```
>> step_potential(5, [10 10 15 25])
   2.0062e+03

   3.9886e+03

   2.0739e+05

   2.7652e+09

>> step_potential(5, [10 10 15 25 30 40 50 50 67 68 73])
   2.0062e+03

   3.9886e+03

   2.0739e+05

   2.7652e+09

   3.4711e+11

   5.7379e+15

   1.0163e+20

   2.0325e+20

   1.8413e+27

   6.7740e+27

   6.8940e+29

>> step_potential(5, [100 10 15 25 30 40 50 50 67 68 73])
   2.6612e+41

   2.6612e+41

   2.6612e+41

   2.6612e+41

   2.6612e+41

   2.6612e+41
```

```
   2.6612e+41

   2.6612e+41

   2.6612e+41

   2.6612e+41

   2.6612e+41

>> step_potential(5, [1 10 15 25 30 40 50 50 67 68 73])
   24.7861

   2.0072e+03

   2.0541e+05

   2.7652e+09

   3.4711e+11

   5.7379e+15

   1.0163e+20

   2.0325e+20

   1.8413e+27

   6.7740e+27

   6.8940e+29

>> step_potential(5, [1 10 15 25 30 40 5 5 5 5 10])
   24.7861

   2.0072e+03

   2.0541e+05

   2.7652e+09

   3.4711e+11
```

```
   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

>> step_potential(5, [1 10 15 25 30 40 5 5 5 5000000 10])
   24.7861

   2.0072e+03

   2.0541e+05

   2.7652e+09

   3.4711e+11

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   Inf


>> step_potential(5, [1 10 15 25 30 40 5 5 5 0.5 10])
   24.7861

   2.0072e+03

   2.0541e+05

   2.7652e+09
```

```
   3.4711e+11

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

>> step_potential(5, [1 10 15 25 30 40 5 5 5 1 1])
   24.7861

   2.0072e+03

   2.0541e+05

   2.7652e+09

   3.4711e+11

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

   5.7379e+15

>> step_potential(5, [1 10 15 25 30 40 5 5 5 1 100000])
   24.7861

   2.0072e+03

   2.0541e+05
```

2.7652e+09

3.4711e+11

5.7379e+15

5.7379e+15

5.7379e+15

5.7379e+15

5.7379e+15

Inf

```
temp_sol = evalc('ySol');




        % temp_str_one= str2num(temp_str_one);
        % temp_str_one = str2double(regexp(temp_str_one,'\d*','match'));




        % temp_str_one=str2func(append("@(u)", temp_str_one))

        % intermediate
        syms u;
        result = integral(str2sym(temp_str_two));
        disp(result)
```

```
func_temp = str2double(temp_str_one);
```

```
    % temp_str_two = double(temp_str_two);
    temp_str_two = str2double(temp_str_two);
    temp_str_two = sym(temp_str_two);
```

```
disp('string output')
        disp(class(temp_str_one))
        disp(temp_str_one)
        disp(length(temp_str_one))
        % place all contents of string in first entry
        temp_str_one=convertCharsToStrings(temp_str_one);
        disp(class(temp_str_one))
        % disp(temp_str_one)
        % temp_str_one=str2double(temp_str_one);
        syms u;
        % temp_str_one = append("@(u)", temp_str_one);
        disp(temp_str_one)
        syms x;
        % temp_str_one = str2sym(char(temp_str_one));
        syms u;
        syms x;
        func_temp = str2double(temp_str_one);
        func_temp = matlabFunction(str2sym(temp_str_one));
        integral_term_one = integral(func_temp, 0 , 1, 'RelTol',0,'AbsTol',1e-12);
        disp('printing first term')
        disp(integral_term_one)
```

```
    % repeat
    temp_str_three = temp_sol(temp_pos(temp_index+2)+1:temp_pos(temp_index+3)-3);
    % array_temp(temp_index+3) = str2num(temp_pos(temp_index+2)+1:temp_pos(temp_index+3)-3);
    x_1 = strfind(temp_str_three,'(');
    x_2 = strfind(temp_str_three,',');
    temp_str_three = temp_str_three(x_1(1)+1:x_2(1)-1);
    temp_str_three=convertCharsToStrings(temp_str_three);
    disp('third integral term')
    disp(temp_str_three)
    temp_str_three = matlabFunction(str2sym(temp_str_three));
```

```
syms u;
integral_term_three = integral(temp_str_three,0,1 , 'RelTol',0,'AbsTol',1e-12);
disp(integral_term_three)
% array_temp(temp_index+3) = integral_term_three;

% replace u with v for variable of integration
u_positions = strfind(temp_str_four, 'u');
x_1 = strfind(temp_str_four,'(');
x_2 = strfind(temp_str_four,'*');
temp_str_four = (temp_str_four(x_1(1)+1:x_2(1)+1));
disp(temp_str_four)
