Graph Topology Inference from Regime Dynamics

Ashuthosh Bharadwaj
Signal Processing and Communications Research Centre
Student, IIIT Hyderabad, India
ashuthosh.bharadwaj@research.iiit.ac.in

I. Introduction

Graphs allow us to capture highly complicated relationships between entities. With the recent explosion in the amount of data we create on a daily basis, we can see that graphs naturally form the basis for understanding it in an organised manner, thus owing to its extensive use in modelling complicated behaviours like different regions of the brain [2], Epidemic Modelling, Protein Interactions, Social and Financial Processes.The advent of new fields like Graph Signal Processing signify our need to understand signals on irregular domains, unlike time(sound) and space(image). Networks consist of simply two components. The Nodes, which represent the different entities that are being modelled, and the Edges, which represent the connection or contextual distance between the entities.

In this report, we will deal with the learning of a randomly generated graph which follows the regime dynamics 'P', as given in [1]. This network is seen in the context of Population dynamics.

II. General Networks and their Behaviour

In [1], we see the authors define multiple types of network behaviours, which dictate how a given graph signal will propagate through it. The differences in behaviour could be seen in many ways; One being that general networks in different fields behave differently-like the fact that a social network would behave differently when an opinion update-type graph signal propagates through it, compared to a Protein network that measures the protein concentration levels in a particular setting. The authors ordered the behaviours of different types of networks by setting up a general equation for all network behaviours, and sorting by the tweaks required to the original parameters.

$$\frac{dx_i}{dt} = M_0(x_i) + \sum_{j=1}^{N} A_{ij} M_1(x_i) M_2(x_j) \tag{1}$$

(1) gives us the behaviour of any general network. It is evident that the nonlinear function triplet $\mathbf{M} = (M_0, M_1, M_2)$ controls the behaviour as they can change between different domains of networks. The function M_0 controls the self-behaviour of every node, meaning that in the event of a trivial network where no other nodes are considered, the decay/increase in that signal's value depends on itself, for example, a network that measures the concentration of a singular radioactive substance.The other two functions M_1, M_2 control the interactions between the nodes via the Adjacency coefficient, or the weight between any two nodes in the network.

The following equations give us an understanding of which parameters control the regime of behaviour.

In the paper, the authors are tasked with finding the propagation time of a signal on a graph from node j to i, denoted as $T(j \to i).\tau_i$ is the corresponding time $T(j \to i)$ where j is the node nearest to i.

$$\tau_i \sim S_i^{\theta} \tag{2}$$

Where θ is

$$\theta = -2 - \Gamma(0) \tag{3}$$

The parameter $\Gamma(0)$ is determined by the system dynamics $\mathbf{M}$, through the leading powers of the Hahn series expansion.

$$Y(R^{-1}(x)) = \sum_{n=0}^{\infty} C_n x^{\Gamma(n)} \tag{4}$$

Where where $Y(x) = (d(M_1 R)/dx)^{-1}$, $R(x) = -M_1(x)/M_0(x)$ and $R^{-1}(x)$ denotes its inverse function. Now, based on the parameter θ,

1) Distance limited Propagation $\theta = 0$
2) Degree limited Propagation $\theta > 0$
3) Composite Propagation $\theta < 0$

The regime 'P' falls in the second category, Degree limited Propagation.

III. Problem Formulation

Given signal values on an unknown Network $G = (\mathcal{V}, \mathcal{E})$ with number of nodes $N = |\mathcal{V}|$, with adjacency matrix A, we have to infer the values of the entries of A, and hence the network's structure. Let us take a look at the structure of the adjacency matrix A:

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

We observe that the diagonal entries of A are zero, owing to the fact that the network is devoid of any self-loops. The matrix is also drafted to be symmetric, which is due to our assumption that the network undirected. Now our task is concretely to get the values of the $\frac{N(N-1)}{2}$ off-diagonal entries. Representing the variables can be accomplished using the half-vectorization of the matrix, which is denoted by :

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

We use the forward euler scheme to set up a sequence of first-difference approximations of (1) at different intervals, spaced out by a small value of delta. This discretization of the equation can be realised as:

$$\frac{\Delta x_i}{\Delta t}\bigg|_{t=t_j} = \frac{x_i(t_j+\Delta t) - x_i(t_j)}{\Delta t}$$

$$\frac{x_i(t_j+\Delta t) - x_i(t_j)}{\Delta t} - M_0(x_i) = \sum_{k=1}^{N} A_{ik} M_1(x_i) M_2(x_k)$$

