Problem Class 2: Finite difference methods for boundary value problems

David Dickinson d.dickinson@york.ac.uk

Semester 1

1 Recap of solving BVP problems with finite differences

In lectures 5 and 6 we looked at both the general idea behind solving boundary value problems (like ODEs where we want to enforce certain conditions at multiple locations) and some ways to implement this in Python using scipy.

There are many problems that we want to solve that are in this form. A simple example is Poisson's equation in 1D

$$-\frac{d^2\phi}{dx^2} = \rho(x) \tag{1}$$

It's possible to convert this to two first order ODEs similar to the problems we looked at solving with solve_ivp in the last problem class, so why do we need any extra tools? The key difference is that whilst integrating the equation with solve_ivp allows us to specify the (initial) conditions at one point, we cannot specify anything about the solution $\phi(x)$ at any other location. For example we may wish to solve eq. (1) in a box of length L (from $-L/2$ to $L/2$) with the condition that ϕ must be zero at either end of the box.

The first step in solving this type of problem is to introduce the finite difference approximation of the derivatives. For example we have the (second order) central difference formula:

$$\left.\frac{d^2\phi}{dx^2}\right|_{x=x_i} \approx \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\delta x^2} \tag{2}$$

where we have evaluated our function ϕ on a grid with spacing δx. This then allows us to write the system in terms of a matrix operation, $\underline{\boldsymbol{M}} \cdot \phi = \rho$, where ϕ is the function we want to solve for, ρ is the known right hand side and $\underline{\boldsymbol{M}}$ is the matrix operator. Typically (although not always) the matrix operator is tri-diagonal (i.e. only has non-zero entries on the diagonal and one element directly above and below the diagonal). In the simple example of eq. (1), using eq. (2) for the derivative, we find that the matrix is indeed tridiagonal and has elements $A_i = -1/\delta x^2$, $B_i = 2/\delta x^2$ and $C_i = -1/\delta x^2$ where B_i is the diagonal term and A_i and C_i are the below and above diagonal terms respectively.

Before we can use this representation of the system to solve for ϕ we must look at how we can enforce the boundary conditions in this picture. Two common types of boundary condition are Dirichlet (fixed value) and Neumann (fixed gradient). To enforce either of these boundary conditions we must manipulate the first and last rows in the matrix operator. For example the first row is

$$B_1\phi_1 + C_1\phi_2 = \rho_1$$

To enforce Dirichlet boundaries to fix $\phi = a$ on the left hand boundary we can set $B_1 = 1$, $C_1 = 0$ and $\rho_1 = a$. This makes the first row of the matrix simply read $\phi_1 = a$. Zero-Neumann conditions ($d\phi/dx = 0$) could be enforced using $B_1 = -1/\delta x$, $C_1 = 1/\delta x$ and $\rho_1 = 0$, such that the first row gives $(\phi_2 - \phi_1)/\delta x = 0$. We can recognise the left hand side as being the forward difference approximation of $d\phi/dx$.

The following provides an example of how we can solve eq. (1) under the boundary conditions $\phi(x = -L/2) = 0$ and $\phi(x = L/2) = 0.05$. Note that we can find the error on the solution by calculating $R = \underline{\boldsymbol{M}} \cdot \phi$ and comparing with the known right hand side, ρ. The solution produced is shown in fig. 1.

```
#Imports
from scipy.linalg import solve
from numpy import zeros,linspace,exp,abs
import matplotlib.pyplot as plt
```

```
#Parameters
length = 1.0 ; nx = 100
xval=linspace(-length/2,length/2,num=nx)
dx = xval[1]-xval[0]

#Create matrix operator and rhs vector
M = zeros((nx,nx))

#Set elements, -d^2/dx^2  -- Skip boundaries
rho = exp(-((xval)**2)/1.0e-2) #Gaussian rho
for i in range(1,nx-1):
    M[i,i-1] = -1.0/dx**2
    M[i,i]   =  2.0/dx**2
    M[i,i+1] = -1.0/dx**2

#Boundaries
#/Lower -- Dirichlet:0
M[0,0] = 1.0 ; rho[0] = 0.0
#/Upper -- Dirichlet:0.05
M[-1,-1] = 1.0 ; rho[-1] = 0.05

#Solve
sol = solve(M,rho)

#Substitute solution back in to calculate RHS
rc=M.dot(sol)
#Print error
print("Max absolute error is {num}".format(num=abs(rc[1:-1]-rho[1:-1]).max()))
#Plot solution
plt.plot(xval,sol,"-") ; plt.xlabel(r"$x$")
plt.ylabel(r"$\phi$") ; plt.show()
```

