Advanced Computational Methods 1
FEEG6002
PDE Methods

Samuel F. Gilonis
24267856
sg2e10@soton.ac.uk

Introduction

The objective of this assignment is to solve linear Partial Differential Equations (PDEs) using numerical methods rather than analytically. The numerical methods to be explored include Python's inbuilt solver (Section 1), the Successive Over-Relaxation (SOR) method (Section 2) and the "red-black" Gauss-Seidel method (Section 4). These numerical methods are to be implemented using the Python programming language along with an exploration of different stencils for the discretisation of a PDE by averaging over nearby points (Section 3), and a brief discussion of the Multigrid method (Section 5).

1 Question 1: Using Python's inbuilt solver

1.1 Improving the code

In this section a new and more efficient version of the code presented in Section 4 of the course notes [2] is used to produce an output that checks whether the solution obeys $\nabla^2 u = 2$ at $(0.5, 0.5)$.

To improve speed and to enhance readability, the code which populates the matrix $\mathbf{A}$ was rewritten as shown in Appendix 6.2. This code uses far fewer lines of code (55 rather 229) than the version in the notes. There is also an important error in the code given in the notes: the grid spacing, h, is given as:

$$h = \frac{1}{n+2}$$

However, this is incorrect. The grid will have $n+2$ grid points in each direction (as it includes the boundaries) but the grid will be $n-1$ cells wide. Therefore h should be given by:

$$h = \frac{1}{n+1} \tag{1}$$

1.2 Setting up the matrices

The PDE to be solved is Poisson's equation in two-dimensions:

$$\nabla^2 u = \rho \tag{2}$$

i.e.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho \tag{3}$$

Where $u = u(x, y)$, $\rho(0.5, 0.5) = 2$ and $\rho = 0$ elsewhere where $x, y \in [0, 1]$ and $u = 0$ at the boundaries.

The first method employs a first order central finite difference scheme using a uniform grid spacing, h, and therefore has an equal number of grid points in the x and y directions. This is given by:

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

The grid nodes can be rearranged into a column vector so that a matrix problem can be solved to find a solution to the PDE as a matrix:

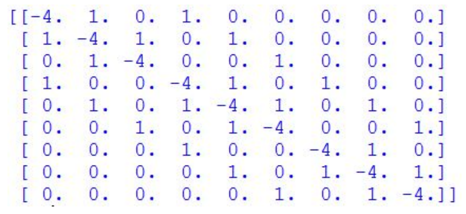
$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{5}$$

A regular ordering method will be implemented, i.e. it will be of the form [row,column] or $[i,j]$ where the bottom left index will be $[0,0]$ and the top right index will be $[n-1,n-1]$ for a grid with n points in each direction.

Using this indexing we can write the stencil used to discretise the domain as:

$$\frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j}}{h^2} = \rho \tag{6}$$

This can then be expressed in the form of a a matrix of coefficients multiplied by the column vector **u**. The matrix of coefficients, **A**, is shown in Figure 1.



```
[[-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.]]
```

Figure 1: The matrix **A**, for the stencil shown in Equation 6 and where $n = 3$

As $\rho(0.5, 0.5) = 2$ is in the middle of the domain, we must have an odd number of grid points, i.e. n must be odd in order for this boundary condition to be satisfied. As the **b** matrix must have the same number of elements as the **u** matrix (which must have $n \times n$ elements), to create the **b** matrix we only need to create a column vector of n zeros and then set the middle element equal to 2 (since $\rho(0.5, 0.5) = 2$).

Having set up the matrices **A** and **b** we can now use Python's inbuilt solver, 'numpy.linalg.solve', to calculate the column vector **u**.

1.3 Checking the code

Using numpy's linalg.solve function we can obtain the values for **u**, e.g. when $n = 3$ the following values are obtained for $\mathbf{u}h^2$:

$$[-0.125 - 0.25 - 0.125 - 0.25 - 0.75 - 0.25 - 0.125 - 0.25 - 0.125]$$

These values can then be rearranged into a 3×3 matrix (as $n = 3$) and then plotted against the spatial coordinates x and y. We can then use the value of u at the midpoint to obtain the Laplacian at the midpoint using Equation (6) and thus check that the boundary condition has been satisfied. The code was implemented successfully and an exact value of 2.0 was returned within floating point accuracy (the error was of the order of 10^{-16}).

