M345SC 2019 Homework 3 Solutions

Part 1. (10 pts)

Consider the following model for the evolution of nonlinear waves:

$$\frac{\partial g}{\partial t} + \beta \mathcal{N}(g) = \alpha \frac{\partial^2 g}{\partial x^2} + g,$$

where $g(x,t)$ is a complex periodic function with period, $L = 100$, α and β are complex model parameters, and $\mathcal{N}(g)$ is a nonlinear function. A mostly-complete function (*nwave*) for computing solutions to this model has been provided in *p1_template.py*. Starting from a provided initial condition, the solution is marched forward in time using *odeint* which calls *RHS*. *RHS* is incomplete.

1) Complete *RHS* so that it uses discrete Fourier transforms to compute $\frac{\partial^2 g}{\partial x^2}$. Code has been provided which you can use to display the computed solution.

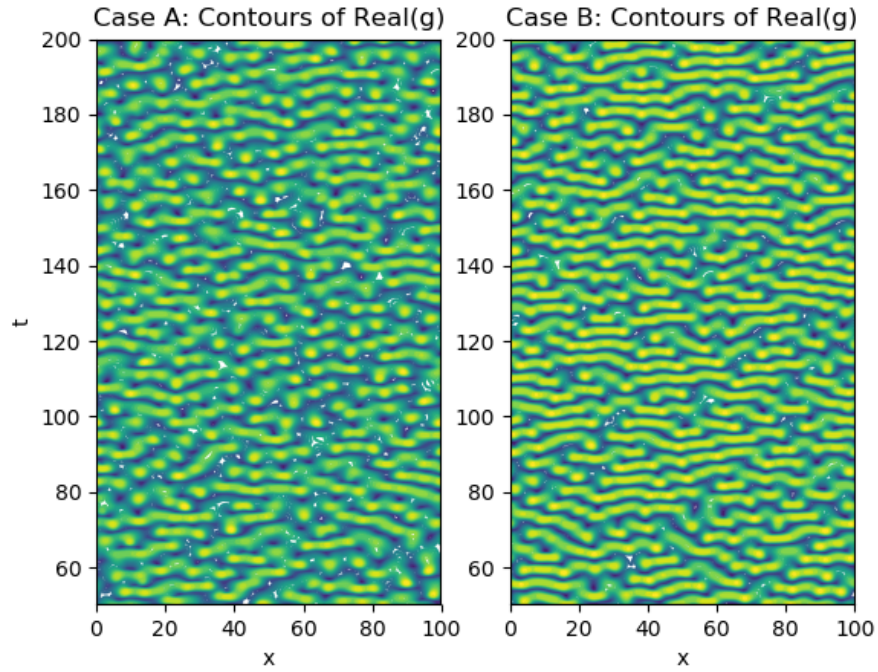
ANS: Let $\hat{g}_n$ correspond to the *nth* Fourier coefficient in the Fourier expansion of $g(x)$. Then, the Fourier coefficients of the second derivative of g can be expressed as, $-k_n^2 \hat{g}_n$ where $k_n = 2\pi n/L$. So, d^2g/dx^2 can be approximated by 1) computing the DFT of g, 2) computing the product of the DFT with k^2, and then 3) computing the inverse DFT of this product. The relevant portion of *nwave* is below:

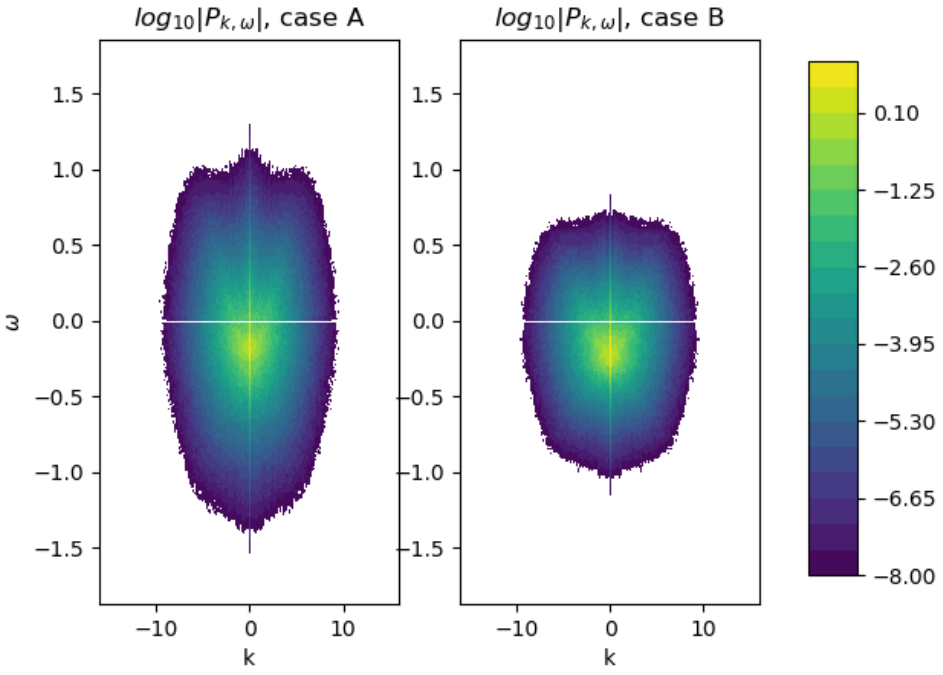
```
#generate grid
L = 100
x = np.linspace(0,L,Nx+1)
x = x[:-1]

#generate wavenumbers
k = np.zeros_like(x)
if Nx%2==0:
    k[:Nx//2]=np.arange(0,Nx//2)
    k[Nx//2:]=np.arange(-Nx//2,0)
else:
    k[:Nx//2+1]=np.arange(0,Nx//2+1)
    k[Nx//2+1:]=np.arange(-Nx//2+1,0)
k = k * 2*np.pi/L

k2m = -(k**2)


def RHS(f,t,alpha,beta):
    """Computes dg/dt for model eqn.,
    f[:N] = Real(g), f[N:] = Imag(g)
    Called by odeint below
    """
    g = f[:Nx]+1j*f[Nx:]
    #add code here
    d2g = fft.ifft(k2m*fft.fft(g))

    #-----------
    dgdt = alpha*d2g + g -beta*g*g*g.conj()
    df = np.zeros(2*Nx)
    df[:Nx] = dgdt.real
    df[Nx:] = dgdt.imag
    return df
```

