SC3 Hand-in

March 27, 2020

[1]:
```
#imports
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy.special import hankel1
from scipy.integrate import odeint
import time
plt.style.use('ggplot')
```

[2]:
```
#loading in data
os.chdir(r'C:\Users\milok\Desktop\Sci Comp CW3')
data1 = np.load('data1.npy')
data2 = np.load('data2.npy')
data3 = np.load('data3.npy')
A = np.load('data2.npy')
r = np.load('r.npy')
th = np.load('theta.npy')
t=np.arange(0,20*np.pi/80,np.pi/80)
```

0.1 Part 1

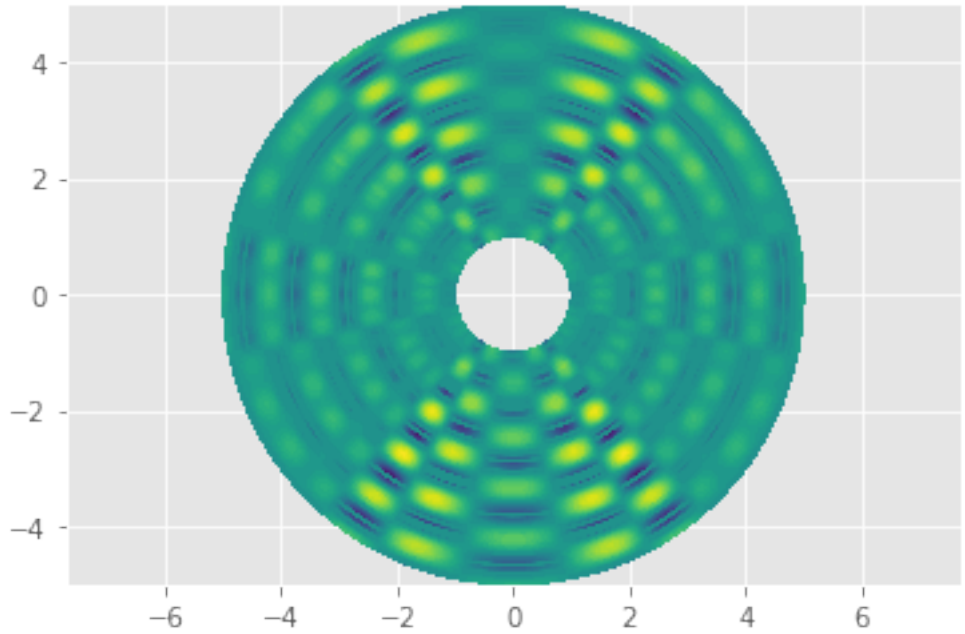
1)

[3]:
```
#defining hfield
def hfield(r,th,h,levels=50):
    """Displays height field stored in 2D array, h,
    using polar grid data stored in 1D arrays r and th.
    Modify as needed.
    """
    thg,rg = np.meshgrid(th,r)
    xg = rg*np.cos(thg)
    yg = rg*np.sin(thg)
    plt.figure()
    plt.contourf(xg,yg,h,levels)
    plt.axis('equal')
    return None
```

[4]:
```
def repair2(R,p,l=1.0,niter=10,inputs=()):
    """
    Question 1.1: Repair corrupted data stored in input
    array, R.
    Input:
        R: 2-D data array (should be loaded from data1.npy)
        p: dimension parameter
        l: l2-regularization parameter
        niter: maximum number of iterations during optimization
        inputs: can be used to provide other input as needed
    Output:
        A,B: a x p and p x b numpy arrays set during optimization
    """
    #problem setup
    R0 = R.copy()
    a,b = R.shape
    iK,jK = np.where(R0 != -1000) #indices for valid data
    aK,bK = np.where(R0 == -1000) #indices for missing data

    S = set()
    for i,j in zip(iK,jK):
            S.add((i,j))

    #Set initial A,B
    A = np.ones((a,p))
    B = np.ones((p,b))

    #Create lists of indices used during optimization
    mlist = [[] for i in range(a)]
    nlist = [[] for j in range(b)]

    for i,j in zip(iK,jK):
        mlist[i].append(j)
        nlist[j].append(i)

    dA = np.zeros(niter)
    dB = np.zeros(niter)

    for z in range(niter):
        Aold = A.copy()
        Bold = B.copy()

        #Loop through elements of A and B in different
        #order each optimization step
        for m in np.random.permutation(a):
            for n in np.random.permutation(b):
                if n < p: #Update A[m,n]
```

```
                    Asum = 0
                    #took this out of the for loop
                    Bfac = sum(B[n,mlist[m]]**2)
                    for j in mlist[m]:
                        #got rid of this for loop
                        Rsum = sum(np.delete(A[m,:],n)*np.delete(B[:,j],n))
                        Asum += (R[m,j] - Rsum)*B[n,j]
                    A[m,n] = Asum/(Bfac+l) #New A[m,n]
                if m < p:
                    #took this out of the for loop
                    Afac =sum(A[nlist[n],m]**2)
                    Bsum = 0
                    for i in nlist[n]:
                        #got rid of this for loop
                        Rsum = sum(np.delete(A[i,:],m)*np.delete(B[:,n],m))
                        Bsum += (R[i,n]-Rsum)*A[i,m]
                    B[m,n] = Bsum/(Afac+l)
        dA[z] = np.sum(np.abs(A-Aold))
        dB[z] = np.sum(np.abs(B-Bold))
        print("z,dA,dB=",z,dA[z],dB[z])
    return A,B

A,B = repair2(data1,10,niter=15,l=50)
R = np.matmul(A,B)
hfield(r,th,R)
```

```
z,dA,dB= 0 1741.4582434036083 3575.461965864156
z,dA,dB= 1 2346.8105817422975 1049.3240525439107
z,dA,dB= 2 1657.881890681112 371.71981035562675
z,dA,dB= 3 765.4156403418049 309.4795903504663
z,dA,dB= 4 399.8724925029137 414.7293833992647
z,dA,dB= 5 76.50322914583612 89.59725766424374
z,dA,dB= 6 1.236330011024144 3.0395682288474095
z,dA,dB= 7 0.20165849103813727 0.06138254939622664
z,dA,dB= 8 0.0055081957094216035 0.010248541221799226
z,dA,dB= 9 0.0007404214947236589 0.0004492294817377299
z,dA,dB= 10 5.0057185253064915e-05 7.699731790687698e-05
z,dA,dB= 11 1.0303443623568249e-05 1.8194330587794028e-06
z,dA,dB= 12 2.5948290854749447e-07 1.0897877551174687e-06
z,dA,dB= 13 7.963319956477633e-08 3.2706512258141595e-08
z,dA,dB= 14 2.518744625985502e-09 7.250098908583567e-09
```



[5]:
```python
#repaired function
def repair2(R,p,l=1.0,niter=10,inputs=()):
    """
    Question 1.1: Repair corrupted data stored in input
    array, R.
    Input:
        R: 2-D data array (should be loaded from data1.npy)
        p: dimension parameter
        l: l2-regularization parameter
        niter: maximum number of iterations during optimization
        inputs: can be used to provide other input as needed
    Output:
        A,B: a x p and p x b numpy arrays set during optimization
    """
    #problem setup
    R0 = R.copy()
    a,b = R.shape
    iK,jK = np.where(R0 != -1000) #indices for valid data
    aK,bK = np.where(R0 == -1000) #indices for missing data

    S = set()
    for i,j in zip(iK,jK):
            S.add((i,j))

    #Set initial A,B
    A = np.ones((a,p))
```

```
    B = np.ones((p,b))

    #Create lists of indices used during optimization
    mlist = [[] for i in range(a)]
    nlist = [[] for j in range(b)]

    for i,j in zip(iK,jK):
        mlist[i].append(j)
        nlist[j].append(i)

    dA = np.zeros(niter)
    dB = np.zeros(niter)

    for z in range(niter):
        Aold = A.copy()
        Bold = B.copy()

        #Loop through elements of A and B in different
        #order each optimization step
        for m in np.random.permutation(a):
            for n in np.random.permutation(b):
                if n < p: #Update A[m,n]
                    Bfac = 0.0
                    Asum = 0
                    for j in mlist[m]:
                        Bfac += B[n,j]**2
                        Rsum = 0
                        for k in range(p):
                            if k != n: Rsum += A[m,k]*B[k,j]
                        Asum += (R[m,j] - Rsum)*B[n,j]
                    A[m,n] = Asum/(Bfac+l) #New A[m,n]
                if m < p:
                    Afac = 0.0
                    Bsum = 0
                    for i in nlist[n]:
                        Afac += A[i,m]**2
                        Rsum = 0
                        for k in range(p):
                            if k != m:
                                Rsum+=A[i,k]*B[k,n]
                        Bsum += (R[i,n]-Rsum)*A[i,m]
                    B[m,n] = Bsum/(Afac+l)
        dA[z] = np.sum(np.abs(A-Aold))
        dB[z] = np.sum(np.abs(B-Bold))
        print("z,dA,dB=",z,dA[z],dB[z])
    return A,B
```

In the original repair function when calculating the 'Rsum' it iteratively increased the value ignoring the nth element of the arrays. Instead we can just delete the nth element from each array and then calculate the dot product ignoring the for loop. Similarly when calculating the Bfac it iterated through each of valid rows of B and summed the squares, we shortcut this by defining a new matrix on the valid rows and summing the squares of the elements eliminating the need for a for loop.

