Selected Solutions to Assignment #6

Problems 4.3, exercise 3: In the given situation,

- PA is A with its rows swapped according to the permutation $\vec{p} = (p_1, \dots, p_n)$,
- P^{-1} is the matrix which undoes the permutation given by P, and also $P^{-1} = P^\top$ (the transpose),
- AP is A with its columns swapped according to the inverse permutation of $\vec{p}$,
- AP^{-1} is A with its columns swapped according to $\vec{p}$, and
- PAP^{-1} is A with its rows and columns swapped according to $\vec{p}$.

Problems 4.3, exercise 12: The matrices of this form, where the entries are zero everywhere more than one location below the diagonal, are called "upper Hessenberg." The point is that, as long as there is no pivoting, solving such a system is only an $O(n^2)$ amount of work, instead of the usual $O(n^3)$ for general Gauss elimination. My routine is:

hessen.m

```
function x = hessen(A,b);
% HESSEN  Solve system A x = b by Gauss elimination and
%   back-substitution, without pivoting, on matrices A with
%   upper Hessenberg form    * * * * *
%                            * * * * *
%                      A =   0 * * * *
%                            0 0 * * *
%                            0 0 0 * *

[n m] = size(A);
if n ~= m, error('A not square'), end
if size(b,2) ~= 1, error('b not a column vector'), end
if n ~= size(b), error('b wrong size'), end

for i = 2:n
  for k = i+2:n
    if A(k,i-1) ~= 0, error('not upper Hessenberg form'), end
  end
  r = A(i,i-1) / A(i-1,i-1);             % the one multiplier per row
  A(i,i:n) = A(i,i:n) - r * A(i-1,i:n);  % implied loop over columns
  b(i) = b(i) - r * b(i-1);
end

x = bs1(triu(A),b);            % we've already got a back-sub routine
```

I tested it this way, which computes a relative error between the usual "`A\b`" result and the `hessen(A,b)` result:

```
>> n=10; A=randn(n); A=triu(A,-1); b=rand(n,1); v=A\b; norm(v-hessen(A,b))/norm(v)
ans =  4.2490e-15
```

Note that `triu(A)` extracts the upper triangular part of a matrix, and `triu(A,-1)` extracts the upper Hessenberg part; see `help triu`. I ran this test a few times, and the resulting relative errors were always in the 10^{-16} to 10^{-14} range, which I interpret as success.

I counted the number of exact multiplications needed to solve $n \times n$ size systems $Ax = b$. (The number of divisions is small by comparison, and the number of subtractions or additions is essentially the same as multiplications.) I got

$$((n-1) + (n-2) + \cdots + 2 + 1) + (n-1) = \frac{n(n-1) + 2(n-1)}{2} = \frac{(n+2)(n-1)}{2}$$

for multiplications in `hessen.m` itself, plus an additional

$$1 + 2 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2}$$

in `bs1.m`, which is called by `hessen.m`. The total number of multiplications to solve a system is $0.5(2n+2)(n-1) = (n+1)(n-1) = n^2 - 1$, which is much less than the $(1/3)n^3 + O(n^2)$ we had for Gauss elimination.

Problems 4.3, exercise 13: The algorithm is on page 180.

The number of divisions is $2(n-1) + 1 + n - 1 = 3n - 2$. The number of multiplications is one less, $3n-3$. The number of subtractions is the same as the number of multiplications. There are no additions. Thus the total number of arithmetic operations is $3n - 2 + 3n - 3 + 3n - 3 = 9n - 8$.

But mainly you should take home that solving a tri-diagonal system is an $O(n)$ operation, *and therefore dirt cheap compared to general Gauss elimination.*

Problems 6.1, exercise 1ac: The very simplest way is to use the built-in thing, though I encourage you to try your hand at one of the constructive methods. For the two parts I got:

```
>> polyfit([3 7],[5 -1],1)
ans =
  -1.5000   9.5000
>> polyfit([3 7 1 2],[10 146 2 1],3)
ans =
   5.1271e-16   5.0000e+00  -1.6000e+01   1.3000e+01
```

Thus $p(x) = -1.5x + 9.5$ for part **a** and $p(x) = 0x^3 + 5x^2 - 16x + 13$ (which is actually quadratic ... which is fine) for part **b**.

Problems 6.1, exercise 26: This was an all-MATLAB/OCTAVE gig:

```
>> x = [0 0.5 1];
>> p = polyfit(x,x-9.^(-x),2)
p =
  -0.88889   2.77778  -1.00000
>> roots(p)
ans =
   2.70985
   0.41515
```

Thus my guess for the solution is 0.41515, because the other root of the quadratic is outside of the interval $[0, 1]$. (*Then I did some quicky Newton's method on $x - 9^{-x} = 0$, with initial guess 0.41515, and saw that the root must be very close to 0.408004405374381.*)

Problems 6.1, exercise 16: Before writing code, let's expand the expression:

$$u = \prod_{j=1}^{1} d_j + \prod_{j=1}^{2} d_j + \prod_{j=1}^{3} d_j + \cdots + \prod_{j=1}^{n} d_j = d_1 + d_1\,d_2 + d_1\,d_2\,d_3 + \cdots + d_1\,d_2\,\cdots\,d_{n-1}\,d_n.$$