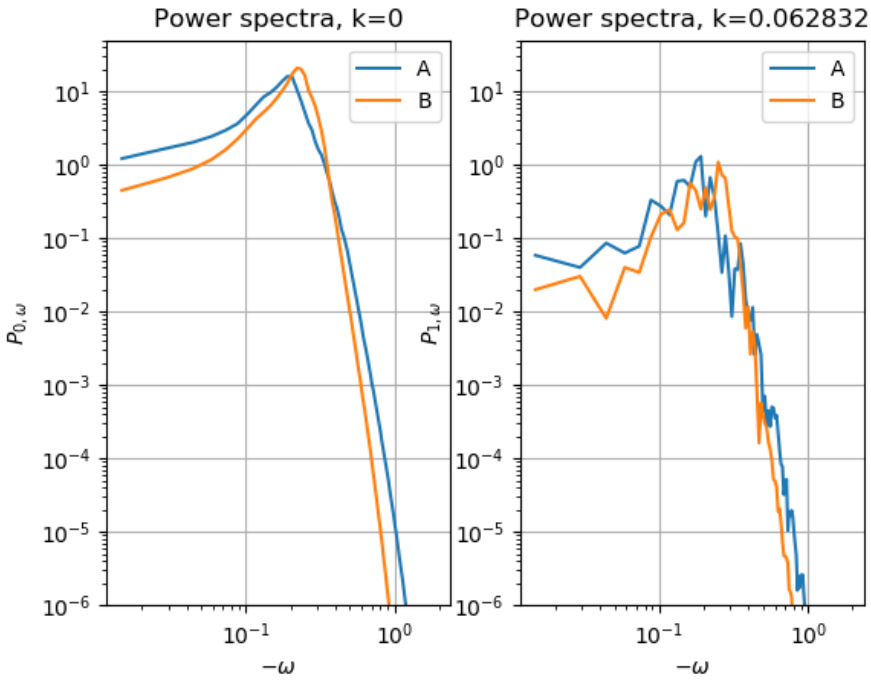
2) Analyze and compare the simulation results for the following two cases, A: $\alpha = 1 - 2i, \beta = 1 + 2i$ and B: $\alpha = 1 - i, \beta = 1 + 2i$. There will be an initial transient as the solution adjusts away from the initial condition which should be discarded (say, for t<50). The input parameters Nt,Nx, and T may need to be varied as you develop your analysis. You should focus on the most energetic wavenumbers and frequencies and also carefully consider if/to what degree each case is chaotic. Your analysis should be structured around a few figures which are generated by code placed in the function, *analyze*. Save these figures and submit them with your codes. Add the corresponding discussion to the docstring of the function.

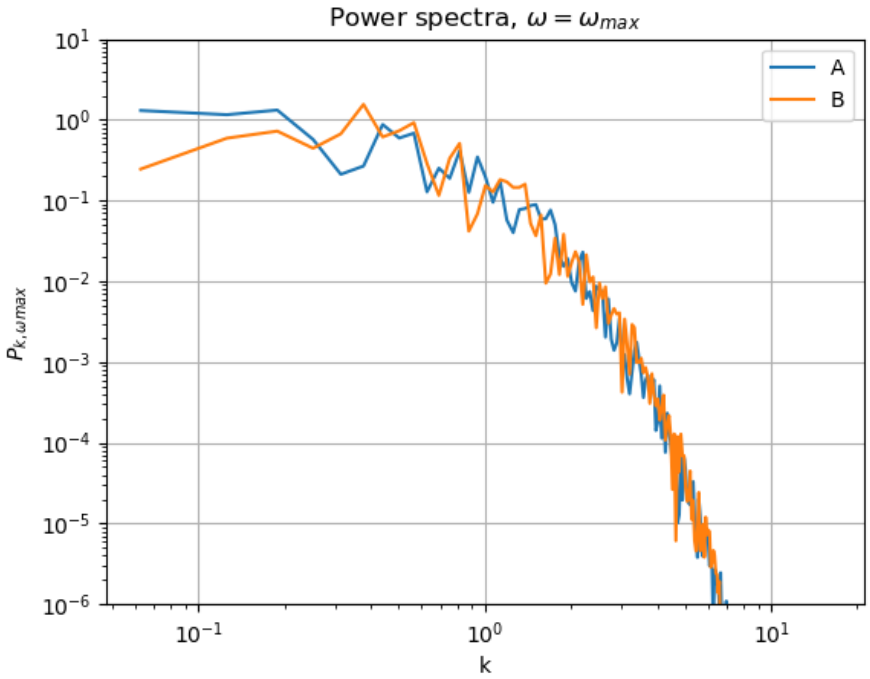
ANS:

Code for this question can be found near the bottom of this page. The first figure below shows contour plots for $Re(g)$ for the two cases (simulations were run with Nx=512 which is sufficiently large for accurate solutions). We see that the solutions are broadly similar but not identical. The task then is to determine how similar or distinct the two cases are using data analysis tools covered in class. After discarding the transient, Welch's method is applied to fine the power spectral densities (PSDs) for time series at each x-point. Discrete Fourier transforms are then used to transform the PSD at each frequency to to the wavenumber domain. The results are first displayed as contour plots in the 2nd figure. The most energetic waves are found near $k = 0$ and a negative non-zero frequency for each case. Additionally, we see that the energy in case A tends to span a broader range of frequencies while the two cases span similar ranges of wavenumbers. More detailed comparisons can be made using the 3rd and 4th figures. In the 3rd figure, power spectra are shown for the two lowest (and most energetic wavenumbers) for these cases. We see that case B peaks at a higher energy and at a slightly higher frequency, while, case A is more energetic at lower frequencies at, as mentioned above, spans a broader range of frequencies. There is little meaningful difference between the the wavenumber spectra for the two cases at their peak frequencies (4th figure). Results for $k = 0$ are not shown but this is the most energetic wavenumber for both cases, and case B is substantially more energetic than A at $k = 0$

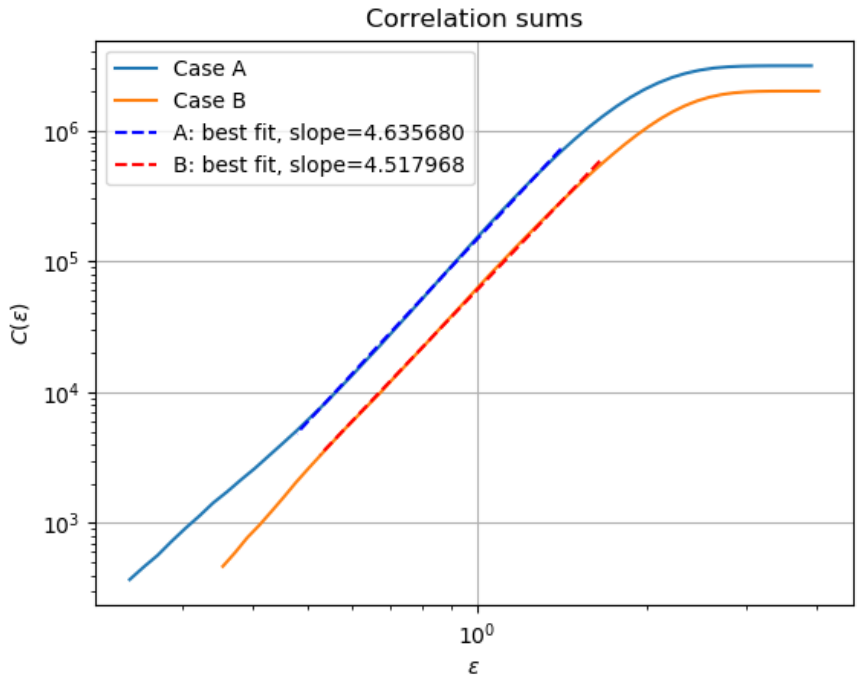








Finally, we consider if and to what degree the two cases are chaotic. This is accomplished by estimating the correlation dimensions for the two cases. Vectors are constructed using time-delays, and the results are shown in the figure below. Both estimated dimensions are well above two, so it is clear that both cases are chaotic. There is a small difference in the estimates for the two dimensions, but rather than draw a distinction, it is better to conclude that the dimensions of the underlying attractors for the two cases are very close.

