Linear Algebra

David Evans

2026-11-13

Introduction

Motivation

Systems of Linear Equations

- One of the most commonly used branches of mathematics in economics
- Many applied problems require solutions to systems of linear equations

$$
\begin{aligned}
y_1 &= ax_1 + bx_2 \\
y_2 &= cx_1 + dx_2
\end{aligned}
$$

- Or, more generally

$$
\begin{aligned}
y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\
&\vdots \\
y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k
\end{aligned}
$$

- Objective: solve for $x_1, \ldots, x_k$ given a_{ij} and $y_1, \ldots, y_n$

Questions We'll Answer

- Does a solution actually exist?
- Are there in fact many solutions, and if so how should we interpret them?
- If no solution exists, is there a "best" approximate solution?
- If a solution exists, how should we compute it?
- What if the equations are **nonlinear**?

> **This Lecture**
>
> We will show how to use Julia to solve these problems numerically — and why solving *nonlinear* systems comes down to solving a sequence of *linear* ones.

Vectors

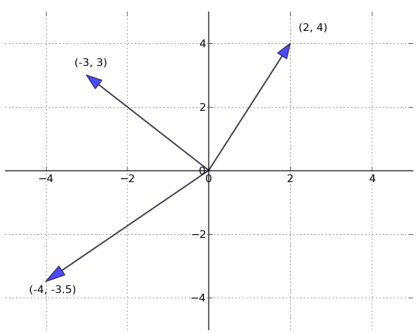
Definition

Vectors

- A vector of length n is just a sequence of n numbers which we write as

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

- The set of all n-vectors is denoted by $\mathbb{R}^n$
- For example, $\mathbb{R}^2$ is the plane, and a vector in $\mathbb{R}^2$ is just a point on the plane



Vectors in Julia

- Julia is built with vectors and matrices as fundamental building blocks

```
1 using LinearAlgebra
```

```
1 x = [1., 2.]
```

```
2-element Vector{Float64}:
 1.0
 2.0
```

- There are multiple ways of constructing arrays:

```
1 x = zeros(3)           # R^3 filled with zeros
2 println("x = $x")
3 y = ones(4)            # R^4 filled with ones
4 println("y = $y")
5 z = collect(LinRange(0, 1, 6))  # R^6 evenly spaced between 0,1
6 println("z = $z")
```

```
x = [0.0, 0.0, 0.0]
y = [1.0, 1.0, 1.0, 1.0]
z = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
```

Editing Vectors

- Once an array is created we can edit elements using `[]`

```
1 x[1] = 1
2 x
```

```
3-element Vector{Float64}:
 1.0
 0.0
 0.0
```

- Using `:` you can access multiple elements at once

```
1 println(x[2:3])
```

```
[0.0, 0.0]
```

Assigning to Slices

- Assigning to slices requires `.=` (dot means "do for each element")

```
1 x[2:3] = 3, 4  # produces error
```

```
ArgumentError: indexed assignment with a single value to possibly many locations is not supported; perhaps use broadcast
Stacktrace:
 [1] setindex_shape_check(::Tuple{Int64, Int64}, ::Int64)
   @ Base ./indices.jl:277
 [2] _unsafe_setindex!(::IndexLinear, A::Vector{Float64}, x::Tuple{Int64, Int64}, I::UnitRange{Int64})
   @ Base ./multidimensional.jl:1045
 [3] _setindex!
   @ ./multidimensional.jl:1036 [inlined]
 [4] setindex!(A::Vector{Float64}, v::Tuple{Int64, Int64}, I::UnitRange{Int64})
   @ Base ./abstractarray.jl:1464
 [5] top-level scope
   @ ~/work/EC410-Source/EC410-Source/Linear Algebra/Linear Algebra Lecture.qmd:140
```

```
1 x[2:3] .= 3, 4  # assigns element-wise
2 x
```

```
3-element Vector{Float64}:
 1.0
 3.0
 4.0
```

Vector Operations

Vector Addition

- When we add two vectors we add them element by element

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

```
1  x = ones(3)
2  y = [2, 4, 6]
3  x + y
```

```
3-element Vector{Float64}:
 3.0
 5.0
 7.0
```

