GLOBAL INITIATIVE OF ACADEMIC NETWORKS

Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module C - Applications

IIT INDORE, 2016

GLOBAL INITIATIVE OF ACADEMIC NETWORKS

Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module C - Applications

https://github.com/ivanslapnicar/GIAN-Applied-NLA-Course

Cover photo: Campuses of the University of Split and IIT Indore

IIT INDORE, 2016

Contents

1 K-means Algorithm

Data clustering is one of the main mathematical applications variety of algorithms have been developed to tackle the problem. K-means is one of the basic algorithms for data clustering.

1.1 Prerequisites

The reader should be familiar with basic linear algebra.

1.2 Competences

The reader should be able to recognise applications where K-means algorithm can be efficiently used and use it.

Credits: The notebook is based on I. Mirošević, Spectral Graph Partitioning and Application to Knowledge Extraction.

1.3 Definitions

Data clustering problem is the following: partition the given set of m objects of the same type into k subsets according to some criterion. Additional request may be to find the optimal k.

K-means clustering problem is the following: partition the set $X = \{x_1, x_2, \cdots, x_m\}$, where $x_i \in \mathbb{R}^n$, into k **clusters** $\pi = \{C_1, C_2, ..., C_k\}$ such that

$$J(\pi) = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - c_i\|_2^2 \to \min$$

over all possible partitions. Here $c_i = \dfrac{1}{|C_i|} \displaystyle\sum_{x \in C_i} x$ is the mean of points in C_i and $|C_i|$ is the cardinality of C_i.

K-means clustering algorithm is the following: 1. *Initialization*: Choose initial set of k means $\{c_1, \ldots, c_k\}$ (for example, by choosing randomly k points from X). 2. *Assignment step*: Assign each point x to one nearest mean c_i. 3. *Update step*: Compute the new means. 4. *Convergence*: Repeat Steps 2 and 3 until the assignment no longer changes.

A **first variation** of a partition $\pi = \{C_1, \ldots, C_k\}$ is a partition $\pi' = \{C_1', \cdots, C_k'\}$ obtained by moving a single point x from a cluster C_i to a cluster C_j. Notice that π is a first variation of itself.

A **next partition** of the partition π is a partition $next(\pi) = \underset{\pi'}{\operatorname{argmin}}\, J(\pi')$.

First Variation clustering algorithm is the following: 1. Choose initial partition π. 2. Compute $next(\pi)$ 3. If $J(next(\pi)) < J(\pi)$, set $\pi = next(\pi)$ and go to Step 2 4. Stop.

1.4 Facts

1. The k-means clustering problem is NP-hard.
2. In the k-means algorithm, $J(\pi)$ decreases in every iteration.

3. K-means algorithm can converge to a local minimum.

4. Each iteration of the k-means algorithm requires $O(mnk)$ operations.

5. K-means algorithm is implemented in the function `kmeans()` in the package Clustering.jl.

6. $J(\pi) = \text{trace}(S_W)$, where

$$S_W = \sum_{i=1}^{k} \sum_{x \in C_i} (x - c_i)(x - c_i)^T = \sum_{i=1}^{k} \frac{1}{2|C_i|} \sum_{x \in C_i} \sum_{y \in C_i} (x - y)(x - y)^T.$$

   Let c denote the mean of X. Then $S_W = S_T - S_B$, where

$$S_T = \sum_{x \in X} (x - c)(x - c)^T = \frac{1}{2m} \sum_{i=1}^{m} \sum_{j=1}^{m} (x_i - x_j)(x_i - x_j)^T,$$
$$S_B = \sum_{i=1}^{k} |C_i|(c_i - c)(c_i - c)^T = \frac{1}{2m} \sum_{i=1}^{k} \sum_{j=1}^{k} |C_i||C_j|(c_i - c_j)(c_i - c_j)^T.$$

7. In order to try to avoid convergence to local minima, the k-means algorithm can be enhanced with first variation by adding the following steps:

   1. Compute $next(\pi)$.
   2. If $J(next(\pi)) < J(\pi)$, set $\pi = next(\pi)$ and go to Step 2.

1.4.1 Example - Random clusters

We generate k random clusters around points with integer coordinates.

```
In [1]: function myKmeans{T}(X::Array{T}, k::Int64)
            # X is Array of Arrays
            m,n=length(X),length(X[1])
            C=Array(Int64,m)
            # Choose random k means among X
            c=X[randperm(m)[1:k]]
            # This is just to start the while loop
            cnew=copy(c)
            cnew[1]=cnew[1]+[1.0;1.0]
            # Loop
            iterations=0
            while cnew!=c
                iterations+=1
                cnew=copy(c)
                # Assignment
                for i=1:m
                    C[i]=findmin([norm(X[i]-c[j]) for j=1:k])[2]
                end
                # Update
                for j=1:k
                    c[j]=mean(X[C.==j])
                end
```

```
            end
            C,c,iterations
        end
```

```
Out[1]: myKmeans (generic function with 1 method)
```

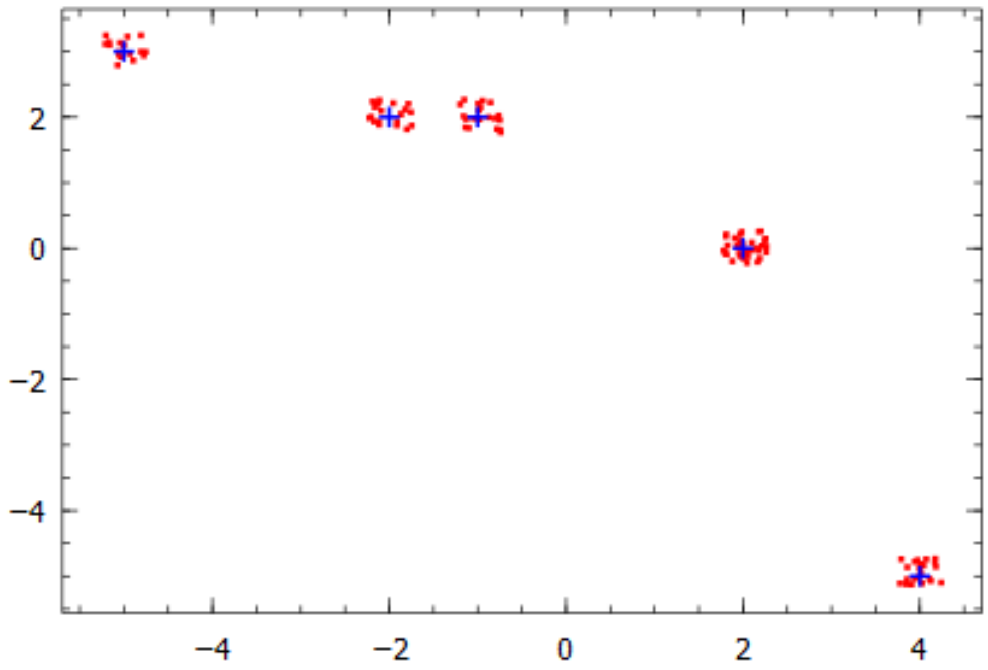
```
In [3]: # Pkg.checkout("Winston")
        using Winston
        using Colors
```

```
In [4]: # Generate points
        k=5
        centers=rand(-5:5,k,2)
        # Number of points in cluster
        sizes=rand(10:50,k)
        # X is array of arrays
        X=Array(Array{Float64,1},sum(sizes))
        first=0
        last=0
        for j=1:k
            first=last+1
            last=last+sizes[j]
            for i=first:last
                X[i]=map(Float64,vec(centers[j,:])+(rand(2)-0.5)/2)
            end
        end
        centers, sizes
```

```
Out[4]: (
        5x2 Array{Int64,2}:
          4  -5
         -1   2
          2   0
         -2   2
         -5   3,

        [22,21,35,22,18])
```

```
In [5]: # Prepare for plot
        function extractxy(X::Array)
            x=map(Float64,[X[i][1] for i=1:length(X)])
            y=map(Float64,[X[i][2] for i=1:length(X)])
            x,y
        end
        x,y=extractxy(X)
        plot(x,y,"r.",centers[:,1],centers[:,2],"b+")
```

```
Out[5]:
```

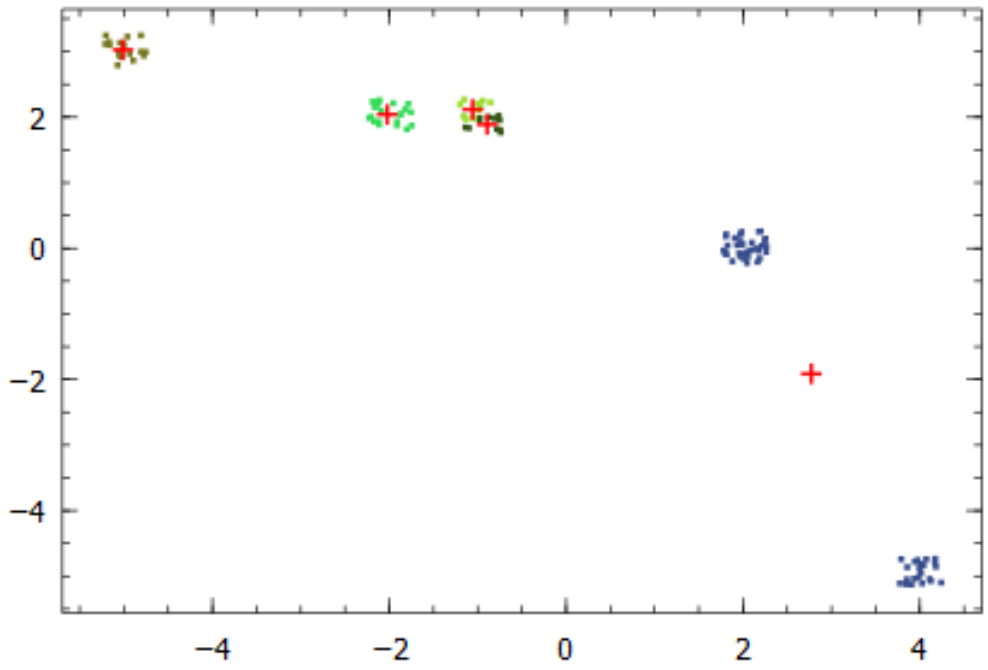



```
In [6]: # Plot the solution
        function plotKmeansresult(C::Array,c::Array,X::Array)
            p=FramedPlot()
            x,y=extractxy(X)
            # Custers
            for j=1:k
                # Random color
                col=RGB(rand(),rand(),rand())
                p1=Points(x[find(C.==j)],y[find(C.==j)],
                "color",col,symbolkind="dot")
                add(p,p1)
            end
            # Means
            cx,cy=extractxy(c)
            p2=Points(cx,cy,color="red",symbolkind="plus")
            add(p,p2)
        end
```

```
Out[6]: plotKmeansresult (generic function with 1 method)
```

```
In [7]: # Cluster the data, repeat several times
        C,c,iterations=myKmeans(X,k)
        plotKmeansresult(C,c,X)
```

```
Out[7]:
```



What happens?

We see that the algorithm, although simple, for this example converges to a local minimum.

Let us try the function `kmeans()` from the package `Clustering.jl`. In this function, the inpout is a matrix where columns are points, number of cluster we are looking for, and, optionally, the method to compute initial means.

If we choose `init=:rand`, the results are similar. If we choose `init=:kmpp`, wjich is the default, the results are better, but convergence to a local minimum is still possible.

Run the clustering several times!

```
seeding_algorithm(s::Symbol) =
    s == :rand ? RandSeedAlg() :
    s == :kmpp ? KmppAlg() :
    s == :kmcen ? KmCentralityAlg() :
    error("Unknown seeding algorithm $s")
```

```
In [8]: # Pkg.add("Clustering")
        using Clustering
```

```
In [9]: methods(kmeans)
```

```
Out[9]: # 1 method for generic function "kmeans":
        kmeans(X::Array{T,2}, k::Int64) at C:\Users\Ivan\.julia\v0.4\Clustering\src\kmeans.j
```

```
In [10]: X1=[x y]'
         out=kmeans(X1,k,init=:rand)
```

```
Out[10]: Clustering.KmeansResult{Float64}(2x5 Array{Float64,2}:
          -1.50878   2.00837     -4.84266  -5.09462   3.98735
           2.01996  -0.00547131   2.98311   3.049    -4.95126,[5,5,5,5,5,5,5,5,5,5  ...  4,3
```

```
In [11]: fieldnames(KmeansResult)
```

```
Out[11]: 8-element Array{Symbol,1}:
          :centers
          :assignments
          :costs
          :counts
          :cweights
          :totalcost
          :iterations
          :converged
```

```
In [12]: out.centers
```

```
Out[12]: 2x5 Array{Float64,2}:
          -1.50878   2.00837     -4.84266  -5.09462   3.98735
           2.01996  -0.00547131   2.98311   3.049    -4.95126
```

```
In [13]: println(out.assignments)
```

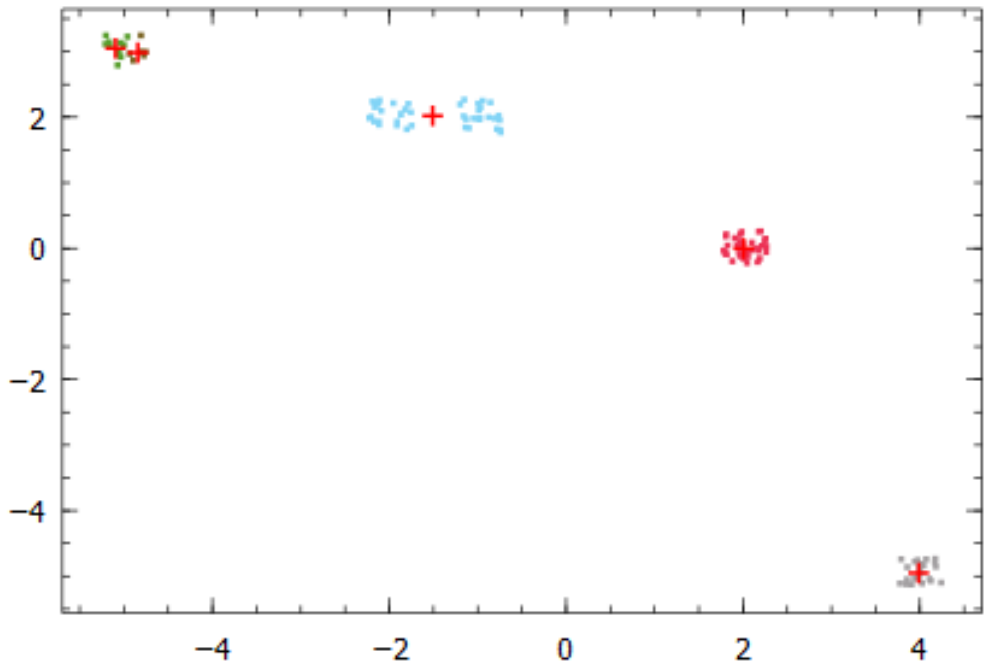
```
[5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2
```

```
In [14]: # We need to modify the plotting function
         function plotKmeansresult(out::KmeansResult,X::Array)
             p=FramedPlot()
             # Custers
             k=size(out.centers,2)
             for j=1:k
                 # Random color
                 col=RGB(rand(),rand(),rand())
                 p1=Points(X[1,find(out.assignments.==j)],
                 X[2,find(out.assignments.==j)],"color",col,symbolkind="dot")
                 add(p,p1)
             end
             # Means
             p2=Points(out.centers[1,:],out.centers[2,:],
             color="red",symbolkind="plus")
             add(p,p2)
         end
```

```
Out[14]: plotKmeansresult (generic function with 2 methods)
```

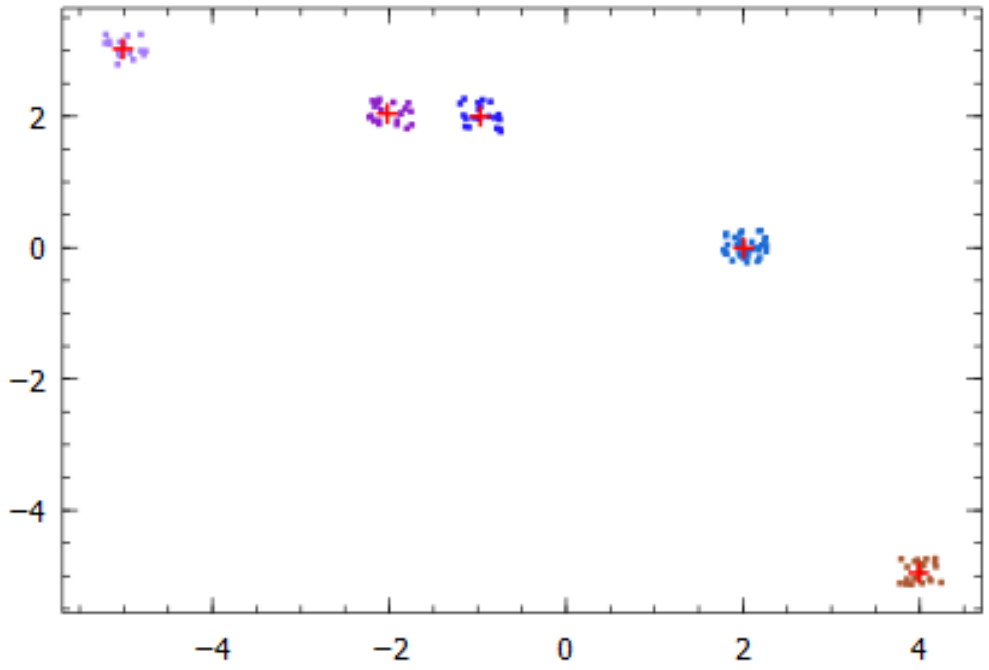
```
In [15]: plotKmeansresult(out,X1)
```

```
Out[15]:
```



In [16]:
```
out=kmeans(X1,k,init=:kmpp)
plotKmeansresult(out,X1)
```

Out[16]:



1.4.2 Example - Concentric rings

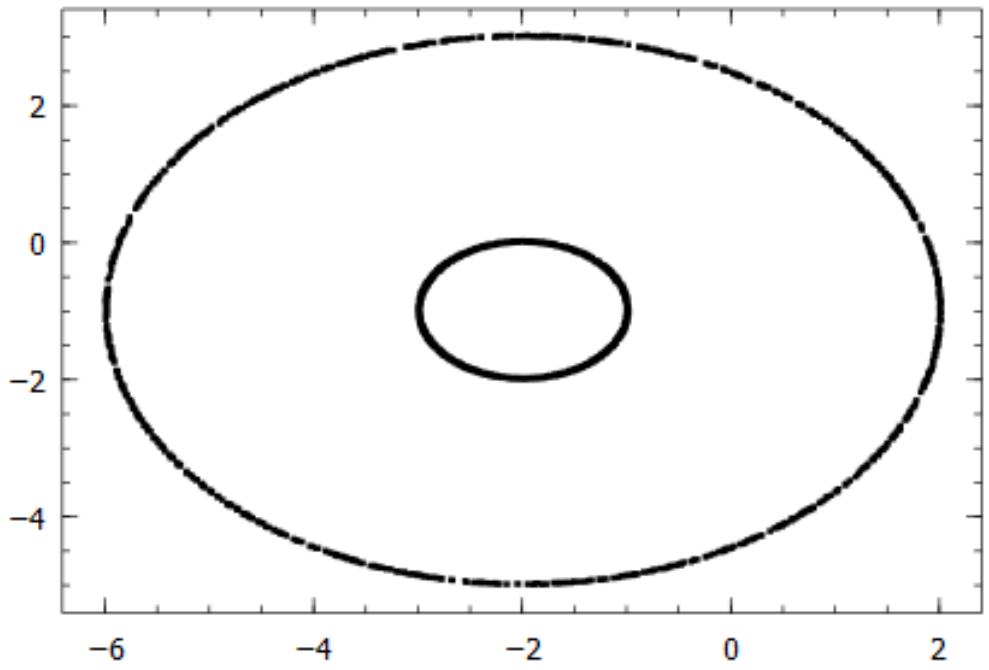
The k-means algorithm works well if clusters can be separated by hyperplanes. In this example it is not the case.

In [17]:
```
# Number of rings, try also k=3
k=2
# Center
center=[rand(-5:5);rand(-5:5)]
# Radii
radii=randperm(10)[1:k]
# Number of points in circles
sizes=rand(1000:2000,k)
center,radii,sizes
```

Out[17]:
```
([-2,-1],[4,1],[1100,1394])
```

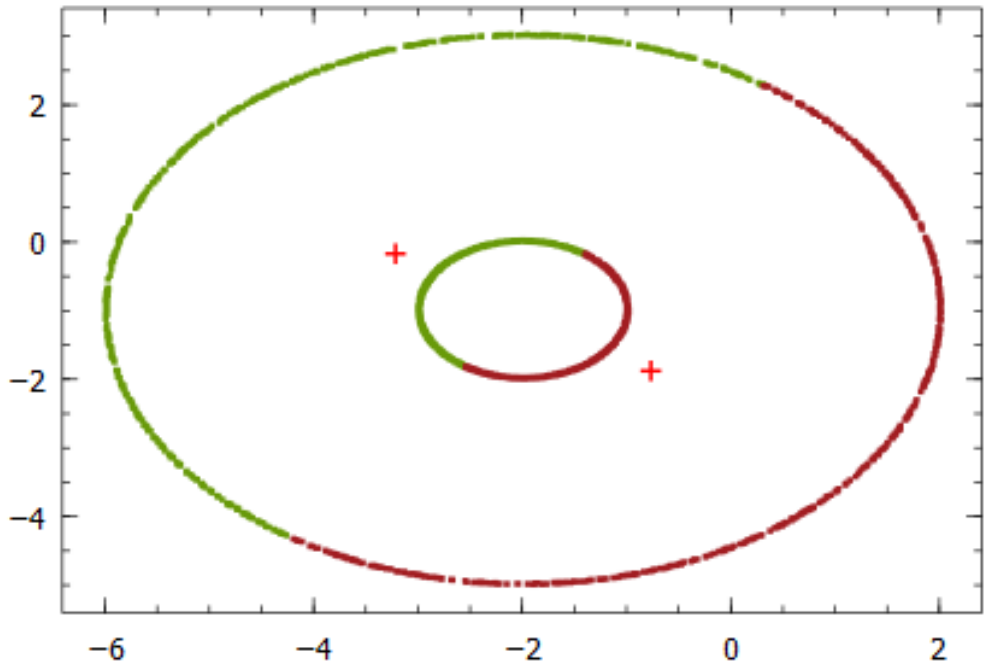
In [18]:
```
# Points
X=Array(Float64,2,sum(sizes))
first=0
last=0
for j=1:k
    first=last+1
    last=last+sizes[j]
    # Random angles
    ϕ=2*π*rand(sizes[j])
    for i=first:last
        l=i-first+1
        X[:,i]=center+radii[j]*[cos(ϕ[l]);sin(ϕ[l])]+(rand(2)-0.5)/50
    end
end
plot(X[1,:],X[2,:],".")
```

Out[18]:



In [19]:
```
out=kmeans(X,k)
plotKmeansresult(out,X)
```

Out[19]:



In [ ]:

2 Spectral Graph Bipartitioning

Many data clustering problems can be interpreted as clustering of vertices of graphs. Graph bipartitioning problem is to partition vertices into subsets such that the connections within subsets are stronger than the connections between different subsets.

Partition of the vertices into two subsetts is done according to signs of the eigenvectors of the second smallest eigenvalue of the Laplacian matrix.

2.1 Prerequisites

The reader should be familiar with basic graph theory, linear algebra, and eigenvalues and eigenvectors.

2.2 Competences

The reader should be able to apply graph spectral bipartitioning and recursive bipartitioning to data clustering problems.

Credits: The notebook is based on I. Mirošević, Spectral Graph Partitioning and Application to Knowledge Extraction.

2.3 Graphs

For more details, see W. H. Haemers, Matrices and Graphs and S. Butler and F. Chung, Spectral Graph Theory and the references therein.

2.3.1 Definitions

A **weighted graph** is an ordered triplet $G = (V, E, \omega)$, where $V = \{1, 2, 3, ..., n\}$ is the set of **vertices** , $E = \{(i, j)\}$ is a set of **edges** connecting vertices, and ω is a set of **weights** of edges. We assume G is undirected.

An **adjacency matrix** of graph G is the matrix A defined as $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise} \end{cases}$.

A **weight matrix** of graph G is the matrix W defined as $W_{ij} = \begin{cases} \omega(e) & \text{if } e = (i,j) \in E, \\ 0 & \text{otherwise} \end{cases}$.

A **Laplacian matrix** of graph G is the matrix $L = D - W$, where $D = \text{diag}(d_1, d_2, \ldots, d_n)$ with $d_i = \sum_{k=1}^{n} W_{ik}$ for $i = 1, \ldots, n$.

A **normalized Laplacian matrix** is the matrix $L_n = D^{-1/2} L D^{-1/2} \equiv D^{-1/2}(D - W)D^{-1/2}$ (*the scaled matrix of* L).

An **incidence matrix** of graph G is the $|V| \times |E|$ matrix I_G. Each row of $I : G$ corresponds to a vertex of G and each column corresponds to an edge of G. In the column corresponding to en edge $e = (i, j)$, all elements are zero except the ones in the i-th and j-th row, which are equal to $\sqrt{\omega(e)}$ and $-\sqrt{\omega(e)}$, respectively.

2.3.2 Examples

Graph types and algorithms are implemented in the package Graphs.jl. See also Graphs.jl Documentation.

```
In [1]: using Graphs
        using IJuliaPortrayals
```

There are two functions to generate graph, `simple_graph()` and `inclist()`. The first is simpler and sufficient for computation. The second is more complex and allows plotting of weights. We shall illustrate both.

```
In [2]: G=simple_graph(7,is_directed=false)
```

```
Out[2]: Undirected Graph (7 vertices, 0 edges)
```

```
In [3]: # Sources, targets, and weight
        sn=[1,1,1,2,2,3,3,3,5,5,6]
        tn=[2,3,4,4,5,4,6,7,6,7,7]
        wn=[2,3,4,7,1,3,2,1,7,3,5]
        [sn tn wn]
```

```
Out[3]: 11x3 Array{Int64,2}:
         1  2  2
         1  3  3
         1  4  4
         2  4  7
         2  5  1
         3  4  3
         3  6  2
         3  7  1
         5  6  7
         5  7  3
         6  7  5
```

```
In [4]: for i=1:length(sn)
            add_edge!(G,sn[i],tn[i])
        end
```

```
In [5]: # Repeat on error!
        GraphViz(to_dot(G),"fdp","svg")
```

```
Out[5]: IJuliaPortrayals.GraphViz("graph graphname {\n1\n2\n3\n4\n5\n6\n7\n1 -- 2\n1 -- 3\n1
```

```
In [6]: edges(G)
```

```
Out[6]: 11-element Array{Graphs.Edge{Int64},1}:
         edge [1]: 1 -- 2
         edge [2]: 1 -- 3
         edge [3]: 1 -- 4
         edge [4]: 2 -- 4
         edge [5]: 2 -- 5
         edge [6]: 3 -- 4
         edge [7]: 3 -- 6
         edge [8]: 3 -- 7
         edge [9]: 5 -- 6
         edge [10]: 5 -- 7
         edge [11]: 6 -- 7
```

```
In [7]: W=weight_matrix(G,wn)
```

```
Out[7]: 7x7 Array{Int64,2}:
 0  2  3  4  0  0  0
 2  0  0  7  1  0  0
 3  0  0  3  0  2  1
 4  7  3  0  0  0  0
 0  1  0  0  0  7  3
 0  0  2  0  7  0  5
 0  0  1  0  3  5  0
```

```
In [8]: issym(W)
```

```
Out[8]: true
```

```
In [9]: L=laplacian_matrix(G,wn)
```

```
Out[9]: 7x7 Array{Int64,2}:
  9  -2  -3  -4   0   0   0
 -2  10   0  -7  -1   0   0
 -3   0   9  -3   0  -2  -1
 -4  -7  -3  14   0   0   0
  0  -1   0   0  11  -7  -3
  0   0  -2   0  -7  14  -5
  0   0  -1   0  -3  -5   9
```

```
In [10]: # Normalized Laplacian matrix, we need Symmetric()
         function normalized(L::Matrix)
             D=1.0./sqrt(diag(L))
             map(Float64,Symmetric([L[i,j]*D[i]*D[j] for i=1:length(D),
                 j=1:length(D)]))
         end
         Ln=normalized(L)
```

```
Out[10]: 7x7 Array{Float64,2}:
  1.0       -0.210819   -0.333333  ...   0.0         0.0        0.0
 -0.210819   1.0         0.0          -0.0953463   0.0        0.0
 -0.333333   0.0         1.0           0.0        -0.178174  -0.111111
 -0.356348  -0.591608   -0.267261      0.0         0.0        0.0
  0.0       -0.0953463   0.0           1.0        -0.564076  -0.301511
  0.0        0.0        -0.178174  ...  -0.564076    1.0       -0.445435
  0.0        0.0        -0.111111     -0.301511   -0.445435   1.0
```

```
In [11]: # The second approach
         g = inclist(ExVertex, ExEdge{ExVertex}; is_directed=false)
         for i=1:7
             add_vertex!(g, ExVertex(i,"$i"))
         end
         for i=1:11
             # add_edge!(g,ExEdge(i,vertices(g)[sn[i]], vertices(g)[en[i]]))
             add_edge!(g, vertices(g)[sn[i]], vertices(g)[tn[i]])
         end
         weight_matrix(g,wn)
```

```
Out[11]: 7x7 Array{Int64,2}:
          0  2  3  4  0  0  0
          2  0  0  7  1  0  0
          3  0  0  3  0  2  1
          4  7  3  0  0  0  0
          0  1  0  0  0  7  3
          0  0  2  0  7  0  5
          0  0  1  0  3  5  0
```

```
In [12]: # Now we add labels in order to print weights
         for i=1:length(g.inclist)
             for j=1:length(g.inclist[i])
                 ei=g.inclist[i][j]
                 attrs = attributes(ei, g)
                 attrs["label"] = wn[edge_index(ei,g)]
             end
         end
```

```
In [15]: # What is the optimal bipartition?
         GraphViz(to_dot(g),"fdp","svg")
```

```
Out[15]: IJuliaPortrayals.GraphViz("graph graphname {\n1 -- 2 [\"label\"=\"2\"]\n1 -- 3 [\"l
```

```
In [16]: g.inclist
```

```
Out[16]: 7-element Array{Array{Graphs.ExEdge{Graphs.ExVertex},1},1}:
          [edge [1]: vertex [1] "1" -- vertex [2] "2",edge [2]: vertex [1] "1" -- vertex [3]
          [edge [1]: vertex [2] "2" -- vertex [1] "1",edge [4]: vertex [2] "2" -- vertex [4]
          [edge [2]: vertex [3] "3" -- vertex [1] "1",edge [6]: vertex [3] "3" -- vertex [4]
          [edge [3]: vertex [4] "4" -- vertex [1] "1",edge [4]: vertex [4] "4" -- vertex [2]
          [edge [5]: vertex [5] "5" -- vertex [2] "2",edge [9]: vertex [5] "5" -- vertex [6]
          [edge [7]: vertex [6] "6" -- vertex [3] "3",edge [9]: vertex [6] "6" -- vertex [5]
          [edge [8]: vertex [7] "7" -- vertex [3] "3",edge [10]: vertex [7] "7" -- vertex [5
```

```
In [17]: # The above incidence list is closely related to the incidence matrix
         IG=zeros(length(g.vertices),g.nedges)
         for i=1:g.nedges
             IG[tn[i],i]=sqrt(wn[i])
             IG[sn[i],i]=-sqrt(wn[i])
         end
         IG
```

```
Out[17]: 7x11 Array{Float64,2}:
          -1.41421  -1.73205  -2.0    0.0       ...   0.0    0.0        0.0        0.0
           1.41421   0.0       0.0   -2.64575       0.0    0.0        0.0        0.0
           0.0       1.73205   0.0    0.0          -1.0    0.0        0.0        0.0
           0.0       0.0       2.0    2.64575       0.0    0.0        0.0        0.0
           0.0       0.0       0.0    0.0           0.0   -2.64575   -1.73205    0.0
           0.0       0.0       0.0    0.0       ...   0.0    2.64575    0.0       -2.23607
           0.0       0.0       0.0    0.0           1.0    0.0        1.73205    2.23607
```

2.3.3 Facts

1. $L = I_G I_G^T$.
2. L is symmetric PSD matrix.
3. $L\mathbf{1} = 0$ for $\mathbf{1} = [1, ..., 1]^T$, thus 0 is an eigenvalue of L and $\mathbf{1}$ is the corresponding eigenvector.
4. If G has c connected components, then L has c eigenvalues equal to 0.
5. For every $x \in \mathbb{R}^n$, it holds $x^T L x = \sum_{i<j} W_{ij}(x_i - x_j)^2$.
6. For every $x \in \mathbb{R}^n$ and $\alpha, \beta \in \mathbb{R}$, it holds $(\alpha x + \beta\mathbf{1})^T L(\alpha x + \beta\mathbf{1}) = \alpha^2 x^T L x$.
7. Assume that the eigenvalues of L are increasingly ordered. Then,
$$0 = \lambda_1(L) \leq \lambda_2(L) \leq \cdots \leq \lambda_n(L) \leq 2 \max_{i=1,\cdots,n} d_i.$$
8. $\sigma(L_n) \subseteq [0, 2]$.

2.3.4 Examples

```
In [18]: # Fact 1
         n=size(L,1)
         norm(L-IG*IG')

Out[18]: 4.041433959105367e-15

In [19]: # Facts 2 and 7
         issym(L), eigvals(L), 2*maximum(diag(L))

Out[19]: (true,[-4.218847493575595e-15,1.9497005798217664,8.500141091860392,12.488755596470

In [20]: # Fact 3
         L*ones(n)

Out[20]: 7-element Array{Float64,1}:
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0
          0.0

In [21]: # Fact 5, explain the difference in formulas
         x=rand(n)
         x'*L*x, sum([W[i,j]*(x[i]-x[j])^2 for i=1:n, j=1:n])/2

Out[21]: ([9.271813337823035],9.271813337823035)
```

```
In [22]: # Fact 6
         α,β=rand(),rand()
         (α*x+β*ones(n))'*L*(α*x+β*ones(n)), α^2*x'*L*x

Out[22]: ([0.9838573972429927],[0.9838573972429961])

In [23]: # Fact 8
         eigvals(Ln)

Out[23]: 7-element Array{Float64,1}:
          3.64292e-17
          0.17559
          0.887695
          1.29
          1.3692
          1.62123
          1.65628
```

2.4 Bipartitioning

2.4.1 Definitions

Let $\pi = \{V_1, V_2\}$ be a partition of V with $V_1, V_2 \neq \emptyset$.
A **cut** of a partition π is the sum of weights of all edges between V_1 and V_2, $cut(\pi) \equiv cut(V_1, V_2) = \sum_{i \in V_1, j \in V_2} W_{ij}$.
A **weight** of a vertex $i \in V$ is the sum of the weights of all egdges emanating from i, $\omega(i) = \sum_{j=1}^{n} W_{ij}$.
A **weight** of a subset $\bar{V} \subset V$ is the sum of the weights of all vertices in $\bar{V}$, $\omega(\bar{V}) = \sum_{i \in \bar{V}} \omega(i)$.
A **proportional cut** of a partition π is $pcut(\pi) = \frac{cut(\pi)}{|V_1|} + \frac{cut(\pi)}{|V_2|}$.
A **normalized cut** of a partition π is $ncut(\pi) = \frac{cut(\pi)}{\omega(V_1)} + \frac{cut(\pi)}{\omega(V_2)}$.

2.4.2 Example

Consider the following partitions (all edges have unit weights):
The left partition π v.s. the right partition π':
$cut(\pi) = 2$ v.s. $cut(\pi') = 3$
$pcut(\pi) = \frac{2}{1} + \frac{2}{11} = 2.18$ v.s. $pcut(\pi') = \frac{3}{6} + \frac{3}{6} = 1$
$ncut(\pi) = \frac{2}{2} + \frac{2}{50} = 1.04$ v.s. $ncut(\pi') = \frac{3}{27} + \frac{3}{25} = 0.23$

2.4.3 Facts

1. The informal description of the bipartitioning problem can be formulated as two problems,

$$\arg\min_{\pi} pcut(\pi) \quad \text{or} \quad \arg\min_{\pi} ncut(\pi).$$

   The first problem favors partitions into subsets with similar numbers of vertices, while the second problem favors partitions into subsets with similar weights.

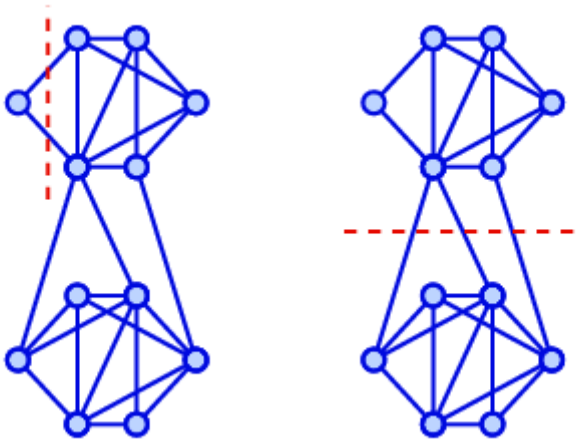


Figure 1: Two partitions

2. Both problems are NP-hard.

3. Approximate solutions can be computed by suitable relaxations in $O(n^2)$ operations.

4. The partition π is defined by the vector y such that

$$y_i = \begin{cases} \frac{1}{2} & \text{for } i \in V_1 \\ -\frac{1}{2} & \text{for } i \in V_2 \end{cases}$$

The proportional cut problem can be formulated as the **discrete** proportional cut problem

$$\begin{aligned} \min_{y_i \in \{-1/2, 1/2\}} & \frac{1}{2} \sum_{i,j} (y_i - y_j)^2 W_{ij}. \\ |\mathbf{y}^T \mathbf{1}| \leq \beta & \end{aligned}$$

Parameter β controls the number of vertices in each subset.

