CS 322: Assignment P4
Due Via CMS: 9AM, Monday, April 3, 2006

Files related to this assignment are available through the course download 322DataCodes.zip or are provided on the course web site http://www.cs.cornell.edu/courses/cs322/2006sp/. When you submit your solutions for grading it must be done through the Course Management System (CMS). You must name your files precisely as we require. For this assignment the files to submit are MEigs.m, Transits.m, and BuildModel.m. The deadline is firm. It is your responsibility to check the announcements page on the website for corrections and hints.

Points are deducted for poor style. Your solutions should be well commented and in particular the functions that you write must have clear specifications. You are expected to exploit fully MATLAB’s vector capability.

You must work either by yourself or with at most one other partner. You are allowed to discuss background issues with others, but the programs you submit must just be the work of you and your partner (if you have one). If something comes up and you are unclear about our Academic Integrity Policy (posted on the course web site) contact a member of the course staff immediately.

Part A (5 pts) The Discrete Cosine Transform

The discrete cosine transform of a vector \( x \in \mathbb{R}^n \) is a matrix-vector product: \( y = C_n x \) where the DCT matrix \( C_n \in \mathbb{R}^{n \times n} \) is defined by

\[
C_n = (c_{ij})
\]

\[
c_{ij} = \cos \left( \frac{(i-1)(2j-1)\pi}{2n} \right).
\]

It’s a 1-liner in MATLAB:

\[
C = \cos((\pi/(2n)) \cdot (0:n-1)' \cdot (1:2:2n-1))
\]

An important property of \( C_n \) is that its inverse is a column scaling of its transpose. Indeed, it can be shown that

\[
C_n C_n^T = D_n = \text{diag}(n,n/2,\ldots,n/2)
\]

and so \( C_n (C_n^T D_n^{-1}) = I_n \), i.e., \( C_n^{-1} = C_n^T D_n^{-1} \). Let’s check this out for \( n = 3 \):

\[
C_3 = \begin{bmatrix}
1 & 1 & 1 \\
\cos(30^\circ) & \cos(90^\circ) & \cos(150^\circ) \\
\cos(60^\circ) & \cos(180^\circ) & \cos(300^\circ)
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 \\
\sqrt{3}/2 & 0 & -\sqrt{3}/2 \\
1/2 & -1 & 1/2
\end{bmatrix}
\]

\[
C_3 C_3^T = D_3 = \begin{bmatrix}
3 & 0 & 0 \\
0 & 3/2 & 0 \\
0 & 0 & 3/2
\end{bmatrix}
\]

\[
C_3^{-1} = C_3^T D_3^{-1} = \begin{bmatrix}
1 & \sqrt{3}/2 & 1/2 \\
1 & 0 & -1 \\
1 & -\sqrt{3}/2 & 1/2
\end{bmatrix} = \begin{bmatrix}
1/3 & 0 & 0 \\
0 & 2/3 & 0 \\
0 & 0 & 2/3
\end{bmatrix} = \begin{bmatrix}
1/3 & 1/\sqrt{3} & 1/3 \\
1/3 & 0 & -2/3 \\
1/3 & -1/\sqrt{3} & 1/3
\end{bmatrix}
\]

The DCT matrix \( C_n \) is highly structured making it possible to execute the matrix-vector product \( C_n x \) fast. In this problem you will come to appreciate this for the case \( n = 8 \). The 8-point DCT is central to JPEG image compression as you will see in Part B of this assignment.

