Introductory Computer Programming Project on

A Study of the Stability of Methods Used for Solving the Least Square Problem

Adhiraj Mandal, Soumya Mukherjee and Prabhat Kumar Jha

Contents

1 Objective of the Project

Consider a linear system of equations having n unknowns and m equations. Symbolically, we wish to find a vector $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ that satisfies

$$Ax = b$$

where $\mathbf{A} \in \mathbf{R}^{\mathbf{m}\times\mathbf{n}}$ and $\mathbf{b} \in \mathbf{R}^{\mathbf{m}}$. In general, such a problem has no solution. In many cases, the system of equations does not admit any solution. In such cases, we try to find a $(n \times 1)$ vector x such that $\|\mathbf{r}\| = \|\mathbf{b} - \mathbf{Ax}\|_{\mathbf{2}}$ is minimized. The problem can be stated as follows:
Given $\mathbf{A}^{\mathbf{m}\times\mathbf{n}}$ (in most cases $m \geq n$) and $\mathbf{b} \in \mathbb{R}^{\mathbf{m}}$, we want to find $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ such that the norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ given by $\|\mathbf{r}\| = \|\mathbf{b} - \mathbf{Ax}\|_{\mathbf{2}}$ is minimized.
A vector $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ minimizes the residual norm $\|\mathbf{b} - \mathbf{Ax}\|_{\mathbf{2}}$, thereby solving the least squares problem if and only if

$$A^T Ax = A^T b \tag{1.1}$$

This system of equations is known as the least square normal equations. In this project we want to study possible methods of solution of the least square normal equation. We will implement and analyze the stability of the methods for solving the normal equations, which are based on :

- **Gaussian Elimination (or equivalently LU decomposition)** of $A^T A$, which reduces to **Cholesky decomposition** in the case of symmetric, positive definite matrices (We also study the effect of partial column pivoting).
- **QR Factorization** of A. The QR Factorization may be performed by using one of the following methods : **Gram-Schmidt Orthogonalisation** (Classical Gram-Schmidt and Modified Gram-Schmidt), **Householder Triangulation** and **Given's Rotation Method**.

2 Conditions for solubility of a system of linear equations

Suppose, $\mathbf{Ax} = \mathbf{b}$ be a system of linear equations, where $\mathbf{A}$ is a $m \times n$ matrix, $\mathbf{x} \in \mathbf{R}^{\mathbf{n}}$ and $\mathbf{b} \in \mathbf{R}^{\mathbf{m}}$. Then, the system of equations has :

- no solution if $\mathbf{b} \notin \mathbf{C}(\mathbf{A})$, the column space of $\mathbf{A}$.
- unique solution if $\mathbf{b} \in \mathbf{C}(\mathbf{A})$ and $\mathbf{A}$ has full column rank.
- infinitely many solutions if $\mathbf{b} \in \mathbf{C}(\mathbf{A})$ and $\mathbf{A}$ has less than full column rank.

3 Concept of Conditioning and Stability

It would be a fine thing if numerical algorithms could provide exact solutions to numerical problems. Since the problems are continuous while digital computers are discrete, however, this is generally not possible. The notion of stability is the standard way of characterizing what is possible—numerical analysts' idea of what it means to get the "right answer," even if it is not exact.

3.1 Floating Point Numbers

IEEE arithmetic is an example of an arithmetic system based on a floating point representation of the real numbers. This is the universal practice on general purpose computers nowadays. In a floating point number system, the position of the decimal (or binary) point is stored separately from the digits, and the gaps between adjacent represented numbers scale in proportion to the size of the numbers. This is distinguished from a fixed point representation, where the gaps are all of the same size.
Specifically, let us consider an idealized floating point number system defined as follows. The system consists

of a discrete subset F of the real numbers It determined by an integer $\beta \geq 2$ known as the base or radix (typically 2) and an integer $t \geq 1$ known as the precision B4 and 53 for IEEE single and double precision, respectively). The elements of F are the number 0 together with all numbers of the form

$$x = (m/\beta^t) \times \beta^e$$

where m is an integer in the range $1 < m < \beta^t$ and e is an arbitrary integer. Equivalently, we can restrict the range to $\beta^{t-1} \geq m \geq \beta^t - 1$ and thereby make the choice of m unique. The quantity m/β^t is then known as the fraction or mantissa of x, and e is the exponent Our floating point number system is idealized in that it ignores over and underflow. As a result, F is a countably infinite set, and it is self-similar: F = βF.

3.2 Machine Epsilon

The resolution of F is traditionally summarized by a number known as ma- chine epsilon. Provisionally, let us define this number by

$$\epsilon_{machine} = \frac{1}{2}\beta^{1-t}$$

This number is half the distance between 1 and the next larger floating point number. In a relative sense, this is as large as the gaps between floating point numbers get. That is, $\epsilon_{machine}$ has the following property:

For all $x \in R \, \exists \, x^{'} \in F$, such that

$$\frac{\|x - x^{'}\|}{\|x\|} \leq \epsilon_{machine} \tag{3.2.1}$$

For the values of β and t common on various computers, $\epsilon_{machine}$ usually lies between 10^{-6} and 10^{-35}.
Let fl : $\mathbb{R} \to F$ be a function giving the closest floating point approximation to a real number, its rounded equivalent in the floating point system. Then (3.2.1) can be expressed as:

For all $x \in$ R , $\exists$ an ϵ with $|\epsilon| \leq \epsilon_{machine}$, such that

$$\text{fl}(x) = x(1 + \epsilon) \tag{3.2.2}$$

3.3 Floating Point Arithmetic

It is not enough to represent real numbers on a computer ; one must compute with them. On a computer, all mathematical computations are reduced to certain elementary arithmetic operations, of which the classical set is +, —, ×, and ÷. Mathematically, these symbols represent operations on $\mathbb{R}$. On a computer, they have analogues that are operations on F. It is common practice to denote these floating point operations by $\oplus, \ominus, \otimes, \oslash$.
A computer might be built on the following design principle. Let x and y be arbitrary floating point numbers, that is, x,$y \in F$.
Let * be one of the four fundamental operators and let $\circledast$ be its floating point analogue. Then we must have, $x \circledast y = \text{fl}(x * y)$.
If this property holds then the computer has a simple and powerful property.

Fundamental Axiom of Floating Point Arithmetic
For all x and $y \in F$ we have an ϵ , with $|\epsilon| < \epsilon_{machine}$, such that

$$x \circledast y = (x * y)(1 + \epsilon) \tag{3.3.1}$$

In words, every operation of floating point arithmetic is exact up to a relative error of size at most $\epsilon_{machine}$.

3.4 Condition of a Problem

We can view a problem as a function $f : X \to Y$ from a normed vector space X of data to a normed vector space Y of solutions. This function f is usually nonlinear (even in linear algebra), but most of the time it is at least continuous. Typically we shall be concerned with the behavior of a problem f at a particular data point $x \in X$ (the behavior may vary greatly from one point to another). The combination of a problem f with prescribed data x might be called a problem instance. A well-conditioned problem (instance) is one with the property that all small perturbations of x lead to only small changes in $f(x)$. An ill-conditioned problem is one with the property that some small perturbation of x leads to a large change in $f(x)$.

3.4.1 Absolute Condition Number

Let δx denote a small perturbation of x, and write $\delta f = f(x + \delta x) - f(x)$. The absolute condition number $\hat{k} = \hat{k}(x)$ of the problem f at x is defined as

$$\hat{k} = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|} = \sup_{\delta x} \frac{\|\delta f\|}{\|\delta x\|}$$

If f is differentiable, then

$$\hat{k} = \|J(x)\|$$

where $\|J(x)\|$ represents the norm of J(x), the Jacobian of f at x induced by the norms on X and Y.

3.4.2 Relative Condition Number

When we are concerned with relative changes, we need the notion of relative condition. The relative condition number $k = k(x)$ is defined by

$$k = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \left(\frac{\|\delta f\|}{\|\delta x\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) = \sup_{\delta x} \left(\frac{\|\delta f\|}{\|\delta x\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right)$$

If f is differentiable, then

$$k = \frac{\|J(x)\|}{\|f(x)\|/\|x\|}$$

A problem is well-conditioned if k is small (e.g. 1,10 10^2) and ill-conditioned if k is large (e.g. $10^6, 10^{16}$). We state a few results and definitions relating to the condition of a problem

- **Condition of Matrix-Vector Multiplication** - Let $\mathbf{A} \in \mathbf{R^{m \times n}}$ be nonsingular and consider the equation $\mathbf{Ax} = \mathbf{b}$. The problem of computing $\mathbf{b}$, given $\mathbf{x}$, has condition number

  $$k = \|A\| \frac{\|x\|}{\|b\|} \leq \|A\| \|A^{-1}\|$$

  with respect to perturbations of x. The problem of computing $\mathbf{x}$, given $\mathbf{b}$, has condition number

  $$k = \|A^{-1}\| \frac{\|b\|}{\|x\|} \leq \|A\| \|A^{-1}\|$$

  with respect to perturbations of b.