Scalar Multiplication

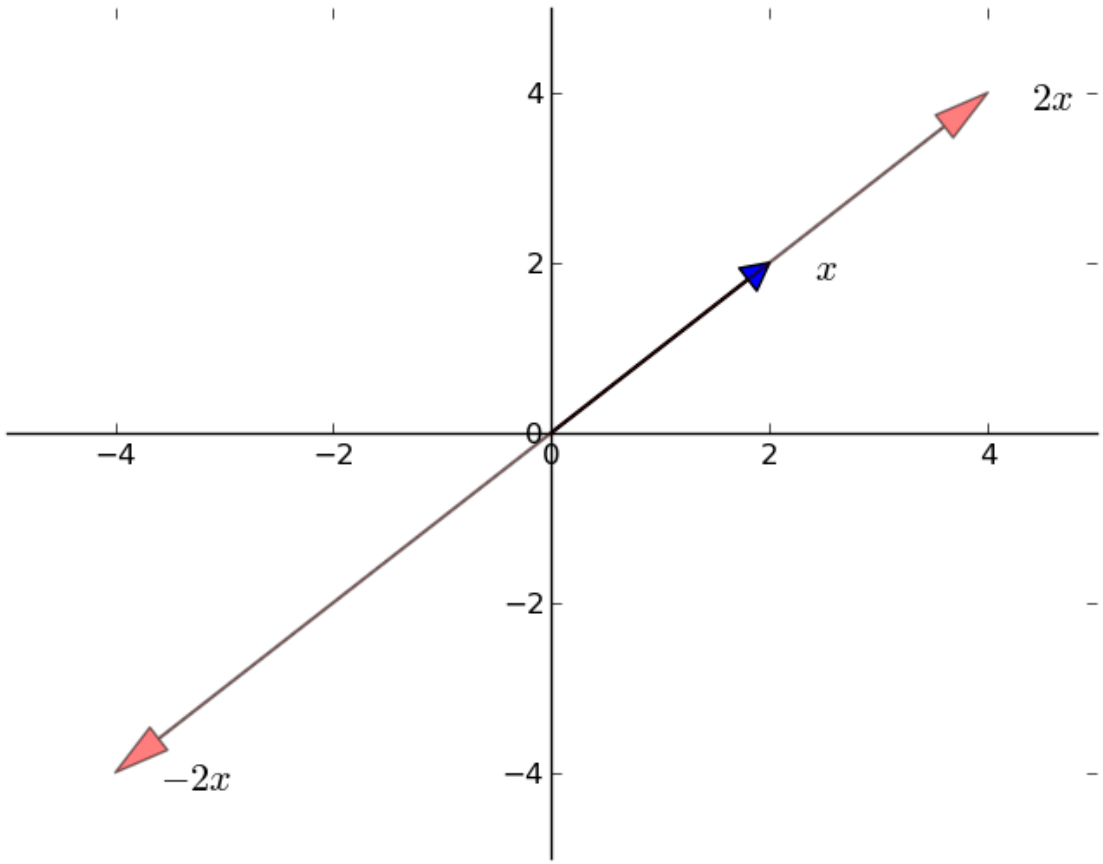
- Scalar multiplication takes a scalar γ and a vector x and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}$$

```
1 4*x
```

```
3-element Vector{Float64}:
 4.0
 4.0
 4.0
```

Scalar Multiplication



Element-wise Multiplication

- Can also multiply x and y element-wise with `.*` notation

```
println(x, y)  # recall what x and y are
xy = x .* y
```

```
[1.0, 1.0, 1.0][2, 4, 6]
```

```
3-element Vector{Float64}:
 2.0
 4.0
 6.0
```

The Dot Operator

In Julia, prefixing an operator with `.` applies it element-wise. This works with any function or operator!

Inner Products and Norms

- The inner product of two vectors $x, y \in \mathbb{R}^n$ is defined as

$$x'y := \sum_{i=1}^{n} x_i y_i$$

- Two vectors are **orthogonal** if their inner product is zero
- The **norm** of a vector x represents its "length":

$$\|x\| := \sqrt{x'x} := \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$$

Inner Products in Julia

```julia
# Two ways of doing inner product
println(sum(x .* y))   # sums over x.*y
println(dot(x, y))     # directly computes dot product
```

```
12.0
12.0
```

```julia
# Two ways of computing the norm
println(sqrt(dot(x, x)))
println(norm(x))
```

```
1.7320508075688772
1.7320508075688772
```

```julia
# Example of orthogonal vectors
z = [-1., 2., -1]
println(dot(x, z))
```

```
0.0
```

Spanning and Linear Independence

Linear Combinations

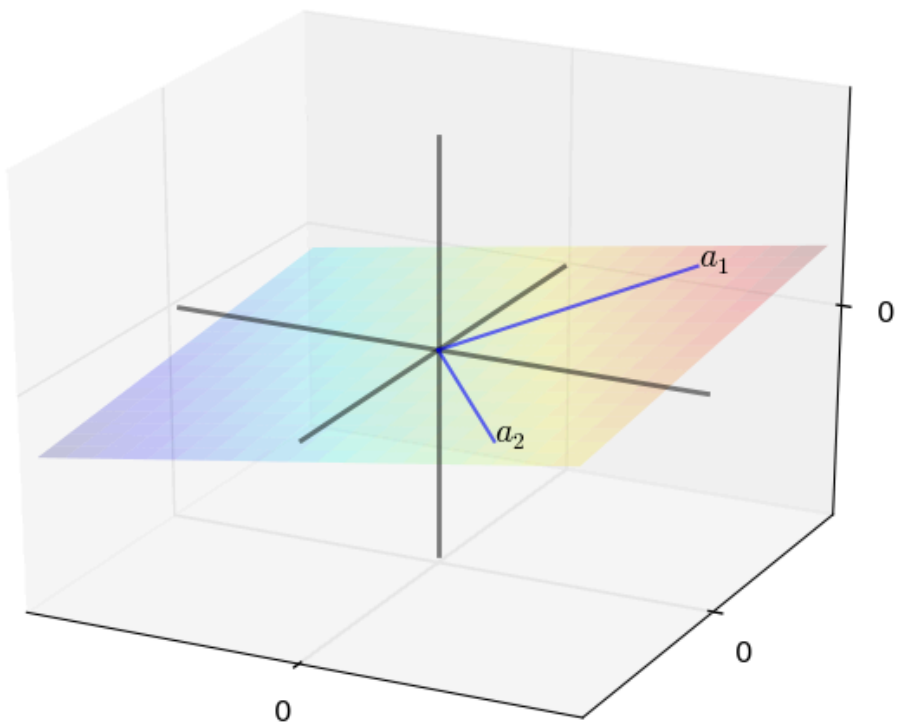
Spanning

- Given vectors $A = \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$, what new vectors can we create using linear operations?
- $y \in \mathbb{R}^n$ is a **linear combination** of A if

