Intro to Linear Systems

Outline for today

- Linear systems: what they are + a few examples
- Gauss-Jordan elimination (as a general-purpose method)
- What can go wrong with roundoff
- A simple fix: (partial) pivoting
- Ill-conditioning: sometimes the *problem* is the issue
- Polynomial interpolation as an example

Linear systems

Application number one: solving linear systems

- The first big thing we will use linear algebra for is solving *systems of linear equations*.
- Hope to convince you that this is **useful** but not entirely **straightforward**!

Linear equations

We all know what "linear" means: straight!

. . .

- A linear equation of one variable is of the form $ax + b = 0$.
  - Solution: A single point on the number line: $x = -\frac{b}{a}$

- A linear equation of two variables is of the form $ax + by + c = 0$.
  - Solution: any pair of values (x, y) that satisfies the equation. These form a straight line in the plane.
- A linear equation of three variables is of the form $ax + by + cz + d = 0$.
  - Solution: any triple of values (x, y, z) that satisfies the equation. These form a plane in three-dimensional space.
- In general, a linear equation of n variables is of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n + b = 0$.

Linear systems

- Linear system: a collection of linear equations involving the same variables $x_1, x_2, \ldots, x_n$.
- Each equation has its own set of coefficients a_{ij} and its own right-hand side b_i. So the 3rd equation might look like $a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n = b_3$.

. . .

When we put them all together, we get a linear system:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m, \end{cases}$$

Why do we care about systems of equations?

We use equations to represent constraints (physics, data, budgets, flows, etc.) When there are multiple constraints, we get a system of equations.

. . .

Examples:

. . .

Of course, many equations we can think of won't be linear… But many things are!

- Once you write a model down, it often becomes (Ax=b)
- Then the question is: can we solve it **reliably**?

Examples of linear systems (we aren't going to solve them yet!)

Example 1: railroad cars

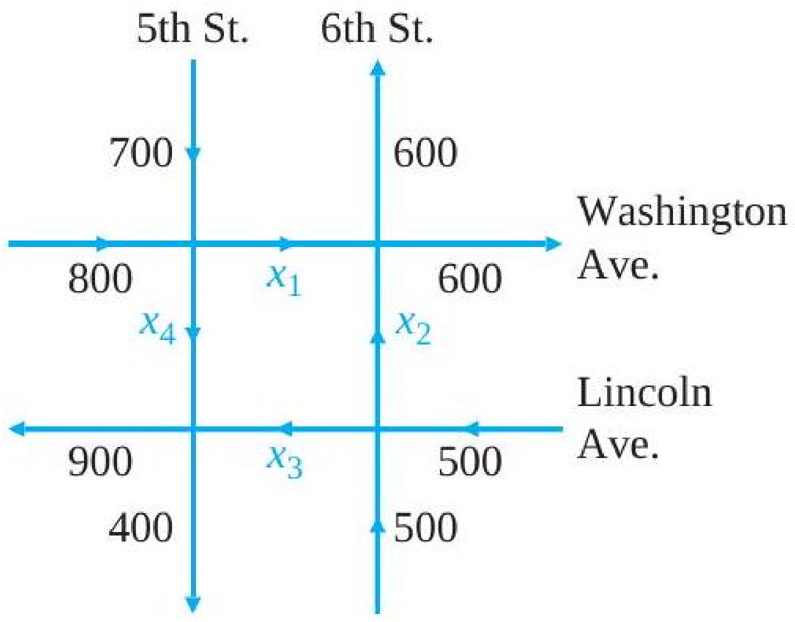
A chemical manufacturer wants to lease a fleet of 24 railroad tank cars with a combined carrying capacity of 520,000 gallons.

- Tank cars with three different carrying capacities are available:
  - 8,000 gallons
  - 16,000 gallons
  - 24,000 gallons.
- How many of each type of tank car should be leased?

. . .

Will we have a single solution?

Example 2: traffic flow.



- Numbers = # of vehicles/hr that enter and leave on that street.
- x_1, x_2, x_3, and x_4: flow of traffic between the four intersections.

- Number of vehicles entering each intersection should always equal the number leaving. E.g.:
  - 1,500 vehicles enter the intersection of 5th Street and Washington Avenue each hour
  - $x_1 + x_4$ vehicles leave this intersection
  - $\rightarrow x_1 + x_4 = 1,500$

- Find the traffic flow at each of the other three intersections.
- What is the # of vehicles that travel from Washington Avenue to Lincoln Avenue on 5th Street?

