MATH2071: LAB 9: The Singular Value Decomposition

1 Introduction

Suppose A is an $m \times n$ matrix. Then there exist two unitary (or orthogonal in the case A is real) matrices: an $m \times m$ matrix U and a $n \times n$ matrix V, and a $m \times n$ "diagonal" matrix S with non-negative numbers on its diagonal sorted by size and zeros elsewhere, so that

$$A = USV^H, \tag{1}$$

where the $\cdot^H$ denotes the Hermitian (or conjugate transpose) of a matrix, and the diagonal entries of S are $S_{kk} = \sigma_k \geq 0$, for $k = 1 \ldots \min(m, n)$ and all the rest zero. The triple of matrices (U, S, V) is called the "singular value decomposition" (SVD) and the diagonal entries of S are called the "singular values" of A. The columns of U and V are called the left and right "singular vectors" of A respectively. You can get more information from a very nice Wikipedia article at `http://en.wikipedia.org/wiki/Singular_value_decomposition`. You can also find a very clear description of Beltrami's original proof of the existence of the SVD in a simple case beginning in Section 2 (p. 5) of G. W. Stewart's U. of Maryland report TR-92-31 (1992) at `http://purl.umn.edu/1868`.

In this lab, you will see some applications of the SVD, including

- The right singular vectors corresponding to vanishing singular values of A span the nullspace of A,
- The right singular vectors corresponding to positive singular values of A span the domain of A.
- The left singular vectors corresponding to positive singular values of A span the range of A.
- The rank of A is the number of positive singular values of A.
- The singular values characterize the "relative importance" of some basis vectors in the domain and range spaces over others. This fact can be used as a compression algorithm for images.
- The Moore-Penrose "pseudo-inverse" of A is computed from the SVD, making it possible to solve the system $Ax = b$ in the least-squares sense.
- "Latent Semantic Indexing" uses SVD to index and sort natural language documents according to their content.

Numerical methods for finding the singular value decomposition will also be addressed in this lab. One "obvious" algorithm involves finding the eigenvalues of $A^H A$, but this is not really practical because of roundoff difficulties caused by squaring the condition number of A. Matlab includes a function called `svd` with signature `[U S V]=svd(A)` to compute the singular value decomposition and we will be using it, too.

This function uses the Lapack subroutine `dgesvd`, so if you were to need it in a Fortran or C program, it would be available by linking against the Lapack library.

You may be interested in a six-minute video made by Cleve Moler in 1976 at what was then known as the Los Alamos Scientific Laboratory. Most of the film is computer animated graphics. Moler recently retrieved the film from storage, digitized it, and uploaded it to YouTube at `http://youtu.be/R9UoFyqJca8`. You may also be interested in a description of the making of the film, and its use as background graphics in the first Star Trek movie, is in his blog at
`http://blogs.mathworks.com/cleve/2012/12/10/1976-matrix-singular-value-decomposition-film/`

This lab will take three sessions. You may find it convenient to print the pdf version of this lab rather than the web page itself.

2 The singular value decomposition

The matrix S in (1) is $m \times n$ and is of the form

$$S_{k\ell} = \begin{cases} \sigma_k & \text{for } k = \ell \\ 0 & \text{for } k \neq \ell \end{cases}$$

and $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_r > 0 = 0 = \cdots = 0$. Since U and V are unitary (and hence nonsingular), it is easy to see that the number r is the rank of the matrix A and is necessarily no larger than $\min(m, n)$. The "singular values," σ_k, are real and positive and are the eigenvalues of the Hermitian matrix $A^H A$.

Supposing that r is the largest integer so that $\sigma_r > 0$, then the SVD implies that the rank of the matrix is r. Similarly, if the matrix A is $n \times n$, then the rank of the nullspace of A is $n - r$. The first r columns of U are an orthonormal basis of the range of A, and the last $n - r$ columns of V are an orthonormal basis for the nullspace of A. Of course, in the presence of roundoff some judgement must be made as to what constitutes a zero singular value, but the SVD is the best (if expensive) way to discover the rank and nullity of a matrix. This is especially true when there is a large gap between the "smallest of the large singular values" and the "largest of the small singular values."

The SVD can also be used to solve a matrix system. Assuming that the matrix is non-singular, all singular values are strictly positive, and the SVD can be used to solve a system.

$$\begin{aligned} b &= Ax \\ b &= USV^H x \\ U^H b &= SV^H x \\ S^+ U^H b &= V^H x \\ V S^+ U^H b &= x \end{aligned} \tag{2}$$

Where S^+ is the diagonal matrix whose diagonal entries are $1/\sigma_k$ for $\sigma_k > 0$ and zero otherwise. It turns out that Equation (2) is too expensive to be a good way to solve nonsingular systems, but is an excellent way to "solve" systems that are singular or almost singular! To this end, the matrix S^+ is the matrix with the same shape as S^T and whose elements are given as

$$S^+_{k\ell} = \begin{cases} 1/\sigma_k & \text{for } k = \ell \text{ and } \sigma_k > 0 \\ 0 & \text{otherwise} \end{cases}$$

Further, if A is close to singular, a similar definition but with diagonal entries $1/\sigma_k$ for $\sigma_k > \epsilon$ for some ϵ can work very nicely. In these latter cases, the "solution" is the least-squares best fit solution and the matrix $A^+ = VS^+U^H$ is called the Moore-Penrose pseudo-inverse of A.

Exercise 1: In this exercise you will use the Matlab `svd` function to solve for the best fit linear function of several variables through a set of points. This is an example of "solving" a rectangular system.

Imagine that you have been given many "samples" of related data involving several variables and you wish to find a linear relationship among the variables that approximates the given data in the best-fit sense. This can be written as a rectangular system

$$\begin{array}{ccccccccccc} a_1 d_{1,1} & + & a_2 d_{1,2} & + & a_3 d_{1,3} & + & \dots & + & d_{1,n+1} & = 0 \\ a_1 d_{2,1} & + & a_2 d_{2,2} & + & a_3 d_{2,3} & + & \dots & + & d_{2,n+1} & = 0 \\ a_1 d_{3,1} & + & a_2 d_{3,2} & + & a_3 d_{3,3} & + & \dots & + & d_{3,n+1} & = 0 \\ \vdots & & \vdots & & \vdots & & \dots & & \vdots & \end{array} \tag{3}$$

where d_{ij} represents the data and a_i the desired coefficients.

