Contents

1 Numerical Optimisation

Suppose we have a functional F and function G defined as follows.

$$F[f] = \int_D \mathcal{L}(f, f_{x_1}, f_{x_2}, \ldots, f_{x_n}, f_{x_1 x_1}, \ldots) d^n x \tag{1}$$

$$G[g] = G(g_i | i \in \mathcal{I}) \tag{2}$$

where the integration is over any domain D, multidimensional or not, infinite or not. $\mathcal{L}$ is a function of $f(x)$ and its derivatives, but not directly of x. Typically there will not be any mixed derivatives or derivatives of order higher than 2 in $\mathcal{L}$.

We need to find f (g) such that it (locally) minimises (or maximises) F (G). Typically infinities in the domain can be cut off numerically, but obviously f will then need to approach zero quickly enough to ensure this does not cause too big errors.

If D is not a nice rectangular subset of $\mathbb{R}^N$, the approach introduced below is not very good: one would need to embed f into a rectangular domain and require $f(x \notin D) = 0$, but this can lead to a very complicated code or a code which spends most of its time adding together billions of zeros. A better approach in these cases is to use a more sophisticated discretisation method, like the finite element method (FEM). A very good FEM library with a natural python interface is FEniCS, Its python binding should be installable by `pip install fenics` but be warned: it requires a large number of dependencies some of which `pip` may not be able to install for you if you do not have them.

1.1 Finite Differences

The simplest way of dealing with F is to write a finite difference approximation, where one replaces differentials and integrals with their definitions but without the lim, i.e.

$$\frac{df(x)}{dx} \to \frac{f(x+h) - f(x-h)}{2h} \tag{3}$$

$$\int_D f(x)dx \to \sum_{x\in D'} f(x)h. \tag{4}$$

As an example, suppose

$$\mathcal{L} = \frac{1}{2}f'(x)^2 + m^2(1 - \cos(f)), \tag{5}$$

so we can compute the Euler-Lagrange equation of $F[f]$:

$$0 = \frac{d}{dx}\frac{d\mathcal{L}}{df'} - \frac{d\mathcal{L}}{df} = f''(x) - m^2\sin(f), \tag{6}$$

which is the (familiar?) sine-Gordon equation. Assuming boundary conditions $f(-\infty) = 0$ and $f(\infty) = 2\pi$, it has a solution

$$f(x) = 4\arctan(\exp^{m(x-a)}), \tag{7}$$

where a is a free integration constant (location of the peak of our soliton); the other integration constant is fixed by the boundary conditions.

We want to find this numerically, using finite difference approximation. We will need to discretise the objective function $F[f]$. The discrete version of $F[f]$ will have the form of $G[g]$ and any sensible algorithm will take advantage of either the Euler-Lagrange equation or the gradient (with respect to g_i) of $G[g]$. On the continuum ($h \to 0$) limit these will be equivalent, but numerically there are subtle differences: using the Euler-Lagrange equation is slightly simpler to implement, but on the other hand it only works in a case where you have $\mathcal{L}$. Using the gradient of $G[g]$ is slightly more generic but even that will of course fail if $G[g]$ is not differentiable. In this case, some sort of regularisation or gradient-free algorithm $---$ such as Powell's algorith which see later $---$ is necessary.

Algorithms that do not use gradient information are generally very slow. However, they are often also able to find a global minimum unlike gradient based methods which simply find the "closest" local minimum which either is or is not a global minimum. Two of the most commonly used such algorithms are *simulated annealing* (very closely related to Metropolis-Hastings) and *genetic algorithm*.

1.2 Discretisation

Using the above discretisation for derivatives, we can write, using lattice spacing h:

$$G[f] = \sum_{x \in D} \left(\frac{1}{2} \Big(\frac{f(x+h) - f(x-h)}{2h} \Big)^2 + m^2 (1 - \cos(f(x))) \right) h \tag{8}$$

and the task now is to find f such that G is minimised. At lattice point x_i we write $f(x+h) = f(x_{i+1})$ etc and the gradient of G wrt $f(x_i)$ becomes

$$\frac{dG}{df(x_i)} = h \left(\frac{f(x_i) - f(x_{i-2})}{4h^2} + m^2 \sin(f(x_i)) - \frac{f(x_{i+2}) - f(x_i)}{4h^2} \right) \tag{9}$$

$$= \frac{1}{4h} \Big(-f(x_{i+2}) + 2f(x_i) - f(x_{i-2}) + 4h^2 m^2 \sin(f(x_i)) \Big) \tag{10}$$

