Chapter_3

May 27, 2019

```
In [1]: import numpy as np
        from scipy.optimize import least_squares
        import matplotlib.pyplot as plt
        import time
        from PIL import Image
        import random
        from scipy import optimize
        import sympy as sp
```

Firstly the notation functions as described in chapter 2, named accrodingly

```
In [2]: def tau(n,m):
            return ((n+m)+1)*(n+m)/2 + m

        def tau_inv(k):

            def fl(k):
                return np.floor((1+np.sqrt(1+8*k))/2) - 1

            i = fl(k)- (k - (fl(k)*(fl(k)+1)/2) )
            j = k - (fl(k)*(fl(k)+1)/2)

            return i,j

        def add_inv(n):

            ans = np.zeros((n+1,2))

            for i in range(len(ans)):

                ans[i,0] = i
                ans[i,1] = n - i

            return ans

        def trinomial ( i, j, k ):
```

```
    #This function was taken from the internet

    #  Licensing:
    #
    #    This code is distributed under the GNU LGPL license.
    #
    #  Modified:
    #
    #    11 April 2015
    #
    #  Author:
    #
    #    John Burkardt
    #

        i = int(i)
        j = int(j)
        k = int(k)

        from sys import exit

        if ( i < 0 or j < 0 or k < 0 ):
            print ( '' )
            print ( 'TRINOMIAL - Fatal error!' )
            print ( '  Negative factor encountered.' )
            exit ( 'TRINOMIAL - Fatal error!' )

        value = 1

        t = 1

        for l in range ( 1, i + 1 ):
            #   value = value * t // l
            t = t + 1

        for l in range ( 1, j + 1 ):
            value = value * t // l
            t = t + 1

        for l in range ( 1, k + 1 ):
            value = value * t // l
            t = t + 1

        return value
```

We may construct the matrix A_r for an affine function now

```
In [3]: def A_r(a,b,c,d,e,f,N=6,rational= False):
```

```
        A = np.zeros((N,N))

        if rational:

            A = sp.zeros(N,N)


        for u in range(N):
            for v in range(N):

                n,m = tau_inv(u)

                i_i, j_j = tau_inv(v)

                i_i = add_inv(int(i_i))

                j_j = add_inv(int(j_j))

                for i in range(i_i.shape[0]):

                    for j in range(j_j.shape[0]):

                        if (i_i[i,0] + j_j[j,0]<=n):

                            if (i_i[i,1] + j_j[j,1]<=m):

                                i = int(i)

                                j = int(j)

                                B = trinomial( i, j, n-i-j )*trinomial( int(i_i[i,1]), int


                                C = a**i * c**int(i_i[i,1]) * b**j * d**int(j_j[j,1]) * e*

                                A[u,v] += int(B)*C

        return A
```

Test this function in floating point.

```
In [4]: A_r(1/3,0,1/2,1/2,0,0)

Out[4]: array([[1.        , 0.        , 0.        , 0.        , 0.        ,
                0.        ],
               [0.        , 0.33333333, 0.        , 0.        , 0.        ,
                0.        ],
               [0.        , 0.5       , 0.5       , 0.        , 0.        ,
```

```
        0.        ],
       [0.        , 0.        , 0.        , 0.11111111, 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , 0.16666667, 0.16666667,
        0.        ],
       [0.        , 0.        , 0.        , 0.25      , 0.5       ,
        0.25      ]])
```

Now in rational arithmetic

```
In [5]: A_r(sp.Rational(1,3),0,sp.Rational(1,2),sp.Rational(1,2),0,0,N=6,rational=True)

Out[5]: Matrix([
        [1,   0,   0,   0,   0,   0],
        [0, 1/3,   0,   0,   0,   0],
        [0, 1/2, 1/2,   0,   0,   0],
        [0,   0,   0, 1/9,   0,   0],
        [0,   0,   0, 1/6, 1/6,   0],
        [0,   0,   0, 1/4, 1/2, 1/4]])
```

```
In [6]: def Phi(a_l,b_l,c_l,d_l,e_l,f_l,p_l,it = 1,n = 6,rational = False):

            assert len(a_l) == len(b_l) and len(a_l) == len(p_l)

            mat = np.zeros((n,n))

            if rational:

                mat = sp.zeros(n,n)

            N = len(a_l)

            for i in range(N):

                mat = p_l[i]*A_r(a_l[i],b_l[i],c_l[i],d_l[i],e_l[i],f_l[i],N=n,rational = ratio

            if not rational:

                mat = np.linalg.matrix_power(mat,it)

            return mat
```

```
In [7]: def moments(a_l,b_l,c_l,d_l,e_l,f_l,p_l,it = 20,n=6,direct = False):

            if direct:

                tmp = Phi(a_l,b_l,c_l,d_l,e_l,f_l,p_l,it = 1,n=n, rational = True) - sp.eye(n)

                tmp = tmp.nullspace()
```

```
            if len(tmp) != 0:
                'Fire!! Nullspace larger than expected'
            ans = (tmp[0])/tmp[0][0]
            return ans
        return Phi(a_l,b_l,c_l,d_l,e_l,f_l,p_l,it = it,n=n)@np.ones(n)
```