temp_str_four_2=temp_str_four;
```

```
for posi=1:length(u_positions)
    temp_str_four_2(posi) = v;
end
% convert temp_str_four_2
temp_str_four_2=convertCharsToStrings(temp_str_four_2);
disp(temp_str_four_2)
temp_str_four_2 = matlabFunction(str2sym(temp_str_four_2));
```

```
1125899906842624 sqrt(pi) erf| u - - | |
      |                           \    2 /

exp| - -------------------------------------- |
      \            3569847868129149          /
```

```
if isequal(A,1)
cond_1 = y(0)==0;
cond_2 = Dy(0)==1;
conds_temp = [cond_1 cond_2];
ySol = dsolve(ode,conds_temp);
else
cond_1 = y(0)==1;
cond_2 = Dy(0)==(1);
conds_temp = [cond_1 cond_2];
ySol = dsolve(ode,conds_temp);
end
```

```
ySol = dsolve(ode,conds_temp);
```

```
pretty(ySol)
y = sym(ySol);

disp('sol')
% process string contents of solution
```

```
cond_1 = y(0)== num2str(time_vec(A));
```

cond_2 = Dy(0)==1;

```
def brownian(x0, n, dt, delta, out=None):

   x0 = np.asarray(x0)

   # For each element of x0, generate a sample of n numbers from a
   # normal distribution.
   r = norm.rvs(size=x0.shape + (n,), scale=delta*sqrt(dt))

   # If `out` was not given, create an output array.
   if out is None:
       out = np.empty(r.shape)

   # This computes the Brownian motion by forming the cumulative sum of
   # the random samples.
   np.cumsum(r, axis=-1, out=out)

   # Add the initial condition.
   out += np.expand_dims(x0, axis=-1)

   return out

# The Wiener process parameter.
# delta = 0.25

# Total time.
T = 10.0
# Number of steps.
N = 50
# Time step size
# dt = T/N

Count_array_2 = np.linspace(1,10,100)

# Initial values of x and array to
# store Brownian Bridge samples

def prep_brownian(Count_array,N,T,delta):
```

```
   Sample_array = np.zeros([len(Count_array_2), N])

   for X in range(len(Count_array)):

       x = np.empty((2,N+1))
       # declare initial value of the
       # Brownian sampling process.

       # x0 = np.asarray(Count_array[X])
       x[:, 0] = Count_array[X-1]

       Z = brownian(x[:,0], N, T/N,delta, out=x[:,1:])
       print(Z)
       Sample_array[X] = Z[1]
   return Sample_array

Sample_array = prep_brownian(Count_array_2, N,T/N,0.01)



# take absolute value of sample displacement

def plot_sample_distribution():
   # to avoid issues with passing an array as input to a function,
   # directly call the brownian sampling procedure above within the
   # function
   Sample_array = prep_brownian(Count_array_2, N,T/N,0.01)
   test_array = { }
   # concat all samples together, also looking to do this with append
   Sample_cat = np.concatenate([Sample_array[1], Sample_array[2], Sample_array[3],
Sample_array[4], Sample_array[5], Sample_array[6], Sample_array[7], Sample_array[8],
Sample_array[9], Sample_array[10]])
   for I in np.linspace(1,len(Sample_cat), len(Sample_cat)):
       I = I+1
       if I<(len(Sample_cat)):
           test_array[I] = Sample_cat[I] - Sample_cat[I-1]
       else:
           test_array[I] = 0

   # after for loop, get values from dict
   x = test_array.values()
   # convert list to numpy array
   x_empty = np.zeros(len(x))
```

```python
    for J in range(len(x)):
        x_empty[J] = x[J]

    x_abs = abs(x_empty)

    # next, proceed to plot the distributions
    # of Brownian samples in the signed
    # and unsigned cases

    fig1 = plt.figure()
    ax1 = fig1.add_subplot(222)
    ax1.plot(x, np.linspace(1,len(x),len(x)))
    ax1.set_xlim([-0.01,0.015])
    ax1.set_title('sampling distribution behavior ')
    ax1.set_xlabel(' nearest neighbor sample variation ')
    ax1.set_ylabel(' sample number ')

    # fig2 = plt.figure()
    ax2 = fig1.add_subplot(221)
    ax2.plot(x_empty, np.linspace(1,len(x_empty),len(x_empty)))
    ax2.set_xlim([ -0.01, 0.015])
    plt.show()

    return test_array,x_abs


Test_Arr = plot_sample_distribution()

def compute_couplings(Test_arry):
    # define the revised couplings
    # as from the PDFs in posts
