Chapter_2

May 27, 2019

```
In [6]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy import optimize
        from scipy.optimize import minimize
        from scipy.spatial.distance import directed_hausdorff
```

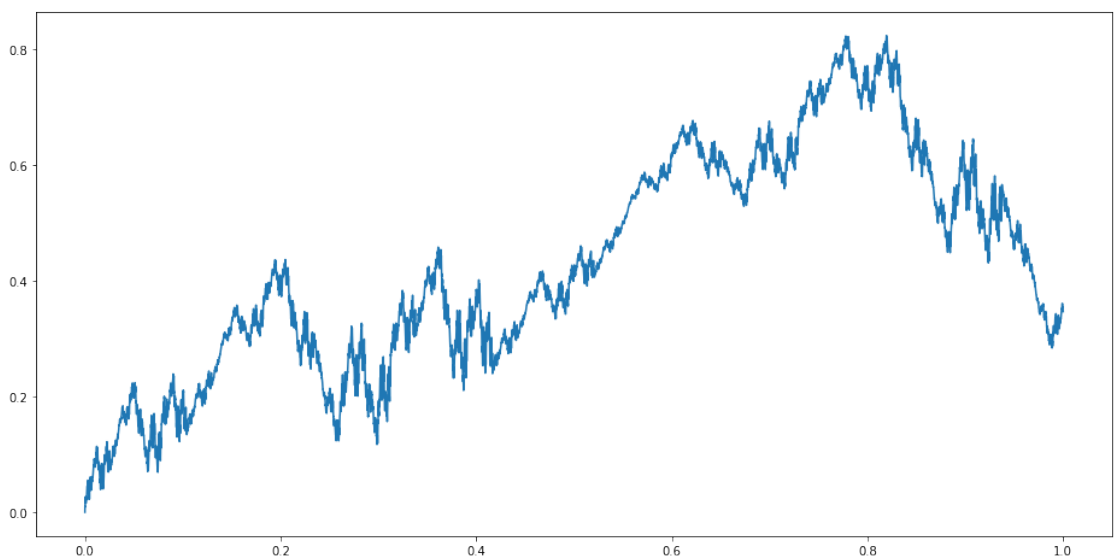
Import the given data and plot it

```
In [8]: A_data = np.load('A.npy')
        b_data = np.load('b.npy')
        d_data = np.load('d.npy')

        data_full = np.load('rough_data.npy')
        data = data_full[data_full[:,0]<=1,:]

        plt.figure(figsize=(16,8))
        plt.plot(data[:,0],data[:,1])
```

```
Out[8]: [<matplotlib.lines.Line2D at 0x2230ec1af98>]
```



Make a function that finds a FIF for a given data set

```
In [9]: def inter_gen(X,Y,d,inc_a = False):

            '''
            input:
            Y,X     - N+1 by 1 numpy array as data points
            d       - N by 1 numpy array as votility factors
            '''

            assert(X.shape == Y.shape)

            n = X.shape[0]

            N = d.shape[0]

            assert(n-1 == N)

            A = []

            b = []

            a = []

            for i in range(1,n):

                A.append(np.array([[ X[i] - X[i-1], 0  ], [Y[i] - Y[i-1] - d[i-1]*Y[-1], d[i-1]

                b.append(np.array([  X[i-1]   , Y[i-1]  ],dtype=float).reshape((2,1)))

                a.append(X[i] - X[i-1])

            #if you want to return the a values for dimension calculations
            if inc_a:

                return A,b,a

            return A,b
```

An implimentation of the Chaos game for Affine maps

```
In [10]: def chaos(A,b,p = None ,it =100,burn = 5):

             '''
             Implimentation of the Chaos game for affine maps
             Imput:
             A     - n by n matrix for IFS
             b     - n by 1 matrix for IFS
             p     - probabilities for maps
```

```
    '''

    assert(len(A) == len(b))

    N = len(A)

    #assign uniform probabilites if not specified
    if p == None:

        p = np.ones(N)/N


    z = np.random.rand(2).reshape([2,1])

    Z = np.zeros((it,2))

    for i in range(it):

        prob = np.random.rand()

        P = 0

        for q in range(N):

            P += p[q]

            if prob < P:


                z = A[q]@z + b[q]

                if i>burn:

                    Z[i,:] = z.flatten()

                break

    return Z
```

