorithm, move and the loglikelihood.

```
def mcmc(func, t, y, x, burn_in=1000, MAX_CHAIN=500000) :
  xchain = []
  i = 0

  for j in range(MAX_CHAIN) :
    x, accepted_move = move(func, t, y, x)
    if( accepted_move) :
      i += 1
      if( i > burn_in) :
        xchain.append(x)
  return np.array(xchain)

def move(func, t, y, x, h = 0.1, TINY=1e-30) :
    pass

def loglikelihood( func, t, y, x, sigma=1) :
    pass

if __name__ == "__main__" :
    t, y, x0 = generate_data()
    guess = np.zeros(2)
    guess[0] = 2
    guess[1] = 0
    xchain = mcmc(test_function, t, y, x0)
    print( "chain length = {2} MCMC mean = {0}, actual = {1}".format(str(
   np.average(xchain, axis=0)), str(x0), xchain.shape[0]))
    pl.hist2d(xchain[:,0],xchain[:,1],bins=100)
    pl.show()
    pl.clf()
    pl.hist(xchain[:,0],bins=100)
    pl.savefig("mcmc_1.pdf")
```

Note that we accumulate the values of $\boldsymbol{\theta}$ as we evolve the chain and we have burn in timescale of about 1000 moves.

Now, we can compute statistics using the distribution of parameters. Go ahead and compute the 25th, 50th, and 75th percentile of the parameters using numpy.percentile.