    Test_arry = Test_arry[0].values()
    # maximum number of nonzero entries in coupling short range
    # is 49, corresponding to 50-1 entries using 1 as a reference
    couplings_short = np.empty([(len(Count_array_2)/10) * N, N])
    # on the other hand, there are
    couplings_long = np.empty([(len(Count_array_2)/10) * 450,450])

    time_normalization_short_couplings = np.empty([(len(Count_array_2)/10) * N, N])
    exponential_power_short_couplings = np.empty([(len(Count_array_2)/10) * N, N])

    # sep_array = np.empty([10,len(Test_arry)/10])
```

```
# temp = np.zeros([ 1 , 50 ])

# times at which each sample is collected
time_array_short= np.zeros([N,1])
time_array_long = np.zeros([((len(Count_array_2)/10)-1)* N, 1])

for KI in range(len(time_array_short)):
    time_array_short[KI]= 0 + ((T/N) * KI)

for KJ in range(len(time_array_long)):
    time_array_long[KJ] = 0 + ((T/N) * KJ)

short_time_sum = sum(time_array_short)
long_time_sum = sum(time_array_long)

for I in range(int(len(Count_array_2)/10)):

    # obtain the bridge samples for each increment
    # of 50 steps taken per box (will automate this
    # to allow for arbitrary steps in the future)

    temp = Test_arry[(50*I)+1:50*(I+1)]
    # obtain remaining bridge samples rather
    # than those in temp vector above

    tmp_array = Test_arry

    for AB in range(len(np.linspace((50*I)+1 , 50*(I+1), (50*(I+1)-
((50*I)+1)-1)))):
        # modify Test_arry to place a 0 in each
        # entry that is captured in temp array given above
        tmp_array[AB] = tmp_array[AB] * 0

    # obtain indices of nonzero elements in Test_arry_tmp

    tmp_array = np.asarray(tmp_array)

    tmp_array = tmp_array[np.nonzero(tmp_array)]

    # proceed to compute the couplings from temp and Test_arry_tmp
    # samples, in addition to the corresponding time_array_short
```

```
    # and time_array_long objects defined outside of the loop, resp

    for YI in range(len(temp)):
        # for init in range(len(temp[YI])):
        temp_2=temp[YI:]
        for YK in range(len(temp_2)):

            # float(time_array_short[YI]) -
float(time_array_short[YK]))))/(float(long_time_sum))* m.exp(-float(temp[YI]) -
float(temp_2[YK]))

            couplings_short[YI][YK] = (1)/(long_time_sum) * m.exp(-1)
            # store individual components of the short couplings into separate
arrays,
            # so that we can determine which component of the coupling is the most
            # numerically dominant

            time_normalization_short_couplings[YI][YK] =  1
            # (time_array_short[YI] - time_array_short[YK])/(long_time_sum)
            exponential_power_short_couplings[YI][YK] = 1
            # m.exp(temp[YI] - temp_2[YK])


            # couplings_short[YI][YK] = float((abs(float(time_array_short[YI]) -
float(time_array_short[YK])))) * m.exp(-(float(temp[YI]) - float(temp_2[YK])))

    # perform the analagous array assignments for the long range couplings,
    # which involve the normalisation by the time_array_long sum variable

    for YJJ in range(len(tmp_array)):
        # for init in range(len(tmp_array[YJJ])):
            tmp_array_2 = tmp_array[YJJ:]
            for YKJ in range(len(tmp_array_2)):
                couplings_long[YJJ][YKJ] =  m.exp(-(float(tmp_array[YJJ]) -
float(tmp_array_2[YKJ])))

                # couplings_long[YJJ][YKJ] = float((abs(float(time_array_long[YJJ])
- float(time_array_long[YKJ]))/(float(long_time_sum)) * m.exp(-(float(tmp_array[YJJ])
- float(tmp_array_2[YKJ])))))
                # float((abs(float(time_array_long[YJJ]) -
float(time_array_long[YKJ]))
```

```python
fig11 = plt.figure()
ax31 = fig11.add_subplot(221)
ax31.plot(np.linspace(1,len(couplings_short), len(couplings_short)) ,
couplings_short)
ax31.set_title('plotting the short range couplings')
ax31.set_xlabel(' discretization grid point ')
ax31.set_ylabel(' coupling ')

ax32 = fig11.add_subplot(222)
