Lecture 2

Linear Equations

Randall Romero Aguilar, PhD
II Semestre 2018
Last updated: August 26, 2018

Universidad de Costa Rica
SP6534 - Economía Computacional

Table of contents

Introduction

Linear equation

An n-dimensional linear equation takes the form

$$Ax = b$$

where

A	is a known $n \times n$ matrix
b	is a known $n \times 1$ vector
x	is an unknown $n \times 1$ vector to be determined

Linear equations are ubiquitous in computational economics

- Linear equations arise naturally in many applications:
    - Linear multicommodity market equilibrium models
    - Finite-state financial market models
    - Markov chain models
    - Ordinary least squares
- Linear equations, however, more often arise indirectly when numerically solving economic models involving nonlinear and functional equations:
    - Nonlinear multicommodity market models
    - Multiperson static game models
    - Dynamic optimization models
    - Rational expectations models

- Because linear equations are fundamental in computational economic applications, we study them carefully.
- In practice, we will often need to solve very large linear equations many times.
- Execution speed, storage requirements, and rounding error are important practical issues.

Gaussian Elimination

Gaussian Elimination

- A linear equation may be solved using Gaussian Elimination.
- Gaussian elimination employs elementary row operations:
    - Interchange two rows
    - Multiply a row by a nonzero constant
    - Add a nonzero multiple of one row to another
- Elementary row operations alter the form of a linear equation without changing its solution.

Example 1:

Gaussian elimination

- Let us use Gaussian elimination to solve the linear equation

$$\begin{bmatrix} 1 & 1 & 2 \\ 3 & 4 & 8 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 6 \end{bmatrix},$$

- … which may also be written

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ 3x_1 & + & 4x_2 & + & 8x_3 & = & 18 \\ 2x_1 & + & x_2 & + & x_3 & = & 6 \end{array}$$

Starting from

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ 3x_1 & + & 4x_2 & + & 8x_3 & = & 18 \\ 2x_1 & + & x_2 & + & x_3 & = & 6 \end{array}$$

Add -3 times row 1 to row 2

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ & & x_2 & + & 2x_3 & = & 3 \\ 2x_1 & + & x_2 & + & x_3 & = & 6 \end{array}$$

Add -2 times row 1 to row 3

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ & & x_2 & + & 2x_3 & = & 3 \\ & - & x_2 & - & 3x_3 & = & -4 \end{array}$$

Add row 2 to row 3

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ & & x_2 & + & 2x_3 & = & 3 \\ & & & - & x_3 & = & -1 \end{array}$$

Multiply row 3 by -1

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & 2x_3 & = & 5 \\ & & x_2 & + & 2x_3 & = & 3 \\ & & & & x_3 & = & 1 \end{array}$$

Solve by backward recursion

$x_3 = 1$

$x_2 = 3 - 2x_3 = 1$

$x_1 = 5 - x_2 - 2x_3 = 2$

Confirm the computed solution is correct by verifying that

$$\begin{bmatrix} 1 & 1 & 2 \\ 3 & 4 & 8 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 6 \end{bmatrix}$$

or, equivalently,

$$\begin{array}{ccccccc} 1 \cdot 2 & + & 1 \cdot 1 & + & 2 \cdot 1 & = & 5 \\ 3 \cdot 2 & + & 4 \cdot 1 & + & 8 \cdot 1 & = & 18 \\ 2 \cdot 2 & + & 1 \cdot 1 & + & 1 \cdot 1 & = & 6 \end{array}$$

- In the preceding example, we used elementary row operations to nullify sub-diagonal terms and transform the linear equation to unit upper diagonal form, making it easier to solve recursively.
- Gaussian elimination is implemented on a computer using an efficient computational and storage strategy called L-U factorization.

Why use Gaussian elimination to solve linear equations?

- Gaussian elimination is the most efficient known method for solving a general n-dimensional linear equation $Ax = b$.
- For large n, Gaussian elimination requires about $n^3/3 + n^2$ multiplication/division operations.
- Explicitly computing $A^{-1}b$ requires about $n^3 + n^2$ operations.
- Cramer's rule requires $(n+1)!$ operations.
- For $n = 10$, the number of operations are

Gaussian Elimination	430
Explicit Inverse	1,100
Cramer's Rule	40,000,000

- The numpy.linalg function `solve` uses Gaussian elimination to solve linear equations.
- For example, to solve the linear equation of the preceding example, execute the script

```
import numpy as np
from numpy.linalg import solve

A = np.array([[1, 1, 2],
              [3, 4, 8],
              [2, 1, 1]])
b = np.array([5, 18, 6])
x = solve(A, b)
print(x)
```

- This should return

```
[ 2.  1.  1.]
```

Rounding Error

Rounding Error

- A computer has finite storage and can represent only finitely many numbers exactly.
- Thus, exact arithmetic and computer arithmetic do not always agree.
- If you attempt to compute a number that cannot be represented exactly on a computer, the result will be rounded to the nearest representable number, introducing rounding error.
- In particular, when adding or subtracting two numbers of extremely different magnitudes, the smaller number is effectively ignored.