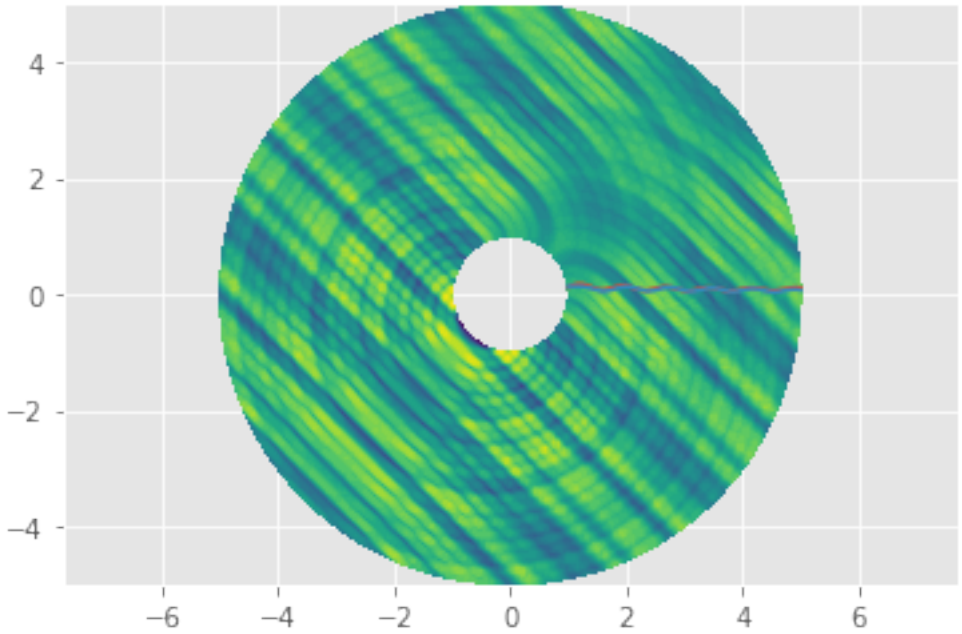
The parameters used were over 15 iterations and l=50, this was done my trial and error as any wildly different l values resulted in a big range of values in the array which mean that the plot didn't show anything resembling the original image. 15 iterations also seemed to be enough to converge every time. This could be improved by…

2i)

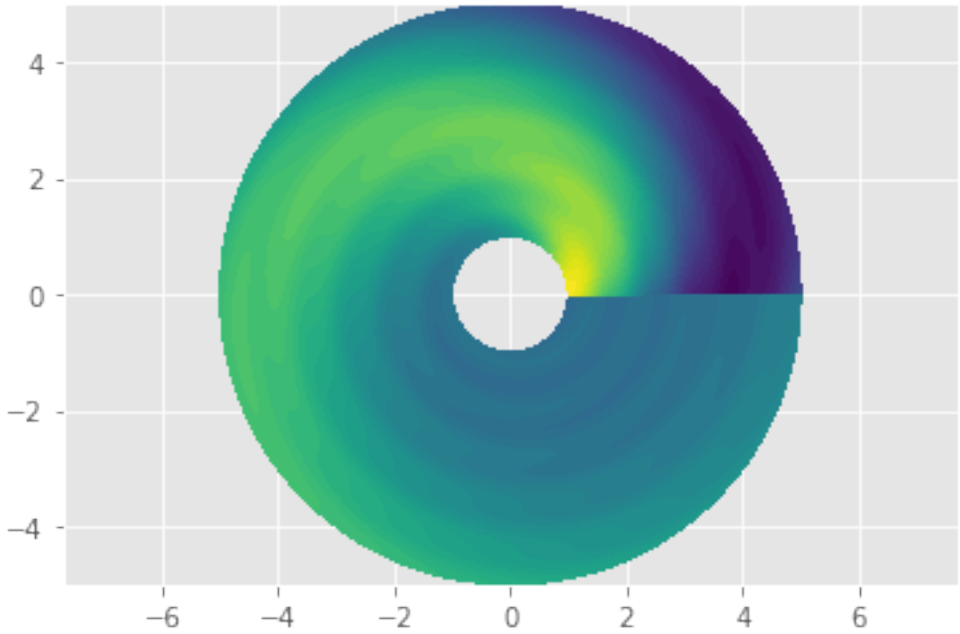
[7]:
```
def outwave(r0):
    """
    Question 1.2i)
    Calculate outgoing wave solution at r=r0
    See code/comments below for futher details
        Input: r0, location at which to compute solution
        Output: B, wave equation solution at r=r0
    """
    import numpy as np
    A = np.load('data2.npy')
    r = np.load('r.npy')
    th = np.load('theta.npy')

    Nr,Ntheta,Nt = A.shape
    B = np.zeros((Ntheta,Nt))

    #plotting the height map for t=0
    hfield(r,th,data3[:,:,0])

    #defining potential function
    def hankel(k,m,w,r,c,tt):
        return k*(np.real(hankel1(m,w*r)*np.exp(-complex(0,1)*c*tt)+np.
↪conj(hankel1(m,w*r)*np.exp(complex(0,1)*c*tt))))

    #attempting to fit the function over a range of possible parameter values
    i,j=0,1
    tt=t[i]
    m=th[j]
    c1=0
    c2=0
    min_err=1000
    iteration=0
    targ=A[:,j,i]
    for c in np.arange(-0.15,0.15,0.05):
        for k1 in np.arange(0,0.07,0.02):
```

```
            for w1 in np.arange(1,1.5,1.5):
                for k2 in np.arange(-0.5,0,0.1):
                    for w2 in np.arange(6,10,0.5):
                        iteration+=1
                        if iteration%200==0:
                            print('iteration={}'.format(iteration))
                        Y = [hankel(k1,m,w1,r,c1,tt)+hankel(k2,m,w2,r,c2,tt)+c␣
↪for r in r]
                        err = np.abs(np.mean(np.abs(Y-targ)))
                        if err < min_err:
                            min_err = err
                            min_Y = Y
                            min_vars = k1,w1,k2,w2,c
    print('minimum variables k1,w1,k2,w2,c are',min_vars,'respectively and␣
↪minimum error is',min_err)
    plt.plot(r,min_Y)
    plt.plot(r,targ)
    plt.show()

    rs =␣
↪[hankel(min_vars[1],m,min_vars[1],r,0,tt)+hankel(min_vars[2],m,min_vars[3],r,0,tt)+min_vars
↪for r in r]
    B = np.zeros((len(th),len(r)))
    for i in range(len(th)):
        m = th[i]
        B[i] = [hankel(min_vars[1],m,min_vars[1],r,0.
↪1,tt)+hankel(min_vars[2],m,min_vars[3],r,-0.1,tt)+min_vars[4] for r in r]
    hfield(r,th,B.T)
    return B.T

outwave(0)
```

```
iteration=200
iteration=400
iteration=600
iteration=800
minimum variables k1,w1,k2,w2,c are (0.02, 1.0, -0.10000000000000009, 8.5, 0.1)
respectively and minimum error is 0.018582102411949904
```




```
[7]: array([[ 1.62200752,  1.62538836,  1.6271751 , …,  0.03718985,
               0.03679116,  0.03640947],
             [ 1.61638659,  1.62046702,  1.62294611, …,  0.03959125,
               0.03911566,  0.03865717],
             [ 1.61059733,  1.61539106,  1.61857677, …,  0.04233   ,
               0.04177854,  0.04124411],
             …,
             [-0.2530064 , -0.27313043, -0.29276703, …,  0.29167575,
               0.28726911,  0.28293482],
             [-0.2461256 , -0.26632295, -0.28604046, …,  0.29610329,
               0.29160637,  0.28717998],
             [-0.23942795, -0.25970203, -0.27950364, …,  0.30074783,
               0.29616797,  0.29165658]])
```



As you can see the script struggles to produce the plot from the differential equation but I will how I did what I did. The second plot is supposed to be a copy of the first plot. I took a particular theta and time to compact the 3 dimensional array into a 1 dimensional array which can be plotted. Then I assumed the function to be comprised of functions of the form of f = H_(1)^m(r)exp(−ict). In particular I assumed it to be the sum of two independent complex conjugates of f added to f. The complex conjugate was included so the function had no imaginary parts and the second one was included since it seems as if there are two waves at play when you look at the actual plot.