A plot of u against x and y can be seen for $n = 3$ and $n = 51$ in Figures 2 and 3 respectively.
For $n = 51$ the inbuilt solver took 2.237s to complete with an absolute error of 4.4408920985e-16.

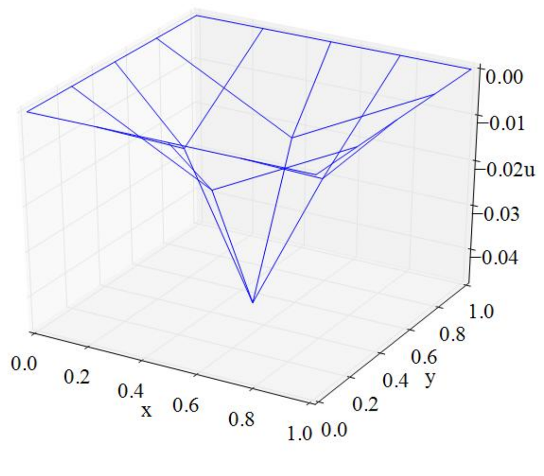


Figure 2: Graph showing the variation of u in two dimensions when $n = 3$

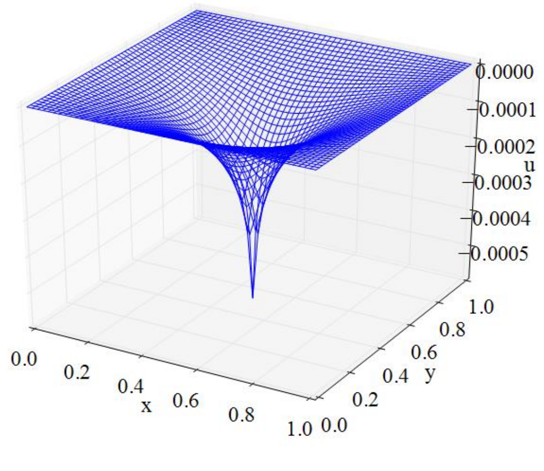


Figure 3: Graph showing the variation of u in two dimensions when $n = 51$

2 Question 2: Successive Over-Relaxation

2.1 Finding u with an iterative method

The objective of this question is to solve the system of linear equations ($\mathbf{A} \cdot \mathbf{u} = \mathbf{b}$) using a variation on the Gauss-Seidel method, known as the Successive Over-Relaxation (SOR) method, rather than Python's inbuilt solver. The algorithms implemented are taken from [2]:

$$u_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} u_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} u_j^{(k)} \right] + (1-\omega) u_i^{(k)}, i = 1, 2, ..n \tag{7}$$

This is very similar to the Gauss-Seidel method but it employs the relaxation parameter, ω. According to [1] the method is convergent for $0 < \omega < 2$ where if $\omega < 1$there is "under relaxation" and if $\omega > 1$ then there is "over relaxation". An estimate for the optimum value of ω is given by the expression given in [1]:

$$\omega_{opt} \approx \frac{2}{1 + \sqrt{1 - (\Delta u^{(k+p)} / \Delta u^{(k)})^{1/p}}} \tag{8}$$

Where p is a positive integer. The code to implement this method is shown in Appendix 6.3.

Initially a rough estimate for ω must be made (e.g. $\omega = 1$), then $k + p$ iterations are carried out so that $\Delta u^{(k+p)}$ and $\Delta u^{(k)}$ can be obtained and substituted into Equation 8 so that ω_{opt} can be found. All subsequent iterations are carried out using this value for ω. The iterations will continue until either the maximum number of iterations permitted has been reached or when the magnitude (or Euclidean norm) change in u changes less between iterations than some specified tolerance.

2.2 Efficiency compared with the inbuilt method

As the direct method likely employs array operations, we should expect that the inbuilt solver will perform very well at low values of n but that it will compare negatively with iterative methods, that rely on for-loops, when computing larger numbers of grid points. A comparison of the speed of the two methods is presented in Table 1.

n	**Time Taken**	**(s)**
	Numpy solver	**SOR solver**
3	0.0000	0.0000
11	0.0000	0.1430
21	0.0320	1.9870
31	0.1680	6.5340
41	0.7060	12.7100
51	2.1530	25.3860
61	5.6260	34.2710
71	13.4200	51.6670

Table 1: Comparison of the computational efficiency of direct and iterative solvers

At values of n higher than 71 the code begins to fail due, displaying, "MemoryError". When $n = 21$ the direct method is 62 times faster than the iterative method. When $n = 51$ this has dropped to 11.8 times faster and when $n = 71$ the direct method is only 3.85 times faster. It can therefore be reasonably assumed that for very high values of n the iterative method will become faster although this would require a high performance computer or to speed-up the code. This is discussed in Section 5.