- **Condition Number of a Matrix** - The condition number of a matrix $\mathbf{A}$ relative to the norm $\|.\|$ is defined as

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

- **Condition of a System of Equations** - Let $\mathbf{b}$ be fixed and consider the problem of computing $\mathbf{x} = \mathbf{A^{-1}b}$, where $\mathbf{A}$ is square and nonsingular. The condition number of this problem with respect to perturbations in A is

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

3.5 Notion of Stability

We have defined a problem as a function f : $X \to Y$ from a normed vector space X of data to a normed vector space Y of solutions.
An algorithm can be viewed as another map $\tilde{f} : X \to Y$ between the same two spaces. We make this definition precise as follows. Let a problem f, a computer whose floating point system satisfies (3.3.1), an algorithm for f (in the loose sense of the term), and an implementation of this algorithm in the form of a computer program be fixed. Given data $\mathbf{x} \in \mathbf{X}$, let this data be rounded to floating point in a matter satisfying (**??**) and then supplied as input to the computer program. Now, run the program. The result is a collection of floating point numbers that belong to the vector space Y (since the algorithm was designed to solve f). Let this computed result be called $\tilde{f}(x)$. As a minimum, f(x) will be affected by rounding errors. Depending on the circumstances, it may also be affected by all kinds of other complications such as convergence tolerances or even the other jobs running on the computer, in cases where the assignment of computations to processors is not determined until runtime. Thus the "function" f(x) may even take different values from one run to the next; it may be multivalued. (In fact, the problem f should really be allowed to be multivalued too; this permits handling of cases where a nonunique solution is acceptable, e.g., either of the two square roots of a complex number.) Yet despite all these complications, we shall find that we can make surprisingly clean statements about f(x), and hence about the accuracy of the algorithms of numerical linear algebra.

3.5.1 Accuracy

An algorithm $\tilde{f}$ for a problem f is accurate if for each $\mathbf{x} \in \mathbf{X}$,

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\epsilon_{machine}) \tag{3.5.1.1}$$

3.5.2 Stability

If the problem f is ill-conditioned, however, the goal of accuracy as defined by (3.5.1.1) is unreasonably ambitious. Rounding of the input data is unavoidable on a digital computer, and even if all the subsequent computations could be carried out perfectly, this perturbation alone might lead to a significant change in the result. Instead of aiming for accuracy in all cases, the most appropriate course of action in general is to aim for stability. We say that an algorithm $\tilde{f}$ for a problem f is stable if for each $\mathbf{x} \in \mathbf{X}$,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(\epsilon_{machine}) \tag{3.5.2.1}$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{machine}) \tag{3.5.2.2}$$

In words, a stable algorithm gives nearly the right answer to nearly the right question.

3.5.3 Backward Stability

Many algorithms of numerical linear algebra satisfy a condition that is both stronger and simpler than stability. We say that an algorithm $\tilde{f}$ for a problem f is backward stable if for each $\mathbf{x} \in \mathbf{X}$,

$$\tilde{f}(x) = f(\tilde{x}) \text{ for some } \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{machine}) \tag{3.5.3.1}$$

This is a tightening of the definition of stability in that the $O(\epsilon_{machine})$ in (3.5.2.1) has been replaced by zero. In words, a backward stable algorithm gives exactly the right answer to nearly the right question. In our study, we will be interested primarily in the backward stability of algorithms.

3.5.4 Accuracy of Backward Stable Algorithm

Suppose we have a backward stable algorithm f for a problem $\mathrm{f} : X \to Y$. In such a case, we are interested in whether the results will be accurate or not. The answer depends on the condition number $k = k(x)$ of f. If $k(x)$ is small, the results will be accurate in the relative sense, but if it is large, the accuracy will suffer proportionately. Suppose a backward stable algorithm is applied to solve a problem $\mathrm{f} : X \to Y$ with condition number k on a computer satisfying the axioms (**??**) and (3.3.1). Then the relative errors satisfy

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(k(x)\epsilon_{machine}) \tag{3.5.4.1}$$

3.5.5 Backward Error Analysis

Backward error analysis involves obtaining an accuracy estimate in two steps. One step is to investigate the condition of the problem. The other is to investigate the stability of the algorithm. Our conclusion is that if the algorithm is stable, then the final accuracy reflects that condition number.
Mathematically, this is straightforward, but it is certainly not the first idea an unprepared person would think of if called upon to analyze a numerical algorithm. The first idea would be forward error analysis. Here, the rounding errors introduced at each step of the calculation are estimated, and somehow, a total is maintained of how they may compound from step to step. Experience has shown that for most of the algorithms of numerical linear algebra, forward error analysis is harder to carry out than backward error analysis. With the benefit of hindsight, it is not hard to explain why this is so. Suppose a tried-and-true algorithm is used, say, to solve $\mathbf{Ax} = \mathbf{b}$ on a computer. It is an established fact that the results obtained will be consistently less accurate when A is ill-conditioned. Now, how could a forward error analysis capture this phenomenon? The condition number of A is so global a property as to be more or less invisible at the level of the individual floating point operations involved in solving $\mathbf{Ax} = \mathbf{b}$. Yet one way or another, the forward analysis will have to detect that condition number if it is to end up with a correct result. In short, it is an established fact that the best algorithms for most problems do no better, in general, than to compute exact solutions for slightly perturbed data. Backward error analysis is a method of reasoning fitted neatly to this backward reality.

4 Algorithms for solving the least squares problem and analysing their stability

4.1 Forward and Backward Substitution

A matrix equation in the form $\mathbf{Lx} = \mathbf{b}$ or $\mathbf{Ux} = \mathbf{b}$ where $\mathbf{L}, \mathbf{U} \in \mathbf{R^{m \times m}}$ and $\mathbf{x} = (\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m})^{\mathbf{T}} \in \mathbf{R^m}$ is very easy to solve by an iterative process called forward substitution for lower triangular matrices L and analogously back substitution for upper triangular matrices U. The process is so called because for lower triangular matrices, one first computes x_1, then substitutes that forward into the next equation to solve for x_2, and repeats through to x_m. In an upper triangular matrix, one works backwards, first computing x_m, then substituting that back into the previous equation to solve for x_{m-1}, and repeating through x_1. Notice that this does not require inverting the matrix.
The matrix equation $\mathbf{Lx} = \mathbf{b}$ can be written as a system of linear equations

$$\begin{array}{ccccccccc} \ell_{1,1}x_1 & & & & & & & = & b_1 \\ \ell_{2,1}x_1 & + & \ell_{2,2}x_2 & & & & & = & b_2 \\ \vdots & & \vdots & & \ddots & & & & \vdots \\ \ell_{m,1}x_1 & + & \ell_{m,2}x_2 & + & \cdots & + & \ell_{m,m}x_n & = & b_m \end{array} \tag{4.1.1}$$

Observe that the first equation $\ell_{1,1}x_1 = b_1$ only involves x_1, and thus one can solve for x_1 directly. The second equation only involves x_1 and x_2, and thus can be solved once one substitutes in the already solved

value for x_1. Continuing in this way, the k-th equation only involves $x_1, \ldots, x_k$, and one can solve for x_k using the previously solved values for $x_1, \ldots, x_{k-1}$.
The resulting formulae are:

$$\begin{aligned} x_1 &= \frac{b_1}{\ell_{1,1}}, \\ x_2 &= \frac{b_2 - \ell_{2,1}x_1}{\ell_{2,2}}, \\ &\vdots \\ x_m &= \frac{b_m - \sum_{i=1}^{m-1} \ell_{m,i}x_i}{\ell_{m,m}}. \end{aligned} \tag{4.1.2}$$

A matrix equation with an upper triangular matrix U can be solved in an analogous way, only working backwards.
The matrix equation $\mathbf{Ux} = \mathbf{b}$ can be written as a system of linear equations

$$\begin{array}{ccccccccc} u_{1,1}x_1 & + & u_{1,2}x_2 & + & \cdots & + & u_{1,m}x_m & = & b_1 \\ & & u_{2,1}x_2 & + & \cdots & + & u_{2,m}x_m & = & b_2 \\ & & & & \ddots & & & & \vdots \\ & & & & & & u_{m,1}x_m & = & b_m \end{array} \tag{4.1.3}$$

The resulting formulae are:

$$\begin{aligned} x_m &= \frac{b_m}{u_{m,1}}, \\ x_{m-1} &= \frac{b_{m-1} - u_{m-1,m}x_m}{u_{m-1,m-1}}, \\ &\vdots \\ x_1 &= \frac{b_1 - \sum_{i=2}^{m} u_{1,i}x_i}{u_{m,m}}. \end{aligned} \tag{4.1.4}$$

The above discussion makes the algorithm for implementing the backward and forward substitution quite obvious. These implementations will be used as helper functions for solving more general systems of equations as we proceed.