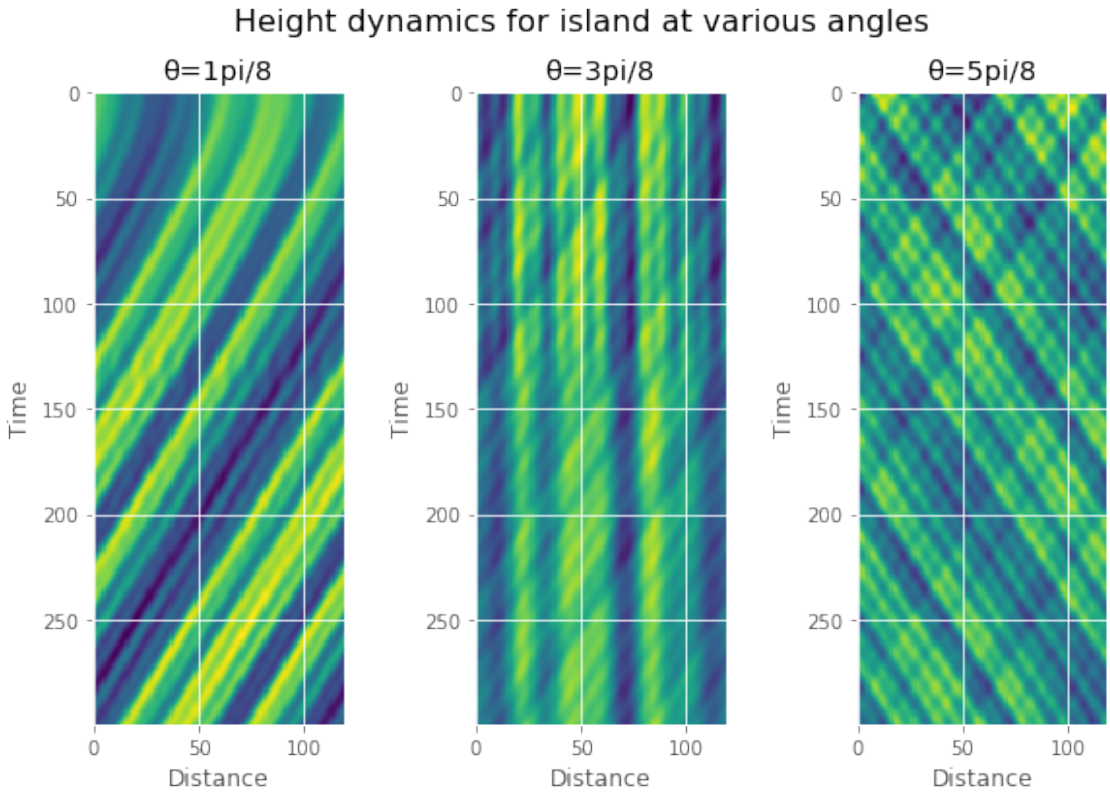
Then I created a 1D array of this for a particular time and theta value as well as for multiple parameters and found which one closely resembled the actual plot by using mean squared error. Computational limitations meant that I was unable to do this for every theta and time value and as such was unable to find a combination of parameters that held for every time and theta value. Therefore I was unable to verify if my assumption of the form of the height values was even correct!

ii)

[8]:
```
data3_0 = data3[:,289//8,:]
data3_1 = data3[:,289*3//8,:]
data3_2 = data3[:,289*5//8,:]
```

[9]:
```
fig, axs = plt.subplots(1,3,figsize=(10,6))
fig.suptitle('Height dynamics for island at various angles',fontsize=16)
for i,j in enumerate([1,3,5]):
    axs[i].imshow(data3[:,289*j//8,:])
    axs[i].set_title(' ={}pi/8'.format(j))
    axs[i].set_ylabel('Time')
    axs[i].set_xlabel('Distance')
```

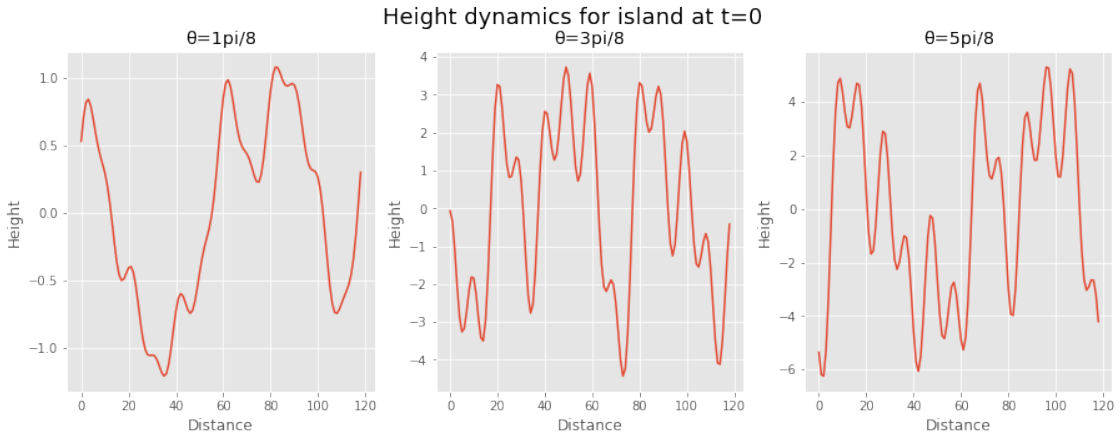
```
plt.show()
```



Here we see how the height of the waves varies at particular angles from the island. Pixels further right indicate height readings from further away from the island, pixels further down indicate height readings from later time periods and the colours transition from blue to yellow indicate low to high waves. In the first plot we see that the waves are moving towards the island at a steady rate similar to how we expect waves to crash onto a beach. The second plot shows that the waves are almost stationary. They move away from the island at a much very slow rate. There also appears to be a much weaker wave moving towards the island much faster which has less effect on the height of the waves. The third plot shows waves moving away from and toward the island with similar effects on the height of the ocean creating a criss-cross effect in the plot. Finally it is worth noting with all the waves there are bands of high readings followed by low bands of low readings which alternate with regularity.

[10]:
```
fig, axs = plt.subplots(1,3,figsize=(15,5))
fig.suptitle('Height dynamics for island at t=0',fontsize=18)
te = np.arange(119)
for i,j in enumerate([1,3,5]):
    axs[i].plot(te,data3[0,289*j//8,:])
    axs[i].set_title(' ={}pi/8'.format(j))
    axs[i].set_ylabel('Height')
    axs[i].set_xlabel('Distance')
```

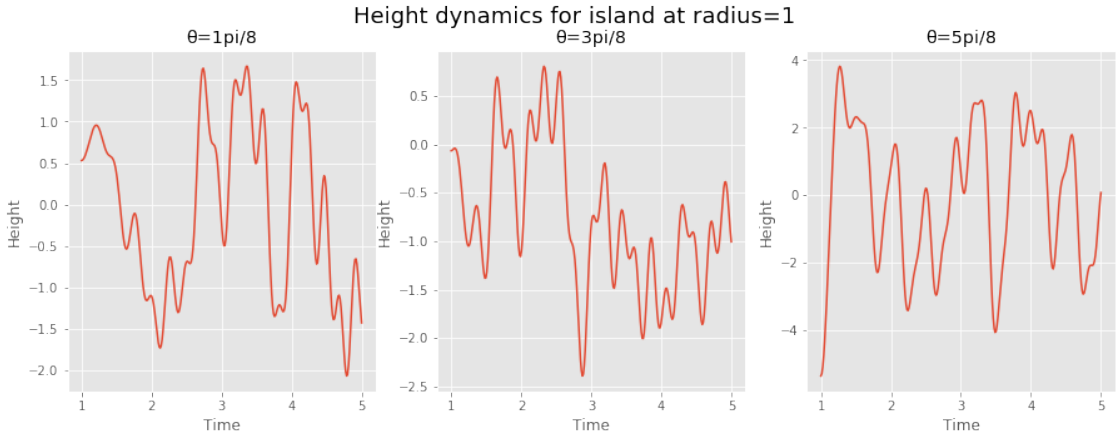
```
plt.show()
```



Taking a closer look at the plots for a particular time point (t=0) we see for the first plot it looks similar to the model we analysed in 1.2i) suggesting that for this angle there is only one direction of wave to be considered here. It oscillates in a sinusoidal manner but there is no clear pattern to be found.

