Selected Solutions for Exercises in

Numerical Methods with MATLAB: Implementations and Applications

Gerald W. Recktenwald

Chapter 8

Solving Systems of Equations

The latest version of this PDF file, along with other supplemental material for the book, can be found at `www.prenhall.com/recktenwald`.

8.6 Manually solve $QRx = b$ for x where

$$Q = \begin{bmatrix} 1/\sqrt{2} & 0 & -1/\sqrt{2} \\ 0 & 1 & 0 \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix} \qquad R = \sqrt{2}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 2 \\ 2\sqrt{2} \\ 4 \end{bmatrix}$$

Hint: Take advantage of the properties of Q identified in the preceding problem.

Solution: Before any detailed (i.e. element-by-element) computations are performed, manipulate the given equation as products of matrices and vectors. From Exercise 8.3 we know that $Q^TQ = I$, so that $Q^T = Q^{-1}$, and Q is an *orthogonal* matrix.

Multiplying both sides of $QRx = b$ by Q^T and simplifying gives

$$Q^T\ QRx = Q^T\ b \qquad \Longrightarrow \qquad Rx = Q^Tb$$

For convenience, let $z = Q^Tb$ (z is a 3×1 column vector). If the z vector is known, we can solve $Rx = z$ for z with backward substitution. Given the preceding manipulations we can obtain an "easy" solution to $QRx = b$ with the following two steps

1. Evaluate $z = Q^Tb$
2. Solve $Rx = z$ with backward substitution

This completes the solution strategy. All that remains is performing the computations.

Use the row view of the matrix-vector product to evaluate Q^Tb

$$z = Q^Tb = \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 0 & 1 & 0 \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 2 \\ 2\sqrt{2} \\ 4 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{2} + 0 + 4/\sqrt{2} \\ 0 + 2\sqrt{2} + 0 \\ -2/\sqrt{2} + 0 + 4/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 6/\sqrt{2} \\ 2\sqrt{2} \\ 2/\sqrt{2} \end{bmatrix}$$

Next solve

$$\sqrt{2}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6/\sqrt{2} \\ 2\sqrt{2} \\ 2/\sqrt{2} \end{bmatrix}$$

with backward substitution. For convenience, first divide through by the factor of $\sqrt{2}$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 6/\sqrt{2} \\ 2\sqrt{2} \\ 2/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 6/2 \\ 2 \\ 2/2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

Work from x_3 up to x_1 (backward substitution):

$$\begin{aligned} x_3 &= 1 \\ x_2 + x_3 &= 2 \quad \Longrightarrow x_2 = 2 - x_3 = 1 \\ x_1 + x_2 + x_3 &= 3 \quad \Longrightarrow x_1 = 3 - x_2 - x_3 = 1 \end{aligned}$$

Therefore the solution is $x = [1, 1, 1]^T$. The following MATLAB statements double-check the manual solution. First, define the Q, R, and b vectors as given

```
>> s2 = sqrt(2);
>> Q = [1/s2 0 -1/s2; 0 1 0; 1/s2 0 1/s2]
Q =
    0.7071         0   -0.7071
         0    1.0000         0
    0.7071         0    0.7071

>> R = s2*[1 1 1; 0 1 1; 0 0 1]
R =
    1.4142    1.4142    1.4142
         0    1.4142    1.4142
         0         0    1.4142

>> b = [2; 2*s2; 4]
b =
    2.0000
    2.8284
    4.0000
```

As a check, verify that $Q^T Q = I$

```
>> Q'*Q - eye(3)
ans =
   1.0e-15 *
   -0.2220         0    0.0224
         0         0         0
    0.0224         0   -0.2220
```

Now, obtain the solution by computing z, and then computing x by backward substitution.

```
>> z = Q'*b
z =
    4.2426
    2.8284
    1.4142

>> x(3) = z(3)/R(3,3)
x =
         0         0    1.0000

>> x(2) = ( z(2) - R(2,3)*x(3) ) / R(2,2)
x =
         0    1.0000    1.0000

>> x(1) = ( z(1) - R(1,2)*x(2) - R(1,3)*x(3) ) / R(1,1)
x =
    1.0000    1.0000    1.0000

>> x = x(:)      %  convert x to column vector
x =
    1.0000
    1.0000
    1.0000
```

We used to the most general form of the backward substitution steps, and did not exploit the fact that all of the upper triangular elements of R are equal to one. Conversion of `x` from a row vector to a column vector with `x = x(:)` is necessary because the `x` was first created as a row vector with the statement `x(3) = z(3)/R(3,3)`.