$$y = \beta_1 a_1 + \beta_2 a_2 + \ldots + \beta_k a_k \text{ for some scalars } \beta_1, \ldots, \beta_k$$

- The set of linear combinations of A is called the **span** of A

Spanning



Spanning Examples

- If A contains only 1 vector $a_1 \in \mathbb{R}^n$, its span is the line through a_1 and the origin
- The canonical basis vectors $e_1, e_2, \ldots, e_n$ in $\mathbb{R}^n$:

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad e_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots, \quad e_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

- The span of $\{e_1, e_2, \ldots, e_n\}$ is all of $\mathbb{R}^n$ because for any $x \in \mathbb{R}^n$:

$$x = x_1 e_1 + x_2 e_2 + \ldots + x_n e_n$$

Spanning: A Counterexample

- Let $A_0 = \{e_1, e_2, e_1 + e_2\} \subset \mathbb{R}^3$
  - If $y = (y_1, y_2, y_3)$ is a linear combination of these vectors then $y_3 = 0$
  - A_0 fails to span all of $\mathbb{R}^3$

Key Insight

Adding more vectors doesn't always increase the span — the new vector must point in a "genuinely new direction."

Linear Independence

Linear Independence

- $A = \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$ is **linearly dependent** if a strict subset has the same span
- A is **linearly independent** if no such subset exists

- Since $\mathbb{R}^n$ can be spanned by n vectors, any subset with $m > n$ elements must be linearly dependent

- Equivalent definitions:
  1. No vector in A can be formed as a linear combination of the others
  2. If $\beta_1 a_1 + \ldots + \beta_k a_k = 0$ then $\beta_1 = \cdots = \beta_k = 0$

Unique Representations

- Linearly independent vectors give **unique** representations in their span
- If $A = \{a_1, \ldots, a_k\}$ is linearly independent and

$$y = \beta_1 a_1 + \ldots + \beta_k a_k = \gamma_1 a_1 + \ldots + \gamma_k a_k$$

- Subtracting:

$$(\beta_1 - \gamma_1)a_1 + \cdots + (\beta_k - \gamma_k)a_k = 0$$

- By linear independence: $\gamma_i = \beta_i$ for all i

Matrices

Definition

Matrices

- An $n \times k$ matrix is a rectangular array with n rows and k columns

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}$$

- Can also view A as a list of k column vectors in $\mathbb{R}^n$:

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_k \end{bmatrix} \quad \text{where} \quad a_i = \begin{bmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{ni} \end{bmatrix}$$

Matrix Properties

- **Row vector**: $n = 1$; **Column vector**: $k = 1$; **Square**: $n = k$
- The **transpose** A' (or A^T) is formed by swapping rows and columns ($a_{ij} \rightarrow a_{ji}$)

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix} \quad A' = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ \vdots & \vdots & \vdots & \\ a_{1k} & a_{2k} & \cdots & a_{nk} \end{bmatrix}$$

- If A is $n \times k$ then A' is $k \times n$

Transpose in Julia

```
1 A = [1 2 3;
2      4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
1 A'  # transpose is natural in Julia
```

```
3×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6
```

Special Matrices

Symmetric Matrices

- For square matrices, if $A = A'$ then A is called **symmetric**

```
A = [1 2; 2 1]
```

```
2×2 Matrix{Int64}:
 1  2
 2  1
```

```
A'  # same as A!
```

```
2×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  2
 2  1
```

- The **principal diagonal** elements are a_{ii} for $i = 1, \ldots, n$

```
A = [1 2;
     3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
diag(A)  # grabs diagonal of A
```

```
2-element Vector{Int64}:
 1
 4
```

Diagonal Matrices

- A is **diagonal** if the only non-zero entries are on the principal diagonal

```
1 diagm(0 => [1, 2, 3])  # puts 1,2,3 on the diagonal
```

```
3×3 Matrix{Int64}:
 1  0  0
 0  2  0
 0  0  3
```

```
1 diagm(1 => [1, 2, 3])  # shift up by 1
```

```
4×4 Matrix{Int64}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0
```

The Identity Matrix

- The **identity matrix** I has ones on the diagonal:

```
1 diagm(0 => ones(3))
```

```
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

$$I = \begin{bmatrix} e_1 & e_2 & \cdots & e_n \end{bmatrix}$$

Matrix Operations

Addition and Scalar Multiplication

- Both carry over directly from vector definitions

$$\gamma A := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}$$

- For $A + B$ both matrices must be $n \times k$