For the second two plots we see that it there is much more variation in the height which, considering all of these readings are from the same island suggests that there is a second wave only acting perpendicular to pi/8. We know this since the first plot is unaffected by this wave. It results in a reading that oscillates at more frequently and with more dramatic swings.

[11]:
```
fig, axs = plt.subplots(1,3,figsize=(15,5))
fig.suptitle('Height dynamics for island at radius=1',fontsize=18)
te = np.arange(119)
for i,j in enumerate([1,3,5]):
    axs[i].plot(r,data3[:,289*j//8,0])
    axs[i].set_title(' ={}pi/8'.format(j))
    axs[i].set_ylabel('Height')
    axs[i].set_xlabel('Time')
plt.show()
```



Here we look at how the closest point to the centre of the island varies through time. Overall we see that the height oscillates continuously throughout the time readings. From one reading to the next we see that the height changes with no discernable pattern but appears to represent some combination of sinusoidal waves.

3)

[12]:
```
def reduce(H,p=20):
    """
    Question 1.3: Construct one or more arrays from H
    that can be used by reconstruct
    Input:
        H: 3-D data array
        inputs: can be used to provide other input as needed
    Output:
        arrays: a tuple containing the arrays produced from H
    """
    M,N,T=H.shape
    H_2 = np.zeros((M,p,T))
    Us = np.zeros((N,T))
    var = 0
    total_var = 0
    G_2 = np.zeros((M,N,T))
    for i in range(T):
        X = H[:,:,i]
        mean = np.outer(np.ones((M,1)),X.mean(axis=0))
        X2 = X - mean
        U,S,VT = np.linalg.svd(X2.T)
        G = np.dot(U.T,X2.T)
        G_2[:,:,i] = G.T
        H_2[:,:,i] = G[:p].T
        Us[:,i] = U[0]
```

```
        var+=sum(S[:p])
        total_var+=sum(S)
    print('The proportion of variance accounted for is {}% using only {}% of␣
↪the matrix'
          .format(np.round(100*var/total_var,3),np.round(100*p/289,3)))

    return H_2,Us,G_2

reduced_H,Us,G_2 = reduce(data3)
```

```
The proportion of variance accounted for is 79.855% using only 6.92% of the
matrix
```

Here we see that the H has been reduced from a 3D array to a smaller 3D array using PCA. For each time period the array has been converted into SVD form. Then the first 20 rows of this form (note that we do not need to worry about ordering as numpy orders the array by eigenvalues by default) since they have the greatest variance. 20 has been chosen as the number of rows as it is a good trade off between size and accuracy to the original matrix.

We see in the function that it has shown that 80% of the variance has been accounted for and this has been achieved with a new matrix that is 7% of the size of the original.

[14]:
```
def reconstruct(G,U,G_2,inputs=()):
    """
    Question 1.3: Generate matrix with same shape as H (see reduce above)
    that has some meaningful correspondence to H
    Input:
        arrays: tuple generated by reduce
        inputs: can be used to provide other input as needed
    Output:
        Hnew: a numpy array with the same shape as H
    """
    M,p,T = G.shape
    N,T = U.shape
    Hnew = np.zeros((M,N,T))

    for i in range(T):
        Hnew = np.outer(U[:,i],G_2[:,0,i]).T

    return Hnew

Hnew = reconstruct(reduced_H,Us,G_2)
```

Here we reconstruct the matrix back in the dimensions of the original matrix by calculating the outer product of the 1st eigenvector of U with the first row of G for each time the data is recorded over. Overall we have found a way to reduce the dataset by 14 times by literally looking at the size of the new array whilst retaining around 80% of the variance.

0.2 Part 2

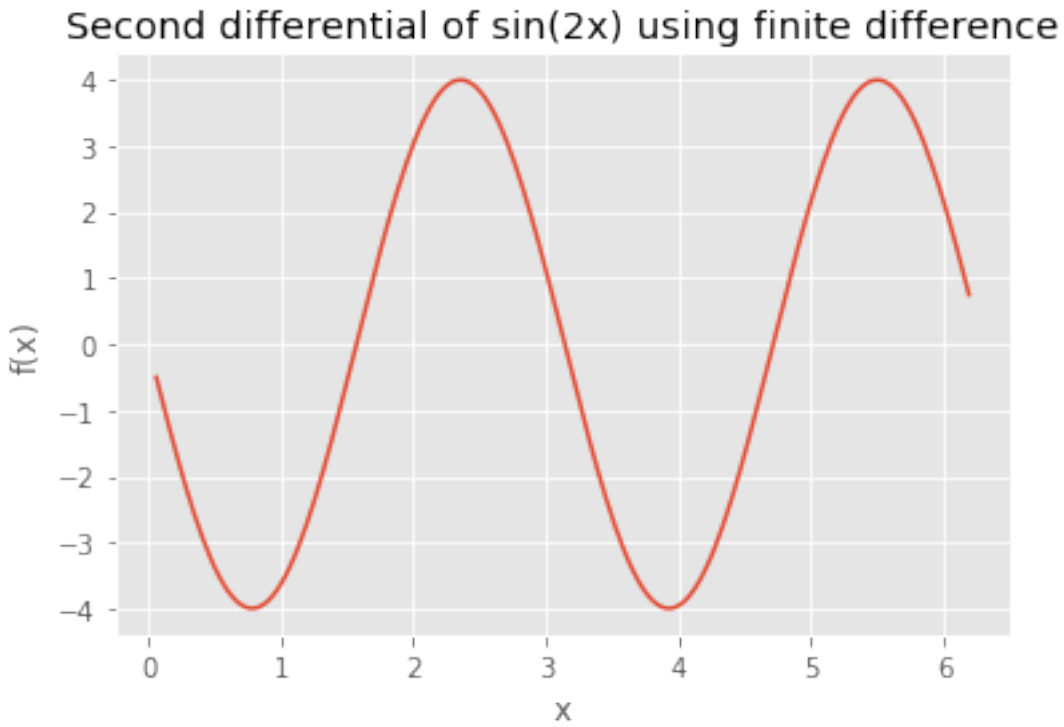
1i)

[15]:
```
def newdiff(f,h):
    """
    Question 2.1 i)
    Input:
        f: array whose 2nd derivative will be computed
        h: grid spacing
    Output:
        d2f: second derivative of f computed with compact fd scheme
    """

    d2f = np.zeros_like(f) #modify as needed

    #Coefficients for compact fd scheme
    alpha = 9/38
    a = (696-1191*alpha)/428
    b = (2454*alpha-294)/535
    c = (1179*alpha-344)/2140

    l = len(f)
    A = np.zeros((l,l))
    d = np.zeros(l)
    h = np.pi/100

    def f1(f,i,alpha,a,b,c,h):
        return (1/h**2)*((c/9)*(f[i+3]-2*f[i]+f[i-3])+(b/
↪4)*(f[i+2]-2*f[i]+f[i-2])+a*(f[i+1]-2*f[i]+f[i-1]))



    A[0,:2] = [1,10]
    d[0] = (1/h**2)*((145/12)*f[0]-(76/3)*f[1]+(29/2)*f[2]-(4/3)*f[3]+(1/
↪12)*f[4])
    A[-1,-2:] = [10,1]
    d[-1] = (1/h**2)*((145/12)*f[-1]-(76/3)*f[-2]+(29/2)*f[-3]-(4/3)*f[-4]+(1/
↪12)*f[-5])

    for i in range(len(A)-4):
        A[i+1][i:i+3] = [alpha,1,alpha]
        d[i+1] = f1(f,i+1,alpha,a,b,c,h)

    A[-2,-3:] = [alpha,1,alpha]
    d[-2] = (1/h**2)*((c/9)*(f[1]-2*f[-2]+f[-5])+(b/
↪4)*(f[0]-2*f[-2]+f[-4])+a*(f[-1]-2*f[-2]+f[-3]))
    A[-3,-4:-1] = [alpha,1,alpha]
```