Example: US population

- The U.S. population was approximately 75 million in 1900, 150 million in 1950, and 275 million in 2000.
- Find a quadratic equation whose graph passes through the points $(0, 75)$, $(50, 150)$, $(100, 275)$

. . .

Wait, what? How is this a **linear** system?

. . .

There are three years that our graph needs to pass through: 1900, 1950, and 2000. We can write these as $t_1 = 0$, $t_2 = 50$, and $t_3 = 100$.

We can write the population as $y_1 = 75$, $y_2 = 150$, and $y_3 = 275$.

. . .

We are asked to find a *quadratic equation*, which means it's of the form $y = ax^2 + bx + c$. We need to find the values of a, b, and c so that the equation passes through our three points.

For the first point, we have $at_1^2 + bt_1 + c = y_1$. Since t_1=0, this becomes $a0^2 + b0 + c = 75$.

If we do this for the other two points, we get the system:

$$\begin{cases} at_1^2 + bt_1 + c = y_1 \\ at_2^2 + bt_2 + c = y_2 \\ at_3^2 + bt_3 + c = y_3 \end{cases}$$

. . .

Is this linear?

. . .

Yes, it is linear in a, b, and c, because none of them is raised to a power other than 1.

The terms that are squared are just coefficients. We can even calculate them: $t_2^2 = 2500$, for instance. Then we can rewrite the system as:

$$\begin{cases} 0a + b0 + c = 75 \\ 2500a + 50b + c = 150 \\ 10000a + 100b + c = 275 \end{cases}$$

. . .

We see that we now have a standard system of linear equations.

Solving linear systems – bring in the linear algebra!

Why not just solve by hand?

- We want procedures that work for many variables, not just (2×2) or (3×3)
- We care about accuracy, but also stability: do small errors get amplified?
- We care about efficiency and reusability (especially when solving (Ax=b) for lots of different (b)'s)

Goals for algorithms

An algorithm should be:

- feasible
- accurate
    - stable
- efficient
    - reusable computations

Example: very simple linear system

We'll get started with an example that is easy enough to solve by hand. We'll pay attention to what we do, and why.

$$\begin{aligned} 2x - y &= 1 \\ 4x + 4y &= 20 \end{aligned}$$

Augmented matrix

It was a little hard to keep track of the variables just there.

(Not really! It shouldn't have been! But it's late when I'm writing this, and my brain doesn't work so well, I need something easier…)

. . .

Make an *augmented matrix* to represent the problem.

Rules for an augmented matrix: - The numbers in the first column are the coefficients of the first variable, for each equation. - The numbers in the second column are the coefficients of the second variable, for each equation. And so on. - The numbers in the last column are the right-hand sides of the equations.

Remember, our two equations were:

$$\begin{aligned} 2x - y &= 1 \\ 4x + 4y &= 20 \end{aligned}$$

. . .

So as an augmented matrix, they become:

$$\begin{array}{ccc} x & y & = r.h.s. \\ \end{array}$$
$$\begin{bmatrix} 2 & -1 & 1 \\ 4 & 4 & 20 \end{bmatrix}$$

. . .

Our goal: manipulate the augmented matrix to solve the problem.

What things can we do while we are solving the problem?

We are going to change around the numbers in our matrix *without changing the solution.*

. . .

What does that mean? It means that we will manipulate the matrix so it represents *different* systems of equations that have the **same solution**.

. . .

For example, these two systems have the same solution:

$$\begin{aligned} 2x - y &= 1 \\ 4x + 4y &= 20 \end{aligned}$$

$$\begin{aligned} 4x + 4y &= 20 \\ 2x - y &= 1 \end{aligned}$$

. . .

We have simply swapped the order of the equations. They still have the same solution.

How does this change the matrix? It swaps the first and second rows.

$$\begin{array}{ccc} x & y = & r.h.s. \\ \end{array}$$
$$\begin{bmatrix} 4 & 4 & 20 \\ 2 & -1 & 1 \end{bmatrix}$$

. . .

Swapping the rows of the matrix is equivalent to swapping the order of the equations, and **it does not change the solution**.

Multiplying a row by a constant

We can also multiply one of our equations by a constant. For example, if we multiply the second equation by 2, we get:

$$\begin{aligned} 2x - y &= 1 \\ 8x + 8y &= 40 \end{aligned}$$

. . .

This has the same solution as the original system.

. . .