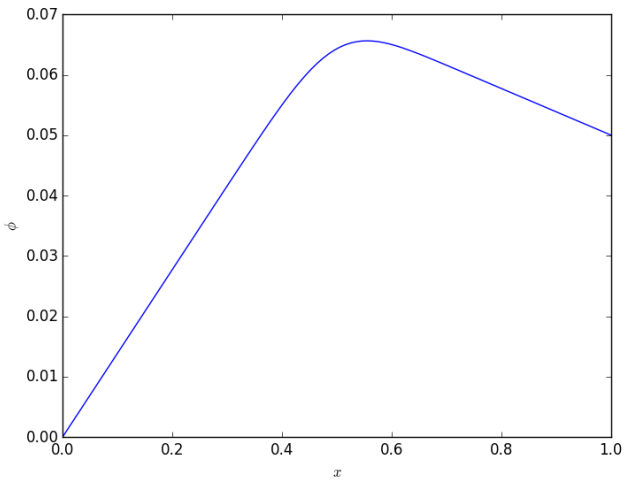

Figure 1: Output of the simple Poisson example.

Tasks:

1. Implement the Poisson example code above and modify it to use sparse matrices. Hint: The sparse matrix type lil_matrix can be accessed like a normal matrix and can be converted to different types (including csr_matrix). See help(scipy.sparse.linalg) for different operations.
2. Use time.time to time the solution step. How does the time taken for the sparse solve compare to the dense solve when $n_x = 10,000$?

An example solution is given in section 4.1

2 Exercise: Extending the problem

In this exercise we will look at extending the code from the previous section in order to change the equation being solved, implement alternative boundary conditions and look at the error on the solution.

Tasks:

1. Modify the code from the previous task to solve:

$$-\frac{d^2\phi}{dx^2} + \frac{d\phi}{dx} = 1$$

   subject to the same boundary conditions as in the example. Hint: The central difference formula for $d\phi/dx$ is

$$\frac{d\phi}{dx} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\delta x}$$

2. Add the option for the boundary conditions at the right hand side ($x = L/2$) to be Neumann (constant gradient) with gradient given by a. Hint: The backward difference formula is $(f_i - f_{i-1})/dx$.
3. Compare the solution to the analytic solution, $\phi(x) = c_1 \exp(x) + c_2 + x$. Hint: You need to apply the boundary conditions to determine c_1 and c_2 $\implies$ the analytic solution is different when using Dirichlet or Neumann.
4. Calculate the rms error on the solution (i.e. calculate $\epsilon = \sqrt{\sum (\phi_a - \phi_n)^2}/n_x$, where ϕ_a and ϕ_n are the analytic and numerical solutions respectively).
5. Investigate how the error varies with the number of grid points (consider the range $10 \leq n_x \leq 10^4$) for both Dirichlet and Neumann conditions. Hint: if $\epsilon \propto n_x^C$ we can determine the order, C, by fitting a straight line through the $\log \epsilon$ vs $\log n_x$ data.
6. Why is there a difference in the order of convergence between Dirichlet and Neumann? Modify the code to improve the order of the Neumann case. Hint: The second order backwards difference formula is $(1.5f_i - 2f_{i-1} + 0.5f_{i-2})/dx$.
7. How does the time taken to solve with $n_x = 10^4$ compare between first and second order Neumann cases?

An example solution is given in section 4.2

3 Exercise: The heat equation in 2D

Let us now consider the heat equation:

$$\frac{\partial u}{\partial t} = k\nabla^2 u - q \tag{3}$$

where u represents temperature, k is the thermal conductivity and q comes from a heat source. For the two-dimensional Cartesian problem we can write

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

If we seek stationary solutions (i.e. where $\partial u/\partial t = 0$) there are two approaches we can take:

1. Set the LHS to 0 in eq. (3) and solve as a BVP.
2. Use our finite difference approximations for ∇^2 to rewrite eq. (3) in a form we can integrate using solve_ivp. We can then integrate in time until there is no change in u between steps.

To begin with let us consider the first approach. Unlike the previous examples the differential operator is now two dimensional, we can however still write the operator in the usual matrix form $\underline{\boldsymbol{A}} \cdot u = q/k$ and so we should still be able to use the same approach to solving this. The trick is to "flatten" the two-dimensional space into a one-dimensional one. If we define a mapping $(x, y) \rightarrow (z)$, considering a problem with n_x grid points in x and n_y grid points in y we can map this 2D problem to a 1D one on a grid of size $n_z = n_x \times n_y$. In practice this is made much easier using a simple function which takes the (x, y) grid co-ordinates and returns the co-ordinate in z[1]. An example of such a function is:

[1] In other words a function which defines the mapping $(x, y) \rightarrow (z)$.

```
#Function to map from (x,y) to (z)
def xyToz(ix,iy,nx,ny):
    #Note: We pass nx even though we"re not using
    #it as we may want to change this mapping later
    #All the iy values at a given ix
    #are in sequence together.
    return ix*ny + iy
```

The final step in being able to find a stationary solution of eq. (3) is to write down our finite difference approximation of ∇^2:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \frac{f_{ix+1,iy} - 2f_{ix,iy} + f_{ix-1,iy}}{\delta x^2} + \frac{f_{ix,iy+1} - 2f_{ix,iy} + f_{ix,iy-1}}{\delta y^2} \tag{4}$$

Note how this operator involves five different locations.

Tasks:

1. Write a sparse matrix solver for stationary solutions of eq. (3) in two dimensions using Dirichlet boundary conditions, assume $n_x = n_y = 100$, $k = 1.0$ and $q = 0$ except on the x boundaries where $q = 1.0$.
2. Make a contour plot of the stationary solution. Hint: The solution will initially be in 1D you will need to convert this back to 2D, see the reshape function. To find out about contouring try help(plt.contourf).
3. *Optional*: Explore the idea of solving eq. (3) with solve_ivp. Hint: We can still use our finite difference formula. What happens at the boundaries? For a further challenge try to write a program that solves the heat equation using solve_ivp and compare the result to that from the BVP method.

An example solution is given in section 4.3

4 Example solutions

4.1 Poisson: dense vs sparse approaches

```
#Imports
from scipy.sparse import lil_matrix
from numpy import zeros,linspace,exp,abs,dot
import matplotlib.pyplot as plt
from time import time

#Parameters
length = 1.0 ; nx = 10000
xval=linspace(-length/2,length/2,num=nx)
dx = xval[1]-xval[0]

#Do we use a dense approach?
dense = True #Make this False to use sparse approach

#Create matrix operator
if not dense:
    M = lil_matrix((nx,nx))
else:
    M = zeros((nx,nx))

#Create right hand side vector
rho = zeros(nx)

#Set elements, -d^2/dx^2  -- Skip boundaries
rho = exp(-((xval)**2)/1.0e-2) #Gaussian rho
for i in range(1,nx-1):
    M[i,i-1] = -1.0/dx**2
    M[i,i]   =  2.0/dx**2
    M[i,i+1] = -1.0/dx**2

#Boundaries
#/Lower -- Dirichlet:0
M[0,0] = 1.0 ; rho[0] = 0.0
#/Upper -- Dirichlet:0.05
M[-1,-1] = 1.0 ; rho[-1] = 0.05

#Final setup and get solve routine
if not dense:
    M = M.tocsr()
    from scipy.sparse.linalg import spsolve as solve
else:
    from scipy.linalg import solve

#Solve (and time)
t1 = time() ; sol = solve(M,rho) ; t2 = time()
print("Time for solve: {tt}s".format(tt=t2-t1))

#Substitute solution back in to calculate RHS and print error
if not dense:
    rc=M*sol
else:
    rc=dot(M,sol)
print("Max absolute error is {num}".format(num=abs(rc[1:-1]-rho[1:-1]).max()))
#Plot solution
plt.plot(xval,sol,"-") ; plt.xlabel(r"$x$")
plt.ylabel(r"$\phi$") ; plt.show()
```

