Department of Physics
Umeå University

September 27, 2018

Matrix computations and applications

Assignment 1

Gustav Nystedt - oi14gnt

Department of computer science

Supervisors: Niclas Börlin & Andrii Dmytryshyn

Contents

1 Introduction 2

2 Theory 2
2.1 Laws of matrix addition 2
2.1.1 Commutative law (proof) 2
2.1.2 Distributive law (proof) 2
2.1.3 Associative law (proof) 2
2.2 Laws of matrix multiplication 3
2.2.1 Distributive law from the left (proof) 3
2.3 Block multiplication 3
2.4 Back substitution 3
2.5 Elimination without pivoting 4
2.6 Block elimination 4

3 Algorithms 5
3.1 Back substitution 5
3.1.1 Pseudocode 5
3.1.2 Tests 5
3.2 Triangular matrix inversion 6
3.2.1 Pseudocode 6
3.2.2 Tests 6

4 Applications 7
4.1 Early collision warning 7
4.1.1 Setting up the problem 7
4.1.2 Pseudocode 9
4.1.3 Tests 9

A Matlab Code 12
A.1 backsubst.m 12
A.2 triinv.m 14
A.3 collision.m 16
A.4 checkLevel.m 17
A.5 MainAero.m 19

1 Introduction

This assignment is divided into three main parts. First there is a theory part where we examine some elementary properties of linear systems and matrices. Secondly we look at some methods for solving linear systems using some useful algorithms and implementing them in Matlab. Lastly we look a slightly bigger problem where we get the chance to apply linear algebra to construct an early collision warning system for air crafts. This is also implemented using Matlab.

2 Theory

2.1 Laws of matrix addition

Definition. The three matrix addition laws are:

$$A + B = B + A \quad (\textit{commutative law}) \tag{1}$$

$$c(A + B) = cA + cB \quad (\textit{distributive law}) \tag{2}$$

$$A + (B + C) = (A + B) + C \quad (\textit{associative law}) \tag{3}$$

2.1.1 Commutative law (proof)

We define addition of matrices as:

$$A + B = (A + B)_{ij} = A_{ij} + B_{ij}$$

Then, from commutative property of addition for real numbers we have:

$$A_{ij} + B_{ij} = B_{ij} + A_{ij}$$

$$= (B + A)_{ij} = B + A$$

$$\underline{Q.E.D}$$

2.1.2 Distributive law (proof)

With similar reasoning as in previous proof, we have:

$$c(A + B) = c((A + B)_{ij}) = c(A_{ij} + B_{ij})$$

$$= cA_{ij} + cB_{ij} = cA + cB$$

$$\underline{Q.E.D}$$

2.1.3 Associative law (proof)

Further:

$$A + (B + C) = (A + (B + C))_{ij} = A_{ij} + (B + C)_{ij}$$

$$= A_{ij} + B_{ij} + C_{ij} = (A + B)_{ij} + C_{ij}$$

$$= (A + B) + C$$

$$\underline{Q.E.D}$$

2.2 Laws of matrix multiplication

Definition. The three matrix multiplication laws are:

$$A(B+C) = AB + AC \quad (distributive\ law\ from\ the\ left) \tag{4}$$
$$(A+B)C = AC + BC \quad (distributive\ law\ from\ the\ right) \tag{5}$$
$$A(BC) = (AB)C \quad (associative\ law\ for\ ABC) \tag{6}$$

2.2.1 Distributive law from the left (proof)

We define the matrix multiplication as:

$$(A(B+C))_{ij} = \sum_{k=1}^{n} A_{ik}(B+C)_{kj}$$
$$= \sum_{k=1}^{n} A_{ik}(B_{kj}+C_{kj}) = \sum_{k=1}^{n} A_{ik}B_{kj} + \sum_{k=1}^{n} A_{ik}C_{kj}$$
$$= (AB)_{ij} + (AC)_{ij} = (AB+AC)_{ij} = AB + AC$$
$$\underline{Q.E.D}$$