How does this change the matrix? It multiplies the second row by 2.

$$\begin{array}{ccc} x & y = & r.h.s. \\ \end{array}$$
$$\begin{bmatrix} 2 & -1 & 1 \\ 8 & 8 & 40 \end{bmatrix}$$

Adding a multiple of one row to another

We did this when we were solving the problem by hand. We subtracted two times the first row from the second row.

How does this change the matrix? It adds -2 times the first row to the second row.

$$\begin{array}{ccc} x & y = & r.h.s. \\ \end{array}$$
$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 6 & 18 \end{bmatrix}$$

Elementary Matrix Operations

When we do these on augmented matrices, the solutions are unchanged…

1. E_{ij} : **Swap**: Switch the ith and j th rows of the matrix.
2. $E_i(c)$: **Scale**: Multiply the ith row by the nonzero constant c.
3. $E_{ij}(d)$: **Add**: Add d times the jth row to the ith row.

Our strategy

We are going to use a sequence of elementary row operations to transform our augmented matrix into a form where we will be able to read off the solution.

. . .

This form is called **reduced row echelon form**.

Reduced row echelon form

Here is an example of an augmented matrix in reduced row echelon form:

$$\begin{array}{ccc} x & y = & r.h.s. \\ \end{array}$$
$$\begin{bmatrix} 1 & 4 & 2 \\ 0 & 1 & 3 \end{bmatrix}$$

. . .

It corresponds to the following system of equations:

$$\begin{aligned} x + 4y &= 2 \\ y &= 3 \end{aligned}$$

. . .

See how easy that is to solve? We can just read from the bottom up: $y = 3$, and then $x + 4(3) = 2$, so $x = 2 - 12 = -10$.

. . .

Any time we can get an augmented matrix into reduced row echelon form, it will be this easy to solve.

Rules for Reduced Row Echelon Form

Every matrix can be reduced by a sequence of elementary row operations to one and only one reduced row echelon form:

- Nonzero rows of R precede the zero rows.
- Column numbers of the leading entries of the nonzero rows, say rows $1, 2, \ldots, r$, form an increasing sequence of numbers $c_1 < c_2 < \cdots < c_r$.
- Each leading entry is a 1.
- Each leading entry has only zeros above it.

. . .

We just need to know which row operations to do, and in what order. Then we can solve any linear system!

An algorithm for getting an augmented matrix into reduced row echelon form

Gauss-Jordan elimination

1. Swap rows to get non-zero number in row 1, column 1
2. Get a **1** in the row 1, column 1.
3. Make all other entries in column 1 **0**.

4. Swap rows to get non-zero number in row 2, column 2. Make this entry **1**. Make all other entries in column 2 **0**.
5. Repeat step 4 for row 3, column 3. Continue moving along the main diagonal until you reach the last row, or until the number is zero.

. . .

Vocabulary: process of getting a 1 in a location, and then making all other entries zeros in that column, is **pivoting**.

The number that is made a 1 is called the pivot element, and the row that contains the pivot element is called the **pivot row**.

G-J Elimination for our toy system

$$\begin{bmatrix} 4 & 4 & 20 \\ 2 & -1 & 1 \end{bmatrix} \xrightarrow{E_1(\frac{1}{4}} \begin{bmatrix} 1 & 1 & 5 \\ 2 & -1 & 1 \end{bmatrix} \xrightarrow{E_{12}(-2)} \begin{bmatrix} 1 & 1 & 5 \\ 0 & -3 & -9 \end{bmatrix}$$

and then…

. . .

$$\begin{bmatrix} 1 & 1 & 5 \\ 0 & -3 & -9 \end{bmatrix} \xrightarrow{E_2(-1/3)} \begin{bmatrix} 1 & 1 & 5 \\ 0 & 1 & 3 \end{bmatrix} \xrightarrow{E_{12}(-1)} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \end{bmatrix}.$$

. . .

Writing this back as a linear system, we have:

$$\begin{aligned} 2x - y &= 1 \\ 4x + 4y &= 20 \end{aligned} \Longrightarrow \begin{aligned} 1 \cdot x + 0 \cdot y &= 2 \\ 0 \cdot x + 1 \cdot y &= 3 \end{aligned} \Longrightarrow \begin{aligned} x &= 2 \\ y &= 3 \end{aligned}$$

Now you try: birds in a tree

There are 2 trees in a garden (tree “A” and “B”) and in both trees are some birds.