The system (3) can be written in matrix form. Recall that the values a_j are the unknowns. For a moment, denote the vector $(\mathbf{a})_j = a_j$, the matrix $(D)_{ij} = d_{ij}$ and the vector $(\mathbf{b})_i = -d_{i,n+1}$, all for $i = 1, \dots, M$ and $j = 1, \dots, N$, where $M \geq N$. With this notation, (3) can be written as the matrix equation $D\mathbf{a} = \mathbf{b}$, where D is (usually) a rectangular matrix. A "least-squares" solution of $D\mathbf{a} = \mathbf{b}$ is a vector $\mathbf{a}$ satisfying

$$\|D\mathbf{a} - \mathbf{b}\|^2 = \min_{\mathbf{x}} \|D\mathbf{x} - \mathbf{b}\|^2,$$

and, since there could, in general, be many such vectors, we require that $\mathbf{a}$ have the smallest norm of all such vectors. It turns out that the vector $\mathbf{a}$ is determined as $D^+\mathbf{b}$.

To see why this is so, suppose that the rank of D is r, so that there are precisely r positive singular values of D. Then $D = USV^T$ and

$$\begin{aligned} \|D - \mathbf{b}\|^2 &= \|USV^T\mathbf{x} - \mathbf{b}\|^2 \\ &= \|SV^T\mathbf{x} - U^T\mathbf{b}\|^2 \\ &= \|S\mathbf{z} - U^T\mathbf{b}\|^2, \quad \text{where } \mathbf{z} = V^T\mathbf{x} \\ &= \sum_{j=1}^{r} (\sigma_j z_j - (U^T\mathbf{b})_j)^2 + \sum_{j=r+1}^{N} ((U^T\mathbf{b})_j)^2 \end{aligned}$$

because S is a matrix with diagonal entries $S_{ii} = \sigma_i$, $i = 1, \dots, r$, with all other entries being zero. Clearly, the sum is minimized when the first term is zero, $z_j = (U^T\mathbf{b})_j/\sigma)j$. The other components, z_j for $j = r+1, \dots, N$ are arbitrary, but $\|\mathbf{z}\|$ will be minimized when they are taken to be zero. Thus, the minimum norm solution for the coefficients $\mathbf{a}$ is given by $\mathbf{a} = V\mathbf{z} = VS^+U^T\mathbf{b} = D^+\mathbf{b}$.

As we have done several times before, we will first generate a data set with known solution and then use `svd` to recover the known solution. Place Matlab code for the following steps into a script m-file called `exer1.m`

(a) Generate a data set consisting of twenty "samples" of each of four variables using the following Matlab code.

```
N=20;
d1=rand(N,1);
d2=rand(N,1);
d3=rand(N,1);
d4=4*d1-3*d2+2*d3-1;
```

It should be obvious that these vectors satisfy the equation

$$4d_1 - 3d_2 + 2d_3 - d_4 = 1,$$

but, if you were solving a real problem, you would not be aware of this equation, and it would be your objective to discover it, *i.e.,* to find the coefficients in the equation.

(b) Introduce small "errors" into the data. The `rand` function produces numbers between 0 and 1.

```
EPSILON=1.e-5;
d1=d1.*(1+EPSILON*rand(N,1));
d2=d2.*(1+EPSILON*rand(N,1));
d3=d3.*(1+EPSILON*rand(N,1));
d4=d4.*(1+EPSILON*rand(N,1));
```

You have now constructed the "data" for the following least-squares calculation.

(c) Imagine the four vectors `d1,d2,d3,d4` as data given to you, and construct the matrix consisting of the four column vectors, `A=[d1,d2,d3,d4]`.

(d) Use the Matlab `svd` function

```
[U S V]=svd(A);
```

Please include the four non-zero values of `S` in your summary, but not the matrices `U` and `V`.

(e) Just to confirm that you have everything right, compute `norm(A-U*S*V','fro')`. This number should be roundoff-sized.

(f) Construct the matrix S^+ (call it `Splus`) so that

$$S^+_{k\ell} = \begin{cases} 1/S(k,k) & \text{for } k = \ell \text{ and } S(k,k) > 0 \\ 0 & \text{for } k \neq \ell \text{ or } S(k,k) = 0 \end{cases}$$

(g) We are seeking the coefficient vector x that

$$x_1 d_1 + x_2 d_2 + x_3 d_3 + x_4 d_4 = 1.$$

Find this vector by setting `b=ones(N,1)` (the coefficients in Equation (3) have been moved to the right and are $-d_{k5} = 1$) using Equation (2) above. Include your solution with your summary. It should be close to the known solution `x=[4;-3;2;-1]`.

Sometimes the data you are given turns out to be deficient because the variables supposed to be independent are actually related. This dependency will cause the coefficient matrix to be singular or nearly singular. When the matrix is singular, the system of equations is actually redundant, and one equation can be eliminated. This yields fewer equations than unknowns and any member of an affine subspace can be legitimately regarded as "the" solution.

Dealing with dependencies of this sort is one of the greatest difficulties in using least-squares fitting methods because it is hard to know which of the equations is the best one to elminate. The singular value decomposition is the best way to deal with dependencies. In the following exercise you will construct a deficient set of data and see how to use the singular value decomposition to find the solution.

Exercise 2:

(a) Copy your m-file `exer1.m` to `exer2.m`. Replace the line

```
d3=rand(N,1);
```

with the line

```
d3=d1+d2;
```

This makes the data deficient and introduces nonuniqueness into the solution for the coefficients.

(b) After using `svd` to find `U, S`, and `V`, print `S(1,1), S(2,2), S(3,3), S(4,4)`. You should see that `S(4,4)` is substantially smaller than the others. Set `S(4,4)` to zero, construct S^+ according to

$$S^+_{k\ell} = \begin{cases} 1/S(k,k) & \text{for } k = \ell \text{ and } S(k,k) > 0 \\ 0 & \text{for } k \neq \ell \text{ or } S(k,k) = 0 \end{cases}$$

and find the coefficient vector `x` as in (2). The solution you found is probably not `x=[4;-3;2;-1]`.

(c) Look at `V(:,4)`. This is the vector associated with the singular value that you set to zero. Since the original system is rank deficient, there must be a non-trivial nullspace. Any solution can be represented as the sum of some particular solution and a vector from the nullspace and the vector `V(:,4)` spans the nullspace. `V(:,4)` is associated with the extra relationship `d1+d2-d3=0` that was introduced in the first step. What multiple of `V(:,4)` can be added to your solution to yield `[4;-3;2;-1]`?

Note that the singular value decomposition allows you to discover deficiencies in the data without knowing they are there (or even if there are any) in the first place. This idea can be the basis of a fail-safe method for computing least-squares coefficients.

In the following exercise you will see how the singular value decomposition can be used to "compress" a graphical figure by representing the figure as a matrix and then using the singular value decomposition to find the closest matrix of lower rank to the original. This approach can form the basis of efficient compression methods.

