CMSC 460 - HW2

Gudjon Einar Magnusson

October 4, 2016

1 Matrix Vector Product

1.1 MatVecProd_a

```
%Using a pair of nested loops that access the matrix entries in column major order
function [p] = MatVecProd_a(A, x)
    [n, n] = size(A);
    p = zeros(n, 1);

    for c = 1:n
        for r = 1:n
            p(r) = p(r) + A(r, c)*x(c);
        end
    end
end
```

1.2 MatVecProd_b

```
%Using a pair of nested loops that access the matrixentries in row major order
function [p] = MatVecProd_b(A, x)
    [n, n] = size(A);
    p = zeros(n, 1);

    for r = 1:n
        for c = 1:n
            p(r) = p(r) + A(r, c)*x(c);
        end
    end
end
```

1.3 MatVecProd_c

```
%Using a single loop which access the matrix entries a column vector at a time
function [p] = MatVecProd_c(A, x)
    [n n] = size(A);
    p = zeros(n,1);

    for c = 1:n
        p = p + A(:, c)*x(c);
    end
end
```

1.4 MatVecProd_d

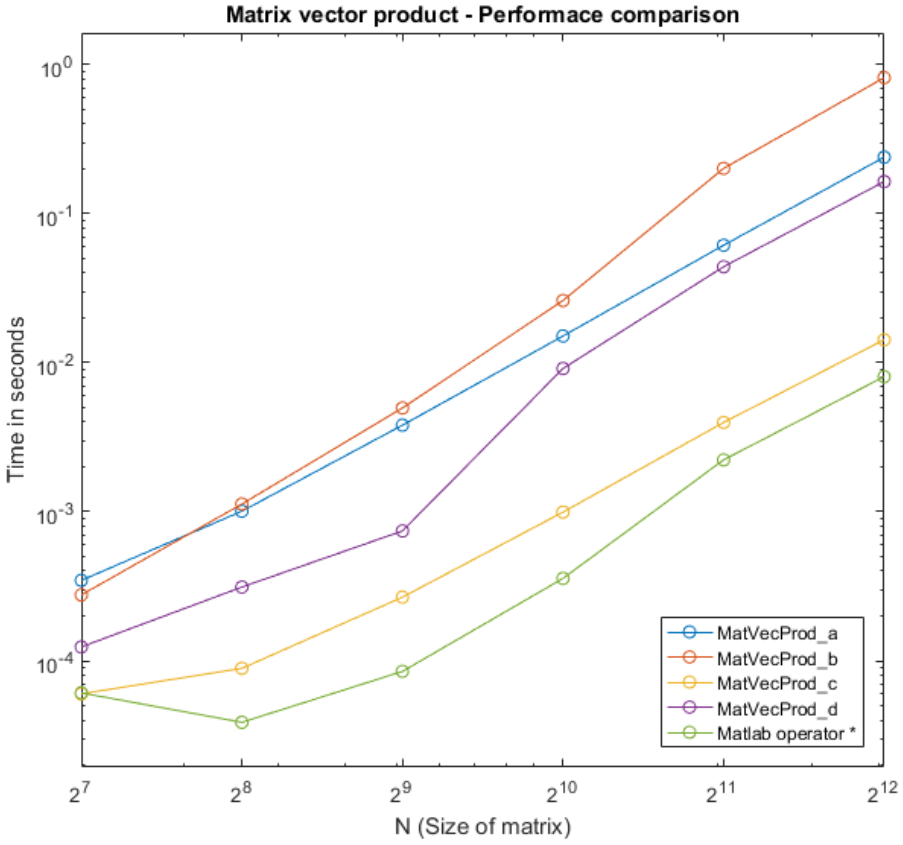
```
%Using a single loop which access the matrix entries a row vector at a time
function [p] = MatVecProd_d(A, x)
    [n n] = size(A);
    p = zeros(n,1);

    for r = 1:n
        p(r) = p(r) + A(r, :)*x;
    end
end
```

1.5 Evaluation

To compare the different implementations I measured the time it took to calculate a matrix vector product for square matrices of different sizes. For each implementation I tested a matrix of size 128, 256, 512,1024, 2048 and 4096. Figure 1.5 shows the result. We can see that the column-major implementations (a and c) are consistently faster than the row-major counter part. We can also see that the vectorized implementations (d and c) get an even greater speed boost. Finally we see that the built in Matlab operator is by far the fastest.

By indexing the matrix in column-major order we gain the benefit of memory proximity and optimal cache hits. This is a noticeable improvement but not as noticeable as changing to a vectorized implementation. By doing that we gain the benefit of the optimization that has been built in by the designers of Matlab. This is even more obvious when looking at the superior speed of the * operator.


Matrix vector product - Performace comparison
Time in seconds
N (Size of matrix)
MatVecProd_a
MatVecProd_b
MatVecProd_c
MatVecProd_d
Matlab operator *

2

See step by step solution in appendix.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{1}{6} & \frac{5}{14} & 1 \end{bmatrix} U = \begin{bmatrix} -6 & 3 & -15 \\ 0 & -7 & 3 \\ 0 & 0 & -14 \end{bmatrix} P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 3 \\ -1 \\ -2 \end{bmatrix}$$

This answer matches what the *lu* function returns, except that L is flipped horizontally. This is just a matter of notation.

