Derivative based graph learning in P-regime

In [1]:
```
%pylab inline
import matlab
import matlab.engine
MATLAB = matlab.engine.start_matlab()
```

```
Populating the interactive namespace from numpy and matplotlib
```

P regime, N=5 Nodes

The set of functions M for P-regime of networks:

$$M = (M_0, M_1, M_2) = (-(\cdot)^{0.5}, 1, (\cdot)^{0.2})$$

Central equation for node i:

$$\frac{dx_i}{dt} = -x_i^{0.5} + \sum_{j=1}^{N} A_{ij} x_j^{0.2}$$

Vectorized and using power notation:

$$\frac{d\vec{x}}{dt} = -\vec{x}^{0.5} + I_{N\times N} A \vec{x}^{0.2}$$

We will evolve the system as per the above equation with the following adjacency matrix:

In [2]:
```
N = 5
A_gt = rand(N,N)
for i in range(N): A_gt[i,i] = 0
A_gt = maximum(A_gt, A_gt.T)
```

In [3]:
```
A_gt, A_gt - A_gt.T
# checking if the ground truth matrix is symmetric
```

Out[3]:
```
(array([[0.        , 0.43422646, 0.61203273, 0.95795378, 0.83636024],
        [0.43422646, 0.        , 0.78180358, 0.96370395, 0.78387813],
        [0.61203273, 0.78180358, 0.        , 0.46567556, 0.81813326],
        [0.95795378, 0.96370395, 0.46567556, 0.        , 0.34677407],
        [0.83636024, 0.78387813, 0.81813326, 0.34677407, 0.        ]]),
 array([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]))
```

In [4]:
```
dt = 0.01
x0 = abs(randn(N,))
x0
#  setting up initial conditions and the time resolution
```

Out[4]:
```
array([0.26848081, 0.81428136, 0.12614298, 0.47199161, 0.86306178])
```

In [5]:
```
x = x0
Traj = []

for i in range(1000):
    Traj.append(x)
    x = x + (-(abs(x))**(0.5) + dot(A_gt,x**(0.2)))*dt
```

Issue

The value of the graph signal depends on the previous update, and the 'how' is dictated by the differential equation But the first term, $\vec{x}^{0.5}$ will throw some errors in our simulation if any of the components of $\vec{x}$ are negative.

A quick fix was to take the absolute value before applying the square root.

In [6]:
```
xp = [Traj[i][0] for i in range(1000)]
x_0 = matlab.double(xp)
MATLAB.plot(x_0)
```

Out[6]:
```
<matlab.object at 0x1ae8b079a90>
```

In [7]:
```
x1 =  [Traj[i][1] for i in range(1000)]
x_1 = matlab.double(x1)

x2 =  [Traj[i][2] for i in range(1000)]
x_2 = matlab.double(x2)

x3 =  [Traj[i][3] for i in range(1000)]
x_3 = matlab.double(x3)

x4 =  [Traj[i][4] for i in range(1000)]
x_4 = matlab.double(x4)
```

In [8]:
```
MATLAB.workspace['ex0'] = x_0
MATLAB.workspace['ex1'] = x_1
MATLAB.workspace['ex2'] = x_2
MATLAB.workspace['ex3'] = x_3
MATLAB.workspace['ex4'] = x_4
```

In [9]:
```
MATLAB.eval("var1 = [ex0; ex1; ex2; ex3; ex4];",nargout = 0)
MATLAB.eval("for k = 1:5 subplot(3,2,k); plot(var1(k,:)); end",nargout = 0)
```

We will now try to learn back the coefficients of A using some Matrix inversion;

For different samples in `Traj` which track the movement of the signal at all nodes, we will take N such samples and compute the finite difference based derivative and set it equal to the affine RHS of the central equation evaluated at that point in time (we will use left hand derivative)

In [10]:
```
samples = [Traj[9*i] for i in range(99)]
derivs = [(samples[i+1] - samples[i])/(9*dt) for i in range(98)]
# larger timescale used
```

We have many equations of the form

$$\frac{\vec{x}_{i+1} - \vec{x}_i}{t_{i+1} - t_i} = -\sqrt{\vec{x}_i} + A\sqrt[5]{\vec{x}_i}$$

$$\frac{\vec{x}_{i+1}-\vec{x}_i}{t_{i+1}-t_i}+\sqrt{\vec{x}_i}=A\sqrt[5]{\vec{x}_i}$$

$$\vec{v}_i = A\vec{u}_i$$

To solve this, we can simply adjoin different instances of the above so that it forms:

$$V = AU$$

where U, V are both NxN matrices (columns of U,V follow the first equation) This means, using N instances of first eq we can try to get back the Matrix A.