Exercise 3:

(a) This NASA photo (`TarantulaNebula.jpg`) comes from the Hubble telescope and presents a dramatic picture of this extra-galactic formation. Download it.

(b) Matlab provides various image processing utilities. In this case, read the image in using the following command.

```
nasacolor=imread('TarantulaNebula.jpg');
```

The variable `nasacolor` will be a $567 \times 630 \times 3$ matrix of integer values between 0 and 255.

(c) Display the color plot using the command

```
image(nasacolor)
```

(d) The third subscript of the array `nasa` refers to the red, green, and blue color components. To simplify this exercise, turn it into a greyscale with ordinary double precision values 0-255 using the following commands:

```
nasa=sum(nasacolor,3,'double');  %sum up red+green+blue
m=max(max(nasa));                %find the max value
nasa=nasa*255/m;                 %make this be bright white
```

The result from these commands is that `nasa` is an ordinary 567×630 array of double precision numbers.
Remark: This is a cheap but dirty way to create a grayscale image from an rgb image. It is good enough for our purpose here.

(e) Matlab has the notion of a "colormap" that determines how the values in the matrix will be colored. Use the command

```
colormap(gray(256));
```

to set a grayscale colormap. Now you can show the picture with the command

```
image(nasa)
title('Grayscale NASA photo');
```

Please do not send me this plot.

(f) Use the command `[U S V]=svd(nasa);` to perform the singular value decomposition. Do not include these matrices with your summary!

(g) Plot the singular values on a semilog scale: `semilogy(diag(S))`. You should observe that the values drop off very rapidly to less than 2% of the maximum in fewer than 50 values. Please send me this plot with your summary.

(h) Construct the three new matrices

```
nasa100=U(:,1:100)*S(1:100,1:100)*V(:,1:100)';
nasa50=U(:,1:50)*S(1:50,1:50)*V(:,1:50)';
nasa25=U(:,1:25)*S(1:25,1:25)*V(:,1:25)';
```

These matrices are of lower rank than the `nasa` matrix, and can be stored in a more efficient manner. (The `nasa` matrix is 567×630 and requires 357,210 numbers to store it. In contrast, nasa50 requires subsets of U and V that are 567×50 and the diagonal of S, for a total of 56,750. This is better than four to one savings in space.) Do not include these matrices with your summary!

(i) Plot the three matrices `nasa100`, `nasa50` and `nasa25` as images, including descriptive titles. You should only very slight differences between the original and the one with 100 singular values, some noticable differences with 50 singular values while you should see serious degradation of the image in the case of 25 singular values. Please include the 25 singular value case with your summary.

3 Two numerical algorithms

We now turn to the question of how to compute the SVD.

The easiest algorithm for SVD is to note the relationship between it and the eigenvalue decomposition: singular values are the square roots of the eigenvalues of $A^H A$ or AA^H. Indeed, if $A = USV^H$, then

$$A^H A = (VSU^H)(USV^H) = V(S^2)V^H, \text{ and}$$
$$AA^H = (USV^H)(VSU^H) = U(S^2)U^H.$$

So, if you can solve for eigenvalues and eigenvectors, you can find the SVD.

Unfortunately, this is not a good algorithm because forming the product $A^H A$ roughly squares the condition number, so that the eigenvalue solution is not likely to be accurate. Of course, it will work fine for small matrices with small condition numbers and you can find this algorithm presented in many web pages. Don't believe everything you see on the internet.

A more practical algorithm is a Jacobi algorithm that is given in a 1989 report by James Demmel and Krešimir Veselić, that can be found at `http://www.netlib.org/lapack/lawnspdf/lawn15.pdf`. The algorithm is a one-sided Jacobi iterative algorithm that appears as Algorithm 4.1, p32, of that report. This algorithm amounts to the Jacobi algorithm for finding eigenvalues of a symmetric matrix. (See, for example, J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965, or G. H. Golub, C. F. Van Loan, *Matrix Computations* Johns Hopkins University Press, 3rd Ed, 1996.)

The algorithm *implicitly* computes the product AA^T and then uses a sequence of "Jacobi rotations" to diagonalize it. A Jacobi rotation is a 2×2 matrix rotation that annihilates the off-diagonal term of a symmetric 2×2 matrix. You can find more in a nice Wikipedia article located at `http://en.wikipedia.org/wiki/Jacobi_rotation`.

Given a 2×2 matrix M with

$$M^T M = \begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix}$$

It is possible to find a "rotation by angle θ matrix"

$$\Theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

with the property that $\Theta^T M^T M \Theta = D$ is a diagonal matrix. Θ can be found by multiplying out the matrix product, expressing the off-diagonal matrix component in terms of $t = \tan\theta$, and setting it to zero to arrive at an equation

$$t^2 + 2\zeta t - 1 = 0$$

where $\zeta = (\beta - \alpha)/(2\gamma)$. The quadratic formula gives $t = \tan\theta$, and then $\sin\theta$ and $\cos\theta$ can be recovered. There is no need to recover θ itself.

The following algorithm repeatedly passes down the diagonal of the implicitly-constructed matrix AA^T, choosing pairs of indices $k < j$, constructing the 2×2 submatrix from the intersection of rows and columns, and using Jacobi rotations to annhilate the off-diagonal term. Unfortunately, the Jacobi rotation for the pair $k = 2$, $j = 10$ messes up the rotation from $k = 1$, $j = 10$, so the process must be repeated until convergence. At convergence, the matrix S of the SVD has been implicitly generated, and the right and left singular vectors are recovered by multiplying all the Jacobi rotations together. The following algorithm carries out this process.

Note: For the purpose of this lab, that matrix A will be assumed to be square. Rectangular matrices introduce too much algorithmic complexity.

Algorithm

Given a convergence criterion ϵ, a *square* matrix A, a matrix U, that starts out as $U = A$ and ends up as a matrix whose columns are left singular vectors, and another matrix V that starts out as the identity matrix and ends up as a matrix whose columns are right singular vectors.

repeat

for all pairs $k < j$

($compute$ $\begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix} \equiv$ *the* (k, j) *submatrix of* $U^T U$)

$\alpha = \sum_{\ell=1}^{n} U_{\ell k}^2$

$\beta = \sum_{\ell=1}^{n} U_{\ell j}^2$

$\gamma = \sum_{\ell=1}^{n} U_{\ell k} U_{\ell j}$

(*compute the Jacobi rotation that diagonalizes* $\begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix}$)

$\zeta = (\beta - \alpha)/(2\gamma)$

$t = \text{signum}(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$

$c = 1/\sqrt{1 + t^2}$

$s = ct$

(*update columns* k *and* j *of* U)

for $\ell = 1$ **to** n

$T = U_{\ell k}$

$U_{\ell k} = cT - sU_{\ell j}$

$U_{\ell j} = sT + cU_{\ell j}$

endfor

(*update the matrix* V *of right singular vectors*)

for $\ell = 1$ **to** n

$T = V_{\ell k}$

$V_{\ell k} = cT - sV_{\ell j}$

$V_{\ell j} = sT + cV_{\ell j}$

endfor

endfor

until all $|\gamma|/\sqrt{\alpha\beta} \leq \epsilon$

The computed singular values are the norms of the columns of the final U and the computed left singular vectors are the normalized columns of the final U. As mentioned above, the columns of V are the computed right singular vectors.