Make a function that evaluates the constants given in the tex document

```
In [11]: def cons(X,Y,Sx,Sy,i):

             '''
             input:
             Y,X     - n by 1 numpy array as data points
             Sx,Sy   - N by 1 locations of the interpolation points
             i       - which partition wants to be calculated
```

```
    '''

    Z = X[(Sx[i]>X)*(X>Sx[i-1])]

    W = Y[(Sx[i]>X)*(X>Sx[i-1])]

    c0 = 0

    c1 = 0

    c2 = 0

    for j in range(len(Z)):

        c0 += (Sy[i-1] + ((Z[j]-Sx[i-1])*(Sy[i]-Sy[i-1]))/(Sx[i]-Sx[i-1]) - W[j])**2

        c1 += 2*(Sy[i-1] + ((Z[j]-Sx[i-1])*(Sy[i]-Sy[i-1]))/(Sx[i]-Sx[i-1]) - W[j])*(S

        c2 += (Sy[-1]*((Z[j]-Sx[i-1])/(Sx[i]-Sx[i-1])) - Y[np.abs(X - (Z[j]-Sx[i-1])/


    return c0,c1,c2
```

Function that finds the optimal d_i that in unconstrained

```
In [12]: def find_d_i(X,Y,Sx,Sy,i):

             c0,c1,c2 = cons(X,Y,Sx,Sy,i)

             return 0.5*c1/c2
```

Example provided in thesis

```
In [13]: X_ex = np.array([0,0.1    ,0.5,0.6,1]).reshape((5,1))
         Y_ex = np.array([0,-0.1   ,1   ,1   ,0]).reshape((5,1))
         Sx_ex = np.array([0,0.1,1]).reshape((3,1))
         Sy_ex = np.array([0,-0.1,0]).reshape((3,1))
         cons(X_ex,Y_ex,Sx_ex,Sy_ex,2)
         find_d_i(X_ex,Y_ex,Sx_ex,Sy_ex,2)

Out[13]: array([1.05])
```

Function that selects a subset of n points

```
In [14]: def red_res(n,Z):

             res = np.linspace(0,1,n+1)
```

```
    boo = (Z[:,0]==0)

    for i in range(1,len(res)):

        boo += (Z[:,0]==min(Z[:,0], key=lambda x:abs(x-res[i])))

    return  Z[boo,:]
```

Function that finds the unconstrained solution to the optimisation

```
In [15]: def uncon_sol(data,N):

             S = red_res(N,data)

             Sx = S[:,0]

             Sy = S[:,1]

             Xd = data[:,0]

             Yd = data[:,1]

             d = np.zeros((len(Sx)-1,1))

             c = np.zeros((len(Sx)-1,3))

             for i in range(1,len(Sx)):

                 di = find_d_i(Xd,Yd,Sx,Sy,i)

                 d[i-1] = np.sign(di)*np.min([np.abs(di),0.95])

                 c[i-1,:] =  cons(Xd,Yd,Sx,Sy,i)

             return Sx,Sy,d,c
```

Unconstrained solution with 16 maps

```
In [16]: Sx,Sy,d_un,c_un = uncon_sol(data,16)
         s = np.sign(d_un)
         d_un

Out[16]: array([[ 0.25107463],
                [-0.02708155],
                [-0.01287294],
                [ 0.08754844],
                [-0.097674  ],
                [ 0.31655053],
                [-0.02924108],
```

```
       [-0.01322763],
       [-0.07045796],
       [ 0.0328213 ],
       [-0.1909197 ],
       [-0.09893547],
       [-0.01666951],
       [ 0.04348496],
       [-0.08118093],
       [-0.19134702]])
```

```
In [17]: def obj_fun_un(d):
             return np.dot(c_un[:,2],d**2) - np.dot(c_un[:,1],d) + np.sum(c_un[:,0])

         def obj_der_un(d):
             return 2*d*c_un[:,2]-c_un[:,1]
```