If \( c_j = \cos(j\pi/16) \), for \( j = 1:7 \), then

\[
C_8 = \begin{bmatrix}
c_1 & c_3 & c_5 & c_7 & -c_5 & -c_3 & -c_1 \\
c_2 & -c_6 & -c_2 & -c_6 & c_2 & c_6 \\
c_3 & -c_7 & -c_1 & -c_5 & c_5 & c_1 & c_7 \\
c_4 & -c_4 & c_4 & -c_4 & c_4 & -c_4 & c_4 \\
c_5 & -c_1 & c_7 & c_3 & c_3 & c_7 & c_1 & c_5 \\
c_6 & -c_2 & -c_6 & -c_2 & -c_6 & c_2 & c_6 \\
c_7 & -c_5 & c_3 & c_1 & -c_3 & c_5 & -c_7
\end{bmatrix}
\]
In general, an $n = 8$ matrix-vector product requires $2n^2 = 128$ flops, but when the DCT matrix is involved, then work can be reduced by about 60%. The key idea is to observe that $C_8$ can be written as a product $C_8 = Q_3Q_2Q_1$, where

$$Q_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \quad Q_2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Q_3 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_1 & c_3 & c_5 & c_7 \\ 0 & 0 & c_2 & -c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_3 & -c_7 & -c_1 & -c_5 \\ c_4 & -c_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_5 & -c_1 & c_7 & c_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_7 & -c_5 & c_3 & -c_1 \end{bmatrix}$$

The DCT $y = C_8x = Q_3Q_2Q_1x = Q_3(Q_2(Q_1x))$ turns into a 3-step process:

- $u = Q_1x$, this involves 8 flops.
- $v = Q_2u$, this involves 4 flops.
- $y = Q_3v$, this involves 37 flops.

The inverse DCT $y = C_8^{-1}x = C_8^T D_8^{-1} x = (Q_3Q_2Q_1)^T D_8^{-1} x = Q_1^T(Q_2^T Q_3^T (D_8^{-1} x)$ can also be computed very efficiently

- $u = D_8^{-1} x$, this involves 8 flops.
- $v = Q_3^T u$, this involves 37 flops.
- $w = Q_2^T v$, this involves 4 flops.
- $y = Q_1^T w$, this involves 8 flops.

Study the above “$Q$-products” carefully and confirm the flop counts. Using these ideas, implement the following functions so that they perform as specified:

```matlab
function Y = DCT8(X)
% X is an 8-by-m matrix
% Y = C*X where C is the 8-by-8 DCT matrix.

function Y = IDCT8(X)
% X is an 8-by-m matrix
% Y satisfies C*Y = X where C is the 8-by-8 DCT matrix.
```

Your implementations should be vectorized, i.e., `DCT8` should involve 49 length-m vector operations while `IDCT8` should involve 57 length-m vector operations. Test your implementations on the script `P4A` and submit `DCT8` and `IDCT8` to CMS.

**Part B (5 pts) The JPEG Computation**

In this problem you will replicate the “matrix part” of the JPEG image compression scheme. (The “coding part” of the process deals with compressed representations of long bit strings that have long substrings of zeros. Let’s not go there!) A link to a JPEG tutorial is on the course website.

The starting point for us is an $m$-by-$n$ integer matrix whose entries are between 0 and 255. We’ll call this the picture matrix. Entry $a_{ij}$ is the grayness value of pixel $(i,j)$ with 0 corresponding to black and 255 corresponding to white. Assume $m = 8m_1$ and $n = 8n_1$ and think of $A$ as an $m_1$-by-$n_1$ block matrix with 8-by-8 blocks, e.g.,

$$A = \begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\
A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\
A_{41} & A_{42} & A_{43} & A_{44} & A_{45}
\end{bmatrix}$$

$m = 32$, $n = 40$

$A_{34} = A(17:24, 25:32)$
JPEG compression transforms the picture matrix $A$ “block-by-block” to another matrix $B$

\[
\begin{array}{cccccc}
A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\
A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\
A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \\
\end{array}
\rightarrow
\begin{array}{cccccc}
B_{11} & B_{12} & B_{13} & B_{14} & B_{15} \\
B_{21} & B_{22} & B_{23} & B_{24} & B_{25} \\
B_{31} & B_{32} & B_{33} & B_{34} & B_{35} \\
B_{41} & B_{42} & B_{43} & B_{44} & B_{45} \\
\end{array}
\]

according to the rule

$$B_{ij} = \text{round} \left( \frac{(C_8 A_{ij} C_8^T) \cdot Q}{Q} \right)$$

where $Q$ is an 8-by-8 quantization matrix. A popular choice is

$$Q = \begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \\
\end{bmatrix}.$$