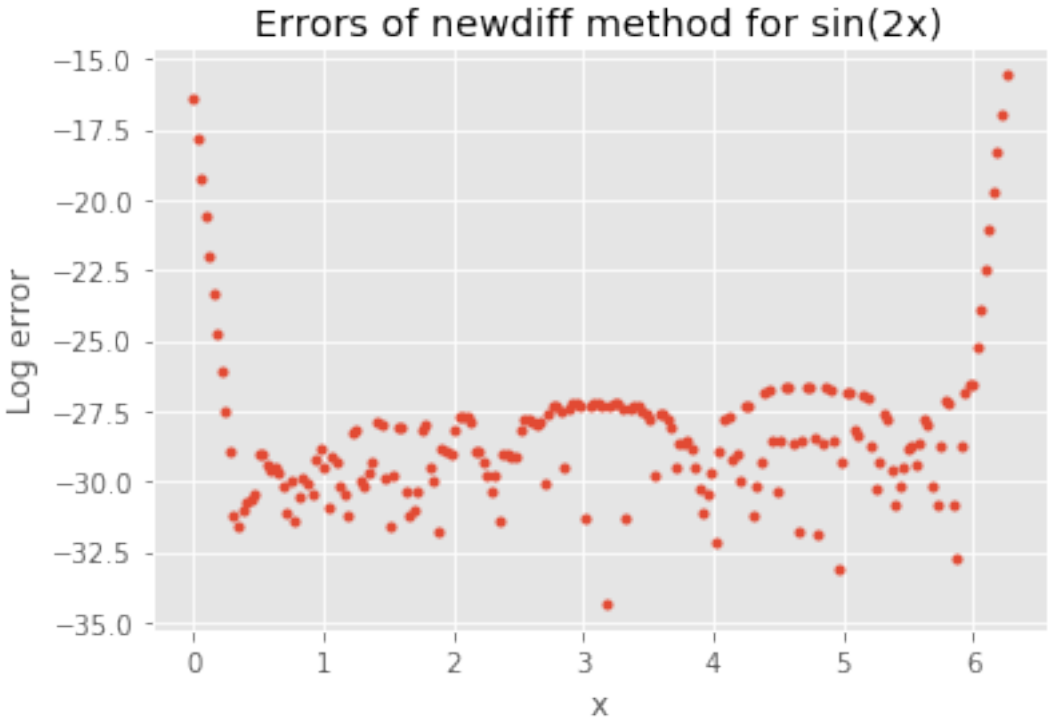
```
    d[-3] = (1/h**2)*((c/9)*(f[0]-2*f[-3]+f[-6])+(b/
↪4)*(f[-1]-2*f[-3]+f[-5])+a*(f[-2]-2*f[-3]+f[-4]))
    d2f = np.linalg.solve(A,d)
    return d2f
```

1ii)