5. The normalized cut problem can be formulated as the **discrete** normalized cut problem

$$\begin{aligned} \min_{y_i \in \{-1/2, 1/2\}} & \frac{1}{2} \sum_{i,j} (y_i - y_j)^2 W_{ij}. \\ |y^T D \mathbf{1}| \leq \beta & \end{aligned}$$

Parameter β controls the weights of each subset.

6. Using the Fact 5 above, the discrete proportional cut problem can be formulated as the **relaxed** proportional cut problem

$$\begin{aligned} \min_{y \in \mathbb{R}^n} & \quad y^T L y. \\ |y^T \mathbf{1}| \leq 2\beta/\sqrt{n} & \\ y^T y = 1 & \end{aligned}$$

Similarly, the discrete normalized cut problem can be formulated as the **relaxed** normalized cut problem

$$\begin{aligned} \min_{y \in \mathbb{R}^n} & \quad y^T L_n y. \\ |y^T D \mathbf{1}| \leq \beta/\sqrt{\theta n} & \\ y^T D y = 1 & \end{aligned}$$

7. Solutions of the relaxed problems follow from the following theorem. Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix with eigenvalues $\lambda_1 < \lambda_2 < \lambda_3 \leq \cdots \leq \lambda_n$ and let $v^{[1]}, v^{[2]}, \ldots, v^{[n]}$ be the corresponding eigenvectors. For the fixed $0 \leq \alpha < 1$, the solution of the problem

$$\begin{aligned} &\min_{y \in \mathbb{R}^n} \quad y^T A y \\ &\left| y^T v^{[1]} \right| \leq \alpha \\ &y^T y = 1 \end{aligned}$$

is $y = \pm\alpha v^{[1]} \pm \sqrt{1-\alpha^2} v^{[2]}$.

8. For $0 \leq \beta < n/2$, the solution of the relaxed proportional cut problem is

$$y = \pm\frac{2\beta}{n\sqrt{n}}\mathbf{1} \pm \sqrt{1 - 4\frac{\beta^2}{n^2}} v^{[2]},$$

where $v^{[2]}$ is an eigenvector corresponding to $\lambda_2(L)$. $v^{[2]}$ the **Fiedler vector**. Since the first summand carries no information, V is partitioned according to the signs of the components of $v^{[2]}$:

$$V_1 = \{i : v_i^{[2]} < 0\}, \quad V_2 = \{i : v_i^{[2]} \geq 0\}.$$

Notice that the value of β is irrelevant for the solution.

9. For $0 \leq \beta < \sqrt{\theta n} \left\| D^{\frac{1}{2}}\mathbf{1} \right\|_2$, the solution of the relaxed proportional cut problem is

$$y = \pm\frac{\beta}{\sqrt{\theta n}\left\| D^{\frac{1}{2}}\mathbf{1} \right\|_2^2}\mathbf{1} \pm \sqrt{1 - \frac{\beta^2}{\theta n \left\| D^{\frac{1}{2}}\mathbf{1} \right\|_2^2}} D^{-\frac{1}{2}} v^{[2]},$$

where $v^{[2]}$ is an eigenvector corresponding to $\lambda_2(L_n)$. V is partitioned according to the signs of the components of $v^{[2]}$, as above.

10. Neither of the relaxed algorithms is guaranteed to solve exactly the true (proportional / normalized) cut problem. However, the computed solutions are in the right direction. Whether to use proportional or normalized cut formulation, depends upon the specific problem.

```
In [24]: # Voila!
         λ,v=eigs(L,nev=2,which=:SM, v0=ones(n))
```

```
Out[24]: ([2.5376526277146444e-16,1.9497005798217648],
         7x2 Array{Float64,2}:
          -0.377964  -0.383259
          -0.377964  -0.373084
          -0.377964  -0.145189
          -0.377964  -0.380089
          -0.377964   0.423708
          -0.377964   0.408506
          -0.377964   0.449408,

         2,1,7,[0.0,0.0,0.0,0.0,0.0,0.0,0.0])
```

```
In [25]: λ,v=eigs(Ln,nev=2,which=:SM, v0=ones(n))
```

```
Out[25]: ([5.2589511692770613e-17,0.17610318098079664],
         7x2 Array{Float64,2}:
          -0.344124   0.332619
          -0.362738   0.362348
          -0.344124   0.124387
          -0.429198   0.461229
          -0.380443  -0.413347
          -0.429198  -0.453818
          -0.344124  -0.391225,

         2,1,7,[0.0,0.0,0.0,0.0,0.0,0.0,0.0])
```

2.4.4 Example - Concentric circles

A **complete graph** has edges connecting each pair of vertices.

To a set of points $X = \{x_1, x_2, \cdots, x_m\}$, where $x_i \in \mathbb{R}^n$, we assign a weighted complete graph $G = (V, E)$ with m vertices, where the vertex $j \in V$ corresponds to the point $x_j \in X$.

The main idea is to assign weight of an edge $e = (i, j)$ which reflects the distance between x_i and x_j, something like $\omega(e) = \dfrac{1}{\text{dist}(x_i, x_j)}$.

However, this has to be implemented with care. For example, using simple Euclidean distance yield the same results as the function `kmeans()`. In this example we use Gaussian kernel, that is

$$\omega(e) = e^{-\|x_i - x_j\|_2^2/\sigma^2},$$

where the choice of σ is based on experience.

The computation of various distances is implemented in the package Distances.jl.

N.B. A complete graph is generated by the function `simple_complete_graph(m, is_directed=false)`, but we will construct the Laplace matrix directly.

```
In [27]: using Winston
         using Colors
         using Distances
```

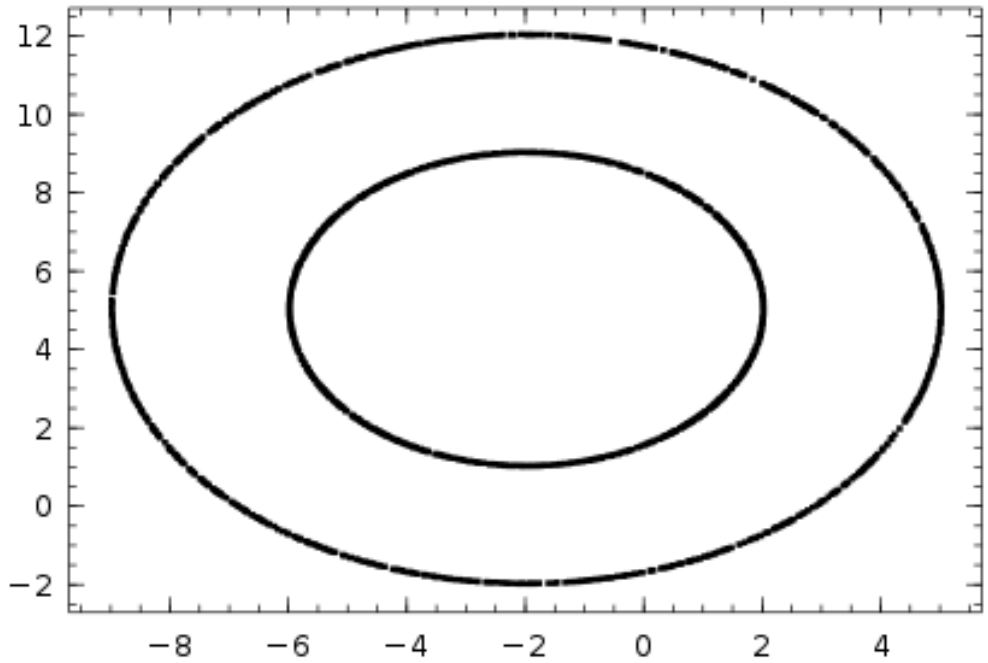
```
In [28]: # Generate two concentric rings
         k=2
         # Center
         center=[rand(-5:5);rand(-5:5)]
         # Radii
         radii=randperm(10)[1:k]
         # Number of points in circles
         sizes=rand(1000:2000,k)
         center,radii,sizes
```

```
Out[28]: ([-2,5],[4,7],[1356,1281])
```

```
In [29]: # Points
         m=sum(sizes)
         X=Array(Float64,2,m)
         first=0
         last=0
         for j=1:k
```

```
        first=last+1
        last=last+sizes[j]
        # Random angles
        ϕ=2*π*rand(sizes[j])
        for i=first:last
            l=i-first+1
            X[:,i]=center+radii[j]*[cos(ϕ[l]);sin(ϕ[l])]+(rand(2)-0.5)/50
        end
    end
    Winston.plot(X[1,:],X[2,:],".")
```

Out[29]:



In [30]:
```
# Weight matrix
W=1./pairwise(SqEuclidean(),X)
```

Out[30]:
```
2637x2637 Array{Float64,2}:
 Inf             0.457395      0.0158114   ...    0.00997808    0.0125469
   0.457395    Inf             0.0157373         0.0121591     0.0168873
   0.0158114     0.0157373   Inf                 0.025944      0.0164951
   0.0353747     0.0249019     0.0340432         0.00893709    0.00833383
   0.024064      0.0334354     0.0350781         0.0771572     0.111741
   0.0167554     0.0156965     0.609438    ...    0.0180191     0.0127445
   0.0230769     0.0314475     0.0375676         0.0845751     0.109066
   0.0156824     0.0163756     0.961413          0.0371432     0.0214349
   0.0332082     0.0533777     0.0251584         0.0423283     0.0838646
   0.0166133     0.0156581     0.748535          0.0186008     0.0130282
   0.168689      1.02875       0.0163382   ...    0.0145565     0.0218437
```

```
 0.0359794     0.0251762     0.0333527        0.00888272    0.00831058
 0.0161225     0.0156238     3.06465          0.021815      0.0145774
 ⋮                                        ⋱     ⋮
 0.0726458     0.0393593     0.00901434  …     0.00540019    0.00626134
 0.0346555     0.0213307     0.0108177        0.00509938    0.00534972
 0.0866982     0.0474401     0.00872584       0.00556881    0.00664293
 0.0165206     0.0246242     0.0124928        0.0367608     0.169859
 0.00886962    0.0100979     0.0417369        0.18474       0.0381231
 0.0171308     0.0126329     0.0164296   …     0.00553263    0.00513346
 0.0358082     0.0218727     0.0106917        0.0051005     0.00537597
 0.00915303    0.00841185    0.0672449        0.0105206     0.00742372
 0.00827937    0.00862625    0.091837         0.0320152     0.015198
 0.0367991     0.067971      0.00910421       0.0119542     0.0218416
 0.00997808    0.0121591     0.025944    …    Inf             0.120333
 0.0125469     0.0168873     0.0164951        0.120333      Inf
```

In [31]:
```
# Laplacian matrix
for i=1:m
    W[i,i]=0
end
L=diagm(vec(sum(W,2)))-W
norm(L*ones(m))
```

Out[31]:
```
7.743959811168703e-8
```

In [32]:
```
# Notice \lambda_1
λ,v=eigs(L,nev=2,which=:SM, v0=ones(m))
```

Out[32]:
```
([1.2214027539589119e-11,40.78722055807862],
2637x2 Array{Float64,2}:
 0.0194735  -0.0159353
 0.0194735  -0.0109768
 0.0194735   0.0135261
 0.0194735  -0.0126673
 0.0194735   0.0149154
 0.0194735   0.00879501
 0.0194735   0.0154361
 0.0194735   0.016183
 0.0194735   0.00961755
 0.0194735   0.00924953
 0.0194735  -0.00735026
 0.0194735  -0.0128161
 0.0194735   0.0113803
 ⋮
 0.0194735  -0.0329933
 0.0194735  -0.0327261
 0.0194735  -0.032315
 0.0194735   0.0180042
 0.0194735   0.0341964
 0.0194735  -0.0226161
 0.0194735  -0.0328224
```

```
 0.0194735   0.0100325
 0.0194735   0.0290272
 0.0194735  -0.00564802
 0.0194735   0.0329114
 0.0194735   0.0259708 ,

2,2,37,[-1.58065e-5,-1.1247e-6,3.55505e-5,-1.83049e-5,-1.44014e-5,2.40401e-5,1.037
```

In [33]:
```
# Define clusters
C=ones(Int64,m)
C[find(v[:,2].>0)]=2
```

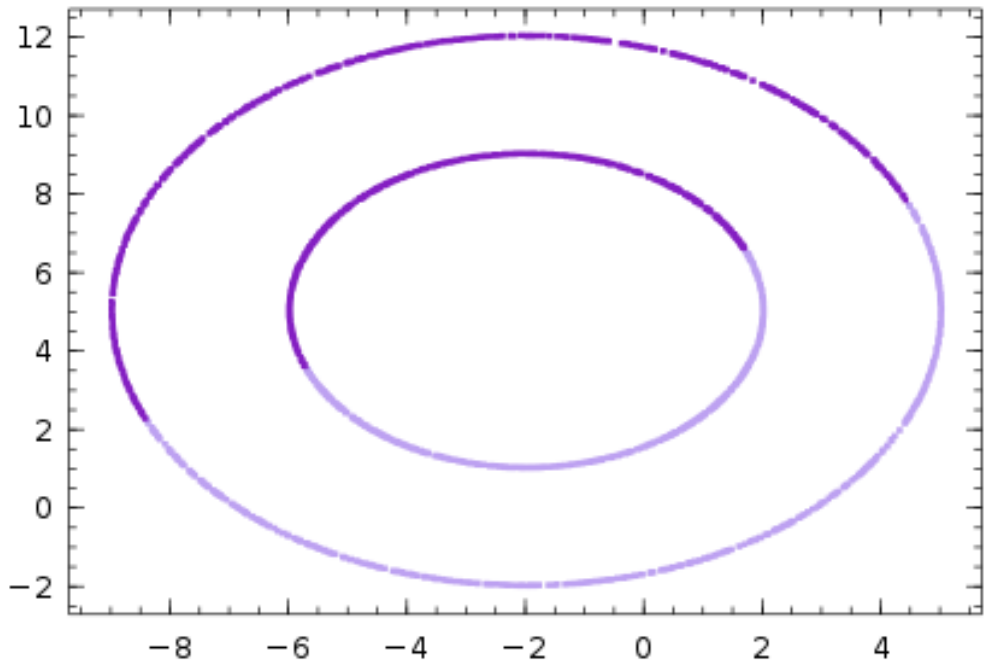
Out[33]:
```
2
```

In [34]:
```
# Yet another plotting function
function plotKpartresult(C::Vector,X::Array)
    p=FramedPlot()
    k=maximum(C)
    for j=1:k
        # Random color
        col=RGB(rand(),rand(),rand())
        p1=Points(X[1,find(C.==j)],
        X[2,find(C.==j)],"color",col,symbolkind="dot")
        add(p,p1)
    end
    p
end
```

Out[34]:
```
plotKpartresult (generic function with 1 method)
```

In [35]:
```
plotKpartresult(C,X)
```

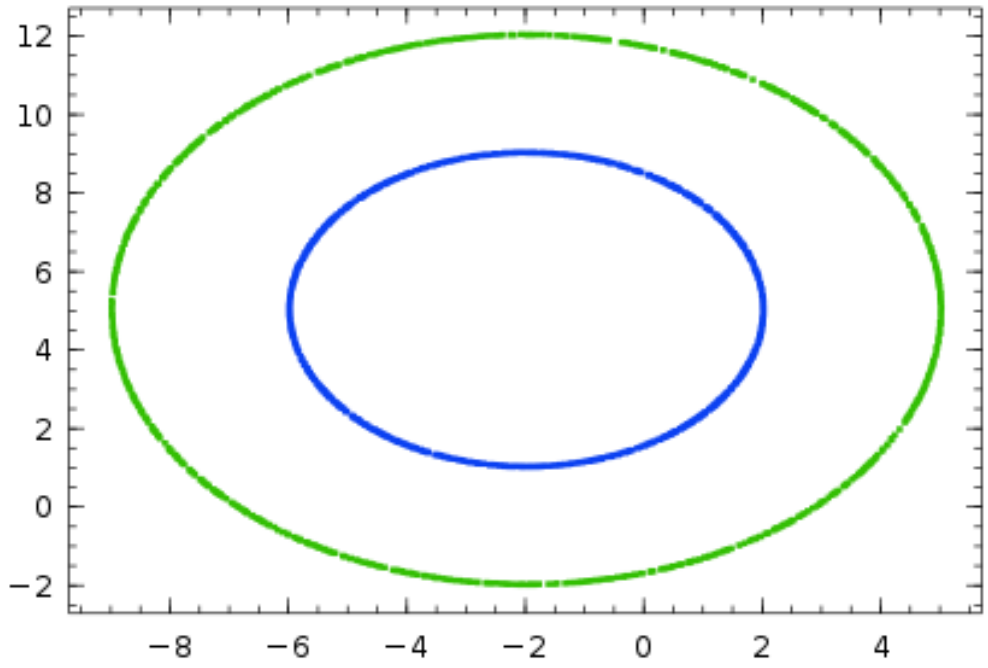
Out[35]:



This is the same partitioning as obtained by `kmeans()`. Let us try Gaussian kernel. A rule of thumb is: if rings are close, use $\sigma < 1$, if rings are apart, use $\sigma > 1$.

```
In [36]: σ=1.0 # 0.1
         W=exp(-pairwise(SqEuclidean(),X)/σ^2)-I
         L=diagm(vec(sum(W,2)))-W
         λ,v=eigs(L,nev=2,which=:SM, v0=ones(m))
         C=ones(Int64,m)
         C[find(v[:,2].>0)]=2
         plotKpartresult(C,X)
```

Out[36]:



2.5 Recursive bipartitioning

2.5.1 Definitions

Let $G = (V, E)$ be a weighted graph with weights ω.
Let $\pi_k = \{V_1, V_2, ..., V_k\}$ be a k-partition of V, with $V_i \neq \emptyset$ for $i = 1, \ldots, k$.
The previous definition of $cut(\pi) \equiv cut(\pi_2)$ extends naturally to k-partition. A **cut** of a partition π_k is

$$cut(\pi_k) = \sum_{i<j} cut(V_i, V_j),$$

where $cut(V_i, V_j)$ is interpreted as a cut of the bipartition of the subgraph of G with vertices $V_1 \cup V_2$.

A **proportional cut** of a partition π_k is

$$pcut(\pi_k) = \sum_{\substack{i,j=1 \\ i<j}}^{k} \left(\frac{cut(V_i, V_j)}{|V_i|} + \frac{cut(V_i, V_j)}{|V_j|} \right) = \sum_{i=1}^{k} \frac{cut(V_i, V \backslash V_i)}{|V_i|}.$$

A **normalized cut** of a partition π_k is

$$ncut(\pi_k) = \sum_{\substack{i,j=1 \\ i<j}}^{k} \left(\frac{cut(V_i, V_j)}{\omega(V_i)} + \frac{cut(V_i, V_j)}{\omega(V_j)} \right) = \sum_{i=1}^{k} \frac{cut(V_i, V \backslash V_i)}{\omega(V_i)}.$$

2.5.2 Fact

If we want to cluster vertices of graph $G = (V, E)$ into k clusters, we can apply the following recursive algorithm:

1. *Initialization*: Compute the bipartition $\pi = \{V_1, V_2\}$ of V. Set the counter $c = 2$.
2. *Recursion*: While $c < k$ repeat
    1. compute the bipartition of each subset of V
    2. among all $(c + 1)$-partitions, choose the one with the smallest $pcut(\pi_{c+1})$ or $ncut(\pi_{c+1})$, respectively
    3. Set $c = c + 1$
3. Stop

There is no guarantee for optimality of this algorithm. Clearly, the opriomal k-partiton may be a subset of one of the discarded partitions.

```
In [ ]:
```

3 Spectral Graph K-partitioning

Instead of using recursive spectral bipartitioning, the graph k-partitioning problem can be solved using k eigenvaectors which correspond to k smallest eigenvalues of Laplaciain matrix or normalized Laplacian matrix, respectively.

Suggested reading is U. von Luxburg, A Tutorial on Spectral Clustering, which includes the quote *"spectral clustering cannot serve as a "black box algorithm" which automatically detects the correct clusters in any given data set. But it can be considered as a powerful tool which can produce good results if applied with care."*

3.1 Prerequisites

The reader should be familiar with k-means algorithm, spectral graph bipartitioning and recursive bipartitioning.

3.2 Competences

The reader should be able to apply graph spectral k-partitioning to data clustering problems.

Credits: The notebook is based on I. Mirošević, Spectral Graph Partitioning and Application to Knowledge Extraction.

3.3 The relaxed problem

Let $G = (V, E)$ be a weighted graph with weights ω, with weights matrix W, Laplacian matrix $L = D - W$, and normalized Laplacian matrix $L_n = D^{-1/2}(D - W)D^{-1/2}$.

Let the k-partition $\pi_k = \{V_1, V_2, ..., V_k\}$, the cut $cut(\pi_k)$, the proportional cut $pcut(\pi_k)$ and the normalized cut $ncut(\pi_k)$ be defined as in the Spectral Graph Bipartitioning notebook.

3.3.1 Definition

Partition vectors of a k-partition π_k are

$$
\begin{aligned}
h_1 &= [\overbrace{1, \cdots, 1}^{|V_1|}, 0, \cdots, 0, \cdots, 0, \cdots, 0]^T \\
h_2 &= [0, \cdots, 0, \overbrace{1, \cdots, 1}^{|V_2|}, \cdots, 0, \cdots, 0]^T \\
&\vdots \\
h_k &= [0, \cdots, 0, 0, \cdots, 0, \cdots, \overbrace{1, \cdots, 1}^{|V_k|}]^T.
\end{aligned}
$$

3.3.2 Facts

1. Set

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_k \end{bmatrix}, \quad x_i = \frac{h_i}{\|h_i\|_2},$$
$$Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_k \end{bmatrix}, \quad y_i = \frac{D^{1/2}h_i}{\|D^{1/2}h_i\|_2}.$$

It holds

$$cut(V_i, V\backslash V_i) = h_i^T(D-W)h_i = h_i^T L h_i, \quad \omega(C_i) = h_i^T D h_i, \quad |C_i| = h_i^T h_i,$$
$$pcut(\pi_k) = \frac{h_1^T L h_1}{h_1^T h_1} + \cdots + \frac{h_k^T L h_k}{h_k^T h_k} = x_1^T L x_1 + \cdots + x_k^T L x_k = \text{trace}(X^T L X),$$
$$ncut(\pi_k) = \frac{h_1^T L h_1}{h_1^T D h_1} + \cdots + \frac{h_k^T L h_k}{h_k^T D h_k} = \text{trace}(Y^T L_n Y).$$

2. The **relaxed** k-partitioning problems are trace-minimization problems,

$$\min_{\pi_k} pcut(\pi_k) \geq \min_{\substack{X^T X = I \\ X \in \mathbb{R}^{n\times k}}} \text{trace}(X^T L X),$$
$$\min_{\pi_k} ncut(\pi_k) \geq \min_{\substack{Y^T Y = I \\ Y \in \mathbb{R}^{n\times k}}} \text{trace}(Y^T L_n Y).$$

3. **Ky-Fan Theorem**: Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix with eigenvalues $\lambda_1 \leq \cdots \leq \lambda_n$. Then
$$\min_{\substack{Z \in \mathbb{R}^{n\times k} \\ Z^T Z = I}} \text{trace}\left(Z^T A Z\right) = \sum_{i=1}^{k} \lambda_i.$$

4. Let $\lambda_1 \leq \cdots \leq \lambda_n$ be the eigenvalues of L with eigenvectors $v^{[1]}, \cdots, v^{[k]}$. The solution of the relaxed proportional cut problem is the matrix $X = \begin{bmatrix} v^{[1]} & \cdots & v^{[k]} \end{bmatrix}$, and it holds $\min_{\pi_k} pcut(\pi_k) \geq \sum_{i=1}^{k} \lambda_i$.

5. Let $\mu_1 \leq \cdots \leq \mu_n$ be the eigenvalues of L_n with eigenvectors $w^{[1]}, \cdots, w^{[k]}$. The solution of the relaxed normalized cut problem is the matrix $Y = \begin{bmatrix} w^{[1]} & \cdots & w^{[k]} \end{bmatrix}$, and it holds $\min_{\pi_k} ncut(\pi_k) \geq \sum_{i=1}^{k} \mu_i$.

6. It remains to recover the k-partition. The k-means algorithm applied to rows of the matrices X or $D^{-1/2}Y$, will compute the k centers and the assignment vector whose i-th component denotes the subset V_j to which the vertex i belongs.

3.3.3 Example - Graph with three clusters

```
In [1]: using Graphs
        using IJuliaPortrayals
        using Clustering
```

```
In [2]: # Sources, targets, and weight
        sn=[1,1,1,2,2,3,2,3,5,6,7,7,8]
        tn=[2,3,4,3,4,4,7,5,6,9,9,8,9]
        wn=[2,3,4,4,5,6,1,1,7,1,3,4,2]
        [sn tn wn]
```

```
Out[2]: 13x3 Array{Int64,2}:
         1  2  2
         1  3  3
         1  4  4
         2  3  4
         2  4  5
         3  4  6
         2  7  1
         3  5  1
         5  6  7
         6  9  1
         7  9  3
         7  8  4
         8  9  2
```

```
In [3]: n=9
        m=length(sn)
        G = inclist(ExVertex, ExEdge{ExVertex}; is_directed=false)
        for i=1:9
            add_vertex!(G, ExVertex(i,"$i"))
        end
        for i=1:m
            add_edge!(G, vertices(G)[sn[i]], vertices(G)[tn[i]])
        end
        for i=1:length(G.inclist)
            for j=1:length(G.inclist[i])
                ei=G.inclist[i][j]
                attrs = attributes(ei, G)
                attrs["label"] = wn[edge_index(ei,G)]
            end
        end
```

```
In [5]: # What is the optimal tripartition?
        GraphViz(to_dot(G),"neato","svg")
```

```
Out[5]: IJuliaPortrayals.GraphViz("graph graphname {\n1 -- 2 [\"label\"=\"2\"]\n1 -- 3 [\"la
```

```
In [6]: L=laplacian_matrix(G,wn)
```

```
Out[6]: 9x9 Array{Int64,2}:
          9  -2  -3  -4   0   0   0   0   0
         -2  12  -4  -5   0   0  -1   0   0
         -3  -4  14  -6  -1   0   0   0   0
         -4  -5  -6  15   0   0   0   0   0
          0   0  -1   0   8  -7   0   0   0
          0   0   0   0  -7   8   0   0  -1
```

```
  0  -1   0   0   0   0   8  -4  -3
  0   0   0   0   0   0  -4   6  -2
  0   0   0   0   0  -1  -3  -2   6
```

```
In [7]: # Normalized Laplacian matrix
        function normalized(L::Matrix)
            D=1.0./sqrt(diag(L))
            n=length(D)
            Ln=map(Float64,Symmetric([L[i,j]*D[i]*D[j] for i=1:n,j=1:n]))
            # Set diagonal exactly to 1
            for i=1:n
                Ln[i,i]=1.0
            end
            Ln
        end
```

```
Out[7]: normalized (generic function with 1 method)
```

```
In [8]: Ln=normalized(L)
```

```
Out[8]: 9x9 Array{Float64,2}:
          1.0       -0.19245   -0.267261   ...   0.0        0.0        0.0
         -0.19245    1.0       -0.308607       -0.102062   0.0        0.0
         -0.267261  -0.308607   1.0             0.0        0.0        0.0
         -0.344265  -0.372678  -0.414039        0.0        0.0        0.0
          0.0        0.0       -0.0944911       0.0        0.0        0.0
          0.0        0.0        0.0         ...   0.0        0.0       -0.144338
          0.0       -0.102062   0.0             1.0       -0.57735   -0.433013
          0.0        0.0        0.0            -0.57735    1.0       -0.333333
          0.0        0.0        0.0            -0.433013  -0.333333   1.0
```

```
In [9]: # Proportional cut, the clustering is visible in the components of v_2 and v_3
        λ,Y=eigs(L,nev=3,which=:SM, v0=ones(n))
```

```
Out[9]: ([4.934324553889581e-16,0.7885231802373419,1.2104922881233322],
        9x3 Array{Float64,2}:
         -0.333333   0.383624   -0.121476
         -0.333333   0.323492   -0.129956
         -0.333333   0.344649   -0.0752859
         -0.333333   0.367298   -0.115116
         -0.333333  -0.0953074   0.612081
         -0.333333  -0.147422    0.60443
         -0.333333  -0.355516   -0.282486
         -0.333333  -0.424836   -0.311411
         -0.333333  -0.395981   -0.18078  ,

        3,1,9,[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0])
```

```
In [10]: out=kmeans(Y',3)
```

```
Out[10]: Clustering.KmeansResult{Float64}(3x3 Array{Float64,2}:
          -0.333333  -0.333333  -0.333333
           0.354766  -0.392111  -0.121365
          -0.110459  -0.258226   0.608256,[1,1,1,1,3,3,2,2,2],[0.0009541957420936553,0.0013
```

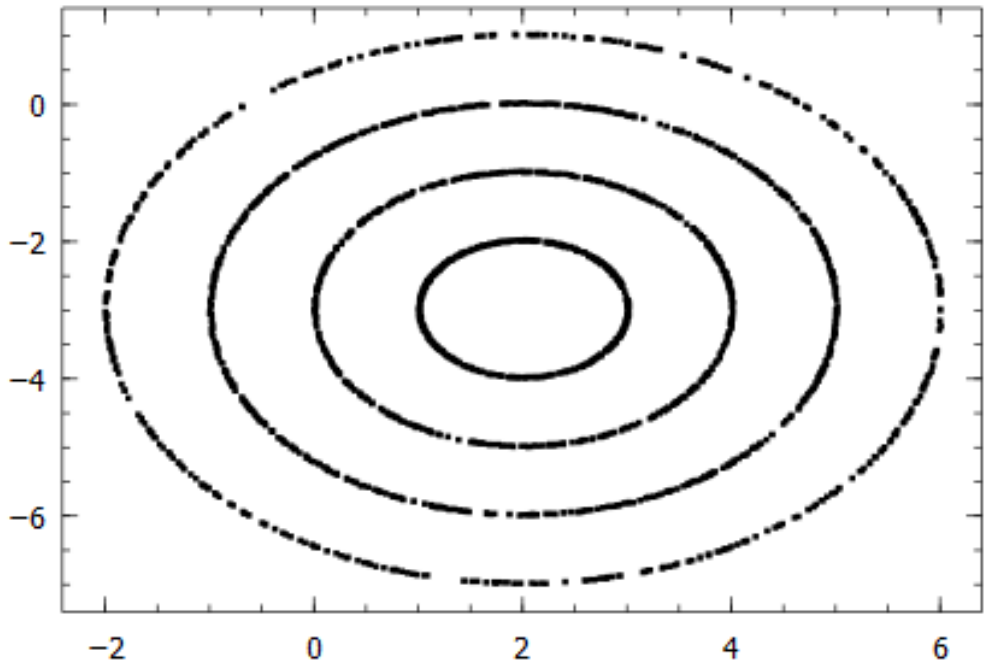
```
In [11]: # Normalized cut
         # Lanczos cannot be used for "smallest in magnitude" evals of singular matrix
         # The above case is exception, rather than rule.
         # λ,Y=eigs(Ln,nev=3,which=:SM)
         μ,Y=eig(Ln)
         Y=Y[:,1:3]
         D=sqrt(diag(L))
         Y=diagm(1.0./D)*Y
         out=kmeans(Y',3)

Out[11]: Clustering.KmeansResult{Float64}(3x3 Array{Float64,2}:
          -0.107833  -0.107833   -0.107833
          -0.144522   0.0909339  -0.104042
           0.131546   0.011189   -0.19892 ,[2,2,2,2,3,3,1,1,1],[6.017933267283421e-5,0.0001
```

3.3.4 Example - Concentric rings

```
In [13]: using Winston
         using Colors
         using Distances

In [14]: function plotKpartresult(C::Vector,X::Array)
             p=FramedPlot()
             for j=1:k
                 # Random color
                 col=RGB(rand(),rand(),rand())
                 p1=Points(X[1,find(C.==j)],
                 X[2,find(C.==j)],"color",col,symbolkind="dot")
                 add(p,p1)
             end
             p
         end

Out[14]: plotKpartresult (generic function with 1 method)

In [15]: # Generate concentric rings
         k=4
         # Center
         center=[rand(-5:5);rand(-5:5)]
         # Radii
         radii=randperm(k)[1:k]
         # Number of points in circles
         sizes=rand(500:1000,k)
         center,radii,sizes

Out[15]: ([2,-3],[2,3,4,1],[537,720,501,562])

In [16]: # Points
         m=sum(sizes)
         X=Array(Float64,2,m)
         first=0
         last=0
```

```
        for j=1:k
            first=last+1
            last=last+sizes[j]
            # Random angles
            ϕ=2*π*rand(sizes[j])
            for i=first:last
                l=i-first+1
                X[:,i]=center+radii[j]*[cos(ϕ[l]);sin(ϕ[l])]+(rand(2)-0.5)/50
            end
        end
        Winston.plot(X[1,:],X[2,:],".")
```

Out[16]:




```
In [17]: S=pairwise(SqEuclidean(),X)
         # S=pairwise(Cityblock(),X)
         β=10
```

Out[17]: 10

```
In [18]: W=exp(-β*S)
         D=vec(sum(W,2))
         L=diagm(D)-W
         Ln=normalized(L)
```

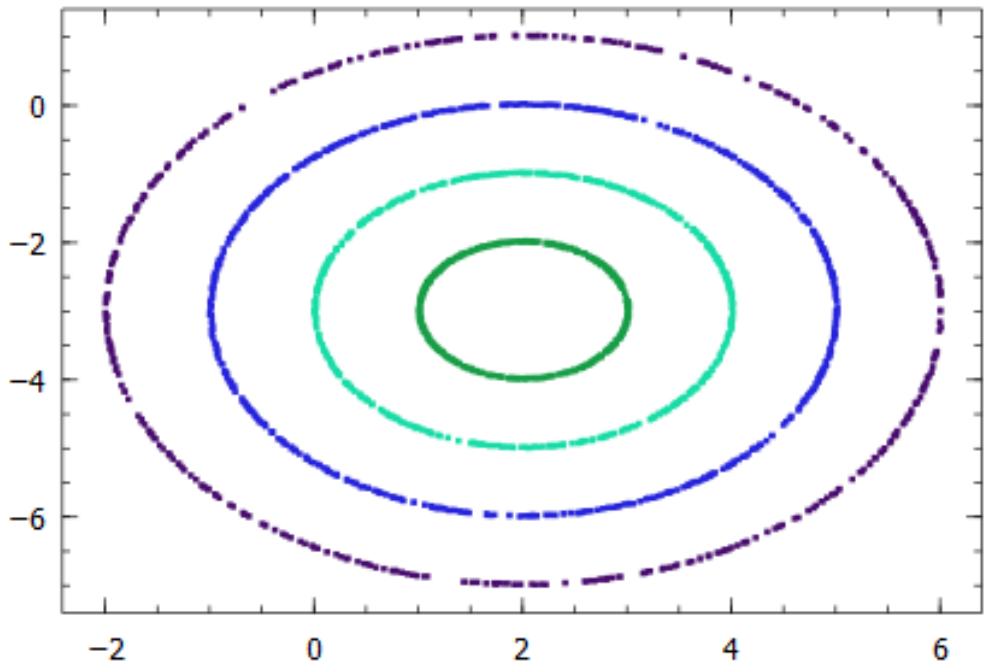
Out[18]:
```
2320x2320 Array{Float64,2}:
  1.0          -1.209e-71    -1.92834e-39  ...  -3.03953e-36  -2.26561e-22
 -1.209e-71     1.0          -1.64755e-37      -9.66287e-11  -4.5086e-27
```

```
-1.92834e-39  -1.64755e-37   1.0             -1.37265e-35  -7.24795e-7
-1.10739e-67  -4.64815e-7   -3.7341e-19      -2.31656e-18  -1.54339e-17
-4.17992e-65  -2.46139e-10  -9.45763e-15     -3.75481e-21  -3.33328e-15
-0.0430634    -2.63725e-71  -7.65042e-38  ...  -1.9703e-36   -1.72084e-21
-9.94939e-39  -1.48386e-31  -4.85096e-71     -9.26147e-11  -6.14966e-41
-4.20003e-34  -8.56239e-37  -6.4158e-72      -9.5739e-13   -3.91767e-41
-4.42458e-71  -0.0184151    -7.57074e-32     -1.68822e-12  -4.22441e-24
-4.62107e-19  -8.20901e-52  -7.36016e-68     -1.71364e-19  -8.31121e-38
-1.58201e-65  -5.02428e-10  -3.67209e-15  ...  -5.699e-21    -1.79625e-15
-7.92223e-51  -2.89374e-20  -6.82861e-68     -8.31717e-8   -1.78269e-40
-3.90653e-70  -5.5743e-5    -1.52412e-23     -1.64354e-16  -4.51975e-20
 ⋮                                          ⋱
-1.11738e-40  -7.05814e-8   -8.67089e-18     -3.89663e-7   -2.47753e-9
-8.0823e-7    -3.83567e-41  -1.30473e-22     -6.47989e-18  -6.51872e-9
-3.99139e-33  -1.80783e-13  -6.57497e-38  ...  -0.012678     -1.16312e-18
-1.94695e-14  -9.75587e-35  -3.96021e-10     -1.28889e-19  -0.00229839
-1.39542e-28  -8.36435e-21  -5.09602e-7      -2.1535e-15   -0.00570527
-1.34689e-35  -4.28869e-11  -1.42933e-35     -0.0191767    -1.1088e-17
-1.21009e-20  -7.45313e-29  -3.37741e-7      -2.12712e-18  -0.0196547
-1.94128e-29  -4.21748e-20  -4.11408e-7   ...  -3.56207e-15  -0.00404644
-2.50425e-15  -1.9623e-31   -6.40343e-40     -7.79185e-9   -2.87603e-18
-4.92935e-7   -1.75549e-40  -6.2295e-29      -1.13234e-15  -4.33194e-12
-3.03953e-36  -9.66287e-11  -1.37265e-35      1.0          -8.38377e-18
-2.26561e-22  -4.5086e-27   -7.24795e-7      -8.38377e-18   1.0
```

In [19]:
```
# Normalized Laplacian
λ,Y=eigs(Ln,nev=k,which=:SM, v0=ones(m))
@show λ
Y=diagm(1.0./sqrt(D))*Y
out=kmeans(Y',k)
plotKpartresult(out.assignments,X)
```

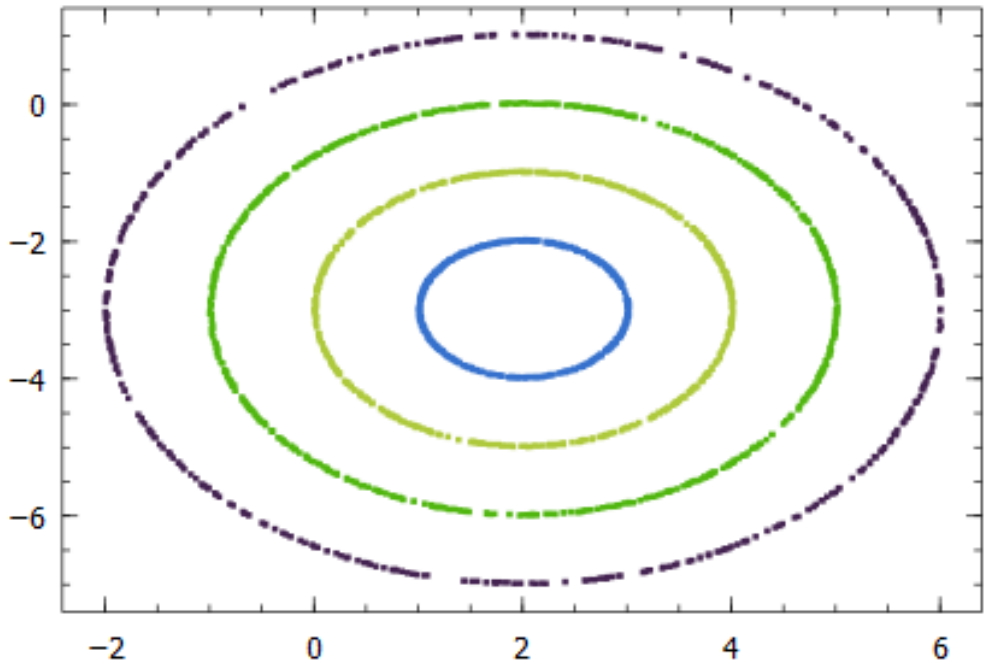
```
λ = [-3.112883198138873e-16,2.827959511967512e-5,0.00010276007138703098,0.00016316023116458
```

Out[19]:



In [20]:
```
# Laplacian
λ,Y=eigs(L,nev=k,which=:SM, v0=ones(m))
out=kmeans(Y',k)
plotKpartresult(out.assignments,X)
```

Out[20]:



Caveat! There is a little bit of cheating here.