```
1 A = [1 2; 3 4]
2 B = diagm(0 => ones(2))
3 2*A
```

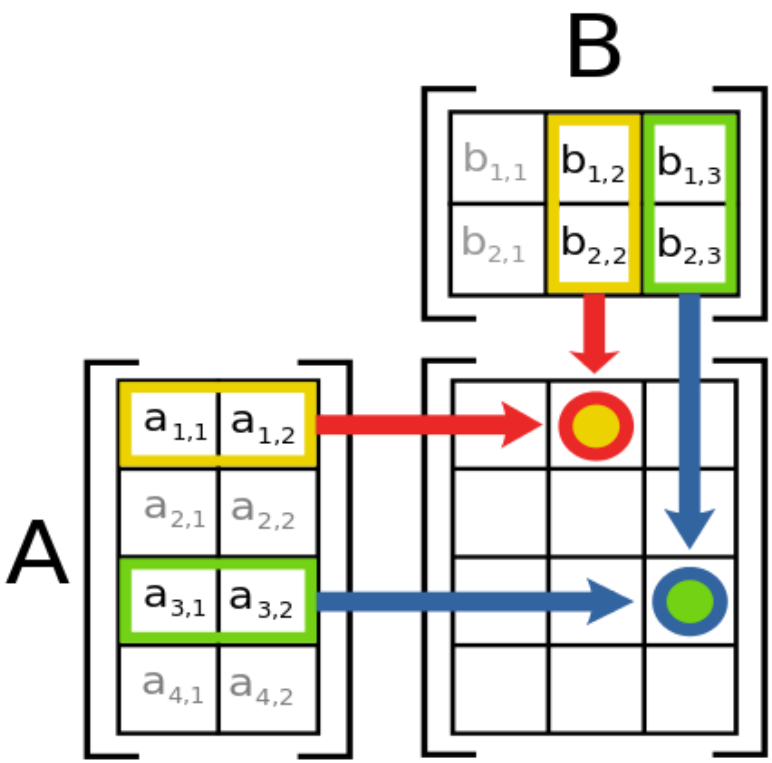
```
2×2 Matrix{Int64}:
 2  4
 6  8
```

```
1 A + B
```

```
2×2 Matrix{Float64}:
 2.0  2.0
 3.0  5.0
```

Matrix Multiplication

- The i, j-th element of AB is the inner product of row i of A and column j of B



- A must be $n \times k$ and B must be $k \times m$
- In general $AB \neq BA$

Matrix-Vector Multiplication

- Multiplying $n \times k$ matrix A with $k \times 1$ vector x gives $n \times 1$ vector:

$$Ax = \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix}$$

Matrix-Vector Multiplication as Linear Combination

- If $a_1, \ldots, a_k$ are the columns of A:

$$Ax = x_1 a_1 + x_2 a_2 + \ldots + x_k a_k$$

- Thus Ax is in the **span** of A
- Identity matrices have the property: $IA = A = AI$

Matrix Multiplication in Julia

```
1  A = [1 2; 3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
1  I2 = diagm(0 => ones(2))
```

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```
1  A * I2   # should just give A
```

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

Element-wise vs. Matrix Multiplication

- Element-wise multiplication via `.*`

```
1 A .* I2
```

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  4.0
```

- Julia has a built-in variable `I` for the identity matrix

```
1 A * I
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

Broadcasting

Broadcasting

- Broadcasting replicates operations across dimensions using `.` notation
- Allows you to, for instance, add a vector to each column of a matrix

```
1  A = diagm(0 => ones(2))
```

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```
1  b = [1, 2]
```

```
2-element Vector{Int64}:
 1
 2
```

```
1  A .+ b   # adds b to each column of A
```

```
2×2 Matrix{Float64}:
 2.0  1.0
 2.0  3.0
```

Broadcasting Rules

- Julia compares array shapes element-wise, starting from trailing dimensions
- Two dimensions are compatible when:
  1. They are equal, or
  2. One of them has length 1

Can these arrays be broadcast together?

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

What about these?

```
A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3
```

No — the second dimensions do not match.

Solving Systems of Equations

Formulation

Systems as Matrix Equations

- Our original system

$$
\begin{aligned}
y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\
&\vdots \\
y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k
\end{aligned}
$$

- can be written as $y = Ax$
- Since $Ax = x_1a_1 + \ldots + x_ka_k$, the solution requires y to be in the span of A
- If the columns of A are linearly independent and span $\mathbb{R}^n$, a **unique** solution exists
- We say A has **full column rank**

Regular Case

The Regular Case

- When A is $n \times n$ with linearly independent columns:
  - $y = Ax$ has a solution for every y, and this solution is unique
- Define $g(y)$ as the solution: $y = Ag(y)$
- $g(y)$ is a linear function:

$$g(\alpha w + \beta z) = \alpha g(w) + \beta g(z)$$

- Any linear function can be represented by a matrix, so there exists A^{-1} such that $g(y) = A^{-1}y$

The Inverse Matrix

- If A is $n \times n$ with full column rank, there exists A^{-1} such that

$$I = A^{-1}A = AA^{-1}$$

- Pre-multiplying $y = Ax$ by A^{-1}:

$$A^{-1}y = A^{-1}Ax = x$$

- **Determinant test**: if $\det(A) \neq 0$ then A is invertible
- If $\det(A) = 0$ then the columns are linearly dependent and no inverse exists

Key Result

$x = A^{-1}y$ is the solution to $y = Ax$ when A is square and invertible.

Solving in Julia: The Inverse