[16]:
```
def findiff(f,h):
    fdash = [(f[i]-f[i+2])/(2*h) for i in range(len(f)-2)]
    fdash2 = [(fdash[i]-fdash[i+2])/(2*h) for i in range(len(fdash)-2)]
    return fdash2

x = np.arange(0,2*np.pi,np.pi/100)
f = [np.sin(2*x) for x in x]
h = np.pi/100

plt.plot(x[2:-2],findiff(f,h))
plt.title('Second differential of sin(2x) using finite difference')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```



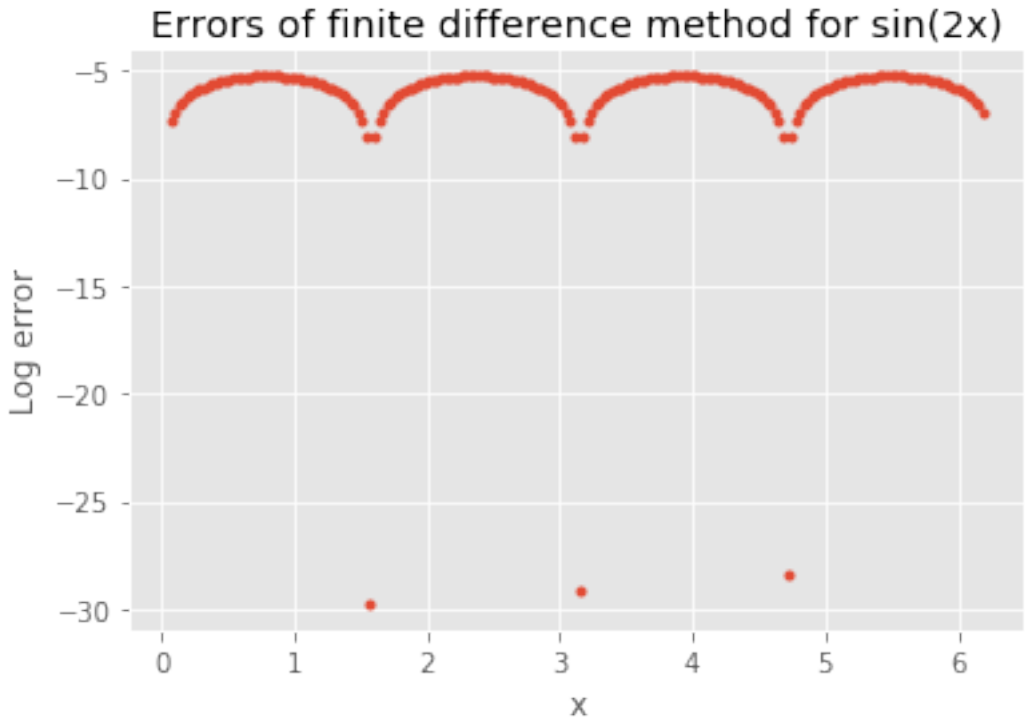
[17]:
```
x = np.arange(0,2*np.pi,np.pi/100)
f = [np.sin(2*x) for x in x]
actdiff = [-4*np.sin(2*x) for x in np.arange(0,2*np.pi,np.pi/100)]
diff = newdiff(f,1/100)
fig = plt.figure()
error = [abs(diff[i] - actdiff[i]) for i in range(len(diff))]
plt.plot(x,np.log(error),'.')
plt.title('Errors of newdiff method for sin(2x)')
plt.ylabel('Log error')
plt.xlabel('x')
plt.show()
```



The newdiff method was tested using sin(2x) between 0 and 2pi where it produced errors ranging in magnitude of e^-15 to e^-35. There appears to be a pattern in these errors where the errors are maximised around pi/2, pi, and 3pi/2 suggesting that the errors were greatest when sin(2x)=0. Also the errors increased dramatically when you got closer the start and end points suggesting that these increased the relative error.

[18]:
```
x = np.arange(0,2*np.pi,np.pi/100)
f = [np.sin(2*x) for x in x]
h = np.pi/100
actdiff = [-4*np.sin(2*x) for x in np.arange(0,2*np.pi,np.pi/100)][2:-2]
diff = findiff(f,h)
error = [abs(diff[i]-actdiff[i]) for i in range(len(diff))]
```

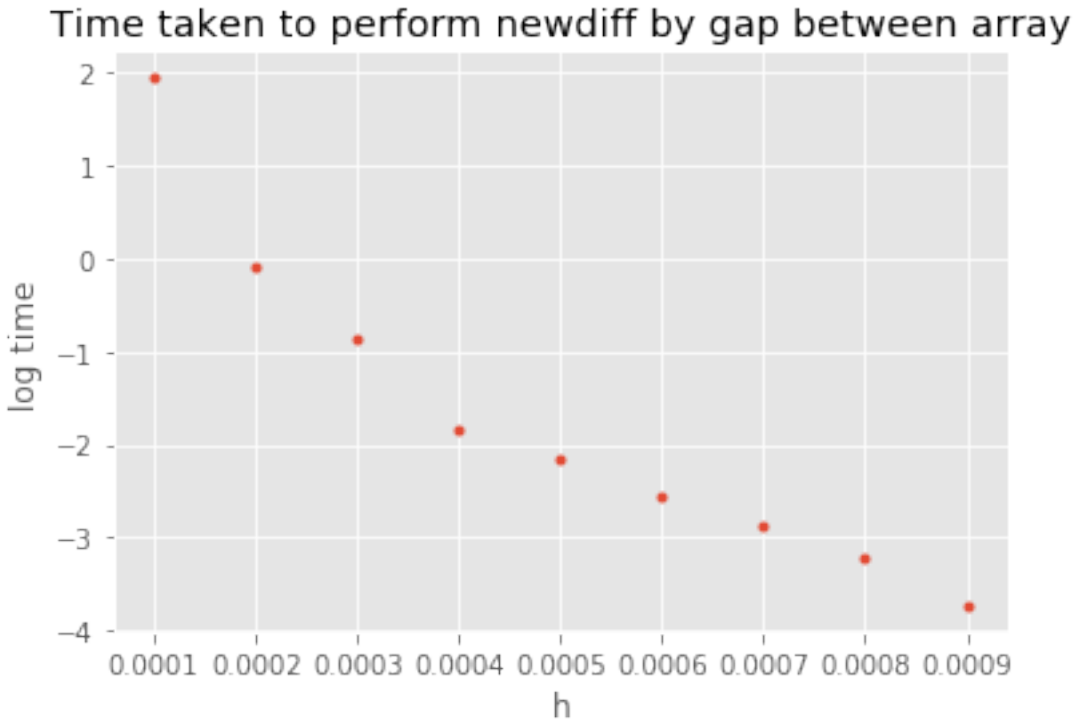
```
fig = plt.figure()
plt.plot(x[2:-2],np.log(error),'.')
plt.title('Errors of finite difference method for sin(2x)')
plt.ylabel('Log error')
plt.xlabel('x')
plt.show()
```



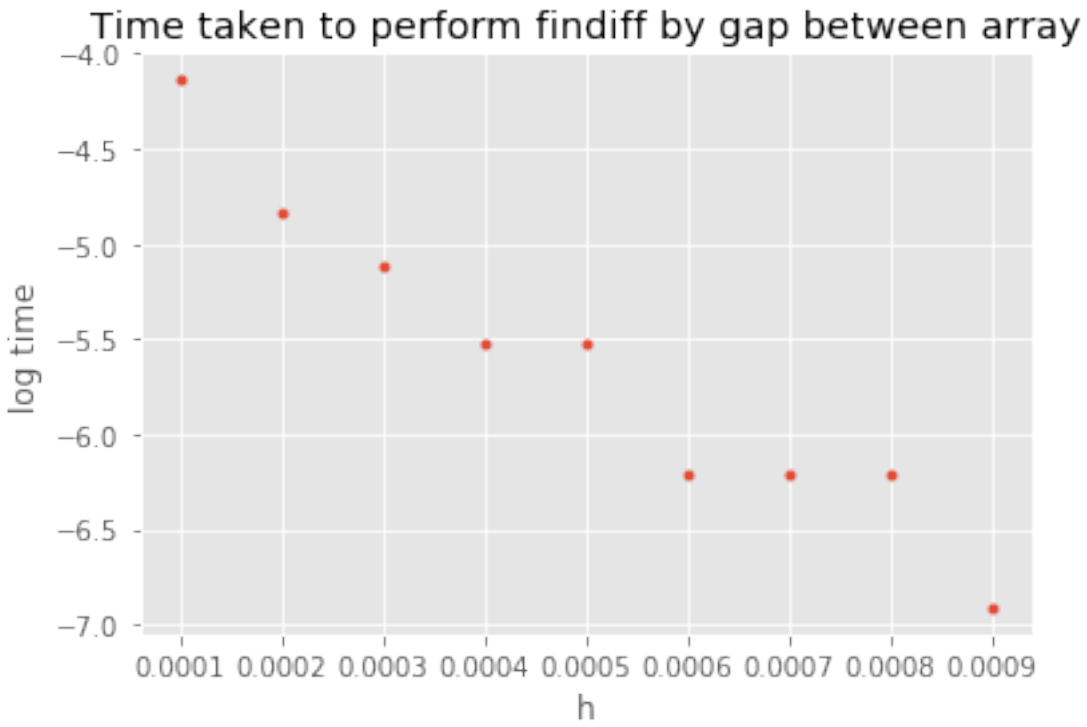
Looking at the errors in finite difference we see that they range from e^-5 to e^-10 with a few exceptions where sin(2x)=0. This relative error is much greater than for the newdiff method. Perhaps this is because the finite difference method has been calculated by repeating the derivative method twice therefore compounding any errors. The errors increase and decrease in a clear pattern where the errors seem greatest when sin(2x) is greatest and where $d^{2(\sin(2x))/dx}2$ is also greatest but it is unclear exactly why.

[19]:
```
hs = np.arange(1/10000,1/1000,1/10000)
newdifft = np.zeros(len(hs))
for i,h in enumerate(hs):
    x = np.arange(0,2*np.pi,2*np.pi*h)
    f = [np.sin(2*x) for x in x]
    start = time.time()
    newdiff(f,h)
    end = time.time()
    newdifft[i] = np.log(end-start)
```

```
plt.plot(hs,newdifft,'.')
plt.ylabel('log time')
plt.xlabel('h')
plt.title('Time taken to perform newdiff by gap between array')
plt.show()
```



[20]:
```
hs = np.arange(1/10000,1/1000,1/10000)
newdifft = np.zeros(len(hs))
for i,h in enumerate(hs):
    x = np.arange(0,2*np.pi,2*np.pi*h)
    f = [np.sin(2*x) for x in x]
    start = time.time()
    findiff(f,h)
    end = time.time()
    newdifft[i] = np.log(end-start)
plt.plot(hs,newdifft,'.')
plt.ylabel('log time')
plt.xlabel('h')
plt.title('Time taken to perform findiff by gap between array')
plt.show()
```



Both plots show a similar trend where the bigger the gap between the readings the quicker the algorithm calculated the second derivative. They also decreased in a similar pattern where the time taken is not quite proportional to log time. But if we the base of the logarithm is changed we would see the line become proportional.

For the specified lengths of h we see that finite difference methods far outperformed our newdiff method. This is likely to do with the fact that there are far more operations involved in newdiff so for this length of task it seems inaccurate. However if we consider time vs accuracy the newdiff method is far better since it recorded errors e^20 times smaller. The size of h that you would need to match these errors would require more time than newdiff method.

Overall if you need a very precise reading the newdiff method would be favoured however if you need a less precise reading for the second differential then the finite difference method is just fine.

2)

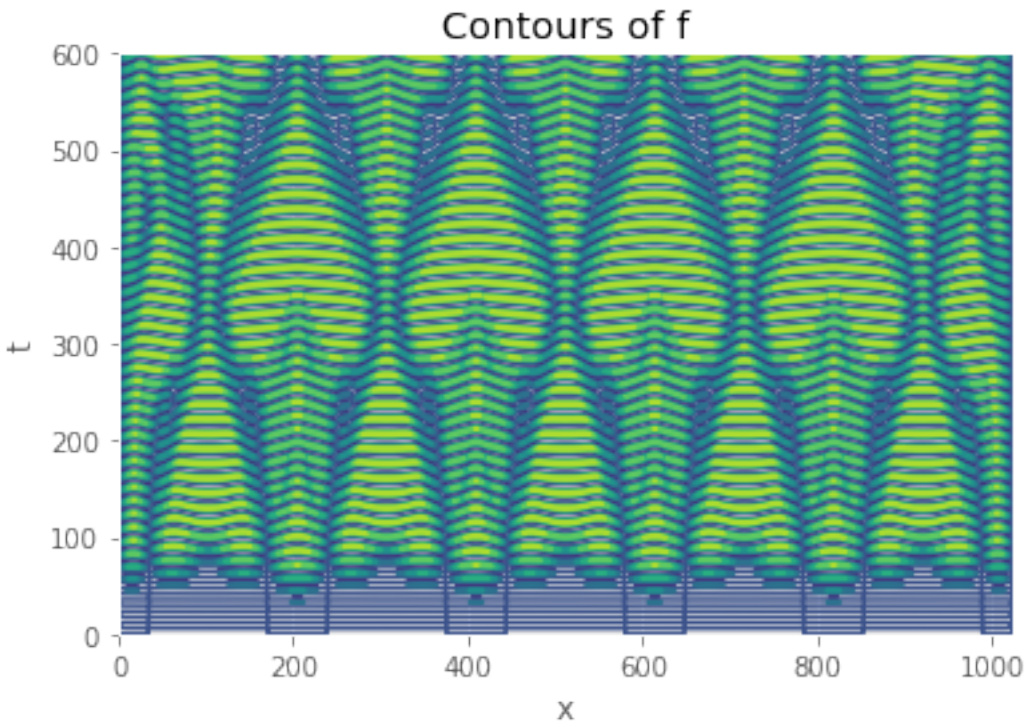
[21]:
```
def microbes(phi,kappa,mu,L = 1024,Nx=1024,Nt=1201,T=600,display=False):
    """
    Question 2.2
    Simulate microbe competition model