4.2 Extended problem: Neumman boundary conditions and convergence order

This study is aided if we define a function which takes n_x (and other options) and return the solution and error. An example of this is

```
from scipy.sparse import lil_matrix
```

```
from scipy.sparse.linalg import spsolve as solve
from numpy import zeros,linspace,exp,polyfit,log,sqrt,mean,array
import matplotlib.pyplot as plt
from time import time
#Main function for solving -d^2/dx^2 + d/dx = 1.0
def solveEq(nx=10,aVal=0.0,dense=False,
    secondOrder=False,dirichlet=False):
    """Solve Poisson"s equation on grid controlled by input.
    Returns x,phi,err,time"""
    #Define a function that gives the analytic solution
    #phi = c1*exp(x) + c2 + x
    if dirichlet:
        def anSol(xval,aVal):
            c2 = aVal/(1-exp(1.0))-0.5*(1+exp(1.0))/(1-exp(1.0))
            c1 = exp(0.5)*(0.5-c2)
            return c1*exp(xval)+c2+xval
    else:
        def anSol(xval,aVal):
            c2 = 0.5-(aVal-1)*exp(-1.0)
            c1 = ((aVal-1)*exp(-0.5))
            return c1*exp(xval)+c2+xval
    length = 1.0
    xval=linspace(0.0,length,num=nx)-length*0.5 ; dx = length/(nx-1.0)
    #Create matrix operator
    M = lil_matrix((nx,nx))
    #Set elements, -d^2/dx^2 +d/dx -- Skip boundaries
    rho = zeros(nx) + 1.0
    for i in range(1,nx-1):
        M[i,i-1] = -1.0/dx**2 - 0.5/dx
        M[i,i]   =  2.0/dx**2
        M[i,i+1] = -1.0/dx**2 + 0.5/dx
    #Boundaries
    M[0,0] = 1.0 ; rho[0] = 0.0 #Always do Zero-dirichlet on left
    if dirichlet: #Dirichlet BC
        M[-1,-1] = 1.0
    else: #Neumann
        if secondOrder: #Second order
            M[-1, -1] =  1.5/(dx)
            M[-1, -2] = -2.0/(dx)
            M[-1, -3] =  0.5/(dx)
        else: #First order
            M[-1,-1] = 1.0/dx ; M[-1,-2]=-1.0/dx
    rho[-1] = aVal
    #Final setup
    M = M.tocsr()
    #Solve
    t1 = time() ; sol = solve(M,rho) ; t2 = time()
    #Calculate rms absolute error
    err=(sol-anSol(xval,aVal)); err=sqrt(mean(err*err))
    return xval,sol,err,t2-t1
```

This can then be used in a loop over n_x to study how quantities vary. You should find that in the Dirichlet case the error decreases as n_x^2 whilst in the simple Neumann case the error decreases as n_x. This might look like the following

```
nxval=array([10,20,30,35,40,45,50,100,500,1000,5000,10000])
err=[] ; tm=[]
for n in nxval:
    x,p,e,t = solveEq(nx=n,aVal=0.05,
        secondOrder=True,dirichlet=False)
    err.append(e) ; tm.append(t)
#Plot error vs nx
plt.plot(log(nxval),log(err),"-x") ; plt.xlabel(r"log$(n_x)$")
plt.ylabel(r"log$(\epsilon)$") ; plt.show()
#Fit log(error)
fit=polyfit(log(nxval),log(err),1)
order=fit[0]
```

```
print("The error scales with order : {ord}".format(ord=order))
```

4.3 The heat equation in 2D

```
from scipy.sparse import lil_matrix
from scipy.sparse.linalg import spsolve as solve
from numpy import zeros,linspace,exp,polyfit,log,sqrt,mean,array
import matplotlib.pyplot as plt
from time import time

#Define problem size
nx = ny = 100
length = 1.0 #Assume square box for simplicity
xval = yval = linspace(0,length,nx)
dx = dy = xval[1]
#Setup "flattened" problem
nz = nx * ny
M = lil_matrix((nz,nz))
def xyToZ(ix,iy,nx,ny):
    return ix*ny+iy

#Now create rhs
q = zeros(nz)

#Populate centre of matrix
t0=time()
for ix in range(1,nx-1):
    for iy in range(1,ny-1):
        #Find indices, this point and ix+/-1, iy+/-1
        indx = xyToZ(ix,iy,nx,ny)
        indx_xp1 = xyToZ(ix+1,iy,nx,ny)
        indx_xm1 = xyToZ(ix-1,iy,nx,ny)
        indx_yp1 = xyToZ(ix,iy+1,nx,ny)
        indx_ym1 = xyToZ(ix,iy-1,nx,ny)

        #Set values
        M[indx,indx] = -4.0 / dx
        M[indx,indx_xp1] = 1.0/dx
        M[indx,indx_xm1] = 1.0/dx
        M[indx,indx_yp1] = 1.0/dx
        M[indx,indx_ym1] = 1.0/dx

#Set boundary values
for ix in [0,nx-1]: #Loop over x-boundaries
    for iy in range(ny): #All y values
        indx = xyToZ(ix,iy,nx,ny)
        q[indx] = 1.0
        M[indx,indx] = 1.0
for iy in [0,ny-1]: #Loop over y-boundaries
    for ix in range(nx): #All x values
        indx = xyToZ(ix,iy,nx,ny)
        M[indx,indx] = 1.0
t1=time()
#Convert and solve
M=M.tocsr() ; t2=time() ; sol = solve(M,q); t3=time()
#Make solution 2D
solView = sol.reshape((nx,ny))
#Plot
plt.contourf(xval,yval,solView,64)
plt.xlabel(r"$x$") ; plt.ylabel(r"$y$")
plt.colorbar(label="Temperature") ; plt.show()
#Print timing data
print("Time to populate matrix : {tim}s".format(tim=t1-t0))
print("Time to solve           : {tim}s".format(tim=t3-t2))
```