AMath 585, Winter 2018

Assignment 5.

Jithin D. George, No. 1622555

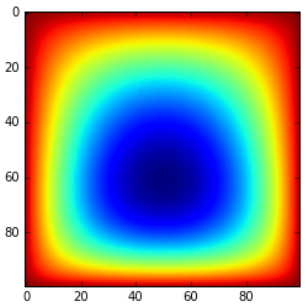
Due Friday, Feb. 16.

1. Write a code to solve Poisson's equation on the unit square with Dirichlet boundary conditions:

$$u_{xx} + u_{yy} = f(x, y), \quad 0 < x, y < 1$$
$$u(x, 0) = u(x, 1) = u(0, y) = u(1, y) = 1.$$

Take $f(x, y) = x^2 + y^2$, and demonstrate numerically that your code achieves second order accuracy. [Note: If you do not know an analytic solution to a problem, one way to check the code is to solve the problem on a fine grid and pretend that the result is the exact solution, then solve on coarser grids and compare your answers to the fine grid solution. However, you must be sure to compare solution values corresponding to the same points in the domain.]

Solution:




```
import numpy as np
import matplotlib.pyplot as plt
N = 12
a = -4*np.ones(N)
b = np.ones(N-1)
c = np.ones(N)
I = np.eye(N)
def tridiag(a, b, c, k1=-1, k2=0, k3=1):
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)
T = tridiag(b,a,b)
S = np.diag(b, -1)+ np.diag(b, 1)
A = np.kron(I,T)+np.kron(S,I )
def f(i,N):
    h = 1/(N+1)
```

```
    x = (i%(N+1))*h
    g = int(i/N ) if i%N ==0 else int(i/N )+1
    y = g*h
    x = (i - (g-1)*N)*h
    return x**2 + y**2
fvec = np.array([])
for i in range(1,N**2+1):
    fvec = np.append(fvec,f(i,N**2))
h = 1/(N+1)
i=0
for row in A:
    fvec[i]= fvec[i]*(h**2)+ sum(row)
    i+=1
fv = 6*(h**2)*fvec
x = np.linalg.solve(A, fvec)
xn = np.reshape(x,(N,N))
plt.imshow(xn)
print(np.linalg.norm(xn-x120[::10,::10], 2))
```

The infinity norm is used here. The solution at N = 120 is used as the 'accurate' solution.

N	Error
6	0.004582454
12	0.001215613
24	0.000276484

The order seems to be $O(h^2)$.

2. Now use the 9-point formula with the correction term described in Sec. 3.5 to solve the same problem as in the previous exercise. Again take $f(x, y) = x^2+y^2$, and numerically test the order of accuracy of your code by solving on a fine grid, pretending that is the exact solution, and comparing coarser grid approximations to the corresponding values of the fine grid solution. Show what order of accuracy you achieve and try to explain why.

Solution:

```
import numpy as np
import matplotlib.pyplot as plt
N = 12
a = -20*np.ones(N)
b = 4*np.ones(N-1)
c = 4*np.ones(N)
d= np.ones(N-1)
I = np.eye(N)
def tridiag(a, b, c, k1=-1, k2=0, k3=1):
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)
T = tridiag(b,a,b)
```

```
R = tridiag(d,c,d)
S = tridiag(d,np.zeros(N),d)
A = np.kron(I,T)+np.kron(S,R)
def f(i,N):
    h = 1/(N+1)
    x = (i%(N+1))*h
    g = int(i/N ) if i%N ==0 else int(i/N )+1
    y = g*h
    x = (i - (g-1)*N)*h
    return x**2 + y**2
fvec = np.array([])
for i in range(1,N**2+1):
    fvec = np.append(fvec,f(i,N**2)) + (h**2)/4
h = 1/(N+1)
i=0

for row in A:
    fvec[i]= fvec[i]*6*(h**2)+ sum(row)
    i+=1
fv = 6*(h**2)*fvec
x = np.linalg.solve(A, fvec)
xn = np.reshape(x,(N,N))
plt.imshow(xn)
print(np.linalg.norm(xn-x120[::10,::10], 2))
```

The infinity norm is used here. The solution at N = 120 is used as the 'accurate' solution.

N	Error
6	5.10230545e-05
12	3.17394091e-06
24	1.99308866e-07

The order seems to be $O(h^4)$.

3. We have discussed using finite element methods to solve elliptic PDE's such as

$$\triangle u = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial\Omega,$$

with *homogeneous* Dirichlet boundary conditions. How could you modify the procedure to solve the *inhomogeneous* Dirichlet problem:

$$\triangle u = f \quad \text{in } \Omega, \quad u = g \quad \text{on } \partial\Omega,$$

where g is some given function? Derive the equations that you would need to solve to compute, say, a continuous piecewise bilinear approximation for this problem when Ω is the unit square $(0,1) \times (0,1)$.