3 Question 3: Implementing a fourth order finite differencing scheme

Whereas the first central difference approximation for the second derivative of a function, the five-point stencil, uses the four neighbouring grid points of a point (to its 'north', 'south', 'east' and 'west'). We can implement a higher order nine-point stencil that will also use the grid points that are in the same direction as the other four but that are two grid points removed from the current point.

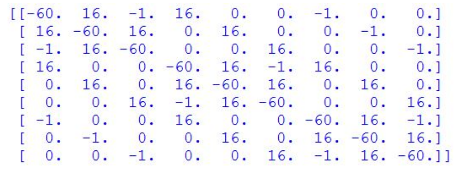
This fourth order central differencing stencil utilises the approximation:

$$f"(x) = \frac{-f(x-2h) + 16f(x-h) - 30f(x) - f(x+2h)}{12h^2} + \mathcal{O}(h^4) \tag{9}$$

Assuming uniform grid spacing in the x and y directions we can obtain the Laplacian, given below in nodal indexing:

$$\nabla^2 u = \frac{-(u_{i-2,,j} + u_{i+2,j} + u_{i,j-2} + u_{i,j+2}) + 16(u_{i-1,,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) - 60u_{i,j}}{12h^2} + \mathcal{O}(h^4) \tag{10}$$

This gives the matrix of coefficients $\mathbf{A}$, shown in Figure 4.



```
[[-60.  16.  -1.  16.   0.   0.  -1.   0.   0.]
 [ 16. -60.  16.   0.  16.   0.   0.  -1.   0.]
 [ -1.  16. -60.   0.   0.  16.   0.   0.  -1.]
 [ 16.   0.   0. -60.  16.  -1.  16.   0.   0.]
 [  0.  16.   0.  16. -60.  16.   0.  16.   0.]
 [  0.   0.  16.  -1.  16. -60.   0.   0.  16.]
 [ -1.   0.   0.  16.   0.   0. -60.  16.  -1.]
 [  0.  -1.   0.   0.  16.   0.  16. -60.  16.]
 [  0.   0.  -1.   0.   0.  16.  -1.  16. -60.]]
```

Figure 4: The matrix $\mathbf{A}$, for the stencil shown in Equation 10 and where $n = 3$

The code used to implement this stencil is shown in Appendix 6.4.

The five-point stencil used in Question 1 has an error in the order of $\mathcal{O}(n^2)$. The new nine-point stencil has an error in the order of $\mathcal{O}(n^4)$. This means that if the number of grid points is doubled the error will be reduced by a factor of 16 for the new stencil compared with a factor of 4 for the original stencil. This allows us to use a coarser grid while maintaining accuracy.

4 Question 4: The Gauss-Seidel "Red-Black" formulation

4.1 Implementation

The central idea behind the "red-black" enhancement to the Gauss-Seidel method is that the nodes can be divided into 'red' and 'black' nodes where all red nodes depend only upon black nodes and all black nodes depend only upon red nodes. When $\mathbf{u}$ is represented as a row vector rather than as an $n \times n$ matrix, the red nodes are those with even indices and the black nodes are those with odd indices. Therefore the grid resembles a chessboard with each row and each column alternating between red and black. When using a five-point stencil it is therefore clear that no node will depend upon any other nodes of the same colour.

The code used to implement the "red-black" solver is shown in Appendix 6.5.

4.2 Speed Comparison with SOR

The speed of the "red-black" solver is compared with that of the SOR solver from Question 2 in Table 2.

n	**Time Taken**	**(s)**
	SOR solver	**“red-black” solver**
3	0.0000	0.0000
11	0.1400	0.2520
21	1.5780	2.2800
31	6.4490	5.5610
41	12.6750	11.1020
51	21.7450	19.4920
61	34.2620	30.8350
71	53.2080	47.2230

Table 2: Speed comparison of the SOR solver and the “red-black” solver

It is interesting to note that “red-black” solver actually performs slower than the SOR solver for lower values of n although this reverses at around $n = 61$. This is probably due to the fact that this is a single core calculation and the key benefit of the “red-black” enhancement is that it is highly suitable to a parallel programming approach as the red and black nodes can be computed entirely separately.