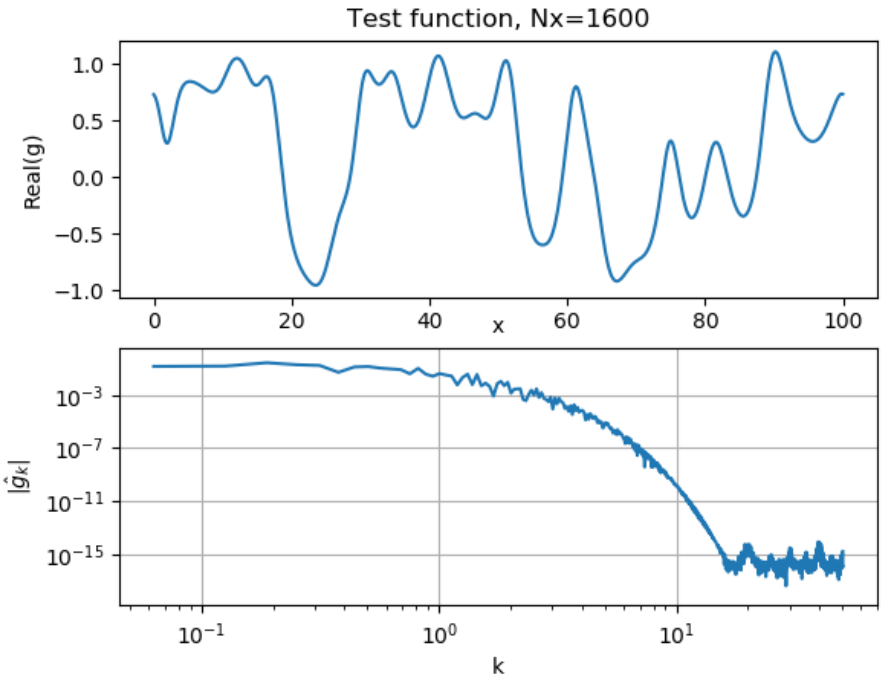


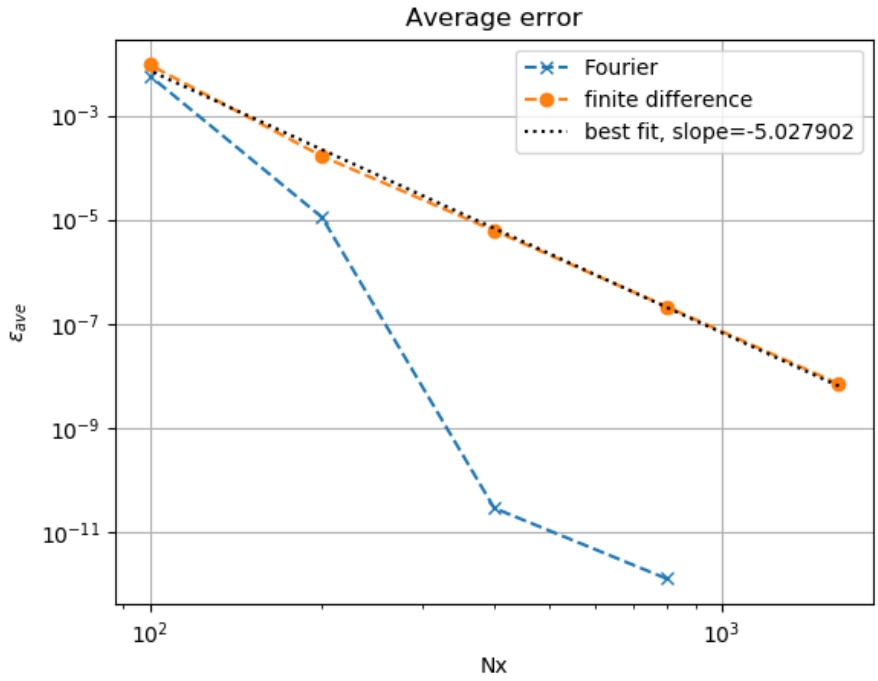
3) Consider whether a compact finite difference method would be superior to Fourier differentiation for nonlinear waves such as those generated by this model. Critically compare DFT-based calculation of $\frac{\partial g}{\partial x}$ with a tridiagonal finite-difference (fd) method with coefficients,

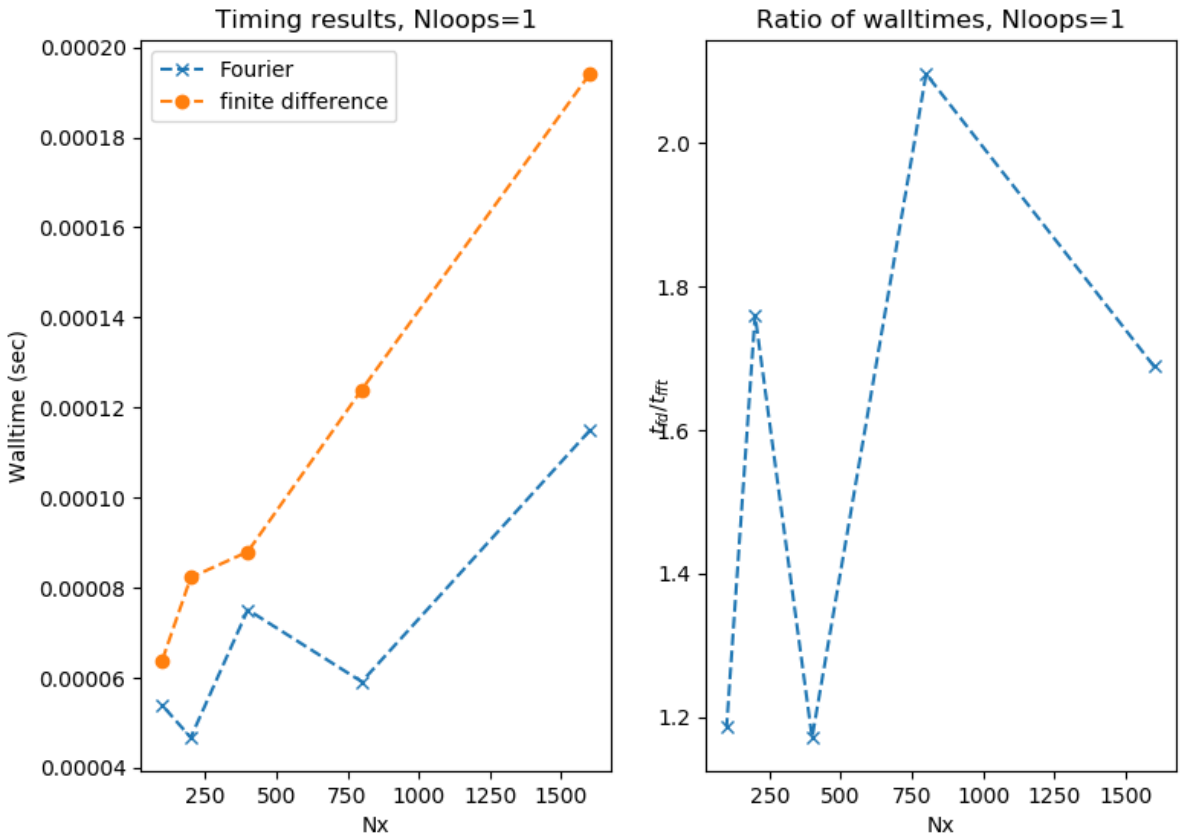
$$\alpha = 3/8, \beta = 0, a = 25/16, b = 1/5, c = -1/80.$$

