1. (30 points) Consider the MATLAB functions mystery1.m, mystery2.m and mystery3.m. In the following, assume that the input parameter $A$ is an $m \times n$ matrix and that the input parameter $\texttt{tol}$ is small, say in the range $10^{-8}$ to $10^{-15}$.

(a) What does mystery1 compute, i.e., what do $u, us, v, vs$ converge to in terms of the input parameter $A$, when the input parameter $v0$ is set to random column vector of length $n$? You should run the program to observe what is happening, using the debugger to watch how the variables change, but you also need carefully explain why this behavior is occurring. You will not get full credit if your explanation is not complete. Also, how does the accuracy of the answer depend on the value of $\texttt{tol}$? Finally, how does the convergence rate depend on the input matrix $A$? Illustrate this with some examples, making use of the $\texttt{iters}$ output parameter.

(b) If you set the input $v0$ for mystery1 to a very specific choice (depending on the parameter $A$) you will see different behavior. Investigate this and explain it.

(c) What does mystery2 compute if $V0$ is set to a random $n \times k$ matrix? Again explain why.

(d) What does mystery3 compute if $V0$ is set to a random $n \times k$ matrix? What is the purpose of the QR factorizations in the loop and why does this make the behavior different to that of mystery2? Be sure to explain all the output parameters. In particular, you must explain the meaning of the output parameters $\texttt{UR}$ and $\texttt{VR}$ and what they are converging to. As earlier, you must explain why this behavior is occurring.

2. (30 points) Download WallOfWindows.jpg on the course web page and read it into MATLAB using $A=$imread('WallOfWindows.jpg','jpg'). Display it with $\texttt{image(A)}$. If you type $\texttt{whos A}$ you will see that $A$ is
a 3456 × 4608 × 3 three-dimensional “matrix” (or tensor) of unsigned 8-bit integers. Split these into red, green and blue components, each a 3456 × 4608 matrix of unsigned integers, via \( A_{\text{red}} = A(:,:,1) \), etc, and convert these to double precision matrices using \texttt{double}.

(a) Compute the economy-sized SVD of each color component matrix separately using \texttt{svd(M,0)}. (If you like you can transpose the component matrix first so it has more rows than columns, as in my notes, but this is actually not necessary; we made this assumption only for convenience.) Then, for each component matrix, compute its nearest rank \( r \) approximation (see p. 6 of \textit{Notes on Eigenvalues and Singular Values}), for \( r = 10, 20, 30 \) and 100. Recombine them into the 3456 × 4608 × 3 format, and then display the resulting four pictures using \texttt{image}, labelling them appropriately. How well do they approximate the original picture in your opinion? This shows that SVD is a useful approach to image compression: instead of storing the whole matrix, you need only to store the part of the SVD needed to construct the rank \( r \) approximation: how much storage does this require, in terms of \( m, n \) and \( r \)?

(b) The problem with the method just described is that it is expensive to call \texttt{svd} on such a large matrix and it would take much too long for a much larger matrix. Instead, make use of one of the \texttt{mystery} functions in the previous question. Can you get pictures that are just as good as using \texttt{svd} with less computational expense? You may find you don’t have to make \texttt{tol} very small to get good pictures. How big can you make \texttt{tol} and still get good pictures, and how does the time required compare to the time required using \texttt{svd}?

3. (40 points) Google uses a famous scheme, PageRank, to rank web pages. We start with an \( n \times n \) \textit{adjacency matrix} \( M \), corresponding to a web of \( n \) pages, with \( M_{ij} = 1 \) if page \( j \) has a link to page \( i \) and \( M_{ij} = 0 \) otherwise. Let \( N_j \) be the number of pages that page \( j \) points to. Then, as long as \( N_j > 0 \), column \( j \) of a matrix \( A \) is set according to the following rule (see A&G p. 224)

\[
A_{ij} = \begin{cases} 
\frac{1}{N_j} & \text{if } M_{ij} = 1 \\
0 & \text{otherwise}. 
\end{cases}
\]

By construction, the \( j \)th column of \( A \) sums up to one. However, if \( N_j = 0 \), which happens when page \( j \) does not point anywhere (the
“dangling node” case), this does not work, so instead we set (see A&G p. 225)

\[ A_{ij} = \frac{1}{n} \text{ for all } i = 1, \ldots, n. \]

Then the \( j \)th column of \( A \) sums to one in this case too.

The “Google matrix” is then defined to be (see A&G, p. 226)

\[
G = \alpha A + (1 - \alpha) \frac{1}{n} ee^T
\]

where \( \alpha \) is a “damping parameter” satisfying \( 0 < \alpha < 1 \) and \( e = [1, \ldots, 1]^T \). Let us denote the eigenvalues of \( G \) by \( \lambda_1, \ldots, \lambda_n \), where

\[
|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|
\]

that is, ordered by “complex modulus” or “complex magnitude” (see A&G, p. 71).

(a) Show that each column of \( G \) also sums to one, and therefore \( \|G\|_1 = 1 \).

(b) On p. 226 of A&G it says we could use any vector \( u \) instead of \((1/n)e\) in the equation for \( G \), but if we did that, the columns of \( G \) might not sum to one. What condition on \( u \) would we need to make sure the columns of \( G \) still sum to one? (We will stick with \((1/n)e\) in the definition of \( G \).)

(c) Show that \( e \), the vector of all ones, is an eigenvector of the transpose of the Google matrix corresponding to the eigenvalue one. (Equivalently, \( e^T \) is a “left eigenvector” of \( G \) corresponding to the eigenvalue one, since \( e^T G = e^T \).) Since the eigenvalues of \( G \) and \( G^T \) are the same, this means that \( G \) also has an eigenvalue one, but we do not know the corresponding eigenvector \( x \). This is what PageRank computes, using the power method.