2.3 Block multiplication

To solve multiply the block matrices, the following steps were carried out:

$$\begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} A & B \\ -CA^{-1}A + C & -CA^{-1}B + D \end{pmatrix}$$

$$= \begin{pmatrix} A & B \\ C - C & -CA^{-1}B + D \end{pmatrix} = \begin{pmatrix} A & B \\ 0 & -CA^{-1}B + D \end{pmatrix}$$

2.4 Back substitution

We want to solve the following upper triangular system:

$$\begin{array}{ccccccc} 2x & + & 2y & + & z & = & 9 \\ & & y & + & 2z & = & 7 \\ & & & & 3z & = & 9 \end{array}$$

To do this we use back substitution, through the following steps:

$$\Rightarrow \quad z = \frac{9}{3} = 3 \quad \Rightarrow \quad y + 2(3) = 7 \quad \Rightarrow \quad y = 1$$
$$\Rightarrow \quad 2x + 2(1) + (3) = 9 \quad \Rightarrow \quad 2x = 4 \quad \Rightarrow \quad x = 2$$

Hence, we have:

$$\begin{cases} x = 2 \\ y = 1 \\ z = 3 \end{cases}$$

For n equations, back substitution requires us to do n divisions, $\frac{n^2-n}{2}$ additions/subtractions and $\frac{n^2-n}{2}$ multiplications. That leaves us with a total of n^2 arithmetic operations.

2.5 Elimination without pivoting

The aim of this assignment is to transform the system of equations below into an equivalent upper triangular system by elementary row operations.

$$\begin{array}{ccccccc} x & + & 2y & + & z & = & 7 \\ 2x & + & 5y & + & 3z & = & 17 \\ 3x & + & 5y & + & 2z & = & 29 \end{array}$$

When doing elementary row operations it is convenient to present the system of equation in matrix form as below.

$$\left(\begin{array}{ccc|c} 1 & 2 & 1 & 7 \\ 2 & 5 & 3 & 17 \\ 3 & 5 & 2 & 29 \end{array}\right)$$

The first step is then to subtract two times the first row from the second row and subtract three times the first row from the third row.

$$\left(\begin{array}{ccc|c} 1 & 2 & 1 & 7 \\ 0 & 1 & 1 & 3 \\ 0 & -1 & -1 & 8 \end{array}\right)$$

The second step is to add the second row to the third row. This is all that is needed to transform the matrix into equivalent upper triangular form.

$$\left(\begin{array}{ccc|c} 1 & 2 & 1 & 7 \\ 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 11 \end{array}\right)$$

Notice, since the third row indicates that $0 = 11$, this system of equations has no solution.

2.6 Block elimination

In this assignment we are given the block matrix:

$$\begin{pmatrix} B & C \\ D - XB & E - XC \end{pmatrix}.$$

and are asked to find which matrix X that will eliminate block (2,1). In other words, we want to solve:

$$D - XB = \mathbf{0}$$

If we assume that B is invertible and of matching size with X, this is easily done by taking the inverse of B and multiplying it with X:

$$XB = D \quad \Rightarrow \quad X = DB^{-1}$$

3 Algorithms

3.1 Back substitution

Here we want to use Matlab in order to solve an upper triangular linear system of the form.

$$Ax = b \tag{7}$$

A is an upper triangular matrix of size $n \times n$, and b is a vector of size $n \times 1$, and the solution should be conducted using backward substitution.

3.1.1 Pseudocode

```
Algorithm 1 Backward substitution
Require: $A$ is upper triangular $\wedge$ $A \in \mathbb{R}^{n\times n}$ $\wedge$ $b \in \mathbb{R}^{n\times 1}$
Ensure: $x = A^{-1}b$
  $x \leftarrow \mathbf{0} \in \mathbb{R}^{n\times 1}$
  for $k \leftarrow [n : -1 : 1]$ do
    $x_{k1} \leftarrow (b_{k1} - A_{k*}x)/A_{kk}$
  end for
```

Notice: Since x is initialised as a zero vector and is further filled from the bottom up, the elements "above" the current row (of an iteration) are all empty. This enables us to save some steps since it is then possible to multiplying the current row of A with the entire x vector, without causing an error in the next step. This is a perk of having the matrix in upper triangular form.