The birds of tree A say to the birds of tree B that if one of you comes to our tree, then our population will be double yours.

Then the birds of tree B tell the birds of tree A that if one of you comes here, then our population will be equal to yours.

How many birds in each tree?

(Solve by making an augmented matrix and doing G-J elimination.)

https://www.mathsisfun.com/puzzles/birds-in-trees.html

Example: Mining

Meyer Ch 1.5

A mine produces *silica*, *iron*, and *gold*

Needs **Money** (in $$), **operating time** (in hours), and **labor** (in person-hours).

- 1 pound of silica needs: $.0055, . 0011 hours of operating time, and .0093 hours of labor.
- 1 pound of iron needs: $.095, . 01 operating hours, and .025 labor hours.
- 1 pound of gold needs: $ 960, 112 operating hours, and 560 labor hours.

Mining example: from story ($\rightarrow$) linear system

Suppose that during 600 hours of operation, exactly $ 5000 and 3000 labor-hours are used.

How much silica (x), iron (y), and gold (z) were produced?

. . .

Set up the linear system whose solution will yield the values for x, y, and z.

. . .

$$\begin{aligned}
.0055x + .095y + 960z &= 5000 \quad \text{(dollars)}\\
.0011x + .01y + 112z &= 600 \quad \text{(operating hours)}\\
.0093x + .025y + 560z &= 3000 \quad \text{(labor hours)}
\end{aligned}$$

. . .

Make the *augmented matrix*:

$$\left[\begin{array}{ccc|c} .0055 & .095 & 960 & 5000\\ .0011 & .01 & 112 & 600\\ .0093 & .025 & 560 & 3000 \end{array}\right]$$

We can solve this using Gauss-Jordan elimination.

```
from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
from sympy.core.numbers import Number
from sympy import print_latex, latex,N
def round_expr(expr, num_digits):
    return expr.xreplace({n : round(n, num_digits) for n in expr.atoms(Number)})
```

```
# Do the Gauss-Jordan elimination
M=Matrix([[.0055,.095,960,5000],[.0011,.01,112,600],[.0093,.025,560,3000]])
round_level = 2;
M2=N(M.elementary_row_op('n->n+km', row=1,row2=0, k=-(M[1,0]/M[0,0])),round_level)
M3=N(M2.elementary_row_op('n->n+km', row=2,row2=0, k=-(M2[2,0]/M2[0,0])),round_level)
M4=N(M3.elementary_row_op('n->n+km', row=2,row2=1, k=-(M3[2,1]/M3[1,1])),round_level)
```

```
from IPython.display import Markdown
def pp(x):
  print("$"+latex(x)+"$")
def ppd(x):
  print("$$"+latex(x)+"$$")
def mm(x):
  Markdown(latex(x))
```

Getting into row echelon form, rounding after 3 digits

$$\begin{bmatrix} 0.0055 & 0.095 & 960 & 5000 \\ 0.0011 & 0.01 & 112 & 600 \\ 0.0093 & 0.025 & 560 & 3000 \end{bmatrix} \xrightarrow[E_{12}(\frac{-1}{5})]{} \begin{bmatrix} 0.0055 & 0.095 & 9.6 \cdot 10^2 & 5.0 \cdot 10^3 \\ 0 & -0.009 & -80.0 & -4.0 \cdot 10^2 \\ 0.0093 & 0.025 & 5.6 \cdot 10^2 & 3.0 \cdot 10^3 \end{bmatrix}$$

$$\xrightarrow[E_{13}(\frac{93}{55})]{} \begin{bmatrix} 0.0055 & 0.095 & 9.6 \cdot 10^2 & 5.0 \cdot 10^3 \\ 0 & -0.009 & -80.0 & -4.0 \cdot 10^2 \\ 0 & -0.14 & -1.1 \cdot 10^3 & -5.4 \cdot 10^3 \end{bmatrix}$$

$$\xrightarrow[E_{23}(\frac{-14}{.9})]{} \begin{bmatrix} 0.0055 & 0.095 & 9.6 \cdot 10^2 & 5.0 \cdot 10^3 \\ 0 & -0.009 & -80.0 & -4.0 \cdot 10^2 \\ 0 & 0 & 1.4 \cdot 10^2 & 5.7 \cdot 10^2 \end{bmatrix}$$

Mining example: solve (and then compare to exact)

```
soln=M4[0:3,0:3].solve(rhs=M4[:,3])
```

This has solutions $\begin{bmatrix} 5.7 \cdot 10^4 \\ 8.9 \cdot 10^3 \\ 4.0 \end{bmatrix}$.