8.12 Starting with the code in the `GEshow` function, develop a `GErect` function that performs Gaussian Elimination only (no backward substitution) for rectangular ($m \times n$) matrices. The `GErect` function should return $\tilde{A}$, the triangularized coefficient matrix, and $\tilde{b}$, the corresponding right hand side vector. Use the `GErect` function to solve Exercise 11.

Partial Solution: The prologue and partial code for the `GErect` function is shown below. The only substantial difference between `GEshow` and `GErect` is that `GErect` does not perform backward substitution.

```
function [At,bt] = GErect(A,b,ptol)
% GErect  Gauss elimination for rectangular coefficient matrices
%         No pivoting is used.
%
% Synopsis:  [At,bt] = GErect(A,b)
%            [At,bt] = GErect(A,b,ptol)
%
% Input:     A,b  = coefficient matrix and right hand side vector
%            ptol = (optional) tolerance for detection of zero pivot
%                   Default:  ptol = 50*eps
%
% Output:  At = triangularized coefficient matrix obtained by elimination
%          bt = right hand side vector transformed by the same row
%                 operations necessary to obtain At

if nargin<3, ptol = 50*eps;  end
[m,n] = size(A);
nb = n+1;   Ab = [A b];    %  Augmented system
fprintf('\nBegin forward elimination with Augmented system:\n');  disp(Ab);

... more code goes here

At = Ab(:,1:n);  bt = Ab(:,nb);
```

8.16 Write an `lsolve` function to solve $Ax = b$ when A is a lower triangular matrix. Test your function by comparing the solutions it obtains with the solutions obtained with the left division operator.

Partial Solution: The `lsolve` function is listed below. The reader is left to complete the Exercise by devising appropriate tests for `lsolve`.

```
function x = lsolve(L,b)
% lsolve  solves the lower triangular system Lx = b
%
% Synopsis:   x = lsolve(L,b)
%
% Input:      L = lower triangular coefficient matrix
%             b = right hand side vector
%
% Output:     x = solution vector

[m,n] = size(L);
if m~=n,  error('L matrix is not square'); end

x = zeros(n,1);       %  preallocate x for speed
x(1) = b(1)/L(1,1);   %  begin forward substitution
for i=2:n
  x(i) = (b(i) - L(i,1:i-1)*x(1:i-1))/L(i,i);
end
```

8.21 (3) The inverse matrix A satisfies $AA^{-1} = I$. Using the column view of matrix–matrix multiplication (see Algorithm 7.5 on page 327) we see that the j^{th} column of A^{-1} is the vector x such that $Ax = e_{(j)}$, where $e_{(j)}$ is the j^{th} column of the identity matrix (e.g., $e_3 = [0,\ 0,\ 1,\ \ldots, 0]^T$). By solving $Ax = e_{(j)}$ for $j = 1, \ldots, n$ the columns of A^{-1} can be produced one at a time.

(a) Write a function called `invByCol` that computes the inverse of an $n \times n$ matrix one column at a time. Use the backslash operator to solve for each column of A^{-1}.

(b) Use the estimates in Table 8.1 to derive an order-of-magnitude estimate for how the flop count of `invByCol` depends on n for an $n \times n$ matrix.

(c) Verify the estimate derived in part (b) by measuring the flop count of `invByCol` for matrices of increasing size. Use `A = rand(n,n)` for `n` $= 2, 4, 8, 16, 32, \ldots, 128$. Compare the flop count of `invByCol` with those of the built-in `inv` command. Note that the order-of-magnitude estimate will only hold as n becomes large. Users of MATLAB version 6 will not be able to use the `flops` function to measure the flops performed by `inv`. In that case, use the estimate that matrix inversion can be performed in $\mathcal{O}(n^3)$ flops.