Cantor set moments calculated iteratively and in rational arithmetic

```
In [8]: moments([1/3,1/3],[0,0],[0,0],[0,0],[0,2/3],[0,0],[1/2,1/2],it = 30)

Out[8]: array([1.    , 0.5   , 0.    , 0.375, 0.    , 0.    ])

In [9]: moments([sp.Rational(1,3),sp.Rational(1,3)],[0,0],[0,0],[0,0],[0,sp.Rational(2,3)],[0,(

Out[9]: Matrix([
        [  1],
        [1/2],
        [  0],
        [3/8],
        [  0],
        [  0]])
```

Now we have a function to compute the Chaos Game and the picture of our fractal measure

```
In [10]: def chaos(IFS,prob,it = 10000,imgxy = 480):

             N = len(prob)

             image = Image.new("RGB", (imgxy, imgxy),"white")

             xa = -0.2
             xb = 1.2
             ya = -0.2
             yb = 1.2

             #starting values
             x=np.random.rand()#0.5
             y=np.random.rand()#0.5

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

             for i in range(it):
```

```
        p = random.random()

        P = prob[0]

        for j in range(N):

            if p < P:


                x0 = IFS[j]*x + IFS[(N) + j]*y + IFS[4*(N) + j]

                y = IFS[2*(N) + j]*x + IFS[3*(N) + j]*y + IFS[5*(N) + j]

                x = x0

                X[i,0] = x
                X[i,1] = y


                if x>xa and x<xb and y>ya and y<yb:

                    jx = int((x - xa) / (xb - xa) * (imgxy - 1))
                    jy = (imgxy - 1) - int((y - ya) / (yb - ya) * (imgxy - 1))

                    if j == 0:
                        image.putpixel((jx, jy), (255,0,0,255))
                    elif j == 1:
                        image.putpixel((jx, jy), (255,255,0,255))
                    else:
                        image.putpixel((jx, jy), (0,0,255,255))


                break

            else:
                P = P + prob[j+1]

    plt.show(image)

    return image, X
```

