Advanced Computational Methods & Modelling

by Prof Simon J. Cox (sjc@soton.ac.uk)

1 What is Computational Modelling?

Computational modelling and the use of associated numerical methods is about choosing the right algorithm or technique for the job in hand. In these notes we will discuss some important methods and the general tips about possible problems which might be encountered, efficiencies of different methods, and stability of techniques are applicable to other numerical techniques.

1.1 Practical Software Design

Practical software design for computational modelling requires a balance between the time spent choosing the correct algorithm for a computation, performing the computation and analysing the results. Python or Matlab can be used for each of these tasks and often people use C or Fortran for larger or more complex cases.

Algorithm	Computation	Results
Matlab/ Python provides a high-level and simple way to design and check algorithms	Matlab/ Python can be used to check small test cases. Consider translating/ compiling to C, C++ or Fortran for larger cases.	The results from computational simulations can be analysed and post-processed with Matlab/ Python.

At the end of these notes there is a short appendix on Matlab for reference. For the Python examples in these notes, we use the Enthought Python build and IPython Console; the winpython build also provides similar functionality. On many Linux machines Python is now commonly installed already, though please refer to the local package manager for your operating system of choice to add in extra functionality. We also use Visual Studio with the free Python Tools for Visual Studio plug-in.

1.2 Python notes

In Python we assume that the following modules have been imported:

```
» import scipy
» import numpy
» from scipy import linalg
```

See the links at the end for more information on Python.

2 Linear Equations - Iterative methods

2.1 Introduction

Direct solvers such as Gaussian Elimination and LU decomposition allow for efficient solving. In this section we introduce iterative solutions methods. The choice of a direct method or an indirect method is a combination of the efficiency of the method (and in general iterative methods are more efficient), the particular structure of the matrix system, a trade-off between compute time and memory, and the computer architecture being used.

Iterative methods work by refining a guess to the solution and converging as quickly as possible from that guess to the actual solution. You may have met iterative methods previously in, for example, the general purpose solution of non-linear equations– such as bisection or Newton-Raphson techniques (along with their more advanced cousins).

Iterative methods for linear systems have become a widespread and powerful tool for solving the most complex scientific and engineering problems and can be extremely effective, especially when starting from a good guess at the final solution – and often effort is expended in making that initial guess as good as possible and which will start you off close to the final solution and yield a more rapid convergence to the answer. Their only drawback is that they may not necessarily converge to a solution for a particular matrix system.

In this section we will assume familiarity with linear equations of the form:

$$A\,x = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} = b \tag{2.1}$$

and their solution by Gaussian Elimination, LU Decomposition, along with issues which can arise such as a singular matrix, ill-conditioning, and poor scaling. We will also assume knowledge of norms matrices and vectors.

2.2 Jacobi Iteration

Consider the set of equations (derived from [1] Ex 3.26)

$$\begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix} \bullet \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ -21 \\ 15 \end{pmatrix} \tag{2.2}$$

These could be written:

$$\begin{aligned} x_1 &= \frac{7 + x_2 - x_3}{4} \\ x_2 &= \frac{21 + 4x_1 + x_3}{8} \\ x_3 &= \frac{15 + 2x_1 - x_2}{5} \end{aligned} \tag{2.3}$$

And we could derive an iteration scheme which cycles through each of the values of x_1, x_2, and x_3 in turn to refine an initial guess. If k is the k^{th} iteration, then $x_1^{(k+1)}$ is the next guess for x_1:

$$\begin{aligned} x_1^{(k+1)} &= \frac{7 + x_2^{(k)} - x_3^{(k)}}{4} \\ x_2^{(k+1)} &= \frac{21 + 4x_1^{(k)} + x_3^{(k)}}{8} \\ x_3^{(k+1)} &= \frac{15 + 2x_1^{(k)} - x_2^{(k)}}{5} \end{aligned} \tag{2.4}$$

Starting with an initial guess of (1, 2, 2) we obtain:

$$\begin{aligned} x_1^{(1)} &= \frac{7 + 2 - 2}{4} = 1.75 \\ x_2^{(1)} &= \frac{21 + 4 + 2}{8} = 3.375 \\ x_3^{(1)} &= \frac{15 + 2 - 2}{5} = 3.00 \end{aligned} \tag{2.5}$$

In general we can write the Jacobi scheme as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right], i = 1, 2, \ldots, n. \tag{2.6}$$

The following table shows subsequent iterations

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	1.0	2.0	2.0
1	1.75	3.375	3.0
2	1.84375	3.875	3.025
3	1.9625	3.925	2.9625
...	...	...	...
19	2.00000	4.00000	3.00000

Python example code from [2]:

```
from pprint import pprint
from numpy import array, zeros, diag, diagflat, dot

def jacobi(A,b,N=25,x=None):
    """Solves the equation Ax=b via the Jacobi iterative method."""
    # Create an initial guess if needed
    if x is None:
        x = zeros(len(A[0]))

    # Create a vector of the diagonal elements of A
```

```
    # and subtract them from A
    D = diag(A)
    R = A - diagflat(D)

    # Iterate for N times
    for i in range(N):
        x = (b - dot(R,x))/D
        pprint(x)
    return x

# Set up problem here
A = array([[4.0, -1.0, 1.0],[4.0, -8.0, 1.0] , [ -2.0, 1.0, 5.0]])
b = array([7.0 , -21.0, 15.0])
guess = array([1.0,2.0,2.0])

# Solve
sol = jacobi(A,b,N=25,x=guess)

print "A:"
pprint(A)

print "b:"
pprint(b)

print "x:"
pprint(sol)
```

Executing yields:

```
>>> python jacobi.py
array([ 1.75 ,  3.375,  3.   ])
array([ 1.84375,  3.875  ,  3.025  ])
array([ 1.9625,  3.925 ,  2.9625])
array([ 1.990625 ,  3.9765625,  3.       ])
array([ 1.99414062,  3.9953125 ,  3.0009375 ])
array([ 1.99859375,  3.9971875 ,  2.99859375])
array([ 1.99964844,  3.99912109,  3.        ])
array([ 1.99978027,  3.99982422,  3.00003516])
array([ 1.99994727,  3.99989453,  2.99994727])
array([ 1.99998682,  3.99996704,  3.        ])
array([ 1.99999176,  3.99999341,  3.00000132])
array([ 1.99999802,  3.99999604,  2.99999802])
array([ 1.99999951,  3.99999876,  3.        ])
array([ 1.99999969,  3.99999975,  3.00000005])
array([ 1.99999993,  3.99999985,  2.99999993])
array([ 1.99999998,  3.99999995,  3.        ])
array([ 1.99999999,  3.99999999,  3.        ])
array([ 2.        ,  3.99999999,  3.        ])
array([ 2.,  4.,  3.])
array([ 2.,  4.,  3.])
```

```
array([ 2.,  4.,  3.])
array([ 2.,  4.,  3.])
array([ 2.,  4.,  3.])
array([ 2.,  4.,  3.])
array([ 2.,  4.,  3.])
A:
array([[ 4., -1.,  1.],
       [ 4., -8.,  1.],
       [-2.,  1.,  5.]])
b:
array([  7., -21.,  15.])
x:
array([ 2.,  4.,  3.])
```

Note: It is a sufficient condition for the matrix to be "strictly diagonally dominant" for the Jacobi method to converge from any given starting vector.

A matrix is said to be strictly diagonally dominant if

$$|a_{ii}| > \sum_{\substack{j=1 \\ i \neq j}}^{N} |a_{ij}|, i = 1, 2, \ldots, n. \tag{2.7}$$

In the example above, we have:

$$\begin{aligned} \text{Row 1: } & |4| > |-1| + |1| \\ \text{Row 2: } & |-8| > |4| + |1| \\ \text{Row 3: } & |5| > |-2| + |1| \end{aligned} \tag{2.8}$$

and the method will always converge for any given starting vector.

Self study: use these values in the Python code above and observe what happens:

```
A = array([[ -2.0, 1.0, 5.0], [4.0, -8.0, 1.0] , [4.0, -1.0, 1.0] ])
b = array([15.0 , -21.0, 7.0])
guess = array([1.0,2.0,2.0])
```

Why?

2.3 Gauss-Seidel (with relaxation)

In the Jacobi scheme at each stage when we update the $x_i^{(k+1)}$ at each iteration we always use the value for $x_i^{(k)}$ from the previous iteration– yet looking at the equations (2.5), why not use the value of $x_1^{(k+1)}$, when we compute $x_2^{(k+1)}$ as this is available to us. Thus the equations would become:

$$\begin{aligned} x_1^{(k+1)} &= \frac{7 + x_2^{(k)} - x_3^{(k)}}{4} \\ x_2^{(k+1)} &= \frac{21 + 4x_1^{(k+1)} + x_3^{(k)}}{8} \\ x_3^{(k+1)} &= \frac{15 + 2x_1^{(k+1)} - x_2^{(k+1)}}{5} \end{aligned} \tag{2.9}$$

Making this change and repeating the above makes the iteration to the solution (2, 4, 3) take only 10 steps- as per the table below

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	1.0	2.0	2.0
1	1.75	3.75	2.95
2	1.95	3.96875	2.98625
3	1.995625	3.99609375	2.99903125
...	...	...	...
10	2.00000	4.00000	3.00000

We can write the Gauss-Seidel method as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right], i = 1, 2, \ldots, n. \qquad (2.10)$$

Now we are using the new values of x as soon as they are available at each iteration.

However, we could do even better and rather than just use the latest value of x we might effectively interpolate (or extrapolate) between the old value of x and the latest value of x by weighting between the two – this yields the Gauss-Seidel method with relaxation:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right] + (1-\omega) x_i^{(k)}, i = 1, 2, \ldots, n, \qquad (2.11)$$

where ω is the relaxation parameter and if $0 < \omega < 1$ then we have "under-relaxation" if $\omega > 1$ then we have "over-relaxation". It is common to also call this method "Successive over relaxation (SOR)".

Self Study – see, for example, Example 2.17 in [3], which shows the method being applied to the equations similar (but NOT identical) to those that we will be working on in equation (4.48). *Hint*: If you save the code for the Example 2.17 as "**`gaussSeidel_run.py`**" then you will need to type "**`python gaussSeidel_run.py`**" at the command line making sure that "**`gaussSeidel.py`**" is in the same directory.

Whilst it is not generally possible to compute the optimal value of ω before starting, a formula exists that could be used during run time to estimate it during the calculation and it can be tuned whilst the calculation progresses (see e.g. [3])

2.4 Other methods

Other methods for the iterative solution of linear equations include

- Generalized minimal residual (GMRES) method
- The Alternating Direction Implicit (ADI) method
- The (pre-conditioned) conjugate gradient method

In general, however, many of the large systems of linear equations that we encounter in science and engineering are derived from the solution of partial differential equations and for these a whole range of other techniques have been developed.

3 Ordinary Differential Equations – Euler, Runge Kutta, Advanced Methods

3.1 Introduction

Differential equations occur frequently in the solution of science and engineering problems to model devices, systems, and the world in which we live. You may already be familiar with a range of analytic techniques which can tackle a wide range of the most commonly occurring differential equations. In this section we show how computers can be also used to solve these equations and extend the range of equations for which we can obtain an accurate solution in reasonable time and also equations for which no closed form solution is possible. A good catalogue of differential equations is Zwillinger [4].