We refer to $B$ as the JPEG matrix. Let’s take a look at the computation of the $B_{ij}$ using an example. Here is a typical block from $A$, i.e., an 8-by-8 patch from the picture:

\[
\begin{bmatrix}
75 & 63 & 66 & 67 & 66 & 71 & 83 & 95 \\
72 & 64 & 71 & 76 & 78 & 82 & 88 & 90 \\
79 & 76 & 78 & 77 & 74 & 76 & 85 & 91 \\
\end{bmatrix}
= AIJ
\]

We take the DCT of the columns $C_8 A_{ij}$ and then the rows $(C_8 A_{ij}) C_8^T$ and get

\[
\begin{bmatrix}
-96.167 & 43.771 & 97.150 & -68.698 & 8.000 & -4.257 & 1.972 & -2.129 \\
44.224 & 18.136 & 0.204 & -8.276 & -5.666 & -6.305 & -2.313 & -1.205 \\
\end{bmatrix}
\]

We take the pointwise divide $(C_8 A_{ij}) C_8^T / Q$ has the effect of accentuating the big-to-small drift:

\[
\begin{bmatrix}
322.6250 & -13.0597 & 7.9399 & -3.5460 & 0.8839 & 0.6711 & 0.1355 & 0.0972 \\
-22.7235 & -5.8967 & 9.3625 & 0.2393 & 1.5635 & 0.4881 & 0.3413 & 0.1744 \\
13.9625 & -0.6769 & -6.9525 & -0.9817 & -0.4095 & -0.2728 & -0.1695 & -0.1317 \\
-9.0296 & -3.4536 & 0.7973 & 2.6658 & 0.4229 & 0.1972 & 0.1707 & 0.1044 \\
-5.3426 & 1.9896 & 2.6257 & -1.2268 & 0.1176 & -0.0391 & 0.0192 & -0.0276 \\
1.8427 & 0.5182 & 0.0037 & -0.1293 & -0.0699 & -0.0606 & -0.0205 & -0.0131 \\
0.1282 & 0.0101 & -0.0858 & 0.1009 & -0.0287 & -0.0579 & -0.0313 & -0.0212 \\
-0.2860 & 0.0887 & 0.2264 & -0.1206 & 0.0876 & 0.0794 & 0.0306 & 0.0177 \\
\end{bmatrix}
\]

Notice that entries range from big to small as we move from the upper left corner to the lower right corner. This is the “miracle” of the DCT; it represents $A_{ij}$ in a particularly handy basis so that the important non-zero information is “compressed” into a small part of the matrix. The script `ShowDCT` displays this basis.

Now look at the quantization matrix $Q$ above. The pointwise divide $(C_8 A_{ij} C_8^T)/Q$ has the effect of accentuating the big-to-small drift:
Rounding this produces

\[
\begin{bmatrix}
323 & -13 & 8 & -4 & 1 & 1 & 0 & 0 \\
-23 & -6 & 9 & 0 & 2 & 0 & 0 & 0 \\
14 & -1 & -7 & -1 & 0 & 0 & 0 & 0 \\
-9 & -3 & 1 & 3 & 0 & 0 & 0 & 0 \\
-5 & 2 & 3 & -1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

This is typical. About 75% of \(B_{ij}\)’s entries are zero. The rest of the JPEG process involves a clever “concatenation” of the nonzero information in all the \(B_{ij}\). In the end the number of bits required to represent the picture is typically reduced by a factor of about 20.