ax32.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax32.set_title('plotting the long range couplings')
ax32.set_xlabel(' discretization grid point ')
ax32.set_ylabel(' coupling ')
ax32.set_xlim([0,500])

ax33=fig11.add_subplot(223)
ax33.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax33.set_title('fractal structure, example 1')
ax33.set_xlabel(' discretization grid point ')
ax33.set_ylabel(' coupling ')
ax33.set_xlim([200,300])
ax33.set_ylim([0.0005,0.0020])

ax43=fig11.add_subplot(224)
ax43.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax43.set_title('fractal structure, example 2')
ax43.set_xlabel(' discretization grid point ')
ax43.set_ylabel(' coupling ')
ax43.set_xlim([400,460])
ax43.set_ylim([0.0032,0.0045])

plt.tight_layout()
plt.show()

# generate another figure to determine the behavior of the
# short range couplings under different time normalizations,
# in addition to different

fignew = plt.figure()

# plot individual components of the short range couplings
```

```
   ax1 = fignew.add_subplot(413)
   # np.linspace(1,len(time_normalization_short_couplings),
len(time_normalization_short_couplings))
   # time_normalization_short_couplings
   ax1.plot(1, 1)
   ax1.set_title(' dependence of the temporal order of the grid sampling point in the
short range coupling value ')
   ax1.set_xlabel(' DGP ')
   ax1.set_ylabel(' value ')
   # on the other hand, plot the other relevant quantities

   ax11 = fignew.add_subplot(413)
   # np.linspace(1,len(exponential_power_short_couplings),
len(exponential_power_short_couplings))
   # exponential_power_short_couplings
   ax11.plot(1 , 1)
   ax11.set_title(' dependence of the location of the grid sampling point in the short
range coupling value ')
   ax11.set_xlabel(' DGP ')
   ax11.set_ylabel(' value ')

   # new figure for additional zoom in subplots

   fig10 = plt.figure()

   ax01 = fig10.add_subplot(331)
   ax01.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
   ax01.set_title('ex 1')
   ax01.set_xlabel(' DGP ')
   ax01.set_ylabel(' C ')
   ax01.set_xlim([100,200])
   ax01.set_ylim([0.0005,0.0032])

   ax02 = fig10.add_subplot(332)
   ax02.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
   ax02.set_title('ex 2')
   ax02.set_xlabel(' DGP ')
   ax02.set_ylabel(' C ')
   ax02.set_xlim([300,325])
   ax02.set_ylim([0.0025, 0.0045])
```

```
ax03=fig10.add_subplot(333)
ax03.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax03.set_title('ex 3')
ax03.set_xlabel(' DGP')
ax03.set_ylabel(' C ')
ax03.set_xlim([0,25])
ax03.set_ylim([0.0005,0.0045])

ax04=fig10.add_subplot(334)
ax04.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax04.set_title('ex 4')
ax04.set_xlabel(' DGP ')
ax04.set_ylabel(' C ')
ax04.set_xlim([40,100])
ax04.set_ylim([0.00000001,0.0045])

ax05=fig10.add_subplot(335)
ax05.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax05.set_title('ex 5')
ax05.set_xlabel(' DGP')
ax05.set_ylabel(' C ')
ax05.set_xlim([200,225])
ax05.set_ylim([0.0010,0.0020])

ax06=fig10.add_subplot(336)
ax06.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax06.set_title('ex 6')
ax06.set_xlabel(' DGP ')
ax06.set_ylabel(' C ')
ax06.set_xlim([275,300])
ax06.set_ylim([0.0010,0.0030])

ax07=fig10.add_subplot(337)
ax07.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
ax07.set_title('ex 7')
ax07.set_xlabel(' DGP ')
ax07.set_ylabel(' C ')
ax07.set_xlim([400,425])
ax07.set_ylim([0.0020,0.0045])

ax06=fig10.add_subplot(338)
ax06.plot(np.linspace(1,len(couplings_long), len(couplings_long)) , couplings_long)
```

```
   ax06.set_title('ex 8')
   ax06.set_xlabel(' DGP ')
   ax06.set_ylabel(' C ')
   ax06.set_xlim([340,430])
   ax06.set_ylim([0.00002,0.0045])

   ax06=fig10.add_subplot(339)
   ax06.plot(np.linspace(1,len(couplings_long), len(couplings_long)), couplings_long)
   ax06.set_title('ex 9')
   ax06.set_xlabel(' DGP ')
   ax06.set_ylabel(' C ')
   ax06.set_xlim([200,300])
   ax06.set_ylim([0.001,0.0020])


   plt.tight_layout()
   plt.show()

