# CHAPTER 2. LINEAR EQUATIONS

This chapter is based on Chapter 2 of the book by Miranda and Fackler.

In a linear equation a *nxn* matrix *A* and a *n*-vector *b* are given, and one must compute the *n*-vector *x* that satisfies *Ax=b*.

There are few linear models in economics. However, linear equations arise as elementary tasks in solution procedures designed to solve more complicated nonlinear models. For example, a static general equilibrium model may be represented by a system of nonlinear equations, which can be solved using Newton's method, which involves solving a sequence of linear equations. The Euler functional equation of a rational expectations model may be solved using a collocation method, which yields a nonlinear equation that in turn is solved as a sequence of linear equations.

## 1. L-U Factorization

Some linear equations *Ax=b* are relatively easy to solve. For example, if *A* is a lower triangular matrix

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

Then, the elements of *x* can be computed recursively using *forward substitution*

$$x_1 = \frac{b_1}{a_{11}}$$

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}$$

$$x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}}$$

…

$$x_n = \frac{b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{nn-1}x_{n-1}}{a_{nn}}$$

This computation can easily be implemented in MATLAB

```
>> for i=1:length(b)

x(i)=(b(i)-A(i,1:i-1)*x(1:i-1))/A(i,i);

end
```

If *A* is an upper triangular matrix, then the elements of *x* can be computed recursively using *backward substitution.*

Most linear equations encountered in practice, do not have a triangular *A* matrix. In such cases, the linear equation is often best solved using the *L-U factorization algorithm*. The L-U algorithm is designed to decompose the *A* matrix into the product of lower and upper triangular matrices, allowing the linear equation to be solved using a combination of backward and forward substitution. The L-U algorithm involves two phases.

In the first *factorization* phase, Gaussian elimination is used to factor the matrix *A* into the product

*A*=*LU*

of a row-permuted lower triangular matrix $L$ and an upper-triangular matrix $U$. A row-permuted lower triangular matrix is simply a lower triangular matrix that has had its rows rearranged.

In the *solution* phase of the L-U algorithm, the factored linear equation

$Ax=(LU)x=L(Ux)=b$

is solved by first solving

$Ly=b$

for $y$ using forward substitution, accounting for row permutations, and then solving

$Ux=y$

for $x$ using backward substitution.

Consider, for example, the linear equation $Ax=b$ where

```
A=[-3 2 3;-3 2 1;3 0 0];A

A =

   -3    2    3

   -3    2    1

    3    0    0



>> [L U]=lu(A); L,U

L =

    1    0    0
```

```
     1     0     1

    -1     1     0
```

```
U =

    -3     2     3

     0     2     3

     0     0    -2
```

The matrix *L* is row-permuted lower triangular; by interchanging the second and third rows, a lower diagonal matrix results. The matrix *U* is upper triangular. Solving $Ly=b$ for $y$ using forward substitution involves solving for $y_1$, then for $y_2$, and finally for $y_3$. Given the solution $y=[10\ 7\ -2]$, the linear equation $Ux=y$ can then be solved using backward substitution, yielding the solution of the original linear equation, $x=[-1\ 2\ 1]$.

For large $n$, it takes approximately $n^3/3+n^2$ long operations (multiplications and divisions) to solve a $nxn$ linear equation using L-U factorization. Explicitly computing the inverse of $A$ and then computing $A^{-1}b$ requires $n^3+n^2$ long operations. Using Cramer's rule requires approximately $(n+1)!$ long operations. To solve a 10x10 linear equations, L-U factorization requires 430 long operations, matrix inversion and multiplication requires 1,100 long operations, and Cramer's rule require 40 million long operations. Thus, the Cramer's rule is never used. If you have to solve the sequence of systems $Ax_j = b_j$, for $j=1,\ldots,m$, computing the inverse of $A$ and multiplying it by $b_j$, is more efficient if $m$ is greater than 3 or 4. Otherwise, the L-U factorization is more efficient.