3.1.2 Tests

To test our algorithm, we use a Matlab script that generates random upper triangular matrices of optional size as well as random vectors of compatible size. This script further solves these randomly generated systems, using both Matlab's built-in backslash operator (`x2 = A\b`) as well as our `backsubst` function (`x = backsubst(A,b)`). Lastly, it computes the norm of `x2 - x` to compare the results between them. By doing this, we are comparing our way of solving the system, using `backsubst` with Matlab's widely recognised way of solving it, using the backslash operator.

The expected/desired result is that the norm of the difference between the solutions should be small. The test was then carried out and gave the following result:

$$A \in \mathbb{R}^{3\times 3} \ \wedge \ b \in \mathbb{R}^{3\times 1} \quad \Rightarrow \quad \texttt{norm(x2 - x)} = 5.20 \cdot 10^{-18}$$
$$A \in \mathbb{R}^{4\times 4} \ \wedge \ b \in \mathbb{R}^{4\times 1} \quad \Rightarrow \quad \texttt{norm(x2 - x)} = 1.55 \cdot 10^{-17}$$
$$A \in \mathbb{R}^{5\times 5} \ \wedge \ b \in \mathbb{R}^{5\times 1} \quad \Rightarrow \quad \texttt{norm(x2 - x)} = 6.93 \cdot 10^{-18}$$
$$A \in \mathbb{R}^{6\times 6} \ \wedge \ b \in \mathbb{R}^{6\times 1} \quad \Rightarrow \quad \texttt{norm(x2 - x)} = 5.60 \cdot 10^{-17}$$

It is quite clear from the results above that the difference is very small between the two methods, and computationally one could argue that it is almost zero. Hence, it seems like our algorithm works well when solving this kind of linear systems.

3.2 Triangular matrix inversion

In this assignment we are asked to construct a function `B = triinv(A)` that takes an upper triangular matrix A and computes its inverse B using the function `backsubst(A,b)`, which is explained in Section 3.1.1.

3.2.1 Pseudocode

Algorithm 2 Triangular matrix inversion

```
Require: $A$ is upper triangular $\wedge$ $A \in \mathbb{R}^{n\times n}$
Ensure: $B = A^{-1}$
  $x \leftarrow \mathbf{0} \in \mathbb{R}^{n\times n}$
  $I \leftarrow \mathbf{I} \in \mathbb{R}^{n\times 1}$
  for $k \leftarrow [1:1:n]$ do
    $B_{*k} \leftarrow$ backsubst($A, \mathbf{I}_{*k}$)
  end for
```

3.2.2 Tests

To test this function, we want to look at the Frobenius matrix norm of $AB - I$ which is defined as:

$$\texttt{norm(A*B-eye(n),'fro')} \tag{8}$$

This is done by generating an upper triangular matrix of optional size with randomised elements between 0 and 100, and then use our function to find its inverse. Further, the Frobenius matrix norm is computed, which is expected to be a number close to zero if the algorithm is carried out correctly.

When doing this for matrices of sizes 3×3 - 6×6, we obtain the following result:

$$A \in \mathbb{R}^{3\times 3} \quad \Rightarrow \quad \texttt{norm(A*B-eye(n),'fro')} = 1.22 \cdot 10^{-16}$$

$$A \in \mathbb{R}^{4\times 4} \quad \Rightarrow \quad \texttt{norm(A*B-eye(n),'fro')} = 2.24 \cdot 10^{-16}$$

$$A \in \mathbb{R}^{5\times 5} \quad \Rightarrow \quad \texttt{norm(A*B-eye(n),'fro')} = 3.90 \cdot 10^{-16}$$

$$A \in \mathbb{R}^{6\times 6} \quad \Rightarrow \quad \texttt{norm(A*B-eye(n),'fro')} = 5.24 \cdot 10^{-16}$$

These are all considerably small numbers and can be taken as close to zero, meaning our algorithm seems to work well.

4 Applications

4.1 Early collision warning

Air traffic control systems are essential for managing air crafts take-off and landing schedule. They often keep track of multiple planes at the same time and the systems are commonly very complex. In this assignment we are looking at a highly simplified system of air crafts, where two planes are taking off at two different location at a given time. A directional velocity is given for each air craft, and our task is to construct a program that will indicate whether the trajectories of the two planes will intervene with each other in a problematic way in a near future.