Solution (a): A is given. The objective is to solve a sequence of problems $Ax = e_{(j)}$, $j = 1, \ldots, n$. Each x becomes a column of A^{-1}. Doing so requires a loop, and a way to define $e_{(j)}$. The following statements do the job

```
for j=1:n
  e = zeros(n,1);  e(j) = 1;
  Ai(:,j) = A\e;
end
```

where `n` is the number of rows in `A`. The expression `Ai(:,j) = A\e` stores the solution to $Ax = e_{(j)}$ in the j^{th} column of `Ai`. The efficiency of the loop can be improved by preallocating memory for `Ai`, and using a fixed zero vector `z = zeros(n,1)` instead of creating a new vector on each pass through the loop. These improvements, along with provisions for input and output and some basic error checking are incorporated into the `invByCol` function listed below.

```
function Ai = invByCol(A)
% invByCol  Compute matrix inverse of a matrix by columns
%
% Synopsis:  Ai = invByCol(A)
%
% Input:     A = square (n by n) matrix
%
% Output:   Ai = inverse of A, if it exists

[m,n] = size(A);
if m~=n,  error('Inverse is defined only for square matrices');    end

Ai = zeros(n,n);      %  pre-allocate for speed
z = zeros(n,1);       %  temporary vector
for j=1:n
  e = z;  e(j) = 1;   %  reset column of I
  Ai(:,j) = A\e;      %  Solve for jth column of A^(-1)
end
```

A simple test of `invByCol` is

```
>> A = rand(5,5);  Ai = invByCol(A);  E = Ai*A - eye(5)
E =
   1.0e-15 *
   -0.1110   -0.1661   -0.2774   -0.0659   -0.1029
   -0.0513    0.2220    0.0081   -0.0912   -0.1724
    0.1372    0.1194    0.4441    0.3129    0.0386
   -0.0205   -0.0691   -0.1924         0    0.0978
   -0.0833    0.0571    0.0031    0.0324         0

>> norm(A,1)
ans =
    3.3992

>> norm(E,1)
ans =
   9.2514e-16
```

Since $\|E\|_1 \ll \|A\|_1$ and A is reasonably well-conditioned (How do we know?), the `invByCol` function appears to be working. Note that the L_1 norm is chosen for efficiency. Both the L_∞ and L_2 norms would give equivalent results. The L_∞ norm would take less flops than the L_1 norm. Both L_1 and L_∞ norms are significantly more efficient than the L_2 norm. For a 5×5 matrix the efficiency differences are irrelevant, however.

Solution (b): Each solution of $Ax = e_{(j)}$ takes $\mathcal{O}(n^3/3)$ flops. There are n columns of A^{-1} so the `invByCol` function takes $n \times \mathcal{O}(n^3/3) = \mathcal{O}(n^4/3)$ flops for an $n \times n$ matrix A. This is an expensive way to compute A^{-1}.

Solution (c): The `demoInvByCol` function (listed below) measures the flops performed by the `invByCol` function and the built-in `inv` function. These functions are applied to a sequence of random $n \times n$ matrices is generated for `n = [2 4 8 16 32 64 128]`. The elements of matrix A are unimportant as long as A is nonsingular. It turns out that the matrices generated by the built-in `rand` function are rarely singular. The flop counts are measured with the built-in `flops` function. Note that these measurements will yield *zero* flops for MATLAB version 6 and later. Thus, the `demoInvByCol` function is useful only to users of MATLAB version 5 and earlier.

The flop count for each n is saved for plotting and analysis. The `powfit` function is used to obtain the least squares fit (see "Fitting Lines to Apparently Nonlinear Functions" in Chapter 9) to

$$f = c_1 n^{c_2}$$

where f is a vector of measured flop counts and n is the vector of matrix dimensions. The c_2 exponent should be 4 for the `invByCol` function because the flop count grows as $\mathcal{O}(4)$. (See solution to part (b), above.) For the built-in `inv` function the c_2 exponent should be 3 because the `inv` function computes the inverse via LU factorization, flop count grows as $\mathcal{O}(3)$.

The `n(3:end)` and `f(n:end)` vectors are passed to `powfit`. The `3:end` index subexpression selects the third through last elements of the vector. This improves the estimate of c_2 because the order of magnitude estimates of the flop counts only applies for large n.

