Numerical Algorithms. Assignment 2.
Krukovich Palina, a12134146

Paper-and-Pencil Exercises

1. Prove the properties of the M_k elementary elimination matrix:

(a) M_k is non-singular

Matrix M_k is lower-triangular matrix containing ones on the diagonal and zeros everywhere off-diagonal except the k-th column below the diagonal.

Thus, we can write the definition of matrix M_k as:

$M_k = I - m_k e_k^T$, where $m_k = [0, 0, \dots, 0, m_{k+1}, \dots m_n]^T$

e_k is a k-th column of I (identity matrix)

The inverse of lower-triangular matrix that has ones all over the main diagonal is calculated by changing the sign of the off-diagonal elements.

$$M_k^{-1} = -(M_k - I) + I = -M_k + 2I = -(I - m_k e_k^T) + 2I =$$
$$= I + m_k e_k^T$$

Given: $M_k = I - m_k e_k^T$

$M_k^{-1} = I + m_k e_k^T$

Prove: $M_k M_k^{-1} = M_k^{-1} M_k = I$

1. $M_k M_k^{-1} = I$

$$(I - m_k e_k^T)(I + m_k e_k^T) = I$$
$$I^2 + I m_k e_k^T - m_k e_k^T I - (m_k e_k^T)^2 = I$$

* $I^2 = I$ * $IA = AI = A$

$$I - (m_k e_k^T)^2 = I$$

* $m_k e_k^T$ is lower-triangular with zero-diagonal. Multiplication of this matrix by itself always causes one of the numbers in scalar multiplication to be zero. So the resulting matrix is always zero.

$I - 0 = I \rightarrow$ true

2. $M_k^{-1} M_k = I$

$$(I + m_k e_k^T)(I - m_k e_k^T) = I$$
$$\underbrace{I^2}_{I} - I \cancel{m_k} e_k^T + m_k \cancel{e_k}^T I - \overbrace{(m_k e_k^T)^2}^{=0} = I$$

Given all the properties proved in 1 true

3. $M_k M_k^{-1} = I$ $M_k^{-1} M_k = I \rightarrow$ non-singular

$M_k M_k^{-1} = M_k^{-1} M_k = I$

(b) $M_k \cdot M_j$ = union of $\mathcal{M}_k$ and $\mathcal{M}_j$, $k \le j$

$$\mathcal{M}_k \cdot \mathcal{M}_j = (I - m_k e_k^T)(I - m_j e_j^T) =$$
$$\underset{I=}{=} I^2 - I m_j e_j^T - m_k e_k^T I + \underbrace{m_k e_k^T \cdot m_j e_j^T}_{=0} =$$
$$= I - m_j e_j^T - m_k e_k^T$$

* $I^2 = I$ * $IA = AI = A$

* Multiplication of $m_k e_k^T$ and $m_j e_k^T$ always produces zero-matrix.

$M_k \mathcal{M}_j = I - m_k e_k^T - m_j e_j^T$, which means that product of M_k and M_j is essentially their union.

Example: $M_1 = \begin{bmatrix} 1 & 0 & 0 \\ \boxed{-2} & 1 & 0 \\ \boxed{1} & 0 & 1 \end{bmatrix}$ $\mathcal{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \boxed{3} & 1 \end{bmatrix}$

$\mathcal{M}_1 \mathcal{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ \boxed{-2} & 1 & 0 \\ \boxed{1} & \boxed{3} & 1 \end{bmatrix}$

(2) Matrix A is singular if $\det(A) = 0$. Is the determinant of matrix a good indicator of near singularity? Does the magnitude of a nonzero determinant give any information about how close to singular the matrix is.

Proof by example:

Property of determinat: $\det(\alpha A) = \alpha^n \det(A)$

1. Let's take $n = 1000$ and $\alpha = 0{,}1$, $A = I$
2. $\det(A) = 1$
3. $\det(\alpha A) = 1 \cdot 10^{-1000} \approx 0$

Determinant of a matrix is sensitive to scaling. Thus, we can not use it to determine how close to singular the matrix is.

We should use condition number: $cond(A) = ||A|| \cdot ||A^{-1}||$

$cond(A) = 1$ $cond(\alpha A) = 1$

Programming Exercises

Introduction

The main goal of this part is to get experience with condition numbers and perturbation theory, investigate methods of condition numbers estimation and implement the LINPACK algorithm for condition numbers estimation.

Octave version: 6.3.0.

Part 1 – A Condition Estimator

In the first part we implement LINPACK condition estimation algorithm. The condition number of a matrix A is calculated with the following formula:

$$cond(A) = ||A|| \cdot ||A^{-1}||$$

The condition estimation algorithm solves the problem of long A^{-1} computing, it estimates the inverse matrix at low cost by optimally choosing vector y, such that $||A^{-1}|| \geq ||z||/||y||$. Vector y is chosen such that this ratio is as much as possible to get a reasonable estimate of $||A^{-1}||$.