Exercise 4:

(a) Copy the following code skeleton to a function m-file named `jacobi_svd.m`. Complete the code so that it implements the above algorithm.

```
function [U,S,V]=jacobi_svd(A)
% [U S V]=jacobi_svd(A)
% A is original square matrix
% Singular values come back in S (diag matrix)
% orig matrix = U*S*V'
%
% One-sided Jacobi algorithm for SVD
% lawn15: Demmel, Veselic, 1989,
% Algorithm 4.1, p. 32

TOL=1.e-8;
MAX_STEPS=40;

n=size(A,1);
U=A;
V=eye(n);
for steps=1:MAX_STEPS
  converge=0;
  for j=2:n
    for k=1:j-1
      % compute [alpha gamma;gamma beta]=(k,j) submatrix of U'*U
      alpha=???  %might be more than 1 line
      beta=???   %might be more than 1 line
      gamma=???  %might be more than 1 line
      converge=max(converge,abs(gamma)/sqrt(alpha*beta));

      % compute Jacobi rotation that diagonalizes
      %    [alpha gamma;gamma beta]
      if gamma ~= 0
        zeta=(beta-alpha)/(2*gamma);
        t=sign(zeta)/(abs(zeta)+sqrt(1+zeta^2));
      else
        % if gamma=0, then zeta=infinity and t=0
```

```
        t=0;
      end
      c=???
      s=???

      % update columns k and j of U
      T=U(:,k);
      U(:,k)=c*T-s*U(:,j);
      U(:,j)=s*T+c*U(:,j);

      % update matrix V of right singular vectors

      ???

    end
  end
  if converge < TOL
    break;
  end
end
if steps >= MAX_STEPS
  error('jacobi_svd failed to converge!');
end

% the singular values are the norms of the columns of U
% the left singular vectors are the normalized columns of U
for j=1:n
  singvals(j)=norm(U(:,j));
  U(:,j)=U(:,j)/singvals(j);
end
S=diag(singvals);
```

(b) Apply your version of `jacobi_svd` to the 2×2 matrix `A=U*S*V'` generated from the following three matrices.

```
U=[0.6  0.8
   0.8 -0.6];
V=sqrt(2)/2*[1  1
             1 -1];
S=diag([5 4]);
```

It is easy to see that `U` and `V` are orthogonal matrices, so that the matrices `U`, `S` and `V` comprise the SVD of `A`. You may get a "division by zero" error, but this is harmless because it comes from `gamma` being zero, which will cause `zeta` to be infinite and `t` to be zero anyhow.

Your algorithm should essentially reproduce the matrices `U`, `S` and `V`. You might find that the diagonal entries of `S` are not in order, so that the `U` and `V` are similarly permuted, or you might observe that certain columns of `U` or `V` have been multiplied by (-1). Be sure, however, that an *even* number of factors of (-1) have been introduced.

If you do not get the right answers, you can debug your code in the following way. First, note that there is only a single term, `k=1` and `j=2` in the double for loop.

i. Bring up your code in the editor and click on the dash to the left of the the last line of code, the one that starts off "`V(:,j)=`". This will cause a red dot to appear, indicating a

"breakpoint." (If you are not using the Matlab desktop, you can accomplish the same sort of thing using the `dbstop` command, placed just after the statement `V(:,j)=`.)

ii. Now, try running the code once again. You will find the code has stopped at the breakpoint.

iii. Take out a piece of paper and calculate the value of `alpha`. There are only two terms in this sum. Did your code value agree with your hand calculation? If not, double-check *both* calculations and fix the one that is in error.

iv. Do the same for `beta` and `gamma`.

v. Similarly, check the values of `s` and `c`.

vi. Press the "Step" button to complete the calculation of `V`. Now multiply (you can do this at the Matlab command prompt) `U*V'`. Do you get `A` back? It can be shown that the algorithm *always* maintains the condition `A=U*V'` when it is correctly programmed. If not, you probably have something wrong in the two statements defining `V`, or, perhaps, you mis-copied the code updating `U` from the web page. Find the error. **Remark:** At this point in the iteration, `U` and `V` have not yet converged to their final values!

(c) Use your `jacobi_svd` to find the SVD, `U1`, `S1` and `V1` of the matrix

```
A1 = [ 1  3  2
       5  6  4
       7  8  9];
```

Check that `U1` and `V1` are orthogonal matrices and that `A1=U1*S1*V1'` up to roundoff. In addition, print `U1`, `S1` and `V1` using `format long` and visually compare them with the singular values you get from the Matlab `svd` function. You should find they agree almost to roundoff, despite the coarse tolerance inside `jacobi_svd`. You also will note that the values do not come out sorted, as they do from `svd`. It is a simple matter to sort them, but then you would have to permute the columns of `U1` and `V1` to match. Again, some columns of `U1` and `V1` might be multiplied by (-1) and, if so, there must be an *even* number of those sign changes. Please include `U1`, `S1` and `V1` in your summary.

You now have an implementation of one SVD algorithm. This algorithm is a good one, with some realistic applications, and is one of the two algorithms available for the SVD in the GNU scientific library. (See `http://www.gnu.org/software/gsl/` for more detail.) In the following sections, you will see a different algorithm.

4 The "standard" algorithm

The most commonly used SVD algorithm is found in Matlab and in the Lapack linear algebra library. (See `http://www.netlib.org/lapack/`.) It is a revised version of one that appeared in Golub and Van Loan. The revised version is presented by J. Demmel, W. Kahan, "Accurate Singular Values of Bidiagonal Matrices," SIAM J. Sci. Stat. Comput., **11**(1990) pp. 873-912. This paper can also be found at `http://www.netlib.org/lapack/lawnspdf/lawn03.pdf`.

The standard algorithm is a two-step algorithm. It first reduces the matrix to bidiagonal form and then finds the SVD of the bidiagonal matrix. Reduction to bidiagonal form is accomplished using Householder transformations, a topic you have already seen. Finding the SVD of a bidiagonal matrix is an iterative process that must be carefully performed in order to minimize both numerical errors and the number of iterations required. To accomplish these tasks, the algorithm chooses whether Golub and Van Loan's original algorithm is better than Demmel and Kahan's, or vice-versa. Further, other choices are made to speed up each iteration. There is not time to discuss all these details, so we will only consider a simplified version of Demmel and Kahan's zero-shift algorithm.