Get function to generate Generalised Sierpinski Triangle

```
In [11]: def GSP(a = 1.1, b = 0.85, par = 'FFF', imgxy = 480, it = 50000, IFS = False):

             #co ordinates of triangles third vertex
             Cy=(float)(0.5*np.sqrt(-1 + 2*a**2 - a**4 + 2*b**2 + 2*a**2*b**2 - b**4))
             Cx=(float)(0.5*(1 - a**2 + b**2))
```

```
#angles for roatation
cosA=(float)(Cx/b)
sinA=(float)(Cy/b)
cosB=(float)((1.0-Cx)/a)
sinB=(float)(Cy/a)

#view box
xa = min(-0.2,Cx-0.2)
xb = max(1.2,Cx+0.2)
ya =-0.2
yb = max(1.2,Cy+0.2)

#starting values
x=0.0
y=0.0

#image
image = Image.new("RGB", (imgxy, imgxy),"white")

if par == 'FFF':

    alphaFFF=(float)(-(-1 + a**2 - b**2))/(float)(2*b)
    betaFFF=(float)(-(-1 - a**2 + b**2))/(float)(2*a)
    gammaFFF=(float)(-(1 - a**2 - b**2))/(float)(2*a*b)

    detaFFF=alphaFFF*alphaFFF
    detbFFF=betaFFF*betaFFF
    detgFFF=gammaFFF*gammaFFF
    pnormFFF=(detaFFF+detbFFF+detgFFF)
    paFFF=(float)(detaFFF)/(float)(pnormFFF)
    pbFFF=(float)(detbFFF)/(float)(pnormFFF)
    pgFFF=(float)(detgFFF)/(float)(pnormFFF)

    FFF=[[cosA*alphaFFF,sinA*alphaFFF,sinA*alphaFFF,-cosA*alphaFFF,0.0,0.0,paFFF]

    mat=FFF

    def f(d) :
        return [alphaFFF**d[0]+betaFFF**d[0]+gammaFFF**d[0]-1]

    alpha = alphaFFF

if par == 'FFN':

    alphaNFF=(float)(b)/float((a**2 + b**2))
    betaNFF=(float)(a)/(float)(a**2 + b**2)
    gammaNFF=-1.0*(float)(1.0-a**2-b**2)/(float)(a**2 + b**2)
```

```
        detaNFF=alphaNFF*alphaNFF
        detbNFF=betaNFF*betaNFF
        detgNFF=gammaNFF*gammaNFF
        pnormNFF=(detaNFF+detbNFF+detgNFF)
        paNFF=(float)(detaNFF)/(float)(pnormNFF)
        pbNFF=(float)(detbNFF)/(float)(pnormNFF)
        pgNFF=(float)(detgNFF)/(float)(pnormNFF)

        NFF=[[cosA*alphaNFF,sinA*alphaNFF,sinA*alphaNFF,-cosA*alphaNFF,0.0,0.0,paNFF]

        mat=NFF

        def f(d) :
            return [alphaNFF**d[0]+betaNFF**d[0]+gammaNFF**d[0]-1]

        alpha = alphaFFN

    if par == 'FNN':

        alphaNNF=(float)(b/(1.0 + b**2))
        betaNNF=(float)(1.0/(1.0 + b**2))
        gammaNNF=(float)(b**2/(1.0 + b**2))


        detaNNF=alphaNNF*alphaNNF
        detbNNF=betaNNF*betaNNF
        detgNNF=gammaNNF*gammaNNF
        pnormNNF=(detaNNF+detbNNF+detgNNF)
        paNNF=(float)(detaNNF)/(float)(pnormNNF)
        pbNNF=(float)(detbNNF)/(float)(pnormNNF)
        pgNNF=(float)(detgNNF)/(float)(pnormNNF)

        NNF=[[cosA*alphaNNF,sinA*alphaNNF,sinA*alphaNNF,-cosA*alphaNNF,0.0,0.0,paNNF]

        mat=NNF

        def f(d) :
            return [alphaNNF**d[0]+betaNNF**d[0]+gammaNNF**d[0]-1]

        alpha = alphaNNF

    if par == 'NNN':
        NNN=[[0.5,0.0,0.0,0.5,0.0,0.0,0.33333],[0.5,0.0,0.0,0.5,0.5,0,0.33333],[0.5,0

        mat=NNN

        def f(d) :
```

```
            return [(0.5)**d[0]+(0.5)**d[0]+(0.5)**d[0]-1]

        alpha = 1/2


    for k in range(it):

        p=random.random()
        if p <= mat[0][6]:
            i=0
        elif p <= mat[0][6] + mat[1][6]:
            i=1
        else:
            i=2

        x0 = x * mat[i][0] + y * mat[i][1] + mat[i][4]
        y  = x * mat[i][2] + y * mat[i][3] + mat[i][5]
        x = x0
        jx = int((x - xa) / (xb - xa) * (imgxy - 1))
        jy = (imgxy - 1) - int((y - ya) / (yb - ya) * (imgxy - 1))

        if i==2:
             image.putpixel((jx, jy), (255,0,0,255))
        elif i==1:
             image.putpixel((jx, jy), (255,255,0,255))
        elif i==0:
             image.putpixel((jx, jy), (0,0,255,255))

    if IFS:

        return mat

    return image
```

Make a function so that the format of IFS is the same

```
In [12]: def convertIFS(IFS_mat):

    N = len(IFS_mat)

    ans = np.zeros(N*6)

    prob = np.zeros(N)

    r = 0
    for j in range(7):
        for i in range(N):
            if j != 6:
```

```
                    ans[r]=IFS_mat[i][j]
                    r = r+1
                prob[i] = IFS_mat[i][6]

        return ans,prob
```

```
In [13]: IFS_STEEM, STEEM_P = convertIFS(GSP(IFS = True))
```

```
In [14]: IFS_STEEM
```

```
Out[14]: array([ 0.09088452,  0.45716038, -0.4519551 ,  0.28744481, -0.49816139,
               -0.21071657,  0.28744481, -0.49816139, -0.21071657, -0.09088452,
               -0.45716038,  0.4519551 ,  0.        ,  0.54283962,  0.54283962,
                0.        ,  0.49816139,  0.49816139])
```

```
In [15]: STEEM_P
```