Example 2:

Roundig error

- In exact arithmetic

$$(\epsilon + 1) - 1 = \epsilon + (1 - 1) = \epsilon$$

- However, in Python computer arithmetic

```
e = 1e-20
x = (e + 1) - 1
y = e + (1 - 1)
```

- … will return

```
x = 0.0
y = 1e-20
```

Pivoting

- Rounding error can cause problems when solving linear equations.
- Consider the linear equation

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

  where $\epsilon = 10^{-17}$.
- One can easily verify that the exact solution is

$$\begin{aligned} x_1 &= \tfrac{1}{1-\epsilon}, && \text{which is slightly more than 1} \\ x_2 &= \tfrac{1-2\epsilon}{1-\epsilon}, && \text{which is slightly less than 1} \end{aligned}$$

- To solve the linear equation using Gaussian elimination, add $-1/\epsilon$ times the first row to the second row

$$\begin{bmatrix} \epsilon & 1 \\ 0 & 1 - 1/\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - 1/\epsilon \end{bmatrix}$$

- then solve recursively

$$\begin{aligned} x_2 &= \tfrac{2-1/\epsilon}{1-1/\epsilon} \\ x_1 &= \tfrac{1-x_2}{\epsilon} \end{aligned}$$

- If you compute x_1 and x_2 in this manner in Python,

```
e = 1e-17
x2 = (2-1/e) / (1-1/e)
x1 = (1-x2)/e
```

the operations return

```
x2 =  1.0
x1 =  0.0
```

- The computed value for x_1 is grossly inaccurate.
- What happened?

- In the first step of Gaussian elimination, we computed

$$x_2 = \frac{2 - 1/\epsilon}{1 - 1/\epsilon}$$

- However, since $1/\epsilon$ is very large compared to 1 or 2, rounding error was introduced, and the computer actually computed

$$x_2 = \frac{-1/\epsilon}{-1/\epsilon}$$

  which evaluated to exactly 1.
- We then computed

$$x_1 = \frac{1 - x_2}{\epsilon}$$

  which evaluated to exactly 0.

- Now solve the linear equation again by Gaussian elimination, but first interchange the two rows, which in theory will not affect the solution

$$\begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

- Now add $-\epsilon$ times the first row to the second row

$$\begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1-2\epsilon \end{bmatrix}$$

- then solve recursively

$$\begin{aligned} x_2 &= \tfrac{1-2\epsilon}{1-\epsilon} \\ \\ x_1 &= 2 - x_2 \end{aligned}$$

- If you compute x_1 and x_2 in this manner in Python

  ```
  e = 1e-17
  x2 = (1-2*e) / (1-e)
  x1 = 2 - x2
  ```

  the operations return

  ```
  x2 =  1.0
  x1 =  1.0
  ```

- The computed values for x_1 and x_2 are a little off, but are much more accurate than the first values we computed.
- Why did interchanging the two rows improve the accuracy of the computed solution?

- The inaccuracy of the first solution was due to rounding error caused by the very small magnitude of the diagonal element ϵ.
- By interchanging the two rows first, we brought a number of much larger magnitude into the diagonal, which reduced rounding error in subsequent computations.
- Interchanging rows to make the magnitude of the diagonal element as large as possible is called pivoting.
- Pivoting substantially enhances the computational accuracy of Gaussian elimination.
- All good linear solution solvers, including the Python backslash operator, employ pivoting.

Ill-Conditioning

Ill-Conditioning

- Consider the n-dimensional linear equation $Ax = b$.
- If small perturbations in b lead to disproportionately large changes in x, we say A is ill-conditioned or nearly singular.
- If A is ill-conditioned, unavoidable rounding errors in representation of b in a computer make it impossible to compute an accurate solution to $Ax = b$.
- Ill-conditioning is endemic to the matrix A and cannot be corrected with simple tricks such as pivoting.
- The only way to deal with ill-conditioning is to avoid it.

Ill-Conditioning and the condition number

- Ill-conditioning is measured by the condition number of A.
- The condition number is the maximum percentage change in the size of x per unit percentage change in the size of b.
- Technically, the condition number is the ratio its largest and smallest singular values.
- Rule of Thumb: Computed value of x loses one significant digit per power of 10 of the condition number of A.
- The condition number is always greater than or equal to 1.