   return couplings_long, time_array_short, time_array_long, len(time_array_short),
len(time_array_long)


Couplings_temp = compute_couplings(Test_Arr)
print(Couplings_temp[0])
# print(Couplings_temp[1])



# compute accompanying values of the analytical solution to N-S
# by constructing analytical solutions with appropriate choices of
# the free parameters alpha, beta, nu, and c (as given in Cases 2.1
# and 2.2)

def f1():

   x_vel_pts = np.linspace(1,10,1000)
   y_vel_pts = np.linspace(1,10,1000)

   # CASE 1: plot example of the x and y
   # components of the velocity
   # from an analytical N-S solution

   B = 0
```

```
    A = 1
    c = 1
    alpha = 2
    beta = - 2
    # nu = 1

    x_vel_arr = np.zeros([len(x_vel_pts), len(x_vel_pts)])
    y_vel_arr = np.zeros([len(y_vel_pts), len(y_vel_pts)])
    vec_total = np.zeros([len(x_vel_pts), len(x_vel_pts)])

    fudge = 0.5

    # generate x and y components for velocity
    for IA in range(len(np.linspace(1,10,100))):
        for IB in range(len(np.linspace(1,10,100))):
            x_vel = A * np.exp(fudge * ((alpha * x_vel_pts[IB]) + (beta *
y_vel_pts[IA]))) + B
            x_vel_arr[IA][IB] = x_vel
            y_vel = (1/beta) * (c- (alpha * B) - (alpha * A * np.exp(fudge*((alpha *
x_vel_pts[IB])+(beta * y_vel_pts[IA])))))
            y_vel_arr[IA][IB] = y_vel

            # compute the total velocity at each point
            vec_total[IA][IB] = np.sqrt((x_vel * x_vel) + (y_vel * y_vel))

    # plot the value of the pressure at
    # each point. For this discretization of
    # the box between [1,10] and [1,10], we
    # know that every 10 points signal a new
    # side of a disjoint box in the rectangular
    # region. We have evaluated the analytic
    # solution at more points over which we
    # sampled.
    fig3 = plt.figure()
    ax3 = fig3.add_subplot(221)
    ax3.plot(np.linspace(1,len(x_vel_arr),len(x_vel_arr)), x_vel_arr)
    ax3.set_title('plotting the x vel cmp')
    ax3.set_xlabel(' discretization grid point ')
    ax3.set_ylabel(' velocity ')
    ax3.set_xlim([0,200])
    ax3.set_ylim([0.6,2.7])
```

```python
# fig4 = plt.figure()
ax4 = fig3.add_subplot(222)
ax4.plot(np.linspace(1,len(y_vel_arr),len(y_vel_arr)),y_vel_arr)
ax4.set_title('plotting the y vel cmp')
ax4.set_xlabel(' discretization grid point ')
ax4.set_ylabel(' velocity ')
ax4.set_xlim([0,200])
ax4.set_ylim([-0.5,3.0])

# fig5 = plt.figure()
ax5 = fig3.add_subplot(223)
ax5.plot(np.linspace(1,len(vec_total),len(vec_total)),vec_total)
ax5.set_title('plotting the total vel cmp')
ax5.set_xlabel(' discretization grid point ')
ax5.set_ylabel(' velocity ')
ax5.set_xlim([0,200])
ax5.set_ylim([0.5,5])

x=np.linspace(1,len(x_vel_pts),len(x_vel_pts))
x4= (-1/float(alpha)) * (4 - (float(beta)*x))
x3= (-1/float(alpha)) * (3 - (float(beta)*x))
x2= (-1/float(alpha)) * (2 - (float(beta)*x))
x1= (-1/float(alpha)) * (1 - (float(beta)*x))

min4 = min(x4)
min3 = min(x3)
min2 = min(x2)
min1 = min(x1)

max4 = max(x4)
max3 = max(x3)
max2 = max(x2)
max1 = max(x1)
# plot the nonconstant pressure
# at each pt, of the form bx - ay
ax65=fig3.add_subplot(224)
# ax65.plot(x, (-1/float(alpha)) * (5 - (float(beta)*x)),c = 'y')
ax65.plot(x, x4)
# plt.hold(True)
ax65.plot(x, x3, c = 'r')
ax65.plot(x, x2, c = 'g')
ax65.plot(x, x1, c = 'b')
```

```
    # make sure that the axes are large enough for all graphs
    ax65.set_xlim([x_vel_pts[0] , x_vel_pts[len(x_vel_pts)-1]])
    ax65.set_ylim([-10,0])
    ax65.set_title('plotting level sets of the pressure')
    ax65.set_xlabel(' discretization grid point ')
    ax65.set_ylabel(' pressure ')
    # ax65.set_ylim([min(min4,min3,min2,min1), max(max4,max3,max2,max1)])
    plt.tight_layout()
    plt.show()

    return x_vel_arr, y_vel_arr, vec_total


output1 = f1()


# fudge = (c/(nu*((alpha*alpha)+ (beta*beta))))
# fudge = round(fudge,2)

def f2():

    x_vel_pts = np.linspace(1,10,100)
    y_vel_pts = np.linspace(1,10,100)