    Input:
    phi,kappa,mu: model parameters
    Nx: Number of grid points in x
    Nt: Number of time steps
```

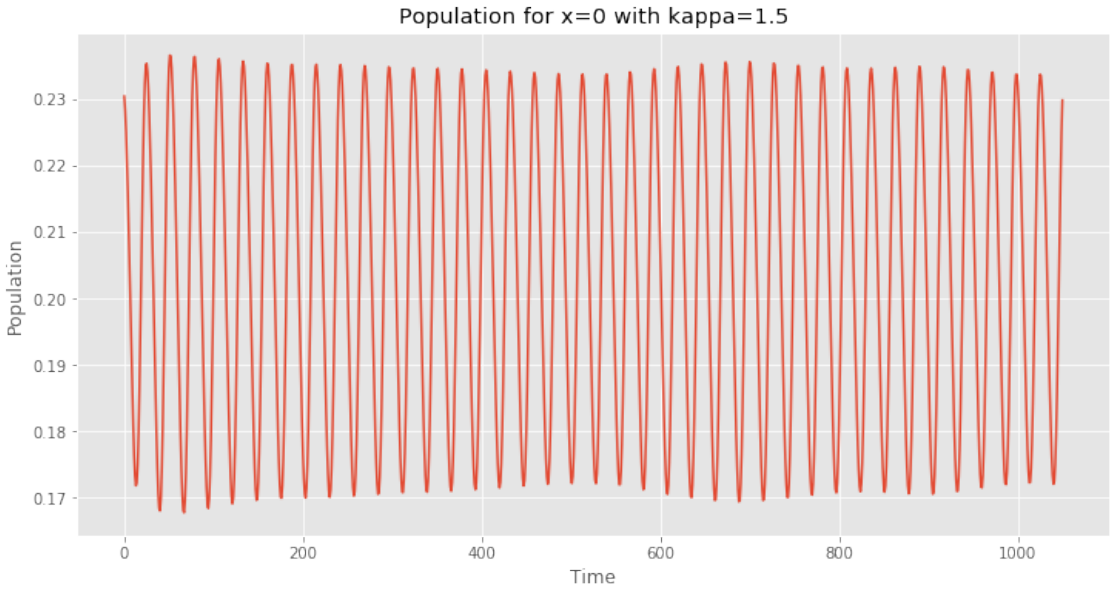
```
    T: Timespan for simulation is [0,T]
    Display: Function creates contour plot of f when true

    Output:
    f,g: Nt x Nx arrays containing solution
    """

    #generate grid
    L = 1024
    x = np.linspace(0,L,Nx)
    dx = x[1]-x[0]
    dx2inv = 1/dx**2

    def RHS(y,t,k,r,phi,dx2inv):
        #RHS of model equations used by odeint

        n = y.size//2

        f = y[:n]
        g = y[n:]

        #Compute 2nd derivatives
        d2f = (f[2:]-2*f[1:-1]+f[:-2])*dx2inv
        d2g = (g[2:]-2*g[1:-1]+g[:-2])*dx2inv

        #Construct RHS
        R = f/(f+phi)
        dfdt = d2f + f[1:-1]*(1-f[1:-1])- R[1:-1]*g[1:-1]
        dgdt = d2g - r*k*g[1:-1] + k*R[1:-1]*g[1:-1]
        dy = np.zeros(2*n)
        dy[1:n-1] = dfdt
        dy[n+1:-1] = dgdt

        #Enforce boundary conditions
        a1,a2 = -4/3,-1/3
        dy[0] = a1*dy[1]+a2*dy[2]
        dy[n-1] = a1*dy[n-2]+a2*dy[n-3]
        dy[n] = a1*dy[n+1]+a2*dy[n+2]
        dy[-1] = a1*dy[-2]+a2*dy[-3]

        return dy


    #Steady states
    rho = mu/kappa
    F = rho*phi/(1-rho)
    G = (1-F)*(F+phi)
```

```
    y0 = np.zeros(2*Nx) #initialize signal
    y0[:Nx] = F
    y0[Nx:] = G + 0.01*np.cos(10*np.pi/L*x) + 0.01*np.cos(20*np.pi/L*x)

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

    #compute solution
    print("running simulation...")
    y = odeint(RHS,y0,t,args=(kappa,rho,phi,dx2inv),rtol=1e-6,atol=1e-6)
    f = y[:,:Nx]
    g = y[:,Nx:]
    print("finished simulation")
    if display:
        plt.figure()
        plt.contour(x,t,f)
        plt.xlabel('x')
        plt.ylabel('t')
        plt.title('Contours of f')