Solution:

The weak formulation of our PDE is given by

$$\langle Lu, v\rangle = \langle f, v\rangle$$

The way we solved for u is by representing u as

$$\hat{u} = \sum_{j=0}^{N} c_j \psi_j$$

where the ψ are bilinear piecewise functions. Since u has inhomogeneous boundaries, we need basis functions for the boundaries.

Then, we solve a system of equations

$$\langle Lu, \psi_j\rangle = \langle f, \psi_j\rangle$$

$$\langle f, \psi_j\rangle = \int\int_{\Omega} f(x,y)\psi_j dxdy$$

$$\langle Lu, \psi_j\rangle = \int\int_{\Omega} (\hat{u}_{xx}\psi_j + \hat{u}_{yy}\psi_j)dxdy$$

$$= -\int\int_{\Omega} (\hat{u}_x\psi_j + \hat{u}_y\psi_j)dxdy + \int_{d\Omega} (\hat{u}_n\psi_j)dr$$

$$= -\sum_{k=0}^{N} c_k \int\int_{\Omega} (\psi_{kx}\psi_j + \psi_{ky}\psi_j)dxdy + \sum_{k=0}^{N} c_k \int_{d\Omega} (\psi_{kn}\psi_j)dr$$

The second term doesn't vanish because we have non-zero boundaries. Evaluating the integrals, we get a system of equations which we can solve to find the coefficients.

The corner boundary basis functions are different from the interior boundary basis functions. The interior ones are rectangular like the ones below

At the top

$$\psi(x,y) = \left\{ \begin{array}{ll} \frac{(x-x_{i-1})(y-y_1)}{(x_i-x_{i-1})(0-y_1)}, & \text{for } [x_{i-1}, x_i] \times [0, y_1] \\ \frac{(x-x_{i+1})(y-y_1)}{(x_i-x_{i+})(0-y_1)}, & \text{for } [x_i, x_{i+1}] \times [0, y_1] \\ 0, & \text{elsewhere} \end{array} \right\}$$

At the bottom

$$\psi(x,y) = \left\{ \begin{array}{ll} \frac{(x-x_{i-1})(y-y_{n-1})}{(x_i-x_{i-1})(1-y_{n-1})}, & \text{for } [x_{i-1}, x_i] \times [y_{n-1}, 1] \\ \frac{(x-x_{i+1})(y-y_{n-1})}{(x_i-x_{i+1})(1-y_{n-1})}, & \text{for } [x_i, x_{i+1}] \times [y_{n-1}, 1] \\ 0, & \text{elsewhere} \end{array} \right\}$$

On the left,

$$\psi(x,y) = \left\{ \begin{array}{ll} \frac{(x-x_1)(y-y_{i-1})}{(-x_1)(y_i-y_{i-1})}, & \text{for } [0, x_1] \times [y_{i-1}, y_i] \\ \frac{(x-x_1)(y-y_{i+1})}{(0-x_1)(y_i-y_{i+1})}, & \text{for } [0, x_1] \times [y_i, y_{i+1}] \\ 0, & \text{elsewhere} \end{array} \right\}$$

On the right,

$$\psi(x,y) = \left\{ \begin{array}{ll} \frac{(x-x_{n-1})(y-y_{i-1})}{(1-x_{n-1})(y_i-y_{i-1})}, & \text{for } [x_{n-1},1]\times[y_{i-1},y_i] \\ \frac{(x-x_{n-1})(y-y_{i+1})}{(1-x_{n-1})(y_i-y_{i+1})}, & \text{for } [x_{n-1},1]\times[y_i,y_{i+1}] \\ 0, & \text{elsewhere} \end{array} \right\}$$

A corner boundary will be non-zero only on half the rectangle(because the other half is outside our domain). For example, in the top left corner,

$$\psi(x,y) = \left\{ \begin{array}{ll} \frac{(x-x_{i+1})(y-y_1)}{(x_i-x_{i+})(0-y_1)}, & \text{for } [x_i,x_{i+1}]\times[0,y_1] \\ 0, & \text{elsewhere} \end{array} \right\}$$

Similarly, we can write down boundary basis functions for the other 3 corners.