In [ ]:

4 Spectral Partitioning of Bipartite Graphs

Typical example of bipartite graph is a graph obtained from a collection of documents presented as a *term* × *document* matrix.

4.1 Prerequisites

The reader should be familiar with k-means algorithm and spectral graph partitioning theory and algorithms.

4.2 Competences

The reader should be able to apply spectral partitioning of bipartite graphs to data clustering problems.
Credits: The notebook is based on I. Mirošević, Spectral Graph Partitioning and Application to Knowledge Extraction.

4.3 Definitions

Undirected bipartite graph G is a triplet $G = (T, D, E)$, where $T = \{t_1, \cdots, t_m\}$ and $D = \{d_1, ..., d_n\}$ are two sets of vertices and $E = \{(t_i, d_j) : t_i \in R, d_j \in D\}$, is a set of edges.
G is **weighted** if there is weight $\omega(e)$ associated with each edge $e \in E$.
For example, D is a set of documents, T is a set of terms (words) and edge $e = (t_i, d_j)$ exists if document d_j contains term t_i. Weight $\omega(e)$ can be number of appearances of the term t_i in the document d_j.
A **term-by-document-matrix** is a matrix $A \in \mathbb{R}^{m \times n}$ with $A_{ij} = \omega((t_i, d_j))$.

4.4 Facts

1. The weight matrix of G is $W = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$.

2. The Laplacian matrix of G is $L = \begin{bmatrix} \Delta_1 & -A \\ -A^T & \Delta_2 \end{bmatrix}$, where Δ_1 and Δ_2 are diagonal matrices with elements $\Delta_{1,ii} = \sum_{j=1}^{n} A_{ij}$ for $i = 1, \ldots, m$, and $\Delta_{1,jj} = \sum_{i=1}^{m} A_{ij}$ for $j = 1, \ldots, n$.

3. The normalized Laplacian matrix of G is $L_n = \begin{bmatrix} I & -\Delta_1^{-\frac{1}{2}} A \Delta_2^{-\frac{1}{2}} \\ -\Delta_2^{-\frac{1}{2}} A^T \Delta_1^{-\frac{1}{2}} & I \end{bmatrix} \equiv \begin{bmatrix} I & -A_n \\ -A_n^T & I \end{bmatrix}$.

4. Let λ be an eigenvalue of L_n with an eigenvector $w = \begin{bmatrix} u \\ v \end{bmatrix}$, where $u \in \mathbb{R}^m$ $v \in \mathbb{R}^n$. Then $L_n w = \lambda w$ implies $A_n v = (1-\lambda)u$ and $A_n^T u = (1-\lambda)v$. Vice versa, if (u, σ, v) is a singular triplet of A_n, then $1 - \sigma$ is an eigenvalue of L_n with (non-unit) eigenvector $w = \begin{bmatrix} u \\ v \end{bmatrix}$.

5. The second largest singular value of A_n corresponds to the second smallest eigenvalue of L_n, and computing the former is numerically more stable.

6. *Bipartitioning algorithm* is the following:

    1. For given A compute A_n.
    2. Compute singular vectors of A_n, $u^{[2]}$ and $v^{[2]}$, which correspond to the second largest singular value, $\sigma_2(A_n)$.
    3. Assign the partitions $T = \{T_1, T_2\}$ and $D = \{D_1, D_2\}$ according to the signs of $u^{[2]}$ and $v^{[2]}$. The pair (T, D) is now partitioned as $\{(T_1, D_1), (T_2, D_2)\}$.

7. *Recursive bipartitioning algorithm* is the following:

    1. Compute the bipartition $\pi = \{(T_1, D_1), (T_2, D_2)\}$ of (T, D). Set the counter $c = 2$.
    2. While $c < k$ repeat
        1. compute bipartitions of each of the subpartitions of (T, D),
        2. among all $(c+1)$-subpartitions, choose the one with the smallest $pcut(\pi_{c+1})$ or $ncut(\pi_{c+1})$, respectively.
    3. Set $c = c + 1$
    4. Stop

8. *Multipartitioning algorithm* is the following:

    1. For given A compute A_n.
    2. Compute k left and right singular vectors, $u^{[1]}, \ldots, u^{[k]}$ and $v^{[1]}, \ldots, v^{[k]}$, which correspond to k largest singular values $\sigma_1 \geq \cdots \geq \sigma_k$ of A_n.
    3. Partition the rows of matrices $\Delta_1^{-\frac{1}{2}} \begin{bmatrix} u^{[1]} & \ldots & u^{[k]} \end{bmatrix}$ and $\Delta_2^{-\frac{1}{2}} \begin{bmatrix} v^{[1]} & \ldots & v^{[k]} \end{bmatrix}$ with the k-means algorithm.

4.4.1 Example - Small term-by- document matrix

```
In [1]: using Graphs
        using IJuliaPortrayals
        using Clustering
```

```
In [2]: # Sources, targets, and weight
        dn=[6,6,6,7,7,7]
        tn=[1,2,3,2,4,5]
        wn=[3,1,2,3,2,3]
        [dn tn wn]
```

```
Out[2]: 6x3 Array{Int64,2}:
         6  1  3
         6  2  1
         6  3  2
         7  2  3
         7  4  2
         7  5  3
```

```
In [3]: G = inclist(ExVertex, ExEdge{ExVertex}; is_directed=false)
        for i=1:5
```

```
        add_vertex!(G, ExVertex(i,"$i"))
        attrs=attributes(G.vertices[i],G)
        attrs["label"]="Term $i"
    end

    for i=1:2
        add_vertex!(G, ExVertex(i+5,"$(i+5)"))
        attrs=attributes(G.vertices[i+5],G)
        attrs["label"]="Document $i"
    end

    for i=1:6
        add_edge!(G, vertices(G)[dn[i]], vertices(G)[tn[i]])
    end
    for i=1:length(G.inclist)
        for j=1:length(G.inclist[i])
            ei=G.inclist[i][j]
            attrs = attributes(ei, G)
            attrs["label"] = wn[edge_index(ei,G)]
        end
    end
```

In [4]:
```
# Plot
GraphViz(to_dot(G),"fdp","svg")
```

Out[4]:
```
IJuliaPortrayals.GraphViz("graph graphname {\n1 [\"label\"=\"Term 1\"]\n1 -- 6 [\"la
```

In [5]:
```
W=weight_matrix(G,wn)
```

Out[5]:
```
7x7 Array{Int64,2}:
 0  0  0  0  0  3  0
 0  0  0  0  0  1  3
 0  0  0  0  0  2  0
 0  0  0  0  0  0  2
 0  0  0  0  0  0  3
 3  1  2  0  0  0  0
 0  3  0  2  3  0  0
```

In [6]:
```
A=W[1:5,6:7]
Δ1=sqrt(sum(A,2))
Δ2=sqrt(sum(A,1))
An=map(Float64,[A[i,j]/(Δ1[i]*Δ2[j]) for i=1:size(A,1), j=1:size(A,2)])
```

Out[6]:
```
5x2 Array{Float64,2}:
 0.707107  0.0
 0.204124  0.53033
 0.57735   0.0
 0.0       0.5
 0.0       0.612372
```

In [7]:
```
# The partitioning - explain the results!
U,σ,V=svd(An)
```

```
Out[7]: (
        5x2 Array{Float64,2}:
         -0.46291    0.604743
         -0.534522  -0.218218
         -0.377964   0.493771
         -0.377964  -0.370328
         -0.46291   -0.453557,

        [0.9999999999999999,0.8838834764831844],
        2x2 Array{Float64,2}:
         -0.654654   0.755929
         -0.755929  -0.654654)
```

4.4.2 Example - Sets of points

```
In [9]: using Gadfly
        using Images
```

```
In [10]: ?sprand
```

```
search: sprand sprandn sprandbool StepRange sparse2adjacencylist
```

```
Out[10]:
```

```
..  sprand([rng,] m,n,p [,rfn])

Create a random ``m`` by ``n`` sparse matrix, in which the probability of any
element being nonzero is independently given by ``p`` (and hence the mean
density of nonzeros is also exactly ``p``). Nonzero values are sampled from
the distribution specified by ``rfn``. The uniform distribution is used in
case ``rfn`` is not specified. The optional ``rng`` argument specifies a
random number generator, see :ref:`Random Numbers <random-numbers>`.
```

```
In [11]: # Define sizes
         m=[200,100,100]
         n=[100,200,100]
         density=[0.5,0.7,0.4]
         A=Array(Any,3)
         for i=1:3
             A[i]=sprand(m[i],n[i],density[i])
         end
         B=blkdiag(A[1],A[2],A[3])
```

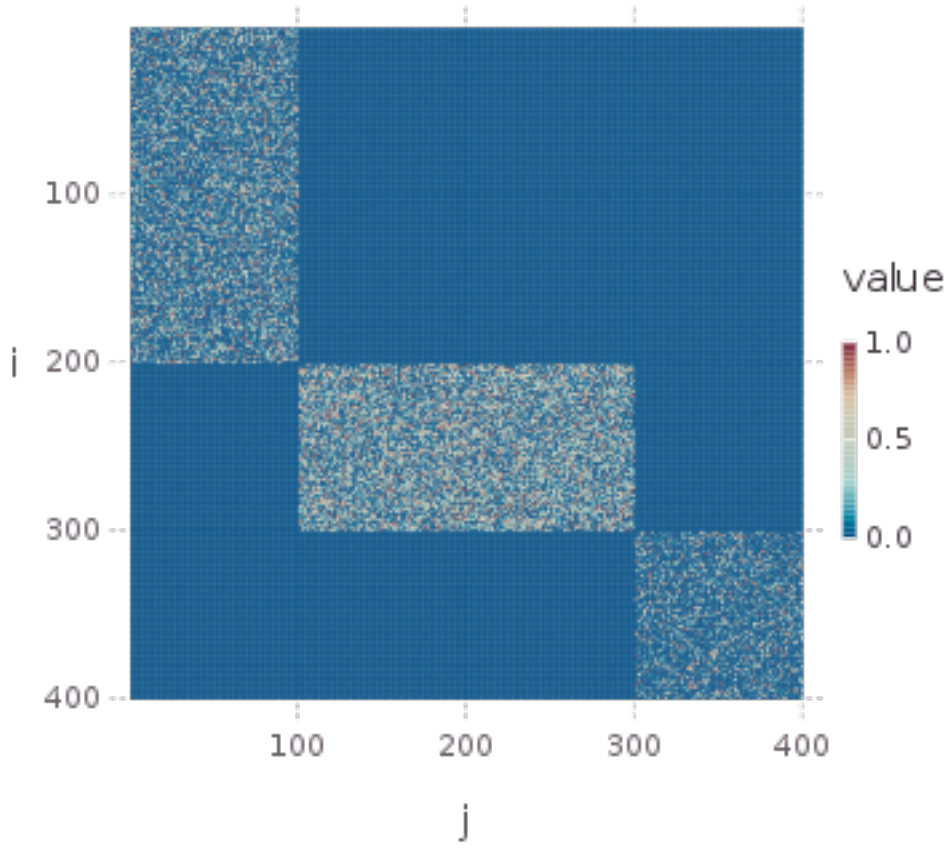
```
Out[11]: 400x400 sparse matrix with 27884 Float64 entries:
                 [1  ,  1]  =  0.185978
                 [2  ,  1]  =  0.930463
                 [3  ,  1]  =  0.951578
                 [6  ,  1]  =  0.288271
                 [8  ,  1]  =  0.144443
                 [9  ,  1]  =  0.698419
```

```
[12 ,   1]  =  0.975249
[13 ,   1]  =  0.429804
[14 ,   1]  =  0.589122
[15 ,   1]  =  0.525998
⋮
[374, 400]  =  0.147314
[381, 400]  =  0.387642
[388, 400]  =  0.777484
[389, 400]  =  0.0377414
[390, 400]  =  0.64925
[393, 400]  =  0.423817
[395, 400]  =  0.67256
[396, 400]  =  0.38278
[398, 400]  =  0.172354
[399, 400]  =  0.231409
[400, 400]  =  0.252861
```

In [12]:
```
pB=spy(B)
draw(Gadfly.PNG("files/pB.png", 4inch, 4inch), pB)
```

In [13]:
```
using Images
load("files/pB.png")
```

Out[13]:



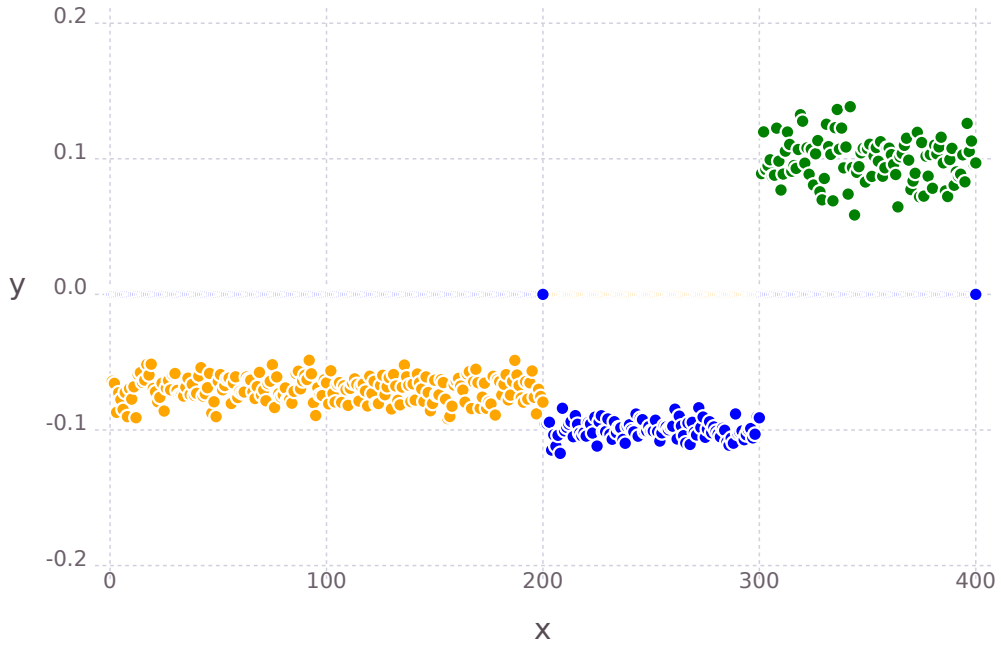

```
In [14]: # The structure of singular vectors reflects the blocks
         U,σ,V=svds(B,nsv=3)
```

```
Out[14]: (
         400x3 Array{Float64,2}:
           1.77862e-17  -0.0640713     3.92523e-17
           1.69812e-17  -0.0656449     4.41589e-17
           1.93962e-17  -0.0870069    -1.47196e-17
           1.03881e-17  -0.0726003    -1.47196e-17
           2.7331e-17   -0.0776765    -1.47196e-17
           5.55819e-18  -0.0848381     3.43458e-17
           9.27643e-18  -0.0723616     4.41589e-17
           1.18064e-17  -0.0901929     7.35981e-17
           1.80162e-17  -0.0695351     1.47196e-17
           5.0982e-18   -0.0772642    -9.81308e-18
           2.15811e-17  -0.0681972    -4.41589e-17
           1.69812e-17  -0.0909592     2.94392e-17
           9.96641e-18  -0.0593217     5.39719e-17
           ⋮
```

```
  3.73935e-18  -2.10951e-17   0.107678
 -6.26427e-18   1.21866e-17   0.0804496
 -1.97112e-18   2.82258e-17   0.0903584
  1.04074e-18   5.58104e-17   0.087246
  2.17761e-18   2.50479e-17   0.08859
  8.28637e-19   4.1659e-17    0.102919
  4.63162e-18   7.16409e-17   0.0829699
  3.73531e-18   2.22698e-18   0.126084
 -1.45603e-18   5.07905e-17   0.105305
 -4.56604e-18   5.60497e-17   0.113088
 -1.87619e-19   2.2772e-17    0.0961044
 -2.98872e-18   6.32192e-17   0.096924   ,

[49.51138125187213,35.58402467955015,20.7459761713298],
400x3 Array{Float64,2}:
  2.75993e-17  -0.117189      0.0
  2.14661e-18  -0.110327     -2.94392e-17
  4.44655e-18  -0.103578      9.81308e-18
  2.03928e-17  -0.0994029    -1.96262e-17
  1.87062e-17  -0.0861458    -1.47196e-17
  1.57929e-17  -0.105235     -5.88785e-17
  1.22663e-17  -0.101038     -6.86915e-17
  8.58644e-18  -0.091924     -9.81308e-18
  1.50263e-17  -0.0975074     0.0
  3.08192e-17  -0.0954747    -6.86915e-17
  1.99328e-18  -0.101718      1.96262e-17
 -6.13317e-18  -0.0902134     9.81308e-18
  2.86726e-17  -0.0936709     4.90654e-17
  ⋮
 -3.11817e-18   6.07276e-17   0.11749
  1.40123e-18   4.44269e-17   0.0834493
 -4.72356e-18   3.30756e-17   0.104671
 -2.28107e-18   1.28458e-16   0.10052
 -3.35475e-18   7.92923e-18   0.0893222
 -8.19683e-18   7.72581e-17   0.11439
 -7.02882e-18   1.05162e-16   0.0782578
 -2.28871e-19   3.84144e-17   0.0811584
 -1.93017e-18   1.56015e-17   0.0833486
 -3.38994e-18   6.06853e-17   0.102537
 -4.82898e-20   5.59232e-17   0.105658
 -1.65241e-18   1.51555e-17   0.0876757  ,

6,2,34,[-0.0304829,0.0678488,0.122835,0.0465063,-0.0822635,-0.0320249,-0.134768,-0
```

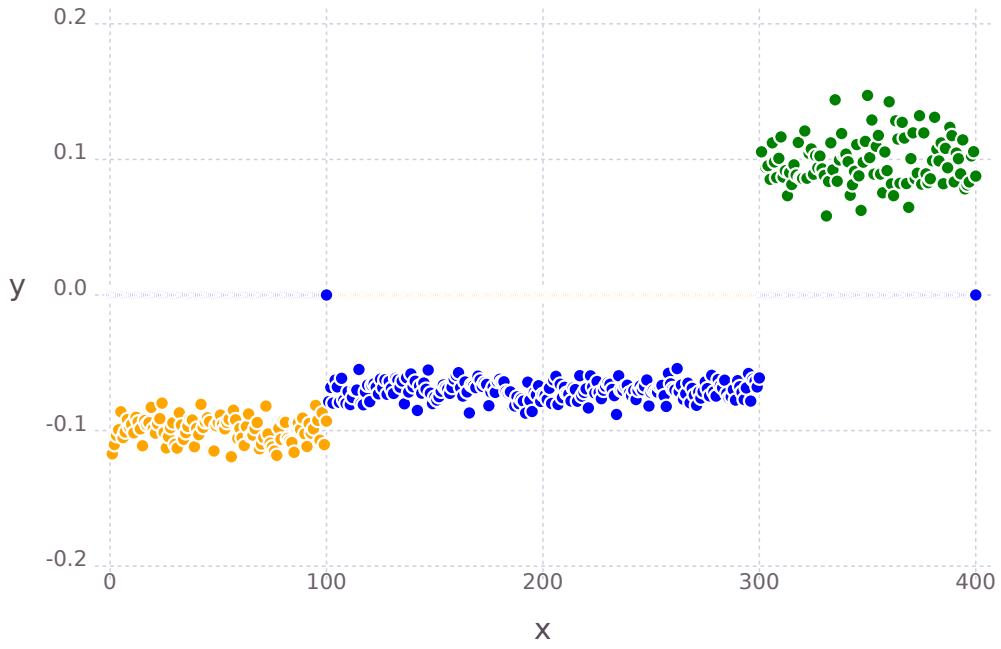
In [15]:
```
# Plot the first three left singular vectors
k=size(B,1)
x=collect(1:k)
Gadfly.plot(layer(x=x,y=U[:,1],Geom.point,Theme(default_color=colorant"blue")),
layer(x=x,y=U[:,2], Geom.point,Theme(default_color=colorant"orange")),
layer(x=x,y=U[:,3], Geom.point,Theme(default_color=colorant"green")))
```

Out[15]:



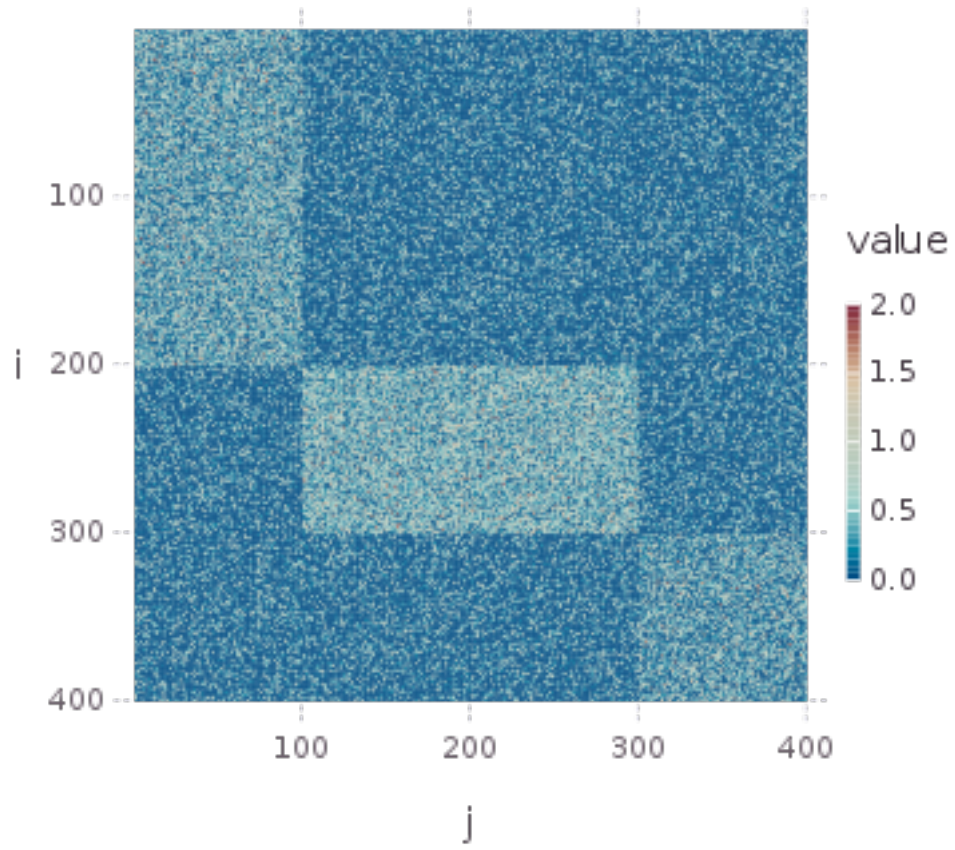
In [16]:
```
# Plot the first three right singular vectors
Gadfly.plot(layer(x=x,y=V[:,1],Geom.point,Theme(default_color=colorant"blue")),
layer(x=x,y=V[:,2], Geom.point,Theme(default_color=colorant"orange")),
layer(x=x,y=V[:,3], Geom.point,Theme(default_color=colorant"green")))
```

Out[16]:



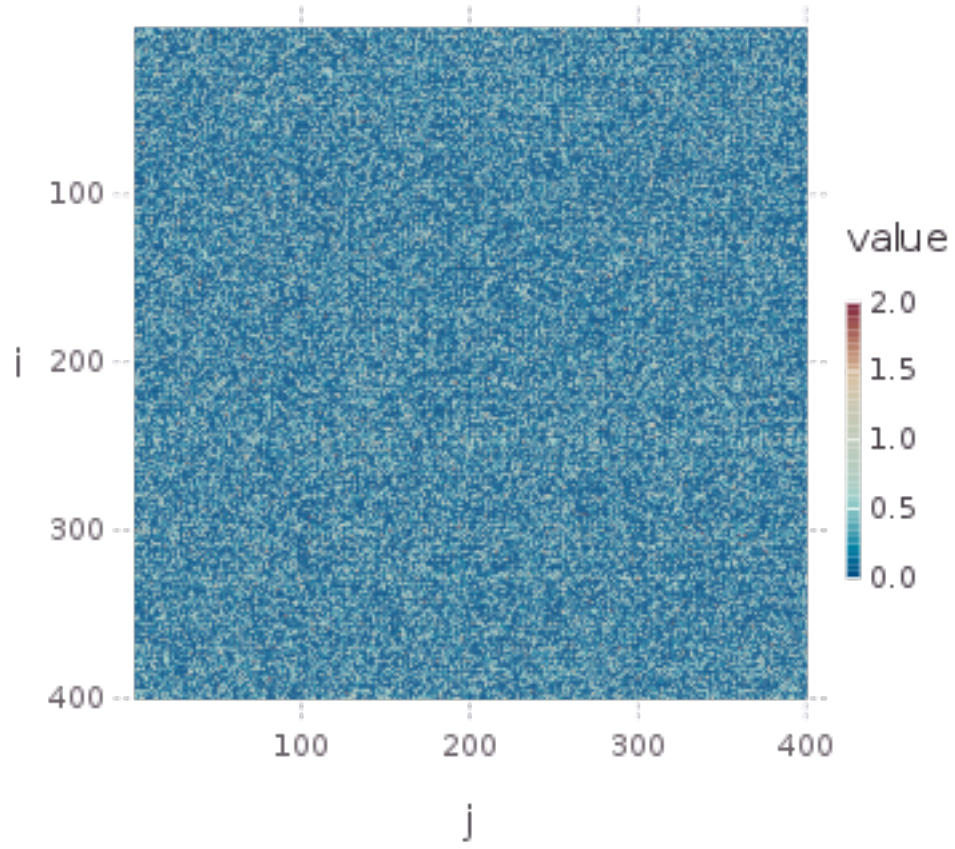
In [17]:
```
# Add random noise
noise=sprand(k,k,0.3)
C=B+noise
pC=spy(C)
draw(Gadfly.PNG("files/pC.png", 4inch, 4inch), pC)
load("files/pC.png")
```

Out[17]:



In [18]:
```
# Apply random permutation to rows and columns of C
D=C[randperm(k),randperm(k)]
pD=spy(D)
draw(Gadfly.PNG("files/pD.png", 4inch, 4inch), pD)
load("files/pD.png")
```

Out[18]:




```
In [19]: # Given D, can we recover C with spectral partitioning?
         U,σ,V=svds(D,nsv=3)
```

```
Out[19]: (
         400x3 Array{Float64,2}:
          -0.0461542  -0.0417776    0.0192639
          -0.0456713  -0.0498393    0.0376449
          -0.0413298  -0.0449522    0.0280559
          -0.0636116   0.0734009    0.00794167
          -0.0710171   0.0720333    0.00821326
          -0.0638242   0.068111     0.0250511
          -0.0419511  -0.0357578    0.0260137
          -0.0403812  -0.0319938   -0.105762
          -0.0494997  -0.0767571    0.0404917
          -0.0479923  -0.0360301    0.013715
          -0.0679837   0.0646284    0.0226709
          -0.0446832  -0.0480517    0.0435775
          -0.0473182  -0.057036     0.0437562
           ⋮
```

```
 -0.0395873  -0.0202009   -0.0966264
 -0.0421794  -0.00977905  -0.0781569
 -0.044084   -0.0446987    0.0156368
 -0.0434713  -0.0503169    0.0339073
 -0.0405218  -0.0164256   -0.0832686
 -0.0395533  -0.0218991    0.010399
 -0.0702424   0.0648613    0.0264853
 -0.070037    0.0689225    0.0138272
 -0.0469718  -0.0518813    0.0476648
 -0.0360289  -0.0188509   -0.0844173
 -0.0717576   0.07616      0.0138428
 -0.0710651   0.0680696    0.0274318 ,

[98.97804751048824,42.34783385097339,25.3545677207195],
400x3 Array{Float64,2}:
 -0.0545155   0.0446359   0.0275708
 -0.0572219   0.0431168  -0.0128002
 -0.0593965   0.0464754   0.00352814
 -0.0515447   0.0425863   0.0315156
 -0.0561552   0.05595     0.00711859
 -0.0594272   0.0483196   0.0267319
 -0.0510914  -0.0737743   0.0404743
 -0.0378788  -0.0286392  -0.0595333
 -0.0505157   0.0435187   0.000107782
 -0.0462244  -0.0723963   0.0360484
 -0.0598628  -0.0684754   0.0575358
 -0.0535399   0.0484217   0.00825927
 -0.0534496   0.0468582   0.0233923
  ⋮
 -0.0510525   0.0341538   0.00415412
 -0.0550226   0.0385065  -0.00272286
 -0.0612558   0.0436634   0.0104033
 -0.05761     0.0488126   0.0226779
 -0.0538432   0.0488656   0.0236288
 -0.0460606  -0.019731   -0.104731
 -0.0483677   0.0274777   0.00558441
 -0.0534077   0.0531874  -0.00304568
 -0.0475775   0.0254482   0.0168455
 -0.0382273  -0.0106551  -0.0934895
 -0.0350081  -0.0301584  -0.0887265
 -0.0549852   0.0592196   0.00136959 ,

6,3,42,[0.119632,-0.478792,-0.019949,-0.182076,-0.0939853,-0.190739,0.251286,0.2142
```

```
In [20]: # Kmeans on rows and U and V
         using Clustering
```

```
In [21]: outU=kmeans(U',3)
```

```
Out[21]: Clustering.KmeansResult{Float64}(3x3 Array{Float64,2}:
          -0.038058   -0.0686002  -0.0435945
```

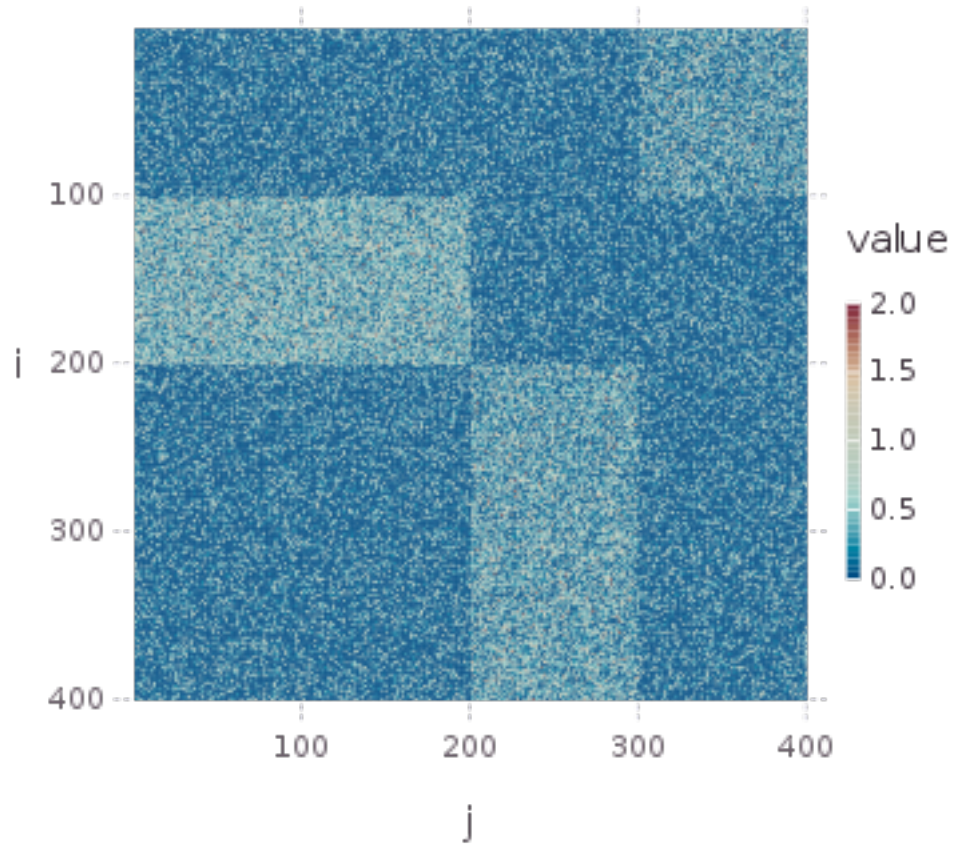
```
 -0.0142882   0.0696828  -0.0483921
 -0.0882487   0.0172062   0.0249332,[3,3,3,2,2,2,3,1,3,3  ...  3,3,1,3,2,2,3,1,2,2]
```

In [22]:
```
outV=kmeans(V',3)
```

Out[22]:
```
Clustering.KmeansResult{Float64}(3x3 Array{Float64,2}:
 -0.0540085  -0.052034   -0.0375294
  0.0431794  -0.0742751  -0.0217079
  0.0112421   0.0389556  -0.0865216,[1,1,1,1,1,1,2,3,1,2  ...  1,1,1,3,1,1,1,3,3,1]
```

In [23]:
```
E=D[sortperm(outU.assignments),sortperm(outV.assignments)]
pE=spy(E)
draw(Gadfly.PNG("files/pE.png", 4inch, 4inch), pE)
load("files/pE.png")
```

Out[23]:



In [ ]:

5 Sparse + Low-Rank Splitting

Suppose we are given a data matrix A, and know that it has a form

$$A = L + S,$$

where L is of low-rank and S is sparse, but we know neither the rank of L nor the non-zero entries of S.

Can we recover L and S?

The answer is YES, with high probability and with an efficient algorithm.

Sparse + Low-rank splitting can be successfully applieed to video surveilance, face recognition, latent semantic indexing, and ranking and collaborative filtering.

5.1 Prerequisites

The reader should be familiar with linear algebra concepts, particularly SVD and its properties and algorithms.

5.2 Competences

The reader should be able to apply sparse + low-rank splitting to real problems.

5.3 References

For more details see E. J. Candes, X. Li, Y. Ma, and J. Wright, Robust Principal Component Analysis?