    # CASE 2: plot example of the x and y
    # components of the velocity
    # from an analytical N-S solution,
    # same as in first case but this time
    # use agreeing signs of alpha and beta.

    B = 0
    A = 1
    c = 1
    alpha = 3
    beta =  3
    # nu = 1

    x_vel_arr_2 = np.zeros([len(x_vel_pts), len(x_vel_pts)])
    y_vel_arr_2 = np.zeros([len(y_vel_pts), len(y_vel_pts)])
    vec_total_2 = np.zeros([len(x_vel_pts), len(x_vel_pts)])

    fudge = 0.5
```

```
   # generate x and y components for velocity
   for IA in range(len(np.linspace(1,10,100))):
       for IB in range(len(np.linspace(1,10,100))):
           x_vel_2 = A * np.exp(fudge * ((alpha * x_vel_pts[IB]) + (beta *
y_vel_pts[IA]))) + B
           x_vel_arr_2[IA][IB] = x_vel_2
           y_vel_2 = (1/float(beta)) * (float(c)- (float(alpha) * float(B)) -
(float(alpha) * A * np.exp(float(fudge)*((float(alpha) * x_vel_pts[IB])+(float(beta) *
y_vel_pts[IA])))))
           y_vel_arr_2[IA][IB] = y_vel_2

           # compute the total velocity at each point
           vec_total_2[IA][IB] = np.sqrt((x_vel_2 * x_vel_2) + (y_vel_2 * y_vel_2))
   fig4 = plt.figure()
   ax6 = fig4.add_subplot(221)
   ax6.plot(np.linspace(1,len(x_vel_arr_2),len(x_vel_arr_2)),x_vel_arr_2)
   ax6.set_title('plotting the x vel cmp')
   ax6.set_xlabel(' discretization grid point ')
   ax6.set_ylabel(' velocity ')
   ax6.set_xlim([60,100])

   # fig7 = plt.figure()
   ax7 = fig4.add_subplot(222)
   ax7.plot(np.linspace(1,len(y_vel_arr_2),len(y_vel_arr_2)),y_vel_arr_2)
   ax7.set_title('plotting the y vel cmp')
   ax7.set_xlabel(' discretization grid point ')
   ax7.set_ylabel(' velocity ')
   # ax3.set_xlim([-0.01,0.015])

   # fig8 = plt.figure()
   ax8 = fig4.add_subplot(223)
   ax8.plot(np.linspace(1,len(vec_total_2),len(vec_total_2)),vec_total_2)
   ax8.set_title('plotting the total vel cmp')
   ax8.set_xlabel(' discretization grid point ')
   ax8.set_ylabel(' velocity ')
   ax8.set_xlim([60,100])

   x=np.linspace(1,len(x_vel_pts),len(x_vel_pts))
   x4= (-1/float(alpha)) * (4 - (float(beta)*x))
   x3= (-1/float(alpha)) * (3 - (float(beta)*x))
   x2= (-1/float(alpha)) * (2 - (float(beta)*x))
```

```
    x1= (-1/float(alpha)) * (1 - (float(beta)*x))

    ax16=fig4.add_subplot(224)
    # ax65.plot(x, (-1/float(alpha)) * (5 - (float(beta)*x)),c = 'y')
    ax16.plot(x, x4)
    # plt.hold(True)
    ax16.plot(x, x3, c = 'r')
    ax16.plot(x, x2, c = 'g')
    ax16.plot(x, x1, c = 'b')
    # make sure that the axes are large enough for all graphs
    ax16.set_xlim([x_vel_pts[0] , x_vel_pts[len(x_vel_pts)-1]])
    ax16.set_ylim([-10,0])
    ax16.set_title('plotting level sets of the pressure')
    ax16.set_xlabel(' discretization grid point ')
    ax16.set_ylabel(' pressure ')
    ax16.set_ylim([-2,10])
    ax16.set_xlim([1,3])

    plt.tight_layout()
    plt.show()

    return x_vel_arr_2, y_vel_arr_2, vec_total_2

output2 = f2()
# print(output2[1])
# print(len(output1[1]))

def f3():

    x_vel_pts = np.linspace(1,10,100)
    y_vel_pts = np.linspace(1,10,100)