In the Jacobi algorithm in the previous section, you saw how the two matrices U and V can be constructed by multiplying the various rotation matrices as the iterations progress. This procedure is the same for the

standard algorithm, so, in the interest of simplicity, most of the rest of this lab will be concerned only with the singular values themselves.

The first step in the algorithm is to reduce the matrix to bidiagonal form. You have already seen how to use Householder matrices to reduce a matrix to upper-triangular form. The idea, you recall, is to start with matrices U_0 and V_0 equal to the identity matrix and $B_0 = A$, so that $A = U_0 B_0 V_0^H$. Then,

1. For each column k of A, you choose a Householder matrix $\overline{H}_k$ in such a way that:
   (a) The matrix $\overline{B}_k = \overline{H}_k B_{k-1}$ has zeros *below* the diagonal in column k; and,
   (b) $U_k = U_{k-1}\overline{H}_k^H$, so that $A = U_k \overline{B}_k V_{k-1}^H$ for each k.
2. Apply a similar procedure for *row* k of A, choosing H_k in such a way that:
   (a) The matrix $B_k = \overline{B}_k H_k$ has zeros to the *right* of the *superdiagonal*; and,
   (b) V_k maintains the identity $A = U_k B_k V_k^H$

You cannot reduce the matrix to *diagonal* form this way because the Householder matrices for the rows would change the diagonal entries originally found for the columns and ruin the original factorization.

Exercise 5: Consider the following incomplete Matlab code, which is very similar to the `h_factor` function you wrote in the previous lab. (The matrices called `Q` and `R` there are called `U` and `B` here.)

```
function [U,B,V]=bidiag_reduction(A)
% [U B V]=bidiag_reduction(A)
% Algorithm 6.5-1 in Golub & Van Loan, Matrix Computations
% Johns Hopkins University Press
% Finds an upper bidiagonal matrix B so that A=U*B*V'
% with U,V orthogonal.  A is an m x n matrix

[m,n]=size(A);
B=A;
U=eye(m);
V=eye(n);
for k=1:n-1
  % eliminate non-zeros below the diagonal
  % Keep the product U*B unchanged
  H=householder(B(:,k),k);
  B=H*B;
  U=U*H';

  % eliminate non-zeros to the right of the
  % superdiagonal by working with the transpose
  % Keep the product B*V' unchanged

  ???

end
```

(a) Copy the above code to a file `bidiag_reduction.m`. Complete the statements with the question marks in them. Remember that `A=U*B*V'` remains true at all steps in the algorithm.

(b) Recover your version of `householder.m` from before or use mine.

(c) Test your code on the matrix `A=pascal(5)`. Be sure that the condition that `A=U*B*V'` is satisfied, that the matrix `B` is bidiagonal, and that the matrices `U` and `V` are orthogonal. Describe how you checked these facts and include the results of your tests in the summary.

(d) Test your code on a randomly-generated matrix

```
A=rand(100,100);
```

Be sure that the condition that `A=U*B*V'` is satisfied, that the matrix `B` is bidiagonal, and that `U` and `V` are orthogonal matrices. Describe how you checked these facts and include the results of your tests in the summary. **Hint:** You can use the Matlab `diag` function to extract any diagonal from a matrix or to reconstruct a matrix from a diagonal. Use "`help diag`" to find out how. Alternatively, you can use the `tril` and `triu` functions.

An important piece of Demmel and Kahan's algorithm is a very efficient way of generating a 2×2 "Givens" rotation matrix that annihilates the second component of a vector. The algorithm is presented on page 13 of their paper and is called "rot." It is important to have an efficient algorithm because this step is the central step in computing the SVD. A 10% (for example) reduction in time for `rot` translates into a 10% reduction in time for the SVD.

Algorithm (Demmel, Kahan) [c,s,r]=rot(f,g)

This algorithm computes the cosine, c, and sine, s, of a rotation angle that satisfies the following condition.

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

if $f = 0$ **then**
  $c = 0$, $s = 1$, and $r = g$
else if $|f| > |g|$ **then**
  $t = g/f$, $t_1 = \sqrt{1 + t^2}$
  $c = 1/t_1$, $s = tc$, and $r = ft_1$
else
  $t = f/g$, $t_1 = \sqrt{1 + t^2}$
  $s = 1/t_1$, $c = ts$, and $r = gt_1$
endif

Remark: The two different expressions for c, s and r are mathematically equivalent. The choice of which to do is based on minimizing roundoff errors.

Exercise 6:

(a) Based on the above algorithm, write a Matlab function m-file named `rot.m` with the signature

```
function [c,s,r]=rot(f,g)
% [c s r]=rot(f,g)
% more comments

% your name and the date
```

(b) Test it on the vector `[f;g]` with `f=1` and `g=0`. This vector already has 0 in its second component so you should get that `c=1`, `s=0`, and `r=1`.

(c) Test it on the vector `[0;2]`. What are the values of c, s and r? Perform the matrix product

```
[c s;-s c]*[f;g]
```

and check that the resulting vector is `[r;0]`.

(d) Test it on the vector `[1;2]`. What are the values of c, s and r? Perform the matrix product

```
[c s;-s c]*[f;g]
```

and check that the resulting vector is `[r;0]`.

(e) Test it on the vector `[-3;2]`. What are the values of c, s and r? Perform the matrix product

```
[c s;-s c]*[f;g]
```

and check that the resulting vector is `[r;0]`.

The standard algorithm is based on repeated QR-type "sweeps" through the bidiagonal matrix. Suppose B is a bidiagonal matrix, so that it looks like the following.

$$\begin{matrix} d_1 & e_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & e_2 & 0 & \dots & 0 \\ 0 & 0 & d_3 & e_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & d_{n-1} & e_{n-1} \\ 0 & 0 & \dots & 0 & 0 & d_n \end{matrix}$$

In its simplest form, a sweep begins at the top left of the matrix and runs down the rows to the bottom. For row i, the sweep first annihilates the e_i entry by multiplying on the right by a rotation matrix. This action introduces a non-zero value $A_{i+1,i}$ immediately *below* the diagonal. This new non-zero is then annihilated by multiplying on the left by a rotation matrix but this action introduces a non-zero value $B_{i,i+3}$, outside the upper diagonal, but the next row down. Conveniently, this new non-zero is annihilated by the same rotation matrix that annihilates e_{i+1}. (The proof uses the fact that e_i was zero as a consequence of the first rotation, see the paper.) The algorithm continues until the final step on the bottom row that introduces no non-zeros. This process has been termed "chasing the bulge" and will be illustrated in the following exercise.