$$U = [\vec{u}_1\vec{u}_2\dots\vec{u}_N]$$

$$V = [\vec{v}_1\vec{v}_2\dots\vec{v}_N]$$

In [11]:
```
M = 80
V = zeros((N,M))
U = zeros((N,M))

for i in range(M):
    V[:,i] =  derivs[i] + sqrt(samples[i])
    U[:,i] = (samples[i])**(0.2)
```

Trying with NC2 equations instead

Trying to unravel Adjacency matrix into a vector instead:

Adjacency has symmetric structure, which means the equations would look like so:

$$A\vec{u} = \begin{bmatrix} 0 & w_{12} & w_{13}\dots w_{1N} \\ w_{21} & 0 & w_{23}\dots w_{2N} \\ \vdots & \vdots & \ddots \quad \dots & \vdots \\ w_{N1} & w_{N2} & w_{N3}\dots 0 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \end{bmatrix} = u_1\cdot\begin{bmatrix} 0 \\ w_{21} \\ w_{31} \\ \vdots \end{bmatrix} + u_2\cdot\begin{bmatrix} w_{12} \\ 0 \\ w_{32} \\ \vdots \end{bmatrix}\dots + u_N\cdot\begin{bmatrix} w_{1N} \\ \vdots \\ w_{N-1N} \\ 0 \end{bmatrix}$$

A is of order $N \times N$, but imformation it stores is captured in $\frac{N^2-N}{2}$

Unraveling the Adjacency matrix mean we only keep track of each pair of nodes, and what the weight of connecting edge is:

If we pre-index;

- 1 -> 1,2
- 1 -> 1,3
- ...
- 1 -> 1,N
- 2 -> 2,3
- 2 -> 2,4 and so on till
- N-1 -> N

How would $\vec{u}$ change accordingly? We can notice that $w_{ij} = w_{ji}$ and they only show up in the linear combination of i, jth columns, which means they are scaled by $u_i + u_j$ Thus, we can simply convert $\vec{u}_{N\times 1} \to \vec{\tilde{u}}_{\frac{N^2-N}{2}\times 1}$ where each entry of $\vec{\tilde{u}}$ follows the indexing shown above; This transformation can help us compute back the Adjacency because now we just have to produce $\vec{\tilde{u}}^T \vec{\tilde{A}}$

Where $\vec{\tilde{A}}$ = weights in indexing mentioned above;

The matrix that helps in this conversion:

Vectorizarion of a matrix

There exists an operation to convert the above equation to a more solvable form

$$V = AU$$

Following the procedure in the paper, we get:

$$\begin{aligned} \text{vech}(A) &= ((U^{\mathrm{T}} \otimes I_{N\times N})D_N)^{\dagger}\,\text{vec}(V) \\ \implies \text{vec}(A) &= D_N * \text{vech}(A) \end{aligned}$$

In [12]:
```python
class Some_Matrices():

    def vec(A):
        return A.flatten('F')

    def E_Matrices(n):
        I = eye(n)
        E = {(i,j):dot(I[:,i].reshape(n,1),I[:,j].T.reshape(1,n)) for i in range(n
        return E

    def T_Matrices(n):
        E_inst = Some_Matrices.E_Matrices(n)
        T = {(i,j): (E_inst[(i,j)] if i == j else E_inst[(i,j)] + E_inst[(j,i)]) fo
        return T

    def u_vecs(n):
        I_nh = eye(n*(n+1)//2)
        u = {(i,j): I_nh[:, int((j)*n + (i+1) - 0.5*(j+1)*j -1)] for i in range(n)
        return u

    def D_Matrix(n):
        num = n*(n+1)//2
        DT = zeros((num,n**2))
        T_inst = Some_Matrices.T_Matrices(n)
        u_inst = Some_Matrices.u_vecs(n)

        for j in range(n):
            for i in range(j,n):
                DT = DT + ((u_inst[(i,j)]).reshape(num,1)).dot(((Some_Matrices.vec
        D = DT.T
        return D

    def make_mat(c,n):
        fullc = zeros((n,n))
        for i in range(n):
            fullc[:,i] = c[i:i+n]
        return fullc
```

In [13]:
```
v = V.flatten('F')
```

In [14]:
```
D = Some_Matrices.D_Matrix(5)
hA = pinv((kron(U.T,eye(N))).dot(D)).dot(v)
vec_A_gt = Some_Matrices.vec(A_gt)
fullA = D.dot(hA)
```