3.2 Example- Analytic case

Suppose we had £1000 and deposited in a bank account earning 10% interest compounded continuously per year, how much would we have after 5 years (from [1] Ex 9.3)? The differential equation governing the amount of money, *y*, is:

$$y' = 0.1y \text{ over } [0,5] \text{ with } y(0) = 1000 \tag{3.12}$$

We can derive an explicit formula for this. The equation is linear and separable with solution:

$$y(t) = Ce^{0.1t} \tag{3.13}$$

where C is an arbitrary constant, but we know that at $t = 0$, $y(0) = 1000$:

$$1000 = y(0) = Ce^{0.1\times 0} = C \tag{3.14}$$

So we have the formula:

$$y(t) = y(0) \times e^{0.1t} \tag{3.15}$$

Thus at $t = 5$ for an initial investment of £1000, we would have

$$y(5) = 1000 \times e^{0.1\times 5} = £1648.72 \,(2 \text{ d.p.}) \tag{3.16}$$

3.3 Euler's method

Suppose we want to find an approximate solution to this "initial value problem". Let [a , b] be an interval over which we want to find the solution to a well posed initial value problem $y' = f(y,t)$ with $y(a)$ given. How might we approximate the solution? Let us construct a set of points $\{(t_k , y_k)\}$ that approximate the solution so $y(t_k) \approx y(t)$. We could chose mesh points and divide the interval up into M equal sub intervals and select mesh points

$$t_k = a + kh \text{ for } k = 0, 1, \ldots, M \text{ where } h = \frac{b-a}{M} \tag{3.17}$$

(h is the step size). We can now begin to approximately solve

$$y' = f(y,t) \text{ over } [t_0, t_M] \text{ with } y(t_0) = y_0. \tag{3.18}$$

Assuming that $y(t)$, $y'(t)$, and $y''(t)$ are continuous, we can use Taylor's theorem to expand $y(t)$ about $t = t_0$. This permits us to write that for every value t there will be a value c_1 between t_0 and t such that:

$$y(t) = y(t_0) + y'(t_0)(t - t_0) + \tfrac{1}{2} y''(c_1)(t - t_0)^2 \qquad (3.19)$$

When $y'(t_0) = f(t_0, y(t_0))$ and $h = (t_1 - t_0)$ are substituted in, we have an expression for $y(t_1)$:

$$y(t_1) = y(t_0) + h\ f(t_0, y(t_0)) + \tfrac{1}{2} h^2\ y''(c_1) \qquad (3.20)$$

If we assume that the step size h is small enough then we can ignore the term in h^2 so we obtain for our set of discretely chosen points:

$$y_1 = y_0 + h\ f(t_0, y_0) \qquad (3.21)$$

This is known as Euler's approximation to approximate the solution curve $y = y(t)$. We repeat this step by step and generate a sequence of points that can approximate the curve and general we have

$$t_{k+1} = t_k + h,\ y_{k+1} = y_k + h\ f(t_k, y_k) \text{ for } k = 0, 1, \ldots, M\text{-}1. \qquad (3.22)$$

A simple Euler solver based on [5] is:

```
# call this file euler2.py
# Based on
# http://code.activestate.com/recipes/577647-ode-solver-using-euler-method/
# FB - 201104096
import math
import numpy as N
import pylab
# First Order ODE (y' = f(t, y)) Solver using Euler method
# ta: initial value of independent variable
# tb: final value of independent variable
# ya: initial value of dependent variable
# n : number of steps (higher the better)
# Returns value of y at xb.
def euler(f, ta, tb, ya, n):
      h = (tb - ta) / float(n)
      t = ta
      y = ya
      for i in range(n):
          y += h * f(t, y)
          t += h
      return y

if __name__ == "__main__":

    # Print out a few sample iterations
    print ("5      steps:", euler(lambda t, y: 0.1*y, 0, 5, 1000, 5))
    print ("60     steps:", euler(lambda t, y: 0.1*y, 0, 5, 1000, 60))
    print ("100    steps:", euler(lambda t, y: 0.1*y, 0, 5, 1000, 100))
    print ("1800   steps:", euler(lambda t, y: 0.1*y, 0, 5, 1000, 1800))
    print ("100000 steps:", euler(lambda t, y: 0.1*y, 0, 5, 1000, 100000))

    #Print out one in detail and compare with analytic answer
    print
    tmp = 1000
    steps=5
```

```
    print('')
    print("Total Steps = ",steps)
    print('')

    for i in range(steps):
       print("Interval is:",5.0*float(i)/float(steps), 5.0*(float(i)+1.0)/float(steps))
       tmp = euler(lambda t, y: 0.1*y, 5.0*float(i)/steps, 5.0*(float(i)+1.0)/steps, tmp, 1)
       print ("Interval", i+1,tmp, "Exact", 1000*math.exp(0.1*5.0*(float(i)+1.0)/steps))

    # Plotting of results
    t2 = N.arange(1,101,1)
    y= N.zeros(100)
    for i in range (100):
        y[i] = euler(lambda t, y: 0.1*y, 0, 5, 1000,t2[i])

    pylab.plot (t2, y)
    pylab.title('Final solution for y at t=5 for M steps')
    pylab.xlabel ('M'); pylab . ylabel ('y(t=5)')
    pylab.show()
```

Inside Python we can use

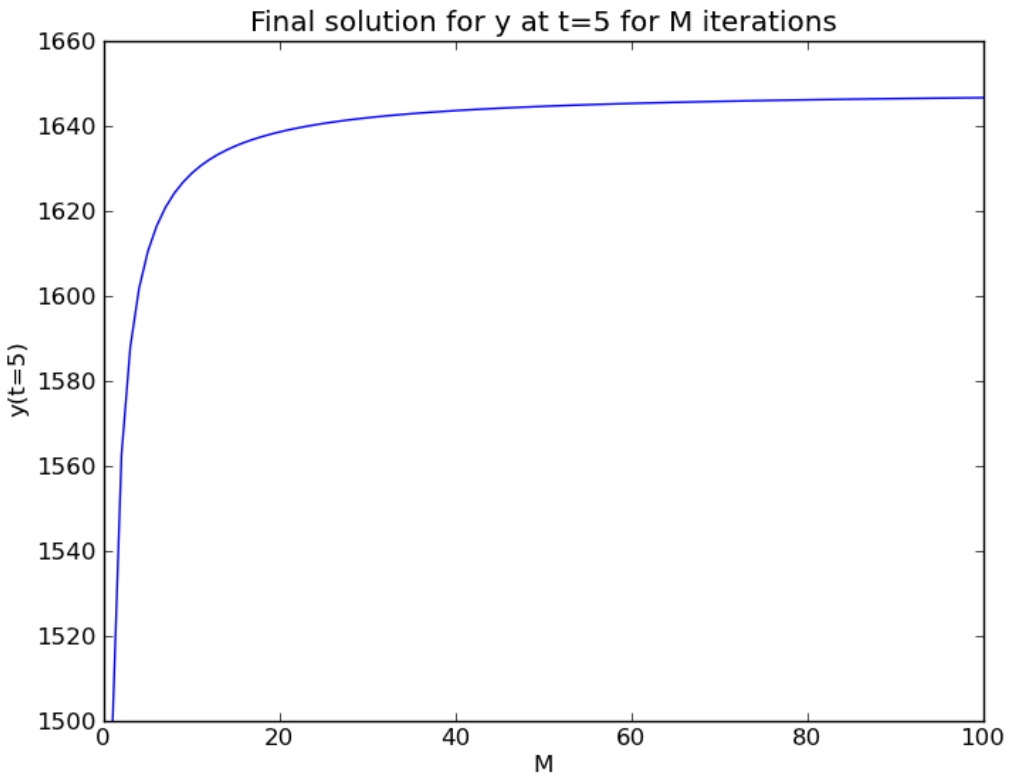
```
>>> execfile("euler2.py")
('5      steps:', 1610.51)
('60     steps:', 1645.3089347785883)
('100    steps:', 1646.6684921165452)
('1800   steps:', 1648.6068013396516)
```

If we run with 100,000 steps we get the solution

```
('100000 steps:', 1648.719209806687)
```

This agrees with the analytic solution (3.16) above to 2 decimal places: £1648.72. Another useful reference implementation is at [6], which also enables you to compare the Python implementation with other languages such as C.

A graph showing how the solution converges as we add more steps and thus decrease the interval size for each step is:



It takes a very small step size to get to an accurate solution (and bear in mind that at 100 iterations we are still only at £1646.67 as the approximate solution). Why? Each successive iteration continues from the result of the previous iteration, so small errors accumulate unless we proceed in very small steps (small h or equivalently large M).

Let us re-run and watch each iteration:

```
('Total Steps = ', 5)

('Interval is:', 0.0, 1.0)
('Interval', 1, 1100.0, 'Exact', 1105.1709180756477)
('Interval is:', 1.0, 2.0)
('Interval', 2, 1210.0, 'Exact', 1221.40275816017)
('Interval is:', 2.0, 3.0)
('Interval', 3, 1331.0, 'Exact', 1349.858807576003)
('Interval is:', 3.0, 4.0)
('Interval', 4, 1464.1, 'Exact', 1491.8246976412704)
('Interval is:', 4.0, 5.0)
('Interval', 5, 1610.51, 'Exact', 1648.7212707001281)
```

These small errors accumulate as the method proceeds. Geometrically each step we are taking a tangent from the function at each point and moving forward along it and then taking a new tangent and repeating.

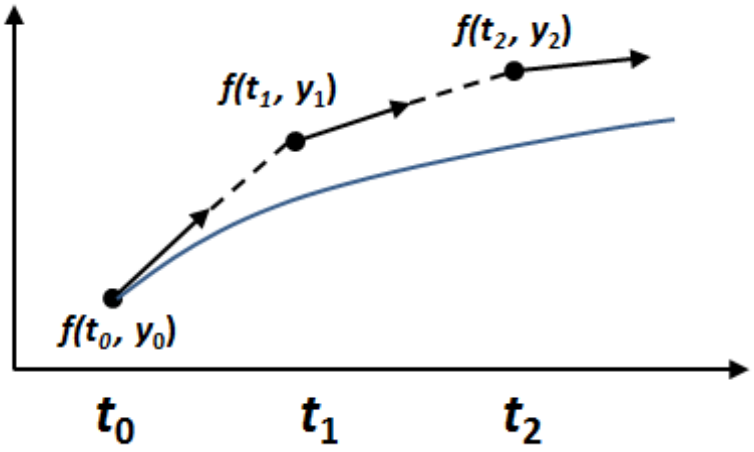


Figure 1 Geometric Interpretation of Euler's Formula (see also 7). We use the derivative at the start point of each interval and then extrapolate to find the next point.

We could clearly do better by taking high order terms in the Taylor expansion of (3.19) and ensure that our approximation to the curve is better. However, this involves growing complexity in computing the derivatives and (potentially) prior knowledge of how many such terms to take to enable the function to be represented accurately. We can also obtain second order accuracy by using the midpoint of the interval to estimate the derivative (see Figure 2).

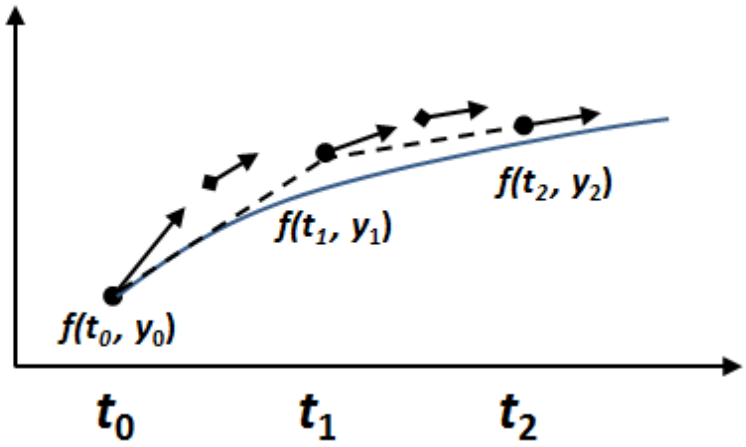


Figure 2 Geometric Interpretation of the "Midpoint method" (see also 7). We use the initial derivative at each step and then find a point halfway in the interval and use the derivative there across the whole interval to work out the next approximate value for the function.

But can we do even better?

3.4 Runge-Kutta Method

This class of methods effectively use a Taylor series of method of higher order and a fourth order method is usually a good compromise between speed and efficiency of the method and complexity of implementation, before it is probably better to consider some other ways to get an even better solution.

We can build on the concept of the midpoint method to take a trial point in the middle of the interval but take into account (in our notation) both *y* and *t* (or *x* more generally). Geometrically we can view this as

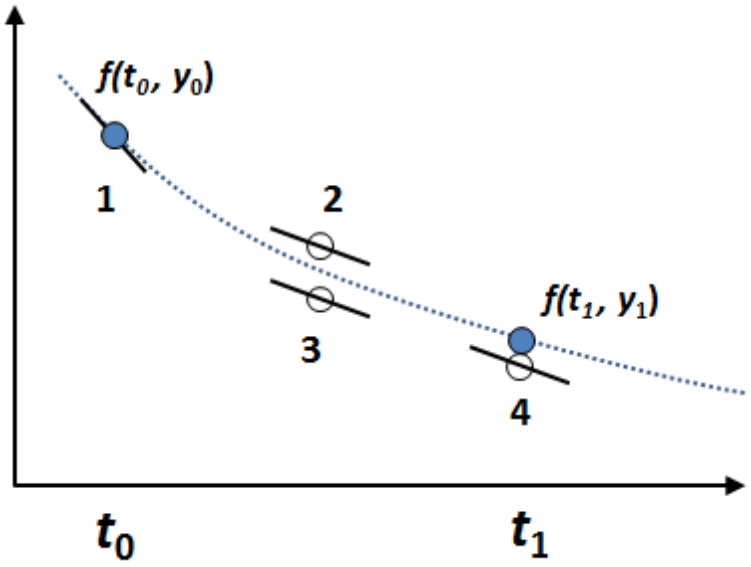


Figure 3 Geometric Interpretation of 4th order Runge-Kutta Method (see also 7).We evaluate the derivative four times (1, 2, 3, 4) and then estimate the new function value $f(t_1, y_1)$- see body of text for formulae and details.