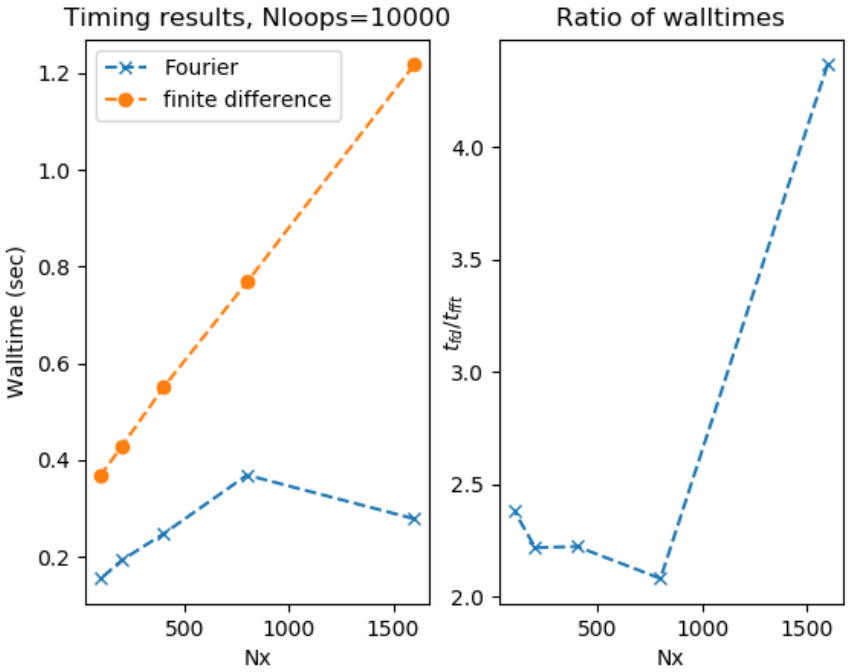
(see the general form of implicit fd schemes introduced in lecture 16, **these coefficients are unrelated to the parameters in the model PDE above**) At the boundaries ($x = 0, x = L - h$), use the one-sided 4th order scheme from lecture. Take simulation results for case B at t=100 as "test waves" for determining which of the two methods is superior. You should generate a few figures to support your conclusion and add a discussion to the docstring of *wavediff* which carefully describes both your reasoning and the contents of your figures. The code in *wavediff* should generate these figures, and you should also save the figures and submit them with your codes.

ANS: Code for this question is at the bottom of this page. Comparison of these two methods requires consideration of accuracy and speed. For accuracy, we take the result of Fourier differentiation with Nx=1600 as a reference solution. Examining the Fourier coefficients of this test function (1st figure below), we see that they decay to around 1e-15 which is close to the best we can hope for. While the results of Fourier differentiation will still have error (around 1e-12) but this is sufficiently small for us to use the result as a reference. This is apparent in the 2nd figure where the average differences between the reference solution and computed derivatives with smaller Nx are shown (Nx=100,200,400,800) We see that the Fourier method is more accurate over the entire range of Nx, and converges much more rapidly than the finite difference method. Interestingly, the error for the finite difference method appears to behave as h^5 even though 8th order accuracy is expected away from the boundaries – this illustrates the global nature of compact schemes. We see that the Fourier method is more accurate for a given Nx, but is it also faster? The answer is yes based on the timing results shown in the 4th figure below. While we expect the running time for the finite difference method to be O(Nx) compared to O(Nx log Nx) for the Fourier, in practice we find that the Fourier is faster, and the time grows more slowly than for the finite difference scheme. This is probably due to low-level optimizations in the numpy FFT routine. Ultimately, these results indicate that Fourier differentiation is preferable to this finite difference method for this class of problems. Note that timing results depend strongly on hardware. Running on my laptop, results suggest that the FD method may be preferable around Nx=50 if a 10-20% error is acceptable. Strangely, the first call with FFT always seems to be considerably slower than subsequent calls suggesting that Python is efficiently storing data in cache.


Test function, Nx=1600
Real(g)
x
$|\hat{g}_k|$
k


Average error
Fourier
finite difference
best fit, slope=-5.027902
ϵ_{ave}
Nx





Part 2. (10 pts)

2.1 Here, you will consider a model for the spread of small perturbations to the number of S, I, and V cells in a network. The model equations are:

$$\frac{dS_i}{dt} = \alpha I_i - (\gamma+\kappa)S_i + \sum_{j=0}^{N-1}(F_{ij}S_j - F_{ji}S_i)$$
$$\frac{dI_i}{dt} = \theta S_i - (\kappa+\alpha)I_i + \sum_{j=0}^{N-1}(F_{ij}I_j - F_{ji}I_i)$$
$$\frac{dV_i}{dt} = \kappa V_i - \theta S_i + \sum_{j=0}^{N-1}(F_{ij}V_j - F_{ji}V_i)$$

Now, θ is a constant parameter like α, γ, κ, and τ. The *flux matrix*, F_{ij}, is defined as before, $F_{ij} = \tau\frac{q_i A_{ij}}{\sum_{k=0}^{N-1} q_k A_{kj}}$ where A_{ij} is the adjacency matrix of the network (which is unweighted and undirected), q_i is the degree of node i. You may assume that each node has at least one link to another node, and with this modified model, S, I, and V may take on positive or negative values.

i. We are interested in the growth of perturbations, and we define a "perturbation energy" as, $e(t) = \sum_{i=0}^{N-1}\left(S_i^2(t) + I_i^2(t) + V_i^2(t)\right)$. Complete *growth1* in *p2_dev.py* so that the maximum growth, $e(t=T)/e(t=0)$, is computed with the model parameters, T, and a networkx graph, G, provided as input. The function should return both the growth and the initial condition that generates the growth. The initial condition should be returned as a 3N-element array. See the function documentation for further details.