The following Matlab code implements the above algorithm. This code will be used in the following exercise to illustrate the algorithm.

```
function B=msweep(B)
% B=msweep(B)
% Demmel & Kahan zero-shift QR downward sweep
% B starts as a bidiagonal matrix and is returned as
% a bidiagonal matrix

n=size(B,1);
for k=1:n-1
  [c s r]=rot(B(k,k),B(k,k+1));

  % Construct matrix Q and multiply on the right by Q'.
  % Q annihilates both B(k-1,k+1) and B(k,k+1)
  % but makes B(k+1,k) non-zero.
  Q=eye(n);
  Q(k:k+1,k:k+1)=[c s;-s c];
  B=B*Q';

  [c s r]=rot(B(k,k),B(k+1,k));

  % Construct matrix Q and multiply on the left by Q.
```

```
  % Q annihilates B(k+1,k) but makes B(k,k+1) and
  % B(k,k+2) non-zero.
  Q=eye(n);
  Q(k:k+1,k:k+1)=[c s;-s c];
  B=Q*B;

end
```

In this algorithm, there are two orthogonal (rotation) matrices, Q, employed. To see what their action is, consider the piece of B consisting of rows and columns $i-1$, i, $i+1$, and $i+2$. Multiplying on the right by the transpose of the first rotation matrix has the following consequence.

$$\begin{bmatrix} \alpha & * & \beta & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & \gamma \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & s & 0 \\ 0 & -s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & 0 & \gamma \end{bmatrix}$$

where asterisks indicate (possible) non-zeros and Greek letters indicate values that do not change. The fact that two entries are annihilated in the third column is not obvious and is proved in the paper.

The second matrix Q multiplies on the left and has the following consequence.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & s & 0 \\ 0 & -s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha & \beta & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & 0 & \gamma \end{bmatrix} = \begin{bmatrix} \alpha & \beta & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & \gamma \end{bmatrix}$$

The important consequence of these two rotations is that the row with three non-zeros in it (the "bulge") has moved from row $i-1$ to row i, (in this example, from row 1 to row 2) while all other rows still have two non-zeros. This action is illustrated graphically in the following exercise.

Exercise 7:

(a) Copy the above code to a file named `msweep.m`. (The "m" stands for "matrix.")

(b) The following lines of code will set roundoff-sized matrix entries in `B` to zero and use Matlab's `spy` routine to display all non-zeros. The `pause` statement makes the function stop and wait until a key is pressed. This will give you time to look at the plot.

```
% set almost-zero entries to true zero
% display matrix and wait for a key
B(find(abs(B)<1.e-13))=0;
spy(B)
disp('Plot completed. Strike a key to continue.')
pause
```

Insert *two* copies of this code into `msweep.m`, one after each of the statements that start off "`B=`."

(c) Apply your `msweep` function to the matrix

```
B=[1 11  0  0  0  0  0  0  0  0
   0  2 12  0  0  0  0  0  0  0
   0  0  3 13  0  0  0  0  0  0
   0  0  0  4 14  0  0  0  0  0
   0  0  0  0  5 15  0  0  0  0
   0  0  0  0  0  6 16  0  0  0
   0  0  0  0  0  0  7 17  0  0
```

```
0  0  0  0  0  0  0  8 18  0
0  0  0  0  0  0  0  0  9 19
0  0  0  0  0  0  0  0  0 10];
```

The sequence of plots shows the "bulge" (extra non-zero entry) migrates from the top of the matrix down the rows until it disappears off the end. Please include any one of the plots with your work. If `B=msweep(B);` what is `B(10,10)`?

Demmel and Kahan present a streamlined form of this algorithm in their paper. This algorithm is not only faster but is more accurate because there are no subtractions to introduce cancellation and roundoff inaccuracy. Their algorithm is presented below.

The largest difference in speed comes from the representation of the bidiagonal matrix B as a pair of *vectors*, d and e, the diagonal and superdiagonal entries, respectively. In addition, the intermediate product, BQ', is not explicitly formed.

Algorithm (Demmel, Kahan)

This algorithm begins and ends with the two vectors d and e, representing the diagonal and superdiagonal of a bidiagonal matrix. The vector d has length n.

$c_{\text{old}} = 1$
$c = 1$
for $k = 1$ to $n - 1$
  $[c, s, r] = \text{rot}(c\, d_k, e_k)$
  if $(k \neq 1)$ **then** $e_{k-1} = r\, s_{\text{old}}$ **endif**
  $[c_{\text{old}}, s_{\text{old}}, d_k] = \text{rot}(c_{\text{old}} r, d_{k+1} s)$
end for
$h = c\, d_n$
$e_{n-1} = h\, s_{\text{old}}$
$d_n = h\, c_{\text{old}}$

Exercise 8:

(a) Based on the above algorithm, write a function m-file named `vsweep.m` ("v" for "vector") with signature

```
function [d,e]=vsweep(d,e)
% [d e]=vsweep(d,e)
% comments

% your name and the date
```

(b) Use the same matrix as in the previous exercise to test `vsweep`. Compare the results of `vsweep` and `msweep` to be sure they are the same. You will probably want to remove the extra plotting statements from `msweep`.

```
B=[1 11  0  0  0  0  0  0  0  0
   0  2 12  0  0  0  0  0  0  0
   0  0  3 13  0  0  0  0  0  0
   0  0  0  4 14  0  0  0  0  0
   0  0  0  0  5 15  0  0  0  0
   0  0  0  0  0  6 16  0  0  0
   0  0  0  0  0  0  7 17  0  0
   0  0  0  0  0  0  0  8 18  0
```

```
   0  0  0  0  0  0  0  0  9 19
   0  0  0  0  0  0  0  0  0 10];
d=diag(B);
e=diag(B,1);
```

(c) Use the following code to compare results of `msweep` and `vsweep` and time them as well. Are the results the same? What are the times? You should find that `vsweep` is substantially faster.

```
n=200;
d=2*rand(n,1)-1; %random numbers between -1 and 1
e=2*rand(n-1,1)-1;
B=diag(d)+diag(e,1);
tic;B=msweep(B);mtime=toc
tic;[d e]=vsweep(d,e);vtime=toc
norm(d-diag(B))
norm(e-diag(B,1))
```

It turns out that repeated sweeps tend to reduce the size of the superdiagonal entries. You can find a discussion of convergence in Demmel and Kahan, and the references therein, but the basic idea is to watch the values of e and when they become small, set them to zero. Demmel and Kahan prove that this does not perturb the singular values much. In the following exercise, you will see how the superdiagonal size decreases.