    # CASE 3: plot example of the x and y
    # components of the velocity
    # from an analytical N-S solution,
    # same as in first case but this time
    # use largely disagreeing signs of alpha and beta.

    B = 0
    A = 1
    c = 1
    alpha = 500
```

```
   beta =  1
   # nu = 1

   x_vel_arr_3 = np.zeros([len(x_vel_pts), len(x_vel_pts)])
   y_vel_arr_3 = np.zeros([len(y_vel_pts), len(y_vel_pts)])
   vec_total_3 = np.zeros([len(x_vel_pts), len(x_vel_pts)])

   fudge = 0.5

   # generate x and y components for velocity
   for IA in range(len(np.linspace(1,10,100))):
       for IB in range(len(np.linspace(1,10,100))):
           x_vel_3 = A * np.exp(fudge * ((alpha * x_vel_pts[IB]) + (beta *
y_vel_pts[IA]))) + B
           x_vel_arr_3[IA][IB] = x_vel_3
           y_vel_3 = (1/float(beta)) * (float(c)- (float(alpha) * float(B)) -
(float(alpha) * float(A) * np.exp(float(fudge)*((float(alpha) *
x_vel_pts[IB])+(float(beta) * y_vel_pts[IA])))))
           y_vel_arr_3[IA][IB] = y_vel_3

           # compute the total velocity at each point
           vec_total_3[IA][IB] = np.sqrt((x_vel_3 * x_vel_3) + (y_vel_3 * y_vel_3))
   fig5 = plt.figure()
   ax9 = fig5.add_subplot(221)
   ax9.plot(np.linspace(1,len(x_vel_arr_3),len(x_vel_arr_3)),x_vel_arr_3)
   ax9.set_title('plotting the x vel cmp')
   ax9.set_xlabel(' discretization grid point ')
   ax9.set_ylabel(' velocity ')
   # ax3.set_xlim([-0.01,0.015])

   # fig10 = plt.figure()
   ax10 = fig5.add_subplot(222)
   ax10.plot(np.linspace(1,len(y_vel_arr_3),len(y_vel_arr_3)),y_vel_arr_3)
   ax10.set_title('plotting the y vel cmp')
   ax10.set_xlabel(' discretization grid point ')
   ax10.set_ylabel(' velocity ')
   # ax3.set_xlim([-0.01,0.015])

   # fig11 = plt.figure()
   # ax11 = fig5.add_subplot(223)
   # ax11.plot(np.linspace(1,len(vec_total_3),len(vec_total_3)),vec_total_3)
   # ax11.set_title('plotting the total vel cmp')
```

```
    # ax11.set_xlabel(' discretization grid point ')
    # ax11.set_ylabel(' velocity ')
    # ax11.set_xlim([-2,10])

    x=np.linspace(1,len(x_vel_pts),len(x_vel_pts))
    x4= (-1/float(alpha)) * (4 - (float(beta)*x))
    x3= (-1/float(alpha)) * (3 - (float(beta)*x))
    x2= (-1/float(alpha)) * (2 - (float(beta)*x))
    x1= (-1/float(alpha)) * (1 - (float(beta)*x))

    ax15=fig5.add_subplot(223)
    # ax65.plot(x, (-1/float(alpha)) * (5 - (float(beta)*x)),c = 'y')
    ax15.plot(x, x4)
    # plt.hold(True)
    ax15.plot(x, x3, c = 'r')
    ax15.plot(x, x2, c = 'g')
    ax15.plot(x, x1, c = 'b')
    # make sure that the axes are large enough for all graphs
    ax15.set_xlim([x_vel_pts[0] , x_vel_pts[len(x_vel_pts)-1]])
    ax15.set_ylim([0,0.05])
    ax15.set_xlim([40,60])
    ax15.set_title('plotting level sets of the pressure')
    ax15.set_xlabel(' discretization grid point ')
    ax15.set_ylabel(' pressure ')
    ax15.set_ylim([0.03, 0.06])
    plt.tight_layout()
    plt.show()

    return x_vel_arr_3, y_vel_arr_3, vec_total_3


output3 = f3()

def f4():

    x_vel_pts = np.linspace(1,10,100)
    y_vel_pts = np.linspace(1,10,100)

    # CASE 3: plot example of the x and y
    # components of the velocity
    # from an analytical N-S solution,
    # same as in first case but this time
```