. . .

How do these compare to the exact solutions? These are

```
M[0:3,0:3].solve(rhs=M[:,3])
```

$$\begin{bmatrix} 56753.6889897841 \\ 8626.56072644719 \\ 4.02951191827469 \end{bmatrix}$$

Doing it again, rounding after 15 digits

```
# Do the Gauss-Jordan elimination
M=Matrix([[.0055,.095,960,5000],[.0011,.01,112,600],[.0093,.025,560,3000]])
round_level = 15;
ml = latex(M)
M2=N(M.elementary_row_op('n->n+km', row=1,row2=0, k=-(M[1,0]/M[0,0])),round_level)
M3=N(M2.elementary_row_op('n->n+km', row=2,row2=0, k=-(M2[2,0]/M2[0,0])),round_level)
M4=N(M3.elementary_row_op('n->n+km', row=2,row2=1, k=-(M3[2,1]/M3[1,1])),round_level)
```

$$\begin{bmatrix} 0.0055 & 0.095 & 960 & 5000 \\ 0.0011 & 0.01 & 112 & 600 \\ 0.0093 & 0.025 & 560 & 3000 \end{bmatrix} \to \begin{bmatrix} 0.0055 & 0.095 & 960.0 & 5000.0 \\ 0 & -0.009 & -80.0 & -400.0 \\ 0.0093 & 0.025 & 560.0 & 3000.0 \end{bmatrix}$$

$$\to \begin{bmatrix} 0.0055 & 0.095 & 960.0 & 5000.0 \\ 0 & -0.009 & -80.0 & -400.0 \\ 0 & -0.135636363636364 & -1063.27272727273 & -5454.54545454545 \end{bmatrix} \to$$
$$\begin{bmatrix} 0.0055 & 0.095 & 960.0 & 5000.0 \\ 0 & -0.009 & -80.0 & -400.0 \\ 0 & 0 & 142.383838383838 & 573.737373737372 \end{bmatrix}$$

```
soln2=M4[0:3,0:3].solve(rhs=M4[:,3])
```

This has solutions $\begin{bmatrix} 56753.6889897845 \\ 8626.56072644727 \\ 4.02951191827468 \end{bmatrix}$.

Reminder, the exact solution was:

```
M[0:3,0:3].solve(rhs=M[:,3])
```

$$\begin{bmatrix} 56753.6889897841 \\ 8626.56072644719 \\ 4.02951191827469 \end{bmatrix}$$

Takeaway (mining example)

- Gauss–Jordan gives the correct answer in **exact arithmetic**.
- In floating-point arithmetic, intermediate rounding can change the computed result.
- Next: we make “accuracy vs stability” concrete with roundoff examples and pivoting.

Comparing the G-J algorithm with our goals

Reminder of goals

An algorithm should be:

- feasible
- accurate
    - stable
- efficient
    - reusable computations

Roundoff errors

Roundoff errors with G-J elimination

Try this on your calculator. What do you get?

$$\left(\left(\frac{2}{3}+100\right)-100\right)-\frac{2}{3}$$

. . .

Should this be (0)?

Roundoff example: tiny ( ) and loss of significance

- Let ϵ be a number so small that our calculator yields $1+\epsilon=1$.
- With this calculator, $1+1/\epsilon=(\epsilon+1)/\epsilon=1/\epsilon$
- Want to solve the linear system
-

$$\begin{aligned} \epsilon x_1 + x_2 &= 1 \\ x_1 - x_2 &= 0. \end{aligned}$$

Roundoff errors with G-J elimination

With our calculator,

$$\begin{bmatrix} \epsilon & 1 & 1 \\ 1 & -1 & 0 \end{bmatrix} \xrightarrow{E_{21}\left(-\frac{1}{\epsilon}\right)} \begin{bmatrix} \epsilon & 1 & 1 \\ 0 & \frac{1}{\epsilon} & -\frac{1}{\epsilon} \end{bmatrix} \xrightarrow{E_2(\epsilon)} \begin{bmatrix} \epsilon & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{E_{12}(-1)} \begin{bmatrix} \epsilon & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{E_1\left(\frac{1}{\epsilon}\right)} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

. . .