This has a nested form:

$$u = d_1\left(1 + d_2\left(1 + \cdots + d_{n-1}(1 + d_n)\ldots\right)\right).$$

The algorithm can now sum from the inside out:

sigmapi.m

```
function u = sigmapi(d);
% SIGMAPI  Compute a certain sum of products:
%   u = \sum_{i=1}^n \prod_{j=1}^i d_j

n = length(d);
if prod(size(d)) ~= n, error('d must be a vector'), end

u = d(n);
for i = n-1:-1:1
```

```
  u = d(i) * (1+u);
end
```

The code does exactly $n-1$ multiplications and the same number of additions; it cannot be better. I tested it this way:

```
>> x=[3 7 1 2]; sigmapi(x)
ans =  87
>> 3 + 3*7 + 3*7*1 + 3*7*1*2
ans =  87
```

Exercise 2: *Sorry about repeatedly saying* "built-in `lu`" *when I meant* "built-in `det`".

(a) Trying out `baddet.m`, it seems to work fine but it is slow on an 8×8 matrix:

```
>> A = magic(3); det(A), baddet(A)
ans = -360
ans = -360
>> abs(det(A) - baddet(A))/abs(det(A))
ans = 0
>> A = randn(8);  abs(det(A) - baddet(A))/abs(det(A))
ans =  1.8954e-16
```

The third input line takes 10 seconds or so to run. It is easy to check it is all spent computing `baddet(A)`.

I compute that if M_n is the number of multiplications in `baddet.m` on an $n \times n$ matrix then, because we have to compute n determinants of $(n-1) \times (n-1)$ minors, and then do an additional n scalar multiplications to combine the results,

$$M_n = n \cdot M_{n-1} + n,$$

a *recurrence relation*. Clearly $M_1 = 0$ and $M_2 = 2$ just by thinking about those cases separately. By using the recurrence relation, $M_3 = 9$, $M_4 = 40$, $M_5 = 205$, and so on.

Though it is not really as useful as the recurrence, we have

$$M_n = n! \left(1 + \frac{1}{2} + \frac{1}{3!} + \cdots + \frac{1}{(n-1)!} \right).$$

Interestingly, this means $M_n \sim (e-1)\, n!$ as $n \to \infty$.

(b) First we need to know that the determinant of an $n \times n$ triangular matrix is the product of its diagonal elements. (This can be shown by expanding-in-minors from the first row for lower triangular matrices, and from the first column for upper triangular matrices.) Thus $\det(L) = 1$ and $\det(U) = u_{11} \cdots u_{nn}$. Therefore if $A = LU$ we have $\det(A) = \det(L)\det(U) = 1 \cdot u_{11} \cdots u_{nn} = \prod_{j=1}^{n} u_{jj}$. On the other hand, the determinant of a permutation matrix is either $+1$ or -1 according to whether the number of row swaps which generated it (from the identity) is either even or odd, respectively. Thus when $PA = LU$ we have

$$\det(A) = (-1)^{(\text{number of row swaps})} \prod_{j=1}^{n} u_{jj}.$$

(c) It was easy to write `gooddet.m` except that the number of row swaps was hard to determine from the `P` that came from the built-in `lu`. So I went ahead and wrote my own little method to find the determinant of `P`. (Note, however, that if we have control of the LU decomposition then it is quite trivial to keep track of the number of row swaps. *I will be very forgiving on this issue when grading.*)

gooddet.m

```
function z = gooddet(A)
% GOODDET   z = gooddet(A)
%   Compute determinant by LU decomposition (Gauss elimination).
%   O(n^3) work instead of O(n!) like baddet.m
```

```
%   Essentially, this is the built-in method for determinant.

n = size(A,2);                              % get number of columns
if n == 1, z = A(1,1);   return,  end    % finish easy case of 1x1 matrix

[L,U,P] = lu(A);
s = detperm(P);         % replacement "s = det(P);" is fine, though "cheating"
z = s * prod(diag(U));
```

detperm.m

```
function s = detperm(P)
% DETPERM  Compute the determinant of a permutation matrix.
% (Merely needed to compute sign in gooddet.m.)

[n m] = size(P);
if n ~= m, error('not square'), end
if n == 1, s = 1; return, end

k = find(P(1,:));        % column index of "1" in first row
if length(k) > 1, error('not a permutation matrix'), end
s = (-1)^(k-1);
for j=2:n-1
  % replace P by the minor:
  if k == 1
    P = P(2:end,2:end);
  elseif k == size(P,2)
    P = P(2:end,1:end-1);
  else
    P = P(2:end,[1:k-1, k+1:end]);
  end
  k = find(P(1,:));
  if length(k) > 1, error('not a permutation matrix'), end
  s = s * (-1)^(k-1);
end
```

Properly implemented, the Gauss elimination done by the built-in `lu` call dominates the time in `gooddet.m`. Recall that Gauss elimination takes $(1/3)n^3 + O(n^2)$ multiplications. I conclude that on a $n \times n$ matrix,

$$\frac{(\#\text{ mults for } \texttt{baddet.m})}{(\#\text{ mults for } \texttt{gooddet.m})} = \frac{n!\left(1+\frac{1}{2}+\frac{1}{3!}+\cdots+\frac{1}{(n-1)!}\right)}{\frac{1}{3}n^3+O(n^2)} \sim \frac{3(e-1)n!}{n^3}.$$

For $n = 30$ this last fraction is about 5×10^{28}, which might as well be ∞. (*You won't often hear a math professor say such a thing ...*)