4. The key to efficiency in the `chebfun` package, which you used in a previous homework exercise, is the ability to rapidly translate between the values of a function at the Chebyshev points, $\cos(\pi j/n)$, $j = 0,\ldots,n$, and the coefficients $a_0,\ldots,a_n$, in a Chebyshev expansion of the function's nth-degree polynomial interpolant: $p(x) = \sum_{j=0}^{n} a_j T_j(x)$, where $T_j(x) = \cos(j \arccos(x))$ is the jth degree Chebyshev polynomial. Knowing the coefficients $a_0,\ldots,a_n$, one can evaluate p at the Chebyshev points by evaluating the sums

$$p(\cos(k\pi/n)) = \sum_{j=0}^{n} a_j \cos(jk\pi/n), \quad k = 0,\ldots,n. \tag{1}$$

These sums are much like the real part of the sums in the FFT,

$$F_k = \sum_{j=0}^{n-1} e^{2\pi ijk/n} f_j, \quad k = 0,\ldots,n-1,$$

but the argument of the cosine differs by a factor of 2 from the values that would make them equal. Explain how the FFT or a closely related procedure could be used to evaluate the sums in (1). To go in the other direction, and efficiently determine the coefficients $a_0,\ldots,a_n$ from the function values $f(\cos(k\pi/n))$, what method would you use?

Solution:

$$2 * p(\cos(k\pi/n)) = 2\sum_{j=0}^{n} a_j \cos(jk\pi/n)$$

$$= \sum_{j=0}^{n} a_j \cos(jk\pi/n) + \sum_{j=0}^{n} a_j \cos(jk\pi/n)$$

$$= \sum_{j=0}^{n} a_j \cos(jk\pi/n) + \sum_{j=0}^{n} a_j \cos((n+j)k\pi/n)$$

$$= \sum_{j=0}^{2n-1} a_j \cos(jk\pi/n) + a_0 + a_n \cos(k\pi)$$

$$= \sum_{j=0}^{2n-1} a_j \cos(jk\pi/n) + a_0 + a_n(-1)^k$$

$$= Re(\sum_{j=0}^{2n-1} a_j e^{jk\pi/n}) + a_0 + a_n(-1)^k$$

Thus,

$$p(\cos(k\pi/n)) = \frac{1}{2}(Re(\sum_{j=0}^{2n-1} a_j e^{jk\pi/n}) + a_0 + a_n(-1)^k)$$

The first term can be calculated using the FFT

Now, the inverse problem - finding the Chebyshev coefficients.

$$p(k) = \frac{1}{2}(Re(\sum_{j=0}^{2n-1} a_j e^{jk\pi/n}) + a_0 + a_n(-1)^k)$$

$$\sum_{k=0}^{n} 2p(k) = \sum_{k=0}^{n} Re(\sum_{j=0}^{2n-1} a_j e^{jk\pi/n}) + \sum_{k=0}^{n} a_0 + \sum_{k=0}^{n} a_n(-1)^k$$

$$\sum_{k=0}^{n} 2p(k) = \sum_{k=0}^{n} Re(\sum_{j=0}^{2n-1} a_j e^{jk\pi/n}) + p(0) + p(n)$$

$$p(0) + \sum_{k=1}^{n-1} 2p(k) + p(n) = \sum_{k=0}^{n} \sum_{j=0}^{2n-1} a_j Re(e^{jk\pi/n})$$

$$p(0) + \sum_{k=1}^{n-1} p(k) + \sum_{k=1}^{n-1} p(k) + p(n) = \sum_{k=0}^{n} \sum_{j=0}^{2n-1} a_j Re(e^{jk\pi/n})$$

Thus, we can see some intuition behind arranging the p values in a particular order to invert the Fourier transform on the right hand side. This intuition is mostly geometric as shown in the last page of this homework.

$$p = \begin{bmatrix} p(0) \\ p(1) \\ p(2) \\ \vdots \\ p(N-1) \\ p(N) \\ p(N-1) \\ \vdots \\ p(2) \\ p(1) \end{bmatrix} \tag{2}$$

Arranging our p values this way in the order of a Fourier Transform terms allows us to take an inverse Fourier Transform.

$$IFFT \begin{bmatrix} p(0) \\ p(1) \\ p(2) \\ \vdots \\ p(N-1) \\ p(N) \\ p(N-1) \\ \vdots \\ p(2) \\ p(1) \end{bmatrix} = 2N \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ \vdots \\ a(N-1) \\ a(N) \\ a(N-1) \\ \vdots \\ a(2) \\ a(1) \end{bmatrix}$$

Thus, taking the ifft of the vector (p(0), p(1),p(2), ...p(n-1), p(n), p(n-1),p(n-2),..., p(2),p(1)) can get you back a vector which contains the coefficients a_n. Rearranging the IFFT values should return the a_n. Writing up code to do this in this way verifies this.

```
from numpy import *
from numpy.fft import fft,ifft

def cheb(y):
    N = len(y) - 1
    yt = real(fft(r_[y,y[-2:0:-1]]))
    yt = yt/(2*N)
    yt = r_[yt[0],yt[1:N]+yt[-1:N:-1],yt[N]]

    return yt
```

```
cheb([1,2,3,4]) = [ 2.5 , -1.33333333,  0. , -0.16666667]
```

which are the right Chebyshev coefficients ($\sum a_n = 1$, $\sum a_n(-1)^n = 4$)