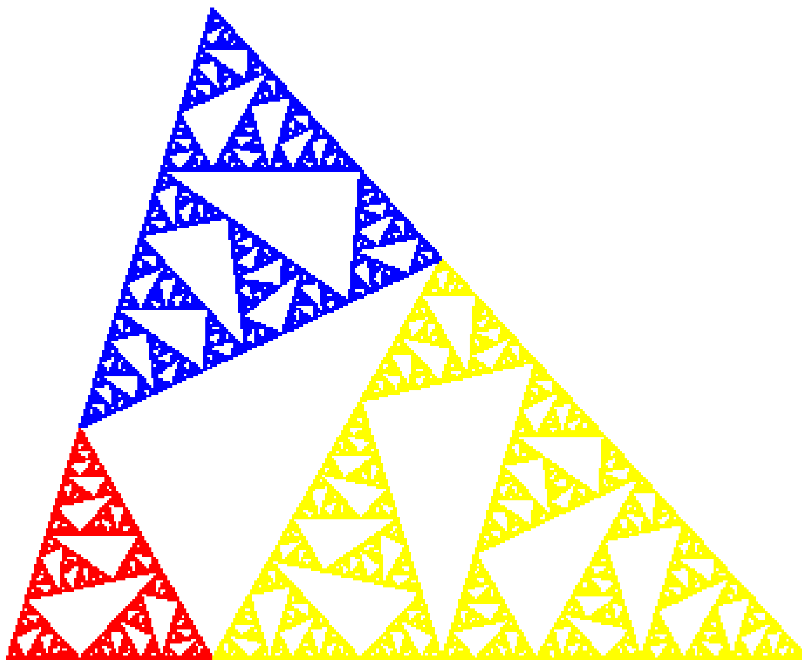
```
Out[15]: array([0.11407481, 0.57381043, 0.31211477])
```

```
In [27]: img, X_STEEM  = chaos(IFS_STEEM,prob = STEEM_P,it = 1000000)
```

The above maps produce 'uniform measure' on a Steemson triangle

```
In [28]: img
```

```
Out[28]:
```



Make a function that takes the IFS parameters from GSP and puts them into the moment format

```
In [29]: def mom_wrap(x,prob,it = 20,n=21,direct = False):

             assert len(x)%6==0

             N = len(x)

             a_l = x[:N//6]

             b_l = x[1*(N//6):2*(N//6)]

             c_l = x[2*(N//6):3*(N//6)]
```

```
        d_l = x[3*(N//6):4*(N//6)]

        e_l = x[4*(N//6):5*(N//6)]

        f_l = x[5*(N//6):]

        p_l =  prob

        return moments(a_l,b_l,c_l,d_l,e_l,f_l,p_l,it = it,n=n,direct = direct)
```

Compute the moments in both the fixed and iterative way

```
In [30]: STEEM_DIRECT_M = np.array(mom_wrap(IFS_STEEM, STEEM_P,direct = True,n=21))
         STEEM_DIRECT_M
```

```
Out[30]: array([[1.00000000000000],
                [0.457999968063630],
                [0.260960601210938],
                [0.271009962111420],
                [0.0995842785556266],
                [0.113191196637661],
                [0.184106973373980],
                [0.0474427803890055],
                [0.0393197038207555],
                [0.0587510249123115],
                [0.135876998941933],
                [0.0264760327157712],
                [0.0162611286845916],
                [0.0193435942525520],
                [0.0335681438678902],
                [0.105854773140229],
                [0.0165114393936808],
                [0.00780212122482904],
                [0.00729221872965016],
                [0.0106822908345115],
                [0.0203948804197367]], dtype=object)
```

```
In [31]: mom_wrap(IFS_STEEM, STEEM_P,it = 20, n = 21)
```