Credits: The author wishes to thank Dimitar Ninevski, a former IAESTE intern, for collecting and preparing some of the material.

5.4 Definitions

Let $A \in \mathbb{R}^{m \times n}$ have rank r, and let $A = U\Sigma V^T$ be its SVD.

The **nuclear norm** of A is $\|A\|_* = \sum_{i=1}^{r} \sigma_i(A)$.

Let $\|A\|_1 = \sum_{i,j} |A_{ij}|$ denote the 1-norm of A seen as a long vector.

Let $\|A\|_\infty = \max_{i,j} |A_{ij}|$ denote the ∞-norm of A seen as a long vector.

Given $\tau > 0$, the **shrinkage operator** $\mathcal{S}_\tau : \mathbb{R} \to \mathbb{R}$ is defined by $\mathcal{S}_\tau(x) = \text{sign}(x) \max\{|x| - \tau, 0\}$, and is extended to matrices by applying it to each element.

Given $\tau > 0$, the **singluar value thresholding operator** is $\mathcal{D}_\tau(A) = U\mathcal{S}_\tau(\Sigma)V^T$.

5.5 Facts

Let $A = L + S$ be the splitting that we are looking for.

1. The problem can be formulated as $\underset{\text{rank}(L) \leq k}{\arg\min} \|A - L\|_2$.

2. The problem makes sense if the **incoherence conditions**

$$\max_i \|U_{:,1:r}^T e_i\|_2^2 \leq \frac{\mu r}{m}, \quad \max_i \|V_{:,1:r}^T e_i\|_2^2 \leq \frac{\mu r}{n}, \quad \|UV^T\|_\infty \leq \sqrt{\frac{\mu r}{m \cdot n}},$$

hold for some parameter μ.

3. If the incoherence conditions are satisfied, the **Principal Component Pursuit estimate**,

$$\begin{array}{l} \arg\min \\ L+S=A \end{array} \|L\|_* + \lambda \|S\|_2,$$

exactly recovers L and S.

4. *Principal Component Pursuit by Alternating Directions* algorithm finds the above estimate

    1. *Initialization*: $S = 0$, $Y = 0$, $L = 0$, $\mu > 0$, $\delta = 10^{-7}$.
    2. *Iterations*: while $\|A - L - S\|_F > \delta \|A\|_F$ repeat
        1. *SV Thresholding*: $L = \mathcal{D}_{\mu^{-1}}(A - S - \mu^{-1}Y)$
        2. *Shrinnkage*: $S = \mathcal{S}_{\lambda\mu^{-1}}(A - L + \mu^{-1}Y)$
        3. *Updating*: $Y = Y + \mu(A - L - S)$

5.5.1 Example - Random matrices

```
In [1]: # Shrinkage
        function Shr{T}(x::Array{T},τ::T)
            sign(x).*max(abs(x)-τ,0)
        end

Out[1]: Shr (generic function with 1 method)

In [2]: A=rand(3,5)

Out[2]: 3x5 Array{Float64,2}:
         0.283973  0.518404   0.497746  0.526763  0.941011
         0.982788  0.0457723  0.354028  0.730977  0.206564
         0.706776  0.105853   0.120872  0.620193  0.360419

In [3]: Shr(A,0.5)

Out[3]: 3x5 Array{Float64,2}:
         0.0       0.0184042  0.0  0.0267633  0.441011
         0.482788  0.0        0.0  0.230977   0.0
         0.206776  0.0        0.0  0.120193   0.0

In [4]: # Singular value thresholding
        function D{T}(A::Array{T}, τ::T)
            # U,σ,V=svd(A)
            # This can be replaced by a faster approach
            U,R=qr(A)
            u,σ,V=svd(R)
            # U*=u
            S=Shr(σ,τ)
```

```
            k=sum(S.>zero(T))
            # U[:,1:k]*diagm(S[1:k])*V[:,1:k]'
            U*(u[:,1:k]*diagm(S[1:k]))*V[:,1:k]'
        end
```

```
Out[4]: D (generic function with 1 method)
```

```
In [5]: D(A,0.5)
```

```
Out[5]: 3x5 Array{Float64,2}:
         0.340016  0.288661   0.308755  0.430314  0.559536
         0.637959  0.0980343  0.227491  0.531491  0.278028
         0.483142  0.11077    0.201255  0.42523   0.274113
```

```
In [6]: function PCPAD{T}(A::Array{T})
            # Initialize
            δ=1.0e-7
            tol=δ*vecnorm(A)
            m,n=size(A)
            S=zeros(A)
            Y=zeros(A)
            L=zeros(A)
            μ=(m*n)/(4*(norm(A[:],1)))
            @show μ
            μ1=one(T)/μ
            λ=1/sqrt(max(m,n))
            λμ1=λ*μ1
            ν=1e20
            maxiter=1000
            iterations=0
            # Iterate
            while (ν>tol) && iterations<maxiter
                iterations+=1
                L=D(A-S+μ1*Y,μ1)
                S=Shr(A-L+μ1*Y,λμ1)
                T=A-L-S
                Y=Y+μ*T
                ν=vecnorm(T)
            end
            L,S, iterations
        end
```

```
Out[6]: PCPAD (generic function with 1 method)
```

```
In [7]: L,S,iter=PCPAD(A)
```

```
μ = 0.5355504703390744
```

```
Out[7]: (
        3x5 Array{Float64,2}:
```

```
 0.283973  0.101354   0.115734   0.526763  0.329952
 0.177779  0.0634518  0.0724547  0.329777  0.206564
 0.29658   0.105853   0.120872   0.550149  0.3446   ,

3x5 Array{Float64,2}:
 -0.0        0.417051    0.382012  0.0        0.611059
  0.805009  -0.0176794   0.281573  0.4012    -0.0
  0.410196   0.0        -0.0       0.070044   0.0158189,

72)
```

```
In [8]: rank(L),norm(A-L-S)
```

```
Out[8]: (1,1.3839379003899308e-7)
```

```
In [9]: m=100
        n=100
        k=10
        sparsity=0.1
        L=rand(m,k)*rand(k,n)
        rank(L)
```

```
Out[9]: 10
```

```
In [10]: S=10*sprand(m,n,sparsity)
         A=L+S
         nnz(S), rank(A)
```

```
Out[10]: (1004,100)
```

```
In [11]: @time L1,S1,iters=PCPAD(A);
```

```
μ = 0.08576041604444747
  3.369214 seconds (31.17 k allocations: 499.971 MB, 2.31% gc time)
```

```
In [12]: iters, rank(L1), norm(L), norm(L-L1), norm(full(S)), norm(S-S1)
```

```
Out[12]: (229,10,251.00436950701618,1.9648385667432708e-5,55.68694271722231,8.939258397233
```

Although there might be no convergence, the splitting is still good.

```
In [13]: S1, full(S)
```

```
Out[13]: (
         100x100 Array{Float64,2}:
           0.0       -0.0       5.07868  ...   0.0        -0.0       -0.0
          -0.0       -0.0       0.0           0.0        -0.0        0.821327
          -0.0        9.31322   0.0           0.0         7.91154   -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       -0.0
```

```
 -0.0       -0.0      -0.0          1.2647e-6   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         7.33238   -0.0
  0.690604   7.3475    0.0      ...   0.0        -0.0       -0.0
  0.0       -0.0       3.94681      0.0        -0.0        0.0
 -0.0       -0.0       0.0          0.0         4.77704   -0.0
  ⋮                              ⋱
  0.0        0.0      -0.0          3.66501     0.763537  -0.0
 -0.0       -0.0       0.0          0.0        -0.0        0.0
 -0.0        0.0       0.0      ...  -0.0        -0.0        7.74401
  0.667603   9.73346   0.0          0.0        -0.0       -0.0
 -0.0       -0.0       4.82665      0.0         0.0        9.56541
  0.0       -0.0      -0.0          0.0        -0.0       -0.0
  0.0        0.0       0.0         -0.0        -0.0        3.91801
 -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       2.7258       0.0         0.0       -0.0
 -0.0       -0.0       0.0         -0.0         0.0       -0.0
 -0.0        0.0       6.0805       0.0        -0.0       -0.0     ,

100x100 Array{Float64,2}:
 0.0       0.0      5.07868  0.0       ...  0.0      0.0       0.0
 0.0       0.0      0.0      0.0            0.0      0.0       0.821326
 0.0       9.31322  0.0      0.0            0.0      7.91154   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      0.0       ...  0.0      0.0       0.0
 0.0       0.0      0.0      0.702433       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       0.0      0.0      0.0            0.0      7.33238   0.0
 0.690604  7.3475   0.0      0.0       ...  0.0      0.0       0.0
 0.0       0.0      3.94681  6.16642        0.0      0.0       0.0
 0.0       0.0      0.0      0.0            0.0      4.77704   0.0
 ⋮                                       ⋱
 0.0       0.0      0.0      0.0            3.66501  0.763536  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       7.74401
 0.667603  9.73346  0.0      0.0            0.0      0.0       0.0
 0.0       0.0      4.82665  0.0            0.0      0.0       9.56541
 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       3.91801
 0.0       0.0      0.0      0.0       ...  0.0      0.0       0.0
 0.0       0.0      0.0      9.90893        0.0      0.0       0.0
 0.0       0.0      2.7258   0.0            0.0      0.0       0.0
 0.0       0.0      0.0      2.10392        0.0      0.0       0.0
 0.0       0.0      6.0805   0.0            0.0      0.0       0.0     )
```

5.5.2 Example - Face recognition

We will try to recover missing features. The images are chosen from the Yale Face Database.

```
In [14]: using Images, Colors
```

```
In [15]: # First a single image - no help
         img=load("files/17.jpg")
```

Out[15]:

```
In [16]: show(img)
```

```
Gray Images.Image with:
  data: 168x192 Array{ColorTypes.Gray{FixedPointNumbers.UFixed{UInt8,8}},2}
  properties:
    colorspace: Gray
    spatialorder:  x y
```

```
In [17]: # Compute the splitting and show number of iterations
         A=float(img.data)
         L,S,iters=PCPAD(A)
         iters, rank(L), norm(A), norm(A-L-S)
```

```
μ = 1.7641865800268313
```

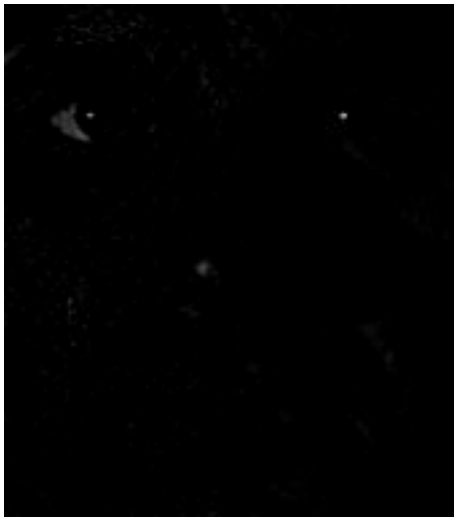
Out[17]: (1000,83,34.627418508422586,0.00014505594109621498)

```
In [18]: display(Image(map(Gray,map(Images.Clamp01NaN(L),L)'))),
         display(Image(map(Gray,map(Images.Clamp01NaN(S),S)')))
```

```
Out[18]: (nothing,nothing)

In [19]: # Another image
         img=load("files/19.jpg")

Out[19]:
```

```
In [20]: A=float(img.data)
         L,S,iters=PCPAD(A)
         @show iters, rank(L), norm(A), norm(A-L-S)
         display(Image(map(Gray,map(Images.Clamp01NaN(L),L)'))),
         display(Image(map(Gray,map(Images.Clamp01NaN(S),S)')))
```

μ = 1.0895784475075525



```
(iters,rank(L),norm(A),norm((A - L) - S)) = (1000,84,51.14350860422345,0.000121074738702399
```

```
Out[20]: (nothing,nothing)
```

5.5.3 Example - Multiple images

Each image of 168×192 pixels is converted to a vector of length 32256. All vectors are stacked in the columns of matrix A, and the low-rank + sparse splitting of A is computed.

```
In [21]: # Load all images in the collection
         dir="./files/yaleB08/"
         files=readdir(dir)
```

```
Out[21]: 64-element Array{ByteString,1}:
          ".directory"
          "yaleB08_P00A+000E+00.pgm"
          "yaleB08_P00A+000E+20.pgm"
          "yaleB08_P00A+000E+45.pgm"
          "yaleB08_P00A+000E-20.pgm"
          "yaleB08_P00A+000E-35.pgm"
          "yaleB08_P00A+005E+10.pgm"
          "yaleB08_P00A+005E-10.pgm"
          "yaleB08_P00A+010E+00.pgm"
          "yaleB08_P00A+010E-20.pgm"
          "yaleB08_P00A+015E+20.pgm"
          "yaleB08_P00A+020E+10.pgm"
          "yaleB08_P00A+020E-10.pgm"
          ⋮
          "yaleB08_P00A-070E+00.pgm"
          "yaleB08_P00A-070E+45.pgm"
          "yaleB08_P00A-070E-35.pgm"
          "yaleB08_P00A-085E+20.pgm"
```

```
 "yaleB08_P00A-085E-20.pgm"
 "yaleB08_P00A-095E+00.pgm"
 "yaleB08_P00A-110E+15.pgm"
 "yaleB08_P00A-110E+40.pgm"
 "yaleB08_P00A-110E+65.pgm"
 "yaleB08_P00A-110E-20.pgm"
 "yaleB08_P00A-120E+00.pgm"
 "yaleB08_P00A-130E+20.pgm"
```

```
In [22]: n=length(files)-1
         img=Array(Any,n)
         B=Array(Any,n)
         for i=1:n
             img[i]=load(joinpath(dir,files[i+1]))
             B[i]=float(img[i].data)
         end
```

```
In [23]: # See the images - last 9 images are meaningless
         # for i=1:n; display(img[i]); end
```

```
In [24]: # Form the big matrix - each image is converted to a column vector
         mi,ni=size(img[1])
         A=Array(Float64,mi*ni,n-9)
         for i=1:n-9
             A[:,i]=vec(B[i])
         end
         size(A)
```

```
Out[24]: (32256,54)
```

```
In [25]: # Now the big SVDs - 3 minutes
         @time L,S,iters=PCPAD(A)
         iters, rank(L), norm(A), norm(A-L-S)
```

```
μ = 1.0288757354994815
172.405835 seconds (82.54 k allocations: 135.759 GB, 31.36% gc time)
```

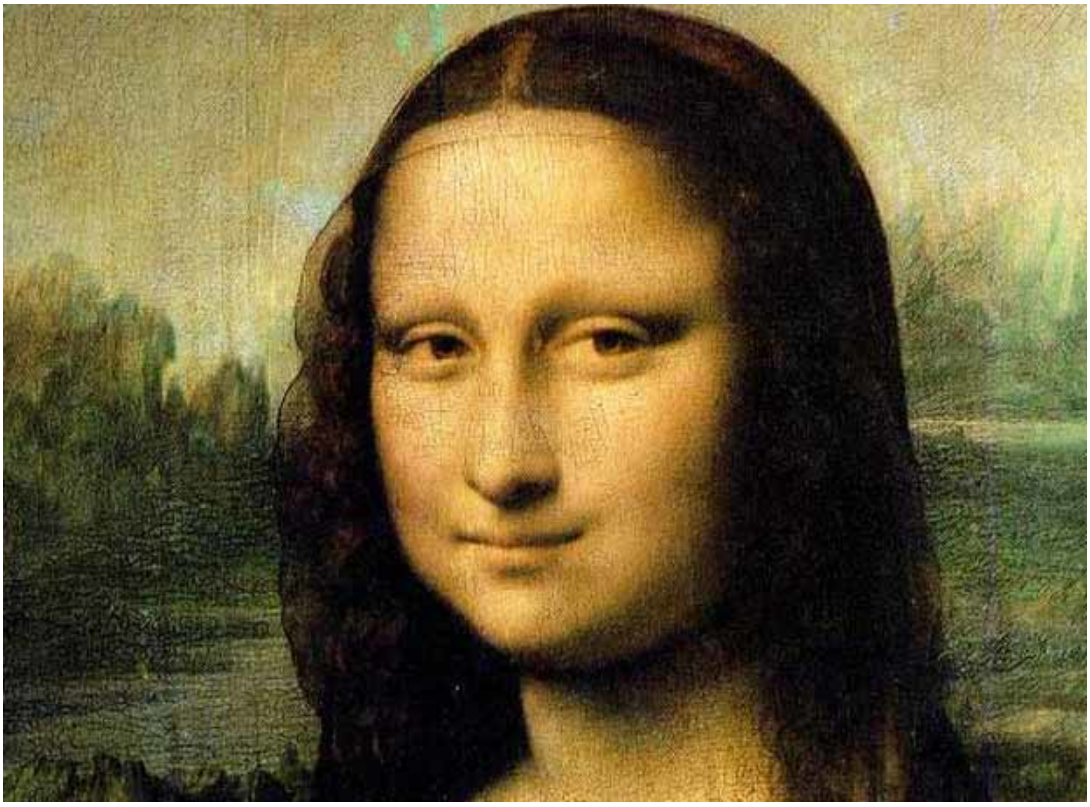
```
Out[25]: (564,27,363.9778534354257,2.0951145993897957e-5)
```

```
In [ ]: for i=1:n-9
            Li=reshape(L[:,i],mi,ni)
            Si=reshape(S[:,i],mi,ni)
            display(vcat( img[i], Image(map(Gray,map(Images.Clamp01NaN(Li),Li))),
            Image(map(Gray,map(Images.Clamp01NaN(Si),Si)))))
        end
```

5.5.4 Example - Mona Lisa's smile

```
In [27]: img=load("files/mona-lisa_1.jpg")
```

```
Out[27]:
```



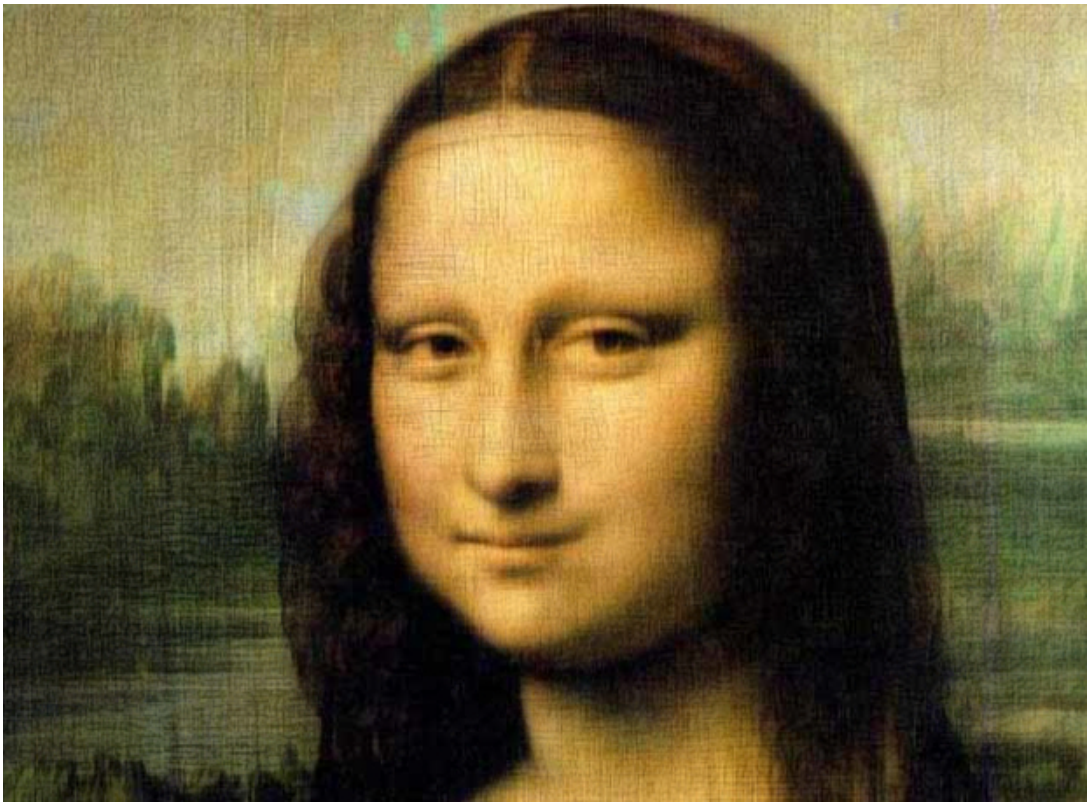
```
In [28]: imgsep=separate(img)
         fieldnames(imgsep)

Out[28]: 2-element Array{Symbol,1}:
          :data
          :properties

In [29]: imgsep.properties

Out[29]: Dict{ASCIIString,Any} with 3 entries:
           "colorspace"   => "RGB"
           "colordim"     => 3
           "spatialorder" => ASCIIString["y","x"]

In [30]: size(imgsep.data)

Out[30]: (456,620,3)

In [31]: # 7 minutes
         @time RL,RS,Riter=PCPAD(float(imgsep.data[:,:,1]))
         GL,GS,Giter=PCPAD(float(imgsep.data[:,:,2]))
         BL,BS,Biter=PCPAD(float(imgsep.data[:,:,3]))
         Giter, rank(GL), norm(float(imgsep.data[:,:,2])),
         norm(float(imgsep.data[:,:,2])-GL-GS)
```

```
μ = 0.5961272467102573
129.708615 seconds (155.51 k allocations: 35.057 GB, 8.44% gc time)
μ = 0.6796352970068623
μ = 1.1278606242265707
```

```
Out[31]: (545,246,221.7432116692531,1.002977978882978e-5)
```

```
In [32]: # Mona Lisa's low-rank and sparse component
         Image(map(RGB,RL,GL,BL))
```

```
Out[32]:
```



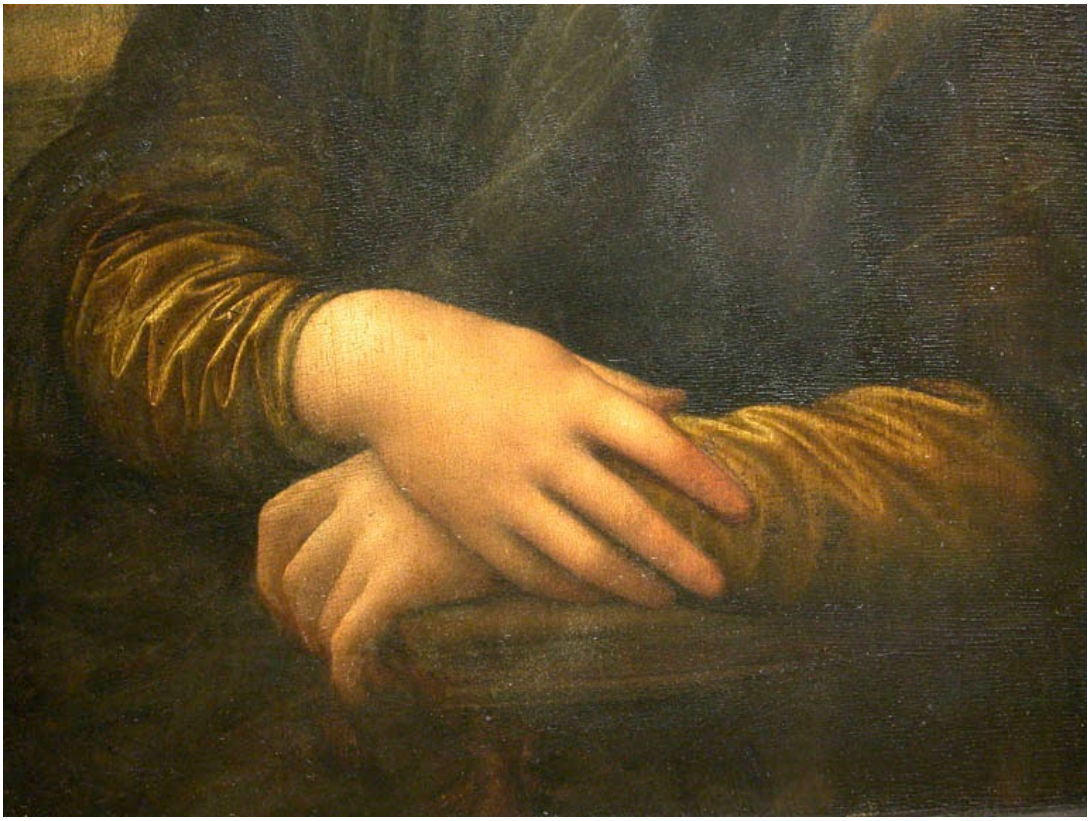
```
In [33]: Image(map(RGB,RS+0.5,GS+0.5,BS+0.5))
```

```
Out[33]:
```

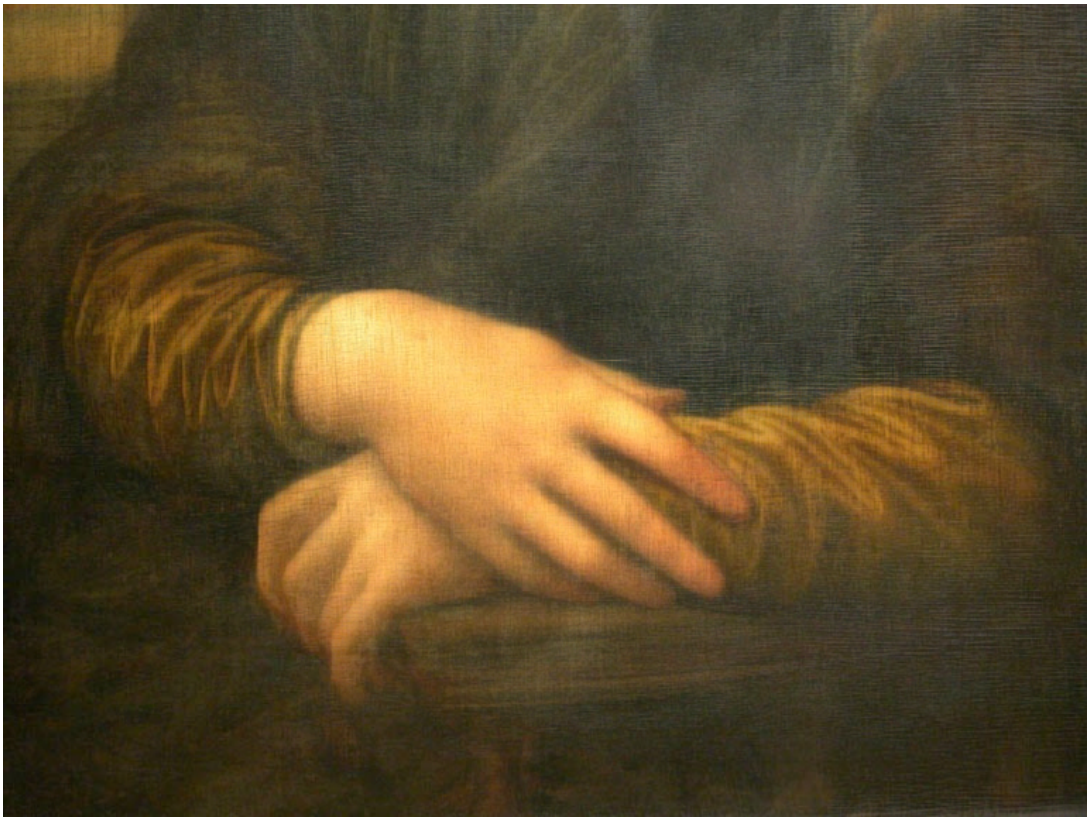
5.5.5 Example - Mona Lisa's hands

```
In [34]: img=load("files/Mona_Lisa_detail_hands.jpg")
```

Out[34]:



```
In [35]: imgsep=separate(img)
         # 7 minutes
         @time RL,RS,Riter=PCPAD(float(imgsep.data[:,:,1]))
         GL,GS,Giter=PCPAD(float(imgsep.data[:,:,2]))
         BL,BS,Biter=PCPAD(float(imgsep.data[:,:,3]))
         Giter, rank(GL), norm(float(imgsep.data[:,:,2])),
         norm(float(imgsep.data[:,:,2])-GL-GS)

μ = 0.71226065731073
184.707345 seconds (87.37 k allocations: 44.723 GB, 8.64% gc time)
μ = 0.8682249892135205
μ = 1.4680512174183444


Out[35]: (397,315,205.57272143820586,9.788398887375607e-6)

In [36]: Image(map(RGB,RL,GL,BL))

Out[36]:
```



In [37]: `Image(map(RGB,RS+0.5,GS+0.5,BS+0.5))`

Out[37]:

In [ ]:

6 Signal Decomposition Using EVD of Hankel Matrices

Suppose we are given a data signal which consists of several nearly mono-components.
Can we recover the mono-components?
The answer is YES, with an efficient algorithm using EVDs of Hankel matrices.
Mono-component recovery can be successfully applied to audio signals.

6.1 Prerequisites

The reader should be familiar to elementary concepts about signals, and with linear algebra concepts, particularly EVD and its properties and algorithms.

6.2 Competences

The reader should be able to decompose given signal into mono-components using EVD methods.

6.3 References

For more details see P. Jain and R. B. Pachori, An iterative approach for decomposition of multi-component non-stationary signals based on eigenvalue decomposition of the Hankel matrix.
Credits: The first Julia implementation was derived in A. M. Bačak, Master's Thesis.

6.4 Extraction of stationary mono-components

6.4.1 Definitions

Let $x \in \mathbb{R}^m$, denote a **signal** with N samples.

Assume x is composed of L **stationary mono-components**, $x = \sum_{k=1}^{L} x^{(k)}$, where $x_i^{(k)} = A_k \cos(2\pi f_k i + \theta_k)$ for $i = 1, 2, \ldots, m$. Here $f_k = \frac{F_k}{F}$ is the **normalized frequency** of $x^{(k)}$, F is the **sampling frequency** of x in Hz, F_k is the sampling frequency of $x^{(k)}$, A_k is the **amplitude** of $x^{(k)}$, and θ_k is the **phase** of $x^{(k)}$. We assume that $F_k < F_{k+1}$ for $k = 1, 2, \ldots, n-1$, and $F > 2F_n$.
A **Hankel matrix** is a (real) square matrix with constant values along the skew-diagonals. More precisely, let $a \in \mathbb{R}^{2n-1}$. An $n \times n$ matrix $H \equiv H(a)$ for which $H_{ij} = A_{i+1,j-1} = a_{i+j-1}$ is a Hankel matrix.

6.4.2 Facts

Let x be a signal with $2n-1$ samples composed of L stationary mono-components.
Let H be an $n \times n$ Hankel matrix corresponding to x and let $H = U\Lambda U^T$ be its EVD (Hankel matrix is symmetric) with $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.
Smilarly, let H_k be the $n \times n$ Hankel matrix corresponding to the k-th component $x^{(k)}$ and let $H_k = U_k \Lambda_k U_k^T$ be its EVD.

1. $H = \sum_{k=1}^{L} H_k$.

2. $H_k = \lambda_k U_{:,k} U_{:,k}^T + \lambda_{n-k+1} U_{:,n-k+1} U_{:,n-k+1}^T$.

6.4.3 Example - Signal with three mono-components

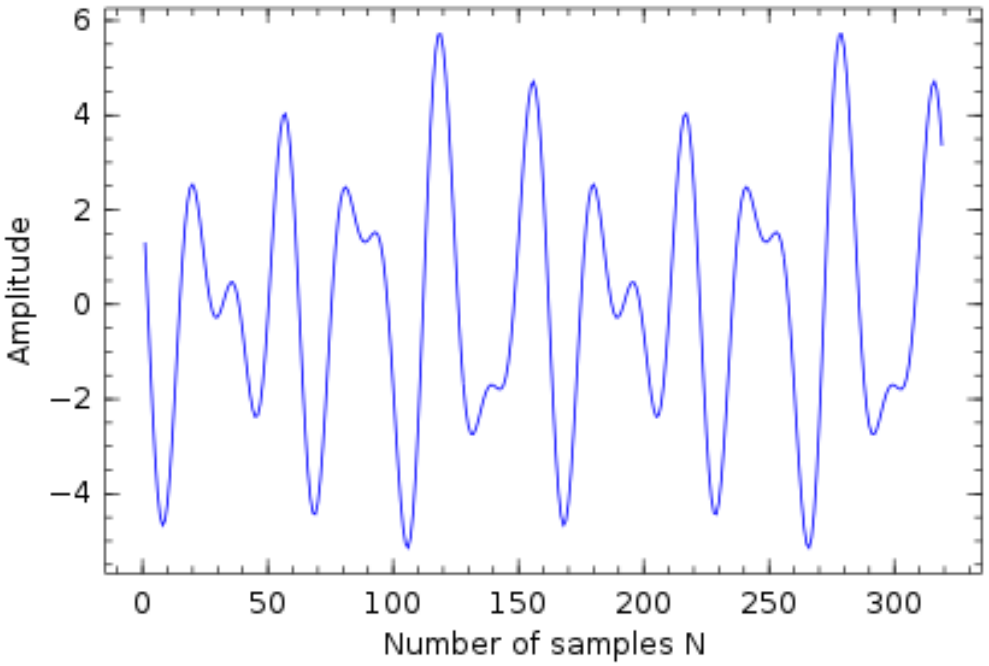
In [2]:
```
using Winston
using SpecialMatrices
```

In [3]:
```
# Small Hankel matrix
a=collect(1:11)
Hankel(a)
```

Out[3]:
```
6x6 SpecialMatrices.Hankel{Int64}:
 1  2  3  4   5   6
 2  3  4  5   6   7
 3  4  5  6   7   8
 4  5  6  7   8   9
 5  6  7  8   9  10
 6  7  8  9  10  11
```

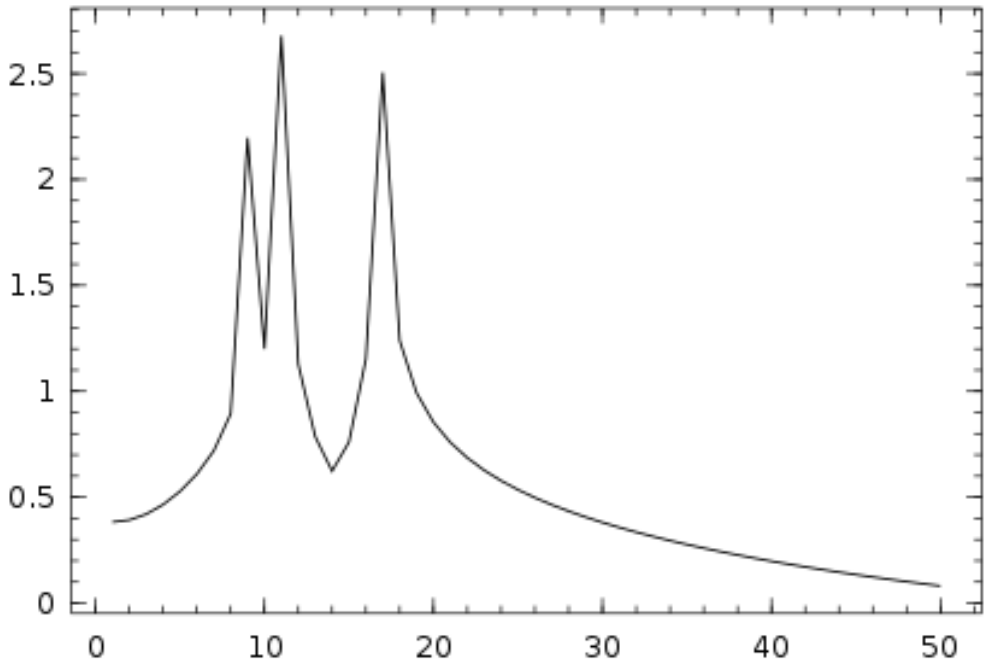
In [4]:
```
# Create the signal
n=160
N=2*n-1
F = 6400
L = 3
A = [3, 2, 1]
Fk= [200, 320, 160]
θ = [pi/2, pi/4, 0]
x = zeros(N)
for k=1:L
    for i=1:N
        x[i]+=A[k]*cos(2*pi*Fk[k]*i/F+θ[k])
    end
end
plot(x,"b",xlabel="Number of samples N", ylabel="Amplitude")
```

Out[4]:



In [5]:
```
# FFT indicates that there are three components
y=fft(x)
plot(log10(abs(y[1:50])))
```

Out[5]:



```
In [6]: # Let us decompose the signal
        H=Hankel(x)
```

```
Out[6]: 160x160 SpecialMatrices.Hankel{Float64}:
          1.3104     0.115875  -1.08857   ...   4.07448    3.35497    2.41421
          0.115875  -1.08857   -2.22028       3.35497    2.41421    1.3104
         -1.08857   -2.22028   -3.20152       2.41421    1.3104     0.115875
         -2.22028   -3.20152   -3.96587       1.3104     0.115875  -1.08857
         -3.20152   -3.96587   -4.46374       0.115875  -1.08857   -2.22028
         -3.96587   -4.46374   -4.66636   ...  -1.08857   -2.22028   -3.20152
         -4.46374   -4.66636   -4.56793      -2.22028   -3.20152   -3.96587
         -4.66636   -4.56793   -4.18585      -3.20152   -3.96587   -4.46374
         -4.56793   -4.18585   -3.55882      -3.96587   -4.46374   -4.66636
         -4.18585   -3.55882   -2.74321      -4.46374   -4.66636   -4.56793
         -3.55882   -2.74321   -1.80783   ...  -4.66636   -4.56793   -4.18585
         -2.74321   -1.80783   -0.827855     -4.56793   -4.18585   -3.55882
         -1.80783   -0.827855   0.121836     -4.18585   -3.55882   -2.74321
          ⋮                                  ⋱
          0.555961   1.35743    2.19081      -1.22175   -0.762656  -0.163073
          1.35743    2.19081    2.99615      -0.762656  -0.163073   0.555961
          2.19081    2.99615    3.70922   ...  -0.163073   0.555961   1.35743
          2.99615    3.70922    4.2674        0.555961   1.35743    2.19081
          3.70922    4.2674     4.61573       1.35743    2.19081    2.99615
          4.2674     4.61573    4.71235       2.19081    2.99615    3.70922
          4.61573    4.71235    4.53309       2.99615    3.70922    4.2674
          4.71235    4.53309    4.07448   ...   3.70922    4.2674     4.61573
          4.53309    4.07448    3.35497       4.2674     4.61573    4.71235
          4.07448    3.35497    2.41421       4.61573    4.71235    4.53309
          3.35497    2.41421    1.3104        4.71235    4.53309    4.07448
          2.41421    1.3104     0.115875      4.53309    4.07448    3.35497
```

```
In [7]: λ,U=eig(full(H))
        λ
```

```
Out[7]: 160-element Array{Float64,1}:
         -240.0
         -160.0
          -80.0
           -6.16457e-13
           -4.39926e-14
           -4.29379e-14
           -3.20429e-14
           -2.73661e-14
           -2.47427e-14
           -2.45914e-14
           -2.36681e-14
           -2.22072e-14
           -2.16047e-14
            ⋮
            2.00395e-14
            2.18714e-14
```

```
  2.29849e-14
  2.32129e-14
  2.50189e-14
  2.80344e-14
  3.08087e-14
  3.0899e-14
  5.36341e-14
 80.0
160.0
240.0
```

We see that the three smallest and the three largest eigenvalues come in pairs and define the three mono-components.
The ratios of the moduli of the eigenvalues correspond to the ratios of the amplitudes of the mono-components.

```
In [8]: # Form the three matrices
        Hcomp=Array(Any,3)
        for k=1:L
            Hcomp[k]=λ[k]*U[:,k]*U[:,k]' + λ[end-k+1]*U[:,end-k+1]*U[:,end-k+1]'
        end
```

```
In [9]: # Compare the first matrix with the Hankel matrix of the first mono-component
        x1 = zeros(N)
        l=1
        for i=1:N
            x1[i]+=A[l]*cos(2*pi*Fk[l]*i/F+θ[l])
        end
```

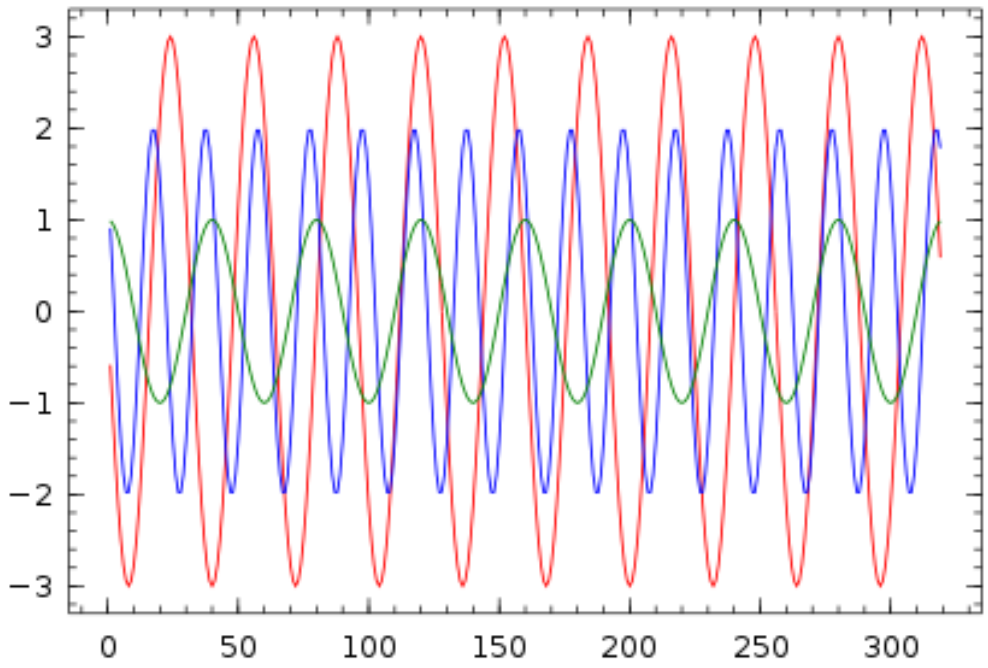
```
In [10]: H1=Hankel(x1)
         eigvals(full(H1)), norm(Hcomp[1]-H1)
```

```
Out[10]: ([-240.0,-1.85479e-13,-1.77059e-13,-1.69081e-13,-1.67474e-13,-1.59156e-13,-1.41293e
```

```
In [11]: # Now we reconstruct the mono-components from the skew-diagonal elements of Hcomp
         xcomp=Array(Array{Float64},L)
         z=Array(Float64,N)
         for k=1:L
             z[1:2:N]=diag(Hcomp[k])
             z[2:2:N]=diag(Hcomp[k],1)
             xcomp[k]=copy(z)
         end
```

```
In [12]: xaxis=collect(1:N)
         plot(xaxis,xcomp[1],"r", xaxis,xcomp[2],"b", xaxis,xcomp[3],"g")
```

```
Out[12]:
```



6.5 Fast EVD of Hankel matrices

Several outer eigenvalues pairs of Hankel matrices can be computed using Lanczos method. If the multiplication Hx is performed using Fast Fourier Transform, this EVD computation is very fast.