ANS: The model equations can be written in the form:

$$d\mathbf{y}/dt = M\mathbf{y},$$

where $\mathbf{y}$ is a 3N-element column vector and M is a 3N x 3N matrix. The solution to these equations at time T is,

$$\mathbf{y}(T) = O(T)\mathbf{y}_0$$

where $\mathbf{y_0}$ corresponds to the initial conditions and $O(T) = exp(MT)$. Noting that $e = \mathbf{y}^T\mathbf{y}$, we can find the maximum growth of the energy by computing the SVD of O, $O = USV^T$. The maximum growth is then the square of the largest singular value contained in S, and the initial condition that generates this is the corresponding row of V^T. The code is below:

```
def growth1(G,params=(0.02,6,1,0.1,0.1),T=6):
    """
    Question 2.1
    Find maximum possible growth, G=e(t=T)/e(t=0) and corresponding initial
    condition.

    Input:
    G: Networkx graph
    params: contains model parameters, see code below.
    T: time for energy growth

    Output:
    G: Maximum growth
    y: 3N-element array containing computed initial condition where N is the
    number of nodes in G and y[:N], y[N:2*N], y[2*N:] correspond to S,I,V

    Discussion: Add discussion here
    """
    a,theta,g,k,tau=params
    N = G.number_of_nodes()

    #Construct flux matrix (use/modify as needed)
    Q = [t[1] for t in G.degree()]

    Pden = np.zeros(N)
    Pden_total = np.zeros(N)
    for j in G.nodes():
        for m in G.adj[j].keys():
            Pden[j] += Q[m]
    Pden = 1/Pden
    Q = tau*np.array(Q)
    F = nx.adjacency_matrix(G).toarray()
    F = F*np.outer(Q,Pden)
    #-----------------------------------
    G=0
    y = np.zeros(3*N)

    #Add code here
```

```
    #Set model parameters
    gktm = -(g+k+tau)
    aktm = -(a+k+tau)
    ktm = (k-tau)

    #-------Set up matrix for RHS of equations--------
    M = np.zeros((3*N,3*N))
    I = np.eye(N)

    #S equations
    M[:N,:N] = gktm*I+F
    M[:N,N:2*N] = a*I

    #I equations
    M[N:2*N,:N] = theta*I
    M[N:2*N,N:2*N] = aktm*I+F

    #R equations
    M[2*N:,:N] = -theta*I
    M[2*N:,2*N:] = ktm*I+F

    #Find maximum growth and initial condition
    O = expm(M*T)
    U,S,VT = np.linalg.svd(O)
    G = S.max()**2
    y = VT[0,:]

    return G,y
```

ii. Now, complete *growth2* so that it finds the maximum growth of $\left[\sum_{i=0}^{N-1} I_i^2(t=T)\right]/e(t=0)$. Aside from this change, the functionality should be the same as in *growth1*.

ANS: This question requires one line of code to be changed from *growth1*. Only rows N+1 to 2N should be included in the SVD calculation:

```
U,S,VT = np.linalg.svd(O[N:2*N,:])
```

iii. Complete *growth3* so that it finds the maximum growth of $\left[\sum_{i=0}^{N-1} (S_i(t=T)V_i(t=T))\right]/e(t=0)$. You do not need to return the corresponding initial condition. Aside from these changes, the functionality should be the same as in *growth1*.

ANS: First, re-write the general solution from i. as:

$$\mathbf{s}(T) = O_1\mathbf{y_0}$$
$$\mathbf{v}(T) = O_3\mathbf{y_0}$$

where $\mathbf{s}(T)$ and $\mathbf{v}(T)$ correspond to the spreader and vulnerable nodes, respectively. Additionally, O_1 corresponds to the first N rows of O while O_3 corresponds to rows 2N+1 to 3N. Then, noting that,

$$\left[\sum_{i=0}^{N-1} (S_i(t=T)V_i(t=T))\right] = 1/2\left(O_1^TO_3 + O_3^TO_1\right)e(t=0),$$

it follows that that the maximum of $\left[\sum_{i=0}^{N-1} (S_i(t=T)V_i(t=T))\right]/e(t=0)$ will correspond to the largest positive eigenvalue of, $1/2\left(O_1^TO_3 + O_3^TO_1\right)$. Here, we have used the fact that the matrix, $O_1^TO_3 + O_3^TO_1$, is symmetric and can be orthogonally diagonalized. Ultimately, this requires five lines of code after the matrix exponential is computed:

```
    #Find maximum growth and initial condition
    O = expm(M*T)
    O1 = O[:N,:]
    O3 = O[2*N:,:]
    B = 0.5*(np.dot(O1.T,O3) + np.dot(O3.T,O1))
    l,v = np.linalg.eig(B)
    G = np.max(l.real)
```

2. You are now tasked with identifying the most "important" network nodes based on measurements at one time of I_i for M different organisms in the early stages of infection (each with N-node networks). The data is provided as an N x M matrix, and the aim is to construct an array, $\hat{I}_i$, which best approximates the total variance of I_i in the dataset. Here, the variance corresponds to

variations across the N nodes in a network, not variations in time. Complete the function *Inew* so that it computes and returns $\hat{I}_i$ as an N-element array (or list).

ANS: Apply PCA closely following the example from lab 8:

```
N,M = D.shape

#Substract mean
D2 = D - np.outer(np.ones((N,1)),D.mean(axis=0))

#Compute transformation matrix using PCA
U,S,VT = np.linalg.svd(D2.T)

#Computed "best" approximation to variance
I = np.dot(U[:,0],D2.T)
```