This allows us to sample multiple points in the "Trajectory" of the graph signal, which is the time-varying vector containing the values of the signal at each node at any given time instant. The equations can be recast in terms of the graph signal itself like so:

$$\frac{d\vec{x}}{dt} = M_0(\vec{x}) + diag(M_1(\vec{x}))AM_2(\vec{x}) \tag{5}$$

This allows us to deal with the forward scheme of all nodes simultaneously. Under the right circumstances (when M_1 is 1 for all nodes at all points in time, meaning the matrix $diag(M_1(x)) = I_{N\times N}$), we can see that the system boils down to a set of linear equations.

$$\frac{\vec{x}(t_j+\Delta t) - \vec{x}(t_j)}{\Delta t} - M_0(\vec{x}(t_j)) = diag(M_1(\vec{x}(t_j)))AM_2(\vec{x}(t_j)) \tag{6}$$

This boils down to the linear system of equations, but in this case, the vectors are determined and the Matrix, which is the Adjacency is not determined.

$$A\vec{u}_j = \vec{v}_j \tag{7}$$

This can be recast as a Least Squares problem after the half-vectorization of the matrix and a conversion of the vector $\vec{u_j}$ to a matrix, which is expressed as:

$$(U)_{N\times\frac{N(N-1)}{2}}(\vec{A}_h)_{\frac{N(N-1)}{2}\times 1} = \vec{V}_{N\times 1} \tag{8}$$

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

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

The above leads us to the formulation of the matrix U, as follows

$$U = \left[\begin{array}{c} \mathbf{0}_{i-1\times N-i} \\ \hline u((i+1:N)) \\ \hline u_i I_{N-i} \end{array} \middle| \right]_{i=1}^{N-1} \tag{9}$$

Thus, with enough observations of $\vec{u}_j$ and $\vec{v}_j$, we can draft a least squares problem involving the above vectors and matrices. The solution to this problem would give us the half-vectorized version of the Adjacency matrix, which can be multiplied by the duplication matrix to be converted back to the Adjacency matrix. We can measure the error as the Frobenius norm between the ground truth and learned adjacency.

IV. Simulation and results

The following section deals with the implementation and realization of the problem discussed above. The implementation includes a class `Some_Matrices()` that deals with the creation/ manipulation of duplication and elimination matrices. This is also used to half-vectorize the matrix. The function `U_synth()` creates the matrix U given above.

The results show that the methodology seems to hold fine, but more number of observations are required as the number of nodes in the Graph increases.This is because the matrix U has order $N \times \frac{N(N-1)}{2}$, and the order of the half-vectorized Adjacency is $\frac{N(N-1)}{2} \times 1$. For us to invert the matrix U, we require it to be atleast of the order of $\mathcal{O}(N^2)$ (So that the psuedoinverse is as close to the real inverse as possible).

References

[1] Chittaranjan Hens et al. “Spatiotemporal signal propagation in complex networks”. In: *Nature Physics* 15.4 (Apr. 2019), pp. 403–412. ISSN: 1745-2481. DOI: 10.1038/s41567-018-0409-0. URL: https://doi.org/10.1038/s41567-018-0409-0.

[2] Weiyu Huang et al. “A Graph Signal Processing Perspective on Functional Brain Imaging”. In: *Proceedings of the IEEE* 106.5 (2018), pp. 868–885. DOI: 10.1109/JPROC.2018.2798928.

Derivative based graph learning in P-regime

```
In [48]: %pylab inline
         import matlab
         import matlab.engine
         MATLAB = matlab.engine.start_matlab()
```

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

P regime, N=50 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 [49]: N = 50
         A_gt = rand(N,N)
         for i in range(N): A_gt[i,i] = 0
         A_gt = maximum(A_gt, A_gt.T)
```

```
In [50]: A_gt, A_gt - A_gt.T
         # checking if the ground truth matrix is symmetric
```

```
Out[50]: (array([[0.        , 0.45102021, 0.76293496, ..., 0.84497008, 0.97048871,
                   0.50694468],
                  [0.45102021, 0.        , 0.71642262, ..., 0.94542764, 0.97080848,
                   0.69959137],
                  [0.76293496, 0.71642262, 0.        , ..., 0.13964869, 0.5133292 ,
                   0.77565296],
                  ...,
                  [0.84497008, 0.94542764, 0.13964869, ..., 0.        , 0.78366592,
                   0.65517674],
                  [0.97048871, 0.97080848, 0.5133292 , ..., 0.78366592, 0.        ,
                   0.81356817],
                  [0.50694468, 0.69959137, 0.77565296, ..., 0.65517674, 0.81356817,
                   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., 0., 0., ..., 0., 0., 0.],
                  [0., 0., 0., ..., 0., 0., 0.]]))
```