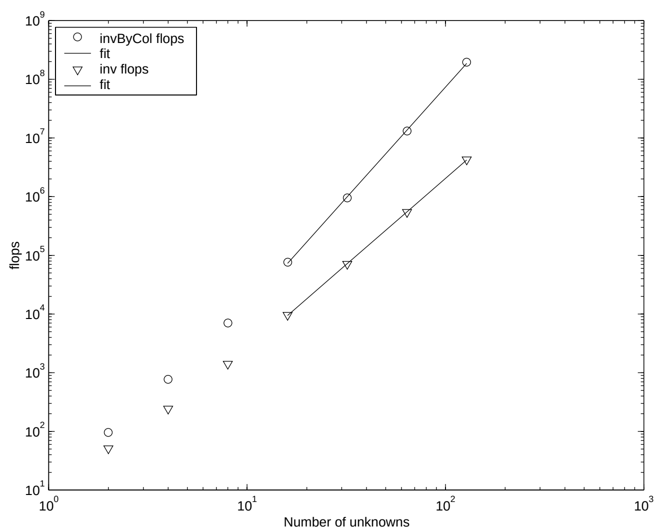
```
function demoInvByCol
% demoInvByCol  Measure flop count behavior of invByCol

% --- Count flops for invByCol and built-in inv functions
n = [2 4 8 16 32 64 128];      %  Sizes of problems to run
for i=1:length(n)
  A = rand(n(i),n(i));
  flops(0);   Ai = invByCol(A);  fcol(i) = flops;
  flops(0);   Ai = inv(A);       fInv(i) = flops;
end

% --- Use least squares fits to obtain exponent of flops relationship
c = powfit(n(4:end),fcol(4:end));
cinv = powfit(n(4:end),fInv(4:end));

% --- Evaluate least squares fits and plot
nfit = n(4:end);
fcolfit = c(1)*nfit.^c(2);
finvfit = cinv(1)*nfit.^cinv(2);
loglog(n,fcol,'o',nfit,fcolfit,'-',n,fInv,'v',nfit,finvfit,'-')
legend('invByCol flops','fit','inv flops','fit',2)
xlabel('Number of unknowns');   ylabel('flops');

% --- Print summary
fprintf('\nFlop counts:\n\n');
fprintf('       n          invByCol        inv\n');
for i=1:length(n)
  fprintf('     %4d    %10d    %10d\n',n(i),fcol(i),fInv(i));
end
fprintf('  exponent       O(%3.1f)        O(%3.1f)\n',c(2),cinv(2));
```

```
function c = powfit(x,y)
% expfit     Least squares fit of data to y = c1*x^c2
%
% Synopsis:  c = powfit(x,y)
%
% Input:     x,y = vectors of independent and dependent variable values
%
% Output:    c = vector of coefficients: y = c(1)*x^c(2)

if length(y)~=length(x),  error('Dimensions of x and y are not compatible');  end

ct = linefit(log(x(:)),log(y(:)));  %  Line fit to transformed data
c = [exp(ct(2))  ct(1)];            %  Extract parameters from transformation
```

Running `demoInvByCol` produces the following output and the plot below.

```
>> demoInvByCol

Flop counts:

        n          invByCol         inv
        2              96            51
        4             768           242
        8            7040          1412
       16           76128          9598
       32          952768         70942
       64        13163520        545040
      128       194685952       4276376
   exponent        O(3.8)        O(2.9)
```

The `invByCol` function is clearly less efficient than the built-in `inv` function. The measured flops exponent for the `inv` function is close to the theoretical value of 3. The measured flops exponent for `invByCol` is somewhat less than the expected value of 4. The discrepancies are likely caused by the relatively small values of n used, and the way that MATLAB counts the flops for the backslash operator.



Exercise 8–21. Flop counts for `invByCol` and `inv`.

Extra Credit: The most costly phase of `invByCol` is repeatedly solving $Ax = e_{(j)}$. Rewrite the `invByCol` function to use LU factorization to reduce the computational work. Factor matrix A once, then (inside a loop) use triangular solves to produce each column of I. Compare the flop count of your new function with the original version of `invByCol`.