```
1  A = [1 2; 3 4]
2  y = ones(2)
3  det(A)   # check det(A) ≠ 0
```

```
-2.0
```

```
1  A_inv = inv(A)
```

```
2×2 Matrix{Float64}:
 -2.0   1.0
  1.5  -0.5
```

```
1  x = A_inv * y
```

```
2-element Vector{Float64}:
 -0.9999999999999998
  0.9999999999999999
```

```
1  A * x   # verify: should give y
```

```
2-element Vector{Float64}:
 1.0
 1.0000000000000004
```

Solving in Julia: The Backslash Operator

- Can also use `A\y` (backslash) to solve directly

```
1  x = A \ y
```

```
2-element Vector{Float64}:
 -1.0
  1.0
```

> **Best Practice**
>
> Prefer `A\y` over `inv(A)*y` — it is more numerically stable and efficient.

Over-determined Systems

Over-determined Systems

- When A is $n \times k$ with $n > k$ (more equations than unknowns)
- This case arises frequently in **least-squares regression**
- The span of A is unlikely to contain an arbitrary $y \in \mathbb{R}^n$

- Instead, find x that minimizes $\|y - Ax\|$
- The solution is $\hat{x} = (A'A)^{-1}A'y$ when A has full column rank
- Julia provides the pseudo-inverse via `pinv`

Least-Squares Example

Fit a line $y = mx + c$ through noisy data:

```
1 x = [0, 1, 2, 3]
2 y = [-1, 0.2, 0.9, 2.1]
```

```
4-element Vector{Float64}:
 -1.0
  0.2
  0.9
  2.1
```

Rewrite as $y = Ap$ where $A = \begin{bmatrix} x & 1 \end{bmatrix}$ and $p = \begin{bmatrix} m & c \end{bmatrix}$:

```
1 A = [x ones(length(x))]
```

```
4×2 Matrix{Float64}:
 0.0  1.0
 1.0  1.0
 2.0  1.0
 3.0  1.0
```

Least-Squares Solution

```
1 m, c = pinv(A) * y
2 print(m, c)
```

```
0.9999999999999998-0.9499999999999997
```

Matches our formula $(A'A)^{-1}A'y$:

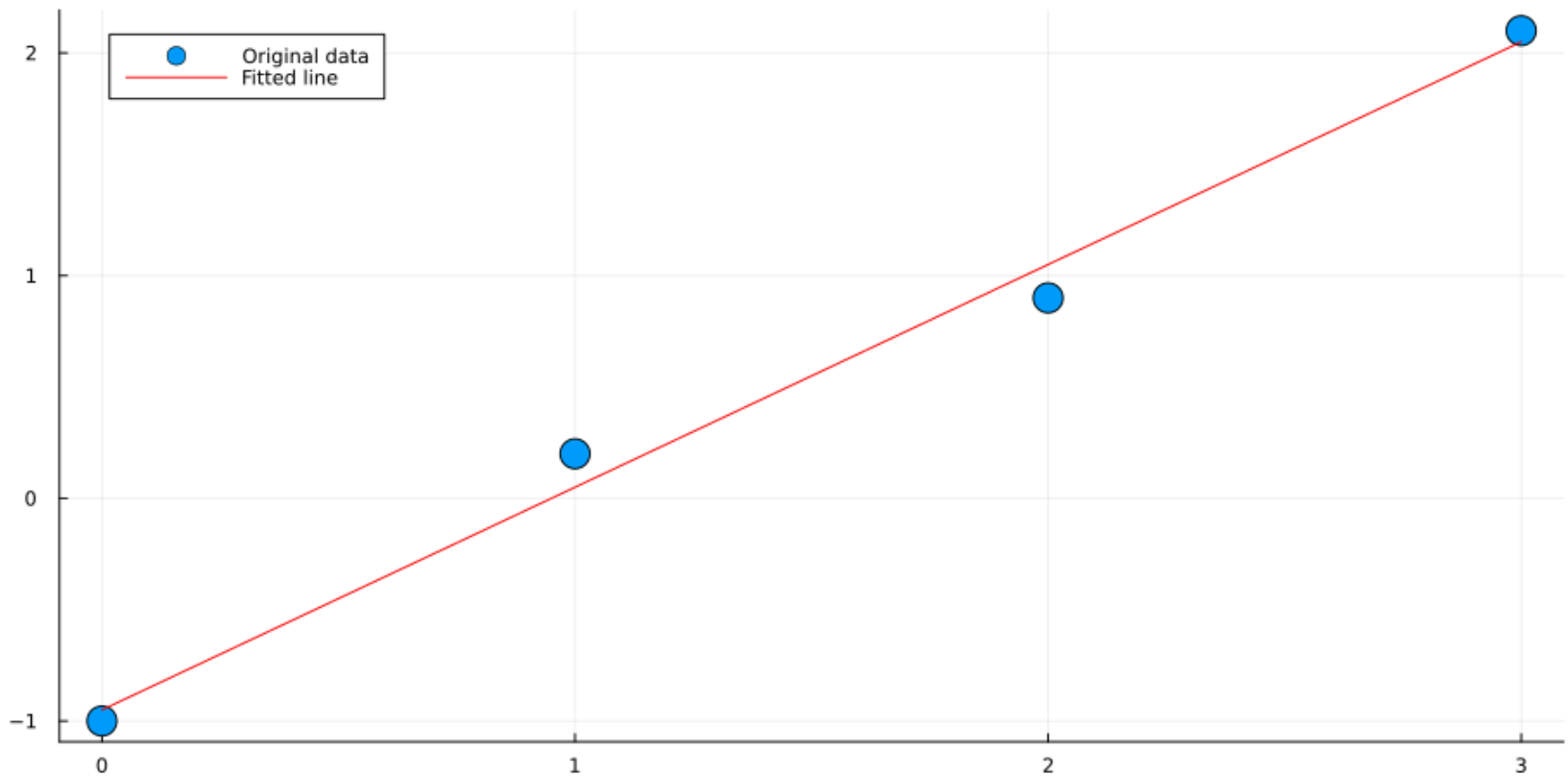
```
1 (A' * A) \ A' * y
```