3

Only thing added to the *lutx* function is a line to initialize *sig* as 1 and a line to flip the sign of *sig* every time a row is swapped.

```
function [L,U,p,sig] = lutx(A)
   [n,n] = size(A);
   p = (1:n)';
   sig = 1;

   for k = 1:n-1
      % Find index of largest element below diagonal in k-th column
      [r,m] = max(abs(A(k:n,k)));
      m = m+k-1;

      % Skip elimination if column is zero
      if (A(m,k) ~= 0)

         % Swap pivot row
         if (m ~= k)
            A([k m],:) = A([m k],:);
            p([k m]) = p([m k]);
            sig = sig*-1;
         end

         % Compute multipliers
         i = k+1:n;
         A(i,k) = A(i,k)/A(k,k);

         % Update the remainder of the matrix
         j = k+1:n;
         A(i,j) = A(i,j) - A(i,k)*A(k,j);
      end
   end

   % Separate result
   L = tril(A,-1) + eye(n,n);
   U = triu(A);
```

mydet gets the determinant of U by computing the product of its diagonal elements and to get the determinant of A it multiplies by *sig*.

```
function [ d ] = mydet( A )
    [~, U, ~, sig] = lutx(A);
    du = prod(diag(U));
    d = sig*du;
```

To verify than my implementation works I used the following script:

```
%% Test mydet
% Set of test matices
TestSet = cell(1, 5);
TestSet{1} = [1 -3 1; 2 -8 -15; -6 3 -15];
TestSet{2} = magic(3);
TestSet{3} = magic(5);
TestSet{4} = rand(5);
TestSet{5} = rand(10);

eps = 10^-8;
pass = 0;
for i = 1:numel(TestSet)
    M = TestSet{i};
    dm = det(M);
    my_dm = mydet(M);

    % Allow for errors less than eps
    pass = pass + (abs(dm-my_dm) < eps);
end

disp([int2str(pass), '/', int2str(numel(TestSet)), ' Passed']);
```

4

To measure the performance of each implementation I tested what order matrix it could solve in 10 seconds. To do this I ran the function repeatedly and increased the size of the matrix in increments of 50. Smaller increments would have given a more accurate result but took longer than I was willing to wait.

explutx
: Solved a Matrix of size 1150 in 11.26s

lutx
: Solved a Matrix of size 1500 in 10.31s

lu
: Solved a Matrix of size 12800 in 10.06s

As you can see the built in *lu* function is by far the fastest, it can handle a order of magnitude larger matrix in the same time as the home brew functions. Using the vectorized implementation gives a noticeable boost but nothing major.

```
function [L,U,p] = lutx(A)
    [n,n] = size(A);
    p = (1:n)';

    for k = 1:n-1
        % Find index of largest element below diagonal in k-th column
        r = -Inf;
        m = k;
        for i = k:n
            if abs(A(i,k)) > r
                r = abs(A(i,k));
                m = i;
            end
        end

        % Skip elimination if column is zero
        if (A(m,k) ~= 0)
            % Swap pivot row
            if (m ~= k)
                for i = 1:n
                    am = A(m, i);
                    ak = A(k, i);
                    A(k, i) = am;
                    A(m, i) = ak;
                end

                pk = p(k);
                p(k) = p(m);
                p(m) = pk;
            end

            % Compute multipliers
            for i = k+1:n
                A(i,k) = A(i,k)/A(k,k);
            end

            % Update the remainder of the matrix
            for i = k+1:n
                for j = k+1:n
                    A(i,j) = A(i,j) - A(i,k)*A(k,j);
                end
            end
        end
    end

    % Separate result
    L = tril(A,-1) + eye(n,n);
    U = triu(A);
```

To verify than my implementation works I used the following script:

```
%% Test explutx
% Set of test matices
TestSet = cell(1, 5);
TestSet{1} = [1 -3 1; 2 -8 -15; -6 3 -15];
TestSet{2} = magic(3);
TestSet{3} = magic(5);
TestSet{4} = rand(5);
TestSet{5} = rand(10);

eps = 10^-8;
pass = 0;
for i = 1:numel(TestSet)
    M = TestSet{i};
    [L1, U1, p1] = lutx(M);
    [L2, U2, p2] = explutx(M);

    % Compare each of the returned values
    % Allow for errors less than eps
    l_pass = sum(abs(L1(:) - L2(:))) < eps;
    u_pass = sum(abs(U1(:) - U2(:))) < eps;
    p_pass = isequal(p1, p2);

    pass_i = (l_pass && u_pass && p_pass);
    pass = pass + pass_i;
end

% Display fraction of tests passed
disp([int2str(pass), '/', int2str(numel(TestSet)), ' Passed']);
```

5

myinv finds the inverse of a matrix by solving $Ax_j = e_j$ where x_j is the j-th column of the inverse matrix and e_j is the j-th column of the identity matrix.

```
function [ X ] = myinv( A )
  [n,n] = size(A);
  [L,U,p] = lutx(A);
  E = eye(n);
  X = zeros(n);

  for i = 1:n
      e = E(:, i);
      % Permutation and forward elimination
      y = forward(L,e(p));
      % Back substitution
      x = backsubs(U,y);
      X(:, i) = x;
  end
end

function x = forward(L,x)
  % FORWARD. Forward elimination.
  % For lower triangular L, x = forward(L,b) solves L*x = b.
  [n,n] = size(L);
  x(1) = x(1)/L(1,1);
  for k = 2:n
     j = 1:k-1;
     x(k) = (x(k) - L(k,j)*x(j))/L(k,k);
  end
end

function x = backsubs(U,x)
  % BACKSUBS.  Back substitution.
  % For upper triangular U, x = backsubs(U,b) solves U*x = b.
  [n,n] = size(U);
  x(n) = x(n)/U(n,n);
  for k = n-1:-1:1
     j = k+1:n;
     x(k) = (x(k) - U(k,j)*x(j))/U(k,k);
  end
end
```

To verify than my implementation works I used a similar script as before. Omitted to save paper.