(d) Look at the bottom of p. 77 in A&G. This defines the spectral radius of a matrix \( B \), denoted \( \rho(B) \), which is the maximum of the eigenvalues of \( B \) measured in complex modulus. Thus, using the ordering of \( \lambda_j \) given above, \( \rho(G) = |\lambda_1| \). The last line on p. 77 explains that, by definition of an induced norm, the spectral radius of \( B \) is always less than or equal to \( \|B\| \) for any induced norm. What does this tell you about all the eigenvalues of \( G \)?

The matrix \( G \) is a “nonnegative” matrix, in the sense that all its entries \( G_{ij} \geq 0 \) (this is not the same as being positive semidefinite; in fact, \( G \) is not a symmetric matrix). The remarkable theory of nonnegative
matrices says that only one eigenvalue of $G$, namely $\lambda_1$, equals one, and that the eigenvector $x$ corresponding to the eigenvalue one is also nonnegative (or nonpositive, since we can always scale it by $-1$, but it’s more convenient to take it as nonnegative). It also says that all the other eigenvalues of $G$, $\lambda_j$, $j = 2, \ldots, n$, satisfy $|\lambda_j| < 1$.

(e) Show that this last conclusion is not true if we allow $\alpha = 1$. In particular, could all the eigenvalues of $G$ be equal to one? What would this say about the page links in the web?

(f) What are the eigenvalues of $G$ when $\alpha = 0$?

(g) Write a function whose input is the adjacency matrix $M$ as a sparse matrix, and the damping parameter $\alpha$, that computes the matrices $A$ and $G$ and the eigenvalues of $G$ using `eig`. This routine requires its input matrix to be full, not sparse, and anyway although $A$ is sparse, $G$ is completely full. Load the adjacency matrix $M$ for the Google 500 graph, which is small enough that the full format for $G$ is not a problem and `eig` is fast. Setting $\alpha = 0.85$, compute the eigenvectors and eigenvalues of $G$ with $[X, \text{Lambda}] = \text{eig}(G)$, and plot all the eigenvalues as complex numbers in the complex plane: $\text{lambda} = \text{diag}(\text{Lambda})$; $\text{plot}($real$(\text{lambda}),\text{imag}(\text{lambda}),'rx'$), axis equal (this plots red crosses). Repeat for $\alpha = 0.5$ and $\alpha = 0.25$, using different colors and symbols. Show them all on the same plot and also plot circles of radius $\alpha$ in compatible colors. Use `legend` to show what’s what. What do you notice about the eigenvalues?

(h) Again for each of $\alpha = 0.85, 0.5$ and 0.25, set $x$ to the column of $X$ corresponding to the eigenvalue one in $\text{lambda}$ (the eigenvalues may not be delivered in sorted order), normalize it so $\|x\|_1 = 1$, and double check that it is a positive vector (multiplying by $-1$ if it is a negative vector; if it has mixed signs you have a bug). Sort its components with $[\text{xsort, index}] = \text{sort}(x, '\text{descend}')$, and print the largest 10 components along with the corresponding URL names in $\text{url}$, namely $\text{url}(\text{index}(1:10))$. These are the highest 10 ranked web pages in the Google 500 graph (the result may depend on $\alpha$).

(i) Now write a function to implement the power method for computing the desired page rank vector $x$, which requires matrix-vector products using the formula for $G$: do not store the full matrix $G$. A key point is that you can multiply $(1/n)e e^T$ onto a vector $v$ to give you $(1/n)e e^T v = (1/n)(e^Tv)e$ without computing or storing $ee^T$. For the same reason, do not explicitly replace the zero
columns of $A$ by the vectors $(1/n)e$; you just need to remember to include these in the matrix vector product.

Set the initial vector to $(1/n)e$. Since $G$ has column sums equal to one, you can omit the normalization step in the power method: the vector should always have 1-norm equal to one. Since we know the eigenvalue is one, instead of the Rayleigh quotient we can use the eigenvector residual norm $\mu = \|Gv - v\|_1$ as a termination criterion (but compute $Gv$ only once, not twice, in the loop). Test your code on the Google 500 matrix with $\alpha = 0.85$, running it until $\mu$ is “sufficiently small”, and make sure that the final $v$ is reasonably close to the normalized $x$ found by `eig`; if not, you have a bug. Output a vector of all the residuals $\mu$ and plot it against the iteration count. By approximately what factor is the residual reduced each step in the final stages of the iteration? How does this relate to $|\lambda_2|$, the second largest eigenvalue delivered by `eig`? (You can use `sort(lambda, 'descend')` to sort the eigenvalues, since it sorts complex numbers by complex modulus, as desired). Repeat for the other two values of $\alpha$, namely, 0.5 and 0.25.

(j) Once you are sure your power method code is working correctly, load the Purdue 77587 graph, which gives $A$ directly, not the adjacency matrix $M$. The column sums of $A$ are already one and there are no zero columns. It is no problem to store the very sparse matrix $A$, but storing $G$ would result in running out of memory. The power method should still work very efficiently. If it is too slow for $\alpha = 0.85$, focus on $\alpha = 0.5$ and $\alpha = 0.25$. Plot the $\mu$ values, and print the top 10 page ranks and corresponding URLs. Estimate the modulus of the second largest eigenvalue, $|\lambda_2|$, based on the convergence rate of $\mu$ and your previous observations.

Credits: I collected the Google graph data using Cleve Moler’s `surfer.m`. The Purdue graph dates from 2001 and was taken from David Gleich’s web page at Purdue.