Exercise 9:

(a) Consider the same matrix you have been using:

```
d=[1;2;3;4;5;6;7;8;9;10];
e=[11;12;13;14;15;16;17;18;19];
```

Use the following code to plot the absolute values $|e_j| : j = 1, \ldots, 9$ for each of 15 repetitions of the algorithm.

```
for k=1:15
  [d,e]=vsweep(d,e);
  plot(abs(e),'*-')
  hold on
  disp('Strike a key to continue.')
  pause
end
hold off
```

starting from the values of `e` and `d` above. You should observe that some entries converge quite rapidly to zero, and they all eventually decrease. Please include this plot with your summary.

(b) Plot $\|\mathbf{e}\|$ *versus* iteration number for the first 40 repetitions, starting from the same values as previously:

```
d=[1;2;3;4;5;6;7;8;9;10];
e=[11;12;13;14;15;16;17;18;19];
```

Please include this plot with your summary.

(c) Plot $\|\mathbf{e}\|$ *versus* iteration number *on a semilog plot* (`semilogy`) for the first 100 repetitions, starting from the above values. You should see that the convergence eventually becomes linear. You can estimate this convergence rate graphically by plotting the function $y = cr^n$ (where c can be taken to be 1.0 in this case), r can be found by trial and error, and n denotes iteration number.

When this line and the line of $\|\mathbf{e}\|$ appear parallel, you have found an estimate for r. To no more than two significant digits, what is the value of r you found? Please include the plot with your summary.

You should have observed that one entry converges very rapidly to zero, and that overall convergence to zero is asymptotically linear. It is often one of the two ends that converges most rapidly. Further, a judiciously chosen shift can make convergence even more rapid. We will not investigate these issues further here, but you can be sure that the software appearing, for example, in Lapack (used by Matlab) for the SVD takes advantage of these and other acceleration methods.

In the following exercise, the superdiagonal is examined during the iteration. As the values become small, they are set to zero, *and the iteration continues with a smaller matrix.* When all the superdiagonals are zero, the iteration is complete.

Exercise 10:

(a) Copy the following code to a function m-file named `bd_svd.m` ("bidiagonal svd").

```
function [d,iterations]=bd_svd(d,e)
% [d,iterations]=bd_svd(d,e)
% more comments

% your name and the date

TOL=100*eps;
n=length(d);
maxit=500*n^2;

% The following convergence criterion is discussed by
% Demmel and Kahan.
lambda(n)=abs(d(n));
for j=n-1:-1:1
  lambda(j)=abs(d(j))*lambda(j+1)/(lambda(j+1)+abs(e(j)));
end
mu(1)=abs(d(1));
for j=1:n-1
  mu(j+1)=abs(d(j+1))*mu(j)/(mu(j)+abs(e(j)));
end
sigmaLower=min(min(lambda),min(mu));
thresh=max(TOL*sigmaLower,maxit*realmin);


iUpper=n-1;
iLower=1;
for iterations=1:maxit
  % reduce problem size when some zeros are
  % on the superdiagonal

  % Don't iterate further if value e values are small
  % how many small values are near the bottom right?
  j=iUpper;
  for i=iUpper:-1:1
    if abs(e(i))>thresh
```

```
      j=i;
      break;
    end
  end
  iUpper=j;
  % how many small values are near the top left?
  j=iUpper;
  for i=iLower:iUpper
    if abs(e(i))>thresh
      j=i;
      break;
    end
  end
  iLower=j;

  if (iUpper==iLower & abs(e(iUpper))<=thresh) | ...
        (iUpper<iLower)
    % all done, sort singular values
    d=sort(abs(d),1,'descend');  %change to descending sort
    return
  end

  % do a sweep
  [d(iLower:iUpper+1),e(iLower:iUpper)]= ...
        vsweep(d(iLower:iUpper+1),e(iLower:iUpper));
end
error('bd_svd: too many iterations!')
```

(b) Consider the same matrix you have used before

```
d=[1;2;3;4;5;6;7;8;9;10];
e=[11;12;13;14;15;16;17;18;19];
B=diag(d)+diag(e,1);
```

What are the singular values that `bd_svd` produces? How many iterations did `bd_svd` need? What is the largest of the differences between the singular values `bd_svd` found and those that the Matlab function `svd` found for the matrix `B`?

(c) Generate a random matrix

```
d=rand(30,1);
e=rand(29,1);
B=diag(d)+diag(e,1);
```

Use `bd_svd` to compute its singular values. How many iterations does it take? What is the largest of the differences between the singular values `bd_svd` found and those that the Matlab function `svd` found for the matrix `B`?

Remark: In the above algorithm, you can see that the singular values are forced to be positive by taking absolute values. This is legal because if a negative singular value arises then multiplying both it and the corresponding column of U by negative one does not change the unitary nature of U and leaves the singular value positive.

You saw in the previous exercise that the number of iterations can get large. It turns out that the number of iterations is dramatically reduced by the proper choice of shift, and tricks such as sometimes choosing to

run sweeps up from the bottom right to the top left instead of always down as we did, depending on the matrix elements. Nonetheless, the presentation here should give you the flavor of the algorithm used.

In the following exercise, you will put your two functions, `bidiag_reduction` and `bd_svd` together into a function `mysvd` to find the singular values of a full matrix.

Exercise 11:

(a) Write a function m-file named `mysvd.m` with the signature

```
function [d,iterations]=mysvd(A)
% [d,iterations]=mysvd(A)
% more comments

% your name and the date
```

This function should:

i. Use `bidiag_reduction` to reduce the matrix A to a bidiagonal matrix, `B`;

ii. Extract `d` and `e`, the diagonal and superdiagonal from `B`; and,

iii. Use `bd_svd` to compute the singular values of A.

(b) Given the matrix `A=magic(10)`, use `mysvd` to find its singular values. How many iterations are required? How do the singular values compare with the singular values of `A` from the Matlab function `svd`?

(c) The singular values can be used to indicate the rank of a matrix. What is the rank of `A`?

5 Extra Credit: Latent semantic indexing (LSI) (8 points)

"Latent semantic indexing" is a very interesting idea that uses SVD to help index natural language documents. One explanation that is more complete than the one here can be found in the Wikipedia: `http://en.wikipedia.org/wiki/Latent_semantic_analysis.`

Suppose you are given a number of documents and a number (not the same number) of words that appear in some or all of the documents. First, create an "incidence matrix" that has a 1 in position (i, j) if the i^{th} document contains at least one occurrance of the j^{th} word, and 0 otherwise.

It is important to realize that if you are given k words and construct a vector with a 1 in positions corresponding to the given words, then multiplying the incidence matrix by this vector will yield a vector $\mathbf{v}$ with entries $0 \leq v_i \leq k$ and with the v_i indicating how many of the given words appear in the i^{th} document.