6.5.1 Definitions

A **Toeplitz matrix** is a (real) square matrix with constant values along the diagonals. More precisely, let $a = (a_{-(n-1)}, a_{-(n-2)}, \ldots, a_{-1}, a_0, a_1, \ldots, a_{n-1}) \in \mathbb{R}^{2n-1}$. An $n \times n$ matrix $T \equiv T(a)$ for which $T_{ij} = T_{i+1,j+1} = a_{i-j}$ is a Toeplitz matrix.

A **circulant matrix** is a Toeplitz matrix where each column is rotated one element downwards relative to preceeding column. More precisely, let $a \in \mathbb{R}^n$. An $n \times n$ matrix $C \equiv C(a) = T(a, a_{1:n-1})$ is a Circulant matrix.

A **rotation matrix** is an identity matrix rotated 90 degrees to the right (or left).

A **Fourier matrix** is Vandermonde matrix $F_n = V(1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1})$, where $\omega_n = exp(2\pi i/n)$ is the n-th root of unity (see the Eigenvalue Decomposition - Definitions and Facts notebook).

6.5.2 Example

Notice different meaning of vector a: in `C=Circulant(a)`, a is the first column, in `T=Toeplitz(a)`, a_i is the diagonal element of the i-th diagonal starting from T_{1n}, and in `H=Hankel(a)`, a_i is the element of the i-th skew-diagonal starting from H_{11}.

```
In [13]: C=Circulant([1,2,3,4,5])
         TC=Toeplitz([2,3,4,5,1,2,3,4,5])
```

```
T=Toeplitz([1,2,3,4,5,6,7,8,9])
H1=Hankel([1,2,3,4,5,6,7,8,9])
J=rotl90(eye(Int64,5))
C,TC,T,H1,J
```

```
Out[13]: (
5x5 SpecialMatrices.Circulant{Int64}:
 1  5  4  3  2
 2  1  5  4  3
 3  2  1  5  4
 4  3  2  1  5
 5  4  3  2  1,

5x5 SpecialMatrices.Toeplitz{Int64}:
 1  5  4  3  2
 2  1  5  4  3
 3  2  1  5  4
 4  3  2  1  5
 5  4  3  2  1,

5x5 SpecialMatrices.Toeplitz{Int64}:
 5  4  3  2  1
 6  5  4  3  2
 7  6  5  4  3
 8  7  6  5  4
 9  8  7  6  5,

5x5 SpecialMatrices.Hankel{Int64}:
 1  2  3  4  5
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
 5  6  7  8  9,

5x5 Array{Int64,2}:
 0  0  0  0  1
 0  0  0  1  0
 0  0  1  0  0
 0  1  0  0  0
 1  0  0  0  0)
```

6.5.3 Facts

For more details see G. H. Golub and C. F. Van Loan, Matrix Computations, p. 202, and the references therein

1. Hankel matrix is the product of a Toeplitz matrix and the rotation matrix.
2. Circulant matrix is normal and, thus, unitarily diagonalizable, with the eigenvalue decomposition

$$C(a) = U \operatorname{diag}(F_n^* a) U^*,$$

where $U = \frac{1}{\sqrt{n}} F_n$. The product $F_n^* a$ can be computed by the *Fast Fourier Transform*(FFT).

3. Given $a, x \in \mathbb{R}^n$, the product $y = C(a)x$ can be computed using FFT as follows:

$$\begin{aligned}
\tilde{x} &= F_n^* x \\
\tilde{a} &= F_n^* a \\
\tilde{y} &= \tilde{x}. * \tilde{a} \\
y &= F_n^{-*} \tilde{y}.
\end{aligned}$$

4. Toeplitz matrix of order n can be embedded in a circulant matrix of order $2n-1$: if $a \in \mathbb{R}^{2n-1}$, then
$$T(a) = [C([a_{n+1:2n-1}; a_{1:n}])]_{1:n,1:n}.$$

5. Further, let $x \in \mathbb{R}^n$ and let $\bar{x} \in \mathbb{R}^{2n-1}$ be equal to x padded with $n-1$ zeros.Then
$$T(a)x = [C([a_{n+1:2n-1}; a_{1:n}])\bar{x}]_{1:n}.$$

6. Fact 1 implies $H(a)x = (T(a)J)x = T(a)(Jx)$.

6.5.4 Examples

```
In [14]: # Fact 1
         full(T)*J
```

```
Out[14]: 5x5 Array{Int64,2}:
          1  2  3  4  5
          2  3  4  5  6
          3  4  5  6  7
          4  5  6  7  8
          5  6  7  8  9
```

```
In [15]: # Fact 2
         a=rand(-8:8,6)
         n=length(a)
         C=Circulant(a)
         ω=exp(2*pi*im/n)
         v=map(Complex,[ω^k for k=0:n-1])
         F=Vandermonde(v)
         U=F/sqrt(n)
         λ=full(F)'*a
```

```
Out[15]: 6-element Array{Complex{Float64},1}:
                    9.0+0.0im
                   -8.0-12.1244im
                    3.0-6.9282im
           7.0+1.74305e-14im
                    3.0+6.9282im
                   -8.0+12.1244im
```

```
In [16]: # Residual
         norm(full(C)*U-U*diagm(λ))
```

```
Out[16]: 4.6092457625395435e-14
```

```
In [17]: ?fft
```

```
search: fft fft! FFTW fftshift rfft ifft bfft ifft! bfft! ifftshift irfft brfft
```

Out[17]:

```
fft(A [, dims])
```

Performs a multidimensional FFT of the array `A`. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_fft()` for even greater efficiency. A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

$$\mathrm{DFT}(A)[k] = \sum_{n=1}^{\mathrm{length}(A)} \exp\left(-i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n].$$

A multidimensional FFT simply performs this operation along each transformed dimension of `A`. Higher performance is usually possible with multi-threading. Use `FFTW.set_num_threads(np)` to use `np` threads, if you have `np` processors.

```
In [18]: # Check fft
         norm(λ-fft(a))
```

```
Out[18]: 4.9661729113191226e-14
```

```
In [19]: # Fact 3 - Circulant() x vector
         import Base.*
         function *{T}(C::Circulant{T},x::Vector{T})
             xt=fft(x)
             vt=fft(C.c)
             yt=vt.*xt
             real(ifft(yt))
         end
```

```
Out[19]: * (generic function with 158 methods)
```

```
In [20]: x=rand(-9:9,n)
```

```
Out[20]: 6-element Array{Int64,1}:
          -4
           5
          -5
          -8
          -7
          -9
```

```
In [21]: [full(C)*x C*x]
```

```
Out[21]: 6x2 Array{Float64,2}:
          -152.0  -152.0
           -65.0   -65.0
             1.0     1.0
            12.0    12.0
            11.0    11.0
           -59.0   -59.0
```

```
In [22]: # Fact 4 - Embedding Toeplitz() into Circulant()
         n=5
         a=rand(-6:6,2*n-1)
         T=Toeplitz(a)
```

```
Out[22]: 5x5 SpecialMatrices.Toeplitz{Int64}:
           5   2   4   0  0
          -4   5   2   4  0
           2  -4   5   2  4
          -4   2  -4   5  2
           4  -4   2  -4  5
```

```
In [23]: C=Circulant([a[n:2*n-1];a[1:n-1]])
```

```
Out[23]: 9x9 SpecialMatrices.Circulant{Int64}:
           5   2   4   0   0   4  -4   2  -4
          -4   5   2   4   0   0   4  -4   2
           2  -4   5   2   4   0   0   4  -4
          -4   2  -4   5   2   4   0   0   4
           4  -4   2  -4   5   2   4   0   0
           0   4  -4   2  -4   5   2   4   0
           0   0   4  -4   2  -4   5   2   4
           4   0   0   4  -4   2  -4   5   2
           2   4   0   0   4  -4   2  -4   5
```

```
In [24]: # Fact 5 - Toeplitz() x vector
         function *{T}(A::Toeplitz{T},x::Vector{T})
             n=length(A.c)
             k=Int(round((n+1)/2))
             C=Circulant([A.c[k:n];A.c[1:k-1]])
             (C*[x;zeros(T,k-1)])[1:k]
         end
```

```
Out[24]: * (generic function with 159 methods)
```

```
In [25]: x=rand(-6:6,n)
```

```
Out[25]: 5-element Array{Int64,1}:
           4
          -5
           0
           3
           0
```

```
In [26]: [full(T)*x T*x]

Out[26]: 5x2 Array{Float64,2}:
           10.0   10.0
          -29.0  -29.0
           34.0   34.0
          -11.0  -11.0
           24.0   24.0

In [27]: # Fact 6 - Hankel() x vector
         function *{T}(A::Hankel{T},x::Vector{T})
             Toeplitz(A.c)*reverse(x)
         end

Out[27]: * (generic function with 160 methods)

In [28]: H1=Hankel(a)

Out[28]: 5x5 SpecialMatrices.Hankel{Int64}:
          0   0   4   2   5
          0   4   2   5  -4
          4   2   5  -4   2
          2   5  -4   2  -4
          5  -4   2  -4   4

In [29]: [full(H1)*x H1*x]

Out[29]: 5x2 Array{Float64,2}:
            6.0    6.0
           -5.0   -5.0
           -6.0   -6.0
          -11.0  -11.0
           28.0   28.0
```

6.5.5 Example - Fast EVD of a Hankel matrix

Given a Hankel matrix H, the Lanczos method can be applied by defining a function (linear map) which returns the product Hx for any vector x. This approach uses the package LinearMaps.jl and is described in the Symmetric Eigenvalue Decomposition - Lanczos Method notebook.

The computation is very fast and allocates little extra space.

IMPORTANT For package SpecialMatrices.jl to work with very large Hankel matrices, we need to modify the corresponding lines in the file `hankel.jl` to

```
getindex(H::Hankel, i, j) = H.c[i+j-1]
isassigned(H::Hankel, i, j) = isassigned(H.c, i+j-1)
```

```
In [30]: using LinearMaps

In [31]: n=size(H,1)
         f(x)=H*x

Out[31]: f (generic function with 1 method)
```

```
In [32]: A=LinearMap(f,n,issym=true)

Out[32]: FunctionMap{Float64}(f,160,160;ismutating=false,issym=true,ishermitian=true,isposde

In [33]: size(A)

Out[33]: (160,160)

In [35]: # Run twice
         @time λA,UA=eigs(A, nev=6, which=:LM)

  0.004551 seconds (4.14 k allocations: 1.131 MB)


Out[35]: ([240.0,-239.99999999999994,-160.00000000000009,160.00000000000006,-80.000000000000
         160x6 Array{Float64,2}:
          -0.0709273  -0.0864252  -0.0584171   -0.0953281    0.00877199  -0.111459
          -0.0527038  -0.0986018  -0.085016    -0.0726105    0.0261      -0.108714
          -0.0324548  -0.106989   -0.103293    -0.0427853    0.0427853   -0.103293
          -0.0109586  -0.111265   -0.111459    -0.00877199   0.0584171   -0.0953281
           0.0109586  -0.111265   -0.108714     0.0261       0.0726105   -0.085016
           0.0324548  -0.106989   -0.0953281    0.0584171    0.085016    -0.0726105
           0.0527038  -0.0986018  -0.0726105    0.085016     0.0953281   -0.0584171
           0.0709273  -0.0864252  -0.0427853    0.103293     0.103293    -0.0427853
           0.0864252  -0.0709273  -0.00877199   0.111459     0.108714    -0.0261
           0.0986018  -0.0527038   0.0261       0.108714     0.111459    -0.00877199
           0.106989   -0.0324548   0.0584171    0.0953281    0.111459     0.00877199
           0.111265   -0.0109586   0.085016     0.0726105    0.108714     0.0261
           0.111265    0.0109586   0.103293     0.0427853    0.103293     0.0427853
           ⋮                                                             ⋮
          -0.0109586   0.111265   -0.00877199   0.111459    -0.108714     0.0261
          -0.0324548   0.106989    0.0261       0.108714    -0.111459     0.00877199
          -0.0527038   0.0986018   0.0584171    0.0953281   -0.111459    -0.00877199
          -0.0709273   0.0864252   0.085016     0.0726105   -0.108714    -0.0261
          -0.0864252   0.0709273   0.103293     0.0427853   -0.103293    -0.0427853
          -0.0986018   0.0527038   0.111459     0.00877199  -0.0953281   -0.0584171
          -0.106989    0.0324548   0.108714    -0.0261      -0.085016    -0.0726105
          -0.111265    0.0109586   0.0953281   -0.0584171   -0.0726105   -0.085016
          -0.111265   -0.0109586   0.0726105   -0.085016    -0.0584171   -0.0953281
          -0.106989   -0.0324548   0.0427853   -0.103293    -0.0427853   -0.103293
          -0.0986018  -0.0527038   0.00877199  -0.111459    -0.0261      -0.108714
          -0.0864252  -0.0709273  -0.0261      -0.108714    -0.00877199  -0.111459   ,

         6,1,20,[4.06863e-15,1.89385e-15,-4.32206e-15,6.27461e-15,9.95217e-15,1.74705e-15,-3
```

6.6 Extraction of non-stationary mono-components

6.6.1 Definitions

Let $x \in \mathbb{R}^m$, denote a **signal** with N samples.

Assume x is composed of L **non-stationary mono-components**, $x = \sum_{k=1}^{L} x^{(k)}$, where $x_i^{(k)} = A_k \cos(2\pi f_k i + \theta_k)$ for $i = 1, 2, \ldots, m$. Assume that the normalized frequencies $f_k = \frac{F_k}{F}$, the sampling frequencies F_k, the amplitudes A_k, and the phases θ_k, all *sightly* change in time.
Let $H \equiv H(x)$ be the Hankel matrix of x. The eigenpair of (λ, u) of H is **significant** if $|\lambda| > \tau \cdot \sigma(H)$. Here $\sigma(H)$ is the spectral radius of H, and τ is the **significant threshold percentage** chosen by the user depending on the type of the problem.

6.6.2 Fact

The following algorithm decomposes the signal x:

1. Choose τ and form the Hankel matrix H
2. Compute the EVD of H
3. Choose the significant eigenpairs of H
4. For each significant eigenpair (λ, u)
   (a) Form the rank one matrix $M = \lambda u u^T$
   (b) Define a new signal y consisting of averages of the skew-diagonals of M
   (c) Form the Hankel matrix $H(y)$
   (d) Compute the EVD of $H(y)$
   (e) Choose the significant eigenpairs of $H(y)$
   (f) **If** $H(y)$ has only two significant eigenpairs, declare y a mono-component, **else** go to step 4.

6.6.3 Example - Note A

Each tone has its fundamental frequency (mono-component). However, musical instruments produce different overtones (harmonics) which are near integer multiples of the fundamental frequency. Due to construction of resonant boxes, these frequencies slightly vary in time, and their amplitudes are contained in a time varying envelope.
Tones produces by musical instruments are nice examples of non-stationary signals. We shall decompose the note A4 played on piano.
For manipulation of recordings, we are using package WAV.jl. Another package with similar functionality is the package AudioIO.jl.

```
In [36]: # Pkg.checkout("WAV")
         using WAV
```

```
In [37]: whos(WAV)
```

```
                            WAV     85 KB     Module
                       WAVArray    220 bytes  DataType
            WAVE_FORMAT_ALAW       2 bytes   UInt16
      WAVE_FORMAT_IEEE_FLOAT       2 bytes   UInt16
           WAVE_FORMAT_MULAW       2 bytes   UInt16
             WAVE_FORMAT_PCM       2 bytes   UInt16
                  WAVFormat    184 bytes  DataType
```

```
WAVFormatExtension    136 bytes  DataType
   bits_per_sample    572 bytes  Function
      isextensible    541 bytes  Function
          isformat   1722 bytes  Function
         wavappend   2541 bytes  Function
           wavplay   2782 bytes  Function
           wavread   4687 bytes  Function
          wavwrite   8560 bytes  Function
```

```
In [38]: ?wavread

search: wavread WAVE_FORMAT_IEEE_FLOAT wavwrite WAVFormatExtension
```

Out[38]:

No documentation found. `WAV.wavread` is a generic `Function`.

```
# 5 methods for generic function "wavread":
wavread(io::IO) at /home/slap/.julia/v0.4/WAV/src/WAV.jl:583
wavread(filename::AbstractString) at /home/slap/.julia/v0.4/WAV/src/WAV.jl:625
wavread(filename::AbstractString, fmt::AbstractString) at /home/slap/.julia/v0.4/WAV/src/WA
wavread(filename::AbstractString, n) at /home/slap/.julia/v0.4/WAV/src/WAV.jl:632
wavread(filename::AbstractString, n, fmt) at /home/slap/.julia/v0.4/WAV/src/WAV.jl:633
```

```
In [39]: # Load a signal
         s, Fs = wavread("files/piano_A41.wav")
```

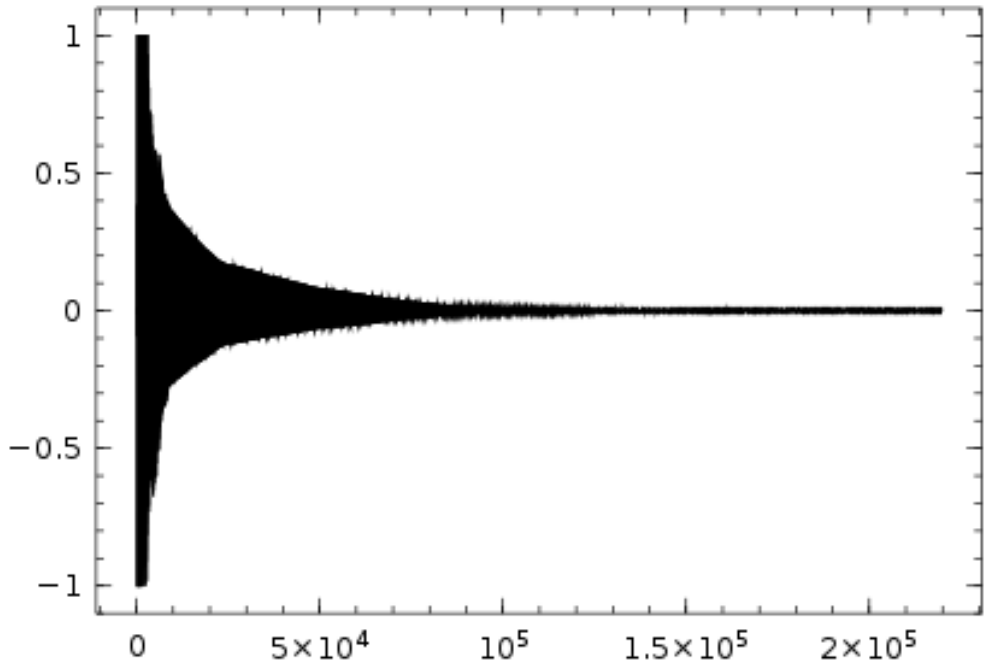
```
Out[39]: (
         219712x1 Array{Float64,2}:
          -0.0101321
          -0.0102542
          -0.0102542
          -0.0101321
          -0.00994903
          -0.00964385
          -0.00933866
          -0.00909452
          -0.00878933
          -0.00860622
          -0.00842311
          -0.00799585
          -0.00720237
           ⋮
           0.0
           0.0
           0.0
           0.0
          -6.1037e-5
          -6.1037e-5
```

```
 -6.1037e-5
  0.0
  0.0
  0.0
  0.0
  0.0       ,

44100.0f0,0x0010,Dict{Symbol,Any}(:fmt=>WAV.WAVFormat(0x0001,0x0001,0x0000ac44,0x00
```

In [41]:
```
# Play the signal
wavplay(s,Fs)
```

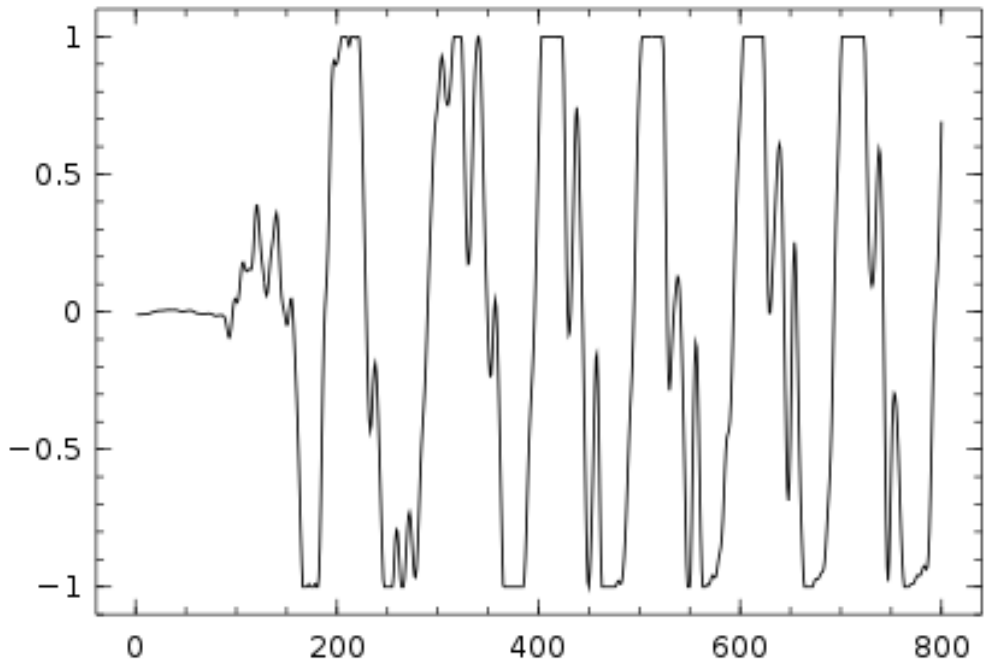
In [42]:
```
# Plot the signal
plot(s)
```

Out[42]:



In [43]:
```
plot(s[1:800])
```

Out[43]:



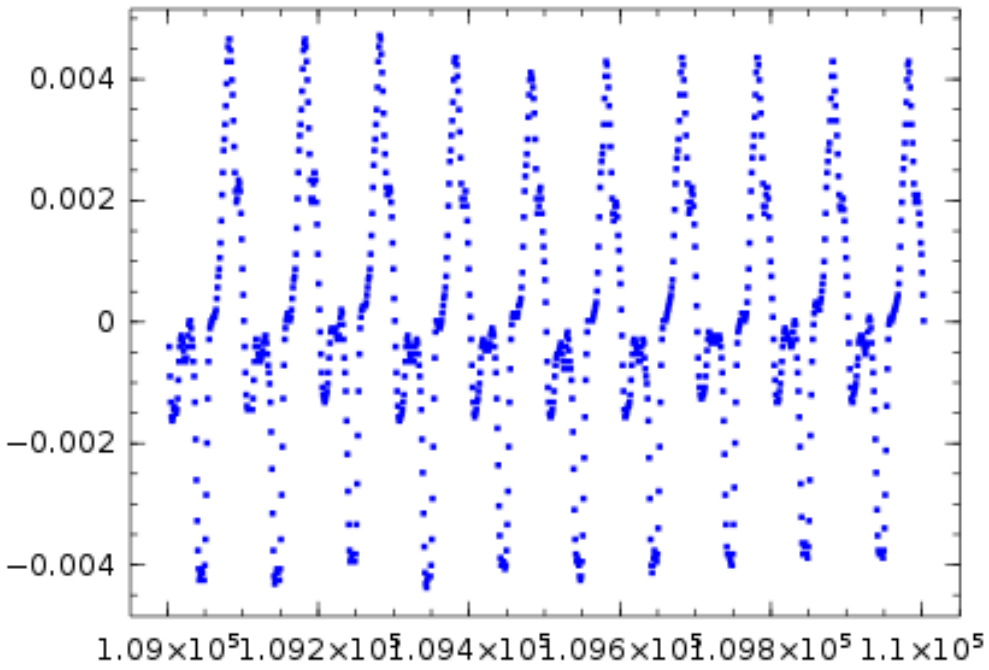
Let us visualize the signal in detail using the approach from the Julia is Fast notebook.

```
In [44]: using Interact
```

```
In [45]: @manipulate for k=1:1000:size(s,1)
             plot(collect(k:k+1000),s[k:k+1000],"b.")
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(109001, nactions=0),"k",109001,1:1000:219001,true)
```

```
Out[45]:
```




```
In [46]: # Last part of the signal is just noise, so we read a shorter signal. N must be od
         s, Fs = wavread("files/piano_A41.wav",100001)
```

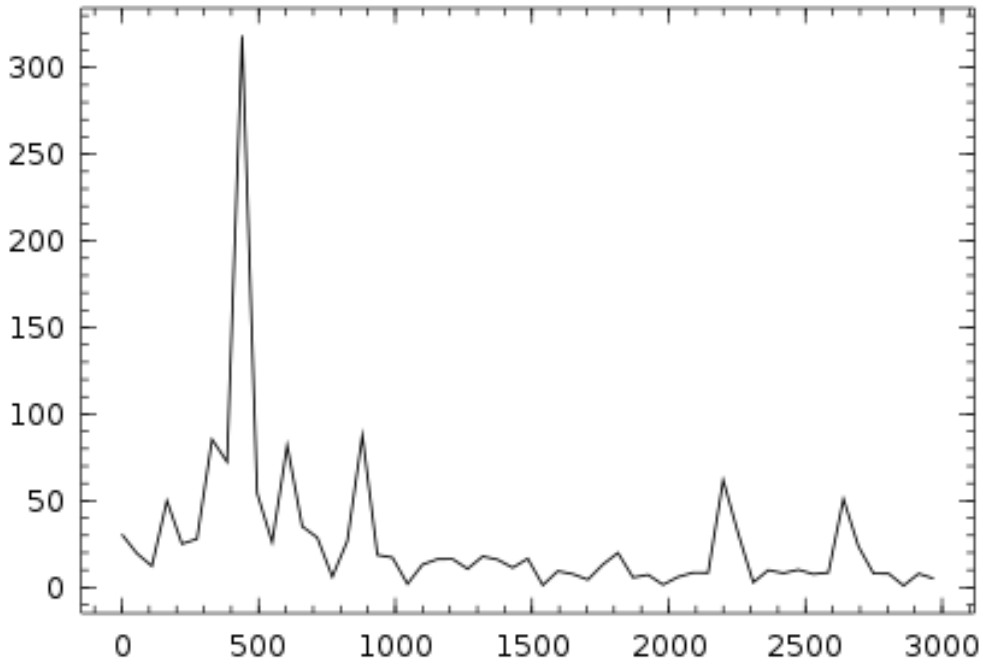
```
Out[46]: (
         100001x1 Array{Float64,2}:
          -0.0101321
          -0.0102542
          -0.0102542
          -0.0101321
          -0.00994903
          -0.00964385
          -0.00933866
          -0.00909452
          -0.00878933
          -0.00860622
          -0.00842311
          -0.00799585
          -0.00720237
           ⋮
           0.00335704
           0.00317392
           0.00286874
           0.00231941
           0.00152593
           0.000549333
          -0.000427259
          -0.00134281
```

```
 -0.0021363
 -0.00262459
 -0.00286874
 -0.00305185 ,

 44100.0f0,0x0010,Dict{Symbol,Any}(:fmt=>WAV.WAVFormat(0x0001,0x0001,0x0000ac44,0x00
```

In [47]:
```
wavplay(s,Fs)
```

In [48]:
```
# Check the signal with FFT
Fd=110
N=convert(Int32,ceil(Fs/Fd))
xx=collect(0:Fs/(2N):3000)
nn=length(xx)
plot(xx,abs(fft(s[1:800]))[1:nn])
```

Out[48]:



In [49]:
```
# Form the Hankel matrix
# IMPORTANT - Do not try to display H - it is a 50001 x 50001 matrix.
x=vec(s)
H=Hankel(x);
```

In [50]:
```
size(H), H[100,200]
```

Out[50]:
```
((50001,50001),0.727927488021485)
```

In [51]:
```
@time fft(x);
```

```
  0.022768 seconds (66 allocations: 3.055 MB)
```

```
In [52]: # We are looking for 20 eigenvalue pairs
         n=size(H,1)
         f(x)=H*x
         A=LinearMap(f,n,issym=true)
         size(A)
```

```
Out[52]: (50001,50001)
```

```
In [53]: @time λ,U=eigs(A, nev=40, which=:LM)
```

```
  4.749972 seconds (28.89 k allocations: 1.698 GB, 3.86% gc time)
```

```
Out[53]: ([1802.91,-1799.5,1297.45,-1296.27,-537.888,537.759,416.906,-415.32,410.179,-410.15
         50001x40 Array{Float64,2}:
          -0.0147284   -0.00759017    0.0125584    ...   0.000934865  -0.0328479
          -0.0151816   -0.00665314    0.0135616       -0.00407886   -0.0344853
          -0.0155747   -0.00568943    0.014351        -0.0088179    -0.0358005
          -0.0159061   -0.00470276    0.014914        -0.0130764    -0.0366518
          -0.0161744   -0.00369697    0.0152413       -0.0166468    -0.0369278
          -0.0163783   -0.00267602    0.0153278    ...  -0.0193304    -0.0365613
          -0.0165171   -0.00164396    0.0151718       -0.020949     -0.0355414
          -0.0165903   -0.000604893   0.0147759       -0.0213583    -0.0339241
          -0.0165975    0.000437052   0.0141463       -0.0204607    -0.0318377
          -0.0165389    0.00147771    0.013293        -0.0182185    -0.0294777
          -0.0164148    0.00251295    0.0122293    ...  -0.0146626    -0.0270884
          -0.0162257    0.00353864    0.0109722       -0.0098976    -0.0249307
          -0.0159723    0.00455073    0.00954145      -0.00409871   -0.0232418
           ⋮                                        ⋱
           0.00208837   0.000150943   0.00186399       0.000133997  -0.000140931
           0.00209398   1.89465e-5    0.00175745   ...   7.30299e-5   -0.000137052
           0.00209137  -0.000113123   0.00162337       9.67986e-6   -0.000122122
           0.00208056  -0.000244744   0.00146385      -5.28667e-5   -9.9483e-5
           0.00206158  -0.0003754     0.00128141      -0.000111942  -7.34335e-5
           0.00203452  -0.000504576   0.0010789       -0.000165542  -4.84584e-5
           0.00199948  -0.000631765   0.000859501  ...  -0.00021239   -2.87006e-5
           0.0019566   -0.000756467   0.000626668     -0.000251889  -1.73059e-5
           0.00190604  -0.00087819    0.00038406      -0.000283974  -1.59346e-5
           0.00184799  -0.000996451   0.000135499     -0.000308894  -2.44981e-5
           0.00178267  -0.00111078   -0.000115097     -0.000326988  -4.12385e-5
           0.00171032  -0.00122072   -0.000363773  ...  -0.000338463  -6.2934e-5   ,

         40,3,123,[0.127659,0.307902,0.495536,0.696658,0.888305,0.97497,0.873073,0.590491,0
```

```
In [54]: # Count the eigenvalue pairs (+-) larger than the 10% of the maximum
         τ=0.1
         L=round(Int,(sum(abs(λ).>(τ*maxabs(λ)))/2))
```

```
Out[54]: 11
```

At this point, the implementation using full matrices is rather obvious. However, we cannot do that, due to large dimension. Recall, the task is to define Hankel matrices H_k for $k = 1, \ldots, L$, from the signal obtained by averaging the skew-diagonals of the matrices

$$H_k = \lambda_k U_{:,k} U_{:,k}^T + \lambda_{n-k+1} U_{:,n-k+1} U_{:,n-k+1}^T,$$

without actually forming the matrices.

This is a nice programming excercise which can be solved using $\cdot$ products.

```
In [55]: function myaverages{T}(λ::T, u::Vector{T})
             n=length(u)
             x=Array(Float64,2*n-1)
             # Average lower diagonals
             for i=1:n
                 x[i]=dot(u[1:i],reverse(u[1:i]))/i
             end
             for i=2:n
                 x[n+i-1]=dot(u[i:n],reverse(u[i:n]))/(n-i+1)
             end
             λ*x
         end

Out[55]: myaverages (generic function with 1 method)

In [56]: # A small test
         u=[1,2,3,4,5]
         u*u'

Out[56]: 5x5 Array{Int64,2}:
          1   2   3   4   5
          2   4   6   8  10
          3   6   9  12  15
          4   8  12  16  20
          5  10  15  20  25

In [57]: myaverages(1,u)

Out[57]: 9-element Array{Float64,1}:
           1.0
           2.0
           3.33333
           5.0
           7.0
          11.0
          15.3333
          20.0
          25.0
```

We now execute the first step of the algorithm from the above Fact.