Calculated solution: $x_1 = 0, x_2 = 1$

Correct answer should be

$$x_1 = x_2 = \frac{1}{1+\epsilon} = 0.999999099999990\ldots$$

Sensitivity to small changes

Problem arose because we took a computational step where we added two numbers of very different scale, essentially losing the smaller number.

Led to a big changes in output!

There is no general cure for these difficulties…

Want to be *aware* of them, know when we are doing computations that might be susceptible.

Partial pivoting

We can improve performance of G-J by introducing a new step into the algorithm:

1. Find the entry in the left column with the largest absolute value. This entry is called the pivot. Perform row interchange (if necessary), so that *the pivot* is in the first row.
2. Use a row operation to get a 1 as the entry in the first row and first column.

3. Use row operations to make all other entries as zeros in column one.
4. Interchange rows if necessary to obtain a nonzero number with the largest absolute value in the second row, second column. Use a row operation to make this entry 1. Use row operations to make all other entries as zeros in column two.
5. Repeat step 4 for row 3, column 3. Continue moving along the main diagonal until you reach the last row, or until the number is zero.

Using partial pivoting in our example

Ill-conditioned linear systems

Ill-conditioned linear systems

A system of linear equations is said to be **ill-conditioned** when some small perturbation in the system (in the bs) can produce relatively large changes in the exact solution (in the x's). Otherwise, the system is said to be **well-conditioned**.

Ill-conditioned linear systems

Consider

$$\begin{aligned} .835x + .667y &= .168, \\ .333x + .266y &= .067, \end{aligned}$$

Meyer Ch 1.6

Exact solution:

$$x = 1 \quad \text{and} \quad y = -1.$$

But if we change just one digit...

$$\begin{aligned} .835x + .667y &= .168, \\ .333x + .266y &= .066, \end{aligned}$$

Now exact solution:

$$\hat{x} = -666 \quad \text{and} \quad \hat{y} = 834$$

Measurement error (→) big changes in the exact solution

What if b_1 and b_2 are the results of an experiment, need to be read off a dial? Suppose:

- dial can be read to tolerance of $\pm.001$,
- values for b_1 and b_2 are read as .168 and .067, respectively. Then the exact solution is

$$(x, y) = (1, -1)$$

- But due to uncertainty, we have

$$.167 \leq b_1 \leq .169 \quad \text{and} \quad .066 \leq b_2 \leq .068$$

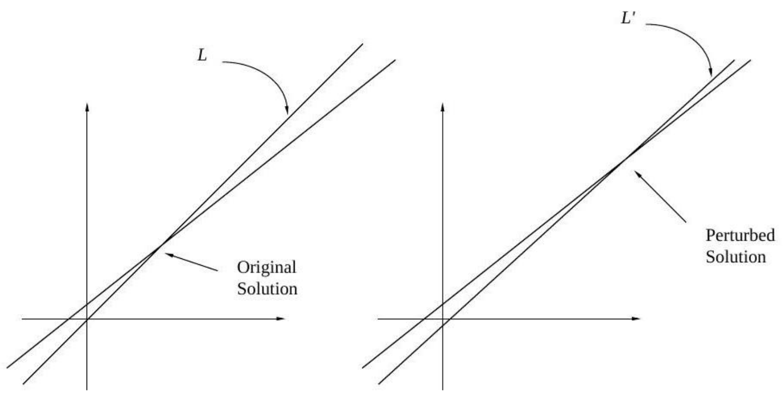
What range of solutions could we see?

Table 1: Possible readings

b_1	b_2	x	y
.168	.067	1	-1
.167	.068	934	-1169
.169	.066	-932	1169

Geometrical interpretation

If two straight lines are almost parallel and if one of the lines is moved only slightly, then the point of intersection is drastically altered.



The point of intersection is the solution of the associated 2×2 linear system, so this is also drastically altered.

***Takeaway: conditioning is about the* problem**

- Often in real life, coefficients are empirically obtained
- Will be off from "true" values by small amounts
- For ill-conditioned systems, this means that solutions can be very far off from true solutions
- We'll cover techniques for quantifying conditioning, later in quarter
- For now, can just try making small changes to some coefficients. Big changes in result? Ill-conditioned system!

Bridge to the next example: interpolation can create ill-conditioned systems

- When we fit high-degree polynomials from data, we often build matrices (e.g. Vandermonde matrices) that can be numerically nasty.
- Next: polynomial interpolation as a concrete "looks reasonable, behaves badly" example.