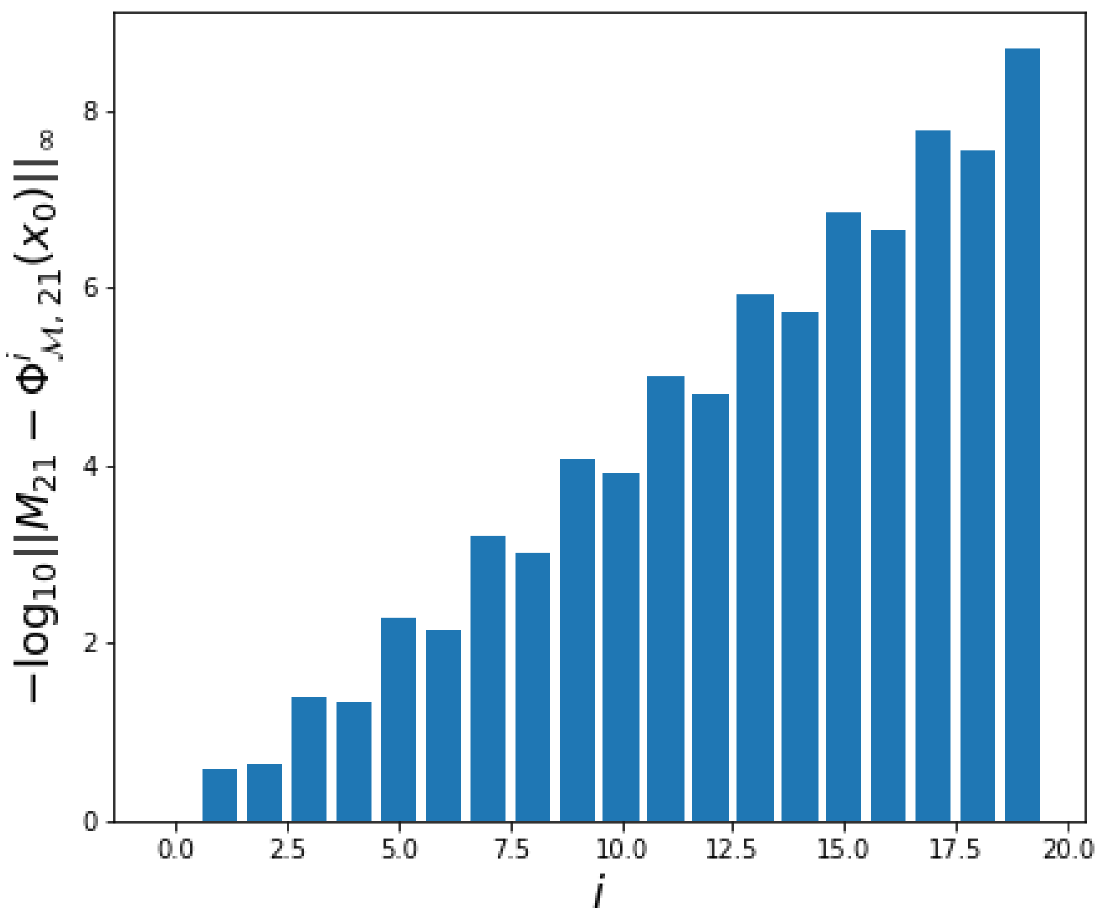
```
Out[31]: array([1.        , 0.45799997, 0.2609606 , 0.27100996, 0.09958428,
                0.1131912 , 0.18410698, 0.04744278, 0.0393197 , 0.05875103,
                0.135877  , 0.02647603, 0.01626113, 0.01934359, 0.03356814,
                0.10585478, 0.01651144, 0.00780212, 0.00729222, 0.01068229,
                0.02039488])
```

```
In [32]: plot = np.zeros(20)

         for i in range(20):
```

```
        error = np.max(np.abs(STEEM_DIRECT_M.T - mom_wrap(IFS_STEEM, STEEM_P,it = i, n = 2
        plot[i] = error
```

```
In [42]: plt.figure(figsize=(7,6))
         plt.ylabel('$- \log_{10} \|| M_{21} -  \Phi_{ \mathcal{M}, 21}^i (x_0) \||_{\infty} $
         plt.xlabel("$i$",size = 18)
         plt.bar((range(20)),-np.log10(plot))
```



Evaluate the integral with Eltons Theorem

```
In [34]: def elt(X,size = 21):
             ans = np.zeros(size)
             for k in range(size):
```

```
            i , j = tau_inv(k)

            ans[k] = np.sum(np.power(X[:,0],i)*np.power(X[:,1],j),axis = 0)/len(X)

        return ans
```