5 Question 5: Further analysis

5.1 Speeding up the code

There are many methods by which we might speed up the code. The most obvious is to use a faster computer but as this is an expensive option it is worth examining others. Using compiled languages such as C, rather than an interpreted language such as Python can make a huge difference. In [3] Fanghor estimates that plain C is approximately 36 times faster than plain python and optimised C is twice as fast again.

However, for this improvement in speed a large price is paid in terms of ease of writing and debugging as well as readability. Therefore we might consider Python tools such as Weave and Cython. Weave allows the user to write inline C code into a Python program. In [3] Fanghor estimates that using a Python Weave, code can be executed at almost twice the speed of plain C and therefore almost as fast as optimised C. Cython in essentially Python with C data types and is a way of ’translating’ Python into C. Cython is very attractive from an ease of use perspective and according to [3], typed Cython is around 4 times faster than basic Python. By optimising a little this can be increased to 40 times.

Another way of speeding up execution would be to, as discussed in Section 4.2, implement a parallel programming approach using the “red-black” solver. Other, more efficient algorithms could also be employed such as the Multigrid Method discussed in Section 5.2.

5.2 Comparison with Multigrid Method

According to [2] an optimal SOR method will have an order of $\mathcal{O}(n^{3/2})$. Therefore for large matrix systems such a method will be very computationally intensive. There are two essential ideas behind the multigrid method:

1. As the SOR method is iterative, having a good estimate for the solution at the beginning will drastically reduce computing time.
2. The SOR method is very efficient in removing high frequency errors (usually within just a few iterations) but performs much more poorly in removing low frequency errors.

The multigrid method therefore takes a problem that has been discretised onto a fine mesh and maps it onto a coarse mesh. A solution is then quickly obtained for this coarse mesh using the SOR method as low frequency errors behave like high frequency errors on a coarse mesh. This solution is then projected back onto a fine mesh and is used as the initial estimate for the SOR solver. Of course, the SOR solver does not have to be used, the multigrid method can be employed with vanilla Gauss-Seidel or the Jacobi method.

As the name implies more than two meshes can be used, with the problem being mapped from the coarsest mesh onto successively finer ones.

This method is particularly suited to problems with very large grids as in these cases it will cut the number of iterations required to find a solution significantly. However, for small grids, the complexity of the algorithm (not to mention the code) may result in it actually taking longer than more traditional iterative solvers.

6 Appendix

6.1 Libraries

```python
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
```

6.2 Code to populate matrix A

```python
def matrix_setup(n):
    """
    This function sets up the matrices A and b using the first order
    central differences stencil to discretise Poisson's equation in 2D.

    """
    N = n**2 # Number of points
    h = 1./(n+1) # gridspacing
    A = np.zeros([N, N]) # initialise A

   #Diagonals
    lead_diag = np.diag(np.ones(N)*-4, 0)
    outer_diags = np.ones(N-1)
    for i in range(n-1, N-1, n):
        outer_diags[i] = 0
    outer_diags = np.diag(outer_diags, 1) + np.diag(outer_diags, -1)

    #Diagonals dependent on n
    n_diags = np.diag(np.ones(N-n), n) + np.diag(np.ones(N-n), -n)

    #Populate A matrix
    A += lead_diag + outer_diags + n_diags
    A = A/(h**2)

   #Populate the RHS b matrix
    b=np.zeros(N)
    b[(N-1)/2]=2
    return A,b # The matrix problem A.u = b can now be solved to give a solution
    to the PDE.
```