```
2-element Vector{Float64}:
  0.9999999999999998
 -0.9499999999999997
```

Plotting the Fit

```
1 using Plots
```

```
1 scatter(x, y, label="Original data", markersize=10)
2 plot!(x, m*x .+ c, color=:red, label="Fitted line")
```



Under-determined Systems

Under-determined Systems

- If A is $n \times k$ with $n < k$: fewer equations than unknowns
- Expect either no solutions or **infinitely many** solutions

- The **kernel** $\ker(A)$ is the set of all x such that $Ax = 0$
- If x solves $y = Ax$ and $\hat{x} \in \ker(A)$, then $x + \hat{x}$ is also a solution:

$$A(x + \hat{x}) = Ax + A\hat{x} = Ax + 0 = y$$

> ⚠ **Caution**
>
> Under-determined systems have infinitely many solutions. Methods like `pinv` return the solution with smallest norm $\|x\|$.

Application: Leontief Input-Output Model

Setup

Input-Output Analysis

- Wassily Leontief won the 1973 Nobel Prize for **input-output analysis**
- Key idea: industries are interdependent
  - to produce steel you need coal
  - to produce cars you need steel
  - the coal industry needs steel for mining equipment
- Consider an economy with n industries
- Let a_{ij} = amount of good i needed to produce **one unit** of good j
- These form the $n \times n$ **technology matrix** A

The Balance Equation

- x: gross output (total production by each industry)
- d: final demand (households, government, exports)
- Total output must cover intermediate demand (Ax) and final demand (d):

$$x = Ax + d$$

- Rearranging to solve for gross output:

$$x - Ax = d \quad \Longrightarrow \quad (I - A)x = d \quad \Longrightarrow \quad x = (I - A)^{-1}d$$

- $(I - A)^{-1}$ is the **Leontief inverse**

The Leontief Inverse

- Element (i, j) of $(I - A)^{-1}$
  - **total** amount of good i needed to deliver one extra unit of good j to consumers

- Captures the **cascade of indirect effects**:

$$(I - A)^{-1} = I + A + A^2 + A^3 + \cdots$$

Term	Meaning
I	Direct output (the good itself)
A	First-round inputs
A^2	Inputs to produce the inputs
A^k	k-th round of indirect effects

When Does It Exist?

- Need $(I - A)^{-1}$ to exist **and** have non-negative entries
  - otherwise some positive final demand would require negative output
- Sufficient condition: every **column of A sums to less than 1**
  - each industry uses less than \$1 of inputs to produce \$1 of output
- Then each round of indirect effects is smaller than the last: $A^k \to 0$ and $I + A + A^2 + \cdots$ converges

Productive Economy

When the column sums are below 1 the economy is **productive**: the Leontief inverse is a sum of non-negative matrices, so it is non-negative. If industries needed more than \$1 of inputs per \$1 of output, the economy would consume more than it produces.

Three-Sector Example

Setting Up the Technology Matrix

Three sectors: **Agriculture**, **Manufacturing**, **Services**

```
# a_ij = amount of good i needed per unit of good j
# Columns: Agriculture, Manufacturing, Services
A_io = [0.10  0.20  0.05;    # Agriculture inputs
        0.15  0.10  0.10;    # Manufacturing inputs
        0.05  0.15  0.10]    # Services inputs
```

```
3×3 Matrix{Float64}:
 0.1   0.2   0.05
 0.15  0.1   0.1
 0.05  0.15  0.1
```

- Column 1: producing 1 unit of agriculture needs 0.10 agriculture (seeds), 0.15 manufacturing (tractors), 0.05 services (transport)

Checking Productivity

```
  1  sum(A_io, dims=1)   # column sums: inputs per unit of output
```

```
1×3 Matrix{Float64}:
 0.3  0.45  0.25
```

- Every column sums to less than 1 $\Longrightarrow$ the economy is productive