We compute the next point as follows:

$$y_{k+1} = y_k + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4, \tag{3.23}$$

where (with a_1, a_2, a_3, b_1, b_2, … b_6, w_1, w_2, w_3, and w_4 constants to be found):

$$\begin{aligned} k_1 &= hf(t_k, y_k) \\ k_2 &= hf(t_k + a_1 h, y_k + b_1 k_1) \\ k_3 &= hf(t_k + a_2 h, y_k + b_2 k_1 + b_3 k_2) \\ k_4 &= hf(t_k + a_3 h, y_k + b_4 k_1 + b_5 k_2 + b_6 k_3) \end{aligned} \quad . \tag{3.24}$$

This the k_1, … k_4 are values being calculated at various points across the interval. After some derivation (see [1, p459]), we find that

$$\begin{gathered} a_1 = \tfrac{1}{2} \text{ (chosen)}; a_2 = \tfrac{1}{2}; a_3 = 1 \\ b_1 = \tfrac{1}{2}; b_2 = 0 \text{ (chosen)}; b_3 = \tfrac{1}{2}; b_4 = 0; b_5 = 0; b_6 = 1 \\ w_1 = \tfrac{1}{6}; w_2 = \tfrac{1}{3}; w_3 = \tfrac{1}{3} \text{ and } w_4 = \tfrac{1}{6}. \end{gathered} \tag{3.25}$$

Which gives

$$y_{k+1} = y_k + \frac{h\left(f_1 + 2f_2 + 2f_3 + f_4\right)}{6}, \tag{3.26}$$

Where

$$\begin{aligned} f_1 &= f(t_k, y_k) \\ f_2 &= f(t_k + \frac{h}{2}, y_k + \frac{h}{2} f_1) \\ f_3 &= f(t_k + \frac{h}{2}, y_k + \frac{h}{2} f_2) \\ f_4 &= f(t_k + h, y_k + h f_3) \end{aligned}, \qquad (3.27)$$

These correspond to the points 1, 2, 3, 4 marked in Figure 3. A complete derivation of this method is given in e.g. [1, 8 or 9]. Along with the geometric view of the method, we can also think of the derivation and equations in a similar way to how you may have used Simpson's rule for numerical integration.

3.5 Example – Runge-Kutta

(Repeated from previous analytic example). Suppose we had £1000 and deposited in a bank account earning 10% interest compounded per year, how much would we have after 5 years (from [1] Ex 9.3)? The differential equation governing the amount of money, *y*, is:

$$y' = 0.1y \text{ over } [0,5] \text{ with } y(0) = 1000 \qquad (3.28)$$

The solution (see above section 3.2), for an initial investment of £1000, is

$$y(5) = 1000 \times e^{0.1\times 5} = £1648.72 \text{ (2 d.p.)} \qquad (3.29)$$

Code for this is in [10] – you will need printSoln.py and run_kut4.py for this and below is the modified version of example 7.4 that solves the above example.

```
#!/usr/bin/python
## Call this runge_kutta.py
## example7_4 - modified by SJC to solve the example from Euler method section
from numpy import array,zeros
from printSoln import *
from run_kut4 import *
def F(x,y):
    F = zeros(1)
    F[0] = 0.1*y[0]
    return F

x = 0.0                 # Start of integration
xStop = 5.0             # End of integration
y = array([1000.0])     # Initial values of {y}
h = 1.0                 # Step size
freq = 1                # Printout frequency

X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
print ('Final answer:',Y[len(Y)-1])
raw_input("Press return to exit")
```

Running this at the command line using "**`python runge_kutta.py`**" gives

```
          x          y[ 0 ]
   0.0000e+00    1.0000e+03
   1.0000e+00    1.1052e+03
   2.0000e+00    1.2214e+03
   3.0000e+00    1.3499e+03
   4.0000e+00    1.4918e+03
```

```
   5.0000e+00    1.6487e+03
('Final answer:', array([ 1648.7206386]))
```

So even with a step size of 1 we get a solution of £1648.72 (2 d.p.) which agrees with analytic answer… that took >10,000 iterations to achieve using Euler's method.

Another useful reference implementation is at [11], which also enables you to compare the Python implementation with other languages such as C.

3.6 Beyond Runge-Kutta

- Adaptive methods allow us to take big steps when the function is smooth, but tiptoe more carefully when the function is varying more. A typical scheme might try a step size of h and then $2h$ and adapt accordingly.
- More sophisticated methods e.g. Runge-Kutta-Fehlberg (RKF45) is a further refinement of the method which also use a 4^{th} order and 5^{th} order approximation which enable the truncation error to be estimated and thence the step size to be adapted.
- The Bulirsch-Stoer Algorithm takes this one step further (no pun intended) and carefully extrapolates to what would happen if the step size was zero and judicious choice of approximation of the function to produce what is generally considered to be a very good way to solve a wide class of ordinary differential equation problems.
- Higher order ODEs can be converted to sets of first order equations which can be solved using the methods we have described.
- Buyer beware that methods can get stuck if the function has discontinuities in the range…
- You should also familiarise yourself with the "stability" of a method and whether the underlying equations themselves are "stiff" before blindly using a blackbox solver.

3.7 Other Ordinary Differential equations

- Initial value vs. two-point boundary value problems. We have considered ordinary differential equations that have an initial value so only satisfy a boundary at one end – the start in our example. The class of two-point ODEs are those where there are boundary conditions at two points: usually the start and end, though other cases (such as at interior or singular points) are generally considered as in the same class.
- The Shooting method is commonly used to solve these problems see e.g. [7].

4 Partial Differential Equations – applied methods and techniques for simple PDEs

4.1 Introduction

Along with the Ordinary Differential Equations we have considered previously, partial differential equations are another way in which we model problems in science and engineering. Their solution occupies a large amount of the time consumed on large scale high performance computers for tasks such as computational fluid dynamics, computational mechanics and computational electromagnetics which are used for modelling systems as diverse in size scale and complexity as climate and environmental modelling, the Universe, cars, planes, trains, ships and optical devices. Specialist courses exist which will study an individual application domain and its equations in huge detail and often highly tuned and specialised computational methods (and indeed even whole computer architectures and subsystems) have been developed for a particular system. Our purpose in this section is to introduce some of the key concepts and methods that underpin this universe of modelling possibilities.

4.2 Finite Difference methods

Our aim with these methods is to replace the differential operator with an approximation which averages over nearby points and by using a mesh of such points we derive a set of simultaneous equations to solve.

Consider the following two Taylor expansions of a function $f(x)$ around x at a (small) distance, h (see e.g. [3])

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(x) + \frac{h^4}{4!} f^{(4)}(x) + \dots \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2!} f''(x) - \frac{h^3}{3!} f'''(x) + \frac{h^4}{4!} f^{(4)}(x) - \dots \end{aligned} \tag{4.30}$$

If we subtract these two equations we get

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3} f'''(x) + \dots \tag{4.31}$$

Rearranging gives

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{4.32}$$

This is known as the *first central difference approximation* for the *first* derivative of $f(x)$. Higher order partial differential equations may contain terms where these derivatives are required to be discretized, see e.g. [4].

If, instead, we add them two together we get

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12} f^{(4)}(x) + \dots \tag{4.33}$$

Rearranging gives

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2) \tag{4.34}$$

This is known as the *first central difference approximation* for the *second* derivative of $f(x)$. It is possible to derive these for higher order derivatives to various orders of accuracy (see e.g. [4] for listings of these "stencils")

4.3 Example: Laplace equation in 1D

A simple time dependent heat equation reduces to Laplace's equation in steady state (once the system has come to equilibrium). This can be written as:

$$\frac{d^2u}{dx^2} = 0 \text{ with boundary conditions for } u(x) \tag{4.35}$$

for a 1 dimensional system: we will consider simple fixed boundary conditions for the problem.

Consider the problem which represents an infinitely thin rod of length 1 held at 0 degrees at one end and 100 degrees at the other.

$$\frac{d^2u}{dx^2} = 0 \text{ with } u(0)=0 \text{ and } u(1) = 100 \tag{4.36}$$

Discretizing gives us:

$$u''(x) = \frac{u(x-h) - 2u(x) + u(x+h)}{h^2} = 0 \text{ with } u(0)=0 \text{ and } u(1) = 100 \tag{4.37}$$

At each point on the rod, where the mesh separation is h.

(A) Let us consider just 1 unknown mesh point

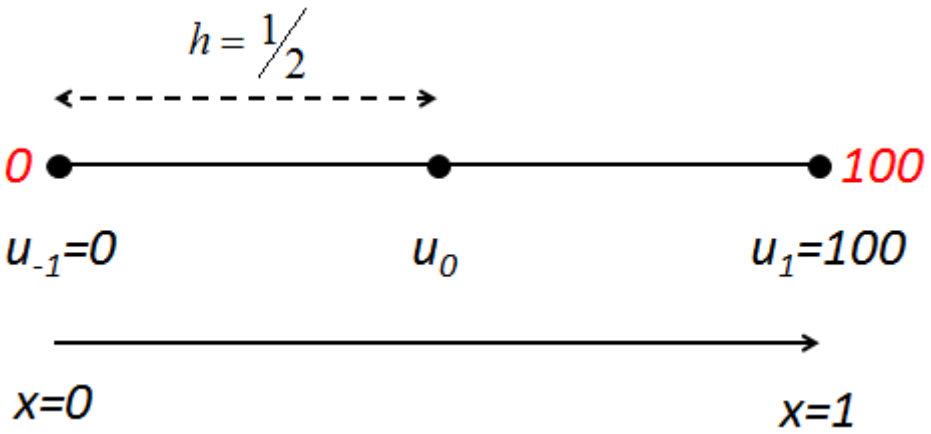


Figure 4 Laplace equation in 1D: 1 (unknown) mesh points

$$\frac{u_{-1} - 2u_0 + u_1}{h^2} = 0 \tag{4.38}$$

Since $u_{-1} = 0$ and $u_1 = 100$, we have:

$$\frac{0 - 2u_0 + 100}{(\frac{1}{2})^2} = 0 \tag{4.39}$$

Thus

$$\begin{aligned} 2u_0 &= 100 \\ \Rightarrow\ u_0 &= 50 \end{aligned} \tag{4.40}$$

(B) Now let us consider 2 (unknown) mesh points

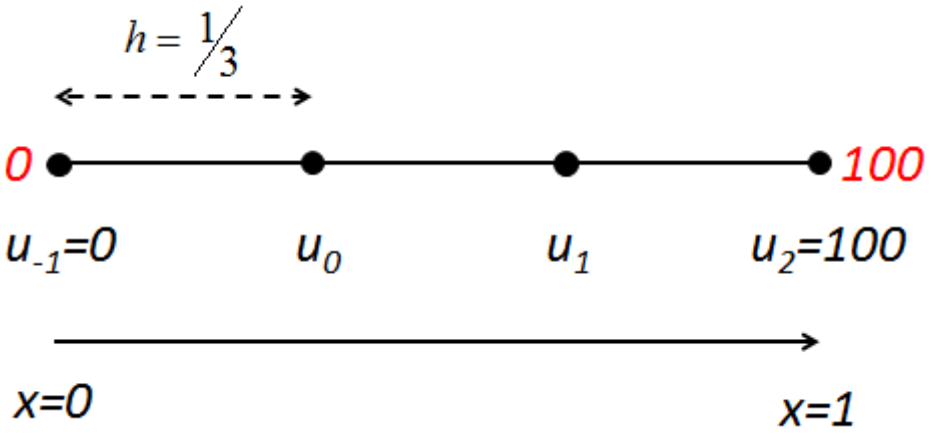


Figure 5 Laplace equation in 1D: 2 (unknown) mesh points

$$\left.\begin{aligned} \frac{u_{-1} - 2u_0 + u_1}{(\frac{1}{3})^2} &= 0 \\ \frac{u_0 - 2u_1 + u_2}{(\frac{1}{3})^2} &= 0 \end{aligned}\right\} \tag{4.41}$$

So:

$$\begin{aligned} 0 - 2u_0 + u_1 &= 0 \\ u_0 - 2u_1 + 100 &= 0 \end{aligned} \tag{4.42}$$

As a matrix this is

$$\begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix} \bullet \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -100 \end{pmatrix} \tag{4.43}$$

Solving gives

$$\begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = \begin{pmatrix} 33\frac{1}{3} \\ 66\frac{2}{3} \end{pmatrix} \tag{4.44}$$