4.1.1 Implementation

The Forward and Backward Substitution are implemented in R along with demonstrations as follows:

Forward Substitution:

```
#Function for forward substitution
#Will solve Lx=y for square non-singular L
fwd_sub=function(L,y)
{
  if (dim(L)[1] != dim(L)[2]) stop("'L' must be a square matrix")
  n=dim(L)[1]
  for (i in 1:n)
  {
    if (identical(diag(L),rep(0,length(diag(L))))) stop("'L' must be a non-singular matrix")
    y[i]=y[i]/L[i,i]
    if((i-1)>0)
    {
      y[i]=y[i]-crossprod(L[i,1:(i-1)],y[1:(i-1)])/L[i,i]
    }
  }
  return(y)
}
```

Backward Substitution:

```
#Function for back substitution
#Will solve Ux=y for square non-singular U
back_sub=function(U,y)
{
  if (dim(U)[1] != dim(U)[2]) stop("'U' must be a square matrix")
  n=dim(U)[1]
  for (j in seq(n,1,-1))
  {
    if (identical(diag(U),rep(0,length(diag(U))))) stop("'U' must be a non-singular matrix")
    y[j]=y[j]/U[j,j]
    if((j-1)>0)
    {
      y[1:(j-1)]=y[1:(j-1)]-(y[j]*U[1:(j-1),j])
    }
  }
  return(y)
}
```

Demonstration:

```
#Demo of fwd_sub(Compare with forwardsolve function in R)
set.seed(0)
M=matrix(runif(9),ncol=3)
M[upper.tri(M)] = 0
M
x=c(5,3,4)
y=M%*%x
y
fwd_sub(M,y)
forwardsolve(M,y)
M%*%fwd_sub(M,y)

#Demo of back_sub(Compare with backsolve function in R)
set.seed(0)
M=matrix(runif(9),ncol=3)
M[lower.tri(M)] = 0
M
x=c(5,3,4)
```

```
y=M%*%x
y
back_sub(M,y)
backsolve(M,y)
M %*% back_sub(M,y)
```

Output:

```
> #Demo of fwd_sub(Compare with forwardsolve function in R)
> set.seed(0)
> M=matrix(runif(9),ncol=3)
> M[upper.tri(M)] = 0
> M
          [,1]      [,2]      [,3]
[1,] 0.8966972 0.0000000 0.0000000
[2,] 0.2655087 0.9082078 0.0000000
[3,] 0.3721239 0.2016819 0.6607978
> x=c(5,3,4)
> y=M%*%x
> y
          [,1]
[1,] 4.483486
[2,] 4.052167
[3,] 5.108856
> fwd_sub(M,y)
     [,1]
[1,]    5
[2,]    3
[3,]    4
> forwardsolve(M,y)
     [,1]
[1,]    5
[2,]    3
[3,]    4
> M%*%fwd_sub(M,y)
          [,1]
[1,] 4.483486
[2,] 4.052167
[3,] 5.108856
> #Demo of back_sub(Compare with backsolve function in R)
> set.seed(0)
> M=matrix(runif(9),ncol=3)
> M[lower.tri(M)] = 0
> M
          [,1]      [,2]      [,3]
[1,] 0.8966972 0.5728534 0.8983897
[2,] 0.0000000 0.9082078 0.9446753
[3,] 0.0000000 0.0000000 0.6607978
> x=c(5,3,4)
> y=M%*%x
> y
          [,1]
[1,] 9.795605
[2,] 6.503324
[3,] 2.643191
> back_sub(M,y)
     [,1]
[1,]    5
[2,]    3
[3,]    4
> backsolve(M,y)
     [,1]
[1,]    5
[2,]    3
[3,]    4
```

```
> M %*% back_sub(M, y)
          [,1]
[1,] 9.795605
[2,] 6.503324
[3,] 2.643191
```

4.1.2 Stability of forward and backward substitution

The following theorem states that the algorithm for backward substitution indicated by (4.1.4) is backward stable. A similar statement holds for forward substitution:

Theorem 1. *Let* $\mathbf{A} \in \mathbf{R^{m \times m}}$ *and* $\mathbf{A}$ *is upper triangular. The algorithm for solving the system* $\mathbf{Ax = b}$ *by backward substitution on a computer satisfying* (3.3.1) *is backward stable in the sense that the computed solution* $\tilde{x} \in R^m$ *satisfies*

$$(A + \delta A)\tilde{x} = b$$

for some upper triangular $\delta A \in R^{m \times m}$ *with*

$$\frac{\|\delta A|}{\|A|} = O(\epsilon_{machine})$$

Specifically, for each i,j

$$\frac{|\delta aij|}{|a_i j|} \leq m_{machine} + O(\epsilon^2_{machine})$$

4.2 Gaussian Elimination (LU Decomposition)

Gaussian elimination is the most intuitive method for solving a system of linear equations. It is the simplest way to solve linear systems of equations by hand, and also the standard method for solving them on computers. We first describe Gaussian elimination in its classical form without pivoting, and then add the feature of row pivoting that is essential to stability.

4.2.1 Gaussian Elimination without pivoting

Gaussian elimination transforms a full linear system into an upper-triangular one by applying simple linear transformations on the left, more specifically by making elementary row transformations.
There are three types of elementary row operations which may be performed on the rows of a matrix:

1. Swap the positions of two rows.
2. Multiply a row by a non-zero scalar.
3. Add to one row a scalar multiple of another.

If the matrix is associated to a system of linear equations, then these operations do not change the solution set. Therefore, if one's goal is to solve a system of linear equations, then using these row operations could make the problem easier.
Let $\mathbf{A^{m \times m}}$ be a square matrix. (The algorithm can also be applied to rectangular matrices, but as this is rarely done in practice, we shall confine our attention to the square case.) The idea is to transform A into an $m \times m$ uppertriangular matrix U by introducing zeros below the diagonal, first in column 1, then in column 2, and so on. This is done by subtracting multiples of each row from subsequent rows. This "elimination" process is equivalent to multiplying A by a sequence of lowertriangular matrices $\mathbf{L_k}$ on the left:

$$\underbrace{\mathbf{L_{m-1}} \dots \mathbf{L_2 L_1}}_{\mathbf{L^{-1}}} \mathbf{A} = \mathbf{U} \tag{4.2.1}$$

Setting $\mathbf{L} = \mathbf{L_1^{-1}L^{-1}}_{\mathbf{m-1}}^{\mathbf{-1}}$, gives $\mathbf{A} = \mathbf{LU}$. Thus we obtain an LU factorization of A,

$$A = LU \tag{4.2.2}$$

where U is upper-triangular and L is lower-triangular. It turns out that L is unit lower-triangular, which means that all of its diagonal entries are equal to 1.

Algorithm 1 Gaussian Elimination without Pivoting

Input : $m \times m$ matrix A with (i, j)-th element a_{ij}

1: U=A, L=I
2: **for** k=1 **to** m-1 **do**
3: **for** j=k+1 **to** m **do**
4: $l_{jk} = u_{jk}/u_{kk}$
$u_{j,k:m} = u_{j,k:m} - l_{jk}u_{k,k:m}$
5: **end for**
6: **end for**

Using the LU decomposition we can proceed as follows for solving the least square normal equations $\mathbf{A^TAx} = \mathbf{A^Tb}$

1. Form the matrix $\mathbf{A^TA}$ and the vector $\mathbf{A^Tb}$.
2. Compute the LU factorization $\mathbf{A^TA} = \mathbf{LU}$.
3. Solve the lower-triangular system $\mathbf{Lw} = \mathbf{A^Tb}$ for $\mathbf{w}$.
4. Solve the upper-triangular system $\mathbf{Ux} = \mathbf{w}$ for $\mathbf{x}$.

4.2.2 Instability of Gaussian Elimination without Pivoting

Unfortunately,the Gaussian elimination without pivoting is unusable for solving general linear systems, for it is not backward stable. The instability is related to another, more obvious difficulty. For certain matrices, Gaussian elimination fails entirely, because it attempts division by zero (One of the diagonal entries become zero). For example,

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

A slight perturbation of the same matrix reveals the more general problem. Suppose we apply Gaussian elimination to Now the process does not fail. Instead, 10^{20} times the first row is subtracted from the second row, and the following factors are produced:

$$L = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, U = \begin{pmatrix} 10^{-20} & 1 \\ 1 & -10^{20} \end{pmatrix}$$

However, suppose these computations are performed in floating point arithmetic with $\epsilon_{machine} \approx 10^{-16}$. The number 10^{-20} will not be represented exactly; it will be rounded to the nearest floating point number. For simplicity, imagine that this is exactly 10^{-20}.Then the floating point matrices produced by the algorithm will be

$$\tilde{L} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, \tilde{U} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & -10^{20} \end{pmatrix}$$

This degree of rounding might seem tolerable at first. After all, the matrix $\tilde{U}$ is close to the correct U relative to $\|Y\|$. However, the problem becomes apparent when we compute the product $\tilde{L}\tilde{L}$:

$$\tilde{L}\tilde{U} = \begin{pmatrix} 10^{-20} & 1 \\ 1^{20} & 0 \end{pmatrix}$$

This matrix is not at all close to A, for the 1 in the (2,2) position has been replaced by 0. If we now solve the system $\tilde{L}\tilde{U}x = b$, the result will be nothing like the solution to $Ax = b$. For example, with $b = (1, 0)^T$ we get $\tilde{x} = (0, 1)^T$, whereas the correct solution is $x \approx (-1, 1)^T$. A careful consideration of what has occurred in this example reveals the following. Gaussian elimination has computed the LU factorization stably: $\tilde{L}$ and $\tilde{U}$ are close to the exact factors for a matrix close to A (in fact, A itself). Yet it has not solved $Ax = b$ stably. The explanation is that the LU factorization, though stable, was not backward stable. As a rule, if one step of an algorithm is a stable but not backward stable algorithm for solving a subproblem, the stability of the overall calculation may be in jeopardy. In fact, for general $m \times m$ matrices A, the situation is worse than this. Gaussian elimination without pivoting is neither backward stable nor stable as a general algorithm for LU factorization. Additionally, the triangular matrices it generates have condition numbers that may be arbitrarily greater than those of A itself, leading to additional sources of instability in the forward and back substitution phases of the solution of $Ax = b$.