Check this is correct with a numerical optimiser

```
In [18]: x0 = np.zeros(len(d_un))
         res_un = minimize(obj_fun_un, x0, jac=obj_der_un )
```

Is correct

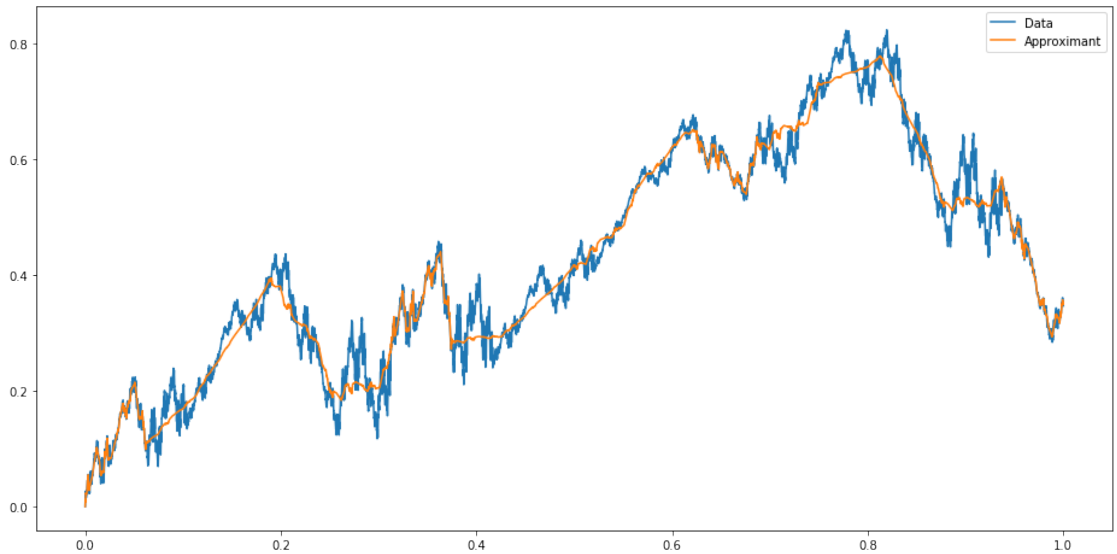
```
In [19]: print(res_un.x)
         print(d_un.flatten())
```

```
[ 0.25107463 -0.02708155 -0.01287294  0.08754844 -0.097674    0.31655053
 -0.02924108 -0.01322763 -0.07045796  0.0328213  -0.1909197  -0.09893547
 -0.01666951  0.04348496 -0.08118093 -0.19134702]
[ 0.25107463 -0.02708155 -0.01287294  0.08754844 -0.097674    0.31655053
 -0.02924108 -0.01322763 -0.07045796  0.0328213  -0.1909197  -0.09893547
 -0.01666951  0.04348496 -0.08118093 -0.19134702]
```

Plot unconstrained solution with initial data

```
In [20]: A_un, b_un , a_un= inter_gen(Sx,Sy,d_un, inc_a = True)
         app_un = chaos(A_un, b_un ,p = None ,it =10000)
         app_un = app_un[app_un[:,0].argsort()]
         plt.figure(figsize=(16,8))
         plt.plot(data[:,0],data[:,1], label = 'Data')
         plt.plot(app_un[:,0],app_un[:,1],label = 'Approximant')
         plt.legend()
```

```
Out[20]: <matplotlib.legend.Legend at 0x2230e935b00>
```