The problem is specified in two dimensions, narrowing it down to finding if, and further when, the trajectories of the air crafts are crossing each other. The requirements given in the assignment makes us interested mainly in a 10 minute future and should meet the following standards:

Level 1 If the paths of the aircraft will not intersect in the future, output No conflict possible.

Level 2 If t_a or Δt is more than 10 minutes, output No conflict.

Level 3 If t_a is at most 10 minutes and Δt is larger than 3 minutes, output Near conflict in XmYYs (time separation ZmWWs), where the reported times are t_a and Δt, respectively.

Level 4 If t_a is at most 10 minutes and Δt is at most 3 minutes, output Conflict in XmYYs (time separation ZmWWs), where the reported times are t_a-3 minutes and Δt, respectively.

Level 5 If t_a and Δt both are at most 3 minutes, output Conflict! Predicted collision in XmYYs (time separation ZmWWs), where the reported times are t_a and Δt, respectively.

The goal is then to write a function [`x,y,t1,t2`] `= collision(p1,v1,p2,v2)` which takes the initial positions and velocities for two air crafts and return their intersection coordinates and at which time each air craft will arrive at this location. This function will then be used in another script, for checking which alert level two given air crafts correspond to.

4.1.1 Setting up the problem

To solve the problem we start by assuming that the air crafts follows linear trajectories defined mathematically as:

$$p_i(t) = P_i + tv_i \tag{9}$$

Breaking this up in their x- and y-direction, we get for each air craft:

$$\begin{cases} x_i(t) = X_i + t_i v_{xi} \\ y_i(t) = Y_i + t_i v_{yi} \end{cases} \tag{10}$$

where X_i and Y_i are the initial x- and y-coordinates, and v_{xi} and v_{yi} are the velocities in the x- and y-direction, both for the i^{th} air craft.

By further defining these equations for two different air crafts and then setting them equal to each other, we can obtain a system of equations that will tell us the time at which these two air crafts trajectories will cross each other (if they cross at all). To obtain this system of equations, the following steps are carried out:

We have the two equations for each air craft trajectory, specified as:

$$\begin{cases} x_1(t) = X_1 + t_1 v_{x1} \\ y_1(t) = Y_1 + t_1 v_{y1} \end{cases} \qquad \begin{cases} x_2(t) = X_2 + t_2 v_{x2} \\ y_2(t) = Y_2 + t_2 v_{y2} \end{cases} \tag{11}$$

By setting their x- and y-trajectories equal to each other we get:

$$\begin{Bmatrix} X_1 + t_1 v_{x1} = X_2 + t_2 v_{x2} \\ Y_1 + t_1 v_{y1} = Y_2 + t_2 v_{y2} \end{Bmatrix} \quad \Rightarrow \quad \begin{Bmatrix} t_1 v_{x1} - t_2 v_{x2} = X_2 - X_1 \\ t_1 v_{y1} - t_2 v_{y2} = Y_2 - Y_1 \end{Bmatrix} \tag{12}$$

We then define:

$$\begin{cases} X_2 - X_1 = X^{'} \\ Y_2 - Y_1 = Y^{'} \end{cases} \tag{13}$$

Which gives us that:

$$\begin{bmatrix} v_{x1} & -v_{x2} \\ v_{y1} & -v_{y2} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} X^{'} \\ Y^{'} \end{bmatrix} \quad \Leftrightarrow \quad \underline{Vt = P^{'}} \tag{14}$$

Solving this system of equations then gives us at which times, t_1 and t_2, the trajectories of the air crafts will intersect each other (if they do).