4.2.3 Gaussian Elimination with pivoting

In the last lecture we saw that Gaussian elimination in its pure form is unstable. The instability of the Classical Gaussian Algorithm can be controlled by permuting the order of the rows of the matrix being operated on, an operation called pivoting. Pivoting has been a standard feature of Gaussian elimination computations since the 1950s.

Pivots

At step k of Gaussian elimination, multiples of row k are subtracted from rows $k + 1, \ldots, m$ of the working matrix X in order to introduce zeros in entry k of these rows. In this operation row k, column k, and especially the entry x_{kk} play special roles. We call x_{kk}, the pivot. From every entry in the submatrix $X_{k+l:m,k:m}$ is subtracted the product of a number in row k and a number in column k, divided by x_{kk}.
However, there is no reason why the $k-th$ row and column must be chosen for the elimination. For example, we could just as easily introduce zeros in column k by adding multiples of some row i with $k < i \leq m$ to the other rows $k, \ldots, m$. In this case, the entry x_{ik} would be the pivot.Similarly, we could introduce zeros in column j rather than column k.
All in all, we are free to choose any entry of $X_{k:m,k:m}$ as the pivot, as long as it is nonzero. The possibility that an entry $x_{kk} = 0$ might arise implies that some flexibility of choice of the pivot may sometimes be necessary, even from a pure mathematical point of view. For numerical stability, however, it is desirable to pivot even when x_{kk} is nonzero if there is a larger element available. In practice, it is common to pick as pivot the largest number among a set of entries being considered as candidates.

Partial Pivoting

If every entry of $X_{k:m,k:m}$ is considered as a possible pivot at step k, there are $O((m-k)^2)$ entries to be examined to determine the largest. Summing over m steps, the total cost of selecting pivots becomes $O(m^3)$ operations, adding significantly to the cost of Gaussian elimination, not to mention the potential difficulties of global communication in an unpredictable pattern across all the entries of a matrix. This expensive strategy is called complete pivoting. In practice, equally good pivots can be found by considering a much smaller number of entries. The standard method for doing this is partial pivoting. Here, only rows are interchanged. The pivot at each step is chosen as the largest of the $m - k + 1$ subdiagonal entries in column k, incurring a total cost of only $O(m - k)$ operations for selecting the pivot at each step, hence $O(m^2)$ operations overall. To bring the k-th pivot into the (k, k) position, no columns need to be permuted; it is enough to swap row k with the row containing the pivot.
This algorithm can be expressed as a matrix product. We have already seen that an elimination step corresponds to left-multiplication by an elementary lower-triangular matrix $\mathbf{L_k}$. Partial pivoting complicates matters by applying a permutation matrix $\mathbf{P_k}$ on the left of the working matrix before each elimination. (A permutation matrix is a matrix with 0 everywhere except for a single 1 in each row and column. That is, it is a matrix obtained from the identity by permuting rows or columns.) After m - 1 steps, A becomes an

upper-triangular matrix U:

$$L_{m-1}P_{m-1}\dots L_2P_2L_1P_1A = U \tag{4.2.3}$$

Algorithm 2 Gaussian Elimination with Pivoting

Input : $m \times m$ matrix A with (i, j)-th element a_{ij}

1: U=A, L=I, P=I
2: **for** k=1 **to** m-1 **do**
3: Select $i \geq k$ to maximise $|u_{ik}|$
$u_{k,K:m} \leftrightarrow u_{i,k:m}$ (interchange two rows)
$l_{k,1:k-1} \leftrightarrow l_{i,1:k-1}$
$p_{k,:} \leftrightarrow p_{i,:}$
4: **for** j=k+1 **to** m **do**
5: $l_{jk} = u_{jk}/u_{kk}$
$u_{j,k:m} = u_{j,k:m} - l_{jk}u_{k,k:m}$
6: **end for**
7: **end for**

4.2.4 Stability of Gaussian Elimination with pivoting

The following theorem states that if the growth factor $\rho = \frac{max_{i,j}|u_{ij}|}{max_{i,j}|a_{ij}|}$ is $O(1)$ uniformly for all matrices of given dimension m, only then Algorithm 2 as discussed above is backward stable.

Theorem 2. *Let the factorization* $\mathbf{PA} = \mathbf{LU}$ *of a matrix* $\mathbf{A} \in \mathbf{R^{m\times m}}$ *be computed by Gaussian elimination with partial pivoting (ALgorithm 2) on a computer satisfying* (**??**) *vand* (3.3.1)*. Then the computed matrices* $\tilde{P}$,$\tilde{L}$ *and* $\tilde{U}$ *satisfy*

$$\tilde{L}\tilde{U} = \tilde{P}A + \Delta A\tilde{x} = b$$

with

$$\frac{\|\Delta A|}{\|A|} = O(\rho\epsilon_{machine}) \tag{}$$

for some $\delta A \in R^{m\times m}$*, where* ρ *is the growth factor for A.*

4.2.5 Implementation

The algorithms for LU decomposition using Gaussian Elimination with and without pivoting are implemented in R. Using these as helper functions, we implement functions for solving the least square normal equations along with demonstrations as follows:

Using Gaussian Elimination without pivoting:

```
#Function for computing LU decomposition of a non-singular square matrix (without pivoting)
LU=function (A) {

  ## check dimension
  n=dim(A)
  if (n[1] != n[2]) stop("'A' must be a square matrix")
  n=n[1]

  ## Gaussian elimination
  for (j in 1:(n - 1)) {

    ind=(j + 1):n

    ## check if the pivot is EXACTLY 0
    piv=A[j, j]
    if (piv == 0) stop(sprintf("system is exactly singular: U[%d, %d] = 0", j, j))

    l=A[ind, j] / piv

    ## update 'L' factor
    A[ind, j]=l

    ## update 'U' factor by Gaussian elimination
    A[ind, ind]=A[ind, ind] - tcrossprod(l, A[j, ind])

    }
  L=diag(1, nrow(A), ncol(A))
  low=lower.tri(L)
  L[low]=A[low]
  U=A[1:nrow(A), ]
  U[low]=0
  list(L = L, U = U)
  }
```

Using Gaussian Elimination with pivoting:

```
#Function for computing LU decomposition of a non-singular square matrix (with pivoting)
LUP=function (A) {

  ## check dimension
  n=dim(A)
  if (n[1] != n[2]) stop("'A' must be a square matrix")
  n=n[1]

  ## initializing pivot vector
  pivot <- 1:n

  ## Gaussian elimination
  for (j in 1:(n-1)) {

    ## select pivot
    m <- which.max(abs(A[j:n, j]))

    ## A[j - 1 + m, j] is the pivot
    if (m > 1L) {
      ## row exchange
```

```
      tmp=A[j, ]
      A[j, ]=A[j - 1 + m, ]
      A[j - 1 + m, ]=tmp
      tmp=pivot[j]
      pivot[j]=pivot[j - 1 + m]
      pivot[j - 1 + m]=tmp
      }

    ind=(j + 1):n

    ## check if the pivot is EXACTLY 0
    piv=A[j, j]
    if (piv == 0) {
      stop(sprintf("system is exactly singular: U[%d, %d] = 0", j, j))
      }

    l=A[ind, j] / piv

    ## update `L` factor
    A[ind, j]=l

    ## update `U` factor by Gaussian elimination
    A[ind, ind]=A[ind, ind] - tcrossprod(l, A[j, ind])

    }

  ## add `pivot` as an attribute and return `A`
  A=structure(A, pivot = pivot)

  L=diag(1, nrow(A), ncol(A))
  low=lower.tri(L)
  L[low]=A[low]
  U <- A[1:nrow(A), ]  ## use "[" to drop attributes
  U[low] <- 0
  list(L = L, U = U, P = attr(A, "pivot"))
  }
```

Solver functions for normal equations using LU decomposition:

```
#Functions to solve least squares problem for matrix A and vector b using LU decomposition
    obtained by Gaussian Elimination with and without pivoting

#Without pivoting
solve_LU=function(A,b)
{
  M=crossprod(A)
  y=crossprod(A,b)
  decomp=LU(M)
  L=decomp$L
  U=decomp$U
  w=fwd_sub(L,y)
  x=back_sub(U,w)
  return(x)
}

#With pivoting
solve_LUP=function(A,b)
{
  M=crossprod(A)
  decomp=LUP(M)
  piv=decomp$P
  L=decomp$L
  U=decomp$U
  y=crossprod(A[piv,],b[piv])
  w=fwd_sub(L,y)
```

```
  x=back_sub(U,w)
  return(x[piv])
}
```

Demonstration:

```
#Demo of LU without pivoting (Compare with lu from Matrix package)
set.seed(0)
M=matrix(runif(9),ncol=3)
LU(M)
library(Matrix)
expand(lu(M))

#Demo of Lu with pivoting
A=matrix(runif(16),ncol=4)
LU(A)
lu(A)

B <- A[c(4, 3, 1, 2), ]
LU(B)
LUP(B)
LUP(B)$pivot
expand(lu(B))
```