In the example above, the approximant does a good job at describing the data in the L^2 sense. However, it does not capture the 'fractal' nature of the object as evinced through the following calculation.

```
In [21]: def dims(a,d,D):
             d0 = d.flatten()
             aD = np.power(a,D-1)
             return np.dot(d0.flatten(),aD.flatten())-1

In [22]: minimize( lambda x : (dims(a_un,np.abs(d_un),x))**2, 1.5).x

Out[22]: array([1.16063787])
```

The dimension from this descriptive model is nothing near the actual dimension of the data of approximately 1.538. The following approximant will match this dimension.

```
In [23]: D_approx  = 1.538

         C = c_un[:,1]*s.flatten()

         K = np.power(a_un,D_approx-1)

         def obj_fun(d):
             return np.dot(c_un[:,2],d**2) - np.dot(C,d) + np.sum(c_un[:,0])

         def obj_der(d):
             return 2*d*c_un[:,2]-C

         ineq_cons = {'type': 'ineq',
                      'fun' : lambda x: np.hstack([(x.flatten()),(1-x.flatten())]) ,
```

```
                'jac' : lambda x: np.vstack([np.diag(np.abs(s).flatten()),np.diag(-np.abs
eq_cons = {'type': 'eq',
           'fun' : lambda x: np.array([dims(np.array(a_un),x,D_approx)]),
           'jac' : lambda x: K }
```

In [24]:
```
x0 = x0 = np.abs(d_un)
res = minimize(obj_fun, x0, method='SLSQP', jac=obj_der,constraints=[eq_cons, ineq_co
print(d_un.flatten())
print(res.x*s.flatten())
```

```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 34.43859959182272
            Iterations: 3
            Function evaluations: 5
            Gradient evaluations: 3
[ 0.25107463 -0.02708155 -0.01287294  0.08754844 -0.097674    0.31655053
 -0.02924108 -0.01322763 -0.07045796  0.0328213  -0.1909197  -0.09893547
 -0.01666951  0.04348496 -0.08118093 -0.19134702]
[ 0.43128383 -0.20729075 -0.19308214  0.26775764 -0.27788321  0.49675974
 -0.20945028 -0.19343683 -0.25066716  0.21303051 -0.3711289  -0.27914468
 -0.19687871  0.22369417 -0.26139014 -0.37155622]
```

In [25]:
```
A_con, b_con = inter_gen(Sx,Sy,res['x']*s.flatten())
Z_con = chaos(A_con, b_con ,p = None ,it =10000)
```

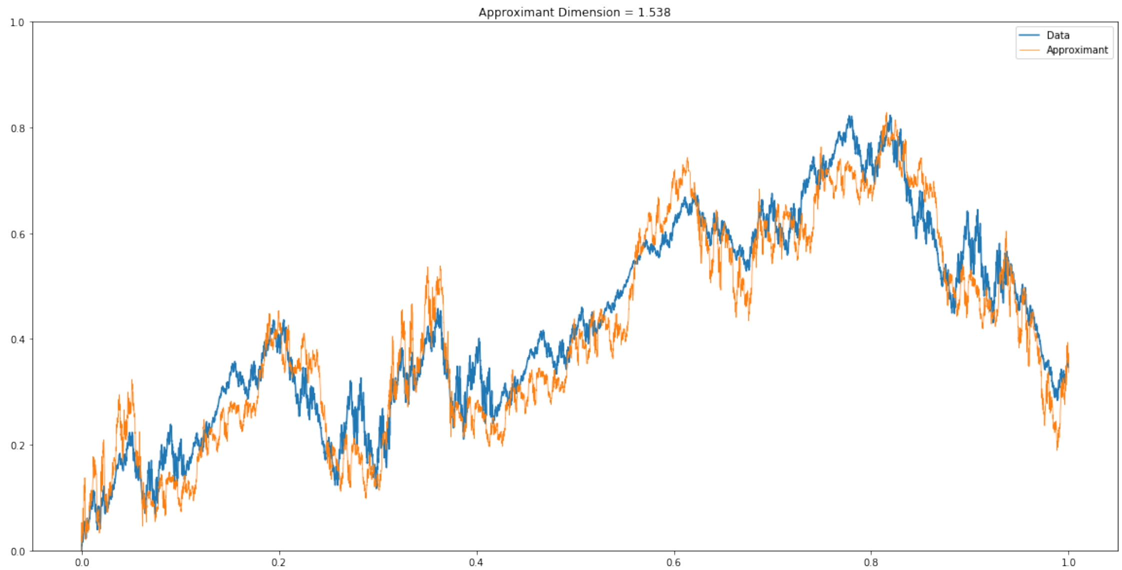
In [26]:
```
Z_con = Z_con[Z_con[:,0].argsort()]

plt.figure(figsize = (20,10))

plt.plot(data[:,0],data[:,1], label = 'Data')

plt.ylim(0,1)
plt.plot(Z_con[:,0],Z_con[:,1], label = "Approximant",linewidth = 0.7)
plt.title("Approximant Dimension = " + str(D_approx))
plt.legend()
```