Notice that `eigs()` returns the eigenvalues arranged by the absoulte value, so the consecutive pairs define the i-th signal. The computation of averages is long - it requires $O(n^2)$ operations and takes several minutes.

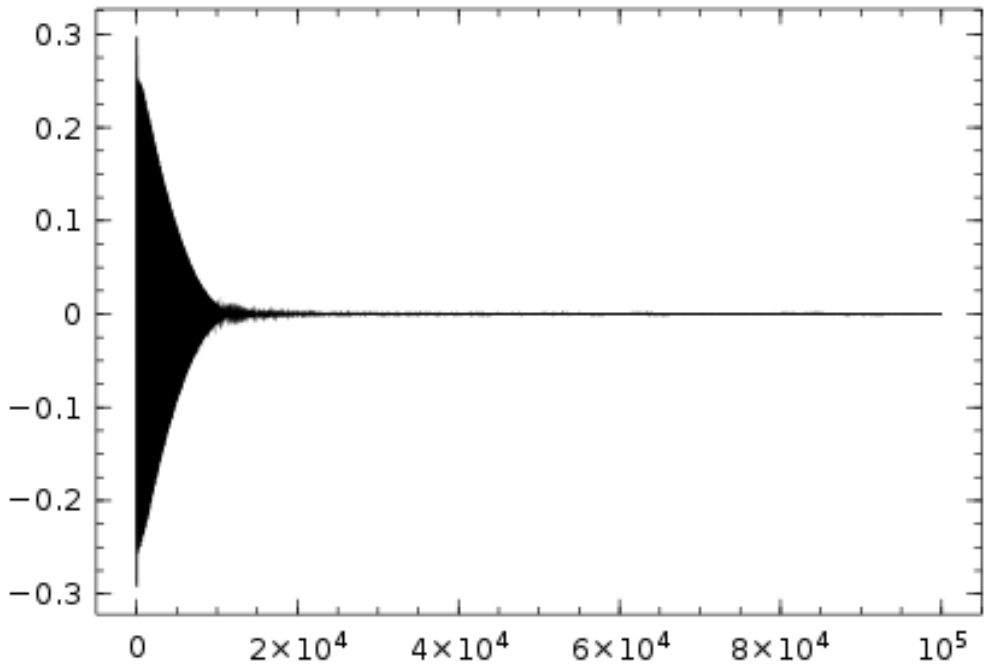
```
In [58]: xcomp=Array(Array{Float64},L)
         for k=1:L
             xcomp[k]=myaverages(λ[2*k-1],U[:,2*k-1])+myaverages(λ[2*k],U[:,2*k])
         end
```

Let us look and listen to what we got:

```
In [59]: # Entire components
         @manipulate for k=1:L
             plot(xcomp[k])
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(6, nactions=0),"k",6,1:11,true)
```

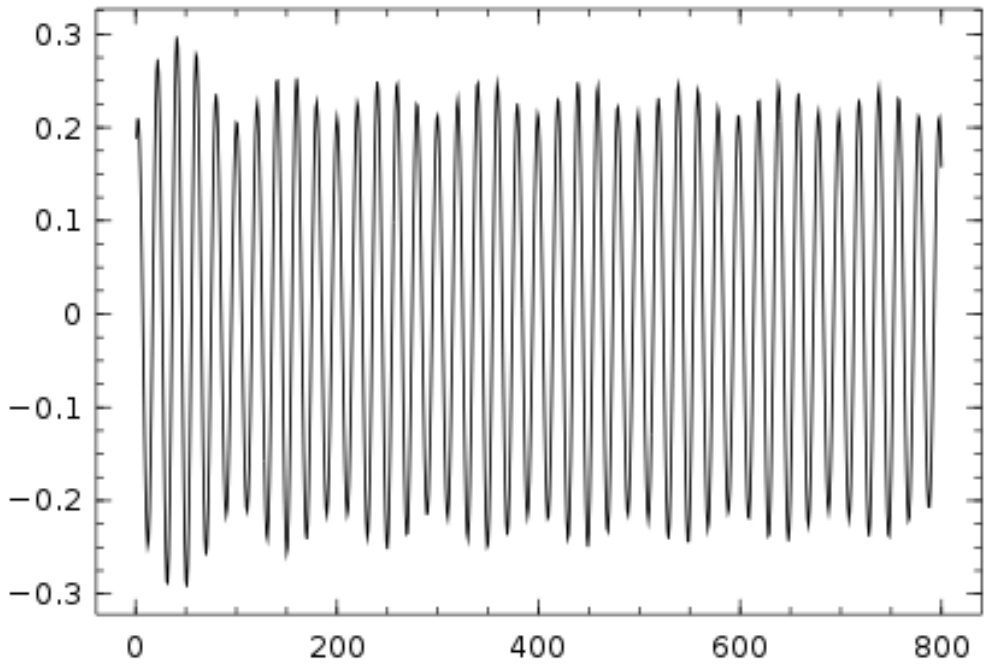
Out[59]:




```
In [60]: # Short parts of components
         @manipulate for k=1:L
             plot(xcomp[k][1:800])
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(6, nactions=0),"k",6,1:11,true)
```

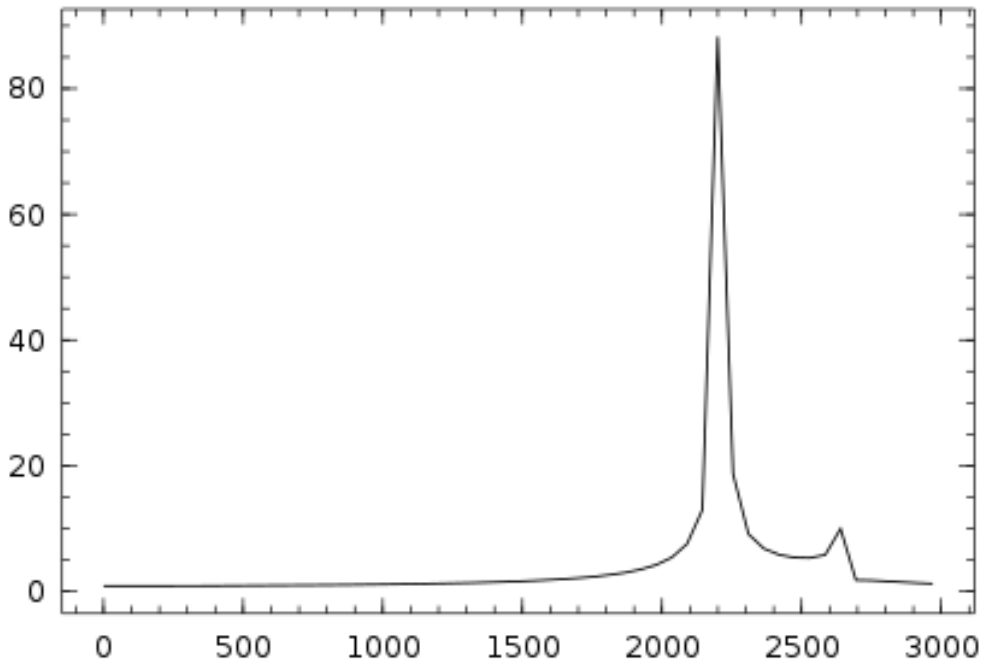
Out[60]:



In [61]:
```
# FFTs of short parts
Fd=110
N=convert(Int32,ceil(Fs/Fd))
xx=collect(0:Fs/(2N):3000)
nn=length(xx)
@manipulate for k=1:L
    plot(xx,abs(fft(xcomp[k][1:800]))[1:nn])
end
```

```
Interact.Slider{Int64}(Signal{Int64}(6, nactions=0),"k",6,1:11,true)
```

Out[61]:



We see that all `xcomp[k]` are clean mono-components - see Physics of Music - Notes:

```
1 = 440 Hz
2 = 880 Hz (2*440,+octave)
3 = 1320 Hz (3*440,+quint)
4 = 440 Hz
5 = 880 Hz
6 = 2200 Hz (5*440,++major terza, C#)
7 = 2640 Hz (6*440,++quint)
8 = 440 Hz
9 = 2200 Hz
10 = 1760 Hz (4*440,++octave)
11 = 2640 Hz
```

N.B. Some mono-components are repeated, and they should be grouped by adding components with absolute weighted correlation larger than some prescribed threshold.

```
In [63]: # Listen to components - wait between k's
         # @manipulate for k in slider(1:L, value=1)
         wavplay(xcomp[1],Fs)
         # end
```

```
In [64]: wavplay(sum([xcomp[i] for i=1:11]),Fs)
```

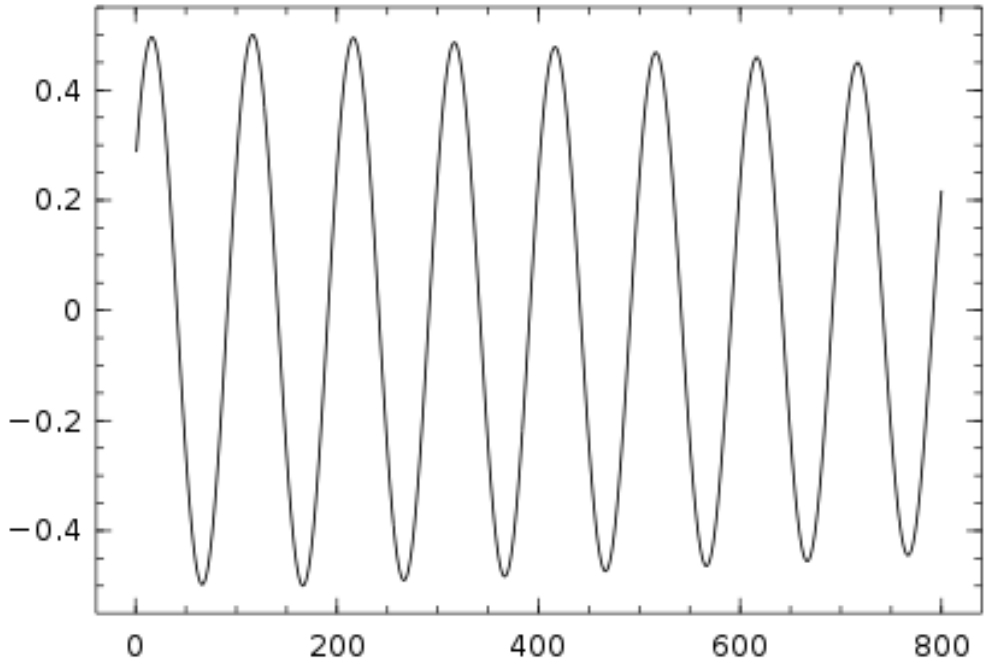
Can we do without averaging?

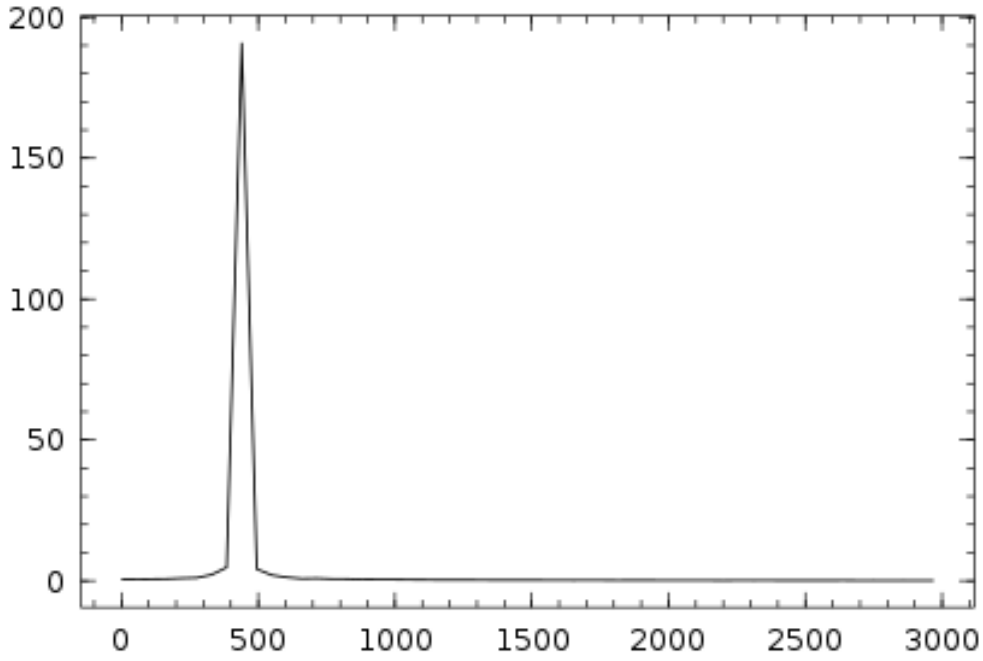
The function `myaverages()` is very slow - 7 minutes, compared to 5 seconds for the eigenvalue computation.

The simplest option is to disregard the averages, and use the first column and the last row of the underlying matrix, as in definition of Hankel matrices. Smarter approach might be to use small random samples to compute the averages.

Let us try the simple approach for the fundamental frequency.

```
In [65]: p=Array(Any,3)
         xsimple=[(λ[1]*U[1,1])*U[:,1]; (λ[1]*U[n,1])*U[2:n,1]]
         xsimple+=[(λ[2]*U[1,2])*U[:,2]; (λ[2]*U[n,2])*U[2:n,2]]
         p[1]=plot(xsimple[1:800])
         p[2]=plot(xx,abs(fft(xsimple[1:800]))[1:nn])
         @show norm(xcomp[1]-xsimple)/norm(xcomp[1])
         display(p[1]), display(p[2])
```






```
norm(xcomp[1] - xsimple) / norm(xcomp[1]) = 0.2335481934493818
```

```
Out[65]: (nothing,nothing)
```

```
In [66]: # Listen
         wavplay(xsimple,Fs)
```

```
In [ ]:
```

7 Compressive Sensing

Images or signals of scientific interest accurately and even exactly from a number of samples which is far smaller than the desired resolution of the image/signal, e.g., the number of pixels in the image. This new technique draws from results in several fields

Suppose we are given a sparse signal.

Can we recover the signal with small number of measurements (far smaller than the desired resolution of the signal)?

The answer is YES, for some signals and carefully selected measurements using l_1 minimization.

7.1 Prerequisites

The reader should be familiar to elementary concepts about signals, with linear algebra concepts, and linear programming.

7.2 Competences

The reader should be able to recover a signal from a small number of measurements.

7.3 References

For more details see

- E. Candès and M. Wakin, An Introduction To Compressive Sampling,
- M. Davenport et al., Introduction to Compressed Sensing
- O. Holtz, Compressive sensing: a paradigm shift in signal processing,
- and the extensive list of Compressive Sensing Resources.

Credits: Daniel Bragg, an IASTE Intern, performed testing of some of the methods.

7.4 Underdetermined systems

Let $A \in \mathbb{R}^{m \times n}$ with $m < n$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$.

7.4.1 Definitions

The system $Ax = b$ is **underdetermined**.

$\|x\|_0$ is the number of nonzero entries of x (*a quasi-norm*).

A matrix A satisfies the **restricted isometry property** (RIP) of order k with constant $\delta_k \in (0, 1)$ if

$$(1 - \delta_k)\|x\|_2^2 \leq \|Ax\|_2^2 \leq (1 + \delta_k)\|x\|_2^2$$

for any x such that $\|x\|_0 \leq k$.

A **mutual incoherence** of a matrix A is

$$\mathcal{M}(A) = \max_{i \neq j} |[A^T A]_{ij}|,$$

that is, the absolutely maximal inner product of distinct columns of A. If the columns of A have unit norms, $\mathcal{M}(A) \in [0, 1]$.

The **spark** of a given matrix A, $spark(A)$, is the smallest number of columns of A that are linearly dependant.

7.4.2 Facts

1. An underdetermined system either has no solution or has infinitely many solutions.
2. The typical problem is to choose the solution of some minimal norm. This problem can be reformulated as a constrained optimization problem

$$\text{minimize } \|x\| \quad \text{subject to} \quad Ax = b.$$

In particular:

3. For the 2-norm, the l_2 minimization problem is solved by SVD: let $\text{rank}(A) = r$ and let $A = U\Sigma V^T$ be the SVD of A. Then

$$x = \sum_{k=1}^{r} \frac{U[:,k]^T b}{\sigma_k} V[:,k].$$

4. For the 1-norm, the l_1 minimization problem is a **linear programming** problem

$$\text{minimize } \ c^T x \quad \text{subject to} \quad Ax = b, \ x \geq 0,$$

for $c^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}$.

5. For the 0-norm, the l_0 problem (which appears in compressive sensing)

$$\text{minimize } \|x\|_0 \quad \text{subject to} \quad Ax = b,$$

is NP-hard.

3. It holds $spark(A) \in [2, m+1]$.
4. For any vector b, there exists at most one vector x such that $\|x\|_0 \leq k$ and $Ax = b$ if and only if $spark(A) > 2k$. This implies that $m \geq 2k$, which is a good choice when we are computing solutions which are exactly sparse.
5. If $k < \frac{1}{2}\left(1 + \frac{1}{\mathcal{M}(A)}\right)$, then for any vector b there exists at most one vector x such that $\|x\|_0 \leq k$ and $Ax = b$.
6. If the solution x of l_0 problem satisfies $\|x\|_0 < \frac{\sqrt{2} - 1/2}{\mathcal{M}(A)}$, then the solution of l_1 problem is the solution of l_0 problem!
7. If A has columns of unit-norm, then A satisfies the RIP of order k with $\delta_k = (k-1)\mathcal{M}(A)$ for all $k < 1/\mathcal{M}(A)$.
8. If A satisfies RIP of order $2k$ with $\delta_{2k} < \sqrt{2} - 1$, then the solution of l_1 problem is the solution of l_0 problem!
9. Checking whether the specific matrix has RIP is difficult. If $m \geq C \cdot k \log\left(\frac{n}{k}\right)$, where C is some constant depending on each instance, the following classes of matrices satisfy RIP with $\delta_{2k} < \sqrt{2} - 1$ with overwhelming probability(the matrices are normalised to have columns with unit norms):
    1. Form A by sampling at random n column vectors on the unit sphere in $\mathbb{R}^m$.
    2. Form A by sampling entries from the normal distribution with mean 0 and variance $1/m$.

3. Form A by sampling entries from a symmetric Bernoulli distribution $P(A_{ij} = \pm 1/\sqrt{m}) = 1/2$.
4. Form A by sampling at random m rows of the Fourier matrix.

10. The **compressive sensing** interpretation is the following: the signal x is reconstructed from samples with m **functionals** (the rows of A).

7.4.3 Example - l_2 minimization

```
In [1]: A=rand(5,8)
        b=rand(5)
        A,b
```

```
Out[1]: (
        5x8 Array{Float64,2}:
         0.8082    0.666244  0.515028  0.865917   ...  0.852832  0.834534  0.615418
         0.74705   0.358617  0.115787  0.0420732     0.696485  0.652061  0.338238
         0.37271   0.981246  0.891748  0.105516      0.218208  0.348347  0.951332
         0.610068  0.583673  0.512855  0.459512      0.32478   0.926941  0.281732
         0.107041  0.46795   0.235619  0.642902      0.207894  0.934571  0.679717,

        [0.1741897819893825,0.6556281204579864,0.16999675769846978,0.09685785580340034,0.582
```

```
In [2]: x=A\b
        U,σ,V=svd(A)
        norm(A*x-b), norm( sum( [(U[:,k]'*b/σ[k])[1]*V[:,k]  for k=1:5])-x)
```

```
Out[2]: (6.261943282059023e-16,2.9414690675170558e-15)
```

7.4.4 Examples - Exact sparse signal recovery

We recover randomly generated sparse signals "measured" with rows of the matrix A. The experiment is performed for types of matrices from Fact 9.

The l_1 minimization problem is solved using the function `linprog()` from the package MathProgBase.jl. This function requires the linear programming solver from the package Clp.jl be installed beforehand (it is a longer compilation).

Random matrices are generated using the package Distributions.jl.

For more details see the documentation.

```
In [4]: using Winston
        using Clp
        using MathProgBase
        using Distributions
```

```
In [5]: whos(MathProgBase)
```

```
                MathProgBase    287 KB     Module
                     linprog   8802 bytes  Function
                  mixintprog   7818 bytes  Function
                    quadprog   7443 bytes  Function
```

```
In [6]: # Small example
        l1 = linprog([-1,0],[2 1],'<',1.5)
```

```
Out[6]: MathProgBase.HighLevelInterface.LinprogSolution(:Optimal,-0.75,[0.75,0.0],Dict{Any,A
```

```
In [7]: fieldnames(l1)
```

```
Out[7]: 4-element Array{Symbol,1}:
         :status
         :objval
         :sol
         :attrs
```

```
In [8]: # Random vectors on a unit sphere
        n=100
        m=40
        k=15
        A=svd(rand(m,n))[3]'
        for i=1:size(A,2)
            A[:,i]=A[:,i]/norm(A[:,i])
        end
        # Check incoherence
        μ=maxabs(A'*A-I)
        # Compute a random vector
        xs=sprand(n,1,k/n)
        @show nnz(xs)
        x=vec(full(xs))
        # Sampling
        b=A*x
```

```
nnz(xs) = 11
```

```
Out[8]: 40-element Array{Float64,1}:
         -0.907472
          0.547809
         -0.019405
          0.428529
         -0.55461
          0.170724
         -0.249453
          0.211141
         -0.00334435
         -0.101365
          0.251732
         -0.455229
         -0.0196577
          ⋮
         -0.375376
         -0.151764
         -0.0712805
```

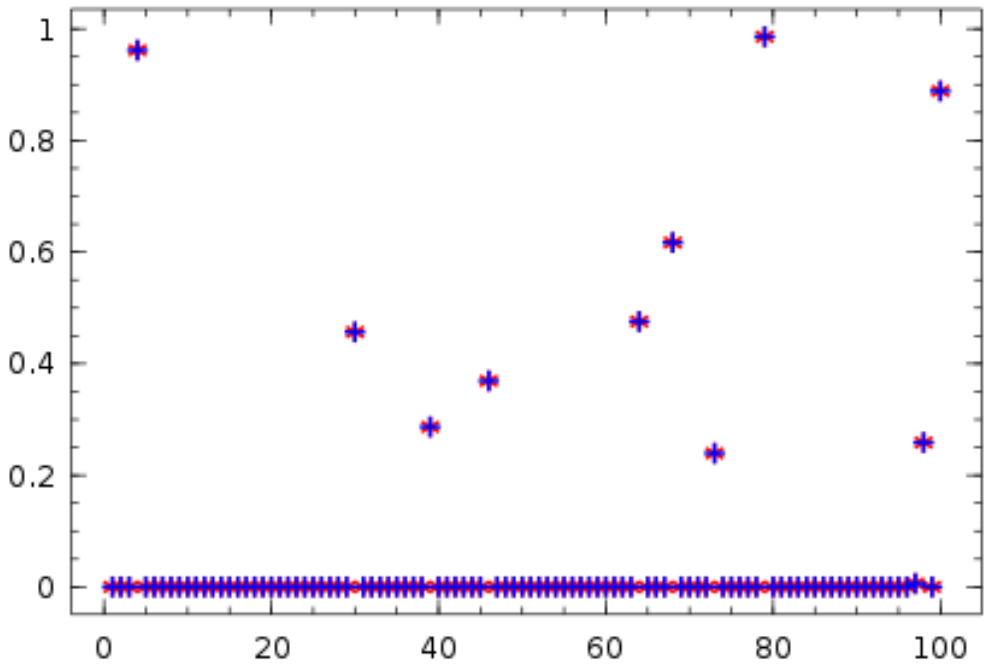
```
 -0.00278336
 -0.177359
 -0.173566
 -0.0643775
  0.0275448
  0.890327
 -0.522219
  0.0418145
  0.156516
```

In [9]:
```
# Recovery
c=ones(n)
l1=linprog(c,A,'=',b,0,Inf)
```

Out[9]:
```
MathProgBase.HighLevelInterface.LinprogSolution(:Optimal,5.545528118916206,[0.0,0.0
```

In [10]:
```
# Check
plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
```

Out[10]:



In [11]:
```
# Let us make some more tests
using Interact
```

In [12]:
```
@manipulate for n=50:50:500, m=20:10:200, k=10:10:100
    A=svd(rand(m,n))[3]'
    for i=1:size(A,2)
```

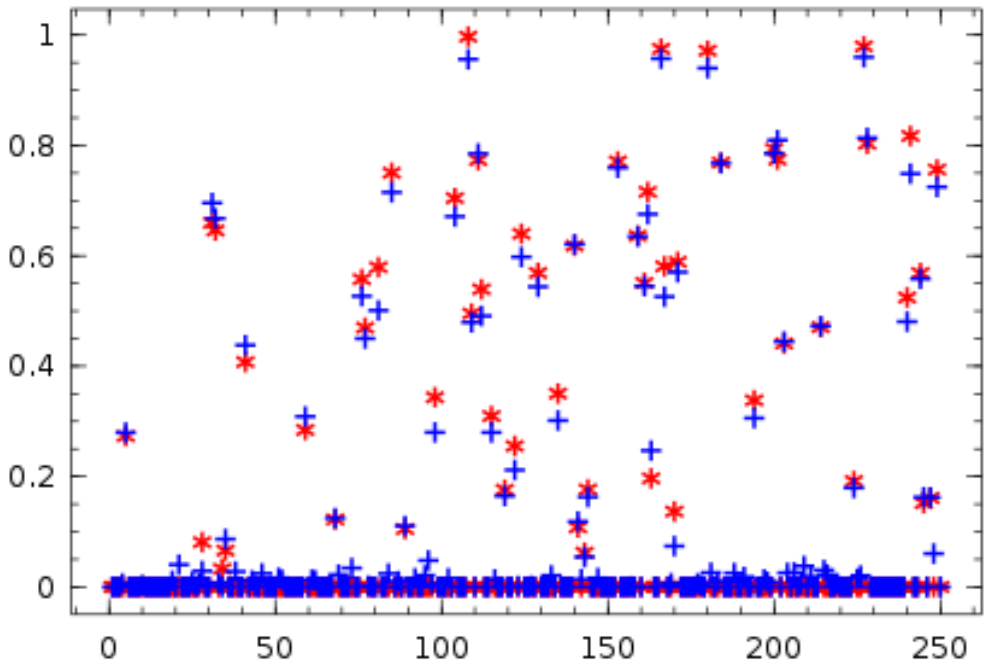
```
            A[:,i]=A[:,i]/norm(A[:,i])
        end
        # Compute a random vector
        xs=sprand(n,1,k/n)
        x=vec(full(xs))
        # Sampling
        b=A*x
        # Recovery
        l1=linprog(ones(n),A,'=',b,0,Inf)
        plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
    end
```

```
Interact.Slider{Int64}(Signal{Int64}(250, nactions=0),"n",250,50:50:500,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(110, nactions=0),"m",110,20:10:200,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(50, nactions=0),"k",50,10:10:100,true)
```

Out[12]:



In [13]:
```
# Normal distribution
@manipulate for n=50:50:500, m=20:10:200, k=10:10:100
    A=rand(Normal(0,1/m),m,n)
    for i=1:size(A,2)
        A[:,i]=A[:,i]/norm(A[:,i])
```

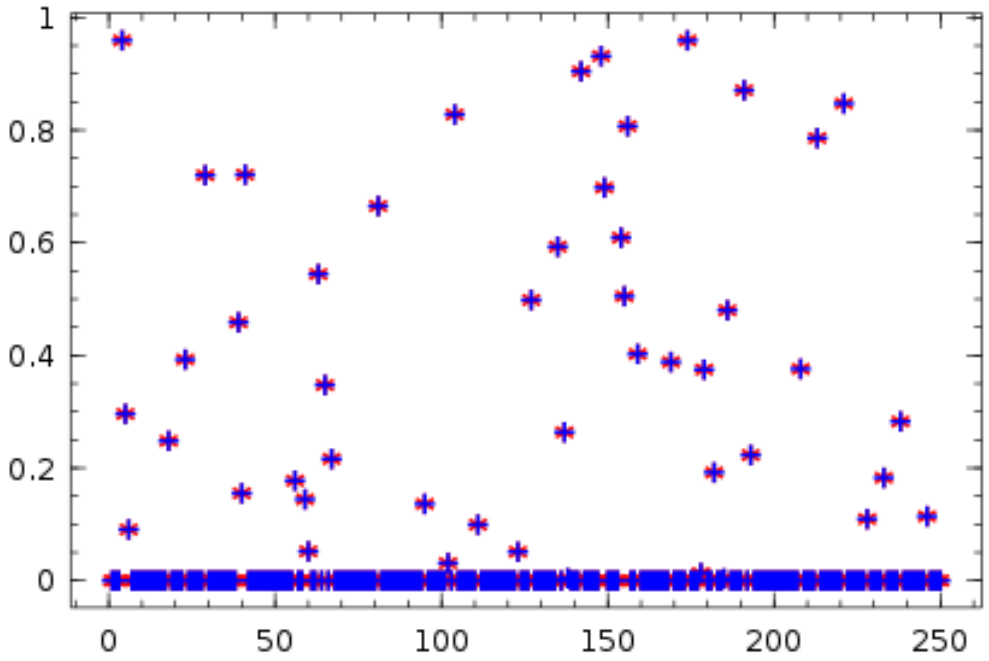
```
            end
            # Compute a random vector
            xs=sprand(n,1,k/n)
            x=vec(full(xs))
            # Sampling
            b=A*x
            # Recovery
            l1=linprog(ones(n),A,'=',b,0,Inf)
            plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
        end
```

```
Interact.Slider{Int64}(Signal{Int64}(250, nactions=0),"n",250,50:50:500,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(110, nactions=0),"m",110,20:10:200,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(50, nactions=0),"k",50,10:10:100,true)
```

Out[13]:



In [14]:
```
# Symmetric Bernoulli distribution
@manipulate for n=50:50:500, m=20:10:200, k=10:10:100
    # The matrix of (-1,1)s
    A=2*(rand(Bernoulli(0.5),m,n)-0.5)
    for i=1:size(A,2)
        A[:,i]=A[:,i]/norm(A[:,i])
```

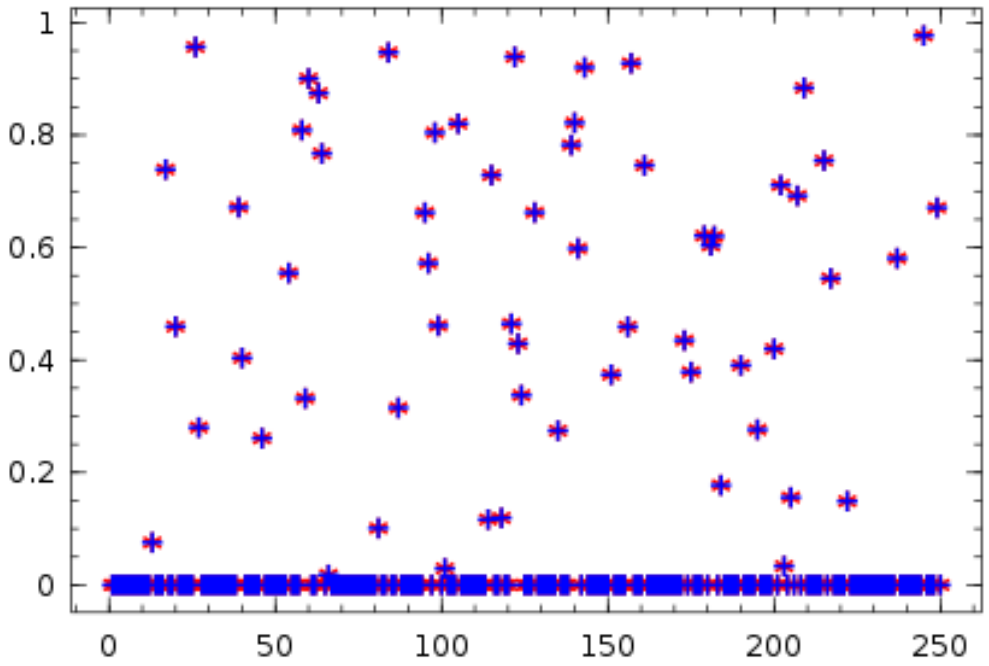
```
        end
        # Compute a random vector
        xs=sprand(n,1,k/n)
        x=vec(full(xs))
        # Sampling
        b=A*x
        # Recovery
        l1=linprog(ones(n),A,'=',b,0,Inf)
        plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
    end
```

```
Interact.Slider{Int64}(Signal{Int64}(250, nactions=0),"n",250,50:50:500,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(110, nactions=0),"m",110,20:10:200,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(50, nactions=0),"k",50,10:10:100,true)
```

Out[14]:



In [15]:
```
# Fourier matrix
@manipulate for n=50:50:500, m=20:10:200, k=10:10:100
    # Elegant way of computing the Fourier matrix
    F=fft(eye(n),1)
    # Select m/2 random rows
    ind=randperm(n)[1:round(Int,m/2)]
```

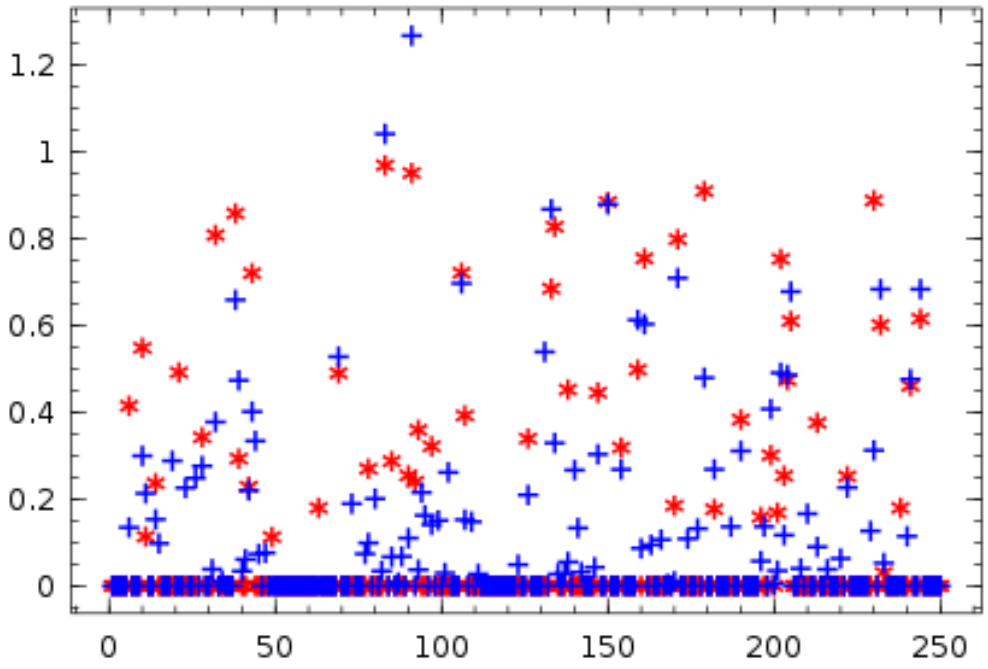
```
        Fm=F[ind,:]
        # We need to work with real matrices due to linprog()
        A=[real(Fm); imag(Fm)]
        for i=1:size(A,2)
            A[:,i]=A[:,i]/norm(A[:,i])
        end
        # Compute a random vector
        xs=sprand(n,1,k/n)
        x=vec(full(xs))
        # Sampling
        b=A*x
        # Recovery
        l1=linprog(ones(n),A,'=',b,0,Inf)
        plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
    end
```

```
Interact.Slider{Int64}(Signal{Int64}(250, nactions=0),"n",250,50:50:500,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(110, nactions=0),"m",110,20:10:200,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(50, nactions=0),"k",50,10:10:100,true)
```

Out[15]:



7.5 Signal recovery from noisy observations

In the presence of noise in observation, we want to recover a vector x from $b = Ax + z$, where z is a stochastic or deterministic unknown error term.

7.5.1 Definition

The **hard thresholding operator**, $H_k(x)$, sets all but the k entries of x with largest magnitude to zero.

7.5.2 Facts

1. The problem can be formulated as l_1 minimization problem

$$\text{minimize } \|x\|_1 \quad \text{subject to} \quad \|Ax - b\|_2^2 \leq \epsilon,$$

   where ϵ bounds the amount of noise in the data.

2. Assume that A satisfies RIP of order $2k$ with $\delta_{2k} < \sqrt{2} - 1$. Then the solution $x^\star$ of the above problem satisfies

$$\|x^\star - x\|_2 \leq C_0 \frac{1}{\sqrt{k}} \|x - H_k(x)\|_1 + C_1 \epsilon,$$

   where x is the original signal.

3. The l_1 problem is a convex programming problem and can be efficiently solved. The solution methods are beyond the scope of this course.

4. If k is known in advance, A satisfies RIP with $\delta_{3k} < 1/15$, and $\|A\|_2 < 1$, the k-sparse aproximation of x can be computed by the *Iterative Hard Thresholding* algorithm

   1. *Initialization*: $x = 0$.
   2. *Iteration*: repeat until convergence $x = H_k(x + A^T(b - Ax))$.