The *analyze* and *wavediff* functions from part 1:

```
def analyze():
  """
  Question 1.2
  Add input/output as needed

  Discussion: Add discussion here
  """

  Nx = 512
  L = 100
  T=100
  Nt=3201
  x = np.linspace(0,L,Nx+1)
  x = x[:-1]

  t = np.linspace(0,T,Nt)

  #Case A
  alpha = 1-2j
  beta = 1+2j

  np.random.seed(1)
  print("Simulating case A...")
  gA = nwave(alpha,beta,Nx,Nt,T)
  print("... done")

  #Case B
  alpha = 1-1j
  np.random.seed(1)
  print("Simulating case B...")
  gB = nwave(alpha,beta,Nx,Nt,T)
  print("... done")

  # Discard transients
  dt = t[1]-t[0]
  i_transient = int(50//dt)+1
  gA = gA[i_transient:,:]
  gB = gB[i_transient:,:]
  t = t[i_transient:]

  #Make contour plots
  gmax = max(np.abs(gA).max(),np.abs(gB).max())

  fig,axes = plt.subplots(1,2)
  #plt.figure()
  axes[0].contour(x,t[:600],gA[:600,:].real,np.linspace(-gmax,gmax,21))
  axes[0].set_xlabel('x')
  axes[0].set_ylabel('t')
  axes[0].set_title('Case A: Contours of Real(g)')

  axes[1].contour(x,t[:600],gB[:600,:].real,np.linspace(-gmax,gmax,21))
  axes[1].set_xlabel('x')
```

```
axes[1].set_title('Case B: Contours of Real(g)')


#Compute spectra
k = np.zeros_like(x)
if Nx%2==0:
    k[:Nx//2]=np.arange(0,Nx//2)
    k[Nx//2:]=np.arange(-Nx//2,0)
else:
    k[:Nx//2+1]=np.arange(0,Nx//2+1)
    k[Nx//2+1:]=np.arange(-Nx//2+1,0)
k = k*2*np.pi/L

wA,PxxA = welch((gA),axis=0)
wB,PxxB = welch((gB),axis=0)

wA = wA*t.size/T
wB = wB*t.size/T

sA = np.fft.fft2(PxxA,axes=(1,))/Nx
sB = np.fft.fft2(PxxB,axes=(1,))/Nx

fA = np.log10(np.abs(sA))
fB = np.log10(np.abs(sB))

print("computed spectra")

#Display spectra
fig,axes = plt.subplots(1,2)
axes[0].contourf(k,wA,fA,linspace(-8,1,21))
axes[0].set_title(r'$log_{10}|P_{k,\omega}|$, case A')
axes[0].set_xlabel('k')
axes[0].set_ylabel(r'$\omega$')

im = axes[1].contourf(k,wB,fB,linspace(-8,1,21))
axes[1].set_title(r'$log_{10}|P_{k,\omega}|$, case B')
axes[1].set_xlabel('k')

fig.subplots_adjust(right=0.8)
cbar = fig.add_axes([0.85,0.15,0.05,0.7])
fig.colorbar(im,cax=cbar)

fig,(ax1,ax2)=plt.subplots(1,2)
ax1.loglog(-wA,np.abs(sA[:,0]),label='A')
ax1.loglog(-wB,np.abs(sB[:,0]),label='B')
ax1.set_xlabel(r'$-\omega$')
ax1.set_ylabel(r'$P_{0,\omega}$')
ax1.legend()
ax1.set_title('Power spectra, k=0')
ax1.grid()
ax1.set_ylim([1e-6,5e1])

ax2.loglog(-wA,np.abs(sA[:,1]),label='A')
ax2.loglog(-wB,np.abs(sB[:,1]),label='B')
ax2.set_xlabel(r'$-\omega$')
ax2.set_ylabel(r'$P_{1,\omega}$')
ax2.legend()
ax2.set_title('Power spectra, k=%f'%(k[1]))
ax2.grid()
ax2.set_ylim([1e-6,5e1])

indA = np.argmax(np.abs(sA[:,0]))
indB = np.argmax(np.abs(sB[:,0]))

plt.figure()
plt.loglog(k,np.abs(sA[indA,:]),label='A')
plt.loglog(k,np.abs(sB[indB,:]),label='B')
plt.xlabel('k')
plt.ylabel(r'$P_{k,\omega max}$')
plt.title(r'Power spectra, $\omega=\omega_{max} $')
plt.legend()
plt.grid()
```

```
    #Compute correlation sums---------
    vA = gA[:,1].real
    dn=100
    vecA = np.vstack([vA[:-dn*5],vA[dn:-dn*4],vA[2*dn:-dn*3],vA[3*dn:-dn*2],vA[4*dn:-dn],vA[5*dn:]])
    vecA = vecA.T

    DA = pdist(vecA)
    print("computed distances")


    e1=DA.min()*4
    e2=DA.max()*1

    Neps=50
    earrayA = np.logspace(np.log10(e2),np.log10(e1),Neps)
    CA = np.zeros_like(earrayA)

    for i,eps in enumerate(earrayA):
        DA = DA[DA<eps]
        CA[i] = DA.size

    vB = gB[:,1].real
    dn=200
    vecB = np.vstack([vB[:-dn*5],vB[dn:-dn*4],vB[2*dn:-dn*3],vB[3*dn:-dn*2],vB[4*dn:-dn],vB[5*dn:]])
    vecB = vecB.T

    DB = pdist(vecB)
    print("computed distances")


    e1=DB.min()*3
    e2=DB.max()*1

    Neps=50
    earrayB = np.logspace(np.log10(e2),np.log10(e1),Neps)
    CB = np.zeros_like(earrayB)

    for i,eps in enumerate(earrayB):
        DB = DB[DB<eps]
        CB[i] = DB.size


    plt.figure()
    plt.loglog(earrayA,CA,label='Case A')
    plt.loglog(earrayB,CB,label='Case B')


    ind1,ind2=18,-12
    Pa = polyfit(log(earrayA[ind1:ind2]),log(CA[ind1:ind2]),1)
    z = earrayA[ind1:ind2]
    plt.plot(z,np.exp(Pa[1])*z**Pa[0],'b--',label='A: best fit, slope=%f' %Pa[0])

    ind1,ind2=18,-8
    Pb = polyfit(log(earrayB[ind1:ind2]),log(CB[ind1:ind2]),1)
    z = earrayB[ind1:ind2]
    plt.plot(z,np.exp(Pb[1])*z**Pb[0],'r--',label='B: best fit, slope=%f' %Pb[0])

    plt.legend()
    plt.xlabel(r'$\epsilon$')