```
    # use largely disagreeing signs of alpha and beta.

    B = 0
    A = 1
    c = 1
    alpha = -1000
    beta =  1
    # nu = 1

    x_vel_arr_4 = np.zeros([len(x_vel_pts), len(x_vel_pts)])
    y_vel_arr_4 = np.zeros([len(y_vel_pts), len(y_vel_pts)])
    vec_total_4 = np.zeros([len(x_vel_pts), len(x_vel_pts)])

    fudge = 0.5

    # generate x and y components for velocity
    for IA in range(len(np.linspace(1,10,100))):
        for IB in range(len(np.linspace(1,10,100))):
            x_vel_4 = A * np.exp(fudge * ((alpha * x_vel_pts[IB]) + (beta *
y_vel_pts[IA]))) + B
            x_vel_arr_4[IA][IB] = x_vel_4
            y_vel_4 = (1/float(beta)) * (float(c)- (float(alpha) * float(B)) -
(float(alpha) * float(A) * np.exp(float(fudge)*((float(alpha) *
x_vel_pts[IB])+(float(beta) * y_vel_pts[IA])))))
            y_vel_arr_4[IA][IB] = y_vel_4

            # compute the total velocity at each point
            vec_total_4[IA][IB] = np.sqrt((x_vel_4 * x_vel_4) + (y_vel_4 * y_vel_4))
    fig6 = plt.figure()
    ax10 = fig6.add_subplot(221)
    ax10.plot(np.linspace(1,len(x_vel_arr_4),len(x_vel_arr_4)),x_vel_arr_4)
    ax10.set_title('plotting the x vel cmp')
    ax10.set_xlabel(' discretization grid point ')
    ax10.set_ylabel(' velocity ')
    # ax3.set_xlim([-0.01,0.015])

    # fig11 = plt.figure()
    ax11 = fig6.add_subplot(222)
    ax11.plot(np.linspace(1,len(y_vel_arr_4),len(y_vel_arr_4)),y_vel_arr_4)
    ax11.set_title('plotting the y vel cmp')
    ax11.set_xlabel(' discretization grid point ')
    ax11.set_ylabel(' velocity ')
```

```
    # ax3.set_xlim([-0.01,0.015])

    # fig12 = plt.figure()

    ax12 = fig6.add_subplot(223)
    ax12.plot(np.linspace(1,len(vec_total_4),len(vec_total_4)),vec_total_4)
    ax12.set_title('plotting the total vel cmp')
    ax12.set_xlabel(' discretization grid point ')
    ax12.set_ylabel(' velocity ')
    # ax4.set_xlim([-0.01,0.015])

    x=np.linspace(1,len(x_vel_pts),len(x_vel_pts))
    x4= (-1/float(alpha)) * (4 - (float(beta)*x))
    x3= (-1/float(alpha)) * (3 - (float(beta)*x))
    x2= (-1/float(alpha)) * (2 - (float(beta)*x))
    x1= (-1/float(alpha)) * (1 - (float(beta)*x))

    ax17=fig6.add_subplot(224)
    # ax65.plot(x, (-1/float(alpha)) * (5 - (float(beta)*x)),c = 'y')
    ax17.plot(x, x4)
    # plt.hold(True)
    ax17.plot(x, x3, c = 'r')
    ax17.plot(x, x2, c = 'g')
    ax17.plot(x, x1, c = 'b')

    # make sure that the axes are large enough for all graphs
    ax17.set_xlim([1,4])
    ax17.set_ylim([-0.003,0.0000001])
    ax17.set_title('plotting level sets of the pressure')
    ax17.set_xlabel(' discretization grid point ')
    ax17.set_ylabel(' pressure ')
    # ax17.set_ylim([0,10])
    # ax17.set_xlim([0,3])

    plt.tight_layout()
    plt.show()

    return x_vel_arr_4, y_vel_arr_4, vec_total_4

output4 = f4()
```

```
# enforce a partition of ONE POSSIBLE region R
# x_values = np.linspace(1,10,10)
# y_values = np.linspace(1,15,10)

# subdivide each of the x and y values into
# further partitions

# new_array_1 = np.zeros([10,10])

# for X in range(len(x_values)):
   # create partitions of x coordinates
   # of each box

# new_array_2 = np.zeros([15,10])

# for Y in range(len(y_values)):
   # create partitions of y coordinates
   # of each box
```

```
if isequal(A,1)
syms y(x);
cond_1 = y(0)==0;
cond_2 = Dy(0)==1;
conds_temp = [cond_1 cond_2];
ySol = dsolve(ode,conds_temp);
else
disp(time_vec(A))
% x = uint8(time_vec(A));
ySol_next = dsolve(ode);
ySol = ySol_next;
end
```