(C) Now let us consider 3 (unknown) mesh points

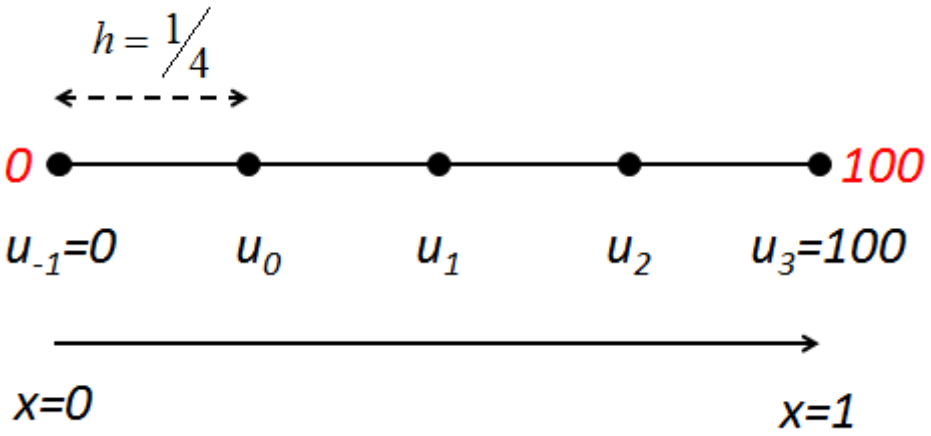


Figure 6 Laplace equation in 1D: 3 (unknown) mesh points

$$\begin{aligned} u_{-1} - 2u_0 + u_1 &= 0 \\ u_0 - 2u_1 + u_2 &= 0 \\ u_1 - 2u_2 + u_3 &= 0 \end{aligned} \tag{4.45}$$

As a matrix this is

$$\begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{pmatrix} \bullet \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -100 \end{pmatrix} \tag{4.46}$$

Solving gives

$$\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 25 \\ 50 \\ 75 \end{pmatrix} \tag{4.47}$$

Python gives

```
>>a=[[-2,1,0],[1,-2,0],[0,1,-2]]
>>b=[[0],[0],[-100]]
>>linalg.solve(a,b)
array([[ 25.],
       [ 50.],
       [ 75.]])
```

The pattern is clear and the solution makes sense – remember that we can solve this using the techniques from the first section (though there are specialised methods for "tri-diagonal matrices").

General pattern is for (N+2) mesh points:

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix} \bullet \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ -100 \end{pmatrix} \tag{4.48}$$

Whilst these solutions are straightforward our aim was to show the process of going from the differential equation to a set of linear equations that we can solve.

4.4 Example: Steady State for Heat equation in 2D

The generalisation of (4.35) in two dimensions is

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \text{ with boundary conditions for } u(x,y) \tag{4.49}$$

We can discretize this in the x and y direction:

$$\frac{u(x-h,y)-2u(x,y)+u(x+h,y)}{h_x^{\,2}} + \frac{u(x,y-h)-2u(x,y)+u(x,y+h)}{h_y^{\,2}} = 0 \tag{4.50}$$

with $u(x, 0)=100$, $u(x, 1)=0$, $u(0, y)$, and $u(1, y) = 0$ and h_x and h_y the mesh size in x and y respectively

If we use a uniform grid then $h_x = h_y = h$ and we have

$$\frac{u(x-h,y)+u(x,y-h)-4u(x,y)+u(x,y+h)+u(x+h,y)}{h^2} = 0 \tag{4.51}$$

We can view this as a "stencil" for averaging over nearby points (where "u" will remain as entries that go into a matrix and v is introduced as a discretisation for the domain)

The stencil we use is

$$\frac{v_{i-1,j}+v_{i,j-1}-4v_{i,j}+v_{i,j+1}+v_{i+1,j}}{h^2} = 0 \tag{4.52}$$

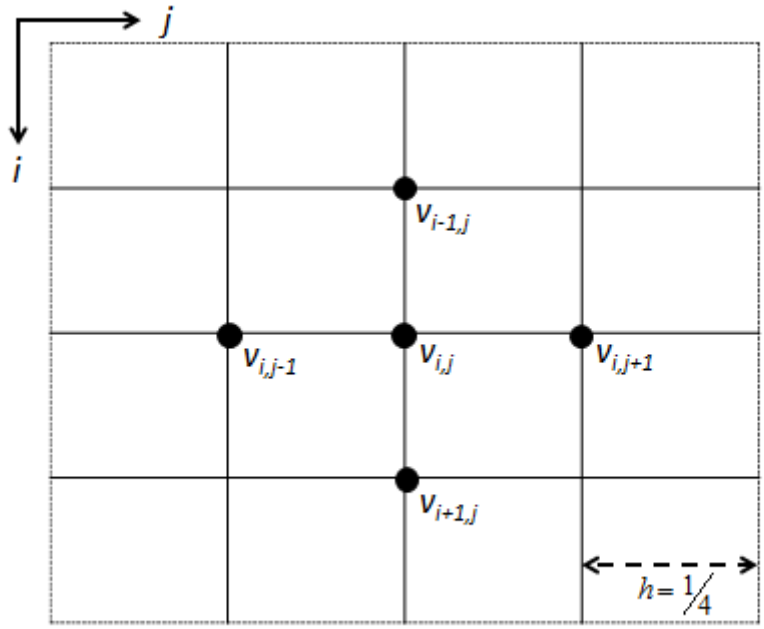


Figure 7 Stencil for *first central difference approximation* for the *second* derivative of function

Figure 8 shows this for a plate with fixed boundary conditions on the edges

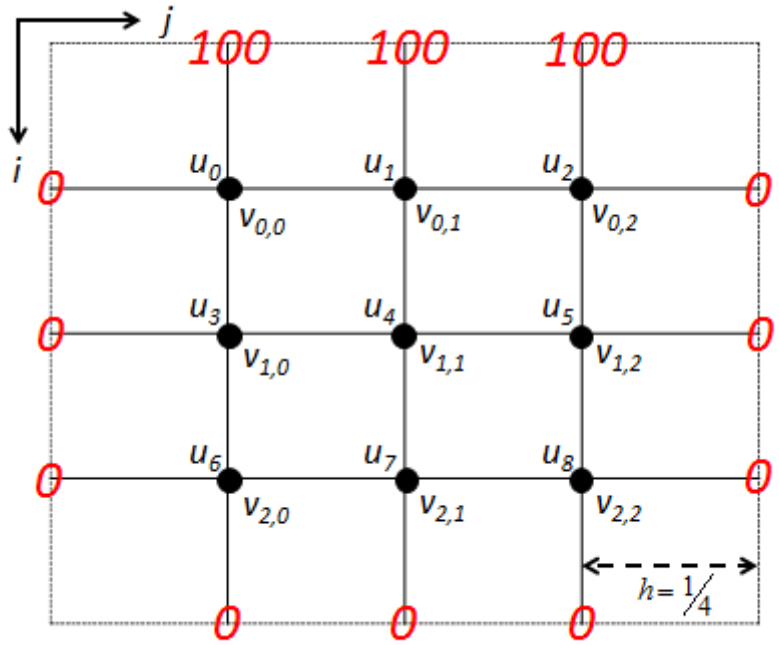


Figure 8 Laplace equation in 2D with mesh.

The plate is of unit size so $0 \le x \le 1$ and $0 \le y \le 1$ and the top edge is held at 100K

Note on index conversion. For an N x N uniform mesh of points so that $h = 1/(N+1)$ we have

$$v(i, j) = u(i \times N + j) \tag{4.53}$$

v		*u*
i	*j*	*k = i*N+j*
0	0	0
0	1	1
0	2	2
1	0	3
1	1	4
1	2	5
2	0	6
2	1	7
2	2	8

Figure 9 Index conversion in 2D

So $v_{i,j}$ can be used easily with a stencil representing a particular way of approximating the derivatives and u_k are indexes into an ACTUAL matrix. We have indexed from "0" in Python and C style. Other languages may index from 1.

(A) With 25 mesh points (so 9 unknowns) we have

(i) First row

$$\begin{aligned} 100+\ 0\ -4\,u_0\ +\ u_1\ \ \ \ +u_3\ \ \ \ =0 \\ 100\ +\ u_0\ -4u_1\ +\ u_2\ \ \ +u_4\ \ \ =0 \\ 100\ +\ u_1-4u_2\ \ \ \ +u_5=0 \end{aligned} \qquad (4.54)$$

(i) Second row

$$\begin{aligned} u_0\ \ \ \ -4u_3\ +\ u_4\ \ \ +u_6\ \ \ =0 \\ u_1\ \ \ +\ u_3-4u_4\ +\ u_5\ \ \ +u_7\ \ \ =0 \\ u_2\ \ \ +\ u_4-4u_5\ \ \ \ +u_8=0 \end{aligned} \qquad (4.55)$$

(i) Third row

$$\begin{aligned} u_3\ \ \ \ -4u_6\ +\ u_7\ \ \ =0 \\ u_4\ \ \ +\ u_6-4u_7\ +\ u_8=0 \\ u_5\ \ \ +\ u_7-4u_8=0 \end{aligned} \qquad (4.56)$$

We can write this as a matrix

$$\begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix} \bullet \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{pmatrix} = \begin{pmatrix} -100 \\ -100 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (4.57)$$

We can solve this using one of the techniques from the first section of the notes. As you can imagine for even more mesh points the matrix rapidly gets large and sparse. We need to unwrap the solution for *u* to map onto the mesh for *v*.

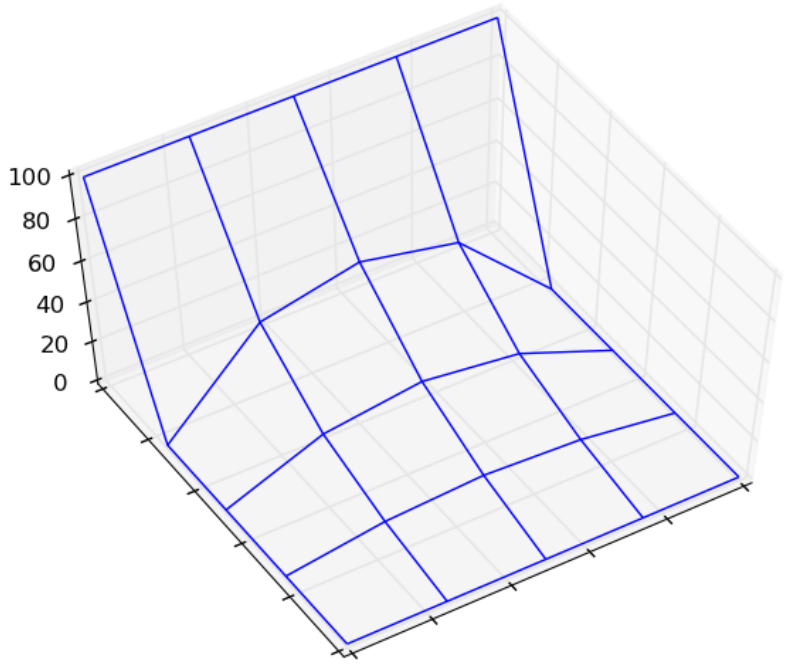


Figure 10 Laplace equation in 2D with mesh with 9 unknowns

For larger matrices we probably need a better way than typing it in by hand. This is the whole matrix for $n = 5$:

```
[[-4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1 -4  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  0  0  0  0 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  1  0  0  0  1 -4  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  1  0  0  0  0 -4  1  0  0  0  1  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  0  0  0  0  1  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0 -4  1  0  0  0  1  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  0  0  0  0  1]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0 -4  1  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4]]
```

Code Sample for the method – you may add your own notes to this. Note:

1) I deliberately have left in other debug/ testing other information to illustrate what a "live" piece of code might look like.
2) This code is NOT designed for efficiency, it is designed to show how all the steps work.
3) For demonstration of the principles, I have not been strict on ordering/ unwrapping operations matrix coordinates and real space (x,y).

```
import numpy,scipy

def embed(a,value):
    # Embed Matrix into an array with the boundary conditions in
    # a is the matrix and value is the value on the (fixed) boundary
    # http://wiki.scipy.org/NumPy_for_Matlab_Users
    size=a.shape[0]
    #print size
    a_tmp=numpy.zeros([size+2,size+2])

    for i in range(1,size+1):
        for j in range(1,size+1):
            a_tmp[i,j]=a[i-1,j-1]

    for i in range(0,size+2):
        a_tmp[0,i]=value
    return a_tmp

#Set up the printing of the array so it shows "nicely"
numpy.set_printoptions(precision=0,linewidth=120)

#n is the size of the mesh with the unknowns in it
# So the matrix will be of size (n+2)
```

```
n=3
n_full=n+2

#The h value is 1/(n+2) : taking into account the intervals
#to get to the boundary
h=1.0/n_full


# Clear matrix and set it up
a=numpy.zeros([n**2,n**2])

#Check indices
#for i in range(0,n):
#    for j in range(0,n):
#        print i,j,i*n+j
#print('===============')

print('===============')
print('Interior')


#Build full matrix
#Interior
for i in range(1,n-1):
    for j in range(1,n-1):
        north = (i-1)*n+j
        west  = i*n+j-1
        index=  i*n+j
        east  = i*n+j+1
        south = (i+1)*n+j

        a[index,north]=1
        a[index,west] =1
        a[index,index]=-4
        a[index,east] =1
        a[index,south]=1
        print i,j,index

print(a)
#Edges

#North/ Top    (nothing further North)
print('===============')
print("Top")

# First row number
i=0
#Note that the range (1,n-1) means that we JUST middle ones
#e.g. if n=5 then range(1,4) =[1,2,3]
for j in range(1,n-1):
    #north = (i-1)*n+j
    west  = i*n+j-1
    index=  i*n+j
    east  = i*n+j+1
    south = (i+1)*n+j

    #a[index,north]=1
    a[index,west] =1
    a[index,index]=-4
    a[index,east] =1
    a[index,south]=1
    print i,j,index
```

```
#West/ Left    (nothing further West)
print('===============')
print("West")

j=0 #First Column Number

for i in range(1,n-1):
    north = (i-1)*n+j
    #west  = i*n+j-1
    index=  i*n+j
    east  = i*n+j+1
    south = (i+1)*n+j

    a[index,north]=1
    #a[index,west] =1
    a[index,index]=-4
    a[index,east] =1
    a[index,south]=1
    print i,j,index

#East/ Right   (nothing further East)
print('===============')
print("East")

j=n-1 # Last Column number
for i in range(1,n-1):
    north = (i-1)*n+j
    west  = i*n+j-1
    index=  i*n+j
    #east  = i*n+j+1
    south = (i+1)*n+j

    a[index,north]=1
    a[index,west] =1
    a[index,index]=-4
    #a[index,east] =1
    a[index,south]=1
    print i,j,index

#South/ Bottom (nothing further South)
print('===============')
print("South")

i=n-1 # Last row number
for j in range(1,n-1):
    north = (i-1)*n+j
    west  = i*n+j-1
    index=  i*n+j
    east  = i*n+j+1
    #south = (i+1)*n+j

    a[index,north]=1
    a[index,west] =1
    a[index,index]=-4
    a[index,east] =1
    #a[index,south]=1
    print i,j,index

print('===============')
print("Corners")
```

```
#Top Left
i=0
j=0
index=  i*n+j
east  = i*n+j+1
south = (i+1)*n+j

a[index,index]=-4
a[index,east] =1
a[index,south]=1
print i,j,index

#Top Right
i=0
j=n-1
west  = i*n+j-1
index=  i*n+j
south = (i+1)*n+j

a[index,west] =1
a[index,index]=-4
a[index,south]=1
print i,j,index

#Bottom Left
i=n-1
j=0
north = (i-1)*n+j
index=  i*n+j
east  = i*n+j+1

a[index,north]=1
a[index,index]=-4
a[index,east] =1
print i,j,index

#Bottom Right
i=n-1
j=n-1
north = (i-1)*n+j
west  = i*n+j-1
index=  i*n+j

a[index,north]=1
a[index,west] =1
a[index,index]=-4
print i,j,index

print('===============')
# THIS IS THE FINAL MATRIX
print a
print("====================")
raw_input("Press return to continue")
print("")

#Reset printing options
numpy.set_printoptions()
numpy.set_printoptions(edgeitems=3,infstr='inf',linewidth=75, nanstr='nan',
precision=8,suppress=False, threshold=1000)
```

```
# Example 3: 2D Heat equation in steady state. i.e. Laplace equation with
boundary conditions.
print("====================")
print("Set up b")
b=numpy.zeros([n**2,1])

for i in range(0,n):

    b[i]=-100.
    print i
for i in range(n,n**2):
    b[i]=0.

print(b)
print("====================")
raw_input("Press return to continue")
print("")

# This sets it up by hand as a check
#b=[[-100],[-100],[-100.0],[0],[0],[0],[0],[0],[0]]
print("In 2D With 9 unknowns, Solution is:")
#Could use iterative method from first part of notes
soln = numpy.linalg.solve(a,b)

print(soln)
print("====================")
raw_input("Press return to continue")
print("")

# Wrap the solution onto grid and embed
soln_wrap=numpy.reshape(soln,[n,n])
soln_full=embed(soln_wrap,100)
print soln_full

#3D Plotting part
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

steps = 2.0+n
print steps

h=1.0/(steps-1)
print('h=',h)
X = np.arange(0, steps, 1)*h
print(X)
print
Y = np.arange(0, steps, 1)*h
X, Y = np.meshgrid(X, Y)

print("X is:")
print(X)

print("Y is:")
print(Y)

#R = np.sqrt(X**2 + Y**2)
```

```
#surf = ax.plot_wireframe(X, Y, R, rstride=1, cstride=1)

#surf = ax.plot_surface(X, Y, soln_full, rstride=1, cstride=1, cmap=cm.coolwarm,
#        linewidth=0, antialiased=False,shade=True)
surf = ax.plot_wireframe(X, Y, soln_full, rstride=1, cstride=1)
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
raw_input("Press return to continue")
```

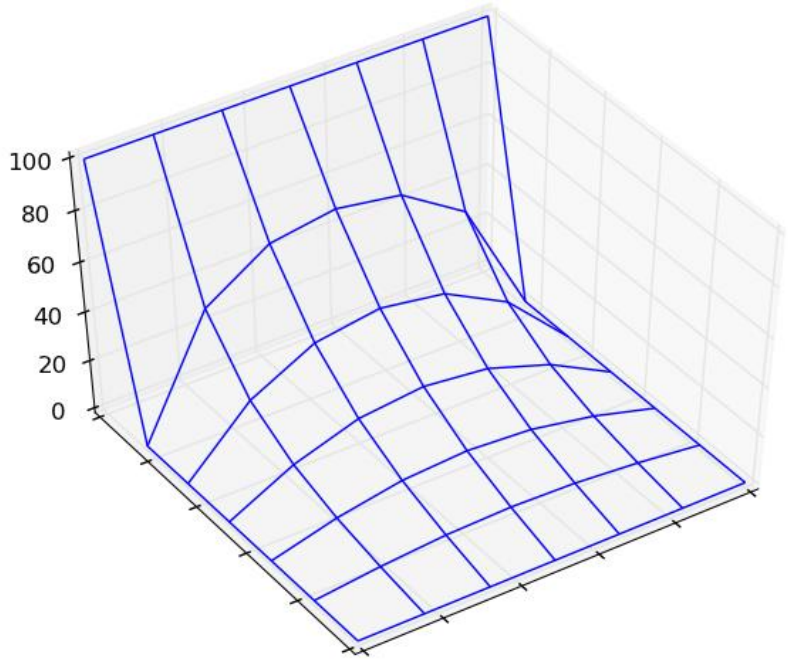


Figure 11 Laplace equation in 2D with mesh with 25 unknowns (so $n = 5$)

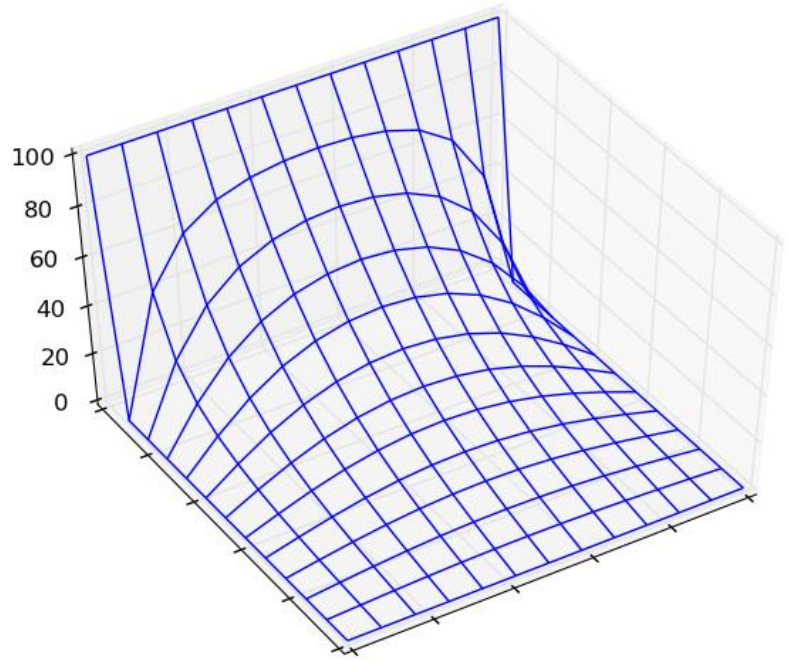


Figure 12 Laplace equation in 2D with mesh with 121 unknowns (so $n = 11$)

…at this point, we should move away from using "solve" and use an iterative method for the solution of the equations, which is where we started in the first part of the notes.

4.5 Order of methods for solving equations

The order of a solver can dramatically change the time to a solution – indeed as much or much more than just buying a bigger computer (or waiting for Moore's Law) e.g. (where n is the size of the matrix)

- Gaussian Elimination/ Dense LU methods ~$O(n^3)$
- Jacobi type ~$O(n^2)$
- Gauss-Seidel ~$O(n^2$: can be better)
- (Optimal) Successive Over-relaxation ~$O(n^{3/2})$
- Conjugate Gradient ~$O(n^{\sim 3/2})$ (depending on precise pre-conditioner used)
- Sparse LU ~$O(n^{3/2})$
- Fast Fourier Transform ~$O(n \log n)$
- Multigrid ~$O(n)$

4.6 Other type of equations

There are a number of different types of differential equations and, in particular, time dependent equations require further procedures where not only is it important to work with a fine enough mesh for the spatial (x, y) components, but also the time steps, t. These include:

- Heat flow: Temperature (space and time)
- Diffusion: Concentration (space and time)
- Electrostatic or Gravitational potential: Potential (space)
- Electromagnetic field strength: Field (space and time)
- Fluid flow: Velocity, Pressure, and Density (space and time)
- Semiconductor modelling: Electron density (space and time)
- Quantum Mechanics: Wave function (space and time)
- Elasticity: Stress and Strain (space and time)

4.7 Other methods

Along with finite difference methods, other very broad categories of methods for solving partial differential equations include:

- Finite element and finite volume methods
- Meshfree or meshless methods
- Particle-based methods
- Monte Carlo methods
- Spectral methods

… and others yet to be invented and researched.

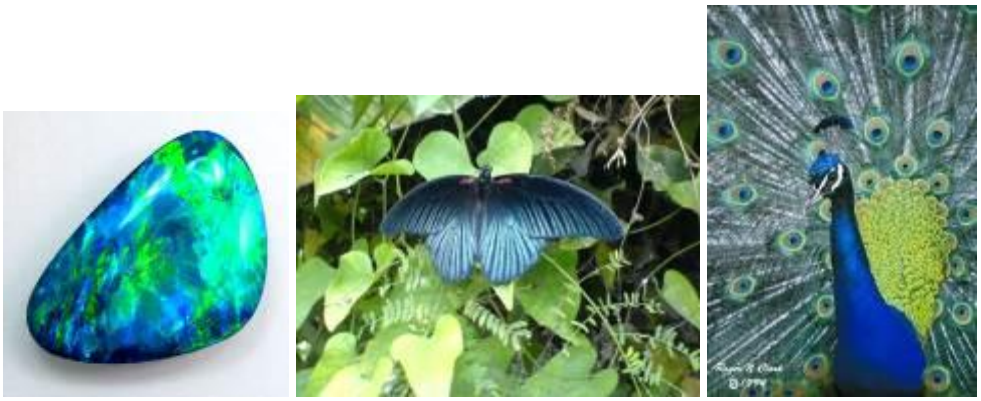
5 Eigenvalue problems

5.1 Introduction

Along with the differential equations we have covered to date, there is another class of problems that are important in science and engineering: eigenvalue problems. They are often associated with vibrations and they also have some interesting and unique solvers which bring together a number of the techniques and methods we have covered.

Examples of eigenvalue "problems"

1) Tacoma Narrows Bridge Collapse:
   http://www.youtube.com/watch?v=3mclp9QmCGs
   http://www.youtube.com/watch?v=j-zczJXSxnw
2) Millenium Bridge:
   http://www.youtube.com/watch?v=eAXVa__XWZ8
   http://www.youtube.com/watch?v=gQK21572oSU
3) Butterflies, Opals, and Peacocks…



(Image copyright as per metadata)

5.2 Background

The generalised eigenvalue problem for two n x n matrices A and B, is to find a set of eigenvalues λ and (non-trivial) vectors x such that:

$$A\,x = \lambda\, B\,x \quad \text{with } x \neq 0 \tag{5.58}$$

If B is the identity matrix, then this reduces to the eigenvalue problem:

$$A\,x = \lambda\, x \quad \text{with } x \neq 0 \tag{5.59}$$

We will focus on the latter equation as techniques to solve the generalised problem are an extension of solving the reduced problem and can be found in the literature [e.g. 12]. Many of the matrices that occur in science and engineering – often as a result of discretizing differential equations – have special properties that can be exploited when solving the eigenvalue problem.