6.3 Implementing the Gauss-Seidel SOR solver

```python
def sor(omega=1.):
    """
    This function that will carry out the Successive Over-Relaxation
    iterations, as seen on p6 of the course notes, that will be used for SOR.
    """
u_old = u.copy()
for i in range(N):
    sigma1 = np.dot(A[i, 0:i], u[0:i])
    sigma2 = np.dot(A[i, i+1:-1], u[i+1:-1])
    u[i] = (omega / A[i,i]) * (b[i] - sigma1 - sigma2) + (1 - omega) * u_old[i]
du = np.sqrt(np.dot(u-u_old,u-u_old))
return du,u
```

```python
"""
Carry out k interations with omega = 1 (k = 10)
Use this to find the change in u between iterations in order to find
an optimum value for omega as shown on p88 Numerical Methods in Engineering With
Python 3 (Jaan Kiusalaas)

"""
for i in range(k+p):
    du_old = du_new
    du_new,u = sor()

omega_opt = 2.0/(1.0 + np.sqrt(1.0 - (float(du_new)/du_old)**(1.0/p)))
if printing == "ON":
    print("Optimum omega = {}".format(omega_opt))

"""
Now perform subsequent interations using the optimised omega

"""

for i in range(n_its):
    du_old = du_new
    du_new,u = sor(omega=omega_opt)
    if du_new < tol:
        break

if i == n_its:
    raise RuntimeError("SOR method has not converged")
```

6.4 Implementing a nine-point stencil

```python
def stencil2_setup(n):
"""
This function creates the A and b matrices fpr the higher order
central differences stencil.

"""
N = n**2 # Number of points
h = 1./(n+1) # gridspacing
A = np.zeros([N, N])# initialise A

#Diagonals
lead_diag = np.diag(np.ones(N)*-60, 0)

outer_diags1 = np.ones(N-1)*16
outer_diags1[n-1::n] = 0
outer_diags1 = np.diag(outer_diags1, 1) + np.diag(outer_diags1, -1)

outer_diags2 = np.ones(N-2)*-1
for i in range(n,N-2,n):
    outer_diags2[i-1]=0
    outer_diags2[i-2]=0
outer_diags2 = np.diag(outer_diags2, 2) + np.diag(outer_diags2, -2)

#Diagonals dependent on n
n_diags1 = np.diag(np.ones(N-n)*16, n) + np.diag(np.ones(N-n)*16, -n)
n_diags2 = np.diag(np.ones(N-2*n)*-1, 2*n) + np.diag(np.ones(N-2*n)*-1, -2*n)

#Populate the matrix A
A += lead_diag + outer_diags1 + outer_diags2 + n_diags1 + n_diags2
A = A/(12*h**2 )

#Populate the RHS b matrix
```

```
b=np.zeros(N)
b[(N-1)/2]=2.0 # Forcing term

return A,b # The matrix problem A.u = b can now be solved to give a
solution to the PDE.
```

6.5 Implementing the “Red-Black” solver

```
"""
Here the Red-Black solver will iterate over all of the 'red'
and then all of the 'black' nodes, employing a vanilla
Gauss-Seidel solver. When u is represented as a column vector
the 'red' nodes are those with even indices and the 'black' nodes
are those with odd indices.

"""

for i in range(1,n_its):
    u_old = u.copy()
    for i in range(0,N,2): #This gives the even i values, i.e. the
                          #indices for the red nodes

        sigma1 = np.dot(A[i, 0:i], u[0:i])
        sigma2 = np.dot(A[i, i+1:-1], u[i+1:-1])
        u[i] = (1. / A[i,i]) * (b[i] - sigma1 - sigma2)

    for i in range(1,N,2): #This gives the odd i values, i.e. the
                           #indices for the black nodes
        sigma1 = np.dot(A[i, 0:i], u[0:i])
        sigma2 = np.dot(A[i, i+1:-1], u[i+1:-1])
        u[i] = (1. / A[i,i]) * (b[i] - sigma1 - sigma2)

    du = np.sqrt(np.dot(u-u_old,u-u_old))

    if du < tol:
            break

if i == n_its:
raise RuntimeError("Red-Black method has not converged")
```

7 References

[1] Kiusalaas, J. (2013) *Numerical Methods in Engineering With Python 3*, 3rd ed. Cambridge: Cambridge University Press.

[2] Cox, S. (2014) *Advanced Computational Methods Notes for FEEG6002*, Southampton: University of Southampton.

[3] Fanghor. (2013) *A short introduction to the C Programming Language*, Southampton: University of Southampton.

[4] Press, W. (2007) *Numerical Recipes 3rd edition: The Art of Scientic Computing*, 3rd ed. Cambridge: Cambridge University Press.