Example: Polynomial interpolation

Polynomial interpolation

Suppose we'd like to find a polynomial that can interpolate a given function.

https://colab.research.google.com/github/quantecon/le
julia.notebooks/blob/main/tools_and_techniques/iter

Take points 0,... and values 0,...

Simple way: finding the coefficients $ c_1, ...c_n $ where

$$P(x) = \sum_{i=0}^{N} c_i x^i$$

. . .

This is a system of equations:

$$\begin{gathered} y_0 = c_0 + c_1 x_0 + \dots c_N x_0^N \\ \dots \\ y_N = c_0 + c_1 x_N + \dots c_N x_N^N \end{gathered}$$

Polynomial interpolation

Or, stacking, $c = \begin{bmatrix} c_0 & \dots & c_N \end{bmatrix}$, $y = \begin{bmatrix} y_0 & \dots & y_N \end{bmatrix}$, and

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^N \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^N \end{bmatrix}$$

Let's try solving this for a simple function, $y = exp(x)$.

Polynomial interpolation

$$x * y$$

```
import numpy as np
import matplotlib.pyplot as plt

# We are introducing a small error into b. Does it have a big impact on the x's we find?
def approx(n_points,jiggle=0):
  x = np.linspace(1,12,num=n_points)
  #xshuff = x+np.random.normal(size=n_points,scale=jiggle)
  y = np.exp(x)+np.exp(x/4)*np.random.normal(size=n_points,scale=jiggle)
  # Get matrix of exponents of x values => A
  n = len(x)
  A = np.zeros([n,n])
  for i in range(n):
    A[::,i] = np.power(x.T,i)
  b = y

  # Solve Ax=b linear eq. system to get
  s = np.linalg.solve(A, b)
  # where x denotes coeffs of polynomial in reverse order
  # Flip polynomial coeffs
  s = np.flip(s,axis=0)
  # Print polynomial coeffs
  return s, x, y, A
s, x, y, A = approx(4)
# Evaluate polynomial at X axis and plot result
```

We'll start with picking 4 interpolating points: $\vec{x} =$ [1. 4.66666667 8.33333333 12.]. Then we have our matrix A:

```
np.set_printoptions(suppress=True)
```

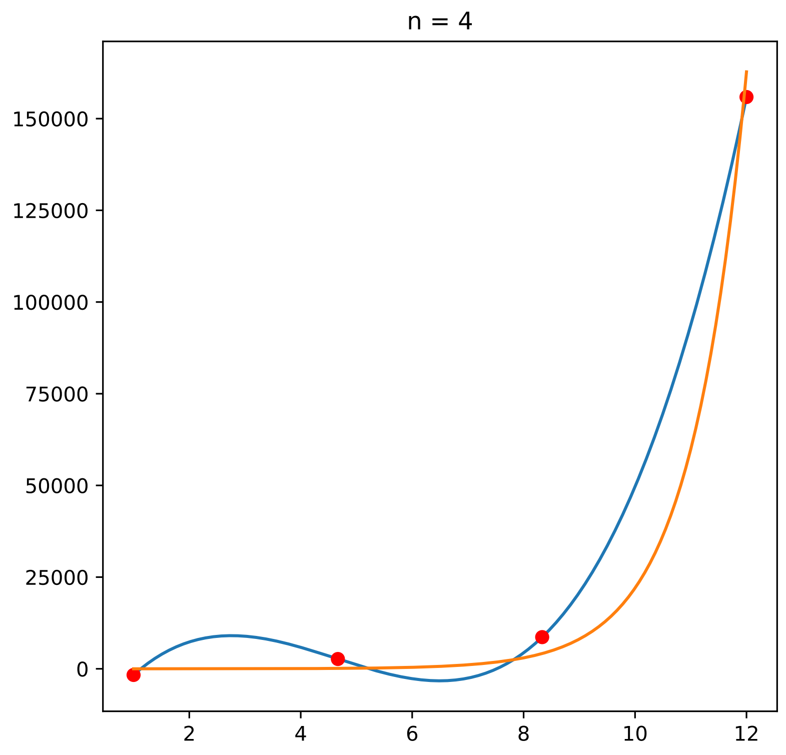
$$A = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^N \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^N \end{bmatrix} = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 4.66666666666667 & 21.7777777777778 & 101.62962962963 \\ 1.0 & 8.33333333333333 & 69.4444444444444 & 578.703703703703 \\ 1.0 & 12.0 & 144.0 & 1728.0 \end{bmatrix}$$

Our y values are $\vec{y} =$ [2.71828183 106.3426754 4160.26200538 162754.791419]. We could solve this using Gauss-Jordan elimination, or here the solving algorithm built into Numpy.