5.3 Example: Hand Calculation

We can solve the eigenvalue problem as follows. If we have

$$A\,x = \lambda\, B\,x \quad \text{with } x \neq 0 \tag{5.60}$$

Then

$$\left(A - \lambda I\right)x = 0 \quad \text{with } x \neq 0 \tag{5.61}$$

For each {eigenvalue, eigenvector} pair. This is true when the following condition on the determinant of the matrix is true:

$$\det(A-\lambda I)=0 \quad \text{with } x\neq 0 \tag{5.62}$$

Consider

$$A=\begin{pmatrix}3 & -1 & 0\\ -1 & 2 & -1\\ 0 & -1 & 3\end{pmatrix} \tag{5.63}$$

We can find the eigenvalues by solving

$$\det(A-\lambda I)=\begin{vmatrix}(3-\lambda) & -1 & 0\\ -1 & (2-\lambda) & -1\\ 0 & -1 & (3-\lambda)\end{vmatrix}=0 \tag{5.64}$$

Which gives

$$(3-\lambda)\begin{vmatrix}(2-\lambda) & -1\\ -1 & (3-\lambda)\end{vmatrix}-(-1)\begin{vmatrix}-1 & -1\\ 0 & (3-\lambda)\end{vmatrix}+(0)\begin{vmatrix}-1 & (2-\lambda)\\ 0 & -1\end{vmatrix}=0 \tag{5.65}$$

Thus we have the characteristic polynomial:

$$\begin{aligned}&(3-\lambda)(2-\lambda)(3-\lambda)-(3-\lambda)-1(3-\lambda)=0\\ \Rightarrow\ &-\lambda^3+8\lambda^2-21\lambda+18-3+\lambda-3+\lambda=0\\ \Rightarrow\ &-\lambda^3+8\lambda^2-19\lambda+12=0\end{aligned} \tag{5.66}$$

This can be solved to give solutions for the eigenvalues as {1,4,3}. It is also possible to work out that the corresponding eigenvectors are:

$$\lambda_1=1 \quad a\begin{pmatrix}1\\2\\1\end{pmatrix};\ \lambda_2=4 \quad b\begin{pmatrix}1\\-1\\1\end{pmatrix};\ \lambda_3=3 \quad c\begin{pmatrix}1\\0\\-1\end{pmatrix}, \tag{5.67}$$

Where *a*, *b*, and, *c* are arbitrary constants. We can use Python to help us with this symbolically

```
>>> from sympy import *
>>> L = symbols('L')
>>> M=Matrix( [[(3-L),-1,0], [-1,(2-L),-1] , [0 , -1 , (3-L)]])
>>> M.det()
-L**3 + 8*L**2 - 19*L + 12
>>> solve(M.det())
[1, 4, 3]
```

But writing out the characteristic polynomial and solving it is generally not a good way to solve as the matrix gets larger.

Clearly if the matrix was of the form

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 3 \end{pmatrix} \tag{5.68}$$

Then we would find the eigenvalues by writing

$$\det(A - \lambda I) = \begin{vmatrix} (1-\lambda) & 0 & 0 \\ 0 & (4-\lambda) & 0 \\ 0 & 0 & (3-\lambda) \end{vmatrix} = 0 \tag{5.69}$$

Thus we have the characteristic polynomial:

$$(1-\lambda)(4-\lambda)(3-\lambda) = 0 \tag{5.70}$$

Which can be trivially solved to give solutions for the eigenvalues as {1,4,3}.

Our first method will attempt to transform the matrix into this form and will also yield the eigenvalues

5.4 Jacobi Method for Eigenvalues

This method is a fairly robust way to extract all of the eigenvalues and eigenvectors of a symmetric matrix. Whilst it is probably only appropriate to use for matrices up to 20 by 20, the principles of how this method operates underpin a number of more complicated methods that can be used more generally to find all of the eigenvalues of a matrix (assuming that finding such eigenvalues is actually a well-posed/ stable/ sensible problem).

The method is based on a series of rotations (Jacobi rotations) which are chosen to eliminate off-diagonal elements. Whilst successive rotations will undo previous set zeros the off-diagonal elements get smaller until eventually we are left with a diagonal matrix. By accumulating products of the transformations as we proceed we obtain the eigenvectors of the matrix. See e.g. [7] for additional details.

Consider the transformation matrix P_{pq}

$$P_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5.71}$$

All diagonal elements are unity apart from two elements c in rows p and q and all off-diagonal are zero apart from the element s and $-s$ (in rows p and q also). We pick

$$\begin{aligned} c &= \cos\phi \\ s &= \sin\phi \end{aligned} \text{, where } \phi \text{ is a rotation angle} \tag{5.72}$$

This transformation matrix is a plane rotation which can transform the matrix:

$$\tilde{A} = P_{pq}^T \bullet A \bullet P_{pq} \tag{5.73}$$

The key concept is that $P_{pq}^T \bullet A$ only changes the rows p and q of A, whilst $A \bullet P_{pq}$ only changes the columns p and q of A and EACH STEP we can judiciously choose the rotation to "zero-off" the elements at these intersection points of the rows and columns p and q.

…

Additional steps/ proofs can be found in e.g. [7].

…

Eventually this leads to a matrix, D, that is diagonal to machine precision or a pre-designated tolerance, whose elements are the eigenvalues of A:

$$D = V^T \bullet A \bullet V \tag{5.74}$$

where

$$V = P_1 \bullet P_2 \bullet P_3 \bullet \cdots \tag{5.75}$$

With P_i being the successive Jacobi rotation matrices and the columns of V are the eigenvectors since $A \bullet V = V \bullet D$

Various enhancements to this basic routine are deployed to determine the order of zeroing off the elements.

Sample code for this can be found in [3]. This code saved as "**jacobi_eig_ex.py**" is a driver routine for it, where we have renamed the code in [3] to "jacobi_eig" to differentiate from the Jacobi method we covered previously and made some tweaks.

```
import numpy as np
import scipy
from numpy import linalg
from jacobi_eig import jacobi_eig

import matplotlib.pyplot as plt

#Create full matrix
#Example 1
A= np.array([[3.,-1,0], [-1,2,-1] , [0 , -1 , 3]])

#Example 2
A= np.array([[8.,-1,3,-1], [-1,6,2,0] , [3 , 2 , 9, 1], [-1, 0, 1, 7]])

#Show A
#print(A)

#Find eigenvalues of A

#Python Jacobi Example
w,v = jacobi_eig(A,tol =  1.0e-9)

# Now sort them into order with argsort
idx = w.argsort()

w = w[idx]
v = v[:,idx]

print('Eigenvalues:')
print w
print ''

print('Eigenvectors:')
```

```
print v
print ''
```

The specific changes to the sample code [3] are

ORIGINAL	MODIFIED
`## module Jacobi`	`## module jacobi_eig`
`def jacobi_eig(a,tol = 1.0e-9`	`def jacobi_eig(a,tol = 1.0e-9)`

`rotate(a,p,k,l)`	`rotate(a,p,k,l)`
	`# Extra debug`
	`print('Step:',i)`
	`print(a)`
	`print'----------------'`

Examples of use with "**`python jacobi_eig_ex.py`**" at the command line

1) For $A = \begin{pmatrix} 3 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 3 \end{pmatrix}$

```
Eigenvalues:
[ 1.  3.  4.]

Eigenvectors:
[[  4.08248290e-01   7.07106781e-01   5.77350269e-01]
 [  8.16496581e-01  -2.59319211e-10  -5.77350269e-01]
 [  4.08248290e-01  -7.07106781e-01   5.77350270e-01]]
```

2) For $A = \begin{pmatrix} 8 & -1 & 3 & -1 \\ -1 & 6 & 2 & 0 \\ 3 & 2 & 9 & 1 \\ -1 & 0 & 1 & 7 \end{pmatrix}$

```
Eigenvalues:
[  3.29569866   6.59233804   8.40766196  11.70430134]

Eigenvectors:
[[ 0.52877937  0.23009661 -0.57304222  0.58229764]
 [ 0.59196687 -0.62897514  0.47230121  0.17577558]
 [-0.53603872 -0.07123465  0.28204972  0.79248727]
 [ 0.2874545   0.73916943  0.60745546  0.04468031]]
```

We can also examine successive steps in the routine to show the zeroing of elements at each step along with reduction in their size even when they are made non-zero in successive iterations.

5.5 Givens and Householder reduction and QR method

For the Jacobi method we iterated until the matrix was diagonal. A refinement of this is to iterate until the starting symmetric matrix is tridiagonal – this is the basis for Givens reduction. Householder's method is a further refinement which is more efficient, but achieves the same aim. Once the matrix is in tridiagonal form, the eigenvalues can be found efficiently.

The QR method uses the same procedure of transformations, but instead focuses (1) on producing a matrix which has zeros in the upper triangle and the diagonal just below this known as the "Hessenberg form" and then (2) on finding the eigenvalues of the resulting Hessenberg matrix. Reduction to the Hessenberg form may use an approach analogous to Gaussian Elimination with pivoting, which is a technique used in the solution of linear equations. These techniques are described in detail in e.g. [7].

5.6 Example: Vibrating String modes

Consider waves on a vibrating string: in differential equation terms we are separating out any time dependent behaviour about how the string might move and just looking at the normal modes of oscillation. This leads to a boundary value eigenvalue problem

$$\phi''(x) + k^2\phi(x) = 0 \text{ with } \phi(0)=0 \text{ and } \phi(L)=0 \qquad (5.76)$$

for a 1 dimensional system which describes the vibration of a string of length L which is pinned at both ends. It can be shown analytically that the solutions of this are $k = n\pi$ where $n = 1, 2, 3, \ldots$ for a string of length $L = 1$.

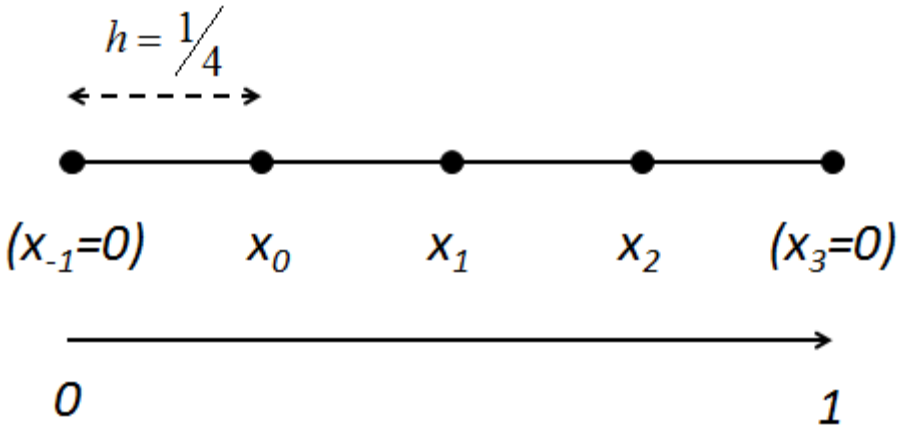


Figure 13 Vibrating String configuration

We can use the following to discretize the operator:

$$\phi''(x) = \frac{\varphi(x-h) - 2\varphi(x) + \varphi(x+h)}{h^2} \quad \text{with } \phi(0) = 0 \text{ and } \phi(L) = 0 \qquad (5.77)$$

At each point on the string, where the mesh separation is h.

This leads to an eigenvalue problem

$$\frac{\varphi(x-h) - 2\varphi(x) + \varphi(x+h)}{h^2} + k^2\varphi(x) = 0 \quad \text{with } \phi(0) = 0 \text{ and } \phi(L) = 0 \qquad (5.78)$$

Simplifying and introducing λ we have

$$-\varphi(x-h)+2\varphi(x)-\varphi(x+h)=\lambda\,\varphi(x) \tag{5.79}$$

Note where we have incorporated the sign change and we will need to scale the λ values to get back to the original k:

$$k=\sqrt{\lambda/h^2} \tag{5.80}$$

Writing out in full we have

$$\begin{aligned} -x_{-1}+2x_0-x_1 &= \lambda\, x_0 \\ -x_0+2x_1-x_2 &= \lambda\, x_1 \\ -x_1+2x_2-x_3 &= \lambda\, x_2 \end{aligned} \tag{5.81}$$

We have $x_{-1} = x_3=0$ so this yields an eigenvalue problem:

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \bullet \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \lambda \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \tag{5.82}$$

which we can solve using the techniques above.

