Lecture 6

Spatial discretisation : finite differences practical

D. Dickinson
d.dickinson@york.ac.uk

York Plasma Institute, School of PET, University of York

Semester 1

Overview of course

This course provides a brief overview of concepts relating to numerical methods for solving differential equations.
Topics to be covered include:

- ~~Integrating ODEs~~
  - ~~Explicit techniques~~
  - ~~Implicit techniques~~
  - ~~Using Scipy to integrate ODEs~~
- Spatial discretisation
  - Finite differencing
  - Spectral methods
  - Finite elements
- Particle In Cell (PIC) approaches
- Continuum techniques

Notes available on the VLE.

Overview this lecture

This lecture will look at

- Sparse matrix structures
- Using SciPy to solve (sparse) linear algebra problems.

Sparse matrices

We saw last time that when using finite difference approximations we can end up with a matrix equation of the form (or close to)

Couples neighbouring points - local interaction

$$\begin{pmatrix} B_1 & C_1 & & & & \\ A_2 & B_2 & C_2 & & & \\ & A_3 & B_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_N & B_N \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{pmatrix} = \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \vdots \\ \rho_N \end{pmatrix}$$

where all the elements of the matrix not shown are zero.

It is quite wasteful to store a large number of zero elements as these aren't going to contribute anything to the calculation. Matrices with mostly zero elements are known as sparse[1] and special formats/structures have been developed for working with sparse matrices.

But, can calculate sparsity: zeros, and non-zero elements

[1] Whilst matrices with mostly non-zero elements are known as dense.

Sparse matrices

By only storing the non-zero elements of a sparse matrix we can drastically reduce the amount of memory required to fit the matrix in RAM. For example consider a tridiagonal $N \times N$ matrix. With $N = 10,000$ we'd need 0.8 GB to hold the full matrix in memory[2] but only 25 MB to hold the three non-zero diagonals. Furthermore if we double N the non-zero diagonals double in size but the full matrix increases by a factor 4! Memory needed increases too quickly

As well as this improvement in memory requirements "sparse aware" linear algebra packages will be able to invoke specialised routines when working with sparse data which will likely be faster than solving a general system.

The ratio of the number of zero elements to the total number of elements is known as the sparsity of the matrix[3]. Whilst it's possible to use sparse methods with a matrix of arbitrary sparsity (i.e. we could use them for a dense matrix) overheads involved are likely to make things worse if the sparsity is not high enough.

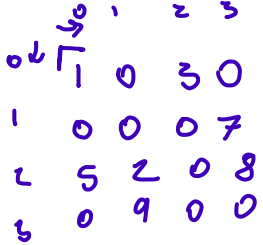
[2] Assuming 8 bytes per element, which is typical of values in "double precision".

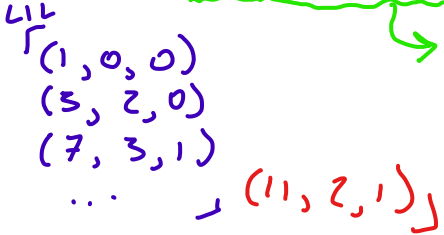
[3] Whilst the ratio of non-zero elements to the total number is known as the density.

Sparse matrices

There are several different sparse matrix storage formats which each have their own advantages and disadvantages. Luckily SciPy provides a sparse module which provides ways to use a number of these different sparse matrix formats. Some of the formats we'll be using are

LIL This is a linked list format. For each row of the matrix we store a list of value pairs, each value pair consists of the column index and the data value at that point. With this format it is quite easy to add extra values, but it's not so good for accessing specific elements.





Have to search through whole list (not great for operations)

- but matrix should be sparse, so list shouldn't be too long

Only records non-zero values.

Can reconstruct matrix easily, or add new value.

Sparse matrices

There are several different sparse matrix storage formats which each have their own advantages and disadvantages. Luckily SciPy provides a sparse module which provides ways to use a number of these different sparse matrix formats. Some of the formats we'll be using are

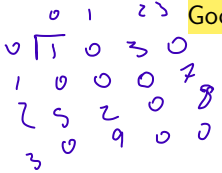
LIL This is a linked list format.

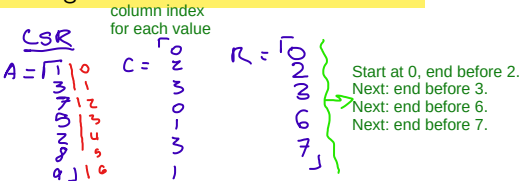
CRS This is the compressed row storage format. More complex than the LIL format; it uses three 1D arrays to represent the $N \times N$ matrix. If N_{nz} is the number of non-zero elements then there will be

- One array of length N_{nz}, A, which just contains each of the non-zero elements.
- An array of length $N + 1$, where the ith element holds the index of A which has the first value of the ith row in the matrix.
- Another array of length N_{nz} which holds the column index of each non-zero element.