```
  1  leontief_inv = inv(I - A_io)
  2  round.(leontief_inv, digits=3)
```

```
3×3 Matrix{Float64}:
 1.162  0.274  0.095
 0.205  1.18   0.143
 0.099  0.212  1.14
```

- Every entry is positive: final demand for **any** good requires extra output of **every** good

Computing Gross Output

Suppose final demand is $d = (100, 200, 150)'$:

```
d = [100.0, 200.0, 150.0]
x = (I - A_io) \ d           # solve (I - A)x = d
println("Final demand:  $d")
println("Gross output:  $(round.(x, digits=2))")
println("Intermediate:  $(round.(x .- d, digits=2))")
```

```
Final demand:  [100.0, 200.0, 150.0]
Gross output:  [185.27, 277.91, 223.28]
Intermediate:  [85.27, 77.91, 73.28]
```

- Gross output exceeds final demand for **every** sector
- The difference $x - d$ is intermediate demand — output used as inputs by other industries

The Multiplier Effect

What if manufacturing demand increases by 50 units?

```
d_new = [100.0, 250.0, 150.0]
x_new = leontief_inv * d_new
println("Change in gross output: $(round.(x_new .- x, digits=2))")
```

```
Change in gross output: [13.7, 59.02, 10.6]
```

- The change equals the **second column** of the Leontief inverse, scaled by 50
- **Every** sector must increase production, not just manufacturing

> **Multiplier Effect**
>
> A demand shock in one sector ripples through the entire economy via input-output linkages.

Nonlinear Systems of Equations

From Linear to Nonlinear

Beyond Linear Systems

- Linear systems $Ax = y$: check $\det(A) \neq 0$, then solve in one line with `A \ y`
- But most equilibrium conditions in economics are **nonlinear**
  - CES demand, Cobb-Douglas production, first-order conditions with diminishing returns
- General problem: find $x \in \mathbb{R}^n$ such that

$$f(x) = 0, \qquad f : \mathbb{R}^n \to \mathbb{R}^n$$

- n equations in n unknowns — the linear case is $f(x) = Ax - y$
- No formula like $x = A^{-1}y$, so we need an **iterative** algorithm

Big Idea

Replace f with a linear approximation, solve that **linear system**, and repeat. Linear algebra is the engine inside nonlinear solvers.

Example: Two-Market Equilibrium

Two goods that are substitutes, with constant-elasticity demand and supply:

$$D_1 = 100\, p_1^{-2} p_2^{0.5}, \quad S_1 = 20\, p_1^{1.5}, \qquad D_2 = 80\, p_2^{-1.5} p_1^{0.3}, \quad S_2 = 15\, p_2^{2}$$

- Equilibrium: excess demand $f(p) = D(p) - S(p)$ is zero in **both** markets

```julia
function excess_demand(p)
    p₁, p₂ = p
    return [100p₁^-2 * p₂^0.5 - 20p₁^1.5,    # market 1: D₁ - S₁
            80p₂^-1.5 * p₁^0.3 - 15p₂^2]     # market 2: D₂ - S₂
end
excess_demand([2.0, 2.0])
```

```
2-element Vector{Float64}:
 -21.213203435596427
 -25.177977468155035
```

- At $p = (2, 2)$ both markets have excess supply: prices are too high
- Prices enter through powers and products — we can't write this as $Ap = y$

Newton's Method

Linearize, Then Solve

- **One equation**: follow the tangent line at the guess x_n to where it hits zero, $x_{n+1} = x_n - f(x_n)/f'(x_n)$
- n **equations**: the tangent line becomes a first-order Taylor approximation

$$f(x) \approx f(x_n) + J(x_n)\,(x - x_n)$$

- $J(x_n)$ is the **Jacobian**: the $n \times n$ matrix of partial derivatives $J_{ij} = \partial f_i / \partial x_j$
- Set the approximation to zero and solve for the step $\Delta = x_{n+1} - x_n$:

$$J(x_n)\,\Delta = -f(x_n), \qquad x_{n+1} = x_n + \Delta$$

Each Newton Step Is a Linear System

$J(x_n)\Delta = -f(x_n)$ is $Ax = y$ with $A = J(x_n)$ and $y = -f(x_n)$. In Julia: `Δ = J(x) \ (-f(x))`

The Jacobian

Differentiate each market's excess demand with respect to each price:

$$J(p) = \begin{bmatrix} -200\,p_1^{-3}p_2^{0.5} - 30\,p_1^{0.5} & 50\,p_1^{-2}p_2^{-0.5} \\ 24\,p_1^{-0.7}p_2^{-1.5} & -120\,p_1^{0.3}p_2^{-2.5} - 30\,p_2 \end{bmatrix}$$

```
function jacobian(p)
    p₁, p₂ = p
    J₁₁ = -200p₁^-3 * p₂^0.5 - 30p₁^0.5     # ∂f₁/∂p₁ (own price)
    J₁₂ = 50p₁^-2 * p₂^-0.5                 # ∂f₁/∂p₂ (cross price)
    J₂₁ = 24p₁^-0.7 * p₂^-1.5               # ∂f₂/∂p₁ (cross price)
    J₂₂ = -120p₁^0.3 * p₂^-2.5 - 30p₂       # ∂f₂/∂p₂ (own price)
    return [J₁₁ J₁₂;
            J₂₁ J₂₂]
end
```