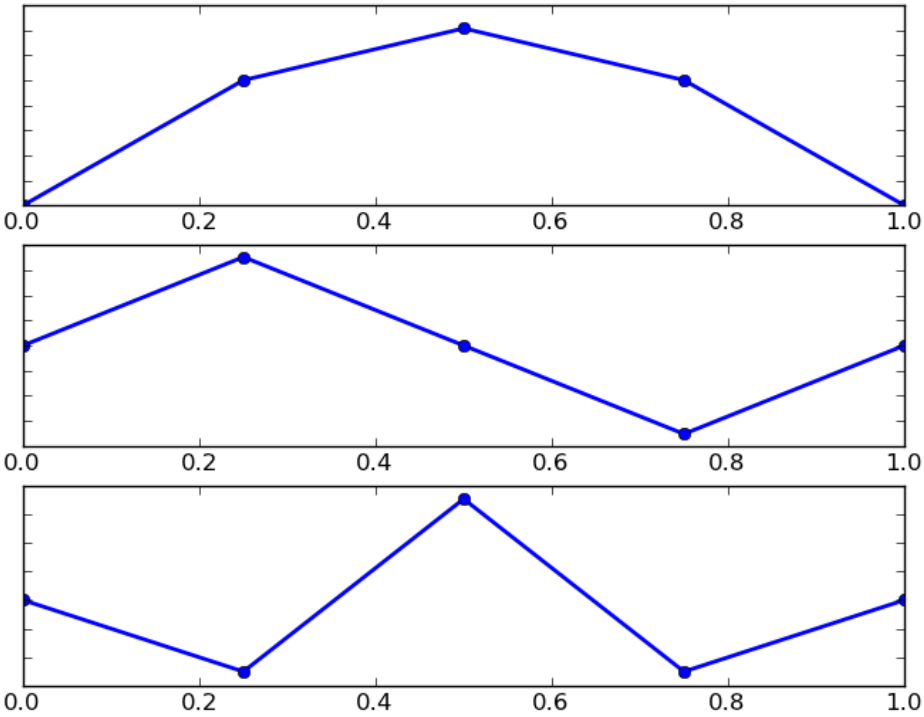


Figure 14 Vibrating String modes (3 unknowns, plot 3)

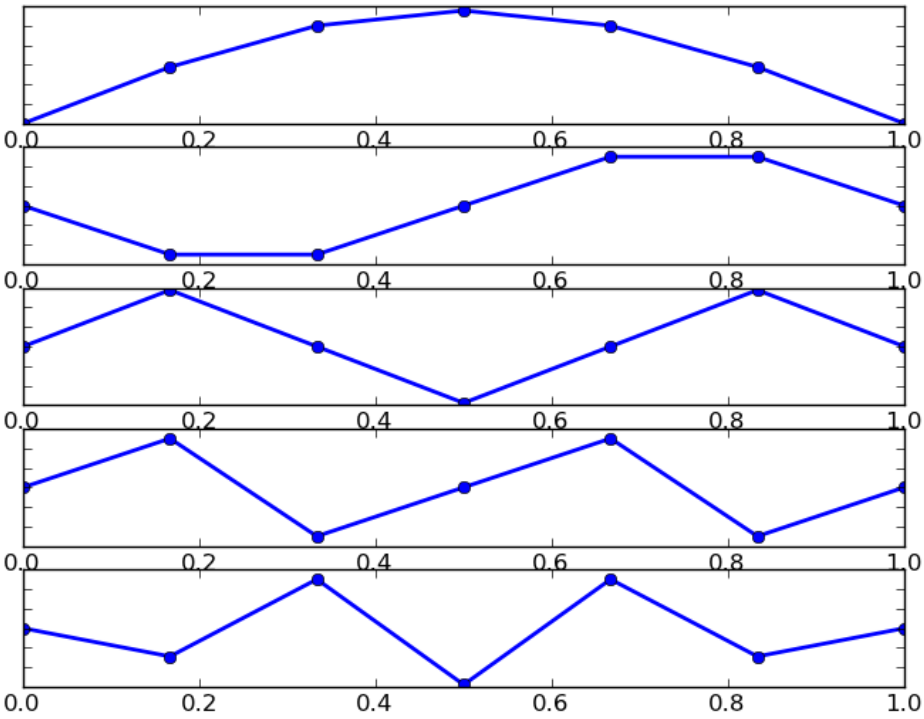


Figure 15 Vibrating String modes (5 unknowns, plot 5)

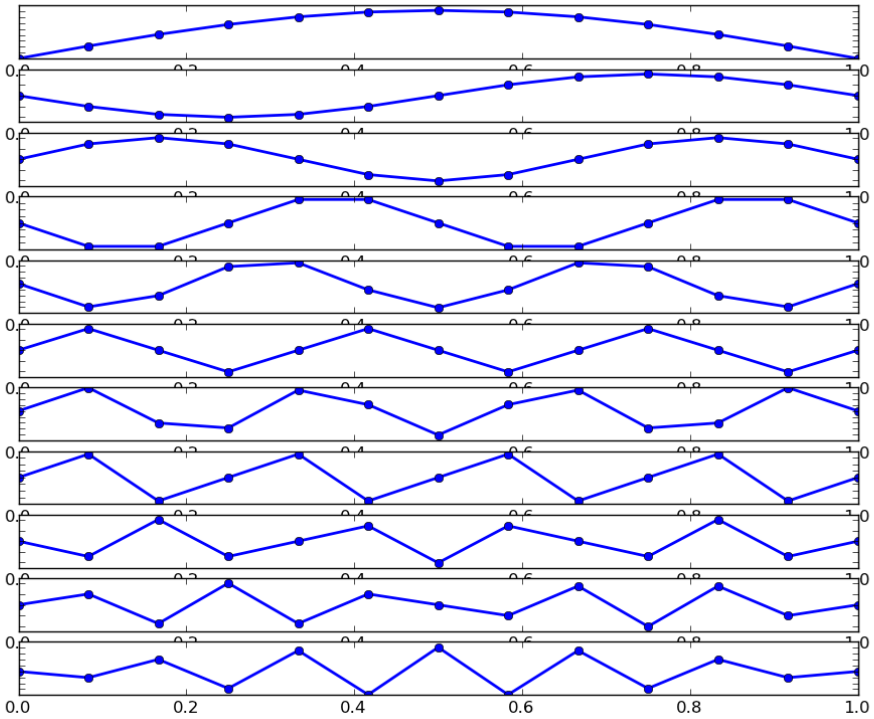


Figure 16 Vibrating String modes (11 unknowns, plot 11)

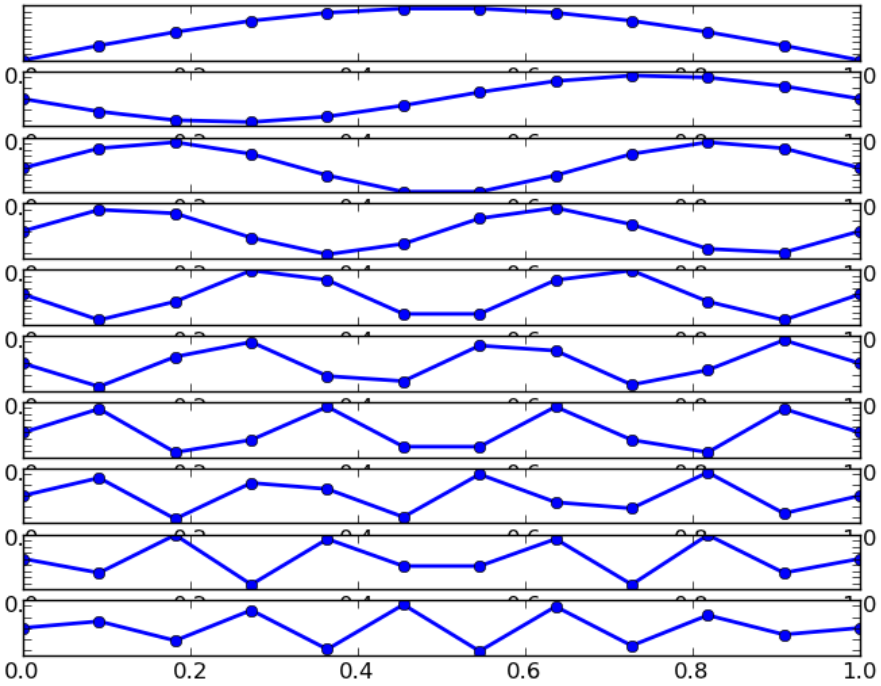


Figure 17 Vibrating String modes (10 unknowns, plot 10)

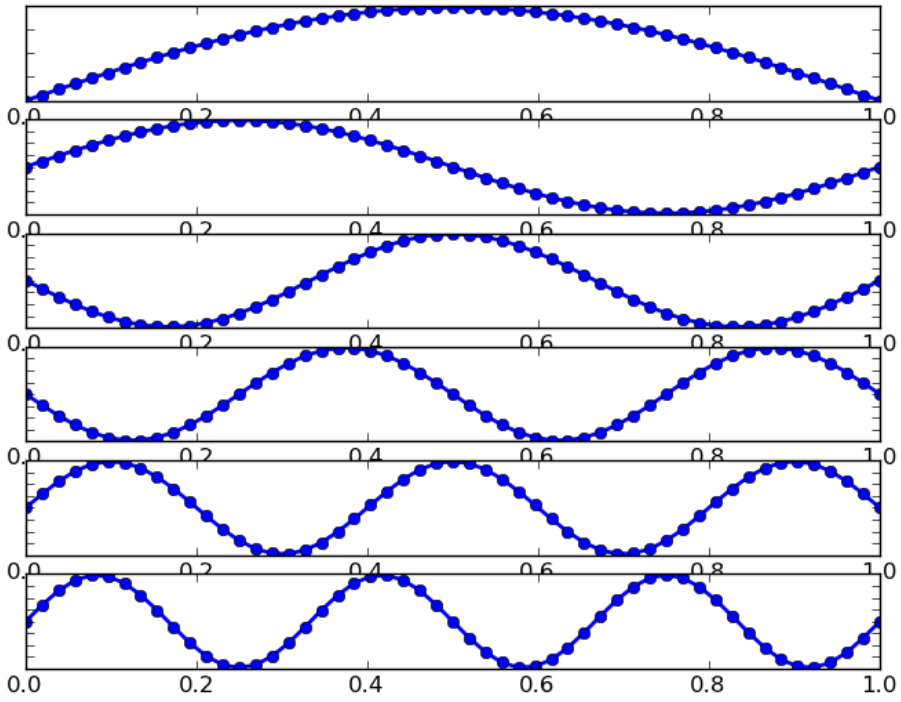


Figure 18 Vibrating String modes (51 unknowns, plotting 5 solutions)

Some observations:

1) We can see the convergence of the eigenvalues to their correct analytic values as we improve the mesh size.
2) We need to use a finer mesh to capture the higher modes of oscillation of the function… but we (generally) observe that only about the lower (1/3) of the eigenmodes are captured well for any mesh size: this is simply due to sampling the wave, which is a frequent class of science and engineering calculations where eigenproblems arise.
3) The matrices have very few non-zero entries in them: they are "sparse"
4) It seems somewhat wasteful to be calculating all of the eigenvalues: not only because of (2) but often we might only need a few of them – perhaps only the smallest or largest.
5) Be careful whether you have an odd or even number of mesh points- think about the problem

We could do better on (1) by buying a bigger computer and just increasing the mesh size or using a higher order discretization for the function. (2) is a property of the underlying physics and those laws are well known to be harder to change. By using better data structures for storing/ handling the arrays we can also address (3). (4) [and by implication not being wasteful because of (2)] can be addressed by using a different algorithm, and this is where we turn our attention next.

Code sample ("normal" caveats apply- NOT designed for efficiency, 'debug' left in, etc.)

```
import numpy as np
import scipy
import math
from numpy import linalg
from numpy import ones
from numpy import linspace
from jacobi_eig import jacobi_eig
import matplotlib.pyplot as plt

#Number of unknowns to solve for (i.e. mesh points in middle of string)
n=5
h=1.0/(n+1)
#print h
#Number of eigenvalues to plot
eigs_to_plot = 5

# Create leading diagonal with 2 on it and(-1) for entries above and below
a=2*ones(n)
b=(-1)*ones(n-1)

#Create full matrix
A= np.diagflat(a)+np.diagflat(b,-1)+np.diagflat(b,1)
#Show A
#print(A)

#Find eigenvalues of A
# Note that The normalized (unit length) eigenvectors, such that the
# column v[:,i] is the eigenvector corresponding to the eigenvalue w[i]

#Python Jacobi Example
w,v = jacobi_eig(A,tol = 1.0e-9)
#Built in version: uses QR
#w,v = linalg.eig(A)
# Now sort them into order with argsort
idx = w.argsort()
# DON't forget to rescale them with "h"
w = w[idx]
kk= np.sqrt(w/h**2)
v = v[:,idx]

print('Eigenvalues:')
print w
print ''

print('k values:')
print kk
print ''

print('Eigenvectors:')
print v
print ''

#Now do some plotting
x_grid=np.linspace(0,1,n+2)
#Append the fixed values of 0 onto the arrays
v=np.concatenate( (np.zeros([1,n]),v,np.zeros([1,n])),axis=0)

for i in range(min(n,eigs_to_plot)):
    ax = plt.subplot(eigs_to_plot ,1,i+1)
    ax.plot(x_grid, v[:,i], 'bo-',linewidth=2)
    ax.set_yticklabels([])

plt.show()
#raw_input("Press return to exit")
```

5.7 Finding just a few eigenvalues: Power and Inverse Power Method

There are also methods that will find just a few of the eigenvalues of a large matrix such as the power or inverse power method. These "power type" iterative methods generally become more useful and relevant when considering larger scale systems in science and engineering. Extension to matrices with complex numbers as entries are also possible. These extend into Krylov subspace methods and (implicity restarted) Arnoldi Iteration13.