It is natural, then to compute the SVD of the incidence matrix and keep only the singular vectors with large singular values, just as was done earlier in compressing the Mars picture. The SVD comes with a bonus: the right singular vectors (the columns of V) provide groupings of words that are related according to their use in the documents, and can be regarded as "concepts" appearing in the documents. The left singular vectors (the columns of U) provide groupings of the documents according to the words. The most surprising of the features of the SVD is that multiplication of the incidence matrix by a vector representing a group of words will identify precisely those documents containing the words but a similar multiplication by the matrix constructed of only a few singular values will identify documents containing *related* words!

In the following exercise, you will see an example of this process from which you can draw your own conclusions.

To generate the exercise, I took a list of titles of publications from the Math Department web site a few years ago and extracted a list of words and constructed an incidence matrix from the titles (regarded as "documents") and the words. You will see how the SVD is able to identify "concepts" consisting of groups of words related because of the way they are used in the titles. You will also see how one could use the SVD to identify documents that employ certain words.

I must mention a caveat. This example is flawed by its use of titles and authors from departmental research reports. In the first place, the number of titles is not large, and in the second place, the titles themselves are too short to clearly elicit relationships among words. Titles and authors are used because it was convenient to gather the data and because you students might be familiar with the authors and topics. A better choice would have been the abstracts of the articles instead of their titles, but these were not conveniently available.

Results in the following exercise can be better understood if you are aware of various specialized groups of Mathematics faculty. For example, the Mathematics Department web pages describe the Mathematical Biology group as consisting of Professors Doiron, Ermentrout, Rubin, Swigon, and Troy, and the Numerical Analysis and Scientific Computing group as consisting of Professors Layton, Neilan, Trenchea, and Yotov. Prof. Neilan joined the faculty after the list of publications used in this exercise was constructed.

Before continuing, you need to understand how a new Matlab data structure is used. It is called a "cell array" and you can think of it as an array of cells in a spreadsheet. The contents of the cells can be just about anything and in this case will be strings of varying lengths. An ordinary Matlab array of strings would have to have all strings of the same length–not a convenient way to store titles of papers.

A cell array is indicated by "braces" or "curly brackets": "{" and "}". An ordinary column vector of numbers might be defined

```
v=[1;2;3;4;5];
```

while a cell array containing strings would be defined using curly brackets as

```
c={'one';'second string';'third';'4';'the fifth string is long.'}
```

There are two ways to refer to the contents of a cell array.

1. The expression `c(3)` means the cell array containing only the third of the cells in the cell array `c`. In this example, `c(3)` is the cell array {`'third'`}.

2. The expression `c{3}` means the *contents* of the third of the cells in the cell array `c`. In this example, `c{3}` is the *string* `'third'`.

Cell arrays can contain any Matlab data in their cells. A cell array might have a string in its first position, a matrix in its second position, a vector in its third position, and another cell array in its fourth position. This flexibility provides great power to the script writer, but we will only be using cell arrays that contain strings of varying lengths, and the reason we are using cell arrays and not string arrays is because a cell array can contain strings of varying lengths while a string array can contain only strings of the same lengths.

For the following exercise, I have constructed three cell arrays and a matrix, and they are included in the file `reports_data.m`.

1. The cell array `titles` is a list of titles of recent research reports that appear on the Math Department web site `http://www.mathematics.pitt.edu/research/technical-reports`.

2. The cell array `authors` is a list of authors corresponding to the titles.

3. The cell array `words` is a list of selected words that appear in the titles. Most of the meaningful words appearing in the titles are included among the `words` cells. There are words that basically are the same (such as "approximation" and "approximations") that are included because I did not want to introduce sophisticated parsing of English words into this exercise.

4. The matrix `incidence` is the incidence matrix so that `incidence(j,k)=1` if `word(k)` appears at least once in `titles(j)` and otherwise `incidence(j,k)=0`.

Exercise 12:

(a) For the example cell array given above and repeated here

```
c={'one';'second string';'third';'4';'the fifth string is long.'}
```

Please answer the following questions.

i. In a few words, what is the difference between `c{2}` and `c(2)`.

ii. What are `numel(c(2))` and `numel(c{2})`? Explain the meanings of the two different values.

(b) If you have not done it already, download the script m-files `reports_data.m`, and also `word_vec.m`. `word_vec` is a function that takes a single string and returns a vector with 1 corresponding to words that are present in the string and 0 for words that are not present.

(c) Run the script m-file `reports_data.m` by typing its name without the "`.m`". This will create the cell arrays `authors` and `words` as well as the incidence matrix `incidence`.

(d) Consider the (nonsense) search string

```
ss = 'The cascade of Brinkman code';
```

use `word_vec` to construct a vector, `v`, corresponding to the words appearing in `ss`. How many of the distinguished words appear in `ss`? Check that `v` is correct with the following commands.

```
words{find(v==1)}
```

Be sure that the words printed are the ones appearing in `ss`.

(e) Use the expression `t=incidence*v;` to generate a vector indicating which titles contain one or more of the distinguished words. What titles are indicated?

(f) Compute the singular value decomposition `[U S V]=svd(incidence);` and plot the singular values. Please include the plot with your summary.

(g) The vectors (colums of) `V` represent groups of words that tend to appear together in the titles. These can be regarded as "concepts" appearing in the titles. Each column of `V` is a unit vector and the few important words tend to have absolute values above 0.2. What are the important words in the first column? In columns 2, 12, and 13?

(h) The vectors (columns of) `U` represent groups of titles with similar words in them. It is hard to see the relationships by looking at the titles, but looking at the *authors* clearly shows authors who tend to work together. What is the set of authors associated with `abs(U(:,2))>0.2`? With column 8? With column 15?

(i) There are exactly two titles containing the word "canard." Use multiplication by the incidence matrix to find them.

(j) The multiplication by a compressed incidence matrix

```
C = U(:,1:25) * S(1:25,1:25) * V(:,1:25)' ;
t = C * v;
```

where `v` is the vector associated with the single word "canard," will yield a selection of titles satisfying the criterion `abs(t) > 0.2`. Please include the list of titles in your summary. It is interesting that one of the titles containing the word "canard" itself does not appear in this list but that all the titles refer to behavior of excited synapses in neurons, from which one could conclude that the word "canard" refers to the behavior of excited synapses under certain circumstances.
Remark: A "canard" is a phenomenon associated with "fast/slow" systems of nonlinear ordinary differential equations near bifurcation points. One explanation can be found in the article, Diener, M., "The canard unchained or how fast/slow dynamical systems bifurcate," *Mathematical Intelligencer* Volume 6, Number 3(1984), pp. 38-49. This article is available online from the library.

Last change `$Date: 2016/02/06 01:32:30 $`