An ill-conditioned matrix: Vandermonde

- Consider the notorious Vandermonde matrices.
- The $n \times n$ Vandermonde matrix has typical element

$$A_{ij} = i^{n-j}$$

- For example, for $n = 4$

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \\ 64 & 16 & 4 & 1 \end{bmatrix}$$

- Let us solve the linear equation

$$Ax = b$$

where A is the $n \times n$ Vandermonde matrix and b is the row-sum of A, that is, the $n \times 1$ vector with typical element

$$b_i = \sum_{j=1}^{n} A_{ij}$$

- By construction, the exact solution to this linear equation is an $n \times 1$ vector x containing all ones.

- To solve the linear equation and evaluate its precision, define the function `errorVander`

```
def errorVander(n):
    A = np.vander(np.arange(1,n+1))
    b = A @ np.ones(n)
    x = solve(A, b)
    error np.max(abs(x-1))
    return x, error
```

- Here, we compute the matrix A using the special numpy utility `vander` and we compute the maximum error among the elements of the computed solution.

- With $n = 4$, executing this function returns, as expected,

```
x =  [ 1.  1.  1.  1.]
error =  0.0
```

- With $n = 64$, however, executing the script returns

```
LinAlgError: Singular matrix
```

- Warning indicates A is ill-conditioned.

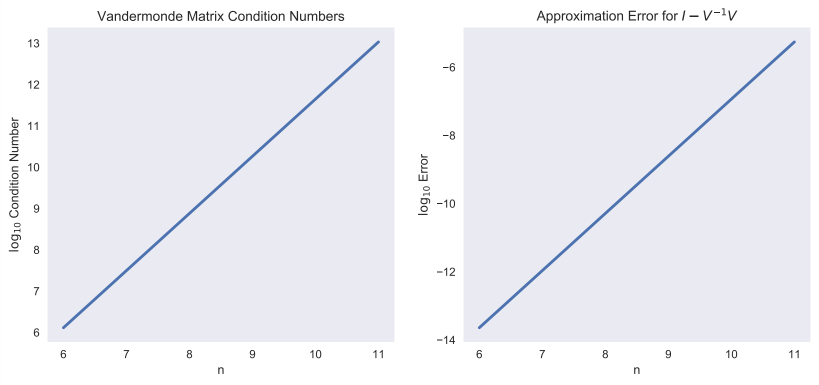

Figure 1: Ill-Conditioning of Vandermonde Matrices

Sparse Matrices

Sparse Matrices

- A sparse matrix is a matrix that consists mostly of zeros.
- Solving $Ax = b$ when A is sparse using conventional Gaussian elimination will consists mostly of meaningless, but costly, operations involving multiplication or addition with zero.
- Execution speed can often be dramatically increased by avoiding these useless operations.

- Scipy has special utilities for efficiently storing sparse matrices and operating with them.
- In particular, in scipy.sparse, `csr_matrix(A)` creates a version of the matrix `A` stored in a sparse matrix format, in which only the nonzero elements and their indices are explicitly stored.

- Execute the script

```python
import numpy as np
import scipy as sp

A = array([[0,0,0,5],
           [0,2,0,0],
           [0,0,0,0],
           [0,0,4,0]])
S = sp.sparse.csr_matrix(A)
print(S)
```

- This should return

```
(0, 3)        5
(1, 1)        2
(3, 2)        4
```

- Storing a sparse matrix in sparse format requires only a fraction of the space required to store it in full format.
- If A has only q percent nonzero entries, the space required to store S will be $3q$ percent of the space required to store A.
- For example a 1000×1000 tridiagonal matrix will require 1 million units of storage in full format, but only 8,994 units of storage in sparse format, a savings of 99%.

- The scipy.sparse.linalg function `spsolve` applies Gaussian elimination to exploit the sparseness of sparse matrix.
- In particular, if `S = csr_matrix(A)` is large but sparse, both

  ```
  x = solve(A, b)
  x = spsolve(S,b)
  ```

  will produce the same results, but the latter expression will execute faster by avoiding unnecessary operations with zeros.

Example 3:

Solving a sparse system of equations

Consider the problem of solving $Ax = b$ when A is a 1000×1000 tridiagonal matrix.

```
T = 1000
A = np.eye(T) - 2*np.eye(T,k=1) + 3*np.eye(T,k=-1)
S = csr_matrix(A)
b = A.sum(axis=1)
```

In an interactive session, if you type `%timeit solve(A,b)` you will get (depending on your computer speed)

```
21.1 ms ± 734 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

as compared to `%timeit spsolve(S,b)`

```
513 µs ± 8.25 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

That is, solving the sparse system took 2.43% as long as doing the full array.