In MATLAB the solution to the linear equation *Ax=b* is returned by the statement *x=A\b*. MATLAB uses the L-U factorization method, except when a simpler method is available (for instance if *A* is symmetric and positive definite, see further).

```
>> b=[10;8;-3];b

b =

    10

     8

    -3



>> x=A\b;x

x =

    -1

     2

     1
```

## 2. Gaussian Elimination

The L-U factors of a matrix *A* are computed using *Gaussian elimination*. The Gaussian elimination algorithm begins with matrices *L* and *U* initialized as *L=I* and *U=A*, where *I* is the identity matrix.

```
>> A=[2 0 -1 2;4 2 -1 4; 2 -2 -2 3; -2 2 7 -3];A
```

```
A =

     2     0    -1     2

     4     2    -1     4

     2    -2    -2     3

    -2     2     7    -3


>> L=eye(4);U=A;L,U

L =

     1     0     0     0

     0     1     0     0

     0     0     1     0

     0     0     0     1

U =

     2     0    -1     2

     4     2    -1     4

     2    -2    -2     3

    -2     2     7    -3
```

The first stage of Gaussian elimination is designed to nullify the subdiagonal entries of the first column of the *U* matrix. The *U* matrix is updated by subtracting 2 times the first row from the second, subtracting 1 time the first row from the third, and subtracting -1 times the first row from the fourth. Then, the subdiagonal entries of matrix *U* become zero. The *L* matrix, which initially equals the identity, is updated by storing the multipliers 2, 1 and -1 as the subdiagonal entries of its first column.

```
>> U(2,:)=U(2, :)-2*U(1,:); U(3,:)=U(3, :)-1*U(1,:); U(4,:)=U(4, :)-(-1)*U(1,:);U

U =

     2     0    -1     2

     0     2     1     0

     0    -2    -1     1

     0     2     6    -1

>> L(2,1)=2;L(3,1)=1;L(4,1)=-1;L

L =

     1     0     0     0

     2     1     0     0

     1     0     1     0

    -1     0     0     1
```

After the first stage of Gaussian elimination we still have *A=LU*.

The second stage elimination is designed to nullify the subdiagonal entries of the second column of the *U* matrix. The *U* matrix is updated by subtracting -1 times the *second* row from the third and subtracting 1 times the *second* row from the fourth. The *L* matrix is updated by storing the multipliers -1 and 1 as the subdiagonal elements of its second column.

```
>> U(3,:)=U(3, :)-(-1)*U(2,:); U(4,:)=U(4, :)-1*U(2,:);U

U =

     2     0    -1     2

     0     2     1     0

     0     0     0     1

     0     0     5    -1
```

```
>> L(3,2)=-1;L(4,2)=1;L

L =

     1     0     0     0

     2     1     0     0

     1    -1     1     0

    -1     1     0     1
```

After the second stage of Gaussian elimination we still have *A=LU*.

In the third stage of Gaussian elimination, one encounters an apparent problem. The third diagonal element of matrix *U* is zero, making it impossible to nullify the subdiagonal entry as before. This difficulty is easily remedied by interchanging the third and fourth rows of *U*. The *L* matrix is updated by interchanging the previously computed multipliers residing in the third and fourth columns.

```
>> v=U(3,:);U(3,:)=U(4,:);U(4,:)=v;U

U =

   2   0  -1   2

   0   2   1   0

   0   0   5  -1

   0   0   0   1


>> v=L(:,3);L(:,3)=L(:,4);L(:,4)=v;L

L =

   1   0   0   0

   2   1   0   0

   1  -1   0   1
```

```
    -1     1     1     0
```

The Gausssian elimination algorithm terminates with a permuted lower triangular matrix *L* and an upper triangular matrix *U* whose product is the matrix *A*.

```
> LU=L*U; LU, A

LU =

     2     0    -1     2

     4     2    -1     4

     2    -2    -2     3

    -2     2     7    -3

A =

     2     0    -1     2

     4     2    -1     4

     2    -2    -2     3

    -2     2     7    -3
```

In theory, Gaussian elimination will compute the L-U factors of any matrix *A*, provided *A* is invertible. If *A* is not invertible, Gaussian elimination will detect this fact by encountering a zero diagonal element in the *U* matrix that cannot be replaced with a nonzero element below it.