In [35]:
```
elt(X_STEEM,size = 21)
```

Out[35]:
```
array([1.        , 0.45787645, 0.26093047, 0.27087659, 0.09955831,
       0.11314294, 0.18396728, 0.04742562, 0.03929397, 0.05871955,
       0.13573601, 0.02646494, 0.01624755, 0.01932706, 0.03355238,
       0.10571572, 0.01650395, 0.00779515, 0.0072837 , 0.01067321,
       0.02038933])
```

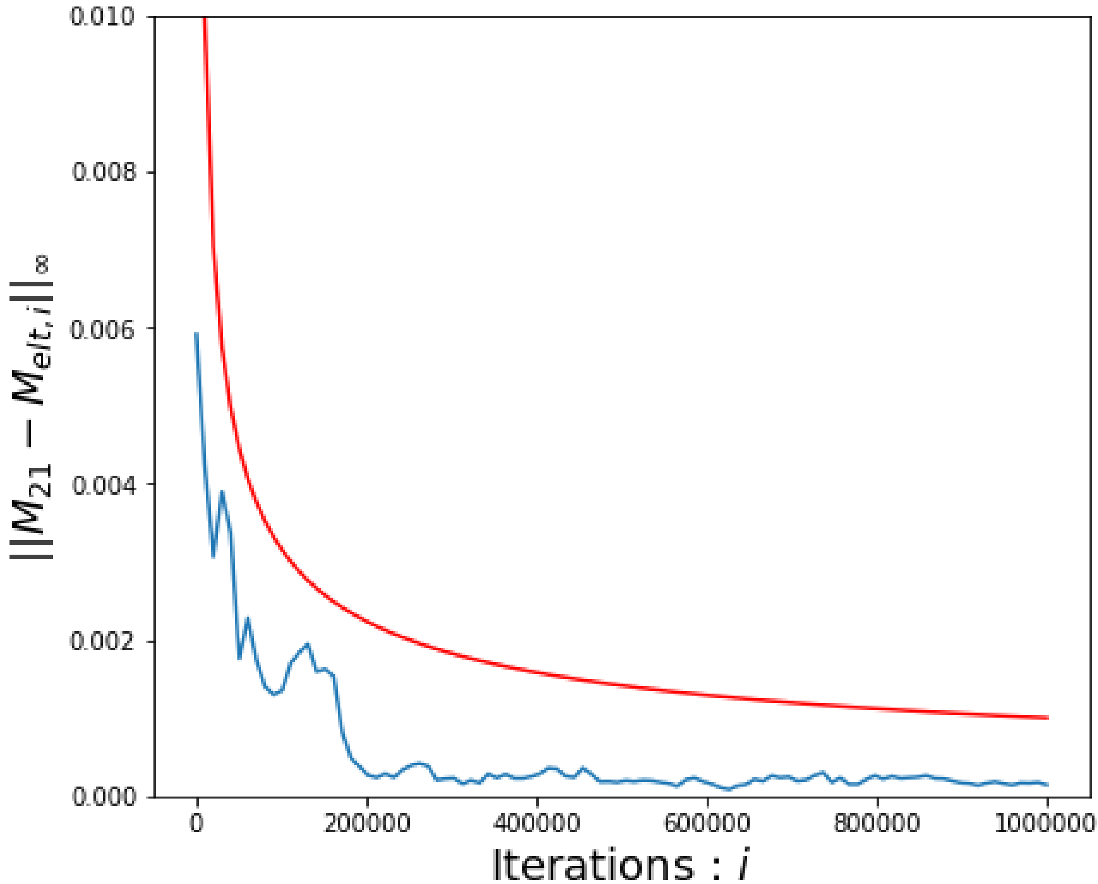
In [36]:
```
plot_elt = np.zeros((100,21))

for i in range(100):

    plot_elt[i,:] = elt(X_STEEM[:10000*(i+1)],size = 21)
```

In [43]:
```
t = np.linspace(1,1000000,100)

plt.figure(figsize=(7,6))
plt.ylim(0,0.01)
plt.plot(t,1/np.sqrt(t),color = 'red')
plt.xlabel('Iterations : $i$',size = 18)
plt.ylabel('$\|| M_{21} - M_{elt,i} \|\|_{\infty}$',size = 18)
plt.plot(t,(np.max(np.abs(plot_elt - STEEM_DIRECT_M.T),axis = 1)))
```



0.010
0.008
0.006
0.004
0.002
0.000
$||M_{21} - M_{e/t,i}||_\infty$
0
200000
400000
600000
800000
1000000
Iterations : i