```
In [51]: dt = 0.01
```

```
x0 = abs(randn(N,))
x0
#  setting up initial conditions and the time resolution
```

Out[51]:
```
array([0.13009207, 1.04890679, 0.36648116, 0.50729973, 0.21106954,
       0.17655112, 0.41564172, 0.71382859, 0.78722375, 0.63517114,
       0.61632934, 0.51636903, 0.20059593, 1.56459104, 0.128074  ,
       2.02486501, 1.25592584, 0.76649194, 1.46951559, 0.08203345,
       0.54765261, 0.27007778, 0.46735601, 1.3447586 , 0.75534744,
       0.49489157, 1.50926532, 1.08535237, 0.99914587, 2.53062256,
       0.21760254, 0.56485017, 1.05158756, 1.88688989, 0.4454826 ,
       0.19378356, 0.26308208, 1.47141481, 0.98663994, 1.54722178,
       0.71603378, 2.29706543, 0.84412499, 0.05928376, 0.28058627,
       1.00939325, 1.83148165, 0.06025959, 0.25172631, 0.23314666])
```

In [52]:
```
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 [53]:
```
xp = [Traj[i][0] for i in range(1000)]
x_0 = matlab.double(xp)
MATLAB.plot(x_0)
```

Out[53]:
```
<matlab.object at 0x266fc8c9e70>
```

In [54]:
```
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 [55]:
```
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 [56]:
```
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 [57]:
```
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.

$$U = [\vec{u}_1\vec{u}_2\ldots\vec{u}_N]$$
$$V = [\vec{v}_1\vec{v}_2\ldots\vec{v}_N]$$

In [58]:
```
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:

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

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:

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

In [59]:
```
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 [60]:
```
v = V.flatten('F')
```

In [62]:
```
D = Some_Matrices.D_Matrix(N)
hA = pinv((kron(U.T,eye(N))).dot(D)).dot(v)
vec_A_gt = Some_Matrices.vec(A_gt)
fullA = D.dot(hA)
```

In [63]:
```
ff = Some_Matrices.make_mat(fullA,N)
ff
```

Out[63]:
```
array([[-9.96414968e+08,  7.22779301e+08,  3.26578115e+08, ...,
         5.41433045e+07,  3.53296896e+08, -2.38859967e+09],
       [ 7.22779301e+08,  3.26578115e+08,  8.30460757e+08, ...,
         3.53296896e+08, -2.38859967e+09,  7.22779301e+08],
       [ 3.26578115e+08,  8.30460757e+08, -3.04508433e+08, ...,
        -2.38859967e+09,  7.22779301e+08, -1.19969451e+08],
       ...,
       [ 5.41433045e+07,  3.53296896e+08, -2.38859967e+09, ...,
        -4.85300922e+07, -2.65978710e+08, -4.39085358e+08],
       [ 3.53296896e+08, -2.38859967e+09,  7.22779301e+08, ...,
        -2.65978710e+08, -4.39085358e+08,  1.58126677e+08],
       [-2.38859967e+09,  7.22779301e+08, -1.19969451e+08, ...,
        -4.39085358e+08,  1.58126677e+08,  8.94602277e+08]])
```

In [64]:
```
vec_A_gt
```

Out[64]:
```
array([0.        , 0.45102021, 0.76293496, ..., 0.65517674, 0.81356817,
       0.        ])
```

Getting the structure of U

Consider :

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

where A is of the form

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

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

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

In [65]:
```
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)
            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 [72]:
```
u = reshape(U[:,0], (N,1))
v = V[:,0]
Bigu = U_synth(N,u)
vV = v
for k in range(1,3):
    Bigu = append(Bigu, U_synth(N, reshape(U[:,k], (N,1))), axis=0)
    vV = append(vV, V[:,k], axis = 0)
```

In [73]:
```
A_vec_try = dot(pinv(Bigu), vV)
```

In [74]:
```
A_vec_try, A_gt
```

Out[74]:
```
(array([0.73447972, 0.69299203, 0.78116232, ..., 1.00545359, 0.89262261,
        0.66583805]),
 array([[0.        , 0.45102021, 0.76293496, ..., 0.84497008, 0.97048871,
         0.50694468],
        [0.45102021, 0.        , 0.71642262, ..., 0.94542764, 0.97080848,
         0.69959137],
        [0.76293496, 0.71642262, 0.        , ..., 0.13964869, 0.5133292 ,
         0.77565296],
        ...,
        [0.84497008, 0.94542764, 0.13964869, ..., 0.        , 0.78366592,
         0.65517674],
        [0.97048871, 0.97080848, 0.5133292 , ..., 0.78366592, 0.        ,
         0.81356817],
        [0.50694468, 0.69959137, 0.77565296, ..., 0.65517674, 0.81356817,
         0.        ]]))
```