This takes care of the compression. How do we “decompress” a JPEG matrix? The idea is simply to visit each block and “undo” the \(Q\)-divide and the DCT’s:

\[
\tilde{A}_{ij} = C_8^{-1}(B_{ij} \ast Q)(C_8^{-1})^T
\]

Complete the following functions so that they perform as specified:

```matlab
function B = JPEG(A,Q)
    % A is an m-by-n picture matrix with the property that m and n are
    % each divisible by 8.
    % Q is an 8-by-8 quantization matrix.
    % B is the corresponding JPEG matrix.
    ...
end

function B = IJPEG(A,Q)
    % A is an m-by-n JPEG matrix with the property that m and n are
    % each divisible by 8.
    % Q is an 8-by-8 quantization matrix.
    % B is the corresponding picture matrix.
    ...
end
```

Here are some linear algebra pointers that can help you develop efficient, fully vectorized implementations:

- Although the Part A function \(\text{DCT8}\) is designed to compute products of the form \(Y = C_8X\), it can also be used to compute products of the form \(G = FC_8^T\) where \(F \in \mathbb{R}^{m \times 8}\). It’s a 1-liner: \(G = \text{DCT8}(F')'\). Likewise, \(\text{IDCT8}\) can be used to compute \(G = F(C_8^{-1})^T\).

- There are obvious double-loop implementations of \(\text{JPEG}\) and \(\text{IJPEG}\) that process one \(A_{ij}\) at a time. But by using reshape you can avoid looping. Hint: \(\text{DCT8(reshape(A,8,m*n/8))}\) computes the products \(C_8A_{ij}\) over all \(i\) and \(j\).

- The operations involving \(Q\) can also be done without loops by making use of the MATLAB function \(\text{kron}\). Here’s a hint:

\[
R = \text{kron(ones(4,5),Q)} \leftrightarrow R = \begin{bmatrix}
Q & Q & Q & Q & Q \\
Q & Q & Q & Q & Q \\
Q & Q & Q & Q & Q \\
Q & Q & Q & Q & Q
\end{bmatrix}
\]

You can test your codes with the script \(\text{P4B}\). Submit both \(\text{JPEG}\) and \(\text{IJPEG}\) to CMS. Sample pictures are also provided.
Part C (10pts) PageRank

Suppose \( A \in \mathbb{R}^{n \times n} \) is a Markov matrix, i.e., \( A \) has nonnegative entries and unit column sums. A vector \( v \in \mathbb{R}^n \) that satisfies \( Av = v \) and \( v_1 + \cdots + v_n = 1 \) is called a Markov vector. We assume in this problem that it is unique. There are two basic approaches to computing the Markov vector. There are iterative methods like

\[
\text{function } [v, err, k] = \text{MarkovVec}(A, \text{tol}, \text{itMax}, v_0) \\
\% A is an n-by-n Markov matrix \\
\% v0 is a column n-vector with nonnegative entries that sum to one. \\
\% tol>=0 and itMax is a positive integer \\
\% v is a column n-vector obtain by performing k steps of the power method, \\
\% k never being larger than itMax. If k<itMax then norm(A*v - v)<= tol. \\
\n\text{n = length(v0);} \\
\text{vNext = A*v0; } \\
\text{k = 1; } \\
\text{err = norm(vNext-v0); } \\
\text{while err>tol & k<itMax } \\
\text{v = vNext; } \\
\text{vNext = A*v; } \\
\text{k = k+1; } \\
\text{err = norm(vNext-v); } \\
\text{end}
\]

These are attractive if \( A \) is large and structured because the only way \( A \) is involved is through matrix-vector multiplication. A special-purpose function \( y = A\text{prod}(x) \) can be implemented to carry out \( y = Ax \).

In contrast to iterative methods are direct methods. These work by using a matrix factorization to compute a null vector of \( A - I \). (Note that the Markov vector satisfies \((A - I)v = 0\).) For example, it can be shown that if we use Gaussian elimination to compute the LU factorization \( P(A - I) = LU \) then

\[
U = \begin{bmatrix} U_1 & g \\ 0 & 0 \end{bmatrix}, \quad U_1 \in \mathbb{R}^{n-1 \times n-1}, \ g \in \mathbb{R}^{n-1}
\]

with nonsingular \( U_1 \). If

\[
w = \begin{bmatrix} -U_1^{-1}g \\ 1 \end{bmatrix}
\]

then \( Uw = 0 \). It follows from \( P(A - I)w = LUw = 0 \) that \( Aw = w \). Thus, \( v = w/\text{sum}(w) \) is the Markov vector.