7.5.3 Example

We construct the k sparse x, form b, add noise, and recover it with the algorithm from Fact 4. The conditions on A are rather restrictive, which means that k must be rather small compared to n and m must be rather large. For convergence, we limit the number of iterations to $50m$.

```
In [16]: n=300
         # k is small compared to n
         k=8
         x=10*sprand(n,1,k/n)
         # Reset k
         k=nnz(x)
         # Define m, rather large
         m=5*round(Int,k*log(n/k))
         # Sampling matrix - normal distribution
         A=rand(Normal(0,1/m),m,n)
         A=A/(norm(A)+1)
         # Form b
         b=vec(A*x)
```

```
# Add noise
noise=rand(m)*1e-5
b+=noise
```

```
Out[16]: 130-element Array{Float64,1}:
  0.047134
  0.0295181
  0.0238846
  0.156232
  0.0365629
  0.100764
  0.0245394
  0.0832066
  0.025514
  0.111253
  0.00582922
 -0.184009
  0.0860141
  ⋮
  0.0986965
  0.0161356
  0.0500226
  0.0580203
  0.161844
 -0.0482701
 -0.0103964
 -0.054215
 -0.0310472
 -0.0137488
 -0.0946905
  0.0712336
```

```
In [17]: # Iterative Hard Thresholding
function H(x::Vector,k::Int)
    y=deepcopy(x)
    ind=sortperm(abs(y),rev=true)
    y[ind[k+1:end]]=0
    y
end
```

```
Out[17]: H (generic function with 1 method)
```

```
In [18]: function IHT(A::Matrix, b::Vector,k::Int)
    # Tolerance
    τ=1e-7
    x=zeros(size(A,2))
    for i=1:50*m
        x=H(x+A'*(b-A*x),k)
    end
    x
end
```

```
Out[18]: IHT (generic function with 1 method)
```

```
In [19]: norm(A),k,m
```

```
Out[19]: (0.1774854293162885,7,130)
```

```
In [20]: y=IHT(A,b,k)
         norm(A*x-b)/norm(b)
```

```
Out[20]: 7.044999744637829e-5
```

```
In [21]: # print([x y])
```

Let us try linear programing in the case of noisy observations.

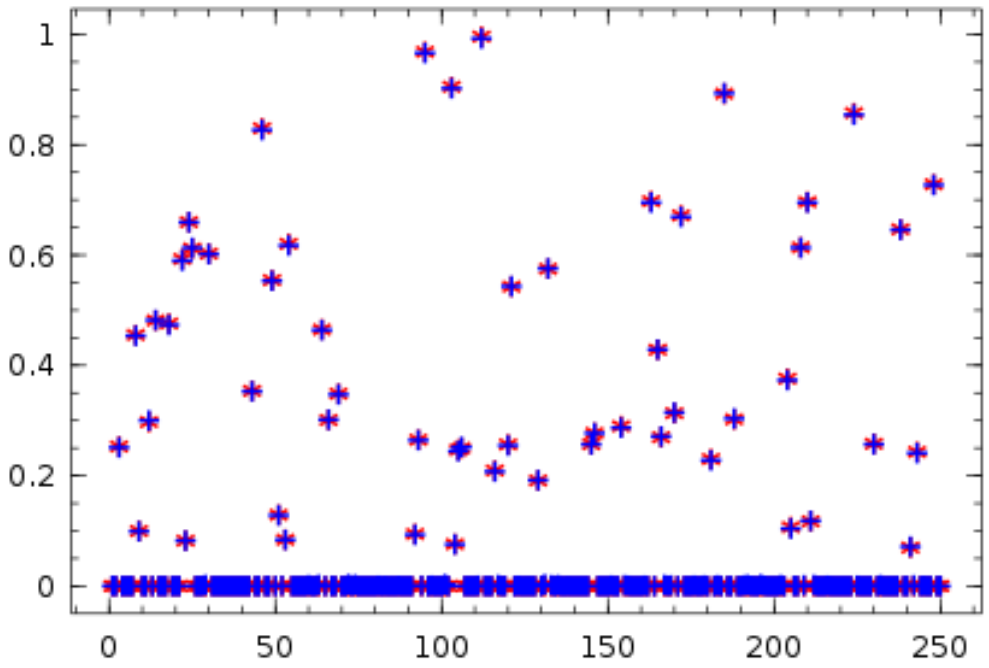
```
In [22]: # Try with noise
         # Normal distribution
         @manipulate for n=50:50:500, m=20:10:200, k=10:10:100
             A=rand(Normal(0,1/m),m,n)
             for i=1:size(A,2)
                 A[:,i]=A[:,i]/norm(A[:,i])
             end
             # Compute a random vector
             xs=sprand(n,1,k/n)
             x=vec(full(xs))
             # Sampling with noise
             b=A*x+(rand(m)-0.5)*1e-3
             # Recovery
             l1=linprog(ones(n),A,'=',b,0,Inf)
             plot(collect(1:n),x,"r*", collect(1:n),l1.sol,"b+")
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(250, nactions=0),"n",250,50:50:500,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(110, nactions=0),"m",110,20:10:200,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(50, nactions=0),"k",50,10:10:100,true)
```

```
Out[22]:
```



7.6 Sensing images

Wavelet transformation of an image is essentially sparse, since only small number of cofficients is significant. This fact can be used for compression.
Wavelet transforms are implemented the package Wavelets.jl.

7.6.1 Example - Lena

The `tif` version of the image has `65_798` bytes, the `png` version has `58_837` bytes, and the `jpeg` version has `26_214` bytes.

```
In [23]: using Wavelets
         using Colors
         using Images
         using TestImages

In [24]: whos(TestImages)

                    TestImages   2743 bytes  Module
                     testimage   1751 bytes  Function


In [25]: img=testimage("lena_gray_256")

Out[25]:
```

```
In [26]: show(img)
```

```
Gray Images.Image with:
  data: 256x256 Array{ColorTypes.Gray{FixedPointNumbers.UFixed{UInt8,8}},2}
  properties:
    colorspace: Gray
    spatialorder:  x y

WARNING: both Images and Winston export "Image"; uses of it in module Main must be qualifie
```

```
In [27]: # Convert the image to 32 bit floats
         x=map(Float32,img.data)
         sizeof(x)
```

```
Out[27]: 262144
```

```
In [28]: # Compute the wavelet transform of x
         xt = dwt(x, wavelet(WT.haar))
```

```
Out[28]: 256x256 Array{Float32,2}:
          124.494      -6.71662    -3.08563    ...   0.0137255    0.00392157
           13.3665      7.0653      3.63842        0.00784314   0.0196078
            1.72669    -3.289      -1.72203        0.0196078   -0.0215686
           -7.8185      1.52595    -0.3076        -0.0019608   -0.0235294
           -4.88774    -3.54069    -3.61808        0.0156863   -0.0196078
            3.03683    -0.580026    2.39632    ...  -0.00784315  -0.00980391
           -2.68431    -1.09467    -7.26575       -0.00588239  -0.00392155
```

```
-6.95172      5.94393      5.7076          0.027451    -0.0294118
-0.184681     2.84828      3.39816         0.0019608    0.0156863
 0.985048     1.15503      1.8799          0.0019608    0.0196079
 0.171813     0.229411     0.176103    ...   0.0039216   -2.09548e-8
 0.403432     2.52194      0.158945        0.00588235   0.00588235
-0.554657    -1.73248      1.86667        -0.0294118   -0.00588235
 ⋮                                      ⋱                  ⋮
 0.0470588    0.0215686   -0.00980389      0.00588236   0.00392157
-0.00196078   0.00980395   0.00196075  ...   0.00980393  -0.00392155
-0.0156862   -0.0196078   -0.00196078     -0.00980392  -0.00980393
-0.027451    -0.00980389   0.0            -0.00588233   0.0
-0.00784314  -0.00588235   0.0             0.0019608    0.0137255
-0.0039216   -0.00196081   0.0             0.0215686    0.00784314
 0.0117647    0.0117647    0.00392157  ...  -0.00784314  -0.0117647
-0.0313725   -0.0176471    0.00196081     -0.0156863    0.00196078
-0.00588235  -0.00196078   0.00588238     -0.00588236   0.00588235
 0.00196081   0.00588235  -0.0627451      -1.02445e-8   0.00392157
 0.12549      0.0470588   -0.0921569       0.0215686   -0.017647
-0.0666667   -0.0960785    0.0          ...  -0.0352941    0.0235294
```

We now set all but the 10% absolutely largest coefficients to zero and reconstruct the image. The images are very similar, which illustrates that the wavelet transform of an image is essentially sparse.

```
In [29]: ind=sortperm(abs(vec(xt)),rev=true)
         # 0.1 = 10%, try also 0.05 = 5%
         τ=0.1
         @show k=round(Int,τ*length(ind))
         xt1=copy(xt)
         xt1[ind[k+1:end]]=0
         # Inverse wavelet transform of the sparse data
         xsparse=idwt(xt1, wavelet(WT.haar))
         # New image
         imgsparse=Images.Image(map(Gray,map(Images.Clamp01NaN(xsparse),xsparse)'))
         # imgsparse=Images.Image(map(Gray,xsparse'))
         display(img), display(imgsparse)
```

```
k = round(Int,τ * length(ind)) = 6554
```

```
Out[29]: (nothing,nothing)
```

There are $k = 6554$ nonzero coefficients in a sparse wavelet representation.

Actual algorithms are elaborate. For more details see J. Romberg, Imaging via Compressive Sampling.

In [ ]:

8 Principal Component Analysis

PCA is an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. PCA analysis is perfromed either using EVD of the covariance matrix or the SVD of the mean-centered data.

8.1 Prerequisites

The reader should be familiar with linear algebra and statistics concepts.

8.2 Competences

The reader should be able to perform PCA on a given data set.

8.3 References

For more details see

- Principal Component Analysis,
- L. I. Smith, A tutorial on Principal Components Analysis,
- J. Shlens, A Tutorial on Principal Components Analysis.

8.4 Definitions

A **data matrix** is a matrix $X \in \mathbb{R}^{m\times n}$, where each column corresponds to a feature (say, certain gene), and each row correspond to an observation (say, individual).
A **mean** of a vector $x \in \mathbb{R}^n$ is $\mu(x) = \dfrac{x_1 + x_2 + \cdots x_n}{n}$.
A **standard deviation** of a vector x is $\sigma(x) = \sqrt{\dfrac{\sum_{i=1}^{n}(x_i - \mu(x))^2}{n-1}}$. A **variance** of a vector x is $var(x) = \sigma^2(x)$.
A **vector of means** of a data matrix X is a row-vector of means of the columns of X, $\mu(X) = \begin{bmatrix}\mu(X_{:,1}) & \mu(X_{:,2}) & \dots & \mu(X_{:,n})\end{bmatrix}$.
A **zero-mean centered data matrix** is a matrix $\bar{X}$ obtained from a data matrix X by subtracting from each column the mean of this column,

$$\bar{X} = \begin{bmatrix}X_{:,1} - \mu(X_{:,1}) & X_{:,2} - \mu(X_{:,2}) & \cdots & X_{:,n} - \mu(X_{:,n})\end{bmatrix} \equiv X - \mathbf{1}\mu(X),$$

where $\mathbf{1} = [1, 1, \dots, 1]^T$.
A **covariance matrix** of a data matrix X is a matrix

$$cov(X) = \frac{1}{n-1}[X - \mathbf{1}\mu(X)]^T[X - \mathbf{1}\mu(X)] \equiv \frac{\bar{X}^T\bar{X}}{n-1}.$$

8.5 Facts

Given a data matrix X, let $cov(X) = U\Lambda U^T$ be the EVD with non-increasingly ordered eigenvalues, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$.

1. $cov(X)$ is a symmetric PSD matrix.
2. $cov(X) = cov(\bar{X})$.
3. Let $T = \bar{X}U$. The columns of T are the **principal components** of $\bar{X}$. In particular:
    1. The first principal component of $\bar{X}$ (or X) is the first column, $T_{:,1}$. It is a projection of the zero-mean centered data set $\bar{X}$ on the line defined by $U_{:,1}$. This is the direction along which the data have the largest variance.
    2. The second column (the second principal component), $T_{:,1}$, is a projection of $\bar{X}$ on the line defined by $U_{:,2}$, which is orthogonal to the first projection. This is direction with the largest variance *after* subtracting the first principal component from $\bar{X}$.
    3. The k-th principal component is the direction with the largest variance *after* subtracting the first $k-1$ principal components from $\bar{X}$, that is, the first principal component of the matrix
    $$\hat{X} = \bar{X} - \sum_{i=1}^{k-1} \bar{X} U_{:,i} U_{:,i}^T.$$
4. Let $\bar{X} = \bar{U}\Sigma V^T$ be the SVD of $\bar{X}$. Then $V = U$ and $T = \bar{U}\Sigma V^T V \equiv \bar{U}\Sigma$.
5. Reconstruction of the principal components is the following:
    1. Full reconstruction is $X = TU^T + \mathbf{1}\mu(X)$.
    2. Reconstruction from the first k principal components is $\tilde{X} = TU_{:,1:k}^T + \mathbf{1}\mu(X)$.
6. Partial reconstructions can help obtaining various insights about the data. For example, the rows of the matrix $T_{:,1:k}$ can be clustered by the k-means algorithm, and the points defined by first three columns of T can be plotted to visualize projections of clusters. Afterwards, the computed clusters can be mapped back to original data.
7. Heuristical guess for number of important clusters is given be the location of the "knee" in the plot of the singular values of $\bar{X}$.

8.5.1 Example - Elliptical data set

We generate a "quasi" elliptical set of points and compute its principal components.

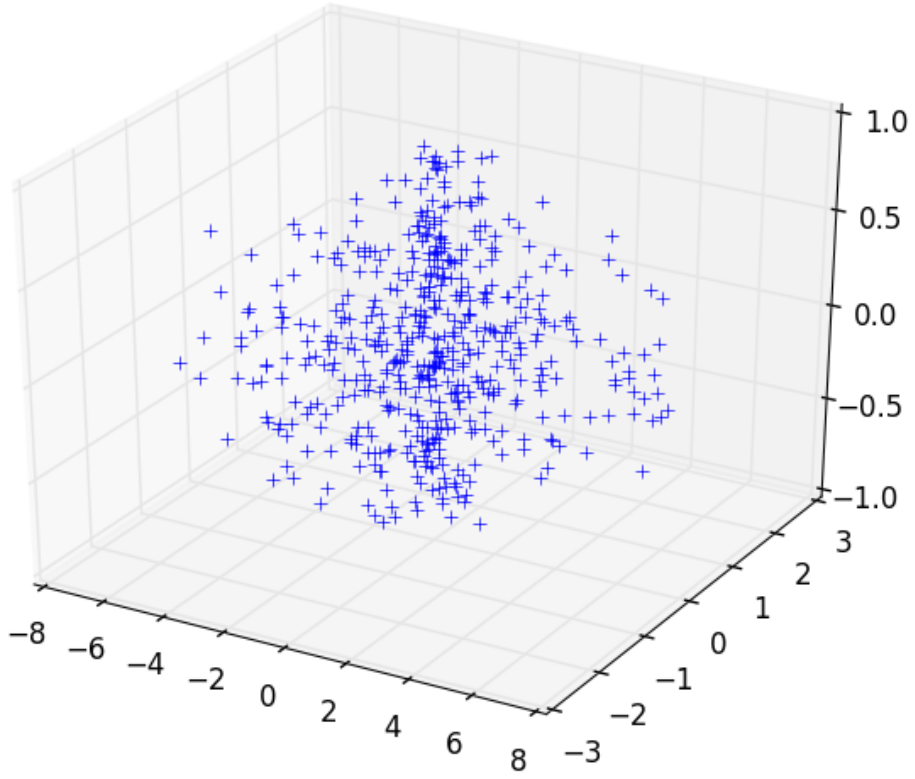
```
In [1]: # Generate data points
        n=3
        m=500
        ax=[8,3,1]
        X=Array(Float64,m,n)
        for i=1:n
            X[:,i]=rand!(X[:,i])*ax[i]
        end
        # Parameters
        u=(rand(m)-0.5)*π
        v=(rand(m)-0.5)*2*π
        for i=1:m
            X[i,1]=X[i,1]*cos(u[i])*cos(v[i])
            X[i,2]=X[i,2]*cos(u[i])*sin(v[i])
            X[i,3]=X[i,3]*sin(u[i])
        end
```

```
In [2]: X0=copy(X)
```

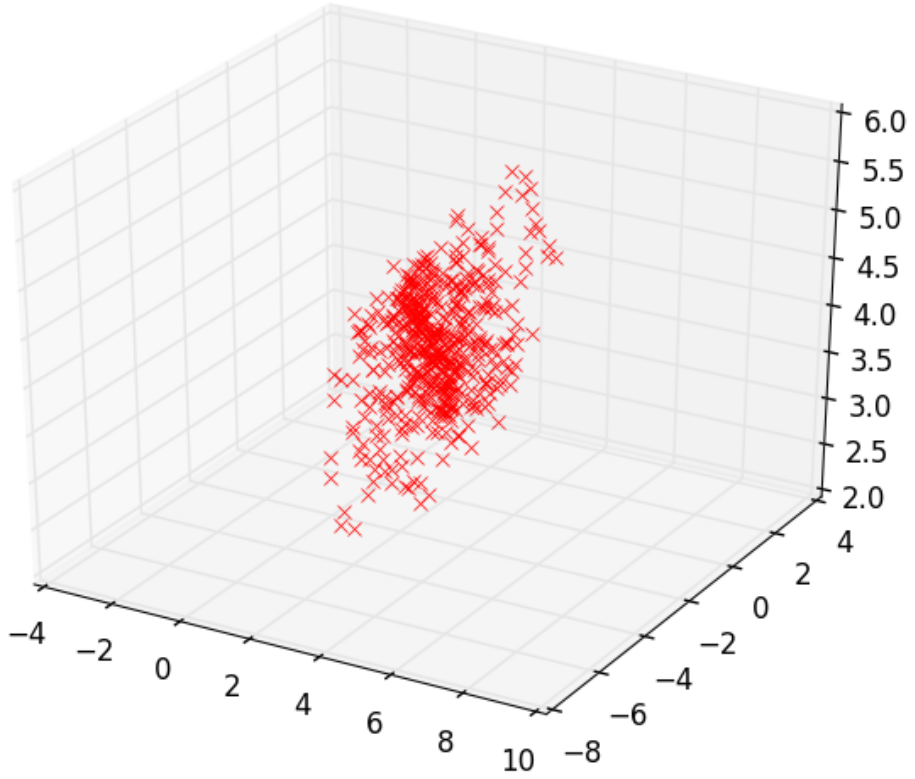
```
Out[2]: 500x3 Array{Float64,2}:
          1.16733     -0.255958    -0.565803
         -1.23365      0.361763     0.361941
          0.121376    -0.0912968   -0.0822644
         -0.817148     1.0718       0.0437761
         -3.4244      -2.10607     -0.252828
         -0.161428     2.72779      0.0398468
          4.17158      0.0224613   -0.211104
         -0.228965    -1.29079      0.0661065
         -1.01772      0.375041     0.882254
         -0.471328     0.0383805   -0.525105
         -1.06224      1.24535      0.0307377
          1.98785      0.101407     0.0888309
          5.62687      1.13674      0.263478
          ⋮
         -0.73109     -0.594582     0.174312
         -3.37536     -1.62934     -0.0363199
         -0.0639544    0.0868432    0.358064
          0.00119598   0.00365277   0.917788
         -0.896771    -0.73641      0.252674
         -0.35889     -0.363813     0.899639
         -2.83092     -0.00918657   0.171193
          0.0249949    1.17926      0.332681
         -4.64606      1.80153      0.00641655
          0.164225    -0.289255     0.430437
         -5.87256     -0.0813349    0.292073
         -1.47656     -0.010128    -0.601791
```

```
In [3]: using PyPlot
```

```
In [4]: plot3D(X0[:,1],X0[:,2],X0[:,3],"+")
```




```
Out[4]: 1-element Array{Any,1}:
         PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c41ffd0>

In [5]: # Compute the means. How good is the RNG?
        μ0=mean(X,1)

Out[5]: 1x3 Array{Float64,2}:
         -0.017171  -0.059913  0.016346

In [6]: # Subtract the means
        X=X.-μ0
        # Rotate by a random orthogonal matrix
        Q,r=qr(rand(3,3))
        X=X*Q;

In [7]: # Translate
        S=[3,-2,4]

Out[7]: 3-element Array{Int64,1}:
          3
         -2
          4

In [8]: X=X.+S'
        # plot3D(X0[:,1],X0[:,2],X0[:,3],"b+")
        plot3D(X[:,1],X[:,2],X[:,3],"rx" )
```



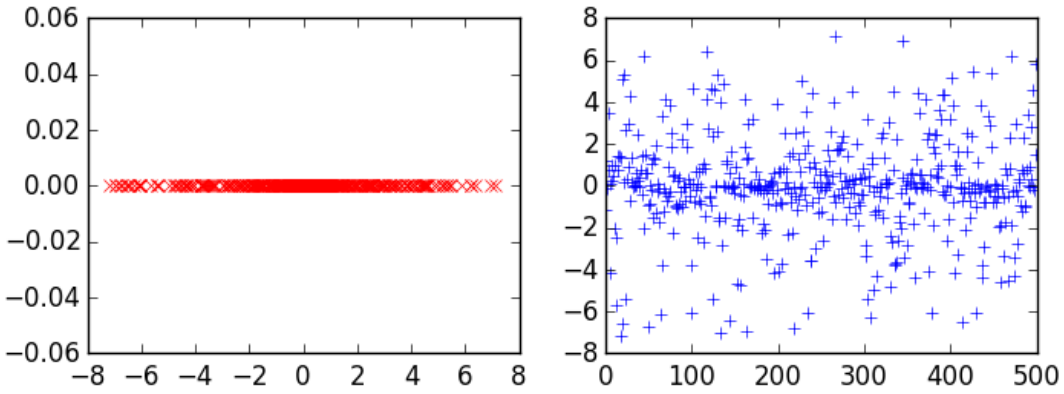

```
Out[8]: 1-element Array{Any,1}:
         PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c9e7e10>

In [9]: C=cov(X)

Out[9]: 3x3 Array{Float64,2}:
          3.37995   -2.4358     0.695062
         -2.4358     2.62768   -0.270418
          0.695062  -0.270418   0.425738

In [10]: μ=mean(X,1)

Out[10]: 1x3 Array{Float64,2}:
          3.0  -2.0  4.0

In [11]: # Fact 2
         cov(X.-μ), (X.-μ)'*(X.-μ)/(m-1)

Out[11]: (
         3x3 Array{Float64,2}:
           3.37995   -2.4358     0.695062
          -2.4358     2.62768   -0.270418
           0.695062  -0.270418   0.425738,
```

```
3x3 Array{Float64,2}:
  3.37995   -2.4358     0.695062
 -2.4358     2.62768   -0.270418
  0.695062  -0.270418   0.425738)
```

```
In [12]: # Principal components, evals are non-decreasing
         λ,U=eig(C)
```

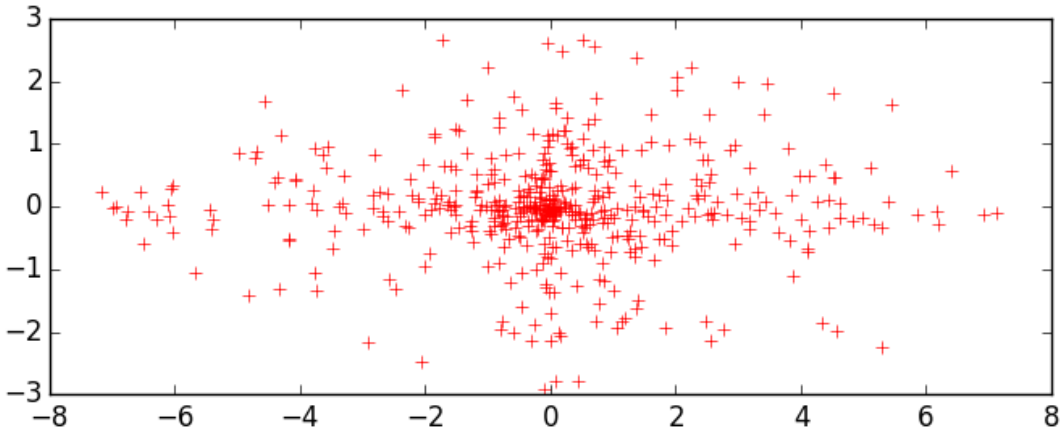
```
Out[12]: ([0.167829579566444,0.700475788350223,5.5650556928060375],
         3x3 Array{Float64,2}:
          -0.441864   0.482251   0.756433
          -0.346583   0.685972  -0.639783
           0.827428   0.544864   0.135967)
```

```
In [13]: # Largest principal component
         T1=(X.-μ)*U[:,3]
         subplot(221)
         plot(T1,zeros(T1),"rx")
         subplot(222)
         plot(T1,"b+")
```



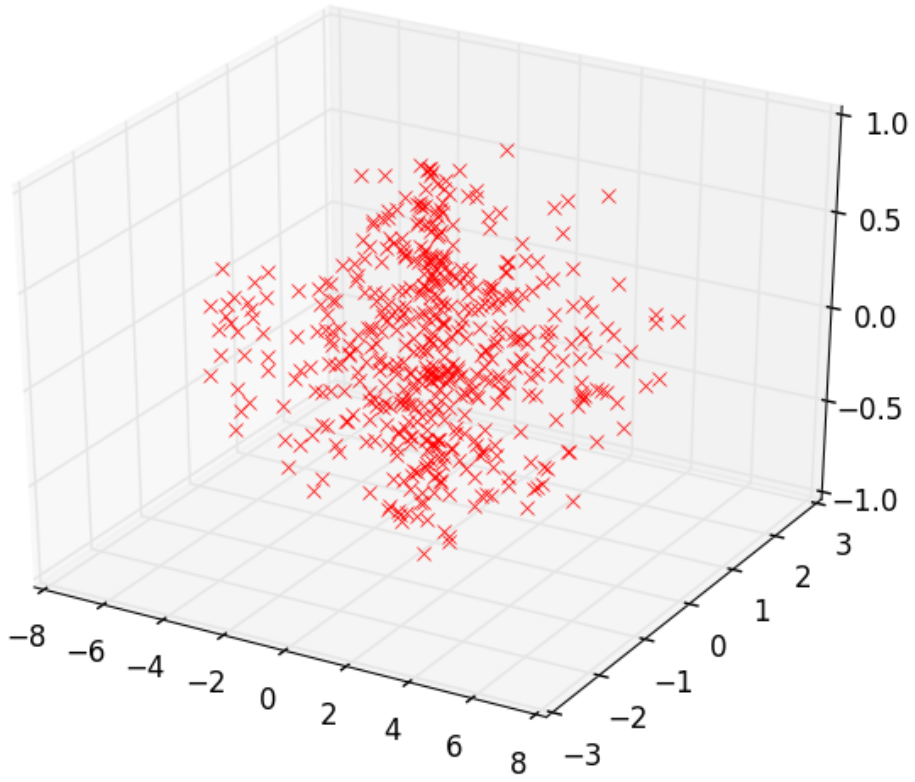

```
Out[13]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x7f5c8c28ec88>
```

```
In [14]: # Two largest principal components
         T2=(X.-μ)*U[:,[3,2]]
         axes(aspect="equal")
         plot(T2[:,1],T2[:,2],"r+")
```

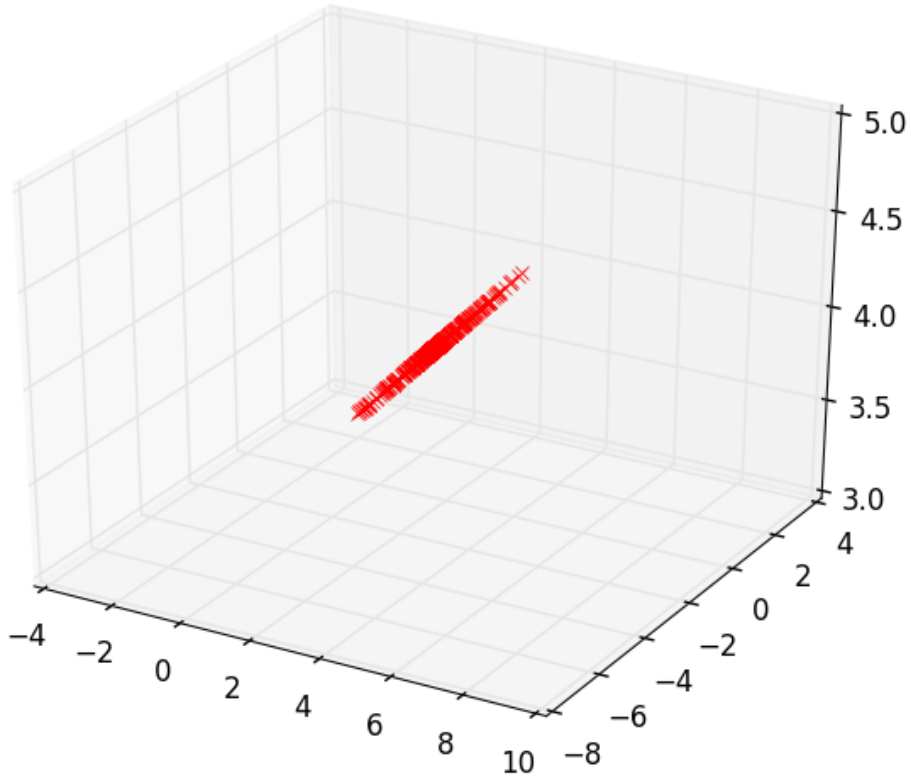



```
Out[14]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x7f5c8c1aa588>
```

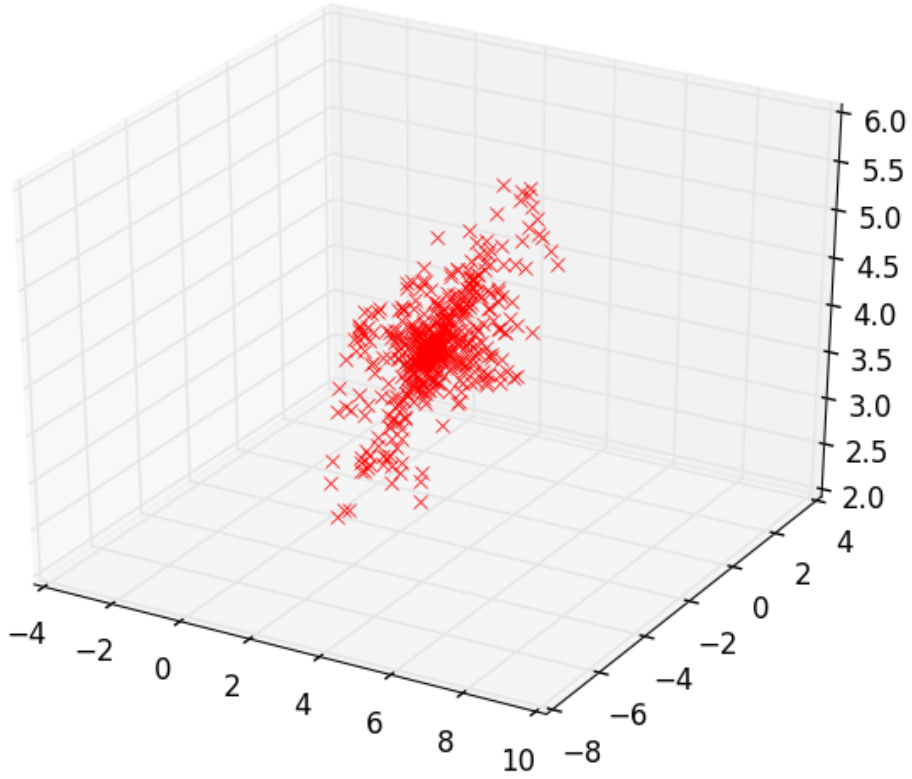
```
In [15]: # All  three principal components
         T=(X.-μ)*U[:,[3,2,1]]
         plot3D(T[:,1],T[:,2],T[:,3],"rx" )
```



```
Out[15]: 1-element Array{Any,1}:
          PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c1767b8>

In [16]: # Fact 5 - Recovery of the largest component
         Y1=T1*U[:,3]'.+μ
         plot3D(Y1[:,1],Y1[:,2],Y1[:,3],"rx" )
```

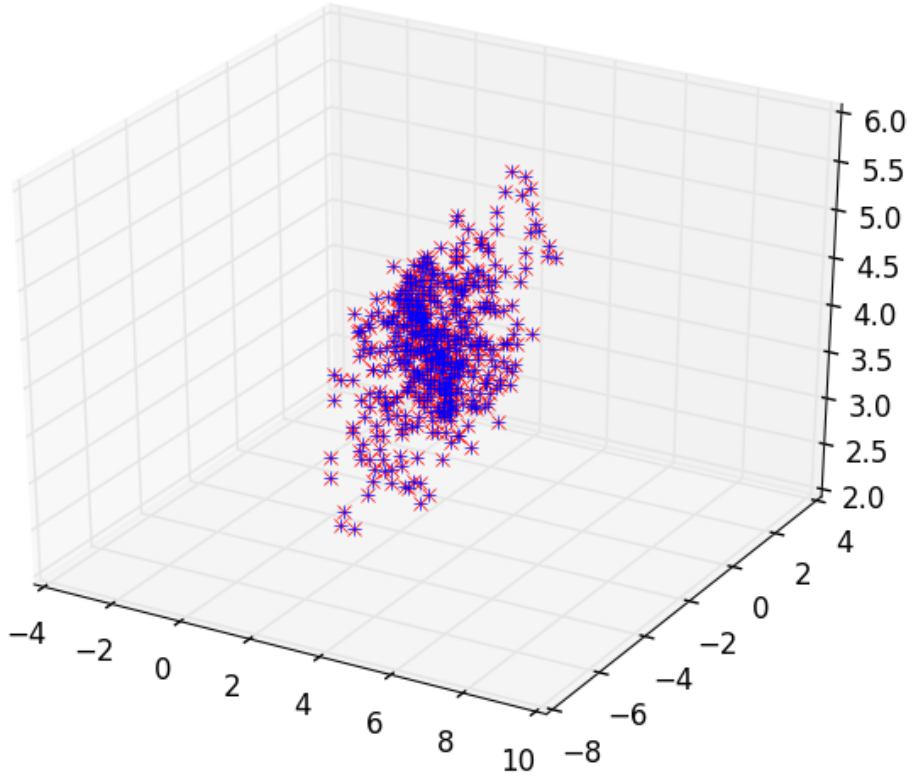



```
Out[16]: 1-element Array{Any,1}:
          PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c15e358>

In [17]: # Recovery of the two largest components
         Y2=T2*U[:,[3,2]]'.+μ
         plot3D(Y2[:,1],Y2[:,2],Y2[:,3],"rx" )
```




```
Out[17]: 1-element Array{Any,1}:
          PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c0b96a0>
```

```
In [18]: # Recovery of all three components (exact)
         Y3=T*U[:,[3,2,1]]'.+μ
         plot3D(Y3[:,1],Y3[:,2],Y3[:,3],"rx" )
         plot3D(X[:,1],X[:,2],X[:,3],"b+" )
```




```
Out[18]: 1-element Array{Any,1}:
          PyObject <mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f5c8c024dd8>

In [19]: # Fact 4 - PCA using SVD
         function myPCA{T}(X::Array{T}, k::Int)
             μ=mean(X,1)
             U,σ,V=svd(X.-μ)
             U[:,1:k]*diagm(σ[1:k])
         end

Out[19]: myPCA (generic function with 1 method)

In [20]: T1s=myPCA(X,1)
         [T1s T1]

Out[20]: 500x2 Array{Float64,2}:
          -1.17542    -1.17542
           1.20316     1.20316
          -0.137076   -0.137076
           0.770821    0.770821
           3.45968     3.45968
           0.0733431   0.0733431
          -4.18796    -4.18796
           0.242628    0.242628
```

```
 0.983668    0.983668
 0.454952    0.454952
 1.01151     1.01151
-2.00889    -2.00889
-5.67404    -5.67404
 ⋮
 0.726233    0.726233
 3.39719     3.39719
 0.0408711   0.0408711
-0.0257068  -0.0257068
 0.894956    0.894956
 0.343693    0.343693
 2.81052     2.81052
-0.0755973  -0.0755973
 4.58015     4.58015
-0.17815    -0.17815
 5.85217     5.85217
 1.46156     1.46156
```

```
In [21]: # The two largest components using SVD
         T2s=myPCA(X,2)
         [T2s T2]
```

```
Out[21]: 500x4 Array{Float64,2}:
 -1.17542    -0.235164   -1.17542     0.235164
  1.20316     0.457833    1.20316    -0.457833
 -0.137076   -0.0364299  -0.137076    0.0364299
  0.770821    1.15201     0.770821   -1.15201
  3.45968    -1.96267     3.45968     1.96267
  0.0733431   2.7905      0.0733431  -2.7905
 -4.18796    -0.0278544  -4.18796     0.0278544
  0.242628   -1.22416     0.242628    1.22416
  0.983668    0.473692    0.983668   -0.473692
  0.454952    0.101401    0.454952   -0.101401
  1.01151     1.33151     1.01151    -1.33151
 -2.00889     0.111321   -2.00889    -0.111321
 -5.67404     1.05624    -5.67404    -1.05624
  ⋮
  0.726233   -0.513798    0.726233    0.513798
  3.39719    -1.48404     3.39719     1.48404
  0.0408711   0.153192    0.0408711  -0.153192
 -0.0257068   0.0770764  -0.0257068  -0.0770764
  0.894956   -0.650127    0.894956    0.650127
  0.343693   -0.281339    0.343693    0.281339
  2.81052     0.124756    2.81052    -0.124756
 -0.0755973   1.24246    -0.0755973  -1.24246
  4.58015     1.97832     4.58015    -1.97832
 -0.17815    -0.227425   -0.17815     0.227425
  5.85217     0.131967    5.85217    -0.131967
  1.46156     0.0773186   1.46156    -0.0773186
```