A couple of observations are in order here: even though our finite difference "stencil" in (3) was just 1-point-wide (in both directions), the gradient includes f evaluated at 2 points removed. This is a generic feature of the approach we have taken and is one of the main reasons the Euler-Lagrange approach is favoured by many whenever there is an Euler-Lagrange equation. The other point has to do with the central difference scheme: no central difference formula of any oddth derivative of $f(x)$ of any order evaluated at x ever includes $f(x)$. This can lead to "chessboard" solutions where the lattice decouples to two independent sublattices: one for even (odd) lattice points where the odd derivatives only depend on values of f at odd (even) points. If there are no terms binding even and odd lattice points together, central differences cannot be used. For our problem, this binding is provided by the trigonometric term.

1.3 Python Solution

Our solution will contain four parts:

- an initial guess generator
- a function to compute the objective function
- a function to compute the gradient, and
- the solver.

Of course at the end we visualise the solution obtained, overlaying it with the exact solution.

Furthermore, we use the Euler-Lagrange equation approach: please see the docstrings for the reason.

```
import numpy
import scipy.optimize
import matplotlib.pyplot

class sineGordon:
    '''A class to represent the discretised energy of the sine-Gordon field with
    given lattice spacing, peak location, and mass and its gradient. We also
    provide an initial guess generator
    '''
    def __init__(self, lattice_spacing=0.1, mass=1.0, peak_location=0.0):
        self.h = lattice_spacing=0.1
        self.m = mass
        self.a = peak_location
    def initial_guess(self, points, amplitude=0.0):
        '''An initial guess with amplitude size random fluctuations around the exact
        solution. The points parameter determines the number of points to
        use. They will be separated by self.h and symmertically around origin; a
        boundary condition is imposed at both ends.
        >>> sG = sineGordon(lattice_spacing=0.1, mass=2.0, peak_location=0.1)
        >>> sG.initial_guess(5)
        array([ 0.        ,  2.36211041,  2.74423296,  3.14159265,  6.28318531])
        '''
        xmax = (points//2-((points+1)%2)/2)*self.h
        xmin = -xmax
        x = numpy.mgrid[xmin:xmax:1j*points]
        state = (4.0 * numpy.arctan(numpy.exp(self.m*(x-self.a))) +
                 amplitude*(numpy.random.random(x.shape)-0.5))
        state[0] = 0.0
        state[-1] = 2*numpy.pi
        self.x = x
        return state
    def energy_density(self, state):
        gradf = numpy.gradient(state,self.h)
        energy_density = self.h*(gradf**2/2 + self.m**2*(1 - numpy.cos(state)))
        return energy_density
    def energy(self, state):
        '''The energy of the sine-Gordon field: h(f'(x)^2/2 + m^2(1 - \cos(f))).
        >>> sG = sineGordon(lattice_spacing=0.1, mass=2.0, peak_location=0.1)
        >>> print("{x:.8f}".format(x=sG.energy(sG.initial_guess(5))))
        105.32743396
        '''
        return self.energy_density(state).sum()
    def gradient(self, state):
        '''The gradient wrt f of the sine-Gordon field energy. We use the Euler-Lagrang
        equations instead of the discrete gradient because it is difficult to
        deal with points near the boundary. The discrete gradient is
        \dfrac{1}{4h}\(-f(x_{i+2}) + 2 f(x_i) + m^2 \sin(f(x_i)) - f(x_{i-2})\)
        so if i is next to the boundary, either i+2 or i-2 will be outside our
        array!
        >>> sG.gradient(sineGordon(lattice_spacing=0.1, mass=2.0, peak_location=0.1).i
        array([   0.        ,   99.32137587,  -19.4809996 , -137.59257666,    0.
        '''
        gradgrad = numpy.gradient(numpy.gradient(state,self.h),self.h)
        gradient = self.m**2*numpy.sin(state) - gradgrad
        gradient[0] = 0.0
        gradient[-1] = 0.0
        return gradient
    def plot(self, state):
        matplotlib.pyplot.gcf()
        matplotlib.pyplot.clf()
        matplotlib.pyplot.plot(self.x, state)
        return

sG = sineGordon(lattice_spacing=0.1, mass=1.0, peak_location=0.0)
state = sG.initial_guess(300, amplitude=0.001)
sol = scipy.optimize.fmin_powell(sG.energy, state)
sol = scipy.optimize.fmin_cg(sG.energy, state, fprime=sG.gradient)
sol = scipy.optimize.fmin_bfgs(sG.energy, state, fprime=sG.gradient)
sG.plot(sol)
```