The extra credit solution is implemented in the `invByColLU` and `demoInvByColLU` functions (neither are listed here). Running `demoInvByColLU` gives

```
>> demoInvByColLU

Flop counts:

       n           invByCol         inv        invByColLU
         2             100           51             35
         4             752          242            226
         8            7168         1444           1588
        16           77216         9594          11816
        32          950400        70836          90960
        64        13186432       545124         713376
       128       194873088      4275098        5649728
  exponent          O(3.7)       O(2.9)         O(3.0)
```

The `invByColLU` function obtains the exact flops exponent value of 3 that is predicted by the order of magnitude work estimates. The built-in `inv` function uses the ideas embodied in `invByColLU` to compute $A^{(-1)}$.

8.25 An alternative way to resolve the singularity in the 3×3 coefficient matrix of Example 8.5 is to modify the elements of the matrix. Write a *trivial* equation involving v_b, v_c, and v_d that has the solution $v_d = 0$. Use this equation to replace the equation for v_d in the 3×3 system in Example 8.5. What is the value of V_{out} for $R_1 = R_3 = R_4 = R_5 = 10\,k\Omega$, $R_2 = 20\,k\Omega$ and $V_{\text{in}} = 5\,V$.

Partial Solution: The trivial equation with the solution $v_d = 0$ is $(0)v_b + (0)v_c + (1)v_d = 0$, or $v_d = 0$.

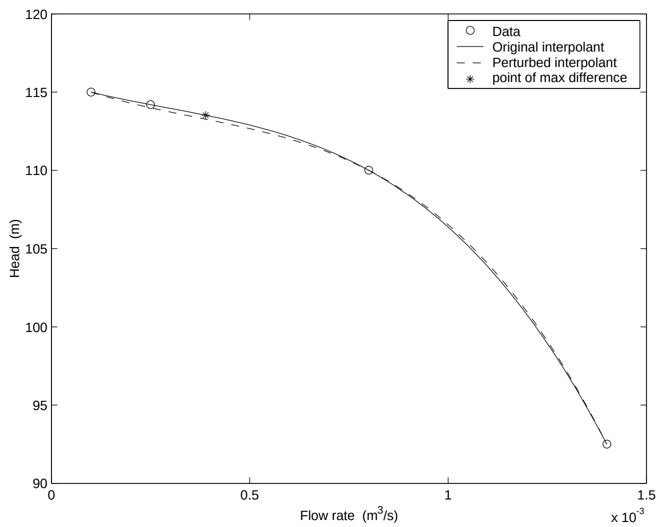
8.33 Use the `pumpCurve` function developed in Exercise 32 to study the effect of perturbing the input data. Specifically, replace the second h value, $h = 114.2$, with $h = 114$, and re-evaluate the coefficients of the cubic interpolating polynomial. Let $\tilde{c}$ be the coefficients of the interpolating polynomial derived from the perturbed data, and let c be the coefficients of the polynomial derived from the original data. What is the relative difference, $(\tilde{c}_i - c_i)/c_i$, in each of the polynomial coefficients? Evaluate and plot $h(q)$ for the two cubic interpolating polynomials at 100 data points in the range $\min(q) \le q \le \max(q)$. What is the maximum difference in h for the interpolants derived from the original and the perturbed data? Discuss the practical significance of the effect perturbing the data on the values of c and the values of h obtained from the interpolant.

Partial Solution: The computations are carried out in `pumpPerturb` (not listed here). Running `pumpPerturb` gives the following output and plot..

```
>> pumpPerturb

Coefficients of cubic interpolant in descending powers of q:

    i        c(i)              ct(i)          percent diff
    1    -1.1870738e+10    -1.3978775e+10        17.8
    2     1.0361305e+07     1.5209790e+07        46.8
```

```
      3    -7.8023933e+03    -1.0627163e+04       36.2
      4     1.1568850e+02     1.1592460e+02        0.2

  Maximum difference of h values =  0.253 (m) occurs at q =     3.89e-04
  Condition numbers:  kappa(A) =    2.7e+10;      kappa(At) =    2.7e+10
```

`c(i)` are the coefficients of the cubic polynomial derived from the unperturbed data. `ct(i)` are the coefficients derived from the perturbed data. Although the coefficients have very large differences in magnitude, the values of the interpolants have a maximum difference of only 0.25 m in head. The perturbation has no significant effect on the condition number of the Vandermonde system.



Exercise 8–33. Interpolants from original and perturbed h data.