In [15]:
```
ff = Some_Matrices.make_mat(fullA,N)
ff
```

Out[15]:
```
array([[0.39837109, 0.14972628, 0.44613149, 0.86696058, 0.97829816],
       [0.14972628, 0.44613149, 0.86696058, 0.97829816, 0.14972628],
       [0.44613149, 0.86696058, 0.97829816, 0.14972628, 0.1996431 ],
       [0.86696058, 0.97829816, 0.14972628, 0.1996431 , 0.82097243],
       [0.97829816, 0.14972628, 0.1996431 , 0.82097243, 1.09227643]])
```

In [16]:
```
vec_A_gt
```

Out[16]:
```
array([0.        , 0.43422646, 0.61203273, 0.95795378, 0.83636024,
       0.43422646, 0.        , 0.78180358, 0.96370395, 0.78387813,
       0.61203273, 0.78180358, 0.        , 0.46567556, 0.81813326,
       0.95795378, 0.96370395, 0.46567556, 0.        , 0.34677407,
       0.83636024, 0.78387813, 0.81813326, 0.34677407, 0.        ])
```

Getting the structure of U

Consider :

$$A\vec{u} = \vec{v}$$

where A is of the form

$$A = \begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1N-1} & a_{1N} \\ a_{12} & 0 & a_{23} & \dots & a_{2N-1} & a_{2N} \\ a_{13} & a_{23} & 0 & \dots & a_{3N-1} & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{1N-1} & a_{2N-1} & a_{3N-1} & \dots & 0 & a_{N-1N} \\ a_{1N} & a_{2N} & a_{3N} & \dots & a_{N-1N} & 0 \end{bmatrix}$$

and hence,

$$\begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1N-1} & a_{1N} \\ a_{12} & 0 & a_{23} & \cdots & a_{2N-1} & a_{2N} \\ a_{13} & a_{23} & 0 & \cdots & a_{3N-1} & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{1N-1} & a_{2N-1} & a_{3N-1} & \cdots & 0 & a_{N-1N} \\ a_{1N} & a_{2N} & a_{3N} & \cdots & a_{N-1N} & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{N-1} \\ v_N \end{bmatrix}$$

$$\implies u_1 \begin{bmatrix} 0 \\ a_{12} \\ a_{13} \\ \vdots \\ a_{1N-1} \\ a_{1N} \end{bmatrix} + u_2 \begin{bmatrix} a_{12} \\ 0 \\ a_{23} \\ \vdots \\ a_{2N-1} \\ a_{2N} \end{bmatrix} + \cdots + u_N \begin{bmatrix} a_{1N} \\ a_{2N} \\ a_{3N} \\ \vdots \\ a_{N-1N} \\ 0 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{N-1} \\ v_N \end{bmatrix}$$

Let us vectorize A as follows:

$$\vec{A} = \begin{bmatrix} a_{12} \\ a_{13} \\ \vdots \\ a_{1N} \\ a_{23} \\ a_{24} \\ \vdots \\ a_{2N} \\ a_{34} \\ \vdots \\ \vdots \\ a_{N-2N-1} \\ a_{N-2N} \\ a_{N-1N} \end{bmatrix}_{\frac{N(N-1)}{2} \times 1}$$

Now, for Au = v, we have to write u as a matrix such that

$$U_{N \times \frac{N(N-1)}{2}} \vec{A} = \vec{v}$$

We can see that the matrix follows a structure according to the previous equations (ui, uj only appear as coefficient of aij terms), thus, U can be realised as a matrix in a recursive block manner as follows

$$U = \left[\begin{array}{c|c} \mathbf{0}_{i-1 \times N-i} & {}^{N-1} \\ \hline u((i+1:N)) & \\ \hline u_i I_{N-i} & {}_{i=1} \end{array} \right]$$

```
In [17]: def U_synth(N,u):
             for i in range(1,N):

                 if i == 1:
                     uL = u[i:,:].T
                     I = u[i-1,:]*eye(N-i)
                     print(shape(uL), shape(I))
                     t = append(uL,I,axis=0)
                     U = t

                 else:
                     Z = zeros((i-1,N-i))
                     uL = u[i:,:].T
                     I = u[i-1,:]*eye(N-i)
                     t = append(Z,uL,axis=0)
                     t = append(t,I,axis=0)
                     U = append(U,t, axis=1)
             return U
```

```
In [ ]:
```