In this problem you will implement a function that computes PageRank using each of these approaches. Recall that the Google Matrix is a Markov matrix:

\[
\text{function } A = \text{GoogleM}(G, p) \\
\% 0 <= p <= 1 \\
\% G is an n-by-n matrix of zeros and ones. \\
\% G(i,j) is one if and only if there is a link from page j to page i \\
\% A is the n-by-n "Google Matrix" \\
\n[n,n] = \text{size}(G); \ z = \text{ones(n,1)}/n; \ A = \text{zeros(n,n)}; \\
\text{for } j=1:n \\
\text{m = sum(G(:,j)); } \\
\text{if m>0 } \\
\text{g_tilde = G(:,j)/m; } \\
\text{else } \\
\text{g_tilde = z; } \\
\text{end } \\
\text{A(:,j) = p*g_tilde + (1-p)*z; } \\
\text{end}
\]

If \( v \) is the Markov vector of the Google matrix, and \( v_1 > v_2 \geq \cdots \geq v_n \), then page \( i_k \) has PageRank \( k \).

\( A \) is highly-structured. The starting point is for you to implement a function \( y = \text{GoogleProd}(x, G, p, \ldots) \) that can efficiently compute the matrix-vector product \( Ax \) where \( A \) is the Google matrix based on the link-structure matrix \( G \). We assume \( G \) to be in sparse format. The key to doing this is to observe that
\[ A = p\tilde{G} + (1 - p)ee^T/n \]

where \( e \in \mathbb{R}^n \) is the vector of all ones. Thus \( Ax = p\tilde{G}x + (1 - p)(ee^T/n)x \) and you should organize the two terms efficiently. Regarding the product \( \tilde{G}x \), if \( G \) has no zero columns then \( \tilde{G} = GD \) where \( D \) is diagonal. In this case \( Gx = GDx = G(Dx) \) could be efficiently carried out since \( G \) is in sparse format. You'll have to modify this chain of reasoning to handle the case if \( G \) has a zero column.

Your task is to implement the following functions so that they perform as specified:

```matlab
function [v,err,k] = Google1(G,p,tol,itMax,v0)

% G is an n-by-n link-structure matrix represented in sparse form
% p is a scalar that satisfies 0 < p < 1
% tol>0 and itMax is a positive integer
% v0 (if present) is a non-negative column n-vector with sum(v0)=1
% If v0 is not included in the calling sequence, then a
% starting vector is automatically generated.
% m is a positive integer
% v is a nonnegative column n-vector with sum(v)=1 obtained by performing
% k steps of the power method,
% k never being larger than itMax. If k<itMax then norm(A*v - v)<= tol.

function v = Google2(G,p,idx)

% G is an n-by-n link-structure matrix represented in sparse form
% p is a scalar that satisfies 0 < p < 1
% idx is a length m vector of indices that satisfy 1<=idx(1)<idx(2)<...<idx(m)<=n
% Let A be the Google matrix associated with the "subweb" defined by G(idx,idx).
% v is the Markov vector associated with A.
```

Develop a good (cheap) heuristic for generating \( v_0 \) given that you gave \( G \) to play with. A key part of your implementation will be the design of \( \text{GoogleProd} \) which should be a subfunction. Your implementation of \( \text{Google2} \) should compute the Markov vector via the LU factorization as shown above.

For your information, \( \text{MarkovVec} \) and \( \text{GoogleM} \) are available on the website, see the Lecture 14 examples. You might want to revisit \( \text{GetWebData} \) and \( \text{ProfileWebs} \) that are part of \( 322\text{Codes} \).

Submit \( \text{Google1} \) and \( \text{Google2} \) to CMS. A test script \( P4C \) is on the website.