Output:

```
> #Demo of LU without pivoting (Compare with lu from Matrix package)
> set.seed(0)
> M=matrix(runif(9),ncol=3)
> LU(M)
$L
          [,1]         [,2] [,3]
[1,] 1.0000000  0.00000000    0
[2,] 0.2960962  1.00000000    0
[3,] 0.4149939 -0.04880764    1

$U
          [,1]      [,2]      [,3]
[1,] 0.8966972 0.5728534 0.8983897
[2,] 0.0000000 0.7385881 0.6786655
[3,] 0.0000000 0.0000000 0.3210956

> library(Matrix)
> expand(lu(M))
$L
3 x 3 Matrix of class "dtrMatrix" (unitriangular)
     [,1]        [,2]        [,3]
[1,]  1.00000000           .           .
[2,]  0.29609623  1.00000000           .
[3,]  0.41499394 -0.04880764  1.00000000

$U
3 x 3 Matrix of class "dtrMatrix"
     [,1]      [,2]      [,3]
[1,] 0.8966972 0.5728534 0.8983897
[2,]         . 0.7385881 0.6786655
[3,]         .         . 0.3210956

$P
3 x 3 sparse Matrix of class "pMatrix"

[1,] | . .
[2,] . | .
[3,] . . |
```

```
>
> #Demo of Lu with pivoting
> A=matrix(runif(16),ncol=4)
> LU(A)
$L
           [,1]      [,2]      [,3] [,4]
[1,] 1.00000000 0.000000 0.000000    0
[2,] 0.09821156 1.000000 0.000000    0
[3,] 0.32740419 1.720958 1.000000    0
[4,] 0.28064348 0.962924 0.216023    1

$U
          [,1]       [,2]       [,3]       [,4]
[1,] 0.629114 0.6870228  0.7176185  0.9347052
[2,] 0.000000 0.3166301  0.9214277  0.1203437
[3,] 0.000000 0.0000000 -1.4406547  0.1385409
[4,] 0.000000 0.0000000  0.0000000 -0.2825737

> lu(A)
'MatrixFactorization' of Formal class 'denseLU' [package "Matrix"] with 4 slots
  ..@ x        : num [1:16] 0.6291 0.3274 0.0982 0.2806 0.687 ...
  ..@ perm     : int [1:4] 1 3 3 4
  ..@ Dimnames: List of 2
  .. ..$ : NULL
  .. ..$ : NULL
  ..@ Dim      : int [1:2] 4 4
>
> B <- A[c(4, 3, 1, 2), ]
> LU(B)
$L
          [,1]      [,2]       [,3] [,4]
[1,] 1.0000000  0.000000  0.0000000    0
[2,] 1.1666196  1.000000  0.0000000    0
[3,] 3.5632398 -5.741589  1.0000000    0
[4,] 0.3499513  1.109491 -0.2568827    1

$U
          [,1]      [,2]       [,3]      [,4]
[1,] 0.1765568 0.4976992  0.7774452 0.1255551
[2,] 0.0000000 0.1892157 -0.5269477 0.5051987
[3,] 0.0000000 0.0000000 -5.0781224 3.3879659
[4,] 0.0000000 0.0000000  0.0000000 0.4780007

> LUP(B)
$L
           [,1]      [,2]      [,3] [,4]
[1,] 1.00000000 0.0000000 0.0000000    0
[2,] 0.32740419 1.0000000 0.0000000    0
[3,] 0.09821156 0.5810716 1.0000000    0
[4,] 0.28064348 0.5595278 0.5911574    1

$U
          [,1]      [,2]      [,3]       [,4]
[1,] 0.629114 0.6870228 0.7176185  0.9347052
[2,] 0.000000 0.5449073 0.1450839  0.3456474
[3,] 0.000000 0.0000000 0.8371235 -0.0805022
[4,] 0.000000 0.0000000 0.0000000 -0.2825737

$P
[1] 3 2 4 1

> expand(lu(B))
$L
4 x 4 Matrix of class "dtrMatrix" (unitriangular)
```

```
      [,1]        [,2]        [,3]        [,4]
[1,] 1.00000000           .           .           .
[2,] 0.32740419 1.00000000           .           .
[3,] 0.09821156 0.58107160 1.00000000           .
[4,] 0.28064348 0.55952780 0.59115744 1.00000000

$U
4 x 4 Matrix of class "dtrMatrix"
      [,1]       [,2]       [,3]       [,4]
[1,]  0.6291140  0.6870228  0.7176185  0.9347052
[2,]           .  0.5449073  0.1450839  0.3456474
[3,]           .           .  0.8371235 -0.0805022
[4,]           .           .           . -0.2825737

$P
4 x 4 sparse Matrix of class "pMatrix"

[1,] . . . |
[2,] . | . .
[3,] | . . .
[4,] . . | .

> #Demo to show Solving using solve_LU or solve_LUP works(Compare with solve function in R)
> A=matrix(runif(9),ncol=3)
> b=4*A[,1]+2*A[,2]+9*A[,3]
> solve_LU(A,b)
      [,1]
[1,]    4
[2,]    2
[3,]    9
> solve_LUP(A,b)
[1] -279.8969  393.3669  132.8090
> solve(crossprod(A),crossprod(A,b))
      [,1]
[1,]    4
[2,]    2
[3,]    9
```

4.3 Symmetric Gaussian Elimination

We turn now to the problem of decomposing a hermitian positive definite matrix into triangular factors. To begin, consider what happens if a single step of Gaussian elimination is applied to a hermitian matrix A with a 1 in the upper-left position:

$$A = \begin{bmatrix} 1 & w^T \\ w & K \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix} \begin{bmatrix} 1 & w^T \\ 0 & K - ww^T \end{bmatrix}$$

Zeros have been introduced into the first column of the matrix by an elementary lower-triangular operation on the left that subtracts multiples of the first row from subsequent rows. Gaussian elimination would now continue the reduction to triangular form by introducing zeros in the second column. However, in order to maintain symmetry, Cholesky factorization first introduces zeros in the first row to match the zeros just introduced in the first column. We can do this by a right upper-triangular operation that subtracts multiples of the first column from the subsequent ones:

$$\begin{bmatrix} 1 & w^T \\ 0 & K - ww^T \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & K - ww^T \end{bmatrix} \begin{bmatrix} 1 & w^T \\ 0 & K - ww^T \end{bmatrix}$$

Note that this upper-triangular operation is exactly the adjoint of the lower- triangular operation that we used to introduce zeros in the first column. Combining the operations above, we find that the matrix A has

been factored into three terms:

$$\begin{bmatrix} 1 & w^T \\ w & K \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^T \end{bmatrix} \begin{bmatrix} 1 & w^T \\ 0 & I \end{bmatrix}$$

The idea of Cholesky factorization is to continue this process, zeroing one column and one row of A symmetrically until it is reduced to the identity.

4.4 Cholesky Factorization

In order for the symmetric triangular reduction to work in general, we need a factorization that works for any $a_{11} > 0$, not just $a_{11} = 1$. The generalization is accomplished by adjusting some of the elements of R_1 by a factor of $\sqrt{a_{11}}$. Let, $\alpha = \sqrt{a_{11}}$ and observe that :

$$\begin{bmatrix} a_{11} & w^T \\ w & K \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ \frac{w}{\alpha} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \frac{K - ww^T}{a_{11}} \end{bmatrix} \begin{bmatrix} \alpha & \frac{w^T}{\alpha} \\ 0 & I \end{bmatrix} = R_1^T A_1 R_1$$

This is the basic step that is applied repeatedly in Cholesky factorization. If the upper left entry of the matrix $K - \frac{ww^T}{a_{11}}$ is positive, the same formula can be used to factor it. Then we have,

$$A_1 = R_2^T A_2 R_2$$

which implies

$$A = R_1^T R_2^T A_2 R_2 R_1$$

. Continuing in this way, we can get the equation,

$$A = R_1^{'} R_2^{'} R_3^{'} ... R_m^{'} R_m ... R_3 R_2 R_1$$

This equation has the form,

$$A = R^T R$$

where, R is upper triangular.A reduction of this kind of Hermitian positive definite matrix is known as a Cholesky Factorization.The description above left one item dangling. How do we know that the upper-left entry of the submatrix $K - ww^T/a_{11}$ is positive? The answer is that it must be positive because $K - ww^T/a_{11}$ is positive definite, since it is the $(m|1) \times (m|1)$ lower-right principal submatrix of the positive definite matrix $R_1^{T-1} A R_1^{-1}$. By induction, the same argument shows that all the submatrices A_j that appear in the course of the factorization are positive definite, and thus the process cannot break down.Now, we state the following theorem.

Theorem 3. *Every Hermitian positive definite matrix $A \in R^{m \times m}$ has a unique Cholesky Factorization.*

Proof. Proof of existence is discussed above. Also, since the analogous quantities are determined at each step of the reduction, the entire factorization is unique. □

4.4.1 The Cholesky decomposition algorithm

The Cholesky decomposition algorithm was first proposed by Andre-Louis Cholesky at the end of the First World War shortly before he was killed in battle. He was a French military officer and mathematician. The idea of this algorithm was published in 1924 by his fellow officer and, later, was used by Banachiewicz. In the Russian mathematical literature, the Cholesky decomposition is also known as the square-root method due to the square root operations used in this decomposition and not used in Gaussian elimination.
Originally, the Cholesky decomposition was used only for dense real symmetric positive definite matrices. At present, the application of this decomposition is much wider. For example, it can also be employed for

the case of Hermitian matrices. In order to increase the computing performance, its block versions are often applied.
In the case of sparse matrices, the Cholesky decomposition is also widely used as the main stage of a direct method for solving linear systems. In order to reduce the memory requirements and the profile of the matrix, special reordering strategies are applied to minimize the number of arithmetic operations. A number of reordering strategies are used to identify the independent matrix blocks for parallel computing systems.
Here we consider the original version of the Cholesky decomposition for dense real symmetric positive definite matrices. For a number of other versions, however, the structure of this decomposition is almost the same (for example, for the complex case): the distinction consists in the change of the majority of real operations by the corresponding complex operations.
For positive definite Hermitian matrices (symmetric matrices in the real case), we use the decomposition $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$ where L is the lower triangular matrix, or the decomposition $\mathbf{A} = \mathbf{U}^{\mathbf{T}}\mathbf{U}$ where U is the upper triangular matrix. These forms of the Cholesky decomposition are equivalent in the sense of the amount of arithmetic operations and are different in the sense of data represntation. The essence of this decomposition consists in the implementation of formulas obtained uniquely for the elements of the matrix L from the above equality. The Cholesky decomposition is widely used due to the following features.