4.1.2 Pseudocode

Algorithm 3 Collision

```
Require: p1 – initial position of air craft 1
         v1 – directional velocity of air craft 1
         p2 – initial position of air craft 2
         v2 – directional velocity of air craft 2
  Trajectories of air craft 1 & 2 cannot be paralell or collinear
Ensure: x – intersection point x-coordinate
        y – intersection point y-coordinate
        $t_1$ – time at intersection point, air craft 1
        $t_2$ – time at intersection point, air craft 2

  $A \leftarrow \mathbf{0} \in \mathbb{R}^{2\times2}$
  $A_{*1} \leftarrow v_1$
  $A_{*2} \leftarrow v_2$

  $b \leftarrow p2 - p1$
  $t \leftarrow A^{-1}b$

  $\mathtt{x} \leftarrow \mathtt{p1}_x + t_1 \times \mathtt{v1}_x$
  $\mathtt{y} \leftarrow \mathtt{p1}_y + t_2 \times \mathtt{v1}_y$

  if $\mathtt{t1} < 0$ or $\mathtt{t2} < 0$ then
    $\mathtt{x} \leftarrow$ NaN
    $\mathtt{y} \leftarrow$ NaN
  end if
```

4.1.3 Tests

To use our `collision` function, we write a function that is called `checkLevel(x,y,t1,t2)`, that takes the information obtained by `collision`. That is, the intersection location coordinates x and y, and the times, t1 and t2, at which each air craft reaches this location. `checkLevel` is then going through each condition stated in section 4.1, and sends the message of which ever of the Levels that the two air crafts are corresponding to.

In order to assure ourselves that the code is working properly, we also need to run a test script. We call this test script `MainAero` and what it does is, that it takes five different cases of our scenario of two air crafts starting at two different locations with know directional velocities. All cases from Level 1 - Level 5 was tested, and the plots given in Figure 1 to 5 shows the trajectories and intersection point for each scenario.

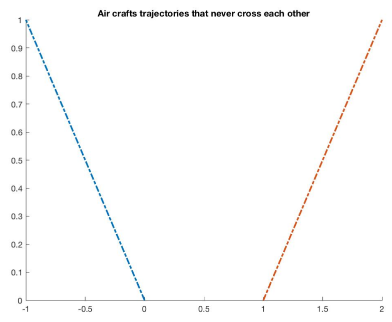

Figure 1: Conflict not possible.

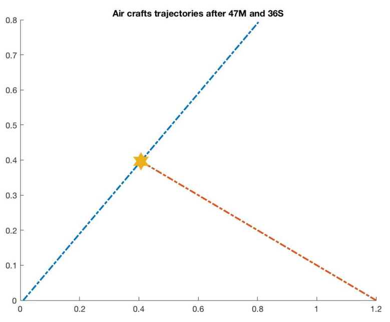

Figure 2: No conflict scenario

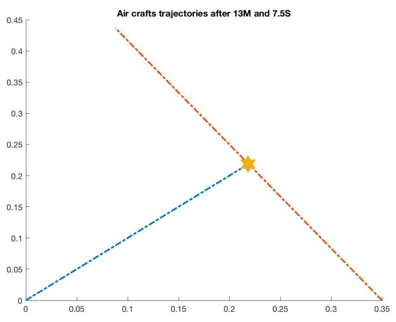

Figure 3: Near conflict scenario.

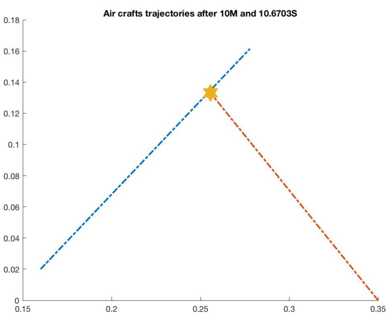

Figure 4: Conflict scenario.

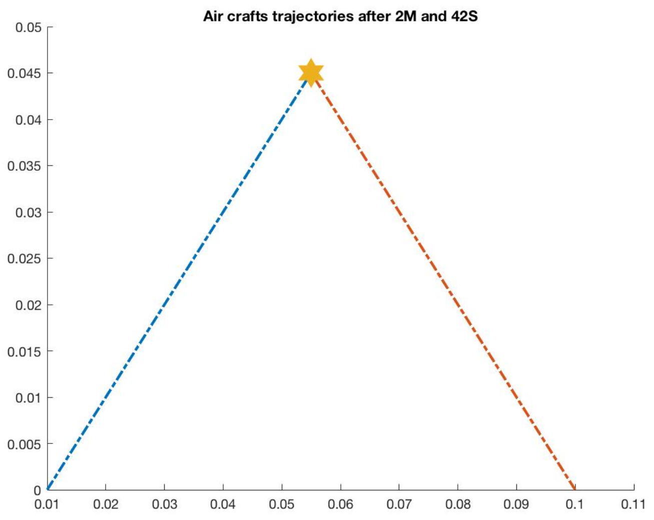

Figure 5: Collision scenario.