Out[26]:
```
<matplotlib.legend.Legend at 0x2230eb5ebe0>
```



This approximation 'looks' good by being close in the L^2 sense and matching the dimension of the data. The Hausdorff distance between these two can be calculated.

```
In [27]: max(directed_hausdorff(data,Z_con)[0], directed_hausdorff(Z_con,data)[0])
```

```
Out[27]: 0.09466496138064823
```

Putting the above working into a single function yields:

```
In [28]: def novel_apx(data,N,D):

             Sx,Sy,d_un,c_un = uncon_sol(data,N)
             s = np.sign(d_un)

             if D == 0:

                 A,b = inter_gen(Sx,Sy,d_un)

                 return A,b

             a_un = np.ones(N)/N

             def dims(a,d,D,s):
                 d0 = d.flatten()
                 aD = np.power(a,D-1)
                 return np.dot(d0.flatten(),aD.flatten())-1

             D_approx = D

             C = c_un[:,1]*s.flatten()
```

```
        K = np.power(a_un,D_approx-1)

        def obj_fun(d):
            return np.dot(c_un[:,2],d**2) - np.dot(C,d) + np.sum(c_un[:,0])

        def obj_der(d):
            return 2*d*c_un[:,2]-C

        ineq_cons = {'type': 'ineq',
                     'fun' : lambda x: np.hstack([(x.flatten()),(1-x.flatten())]) ,
                     'jac' : lambda x: np.vstack([np.diag(np.abs(s).flatten()),np.diag(-np
        eq_cons = {'type': 'eq',
                   'fun' : lambda x: np.array([dims(np.array(a_un),x,D_approx,s)]),
                   'jac' : lambda x: K }


        x0 = x0 = np.abs(d_un)
        res = minimize(obj_fun, x0, method='SLSQP', jac=obj_der,constraints=[eq_cons, ineq

        A, b = inter_gen(Sx,Sy,res['x']*s.flatten())

        print('d values:')
        print(res['x']*s.flatten())

        return A,b
```

```
In [37]: N = 6
         D = 1.538

         A,b = novel_apx(data,N,D)

         Z = chaos(A, b ,p = None ,it =10000)

         Z = Z[Z[:,0].argsort()]

         dH = np.round(max(directed_hausdorff(data,Z)[0], directed_hausdorff(Z,data)[0]),3)

         plt.figure(figsize = (16,9))

         plt.plot(data[:,0],data[:,1], label = 'Data')

         plt.ylim(0,1)
         plt.plot(Z[:,0],Z[:,1], label = "Approximant",linewidth = 1)
         plt.title("Approximant Dimension = " + str(D) +'     ' + 'Hausdorff Distance = ' + str

         plt.legend(fontsize=20)
         plt.savefig('6.pdf')
```

```
        print('Hausdorff Distance:')
        print(dH)
```

```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 99.99978644000196
            Iterations: 5
            Function evaluations: 13
            Gradient evaluations: 5
d values:
[ 0.34847    -0.50338299 -0.42419202  0.51709689  0.46722665 -0.36170804]
Hausdorff Distance:
0.223
```