- **Symmetry of matrices** - The symmetry of a matrix allows one to store in computer memory slightly more than half the number of its elements and to reduce the number of operations by a factor of two compared to Gaussian elimination. Note that the LU-decomposition does not require the square-root operations when using the property of symmetry and, hence, is somewhat faster than the Cholesky decomposition, but requires to store the entire matrix.

- **Accumulation Mode** - The Cholesky decomposition allows one to use the so-called accumulation mode due to the fact that the significant part of computation involves dot product operations. Hence, these dot products can be accumulated in double precision for additional accuracy. In this mode, the Cholesky method has the least equivalent perturbation. During the process of decomposition, no growth of the matrix elements can occur, since the matrix is symmetric and positive definite. Thus, the Cholesky algorithm is unconditionally stable.

Algorithm 3 Cholesky–Banachiewicz Algorithm

Input : $m \times m$ matrix A with (i, j)-th element a_{ij}
Condition : A must be symmetric positive definite

1: U=A, L=I
2: **for** k=1 **to** m-1 **do**
3: $l_{kk} = \sqrt{u_k k}$
$l_{k+1:m,k} = u_{k+1:m,k} - l/l_{kk}$
$u_{k+1:m,k+1:m} = u_{k+1:m,k+1:m} - l_{kk} l_{kk}^T$
4: **end for**
5: $l_{nn} = \sqrt{u_{nn}}$

4.4.2 Stability of Cholesky Factorization

All of the subtleties of the stability analysis of Gaussian elimination vanish for Cholesky factorization. This algorithm is always stable. Intuitively, the reason is that the factors R can never grow large. In the 2-norm, for example, we have $||R|| = ||R^{'}|| = ||A||$. Thus, numbers much larger than the entries of A can never arise.Note that the stability of Cholesky factorization is achieved without the need for any pivoting. Intuitively, one may observe that this is related to the fact that most of the weight of a hermitian positive definite matrix is on the diagonal. For example, it is not hard to show that the largest entry must appear on

the diagonal, and this property carries over to the positive definite submatrices constructed in the inductive process . An analysis of the stability of the Cholesky process leads to the following backward stability result.

Theorem 4. *Let, $A \in R^{m\times m}$ be a Hermitian positive definite matrix, and let a Cholesky Decomposition of A be computed by the above algorithm on a computer satisfies* (**??**) *and* (3.3.1). *For all sufficiently small $\epsilon_{machine}$, this process is guaranteed to run to completion (i.e., no zero or negative corner entries r_{kk} will arise), generating a computed factor $\tilde{R}$ that satisfies*

$$\tilde{R}^T\tilde{R} = A + \delta A$$

where, $\frac{||\delta A||}{||A||} = O(\epsilon_{machine})$, for some $\delta A \in R^{m\times m}$.

Like so many algorithms of numerical linear algebra, this one would look much worse if we tried to carry out a forward error analysis rather than a backward one. If A is ill-conditioned, $\tilde{R}$ will not generally be close to R; the best we can say is $\frac{||\tilde{R}-R||}{||R||} = O(\kappa(A)\epsilon_{machine})$. (In other words, Cholesky factorization is in general an ill-conditioned problem.) It is only the product RTR that satisfies the much better error bound). Thus the errors introduced in R by rounding are large .

4.4.3 Solution of $Ax = b$ and least squares normal equation using Cholesky decomposition

If A is hermitian positive definite, the standard way to solve a system of equations $Ax = b$ is by Cholesky factorization. The previous algorithm reduces the system to $R^TRx = b$, and we then solve two triangular systems in succession: first $R^Ty = b$ for the unknown y, then $Rx = y$ for the unknown x.
By reasoning, it can be shown that this process is backward stable.

Theorem 5. *The solution of hermitian positive definite systems $Ax = b$ via Cholesky factorization is backward stable generating a computed solution $\tilde{x}$ that satisfies*

$$(A + \Delta A)\tilde{x} = b$$

, where, $\frac{||\Delta A||}{||A||} = O(\epsilon_{machine})$ for some $\Delta A \in C^{mm}$

Having obtained the Cholesky decomposition of A, or A^TA in the case of solving the least square problem $\mathbf{Ax} = \mathbf{b}$, we can proceed as follows:

1. Form the matrix A*A and the vector A* b.
2. Compute the Cholesky factorization A*A = R*R.
3. Compute the Cholesky factorization A*A = R*R.
4. Solve the upper-triangular system Rx = w for x.

4.4.4 Implementation of Cholesky Decompsoition

The Cholesky-Banachiewicz Algorithm is implemented in R and used as a helper function to create a solver function for the least square normal equations. Demonstrations are also given below:

Cholesky Decomposition:

```
#Cholesky Decomposition based on Cholesky Banachiewicz Algorithm
chol_decomp=function(A)
{
  n=dim(A)
  if(n[1]!=n[2]) stop("A is not a square matrix")
  n=n[1]
  L=diag(rep(1,n))
  for(k in 1:n-1)
  {
    L[k,k]=sqrt(A[k,k])
    L[(k+1):n,k]=as.vector(A[(k+1):n,k])/L[k,k]
    A[(k+1):n,(k+1):n]=A[(k+1):n,(k+1):n]-tcrossprod(L[(k+1):n,k])
  }
  L[n,n]=sqrt(A[n,n])
  return(L)
}

#Function to solve least squares problem for matrix A and vector b using Cholesky
    decomposition
solve_chol=function(A,b)
{
  M=crossprod(A)
  y=crossprod(A,b)
  L=chol_decomp(M)
  U=t(L)
  w=fwd_sub(L,y)
  x=back_sub(U,w)
  return(x)
}
```

Demonstrations:

```
> #Demo to show Cholesky decomposition works (compare with chol function)
> A=matrix(runif(9),ncol=3)
> B=crossprod(A)
> chol(B)
          [,1]      [,2]      [,3]
[1,] 1.079848 0.6676754 1.1004826
[2,] 0.000000 0.8487850 0.1752928
[3,] 0.000000 0.0000000 0.1414100
> chol_decomp(B)
           [,1]      [,2]    [,3]
[1,] 1.0798479 0.0000000 0.00000
[2,] 0.6676754 0.8487850 0.00000
[3,] 1.1004826 0.1752928 0.14141
> t(chol(B))%*%chol(B)
          [,1]      [,2]      [,3]
[1,] 1.1660714 0.7209878 1.188354
[2,] 0.7209878 1.1662263 0.883551
[3,] 1.1883538 0.8835510 1.261786
> chol_decomp(B)%*%t(chol_decomp(B))
          [,1]      [,2]      [,3]
[1,] 1.1660714 0.7209878 1.188354
[2,] 0.7209878 1.1662263 0.883551
[3,] 1.1883538 0.8835510 1.261786
> B
          [,1]      [,2]      [,3]
[1,] 1.1660714 0.7209878 1.188354
```

```
[2,] 0.7209878 1.1662263 0.883551
[3,] 1.1883538 0.8835510 1.261786
> #Demo to show Solving using solve_chol works(Compare with solve function in R)
> A=matrix(runif(9),ncol=3)
> b=4*A[,1]+2*A[,2]+9*A[,3]
> solve_chol(A,b)
     [,1]
[1,]    4
[2,]    2
[3,]    9
> solve(crossprod(A),crossprod(A,b))
     [,1]
[1,]    4
[2,]    2
[3,]    9
```

4.5 QR decomposition

In linear algebra, a QR decomposition, also known as a QR factorization or QU factorization is a decomposition of a matrix A into a product $A = QR$ of an orthogonal matrix Q and an upper triangular matrix R.

Full QR Factorization

A real $m \times n$ matrix M with m n can be factored as the product of $m \times m$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R. Such a factorization is called a Full QR Factorization of M.

A matrix is called Orthogonal if it is inverse of its transposes. By $m \times n$ upper triangular matrix with $m \geq n$ we mean a matrix whose top $n \times n$ block is upper triangular and bottom $(m-n) \times (m-n)$ block consists of zeros.

(Reduced QR Factorization)

If M is a $m \times n$ real matrix with $m \geq n$, then it can be factored into product of two matrices $\hat{Q}$ and $\hat{R}$, where $\hat{Q}$ is $m \times n$ with orthonormal columns and $\hat{R}$ an upper triangular matrix of dimension $m \times n$.

There are several methods to compute the QR Factorization of real matrices with number of rows greater than or equal to number of columns.Some of the popular methods are based on Gram-Schmidt Orthogonalization, Householder Triangularization/Transformation and Givens rotation Method.

4.5.1 Gram Schmidt Orthogonalisation

It is a very popular method of orthogonalising a set of vectors. The process works like this. At the j^{th} step we wish to find a unit vector $q_j \in < a_1, a_2, ..., a_j >$ that is orthogonal to $q_l, q_2, \ldots, q_{j-1}$ where the notation <> represents the subspace spanned by the vectors within the brackets.