```
jacobian (generic function with 1 method)
```

- Own-price effects (diagonal) are negative; cross-price effects are positive because the goods are substitutes

One Newton Step

```
p = [2.0, 2.0]                              # initial guess
Δ = jacobian(p) \ (-excess_demand(p))       # solve J(p) Δ = -f(p)
p_new = p + Δ                               # take the step
```

```
2-element Vector{Float64}:
 1.6919253416011006
 1.6889430060896855
```

```
println("‖f(p)‖     = $(norm(excess_demand(p)))")
println("‖f(p_new)‖ = $(norm(excess_demand(p_new)))")
```

```
‖f(p)‖     = 32.92310054334073
‖f(p_new)‖ = 1.3883008911248673
```

- One linear solve shrinks excess demand from about 33 to about 1.4
- Now repeat from p_new, and again, until the step is tiny

`newton`: Pseudocode

```
function newton(f, J, x₀; tol = 1e-10, maxiter = 50)
    # Start at the initial guess x₀
    # For n = 1, 2, ..., maxiter:
    #   solve J(x) Δ = -f(x) for the step Δ
    #   update x = x + Δ
    #   stop once the step Δ is tiny
end
```

`newton`: Full Function

```
function newton(f, J, x₀; tol = 1e-10, maxiter = 50)
    x = x₀
    for n in 1:maxiter
        Δ = J(x) \ (-f(x))              # linearize, then solve
        x = x + Δ                       # take the step
        println("iteration $n: ‖f(x)‖ = $(norm(f(x)))")
        if norm(Δ) < tol                # converged: steps are tiny
            return x
        end
    end
    println("Warning: no convergence after $maxiter iterations")
    return x
end
```

```
newton (generic function with 1 method)
```

- `f` and `J` are passed in as arguments — functions are values in Julia, so `newton` works for any system

Watching It Converge

```
1  p_star = newton(excess_demand, jacobian, [2.0, 2.0])
```

```
iteration 1: ‖f(x)‖ = 1.3883008911248673
iteration 2: ‖f(x)‖ = 0.009203787568081082
iteration 3: ‖f(x)‖ = 4.049338246528834e-7
iteration 4: ‖f(x)‖ = 0.0
iteration 5: ‖f(x)‖ = 0.0
```

```
2-element Vector{Float64}:
 1.7069573735223966
 1.6889575066262879
```

- Near the root the error is roughly **squared** each iteration:
  $1.4 \to 0.009 \to 4 \times 10^{-7} \to 0$
  - the number of correct digits roughly doubles (**quadratic convergence**)

> **Linear Systems Take One Step**
>
> If $f(x) = Ax - y$ then $J = A$ everywhere, and the first Newton step lands exactly on $A^{-1}y$. Newton's method generalizes `A \ y`.

Solving in Practice

The Easy Way: `solve`

- In practice, use a well-tested solver: the course package `EC410` provides `solve(f, x0)`

```
using EC410
p_solve = solve(excess_demand, [2.0, 2.0])
```

```
2-element Vector{Float64}:
 1.7069573735223966
 1.6889575066262879
```

```
p_solve ≈ p_star     # same answer as our hand-written Newton loop
```

```
true
```

- No Jacobian needed — the solver computes the derivatives for you
- Under the hood: Newton-type steps (each one a linear solve) plus safeguards for when a full step would overshoot
- One equation? Use `solve1D(f, a, b)` (with a bracket) or `solve1D(f, x0)`

Initial Guesses and Multiple Solutions

- Nonlinear systems can have **several** solutions — which one you find depends on the initial guess

```
g(x) = [x[1]^2 + x[2]^2 - 4,      # circle of radius 2
        x[2] - x[1]^2 + 1]        # parabola
println(solve(g, [1.0, 1.0]))
println(solve(g, [-1.0, 1.0]))
```

```
[1.5174899135519797, 1.3027756377319946]
[-1.5174899135519797, 1.3027756377319946]
```

- Newton also needs $J(x_n)$ to be **invertible**: here $\det J(x) = 0$ whenever $x_1 = 0$, so from the guess $(0, 1)$ the first step is undefined
- Far from a root the linear approximation can be poor, and steps can overshoot

> ⚠ **Always Check**
>
> Verify that $f(x^*) \approx 0$, read any solver warnings, try several initial guesses, and make sure the answer makes economic sense (e.g. positive prices).

Summary

Key Takeaways

Concepts

- Vectors and vector spaces
- Spanning and linear independence
- Matrix operations and multiplication
- Inverse matrices and determinants
- Leontief input-output model
- Nonlinear systems and Newton's method

Julia Tools

- `LinearAlgebra` package
- `dot`, `norm`, `det`
- `inv(A)`, `A\y`
- `pinv(A)` for least-squares
- `J \ (-f)` for a Newton step
- `solve(f, x0)` from `EC410`
- `.*`, `.+` for broadcasting