Good for accessing elements but not to construct.

get non-zero values in particular row





Sparse matrices

There are several different sparse matrix storage formats which each have their own advantages and disadvantages. Luckily SciPy provides a sparse module which provides ways to use a number of these different sparse matrix formats. Some of the formats we'll be using are

LIL This is a linked list format.

CSR This is the compressed row storage format.

CSC This is the compressed column storage format. Basically the same as CSR but with rows and columns switched. Useful when you might want to look at individual columns.

Sparse matrices

There are several different sparse matrix storage formats which each have their own advantages and disadvantages. Luckily SciPy provides a sparse module which provides ways to use a number of these different sparse matrix formats. Some of the formats we'll be using are

- **LIL** This is a linked list format.
- **CSR** This is the compressed row storage format.
- **CSC** This is the compressed column storage format.

Helpfully SciPy allows us to construct a sparse matrix using LIL and then convert it to one of the other formats which are better for doing calculations!

Now lets have a look at this in practice!

Finite difference example

Let's consider the equation

$$\frac{d^2 f}{dx^2} - \frac{df}{dx} + 3 = 0$$

Using the finite difference approximations introduced last time we get

$$\frac{f_{i+1} - 2f_i + f_{i-1}}{\delta x^2} - \frac{f_{i+1} - f_{i-1}}{2\delta x} + 3 = 0$$

which we can rearrange to

$$\underbrace{\left[\frac{1}{\delta x^2} + \frac{1}{2\delta x}\right]}_{\text{A co}} f_{i-1} + \underbrace{\left[\frac{-2}{\delta x^2}\right]}_{\text{B co}} f_i + \underbrace{\left[\frac{1}{\delta x^2} - \frac{1}{2\delta x}\right]}_{\text{C co}} f_{i+1} = \underbrace{-3}_{\text{rhs}}$$

For now we won't specify the boundary conditions

Finite difference example

Now that we have the discretised problem let's start building the matrix representation using the sparse LIL format

```
from scipy.sparse import lil_matrix
length = 1.0                                  #Size of box
N = 10                              #Number of grid points
dx = length / (N-1)                           #Size of step

A = lil_matrix((N,N))      #Create sparse matrix (lil)   10x10

#Set the middle of matrix (no BC)
for i in range(1,N-1):
  A co A[i,i-1] = 1.0/dx**2 + 1.0/(2*dx) #Acts on f_i-1
  B co A[i,i] = -2.0/dx**2                #Acts on f_i
  C co A[i,i+1] = 1.0/dx**2 - 1.0/(2*dx) #Acts on f_i+1
```

Note that here we've not treated the boundaries.

Finite difference example

Next we create the right hand side vector

```
from numpy import zeros
rhs = zeros(N)-3 #Creates a vector of length N
                 #with all elements set to -3
```

Now we should specify the boundaries. We'll pick Dirichlet (fixed value) boundaries where we fix the value to 0

```
#Set Dirichlet boundary, fixed to 0
A[0,0] = 1.0
A[N-1,N-1] = 1.0
rhs[0] = 0.0
rhs[N-1] = 0.0
```

To implement Dirichlet boundaries we set the diagonal corner elements to 1 and the first and last elements of rhs to the value we want the solution to have here (0 in this case).

Finite difference example

Now we've got our matrix, A, and right hand side, rhs, setup we're ready to solve the problem

$$\underline{\boldsymbol{A}} \cdot \boldsymbol{f} = \boldsymbol{rhs}$$

for $\boldsymbol{f}$. To do this we'll use the spsolve routine provided by SciPy.

```
#Convert from LIL to CSR format
A = A.tocsr()

#Use a sparse solve routine from scipy
from scipy.sparse.linalg import spsolve

#Solve
solution = spsolve(A, rhs)
```

So that spsolve will work - doesn't like lil matrices

Note that we convert A from LIL to CSR format before using spsolve.

Finite difference example

Now we have the solution let's check that it looks correct. We can do this by comparing $\underline{\boldsymbol{A}} \cdot \boldsymbol{f}$ with the input $\boldsymbol{rhs}$:

```
#Calculate A.solution to recalculate rhs
rhs_check = A.dot(solution)
#Print each value of rhs and rhs_check
#they should be the same if solution is correct
for i in range(N):
    print(str(i)+" : "+str(rhs[i])+" "+str(rhs_check[i]))
```

dot product between matrix and solution vector

This should print three columns of numbers, if the solution is correct the second and third columns should be (near) identical.

Finally let's plot the solution

End values are 0, not -3, because they are the boundary conditions we imposed.

```
import matplotlib.pyplot as plt
from numpy import linspace
plt.plot(np.linspace(0.0,length,N),solution)
plt.show()
```

Floating point calculations mean that there will be some error in last few digits of precision. Want to use " double precision" to et error into very end of sf for many sf

Summary

We've seen how to solve a two point boundary value problem using sparse matrix routines provided by SciPy. You should look at the other examples on the VLE for lecture 6 which look at more sophisticated boundary conditions, error analysis etc.

Next time we'll have a quick look at alternative methods for calculating derivatives (spectral and finite element techniques) and start to combine the knowledge built up over the previous lectures to look at how a PIC code works.