Let us define,

$$v_j = a_j - (q_1^T aj)q_1 - (q_2^T aj)q_2 - (q_3^T aj)q_3 - ... - (q_{j-1}^T aj)q_{j-1}, \quad j = 1, 2, 3, \ldots, n \tag{4.5.1.1}$$

Now we define,

$$q_j = \frac{v_j}{||v_j||}, \quad j = 1, 2, 3, \ldots, n \tag{4.5.1.2}$$

Then $q_1, q_2, ..., q_n$ are the orthonormal vectors that span the same vector space as by $a_1, a_2, ..., a_n$.

The Gram-Schmidt iteration is the basis of one of the two principal numerical algorithms for computing QR factorizations. It is a process of "triangular orthogonalization," making the columns of a matrix orthonormal via a sequence of matrix operations that can be interpreted as multiplication on the right by upper-triangular matrices.

4.5.2 Gram-Schmidt Projections

Let, $A \in R^{m \times n}$ be a full rank matrix with columns $\{a_j\}$
Let us define,

$$q_i = \frac{P_i a_i}{||P_i a_i||} \quad i = 1, 2, \ldots, n \tag{4.5.2.1}$$

In these formulae, each P_j denotes an orthogonal projector.Specifically, P_j is the mXm matrix of rank $m-(j-1)$ that projects R^m orthogonally onto the space orthogonal to $< q_1, q_2, \ldots, q_{j-1} >$. Now, observe that, q_j as defined above is orthogonal to $q_1, q_2, \ldots, q_{j-1}$, lies in the space $< a_1, a_2, \ldots, a_j >$ and has norm 1. Thus, we see that (4.5.1.2) and (4.5.2.1) are equivalent.

4.5.3 Classical Gram-Schmidt Algorithm

When this process is implemented on a computer, the column vectors of the matrix Q are often not quite orthogonal, due to rounding errors. For the Gram–Schmidt process as described above (sometimes referred to as "classical Gram–Schmidt") this loss of orthogonality is particularly bad; therefore, it is said that the (classical) Gram–Schmidt process is numerically unstable.
The Gram–Schmidt process can be stabilized by a small modification; this version is sometimes referred to as modified Gram-Schmidt or MGS. This approach gives the same result as the original formula in exact arithmetic and introduces smaller errors in finite-precision arithmetic.

4.5.4 Modified Gram-Schmidt Algorithm

In practice, the Gram-Schmidt formulas are not applied as we have indicated above., for this sequence of calculations turns out to be numerically unstable. Fortunately, there is a simple modification that improves matters. The previous method (Classical Gram-Schmidt Algorithm) computes a single orthogonal projection of rank $m-(j-1)$,

$$v_j = P_j a_j \tag{4.5.4.1}$$

.
In contrast, the modified Gram-Schmidt Algorithm computes the same result by a sequence of $j-1$ projections of rank $m-1$.Suppose, $P_{\dagger q}$ denote the rank $m-1$ orthogonal projector onto a space orthogonal to a nonzero vector $q \in C^m$.We can see that,

$$P_j = P_{\dagger q_{j-1}} P_{\dagger q_{j-2}} P_{\dagger q_{j-3}} ... P_{\dagger q_1} a_j \quad j = 1, 2, 3, \ldots, n \tag{4.5.4.2}$$

The Modified Gram-Schmidt Algorithm is based on the use of (4.5.4.2) instead of the previous method. Mathematically, the two methods are equivalent. However, the sequences of arithmetic operations implied by these formulas are different. The modified algorithm calculates v_j by evaluating the following formulas in order.

$$v_j^{(1)} = a_j$$

and

$$v_j^{(k)} = P_{\dagger q_{k-1}} v_j^{(k-1)} \quad k = 2, \ldots, j$$

In a finite precision computer, this method produces smaller errors.

4.5.5 Implementation of QR decomposition based on Gram-Schmidt Orthogonalization

The Classical and Modified Gram-Schmidt Algorithms for QR factorization are implemented in R and used as helper functions to create solver functions for solving the least squares problem. Demonstrations are also given as below :

Classical Gram Schmidt:

```
#QR Decomposition based on Classical Gram Schmidt Algorithm
qr.GSc <- function(X) {

  # Get the number of rows and columns of the matrix
  n=ncol(X)
  m=nrow(X)

  # Initialize the Q and R matrices
  Q=matrix(0, nrow=m, ncol=n)
  R=matrix(0, nrow=n, ncol=n)

  for (j in 1:n) {
    v = X[,j] # Step 1 of the Gram-Schmidt process v1 = a1
    # Skip the first column
    if (j > 1) {
      for (i in 1:(j-1)) {
        R[i,j] = crossprod(Q[,i],X[,j]) # Find the inner product (noted to be q^T a earlier)
        # Subtract the projection from v which causes v to become perpendicular to all
            columns of Q
        v = v - R[i,j] * Q[,i]
      }
    }
    # Find the L2 norm of the jth diagonal of R
    R[j,j] = sqrt(sum(v^2))
    # The orthogonalized result is found and stored in the ith column of Q.
    Q[,j] = v / R[j,j]
  }

  # Collect the Q and R matrices into a list and return
  qrcomp = list('Q'=Q, 'R'=R)
  return(qrcomp)
}

#Function to solve least squares problem for matrix A and vector b using QR decomposition
    based on Classical Gram Schmidt Algorithm

solve_qr.GSc=function(A,b)
{
  QR=qr.GSc(A)
  Q=QR$Q
  R=QR$R
  y=crossprod(Q,b)
  x=back_sub(R,y)
  return(x)
}
```

Modified Gram Schmidt:

```
#QR Decomposition based on Modified Gram Schmidt Algorithm
qr.GSm <- function(X) {

  # Get the number of rows and columns of the matrix
  n=ncol(X)
  m=nrow(X)

  # Initialize the Q and R matrices
  Q=X
```

```
  R=matrix(0, nrow=n, ncol=n)

  for (i in 1:(n-1)) {
    R[i,i] = sqrt(sum((Q[,i])^2))
    Q[,i]=as.vector(Q[,i])/R[i,i] #Normalize the i-th column of Q
      for (j in (i+1):n) {
        R[i,j] = crossprod(Q[,i],X[,j])
        Q[,j] = Q[,j] - R[i,j]*Q[,i]  #Orthogonalize the j-th column of Q
      }
  }
  R[n,n]=sqrt(sum((Q[,n])^2))
  Q[,n] = Q[,n]/R[n,n]

  # Collect the Q and R matrices into a list and return
  qrcomp = list('Q'=Q, 'R'=R)
  return(qrcomp)
}

#Function to solve least squares problem for matrix A and vector b using QR decomposition
    based on Modified Gram Schmidt Algorithm
solve_qr.GSm=function(A,b)
{
  QR=qr.GSm(A)
  Q=QR$Q
  R=QR$R
  y=crossprod(Q,b)
  x=back_sub(R,y)
  return(x)
}
```

Demonstrations:

```
#Demo for QR decomposition using Classical / Modified Gram Schmidt (Compare with qr in R)
qr.GSc(A)
qr.GSm(A)
qr.Q(qr(A),complete=T)
qr.R(qr(A),complete=T)
```

Output:

```
> qr.GSc(A)
$Q
           [,1]       [,2]       [,3]
[1,] 0.7148847  0.4748594  0.51327229
[2,] 0.5049695 -0.8583489  0.09079027
[3,] 0.4836793  0.1942823 -0.85341003

$R
          [,1]      [,2]       [,3]
[1,] 1.095188 0.8069065  1.10434292
[2,] 0.000000 0.4475243 -0.15406629
[3,] 0.000000 0.0000000  0.03116425

> qr.GSm(A)
$Q
           [,1]       [,2]       [,3]
[1,] 0.7148847  0.4748594  0.51327229
[2,] 0.5049695 -0.8583489  0.09079027
[3,] 0.4836793  0.1942823 -0.85341003

$R
          [,1]      [,2]       [,3]
[1,] 1.095188 0.8069065  1.10434292
[2,] 0.000000 0.4475243 -0.15406629
[3,] 0.000000 0.0000000  0.03116425
```

```
> qr.Q(qr(A),complete=T)
             [,1]         [,2]         [,3]
[1,] -0.7148847  0.4748594 -0.51327229
[2,] -0.5049695 -0.8583489 -0.09079027
[3,] -0.4836793  0.1942823  0.85341003
> qr.R(qr(A),complete=T)
           [,1]         [,2]        [,3]
[1,] -1.095188 -0.8069065 -1.10434292
[2,]  0.000000  0.4475243 -0.15406629
[3,]  0.000000  0.0000000 -0.03116425

> #Demo to show Solving using solve_GSc or solve_GSm works(Compare with solve function in R)
> A=matrix(runif(9),ncol=3)
> b=4*A[,1]+2*A[,2]+9*A[,3]
> solve_qr.GSc(A,b)
     [,1]
[1,]    4
[2,]    2
[3,]    9
> solve_qr.GSm(A,b)
     [,1]
[1,]    4
[2,]    2
[3,]    9

> solve(crossprod(A),crossprod(A,b))
     [,1]
[1,]    4
[2,]    2
[3,]    9
```