2 Exercises

2.1 Steepest Descents

Implement the steepest descents algorithm (with constant step-size dt):

1. Compute the gradient of the objective function $G[f_t]$
2. Compute a new solution candidate $f_{t+1} = f_t - \nabla G[f_t]dt$
3. Compute the objective function value for the new candidate $G[f_{t+1}]$
4. If $G[f_t] > G[f_{t+1}]$, accept the new candidate f and go back to 1

When the algorithm stops, f will be no further than h from a local minimum. There are obvious improvements that could be done to make the approximation better, like adjusting the step size until it becomes zero, but there are issues with floating point precision etc with most of these.

Your steepest descents optimiser should take five parameters: initial guess, a function to compute G, another to compute ∇G, the desired step size h and the desired lattice spacing(s) h. The two functions will of course also need to take a candidate solution and the lattice spacing(s) as parameters.

Of course, there will need to be something generating the initial guess, but that depends on the problem to solve, so we leave that to the next exercise.

2.2 Find the Minimum of the Rosenbrock Function

We are already familiar with the Rosenbrock function. Now the task is to find its minimum in the N-dimensional case using the steepest descents algorithm developed in the previous exercise.

A way of doing this using the conjugate gradient or variable metric algorithms would be as follows:

```
import numpy
import scipy.optimize
import matplotlib.pyplot

class Rosenbrock:
    '''A class to represent the discretised energy of the sine-Gordon field with
    given lattice spacing, peak location, and mass and its gradient. We also
    provide an initial guess generator
    '''
    def __init__(self, lattice_spacing=0.1, a=1.0, b=100.0):
        self.h = lattice_spacing=0.1
        self.a = a
        self.b = b
    def initial_guess(self, points):
        '''An initial guess for x and y.
        >>> Rosenbrock(lattice_spacing=0.1, a=2.0, b=20.0).initial_guess((3.0,5.0))
        array([ 3.,  5.])
        '''
        state = numpy.array(points)
        return state
    def energy(self, state):
        '''The value of the Rosenbrock function
        f(x,y) = (a-x)^2 + b(y-x^2)^2

        >>> _Rb = Rosenbrock(a=2.0, b=100.0)
        >>> _Rb.energy((2.0,3.0))
        100.0
        '''
        x,y = state
        a,b = self.a, self.b
        energy = (a-x)**2 + b*(y-x**2)**2
        return energy
    def gradient(self, state):
        '''The gradient wrt x,y of the Rosenbrock function.
        >>> _Rb = Rosenbrock(lattice_spacing=0.1, a=2.0, b=20.0)
        >>> _Rb.gradient(_Rb.initial_guess((3.0,5.0)))
        array([ 962., -160.])
        '''
        x,y = state
        a,b = self.a, self.b
        gradx, grady = 4*b*x**3-4*b*y*x+2*x-2*a, -2*b*x**2+2*b*y
        return numpy.array([gradx,grady])
    def plot(self, state):
        matplotlib.pyplot.figure()
        matplotlib.pyplot.clf()
        x,y = state
        a,b = self.a, self.b
        minimum = (a,a**2)
        YX = numpy.mgrid[minimum[0]-3.0:minimum[0]+1.0:100j,minimum[1]-3.0:minimum[1]+
        matplotlib.pyplot.contourf(YX[1], YX[0], self.energy(YX), 50)
        matplotlib.pyplot.scatter(y, x, marker="o")
        return

'''
```

```
One should be able to solve as such:
>>> _Rb = Rosenbrock(lattice_spacing=0.1, a=2.0, b=5.0)
>>> sol=scipy.optimize.fmin_bfgs(_Rb.energy, _Rb.initial_guess((3.0,5.0)), fprime=_Rb.g
>>> "{x:.6f}, {y:.6f}".format(x=sol[0], y=sol[1])
'2.000000, 4.000000'
'''
Rb = Rosenbrock(a=1.0, b=100.0)
state = Rb.initial_guess((2.0,3.0))
sol = scipy.optimize.fmin_powell(Rb.energy, state)
sol = scipy.optimize.fmin_cg(Rb.energy, state, fprime=Rb.gradient)
sol = scipy.optimize.fmin_bfgs(Rb.energy, state, fprime=Rb.gradient)
Rb.plot(sol)
```