By looking at the images in Figure 1 to 5, we see each scenario visualised. They all gave the right Level messages, which can be checked by the reader if he/she runs the Matlab script given in Apendix A.3 to A.5. In Figure 1 the trajectories of the air crafts never cross in positive time, hence conflict is not possible. In Figure 2 the trajectories cross at an intersection point, but note that the time of the second air craft arrival is 47 minutes which indeed means there is no conflict. In Figure 3 the air crafts arrive at the intersection point with a time difference less than 10 minutes, and this should thus be notified. In Figure 4 there is a conflict. Note however, that the time 10 minutes and 10 seconds is when the second air craft arrives at the intersection point. In Figure 5 a collision is clearly presented. From these tests, it seems like the program alerts the right messages and thus works properly.

A Matlab Code

A.1 backsubst.m

backsubst.m

```
function x = backsubst(A, b)
% BACKSUBST - Function that takes an upper triangular matrix A and a vector
%              of matching size b corresponding to the linear system Ax=b,
%              and returns the solution for x.
%
% MINIMAL WORKING EXAMPLE: Solve Ax = b for
%
%   A = [1 3 6; 0 5 2; 0 0 9];
%   b = [2.3; 1.4; 6.1];
%
%   x = backsubst(A,b); Solve Ax = b for x

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

%Sends error message if the given matrix is non-square
if size(A,1) ~= size(A,2)
   error('Matrix must be square');
end
%Sends error message if A and b are of incompatible sizes
if size(A,2) ~= size(b,1)
   error('Matrix/vector mismatch');
end

x = zeros(length(b),1); %Pre-define x as zero vector for speed
n = length(A); %Define number of rows in the square matrix A

%Starts at the last row and iteratively performs back substitution until it
%reaches the first row of the matrix.
for i = n:-1:1
    x(i) = (b(i)-A(i,:)*x)/A(i,i);
end

%% R kade g ra kod f r att transformera till upper triangular
% for i = 2:length(A)
% A(i:end,:) = A(i:end,:)-(A(i:end,i-1)/A(i-1,i-1))*A(i-1,:)
% end
```

test_backsub.m

```
% TEST_BACKSUBST - Script for testing backsubst.m on randomly
    generated
%                  upper triangular matrix (of random size).
%
%   MINIMAL WORKING EXAMPLE:
%   ">> test_backsubst" to see results in relevant test.

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

clear all
N = 50; %Define that random integers should be taken between 0 and
    N
n = 5; %Define size of nxn matrix

%Construct the randomised diagonal of the matrix random nxn upper
%triangular matrix
diagRand = randi(N,n);
diagRow = diagRand(1,:);

%Construct the randomised nxn upper triangular matrix with random
    integers
%between 0 and N
A = triu(randi(N,n),1) + diag(diagRow,0);
b = 10*rand(n,1); %Construct a randomised vector of matching size (
    nx1)

x = backsubst(A, b); %Solve system using backsubst
x2 = A\b; %Solve the system using backslash operator

normX = norm(x-x2) %Computes the norm of the difference between the
     methods
```