    plt.show()
    return f,g
```

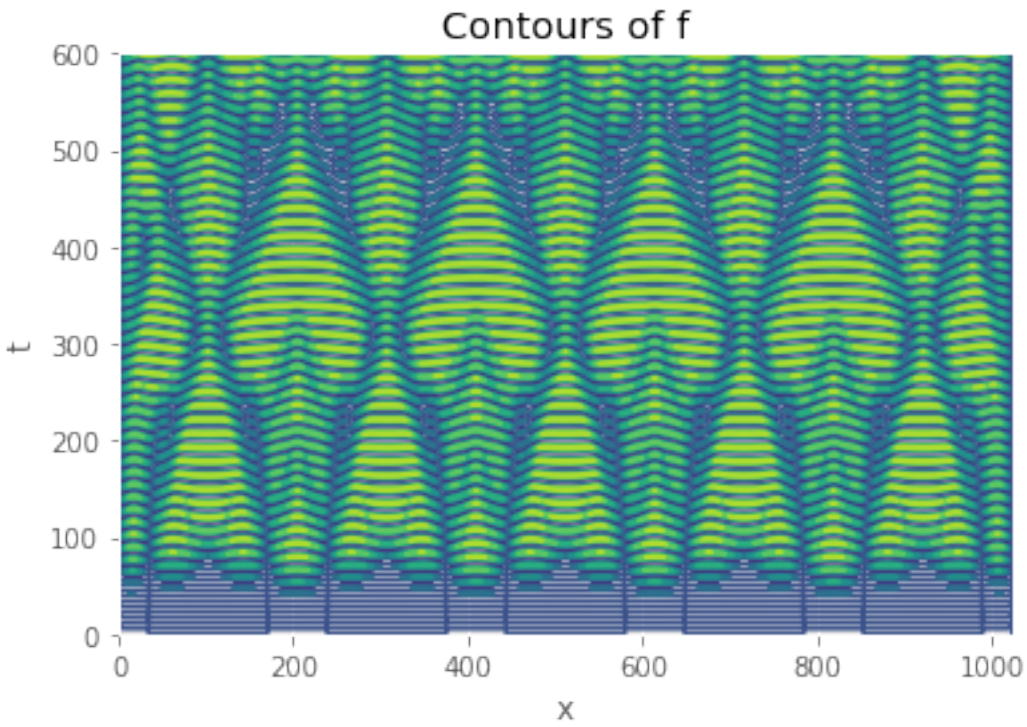
[22]:
```
phi=0.3
L=1024
Nx=1024
fs = [0,0,0]
gs = [0,0,0]
for i,kappa in enumerate([1.5,1.7,2]):
    mu=0.4*kappa
    fs[i],gs[i] = microbes(phi,kappa,mu,L,Nx,display=True)
    fig = plt.figure(figsize=(12,6))
    plt.plot(fs[i][150:,0])
    plt.title('Population for x=0 with kappa={}'.format(kappa))
    plt.ylabel('Population')
    plt.xlabel('Time')
    plt.show()
```

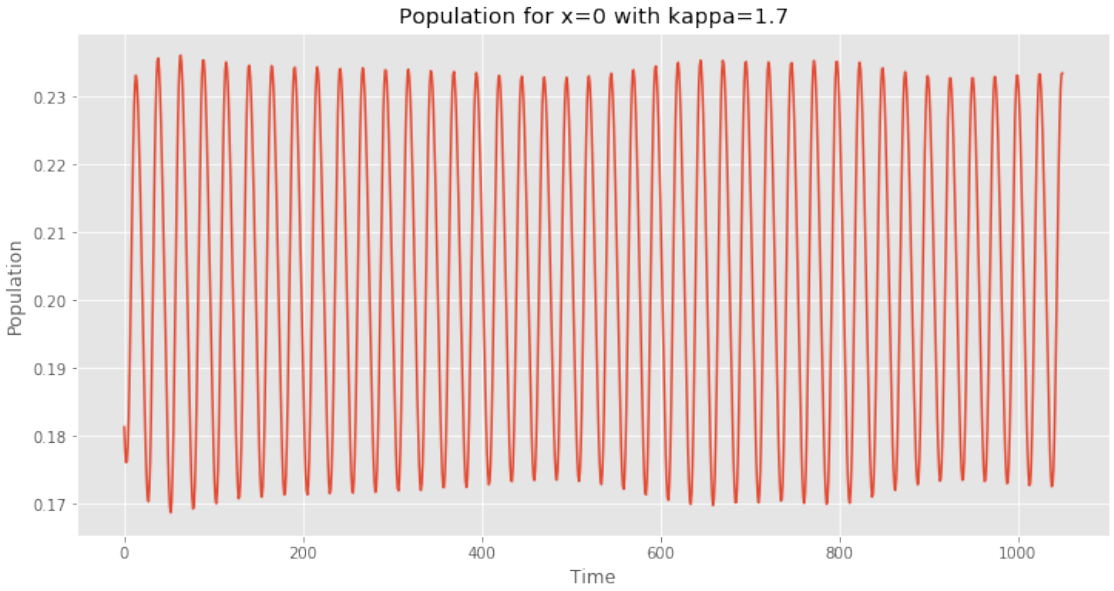
```
running simulation…
finished simulation
```



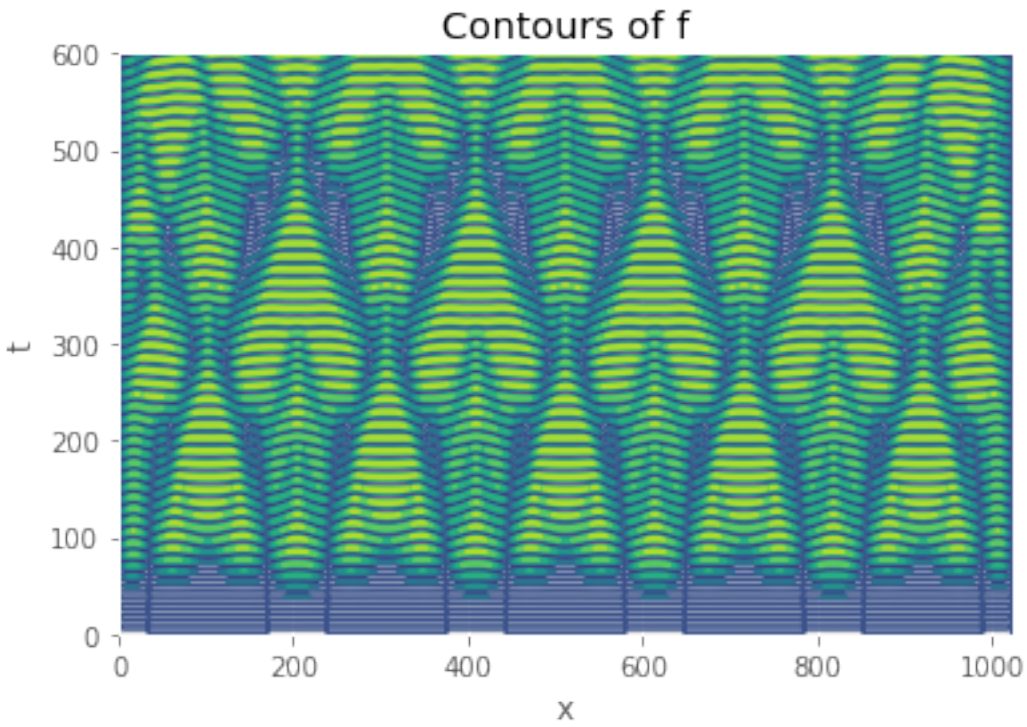


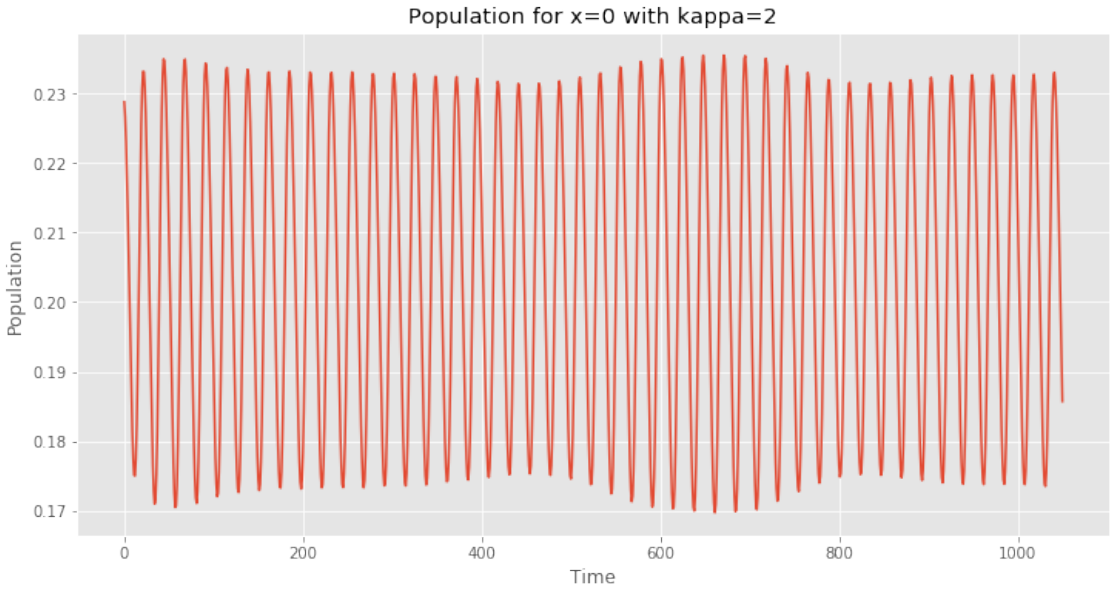

```
running simulation…
finished simulation
```






```
running simulation…
finished simulation
```





So here we see how the population of a microbe varies over time for x values ranging from 0 to 1024 for a few kappa values. We see that the higher the kappa value the bigger the effect the x value has on the population dynamics. In our above plot increasing the kappa value squeezes the plot towards the y-axis. However it does not change the rate at which the population oscillates.

This can be seen in the line plots where the population oscillates up and down at a consistent rate across all of the plots.

Chaotic plots typically have aperiodic dynamics and are sensitive to the initial conditions. Firstly we see that the dynamics are very periodic and we have seen that the plots are not very sensitive to a change in initial conditions so this would suggest that the population dynamics are not very chaotic.

To further determine the chaoticness of the system the fractal dimension could be calculated. Here we could plot the correlation sum for various epsilons and the slope would determine the fractal dimension. If the dimension is higher than expected this would indicate a chaotic system.

3)

For a finite T you could consider the effects of varying the parameters on the populations over time. Perhaps you could fix all of the variables and change one and see how the dynamics of this would vary. You could calculate the fractal dimension of the plots and see which variable contributes the most to the chaos of the system. In our parameter study we only looked at varying kappa but we could derive further results from looking at the other parameters.

Also we could perform a study with regards to the lyapunov exponent. We could do this by using our initial condition then integrating forward using the solution we were given earlier then run this simulation for over a great number of time intervals. Then repeat this for slightly deviated variables and compute the difference between the two solutions using a mean squared error. In linear systems we would expect to see that the distance would change linearly whereas in chaotic systems it would increase exponentially.