8.5.2 Example - Real data

We will cluster three datasets from Workshop "Principal Manifolds-2006":

Data set	# of genes (m)	# of samples (n)
D1	17816	286
D2	3036	40
D3	10383	103

```
In [22]: # Data set D1
         f=readdlm("files/d1.txt")
```

```
Out[22]: 17817x287 Array{Any,2}:
          "CHIP"                          "GSM36777"  ...    "GSM37061"    "GSM37062"
          "1007_s_at"                      0.12              0            -0.43
          "1053_at"                        0.14              0.46         -0.2
          "117_at"                        -0.14             -0.8          -0.29
          "121_at"                        -0.11              0.21          0.24
          "1255_g_at"                     -0.73          ...   0.52          0.72
          "1294_at"                       -0.26             -0.13         -1.06
          "1316_at"                       -0.05             -0.55         -0.05
          "1405_i_at"                     -0.85             -2.92         -0.73
          "1431_at"                       -0.67              0.67          0.58
          "1438_at"                       -0.49          ...   0.87         -1.63
          "1487_at"                       -0.47             -0.11          0.32
          "1494_f_at"                      3.97             -0.61         -0.17
          ⋮                                                 ⋱    ⋮
          "AFFX-r2-Ec-bioC-3_at"          -0.59          ...   0.03         -0.27
          "AFFX-r2-Ec-bioC-5_at"          -0.75             -0.01         -0.44
          "AFFX-r2-Ec-bioD-3_at"          -0.59              0.01         -0.43
          "AFFX-r2-Ec-bioD-5_at"          -0.7               0.14         -0.39
          "AFFX-r2-Hs18SrRNA-3_s_at"       0.23             -1.98          0.23
          "AFFX-r2-Hs18SrRNA-5_at"         1.2           ...  -1.8           0.2
          "AFFX-r2-Hs18SrRNA-M_x_at"       0.57             -0.86          0.61
          "AFFX-r2-Hs28SrRNA-3_at"         1.01             -0.94          0.21
          "AFFX-r2-Hs28SrRNA-5_at"         0.47             -0.85          0.54
          "AFFX-r2-Hs28SrRNA-M_at"         0.63             -1.18          0.59
          "AFFX-r2-P1-cre-3_at"           -0.46          ...   0.09         -0.26
          "AFFX-r2-P1-cre-5_at"           -0.52              0.06         -0.28
```

```
In [23]: sizeof(f)
```

```
Out[23]: 40907832
```

```
In [24]: X=map(Float64,f[2:end,2:end])
```

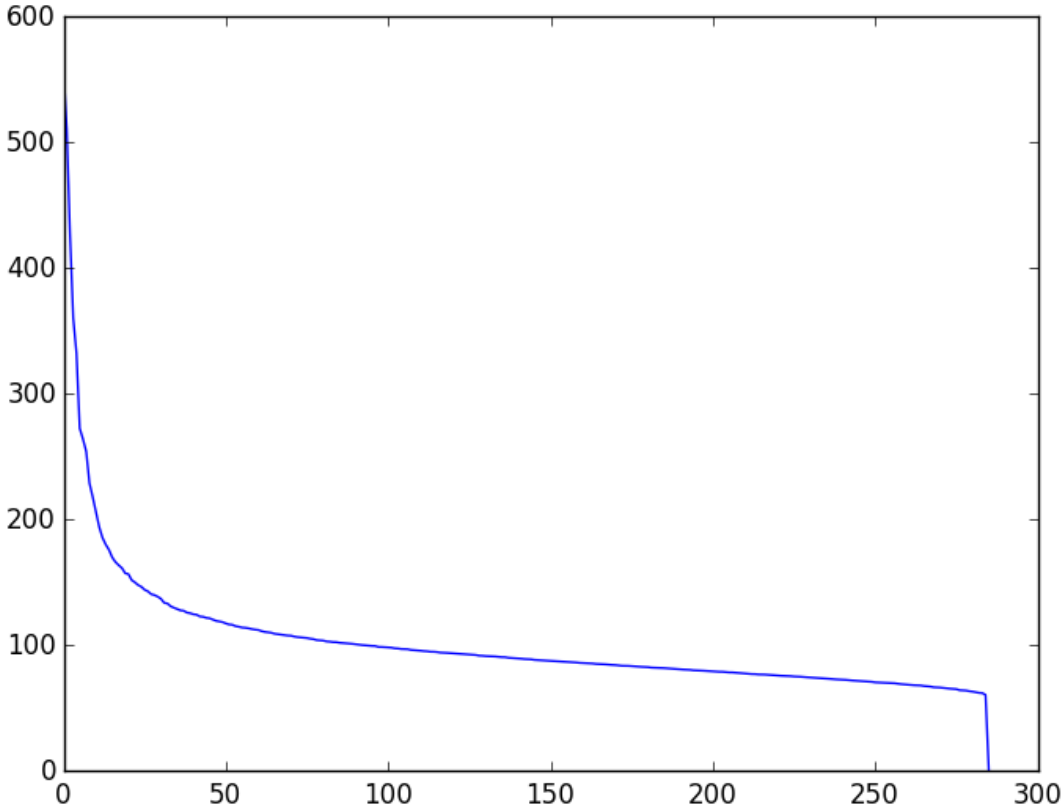
```
Out[24]: 17816x286 Array{Float64,2}:
           0.12   0.88   0.58   0.19   0.27  ...  -0.2   -0.35  -0.24   0.0   -0.43
           0.14  -0.88  -0.22   0.94   1.01       0.36   0.02  -0.37   0.46  -0.2
```

```
-0.14  -0.31   0.2    0.41   0.48      0.56  -0.05   0.1   -0.8   -0.29
-0.11   0.94   0.15   0.02  -0.06      0.05  -0.02   0.12   0.21   0.24
-0.73   1.46  -0.32   1.03   0.39     -0.17  -0.23   1.47   0.52   0.72
-0.26   0.28  -0.24  -0.21  -0.01  …   -0.18   0.38   0.01  -0.13  -1.06
-0.05   0.6    0.0    0.04   0.02     -0.41  -0.6    0.0   -0.55  -0.05
-0.85  -0.31  -2.45   0.69   2.71      0.05   0.75  -0.33  -2.92  -0.73
-0.67   0.63   0.11   0.19   0.28     -0.83  -0.01  -0.1    0.67   0.58
-0.49  -0.98  -0.92   2.08   1.26      0.16  -0.23  -0.19   0.87  -1.63
-0.47   0.08   0.13   0.09   0.67  …    0.0    0.14   0.5   -0.11   0.32
 3.97   0.52   4.93  -0.25   0.16     -0.54  -0.45   0.16  -0.61  -0.17
 0.11   0.37  -0.38  -0.23  -0.72      0.43   0.41  -0.49  -0.28  -0.3
 ⋮                                  ⋱                              ⋮
-0.59   0.36   0.24   0.54   0.22      0.29   0.04   0.44   0.03  -0.27
-0.75  -0.02   0.32   0.42   0.23  …   0.22  -0.09   0.62  -0.01  -0.44
-0.59   0.25   0.4   -0.06  -0.14      0.36   0.13   0.74   0.01  -0.43
-0.7    0.06   0.46  -0.07   0.03      0.39   0.1    0.64   0.14  -0.39
 0.23  -0.01   1.81   0.64  -1.27     -0.32  -0.57   0.1   -1.98   0.23
 1.2   -0.02   2.71   0.48  -4.13     -0.39  -1.04  -0.56  -1.8    0.2
 0.57   0.72   2.45   0.61  -2.12  …  -0.72  -0.21  -0.13  -0.86   0.61
 1.01  -0.4    2.13   0.98  -1.13     -0.18  -0.62   0.22  -0.94   0.21
 0.47   0.19   1.86   0.8   -0.7      -0.37   0.27  -0.07  -0.85   0.54
 0.63   0.75   1.48   0.91  -1.16      0.06  -0.37   0.32  -1.18   0.59
-0.46   0.25   0.42   0.45   0.33      0.34   0.13   0.6    0.09  -0.26
-0.52   0.15   0.47   0.35   0.15  …   0.23   0.09   0.67   0.06  -0.28
```

In [25]:
```
# Clear a big variable
f=[]
```

Out[25]:
```
0-element Array{Any,1}
```

In [26]:
```
# Fact 7 - Plot σ's and observe the knee
μ=mean(X,1)
σ=svdvals(X.-μ)
plot(σ)
```




```
Out[26]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x7f5c8c00a0f0>

In [27]: # PCA on X, keep 20 singular values
         k=20
         T=myPCA(X,k)

Out[27]: 17816x20 Array{Float64,2}:
          -2.60997     1.29427    3.82777   ...    0.416223   -0.908852    -0.339937
           2.53162     6.72032    1.25924        0.950754   -0.53449      0.0961831
           2.63868     2.01813    2.13187        1.72483     0.513846    -0.0914956
          -0.631409   -0.94409    1.88523        0.440639   -0.745012     0.0996248
          -0.8162      0.174124   3.59941        1.30466    -2.34368      0.369603
          -0.274186   -1.29081   -1.33486   ...  -0.473495   -0.62421     -1.26462
           0.401881   -0.738539   1.34847        0.39025    -0.00439161   0.0151878
          21.7269     -4.06805   -9.91123       -2.0272     -1.41633     -0.0263997
          -2.58371    -0.156413  -0.737911      -0.265203    1.58469      0.322162
           4.17899    17.0184     1.83096        2.91487    -0.433558     1.164
           0.772371    0.155414   1.66253   ...  -0.833175    0.397829    -0.700775
          -5.42643    -0.788577   1.64396       -1.44325     0.387156     0.840406
           0.500175   -4.16445   -3.51728        0.221502    0.00109187  -1.79048
           ⋮                                  ⋱
           0.602363   -1.58106    1.88832       -0.291308   -0.863038    -1.15118
           0.539476   -1.8133     2.05333   ...  -0.31821    -1.60131     -1.2452
```

```
 0.485803   -1.87352    2.20944      -0.164564   -0.654407    -1.2894
 0.475569   -1.44394    2.10787      -0.344536   -1.242       -1.30914
 5.56309    -2.43271    4.40758      -2.06542     0.563161    -1.10901
 5.82127     1.07777    2.81812      -2.12045    -0.50111     -0.404985
 4.46274    -2.24485    4.53101   ...  -1.62547    -0.859139    -0.507143
 4.32701     1.4561     3.72729      -1.62872    -0.24935     -0.80077
 3.20833    -3.0066     3.52118      -1.18471    -0.67259     -0.423825
 3.35449    -3.69571    2.98001      -1.24199     0.215023    -0.144022
-0.0676899  -1.15867    2.49808      -0.213136   -1.55336     -0.767101
 0.431614   -1.23256    2.16338   ...   0.0266245  -1.6841      -0.930856
```

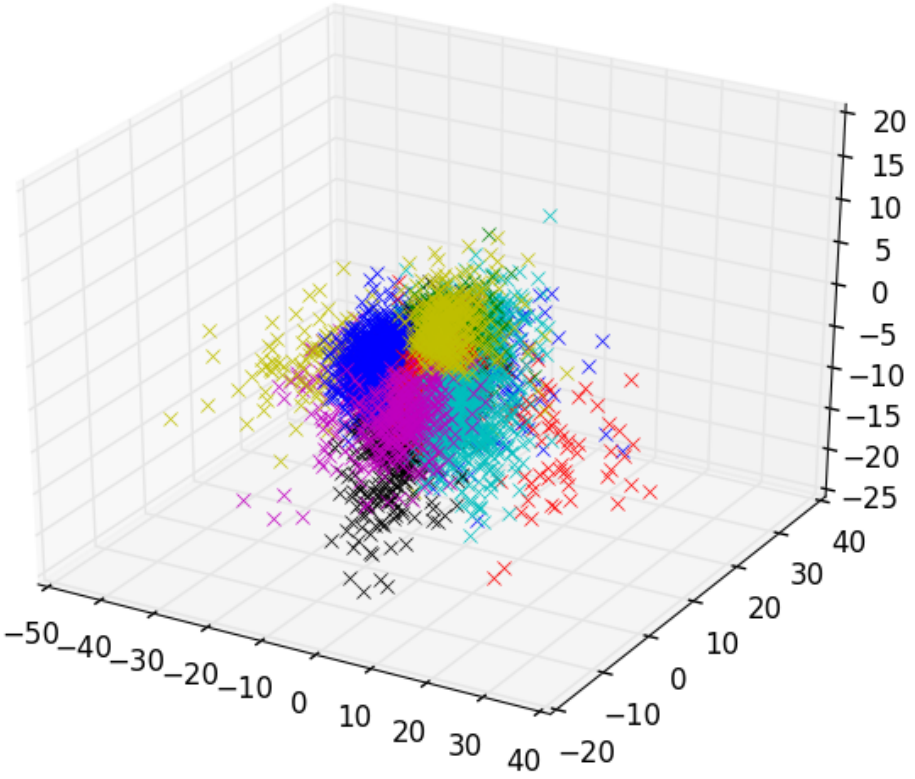
In [28]:
```
# Find k clusters
using Clustering
out=kmeans(T',k)
```

Out[28]:
```
Clustering.KmeansResult{Float64}(20x20 Array{Float64,2}:
 12.7714      0.190977    15.5244    -1.29565    ...  -0.00301044   5.75682
 -1.77035    -0.583934    10.5428    13.2541         -2.74219     -0.745595
  0.829446    1.31075    -10.903     -1.66399        -5.21303      5.3527
  1.90392    -1.27954     -6.5508     2.47194         2.7828       2.18507
 -6.58323    -0.18073      8.26394   -1.07976        -4.60841      2.00032
  2.96367    -0.982709     0.216595   0.681603   ...  -3.26543      0.579801
  0.470521    0.306466     0.653384  -0.104422       -3.5321      -0.826487
 -3.26094    -0.0474791   -4.1905     0.266698        3.6254       0.882696
 -1.32942    -0.982074     0.804532   0.164148        0.701501    -0.587529
 -1.03712    -0.067161     1.02203    0.124679        1.30368      0.946149
  0.897632    0.088324     5.64087    0.105575   ...   1.1923       -0.0464621
 -1.24827     0.238238    -4.21214    0.0949709      -0.593888    -0.00522879
 -0.603898   -0.0963908    5.6174     0.175332       -0.211983     0.105197
 -1.58803    -0.04454     -2.3625     0.29656        -0.310464     0.0577572
 -0.548418   -0.0476998   -1.93845   -0.276284       -0.344757     0.098977
 -0.0962598  -0.186169    -3.09544   -0.0401527  ...  -0.0153958    0.598045
  0.570255   -0.0642281   -2.14385   -0.128652       -0.345612     0.213371
 -0.79388     0.0786471    0.636688  -0.0158132       0.0301866    0.378122
  0.115908   -0.161616     3.21687   -0.483951        0.508225     0.131502
  0.403093   -0.129848    -0.674565  -0.0950792      -0.386024     0.849023   ,[8,16
```

In [29]:
```
# Plot points defined by the first three principal components
function myPlot(C::Vector, T::Array, k::Int)
    P=Array(Any,k)
    for j=1:k
        P[j]=T[C.==j,1:3]
        plot3D(P[j][:,1],P[j][:,2],P[j][:,3],"x")
    end
end
```

Out[29]:
```
myPlot (generic function with 1 method)
```

In [30]:
```
myPlot(out.assignments,T,k)
```

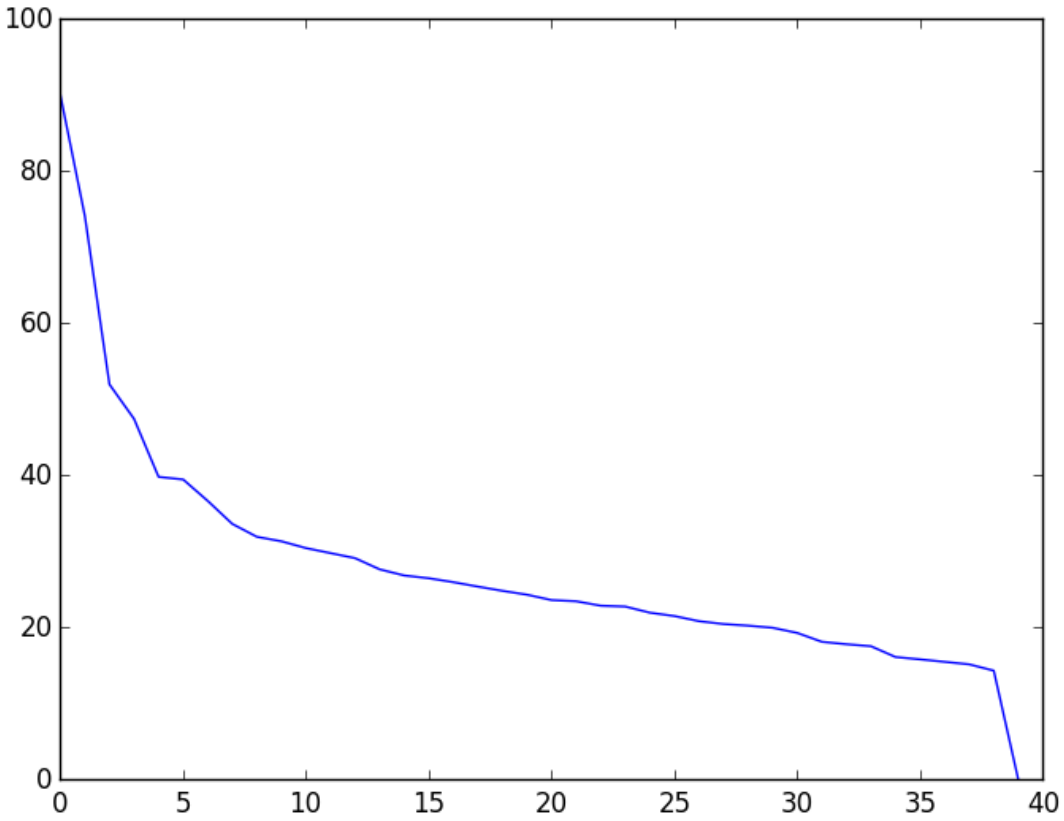



```
In [31]: # Data set D2
         X=readdlm("files/d2fn.txt")
```

```
Out[31]: 3036x40 Array{Float64,2}:
           0.0681   0.072    0.3944  -0.0606  ...  -0.3723  -0.2963  -0.619   -0.4171
           0.1402   0.2517   0.0843   0.1794      -0.1204   0.0086  -0.8177  -0.3646
           0.0081   0.3179   0.5802   0.0162      -0.2968  -0.3981  -1.055   -0.4583
           0.1049   0.3031   0.3048   0.2214      -0.288   -0.0841  -0.7231  -0.2951
           0.0609   0.4262   0.104    0.227       -0.3255  -0.1652  -0.6634  -0.4358
          -0.1163   0.2005  -0.2816   0.1192  ...  -0.2508  -0.034   -0.5112  -0.1964
           0.1276   0.4596   0.3751   0.2379      -0.4025  -0.3775  -0.6091  -0.4265
           0.1362   0.4215   0.0645   0.3017      -0.3727  -0.3352  -0.642   -0.4601
           0.1455   0.3676   0.1463   0.2997      -0.4893  -0.5417  -0.6408  -0.5108
          -0.7093   0.6045  -0.6372  -0.6594      -0.1644  -0.0441   0.4404   0.4663
          -0.0044   0.1012   0.1362   0.432   ...  -0.1594   0.4027  -0.5591  -0.1365
           0.1288   0.6568   0.7199   0.2173      -0.2006   0.0184  -1.1106   0.1015
           0.5014   0.6976   0.9516   0.6017      -1.168   -2.0287  -1.0887  -0.5792
           ⋮                                    ⋱
           0.2222   0.0015   0.0284  -0.3749       0.2817  -0.4384   0.2771   0.7985
           0.2587   0.1443  -0.1444   0.4272  ...  -0.4435  -0.6569  -0.6596  -0.4899
           0.3712   0.3939   0.4392   0.4054      -0.7037  -0.9145  -0.8822  -0.7753
           0.3355   1.0494   1.1719   0.5783      -0.7814  -1.0491  -0.7743   0.041
           0.3646   0.5563   0.375    0.3172      -0.5824  -0.7064  -0.8294  -0.8036
           0.4069   0.7489   0.1103   0.0303      -0.4834  -0.6479  -0.7976  -0.431
```

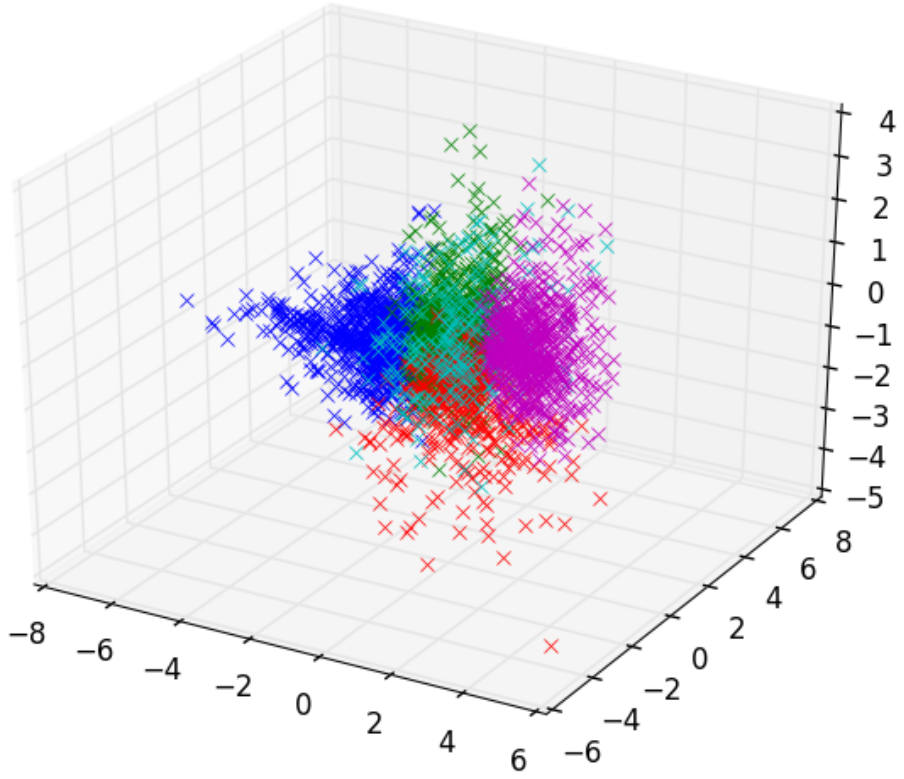
```
-0.5085   0.5377   0.6047  -0.5306  ...  -0.2162  -1.8589   0.2188   0.2223
 0.2777  -0.403   -0.0952   0.2562     -0.303    0.5055  -0.8516   0.2553
-0.0092  -0.1585   0.156   -0.6566      0.1016  -0.2363  -0.0053  -0.3698
 0.1618   0.0412  -0.3044   0.0847      0.5937  -0.0567   0.9136  -0.5945
 0.3653  -0.1373   0.0607   0.2313      0.3029   0.6607  -0.2687  -0.7253
 0.1646  -0.3546  -0.2455  -0.0452  ...  -0.2054   0.2651  -0.3522  -0.7788
```

In [32]:
```
μ=mean(X,1)
σ=svdvals(X.-μ)
plot(σ)
```



Out[32]:
```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x7f5c8bd7ab70>
```

In [33]:
```
k=5
T=myPCA(X,k)
out=kmeans(T',k)
myPlot(out.assignments,T,k)
```



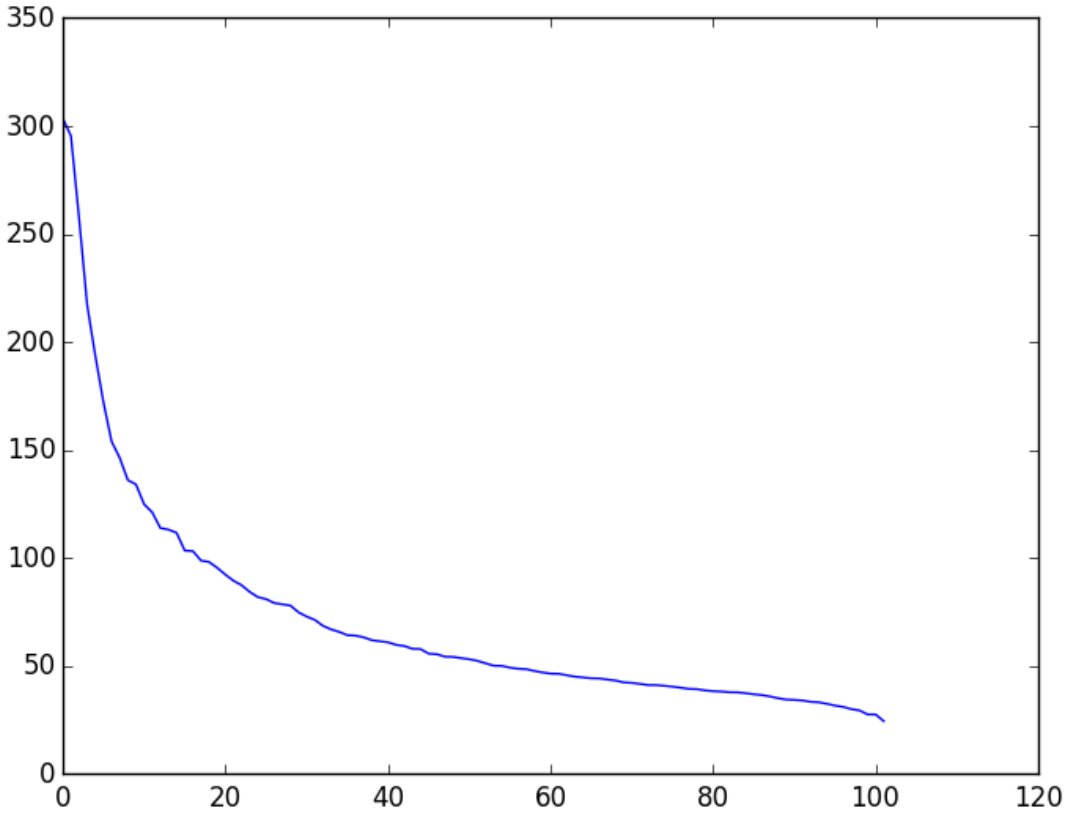

```
In [34]: # Data set D3 which has NULL values.
         X=readdlm("files/d3efn.txt")
```

```
Out[34]: 10387x102 Array{Any,2}:
           0.179  -0.201     1.289     0.039    ...  -0.961  -0.821    -0.551   1.279
           1.042   0.162     0.612     0.472        -0.788  -0.598    -0.078    "NULL"
          -0.022  -0.482      "NULL"  -0.182         0.688   0.758    -0.002   0.108
          -0.597   0.183     0.593    -0.467         0.253  -0.187    -0.707   0.393
           0.699   0.639     0.739     0.349        -0.711  -3.391     0.769   1.519
           1.035  -3.015    -2.255    -2.725    ...  -1.235  -3.185    -0.895   1.845
           0.64   -4.9      -4.24     -4.34         -1.04   -1.72     -1.41    2.62
           0.904  -2.636    -2.446    -1.996        -0.986  -0.986    -0.976   1.114
           0.74   -2.15       "NULL"  -3.43         -1.17   -0.95     -1.78    1.87
           0.771    "NULL"  -1.389    -1.249        -1.829    "NULL"  -1.049   0.391
           0.351  -0.829    -1.239    -0.679    ...  -1.139  -0.279    -0.789   0.441
           0.926  -1.034      "NULL"  -0.654         0.726   0.166    -0.104    "NULL"
          -0.216  -0.126     0.484     0.024        -1.206   0.074    -0.226   1.434
           ⋮                                    ⋱                     ⋮
          -0.995  -0.945      "NULL"    "NULL"  ...   0.205    "NULL"   0.295    "NULL"
          -2.323  -2.043    -1.213    -2.633         0.857  -0.743    -1.683    "NULL"
           1.511   0.681     1.601     0.961         1.141   0.401    -1.119   0.521
           1.007   0.197     1.057     0.547         1.287  -0.043    -0.213   0.547
           0.968   0.688     1.278     0.838         2.228  -0.102    -1.342  -1.232
           1.346   1.396     2.616     1.506    ...  -1.834   0.186    -0.004  -5.214
```

```
 -0.186   1.114     0.814     0.274        0.614  -0.266     0.204  -1.236
 -0.985  -1.715    -1.405    -1.605        0.855   6.205    -0.155  -1.565
 -0.135  -1.315     0.225    -2.025        0.275   0.055    -0.605  -1.735
  0.095  -0.845    -0.895      "NULL"      0.885   0.455     0.255  -0.695
  0.173  -0.057      "NULL"   0.243    ...   0.263   0.143     0.103   0.163
  0.826  -0.054    -1.134     0.436       -0.284  -0.244    -0.164  -0.164
```

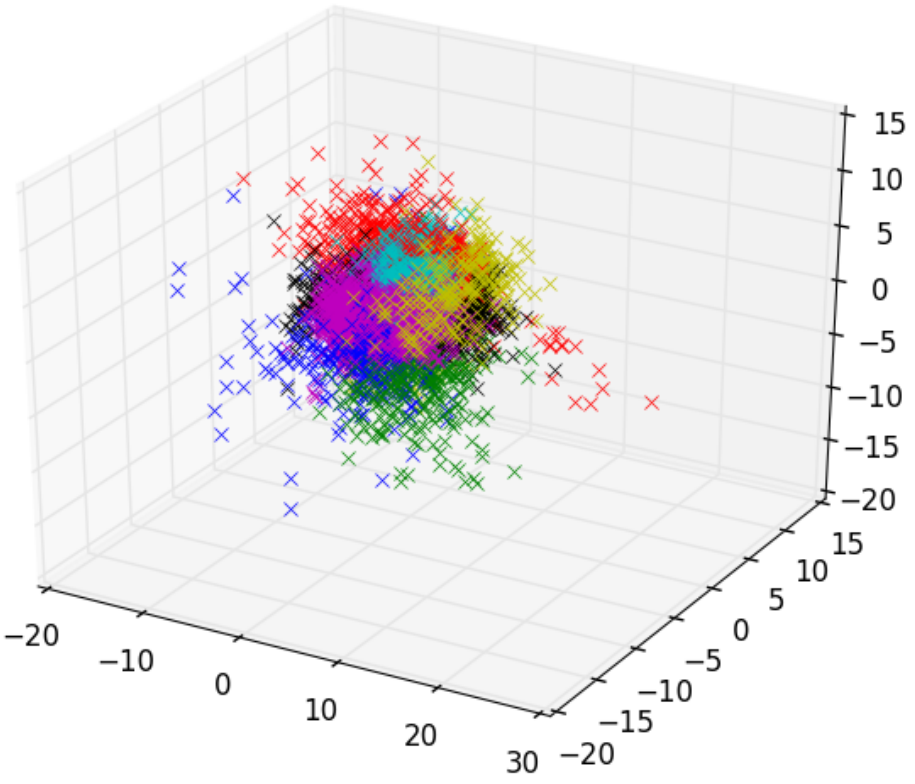
We use the package DataFrames.jl and replace the "NULL" string with NA.

```
In [35]: using DataFrames

In [36]: sf = readtable("files/d3efn.txt", header=false, skipblanks=true,separator =' ',nast

In [37]: # Set NA to zero (the choice is case-dependent).
         X=Array(Float64,size(sf))
         for i=1:size(sf,2)
             X[:,i]=convert(Array,sf[i],0.0)
         end

In [38]: μ=mean(X,1)
         σ=svdvals(X.-μ)
         plot(σ)
```

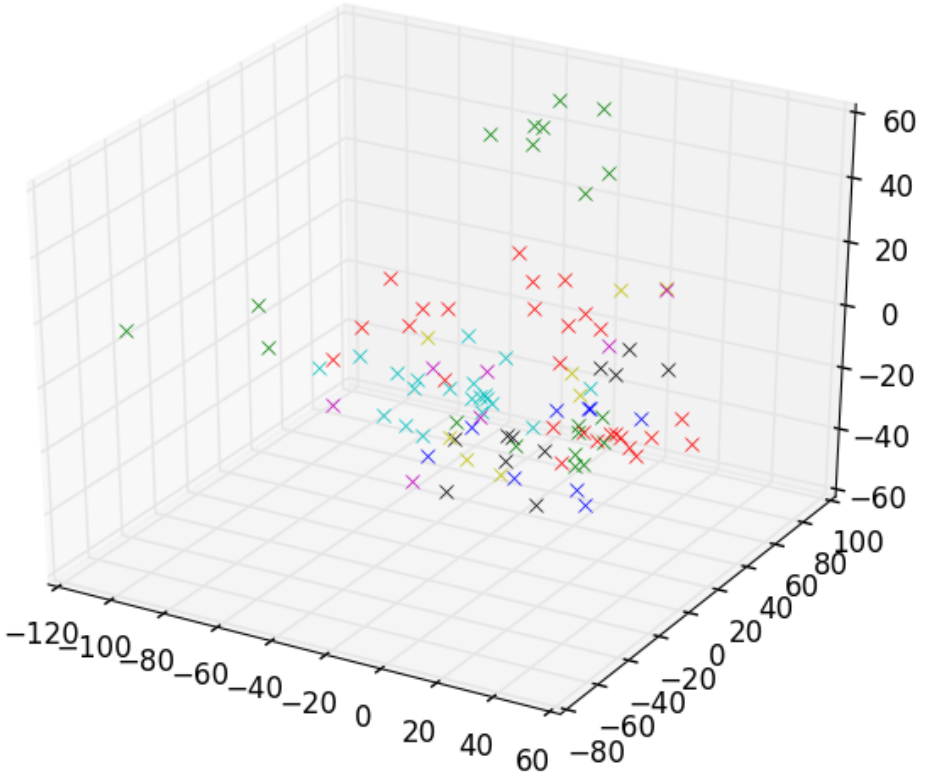



```
Out[38]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x7f5c8bdb4c18>
```

In [39]:
```
k=20
T=myPCA(X,k)
out=kmeans(T',k)
myPlot(out.assignments,T,k)
```



In [40]:
```
# Clustering the transpose
k=20
T=myPCA(X',k)
out=kmeans(T',k)
myPlot(out.assignments,T,k)
```


60
40
20
0
−20
−40
−60
−120
−100
−80
−60
−40
−20
0
20
40
60
−80
−60
−40
−20
0
20
40
60
80
100

In [ ]:

9 Tutorial 5 - Examples in Data Clustering

9.1 Assignment 1

Write a function for recursive graph k-partitioning according to the algorithm at the end of the Spectral Graph Bipartitioning notebook.
Use this function and `kmeans()` to cluster your favourite data sets.

9.2 Assignment 2

Use the function from Assiggnment 1 and `kmeans()` to partition image of a word into letters.

In [ ]:

10 Tutorial 6 - Examples in Document Clustering

10.1 Assignment 1

Cluster terms and documents in your favorite document collection or book: 1. Download the collection. 2. Form the *term* × *document* matrix. Remove stop words, if necessary, and apply stemming, if possible. 3. Normalize the weights, if necessary. 4. Form the matrices A and A_n. 5. Cluster documents (and terms) in k clusters using spectral k-partitioning of bipartite graphs. 6. Comment the solution.

In [ ]:

11 Tutorial 7 - Examples in Sparse + Low-Rank Splitting

11.1 Assignment 1

In a given video, separate the background (low-rank component) from the moving objects (sparse component): 1. Split the video-clip into frames. 2. Convert frames to vectors. 3. Stack vectors to a matrix A. 4. Compute the sparse + low-rank splitting of A. 5. Reconstruct clips of background and moving objects.

Use the video file from the ViSOR repository (see the original link).

Use the package VideoIO for video manipulation.

In [ ]:

12 Tutorial 8 - Examples in Signal Decomposition

12.1 Assignment 1

Write a function `monocomponents()` which decomposes signal with the following algorithm:

1. Choose τ and form the Hankel matrix H
2. Compute the EVD of H
3. Choose the significant eigenpairs of H
4. For each significant eigenpair (λ, u)
   (a) Form the rank one matrix $M = \lambda u u^T$
   (b) Define a new signal y consisting of the elements in the first row and the last column of M
   (c) Form the Hankel matrix $H(y)$
   (d) Compute the EVD of $H(y)$
   (e) Choose the significant eigenpairs of $H(y)$
   (f) **If** $H(y)$ has only two significant eigenpairs, declare y a mono-component, **else** go to step 4.

Test the algorithm on the note A4.

12.2 Assignment 2

Decompose your favorite music part, or the first bar of Beethoven's Piano Sonata No. 8 Op. 13.

13 Tutorial 9 - Examples in Compressive Sensing

13.1 Assignment 1*

Solution to a sudoku puzzle can be formulated as an underdetermined system of linear equations. Implement the approach described in P. Babu, K. Pelckmans, P. Stoica and J. Li, Linear Systems, Sparse Solutions, and Sudoku.

13.2 Assignment 2**

Write a Julia implementation of the code for compression of images referenced in J. Romberg, Imaging via Compressive Sampling.

```
In [ ]:
```

14 Tutorial 10 - Examples in Principal Component Analysis

14.1 Assignment 1

Analyse a data set of your choice with PCA and explain the results.