4.5.6 Householder Transformation

A Householder reflection (or Householder transformation) is a transformation that takes a vector and reflects it about some plane or hyperplane. We can use this operation to calculate the QR factorization of an m-by-n matrix A with m n.
Q can be used to reflect a vector in such a way that all coordinates but one disappear.
Let $\mathbf{x}$ be an arbitrary real m-dimensional column vector of A such that $\|\mathbf{x}\| = |\alpha|$ for a scalar α. If the algorithm is implemented using floating-point arithmetic, then αshould get the opposite sign as the k-th coordinate of $\mathbf{x}$, where x_k is to be the pivot coordinate after which all entries are 0 in matrix A's final upper triangular form, to avoid loss of significance.
Then, where $\mathbf{e}_1$ is the vector $(10\dots0)^T$, $\|\Delta\|$ is the Euclidean norm and I is an m-by-m identity matrix, set
$\mathbf{u} = \mathbf{x} - \alpha\mathbf{e}_1$,
$\mathbf{v} = \dfrac{\mathbf{u}}{\|\mathbf{u}\|}$, .
$Q = I - 2\mathbf{v}\mathbf{v}^\mathsf{T}$.
Q is an m-by-m Householder matrix and

$$Q\mathbf{x} = \begin{pmatrix}\alpha & 0 & \cdots & 0\end{pmatrix}^\mathsf{T}.$$

This can be used to gradually transform an m-by-n matrix A to upper triangular form. First, we multiply A with the Householder matrix Q_1 we obtain when we choose the first matrix column for x. This results in a matrix Q_1A with zeros in the left column (except for the first row).

$$Q_1 A = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

This can be repeated for A' obtained from $Q_1 A$ by deleting the first row and first column), resulting in a Householder matrix $Q_2^{'}$. Note that $Q_2^{'}$ is smaller than Q1. Since we want it really to operate on $Q_1 A$ instead of A' we need to expand it to the upper left, filling in a 1, or in general:

$$Q_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & Q_k' \end{pmatrix}$$

After t iterations of this process, where

$$t = \min(m-1, n)$$

, $R = Q_t \cdots Q_2 Q_1 A$ is an upper triangular matrix. So, with $Q = Q_1^{\mathsf{T}} Q_2^{\mathsf{T}} \dots Q_t^{\mathsf{T}}$, $A = QR$ is a QR decomposition of A.

This method has greater numerical stability than the Gram–Schmidt method above.

4.5.7 Implementation of Householder Transformation

QR factorization using Householder Transformation is implemented in R and used as a helper function to create a solver function for solving the least squares problem. Demonstrations are also given as below :
Householder Transformation:

```
#Function to compute QR Factorization using Householder TRansformation
qr.householder <- function(A) {
  require(Matrix)

  R <- as.matrix(A) # Set R to the input matrix A

  n <- ncol(A)
  m <- nrow(A)
  H <- list() # Initialize a list to store the computed H matrices to calculate Q later

  for (k in 1:min(m,n)) {
    x <- R[k:m,k] # Equivalent to a_1
    e <- as.matrix(c(1, rep(0, length(x)-1)))
    vk <- sign(x[1]) * sqrt(sum(x^2)) * e + x

    # Compute the H matrix
    hk <- diag(length(x)) - 2 * tcrossprod(vk) / as.numeric(crossprod(vk))
    if (k > 1) {
      hk <- bdiag(diag(k-1), hk)
    }

    # Store the H matrix to find Q at the end of iteration
    H[[k]] <- hk

    R <- hk %*% R
  }

  Q <- Reduce("%*%", H) # Calculate Q matrix by multiplying all H matrices
  res <- list('Q'=as.matrix(Q), 'R'=as.matrix(R))
  return(res)
}

```

```
#Function to solve least squares problem for matrix A and vector b using QR decomposition
    obtained by Householder transformation
solve_qr.householder=function(A,b)
{
  QR=qr.householder(A)
  Q=QR$Q
  R=QR$R
  y=crossprod(Q,b)
  x=back_sub(R,y)
  return(x)
}
```

Demonstrations:

```
#Demo for QR decomposition using Householder Transformation (Compare with qr in R)
qr.householder(A)
qr.Q(qr(A),complete=T)
qr.R(qr(A),complete=T)
```

Output:

```
> qr.householder(A)
$Q
             [,1]         [,2]         [,3]
[1,] -0.7148847  0.4748594  0.51327229
[2,] -0.5049695 -0.8583489  0.09079027
[3,] -0.4836793  0.1942823 -0.85341003

$R
                [,1]           [,2]         [,3]
[1,] -1.095188e+00 -8.069065e-01 -1.10434292
[2,] -2.770499e-17  4.475243e-01 -0.15406629
[3,] -1.675029e-18  3.469447e-18  0.03116425

> qr.Q(qr(A),complete=T)
             [,1]         [,2]         [,3]
[1,] -0.7148847  0.4748594 -0.51327229
[2,] -0.5049695 -0.8583489 -0.09079027
[3,] -0.4836793  0.1942823  0.85341003
> qr.R(qr(A),complete=T)
           [,1]         [,2]         [,3]
[1,] -1.095188 -0.8069065 -1.10434292
[2,]  0.000000  0.4475243 -0.15406629
[3,]  0.000000  0.0000000 -0.03116425

> solve_qr.householder(A,b)
     [,1]
[1,]    4
[2,]    2
[3,]    9

> solve(crossprod(A),crossprod(A,b))
     [,1]
[1,]    4
[2,]    2
[3,]    9
```

4.5.8 Given's Rotation Method

QR decompositions can also be computed with a series of Givens rotations. Each rotation zeroes an element in the subdiagonal of the matrix, forming the R matrix. The concatenation of all the Givens rotations forms the orthogonal Q matrix.

In practice, Givens rotations are not actually performed by building a whole matrix and doing a matrix multiplication. A Givens rotation procedure is used instead which does the equivalent of the sparse Givens matrix multiplication, without the extra work of handling the sparse elements. The Givens rotation procedure is useful in situations where only a relatively few off diagonal elements need to be zeroed, and is more easily parallelized than Householder transformations.

4.5.9 Implementation of Given's Rotation Method

QR factorization using Given's Rotation Method is implemented in R and used as a helper function to create a solver function for solving the least squares problem. Demonstrations are also given as below :
Given's Rotation Method:

```
#Helper function for calculating the Given's Rotation Matrix
givens=function(a,b)
{
  max=max(a,b)
  min=min(a,b)
  r=abs(max)*sqrt(1+(min/max)^2)
  c=a/r
  s=-b/r
  return(list('c'=c,'s'=s,'r'=r))
}

# Function to compute QR decomposition using Given's Rotation Method
qr.givens=function(A)
{
  n=dim(A)[2]
  m=dim(A)[1]
  P=diag(1,nrow=n,ncol=m)
  for(i in 1:(n-1))
  {
    for(j in (i+1):m)
    {
      if(A[i,i]==0 & A[j,i]==0)
      {
        P_j=diag(1,nrow=n,ncol=m)
      }
      else
      {
        giv=givens(A[i,i],A[j,i])
        c=giv$c
        s=giv$s
        P_j=diag(1,nrow=n,ncol=m)
        P_j[i,i]=c
        P_j[j,j]=c
        P_j[i,j]=-s
        P_j[j,i]=s
      }
      A=P_j %*% A
      P=P_j %*% P
    }
  }
  R=A
  Q=t(P)
  qrcomp = list('Q'=Q, 'R'=R)
  return(qrcomp)
}

#Function to solve least squares problem for matrix A and vector b using QR decomposition
    obtained by Given's Rotation Method
solve_qr.givens=function(A,b)
{
```

```
  QR=qr.givens(A)
  Q=QR$Q
  R=QR$R
  y=crossprod(Q,b)
  x=back_sub(R,y)
  return(x)
}
```

Demonstrations: Demo for QR decomposition using Given's Rotation Method (Compare with qr in R) qr.givens(A) qr.Q(qr(A),complete=T) qr.R(qr(A),complete=T) **Output:**

```
> qr.givens(A)
$Q
           [,1]       [,2]       [,3]
[1,] 0.7148847  0.4748594  0.51327229
[2,] 0.5049695 -0.8583489  0.09079027
[3,] 0.4836793  0.1942823 -0.85341003

$R
                [,1]       [,2]       [,3]
[1,]  1.095188e+00 0.8069065  1.10434292
[2,] -4.180421e-17 0.4475243 -0.15406629
[3,] -6.644835e-17 0.0000000  0.03116425

> qr.Q(qr(A),complete=T)
            [,1]       [,2]       [,3]
[1,] -0.7148847  0.4748594 -0.51327229
[2,] -0.5049695 -0.8583489 -0.09079027
[3,] -0.4836793  0.1942823  0.85341003
> qr.R(qr(A),complete=T)
           [,1]       [,2]       [,3]
[1,] -1.095188 -0.8069065 -1.10434292
[2,]  0.000000  0.4475243 -0.15406629
[3,]  0.000000  0.0000000 -0.03116425

> solve_qr.givens(A,b)
     [,1]
[1,]    4
[2,]    2
[3,]    9

> solve(crossprod(A),crossprod(A,b))
     [,1]
[1,]    4
[2,]    2
[3,]    9
```

5 Summary

To summarize, we can say that the QR decomposition method is numerically more stable than the Gaussian Elimination Method . Also, among the various algorithms available for the QR decomposition , the Given's Rotation Method and the Householder Triangularization method are more stable than the Gram-Schmidt method.