Gauss elimination uses additions and subtractions of rows of matrices, which may have been multiplied by elements of these matrices. If all the elements of matrix *A* are of the same order of magnitude (do not differ by more than 1 to 1,000), we will meet no problem. Otherwise, we would have to add or subtract numbers of very different magnitudes, and that may create problems. *Pivoting*, which means interchanging rows during Gaussian elimination in order to make the magnitude of diagonal elements as large as possible, substantially enhances the reliability and accuracy of a Gaussian elimination routine.

However, pivoting cannot cure all the problems caused by rounding error. Some linear equations are inherently difficult to solve accurately on a computer, despite pivoting. The difficulty occurs when the *A* matrix is structured in such a way that a small perturbation $\delta b$ in the vector *b* induces a large change $\delta x$ in the solution vector *x*. In such a case the *A* matrix is said to be *ill-conditioned.* We can compute the condition number of matrix *A* by typing

```
>> cond(A)

ans =

  51.8900
```

Numerical analysts often use the rough rule of thumb that for each power of 10 in the condition number, one significant digit is lost in the computed solution vector *x*. Thus, if *A* has a condition number of 1,000, the computed solution vector *x* will have about three fewer significant digits than vector *b*.

When, you write an economic model, you should be careful with the choices of the units. If you express the population in number of people, for instance 63,000,000 in France, and the inflation rate as .02, you will probably have ill-conditioned matrices when you simulate your model. The values of the variables should be of the same orders of magnitude. Thus, express population in millions of people and the inflation rate in percent: 63 and 2.

## 3. Cholesky Factorization

There are plenty of ways of decomposing a matrix, besides the L-U factorization. If matrix *A* is symmetric positive definite (a covariance matrix for instance) you can use the Cholesky factorization, which requires only half as many operations as general Gaussian elimination and has the added advantage that it is less vulnerable to rounding error and does not require pivoting.

Then we have

$A=U'U$

where *U* is upper triangular and *U'* is its transpose. Then, the linear equation

$Ax=U'Ux=U'(Ux)=b$

may be solved efficiently by using forward substitution to solve

$U'y=b$

and then using backward substitution to solve

$Ux=y$

The MATLAB \ operator will automatically employ Cholesky factorization, rather than L-U factorization, to solve the linear equation if it detects that *A* is symmetric positive definite.

```
>> A = pascal(6)


A =

     1     1     1     1     1     1

     1     2     3     4     5     6

     1     3     6    10    15    21

     1     4    10    20    35    56

     1     5    15    35    70   126

     1     6    21    56   126   252


>> R = chol(A)

R =

     1     1     1     1     1     1
     0     1     2     3     4     5

     0     0     1     3     6    10

     0     0     0     1     4    10

     0     0     0     0     1     5

     0     0     0     0     0     1


>> R'*R

ans =

     1     1     1     1     1     1

     1     2     3     4     5     6

     1     3     6    10    15    21

     1     4    10    20    35    56

     1     5    15    35    70   126

     1     6    21    56   126   252
```

## 4. Iterative Methods

Algorithm based on Gaussian elimination are called *exact* because they would generate exact solutions for the linear equation $Ax=b$ after a finite number of operations, if not for rounding error. Such methods are ideal for moderately sized linear equations but may be impractical for larger ones.