A.2 triinv.m

triinv.m

```
function B = triinv(A)
% TRIINV - Function that takes an nxn upper triangular matrix A and
     returns
%           the inverse of A.
%
%    Input parameters:
%    A: nxn upper triangular matrix
%
%    Output parameters:
%    B: Inverse of A
%
%    MINIMAL WORKING EXAMPLE: Find inverse of upper triangular
     matrix
%    defined as A = [1 3 6; 0 5 2; 0 0 9];
%
%    B = triinv(A);

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

B = zeros(size(A)); %pre-allocate room for B by making a zero
    matrix (for speed)
I = eye(length(A)); %create the nxn identity matrix
n = length(A); %define number of rows in A to loop over

%perform backsubstitution for each column of B to ultimatelly find
    the
%inverse of A
for i = 1:n
    B(:,i) = backsubst(A,I(:,i));
end
```

test_triinv.m

```
% TEST_TRIINV - Script for testing triinv.m on four randomly
    generated
%                 upper triangular matrix of size 3x3, 4x4, 5x5 and 6
    x6.
%
%   MINIMAL WORKING EXAMPLE:
%   ">> test_triinv" to see results in relevant test.

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...
clear all
k = 1;
for i = 3:6
n = i; %Define size of nxn matrix

%Construct the randomised nxn upper triangular matrix with random
    numbers
%between 0 and 100
A = triu(100*rand(n),1) + diag(100*rand(n,1),0);

B = triinv(A); %Compute inverse of A by using triinv(A)

%Check correctness by computing the Frobenius matrix norm of AB - I
Frob(k) = norm(A*B-eye(n),'fro');
k = k + 1;
end

Frob = Frob'
```

A.3 collision.m

collision.m

```
function [x, y, t1, t2] = collision(p1, v1, p2, v2)
%COLLISION - Function that takes the initial coordinates and velocities of
%           two air crafts and returns the intersection point of their
%           trajectories as well as the time at which each air craft
%           reaches this location.
%
%   Input parameters:
%   p1: x- & y-coordinates of air craft 1's initial position
%   v1: directional velocity of air craft 1
%   p2: x- & y-coordinates of air craft 2's initial position
%   v2: directional velocity of air craft 2
%
%   Output parameters:
%   x: x-coordinate of intersection point
%   y: y-coordinate of intersection point
%   t1: Time at which air craft 1 reaches the intersection point
%   t2: Time at which air craft 2 reaches the intersection point
%
%   MINIMAL WORKING EXAMPLE: Find intersection point coordinates and times
%   of intersection for two air crafts with initial coordinates:
%   p1 = [0.16; 0.02]; p2 = [0.35; 0];
%   and initial velocities:
%   v1 = [0.7; 0.84]; v2 = [-0.56; 0.79];
%
%   [x, y, t1, t2] = collision(p1, v1, p2, v2)

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

%Construct matrix A with v1 and -v2 as column 1 and 2
A = [v1, -v2];
%Construct vector b from subtracting position coordinates p1 from position
%coordinates p2
b = p2-p1;
%Solve the system At=b using backslash operator
t = A\b;
%Give value to t1 and t2 so that the function can return them
t1 = t(1); t2 = t(2);

x = p1(1) + t(1)*v1(1); %Calculate x (for return)
y = p1(2) + t(1)*v1(2); %Calculate y (for return)

%Check if the air craft trajectories will cross in the future.
if t1 < 0 || t2 < 0
   x = NaN; y = NaN; %If not, return NaN for their coordinates
end

end
```

A.4 checkLevel.m

checkLevel.m

```
function[] = checkLevel(x, y, t1, t2)
%CHECKLEVEL - Function that checks alert Levels 1 - 5 for two air plane
%              trajectories, and gives corresponding error messages.
%
%   Input parameters:
%   x: x-coordinate for intersection point of two air craft trajectories
%   y: y-coordinate for intersection point of two air craft trajectories
%   t1: Time at which the fastest air craft reaches the intersection point
%   t2: Time at which the slowest air craft reaches the intersection point
%
%   MINIMAL WORKING EXAMPLE: Check levels for two airplanes with
%   intersection point at x = 2.3 and y = 4.8, and times of intersection
%   t1 = 8.92 and t2 = 38.2.
%
%   >> checkLevel(x, y, t1, t2)
%   >> "Gives level of alert"

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

fprintf('Check result: \n');

dt = abs(t1-t2); %Compute dt = time difference between t1 and t2
dtM = floor(dt*60); %Convert time difference between t1 and t2 to minutes
dtS = (dt*60-dtM); %Seconds after last minute in time difference
ta = min(t1,t2);
taM = floor(ta*60);
taS = (ta*60-taM)*60;

%Level 1 - start by checking if the paths are even crossing each other in
%the future. If not, all other levels are irelevant.
if isnan(x) || isnan(y)
    fprintf('NO CONFLICT POSSIBLE \n \n');

%Check condition for each level

%Level 5
elseif ta <= 3*(1/60) && dt <= 3*(1/60)

    fprintf('CONFLICT! PREDICTED COLISION IN %gM%02gS (%gM%02gS) \n \n',...
        taM, taS, dtM, dtS);

%Level 4
elseif ta <= 1/6 && dt <= 3*(1/60)

    fprintf('CONFLICT IN %gM%02gS (%gM%02gS) \n \n', taM-3, taS,...
        dtM, dtS);

%Level 3
elseif ta <= 1/6 && dt > 3*(1/60)

    fprintf('NEAR CONFLICT IN %gM%02gS (%gM%02gS) \n \n', taM, taS,
    ...
            dtM, dtS);

%Level 2
elseif ta > 1/6 || dt > 1/6
    fprintf('NO CONFLICT \n \n');
end
end
```