The algorithm first factors $A = LU$ to get lower and upper triangular matrices. A major part of the algorithm computes vector w, such that $U^T w = e$. The values of e are computed so that $||w||$ is large.

```
validateattributes(A, {'numeric'}, {'square'});
% Get size of square matrix A - n x n
n = columns(A);
% Factor A = LU
[L, U] = lu(A);
```

First, we implement a for-loop which iterates from 1 to n to compute the elements of $w_1, \dots, w_n$. The first step in the loop to set the value of e_k. There are two choices: $e_k^+ = sign(-t_k) \cdot |e_k|$ and $e_k^- = -e_k^+$. I calculate only the positive variant of e^k, later if negative e_k^- is needed, it is just negated. Also, due to possible rescaling of e_k the value is multiplied by the absolute of itself, so we don't lose the current scale. The next step in the algorithm is to check if rescaling is required to avoid overflows. If $|U_{kk}| < |e_k - t_k|$ than the computed w_k will be more than one. To avoid this, we scale the e_k, such that $|w_k| \leq 1$. Along with w we need to also rescale the t vector and e^k value.

```
% Solve U' * w = e
ek = 1.;
t = zeros(n, 1);
for k = 1 : n
  % Compute ek with respect to possible rescaling
  if (t(k, 1) != 0)
    ek = sign(-t(k, 1)) * abs(ek);
  endif
  % Rescale ek (and t respectively) if needed so that |wk| <= 1
  if (abs(U(k, k)) < abs(ek - t(k,1)))
    scale = abs(U(k, k)) / abs(ek - t(k, 1));
    t(1 : n, 1) = t(1 : n, 1) * scale;
    ek = ek * scale;
  endif
```

Using the computed value of e_k the quantities for each step are calculated by the following formulas: $t_k^+ = e_k^+ - t_k$ and $t_k^- = e_k^- - t_k$. The values of positive and negative w for current step are assigned to 1 if the value of U_{kk} is zero or otherwise computed as follows: $w_k^+ = (e_k^+ - t_k)/U_{kk}$ and $w_k^- = (e_k^- - t_k)/U_{kk}$.

```
  % Compute tk+ = ek+ - tk
  tp = ek - t(k, 1);
  % Compute tk- = ek- - tk
  tn = -ek - t(k, 1);
  % Compute wk+ and wk-
  if ( U(k, k) != .0 )
    % Compute wk+ = (ek+ - tk) / Ukk
    wk = tp / U(k, k);
    % Compute wk- = (ek- - tk) / Ukk
    wkn = tn / U(k, k);
  else
    wk = 1.;
    wkn = 1.;
  endif
```

The final decision on which w to use is made in the end of the loop when all other parameters for this step are computed. To predict the possible growth of $||w||$ the sums of all t_j^+ and t_j^- are calculated for $j = k..n$ and the largest determines the sign of w to choose. The final vector w is stored in t array, because we don't use the old t values in the algorithm, so to avoid addition memory use the t vector in the end contains the w values.

```
% Compute wk
if (k < n)
  % Calculate sum(tj+) and sum(tj-) with j = k..n
  sum_tp = abs(tp);
  sum_tn = abs(tn);
  for j = k + 1 : n
    sum_tn = sum_tn + abs(t(j, 1) + U(k, j) * wkn);
    sum_tp = sum_tp + abs(t(j, 1) + U(k, j) * wk);
  endfor
  % Define growth of ||w|| by comapring sum(tj+) and sum(tj-) with j = k..n
  if (sum_tp < sum_tn)
    % If sum(tj-) is larger than wk = wk-
    for j = k + 1 : n
      t(j, 1) = t(j, 1) + U(k, j) * wkn;
      wk = wkn;
    endfor
  else
    % If sum(tj+) is larger than wk = wk+
    for j = k + 1 : n
      t(j, 1) = t(j, 1) + U(k, j) * wk;
    endfor
  endif
endif
t(k, 1) = wk;
```

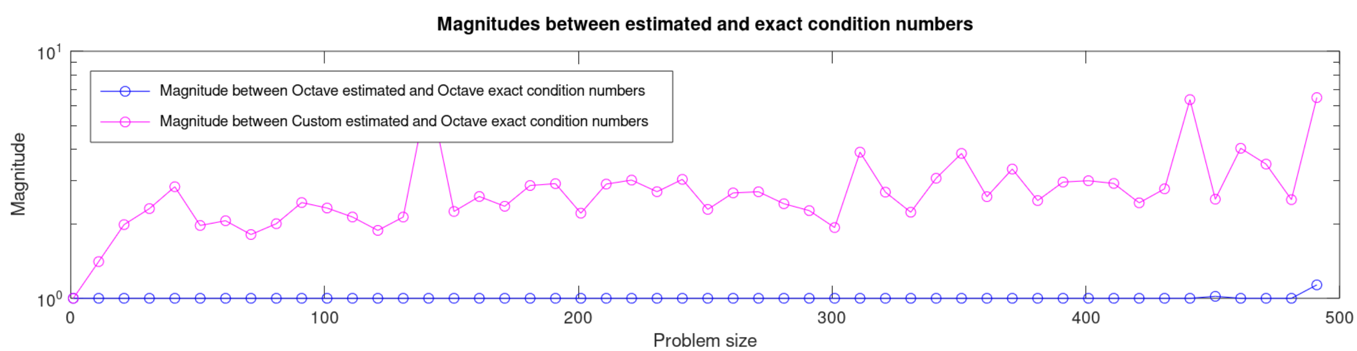
After solving $U^T w = e$, we follow the remaining steps of the algorithm to find the value of condition number estimation.