    plt.ylabel(r'$C(\epsilon)$')
    plt.grid()
    plt.title('Correlation sums')
    #--------------------------------

    return None #modify as needed


def wavediff(Nloops=1):
    """
    Question 1.3
```

```
    Add input/output as needed
    Nloops: number of repetitions used during timing.
    """

    Nx_list=[1600,800,400,200,100,50]

    #Main simulation---
    T=100
    L = 100
    Nx=Nx_list[0]
    x = np.linspace(0,L,Nx+1)
    x = x[:-1]
    k = np.zeros_like(x)
    if Nx%2==0:
        k[:Nx//2]=np.arange(0,Nx//2)
        k[Nx//2:]=np.arange(-Nx//2,0)
    else:
        k[:Nx//2+1]=np.arange(0,Nx//2+1)
        k[Nx//2+1:]=np.arange(-Nx//2+1,0)
    k = k*2*np.pi/L

    #Case B
    alpha = 1-1j
    beta = 1+2j
    Nt=400
    np.random.seed(1)
    print("Simulating case B...")
    g = nwave(alpha,beta,Nx,Nt,T)
    g=g[-1,:]
    print("... done")


    fig,(ax1,ax2)=plt.subplots(2,1)
    ax1.plot(x,g.real)
    ax1.set_xlabel('x',labelpad=-8)
    ax1.set_ylabel('Real(g)')

    fk = np.fft.fft(g)/Nx
    ax2.loglog(k,np.abs(fk))
    ax2.set_xlabel('k')
    ax2.set_ylabel(r'$|\hat{g}_k|$')
    ax2.grid()
    ax1.set_title('Test function, Nx=%d'%(Nx))
    #-----------------

    #FD coefficients---
    a = 25/16
    b = 0.2
    c = -1/80
    al = 0.375

    abc = -17/6
    bbc = 1.5
    cbc = 1.5
    dbc = -1/6
    #----------



    #Differentiate test function using FFT and FD methods with different Nx---
    dg_fft_list=[]
    dt_fft = []
    dg_fd_list=[]
    dt_fd = []

    for Nx in Nx_list:

        iskip = Nx_list[0]//Nx
        gi = g[::iskip]


        #Fourier differentiation---
```

```
        k = np.zeros(Nx)
        if Nx%2==0:
            k[:Nx//2]=np.arange(0,Nx//2)
            k[Nx//2:]=np.arange(-Nx//2,0)
        else:
            k[:Nx//2+1]=np.arange(0,Nx//2+1)
            k[Nx//2+1:]=np.arange(-Nx//2+1,0)
        ik = 1j*k*2*np.pi/L

        t1 = time()
        for i in range(Nloops):
            dg = np.fft.ifft(ik*np.fft.fft(gi))
        t2 = time()

        dg_fft_list.append(dg)
        dt_fft.append(t2-t1)
        #--------------------


        #Finite differences-------
        hinv = Nx/L
        abc2 = abc*hinv
        bbc2 = bbc*hinv
        cbc2 = cbc*hinv
        dbc2 = dbc*hinv
        a2 = 0.5*a*hinv
        b2 = 0.25*b*hinv
        c2 = c/6*hinv


        AB = np.zeros((3,Nx))
        AB[1,:] = 1
        AB[0,1:] = al
        AB[2,0:] = al
        AB[0,1] = 3
        AB[2,-2] = 3
        R = np.zeros(Nx,dtype=complex)
        f = np.zeros(Nx+4,dtype=complex)
        f[2:2+Nx] = gi
        f[:2] = gi[-2:]
        f[-2:] = gi[:2]
        R[0] = abc2*f[2] + bbc2*f[3] + cbc2*f[4] + dbc2*f[5]
        R[-1] = -(abc2*f[-3] + bbc2*f[-4] + cbc2*f[-5] + dbc2*f[-6])

        R[1:-1] = (c2*(-f[:Nx-2]+f[6:Nx+4]) + b2*(-f[1:Nx-1]+f[5:Nx+3]) + a2*(-f[2:Nx]+f[4:Nx+2]))

        t1 = time()
        for i in range(Nloops):
            dg = solve_banded((1,1),AB,R,False,False,check_finite=False)
        t2 = time()

        dg_fd_list.append(dg)
        dt_fd.append(t2-t1)
        #--------------------

    fig,(ax1,ax2)=plt.subplots(1,2)
    ax1.plot(Nx_list,dt_fft,'x--',label='Fourier')
    ax1.plot(Nx_list,dt_fd,'o--',label='finite difference')
    ax1.set_xlabel('Nx')
    ax1.set_ylabel('Walltime (sec)')
    ax1.set_title('Timing results, Nloops=%d'%(Nloops))
    ax1.legend()

    ax2.plot(Nx_list,np.array(dt_fd)/np.array(dt_fft),'x--')
    ax2.set_xlabel('Nx')
    ax2.set_ylabel(r'$t_{fd}/t_{fft}$',labelpad=-8)
    ax2.set_title('Ratio of walltimes')

    #Compute errors---
    eps_fft = []
    eps_fd = []
```

```
    dg_exact = dg_fft_list[0]
    for i,Nx in enumerate(Nx_list):
        iskip = Nx_list[0]//Nx
        eps_fft.append(np.mean(np.abs(dg_exact[::iskip]-dg_fft_list[i])))
        eps_fd.append(np.mean(np.abs(dg_exact[::iskip]-dg_fd_list[i])))

    plt.figure()
    plt.loglog(Nx_list,eps_fft,'x--',label='Fourier')
    plt.loglog(Nx_list,eps_fd,'o--',label='finite difference')
    P=np.polyfit(np.log(Nx_list),np.log(eps_fd),1)
    plt.loglog(Nx_list,np.exp(P[1])*Nx_list**P[0],'k:',label='best fit, slope=%f'%P[0])
    plt.xlabel('Nx')
    plt.ylabel(r'$\epsilon_{ave}$')
    plt.title('Average error')
    plt.legend()
    plt.grid()

    return None #modify as needed
```