A.5 MainAero.m

MainAero.m

```
% MAINAERO - Test script for 5 different scenarios of two air
    crafts leaving each known
%            location with a known speed.
%
%   MINIMAL WORKING EXAMPLE:
%   ">> MainAero" to see results in relevant tests.

% Author: Gustav Nystedt , guny0007@ad.umu.se
% 2018-09-13: Initial version
% 2018-09-27: Edited comments
%
% Function code starts here...

clear all; clc;
%% Define 5 different scenarios for p1, p2, v1, v2
p1_1 = [0.16; 0.02]; p2_1 = [0.35; 0];
v1_1 = [0.7; 0.84]; v2_1 = [-0.56; 0.79];

p1_2 = [0; 0]; p2_2 = [0.2; 0];
v1_2 = [1.6; 0.85]; v2_2 = [-1.78; 1.2];

p1_3 = [0; 0]; p2_3 = [1; 0];
v1_3 = [-1; 1]; v2_3 = [1; 1];

p1_4 = [0.01; 0]; p2_4 = [1.2; 0];
v1_4 = [1; 1]; v2_4 = [-1; 0.5];

p1_5 = [0; 0]; p2_5 = [0.35; 0];
v1_5 = [1; 1]; v2_5 = [-1.2; 2];

%% Call for collision function to find intersection point
    coordinates and
  %and at which time each air craft will be at the intersection
    point.
  %This is done for each of the 5 scenarios.
[x1, y1, t1_1, t2_1] = collision(p1_1, v1_1, p2_1, v2_1);
[x2, y2, t1_2, t2_2] = collision(p1_2, v1_2, p2_2, v2_2);
[x3, y3, t1_3, t2_3] = collision(p1_3, v1_3, p2_3, v2_3);
[x4, y4, t1_4, t2_4] = collision(p1_4, v1_4, p2_4, v2_4);
[x5, y5, t1_5, t2_5] = collision(p1_5, v1_5, p2_5, v2_5);

%% Find longest time to intersetion point. This is for plotting
    purposes
tb_1 = max(t1_1,t2_1);
tb_2 = max(t1_2,t2_2);
tb_3 = max(t1_3,t2_3);
tb_4 = max(t1_4,t2_4);
tb_5 = max(t1_5,t2_5);

%% Cal trajCol.m for plotting the trajectories of the air crafts
trajCol(p1_1, v1_1, p2_1, v2_1, tb_1, 1);
trajCol(p1_2, v1_2, p2_2, v2_2, tb_2, 1);
trajCol(p1_3, v1_3, p2_3, v2_3, tb_3, 1);
trajCol(p1_4, v1_4, p2_4, v2_4, tb_4, 1);
trajCol(p1_5, v1_5, p2_5, v2_5, tb_5, 1);

%% Check which "cautious" level each scenario corresponds to
checkLevel(x1, y1, t1_1, t2_1);
checkLevel(x2, y2, t1_2, t2_2);
checkLevel(x3, y3, t1_3, t2_3);
checkLevel(x4, y4, t1_4, t2_4);
checkLevel(x5, y5, t1_5, t2_5);
```