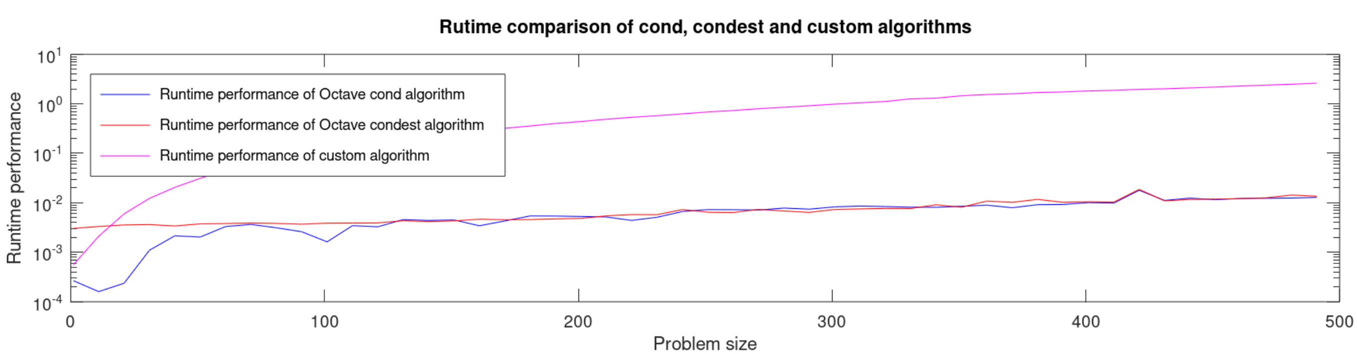
```
compute ‖A‖
factor  A = LU
solve  U^T w = e , choosing  e_k  as described
solve  L^T y = w
solve  Lv = y
solve  Uz = v
RCOND = ‖y‖/(‖A‖ ‖z‖) .
```

```
% Solve L' * y = w (with w stored in t)
y = L' \ t;
% Solve L * v = y
v = L \ y;
% Solve U * z = v
z = U \ v;
% Compute RCOND
RCOND = norm(y, 1) / (norm(A, 1) * norm(z, 1));
% Compute condition number
c = 1 / RCOND;
```

Implemented estimator runs tests successfully for Octave version 6:

The figures below show the comparison of **cond**, **condest** and **est_cond** algorithms in terms of runtime and quality of condition estimation. Magnitude is used to find how much estimated values differ from exact values. As we can see on the first figure, Octave condition estimation function gives good approximations for all the problem sizes, but the custom estimator has inaccuracies. Also, the runtime performance of the custom estimator increases with increasing problem sizes, the complexity of the algorithm is close to $O(n^2)$.





Part 2 – Average Case Perturbation Errors

The relative error in the solution of a linear system is bounded as:

$$\frac{||\Delta x||}{||x||} \leq cond(A)\left(\frac{||\Delta b||}{||b||} + \frac{||E||}{||A||}\right)$$

The Δb and E are randomly generated such that $||\Delta b||$ and $||E||$ are equal to 10^{-8}. To do so we first generate random vector and matrix for $||\Delta b||$ and E. Then we multiply generated values by 10^{-8} and divide by norm. The derivation, why it gives the correct result is presented below for random vector Δb.

$$||\Delta b|| = \sum_{i=1}^{n} \Delta b_i$$

Compute vector $\Delta b'$ such that its norm is equal to 10^{-8}

$$\Delta b' = \frac{\Delta b \cdot 10^{-8}}{||\Delta b||}$$

$$||\Delta b'|| = \sum_{i=1}^{n} \Delta b'_i = \sum_{i=1}^{n} \frac{\Delta b_i \cdot 10^{-8}}{||\Delta b||} = \frac{10^{-8}}{||\Delta b||} \cdot \sum_{i=1}^{n} \Delta b_i = \frac{10^{-8}}{||\Delta b||} \cdot ||\Delta b|| = 10^{-8}$$

Many random A and b are calculated for each step. In my case its 10. For each step we compute the average of left- and right-hand side of the bound.

The figure below shows average values of right- and left-hand side of the bound. Here we can see how perturbations of the input data affect the result. The relative change in the solution is bounded by the sum of relative changes in the input multiplied by the condition number of A. The relative error in the solution is significantly lower than the upper bound, because of the amplification of the upper bound with the value of $cond(A)$. This why we can say that the solution is not the worst case. The high value of condition number means that matrix A is close to being singular.

We also see that change in the solution is proportional to the change in input (the sum of relative changes for b and A).