5.8 Other eigenvalue methods

Methods also exist to find the eigenvalues of real symmetric matrices matrix such as Jacobi's and Householder's method. QR can be used to find all the eigenvalues of a real symmetric and positive definite matrix. It is also possible to exploit structures such as when the matrix is tridiagonal (such as the one above) using as Sturm sequences (see e.g. [3, 8, 9]). For nonsymmetric matrices, general solution can be more difficult but one should study carefully any problem that yields such a matrix to ensure that a different formulation of the problem (or mesh labelling or discretization technique) will yield a symmetric variant for the same problem – though this may not always be possible.

5.9 Python

In Python we have the eig function (based on QR). For the above example, it operates as follows.

```
>>> from numpy import linalg
>>> w, v = linalg.eig(matrix([[3,-1,0],[-1,2,-1],[0,-1,3]]))
>>> w
array([ 1.,  3.,  4.])
>>> v
matrix([[  4.08248290e-01,  -7.07106781e-01,   5.77350269e-01],
        [  8.16496581e-01,   4.02265225e-16,  -5.77350269e-01],
        [  4.08248290e-01,   7.07106781e-01,   5.77350269e-01]])
```

There is no "built-in" Python equivalent in numpy at the time of writing to "eigs" in Matlab for just finding a few eigenvalues of a matrix [see e.g. 14], however there is a sparse solver which can find a few eigenvalues in scipy: this interfaces to ARPACK [15].

6 Monte Carlo Methods

These methods use random numbers to sample an experiment and can be used to solve a number of applied computational modelling problems such numerical integration, solution of partial differential equations, eigenvalue problems and queuing problems. They are also useful in simulating physical systems or determining statistical fluctuations or uncertainties that might arise from a process.

7 References and useful links

7.1 Main books

Whilst there is no single book which covers all of the material for the course, you should find much of it in either of the first two books.

Burden, RL and Faires, JD (2005) "Numerical Analysis." Brooks/Cole ISBN 0534404995.

Fausett L (2007) "Applied Numerical Analysis: Using Matlab" Pearson Education. ISBN 0132397285.

Press, WH, Teukolsky, SA, Vetterling, WT, and Flannery BP (1992, 1996, 2007, and later) "Numerical Recipes in C" , "Numerical Recipes in Fortran", "Numerical Recipes 3rd Edition". These books contain both the code and algorithms. See also www.nr.com for more details and you can also read these books online there too. The PDFs are at http://www.nrbook.com/a/bookcpdf.php (marked as old and obsolete and secured with the FileOpen plugin.)

For more detailed mathematics and proofs of equations see

Stoer J and Bulirsch R (2010) "Introduction to Numerical Analysis" Springer. ISBN 144193006X

7.2 Python Information

This is a really good Python book on Numerical Methods with a focus on Engineering:

Jaan Kiusalaas "Numerical Methods in Engineering with Python" Hardcover: 432 pages Publisher: Cambridge University Press; 2nd edition (29 Jan 2010). ISBN-10 is 0521191327 and ISBN-13 is 978-0521191326. NOTE that the new 3rd Edition is also available from March 2013 (ISBN: 9781107033856) but this is based on Python 3 NOT Python 2.5/2.7.

The source code is available under "Resources" at:

http://www.cambridge.org/gb/academic/subjects/engineering/engineering-mathematics-and-programming/numerical-methods-engineering-python-2nd-edition

For an introduction to Python examples on numerical methods, see Chapter 16 of Prof Fangohr's notes at:

http://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering.pdf [last checked October 2013]

This online tutorial is useful: http://www.learnpythonthehardway.org/

If you have a Windows machine, then you can use

- Python Tools for Visual Studio at http://pytools.codeplex.com/
- and get Visual Studio through Dreamspark: http://www.dreamspark.com/
- Winpython: http://code.google.com/p/winpython/

Devices based around ARM processors running various flavours of Linux are now cheaply and readily available. You might consider getting yourself a Raspberry Pi. Python is one of the languages used on this device:

http://www.raspberrypi.org/

We also use the Beaglebone Black:

http://beagleboard.org/

8 Matlab/ Python Appendix

Matlab includes extensive help and tutorials. Other packages offering a high-level 'problem solving environment (PSE)' for computational modelling include Python (open source), Mathematica (numerical and symbolic calculations), and Maple (symbolic calculations). Spreadsheets such as Excel can also be useful for simple calculations and visualization of results.

8.1 MATLAB and Python features

MATLAB and Python are versatile and interactive tools for performing numerical calculations.

Can be used for

- testing algorithms
- running small programs
- interactive visualisation of data

Other features:

- Specialist toolboxes can be used to solve particular problems
- Numerical Computation
- Interaction visualization and presentation graphics
- High Level Programming Language based on vectors and matrices
- Specialist toolboxes written by experts
- Tools for interface building
- Integrated debugger, editor and performance profiler
- On-line electronic documentation

8.2 Availability of MATLAB and Python

The Matlab User's guide, software, and thousands of pages of online documentation is available in a student version. It is on iSolutions PC Clusters and the major University High Performance Computing (HPC) facilities.

Python is available as Open Source and is on iSolutions machines and widely available on desktop machines, tablet PCs (including Windows, Android and iOS systems) and boards/ devices such as the Raspberry Pi and Beaglebone family. Versions can be found for almost all major operating systems.

8.3 Matlab Summary

8.3.1 Basic Features

- Mathematical calculations can be typed in as you would write them
- Variables are defined using '`a = 3`'
- Once a variable is defined, it can be used in mathematical expressions
- The commands `who` and `whos` display information about variables

8.3.2 Help

- Typing `help <command>` gives a summary of the command
- `lookfor` allows you to search for a keyword
- Under Windows, `help` can be accessed interactively and online

8.3.3 Array Operations

- Commas separate columns of a matrix : "`,`"
- Use a semicolon: "`;`" to start a new row
- To index individual array elements use

  e.g. `x(1), x(3)`

8.3.4 Colon Notation

The colon can be used in several ways

- To index an array:

```
x(start element: step : last)
```

e.g. `x(1:2:5)` returns elements `x(1), x(3), x(5)`

- To construct an array:

```
x=(first number : step : last)
```

e.g. `x=(2:2:6)` is the same as `x=[2,4,6]`

8.3.5 Array Mathematics

Array Manipulation

- Elementwise:

`a.*b` gives $a_{1,1}\times b_{1,1}$ and $a_{1,2}\times b_{1,2}$ etc.

- Conventional linear algebra (the dimensions of the matrices must be compatible)

`a*b` means use $a_{1,1}\times b_{1,1} + a_{1,2}\times b_{2,1} + a_{1,3}\times b_{3,1}$ etc.

8.3.6 2D Plotting

- `plot(x1,y1,s1, x2,y2,s2, x3,y3,s3,...)` places plots of the vectors `(x1,y1)` with style `s1` and `(x2,y2)` with style `s2` etc. on the same axes.
- `xlabel('text')` adds a label to the x axis.
- `ylabel('text')` add a label to the y axis.
- `grid on` turns on a grid over the plot.

8.3.7 Other plots

- `semilogy(x,y)` and `semilogx(x,y)` gives axes marked in powers of 10.
- `loglog(x,y)` plots both axes with a logarithmic scale.
- MATLAB supports many common types of plot (see the booklet).
- `fill(x,y,s)` draws a fills a polygon with the colour `r`.

8.3.8 3D Plotting

- `plot3(x1,y1,z1,s1, x2,y2,s2,z2, .. )` plots the points defined by the triples `(x1,y1,z1)` with style `s1` and `(x2,y2,z2)` with style `s2` etc. on the same axes.
- `zlabel('text')` adds a label to the z axis.
- `mesh(x,y,z)` draws a wire frame grid for the surface defined by `(x,y,z)`.
- `surf(x,y,z)` gives a shaded surface plot for the surface defined by `(x,y,z)`.

8.3.9 3D plotting

- Change the shading using `colormap(map)`

- To examine a coloured map of a matrix, A, use `imagesc(A)`
- `colorbar` displays the colour coding for the matrix shading

8.3.10 Programming MATLAB

- MATLAB provides loops using

```
for k = 1:n
[instructions]
end
```

- MATLAB commands can be put together in a script (or text) file to group together a set of instructions.
- MATLAB also provides tools to build user-friendly interfaces for programs.
- A good comparison of Python and Matlab equivalents using NumPy is at [14] [last checked Oct 2013]

9 Credits and other book/ resources used

Some examples in these notes are based on and derived from various books and texts (in print and out of print). I have given these for reference so you can see them with additional context or with their original notation/ implementation details:

[1] Mathews, J.H. and Fink, K.D. "Numerical methods using Matlab: 3rd edition" Prentice-Hall. ISBN 0132700425. There is a 4th edition of this available (ISBN-13: 978-0130652485)

[2] From http://quantstart.com/articles/Jacobi-Method-in-Python-and-NumPy (last checked Oct 2013)

[3] Jaan Kiusalaas "Numerical Methods in Engineering with Python" Hardcover: 432 pages Publisher: Cambridge University Press; 2nd edition (29 Jan 2010). ISBN-10: 0521191327 and ISBN-13: 978-0521191326

[4] Daniel Zwillinger "Handbook of Differential Equations" Hardcover, Academic Press. ISBN 0127843965. 3rd (Revised) Edition (29th Oct 1997). A CD-Rom Version is available too.

[5] Based: http://code.activestate.com/recipes/577647-ode-solver-using-euler-method/ (last checked Oct 2013)

[6] http://rosettacode.org/wiki/Euler_Method (last checked Oct 13) Useful for comparing between languages e.g. :
http://rosettacode.org/wiki/Euler_Method#Python (Oct 13)
http://rosettacode.org/wiki/Euler_Method#C (Oct 13)

[7] Press, WH, Teukolsky, SA, Vetterling, WT, and Flannery BP (1992, 1996, 2007, and later) "Numerical Recipes in C" , "Numerical Recipes in Fortran", "Numerical Recipes 3rd Edition". These books contain both the code and algorithms. See also www.nr.com for more details and you can also read these books online there too.

[8] Stoer J and Bulirsch R (2010) "Introduction to Numerical Analysis" Springer. ISBN 144193006X

[9] Quarteroni, A., Sacco, R., and Saleri F. "Numerical Mathematics" (Texts in Applied Mathematics 37) Springer-Verlag 2000 ISBN 0387989595. There is also a second edition (2007) with ISBN 3540346589

[10] Jaan Kiusalaas "Numerical Methods in Engineering with Python" Hardcover: 432 pages Publisher: Cambridge University Press; 2nd edition (29 Jan 2010). ISBN: 0521191327 (see notes above in section 7.2)

[11] http://rosettacode.org/wiki/Runge-Kutta (last checked Oct 13) Useful for comparing between languages e.g. :
http://rosettacode.org/wiki/Runge-Kutta#Python (Oct 13)
http://rosettacode.org/wiki/Runge-Kutta#C (Oct 13)

[12] Golub, GH and Van Loan, CF "Matrix Computations" Johns Hopkins University Press; third edition edition (15 Oct 1996) ISBN-10: 0801854148

[13] See http://www-users.cs.umn.edu/~saad/books (last checked Oct 13)
Books available include both excellent works by Yousef Saad:
Saad, Y. "Numerical methods for large eigenvalue problems" (2011)

http://www-users.cs.umn.edu/~saad/eig_book_2ndEd.pdf
Saad, Y. "Iterative Methods for Sparse Linear Systems" (2003)
http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf

[14] http://wiki.scipy.org/NumPy_for_Matlab_Users (last checked Oct 13)

[15] http://docs.scipy.org/doc/scipy/reference/tutorial/arpack.html (last checked Oct 13)