Other methods, called *iterative methods*, can often be used to solve large linear equations more efficiently if the $A$ matrix is sparse, that is, if $A$ is composed mostly of zeros entries. Iterative methods are designed to generate a sequence of increasingly accurate approximations to the solution of a linear equation, but they generally do not yield an exact solution after a prescribed number of steps, even in theory. We will see that these methods can easily be extended to nonlinear equations and so have been extremely used in macroeconomics.

We choose an easily invertible matrix $Q$ and write the linear equation in the equivalent form

$Qx=b+(Q-A)x$

or

$x=Q^{-1}b+(I- Q^{-1}A)x$

This form of the linear equation suggests the iteration rule

$x^{(k+1)}=Q^{-1}b+(I- Q^{-1}a)\, x^{(k)}$

which, if convergent, must converge to a solution of the linear equation.

Ideally, the so-called *splitting matrix* $Q$ will satisfy two criterias. First, $Q^{-1}b$ and $Q^{-1}A$ should be relatively easy to compute. This criteria is met if $Q$ is either diagonal or triangular. Second, the iterates should converge quickly to the true solution of the linear equation. If

$||\mathrm{I}- Q^{-1}A\,||<1$

n any matrix norm, then the iteration rule is a contractionary mapping and is guaranteed to converge to the solution of the linear equation from any initial value. The smaller the value of the matrix norm $\|I- Q^{-1}A\|<1$, the faster the guaranteed rate of convergence of the iterates when measured in the associated vector norm.

The two most popular iterative methods are the Gauss-Jacobi and Gauss-Seidel methods. The Gauss-Jacobi sets *Q* equal to the diagonal matrix formed from the diagonal entries of *A.* The Gauss-Seidel method sets *Q* equal to the upper triangular matrix formed from the upper-triangular elements of *A.* Both methods are guaranteed to converge from any starting value if *A* is diagonally dominant, that is if

$$\left|A_{ii}\right| > \sum_{\substack{i=1 \\ i\neq j}}^{n} \left|A_{ij}\right|, \qquad \forall i$$

Diagonally dominant matrices arise naturally in many computational economic applications.

Let us continue with an exercise

```
>> A=[54 14 -11 2;14 50 -4 29;-11 -4 55 22;2 29 22 95]; b=ones(4,1); A,b

A =

   54   14  -11    2

   14   50   -4   29

  -11   -4   55   22

    2   29   22   95
```

```
b =

     1

     1

     1

     1
```

The L-U factorization can be applied by

```
>> x=A\b;x

x =

    0.0189

    0.0168

    0.0234

   -0.0004
```

Let us compute the solution of the equation by using Gauss-Jacobi iteration

```
>> d=diag(A);x=b;

for it=1:30

dx=(b-A*x)./d;

x=x+dx;
```

```
if norm(dx)<0.0001, break, end

end

>> it, norm(dx),x



it =

    17

ans =

  7.9279e-005

x =

    0.0189

    0.0168

    0.0234

   -0.0004
```

We specified an initial guess for the solution of the linear equation, *b* (we could have specified zero). Iterations continue until the norm of the change *dx* falls below the specified convergence tolerance, set here to 0.0001 or until a specified number of allowable iterations set here to 30 are performed. It takes 17 iterations to the algorithm to reach a precision of four digits.

The following MATLAB script solves the same linear equation using Gauss Seidel iteration

```
>> Q=tril(A);x=b;

>> for it =1:30

dx=Q\(b-A*x);

x=x+dx;

if norm(dx)<0.0001, break, end

end

>> it, norm(dx), x


it =

    10

ans =

  5.8460e-005

x =

    0.0189

    0.0168

    0.0233

   -0.0004
```

This method has converged in a smaller number of iterations.

If we were lazy we would have used the two following commands from compecon

```
>> x=gjacobi(A,b)

x =

    0.0189

    0.0168

    0.0234

   -0.0004


>> x=gseidel(A,b)

x =

    0.0189

    0.0168

    0.0234

   -0.0004
```