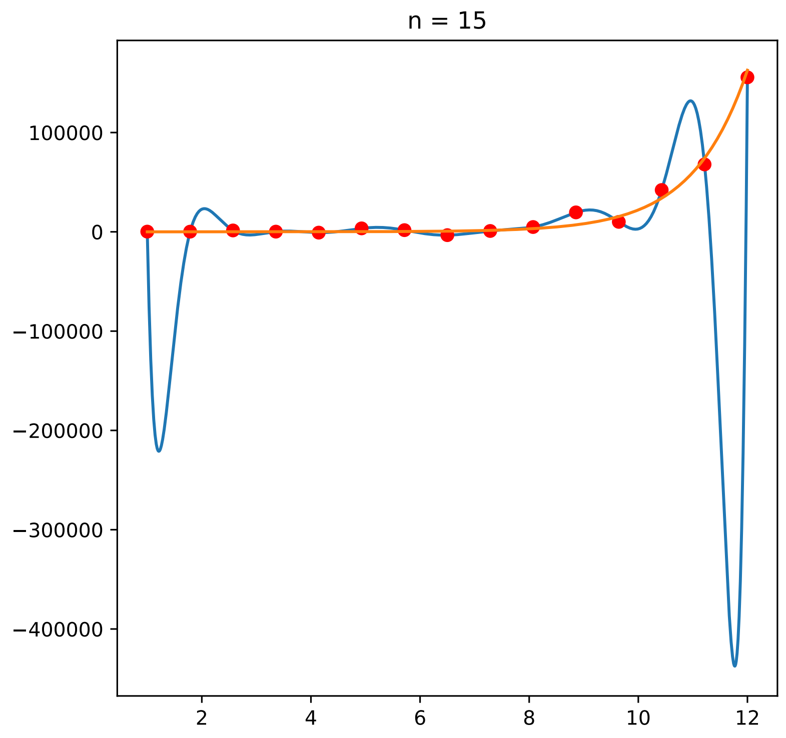
```
def make_plots(s, x, y, A):
  x_axis = np.linspace(np.min(x), np.max(x), num=5000)
  y_axis = np.polyval(s, x_axis)
  y_pred = np.polyval(s,x)
  plt.clf();
  plt.plot(x_axis, y_axis);
  plt.title("n = " + str(len(x)));
  plt.plot(x,y,'ro');
  plt.plot(x_axis,np.exp(x_axis));
  plt.show();

def error_norm(s,x,y,A):
    y_pred = np.polyval(s,x)
    return(np.linalg.norm(y_pred-np.exp(x),np.inf))
```

Results for n=4 and n=10 points

```
_ = plt.figure(figsize=(6,6));
make_plots(*approx(4,jiggle=1000));
```



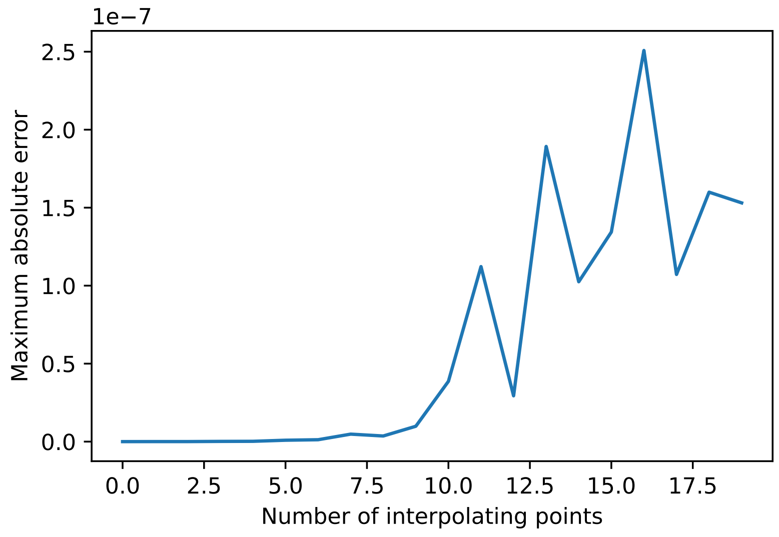
```
_ = plt.figure(figsize=(6,6));
make_plots(*approx(15,jiggle=1000));
```



Max error for different values of n

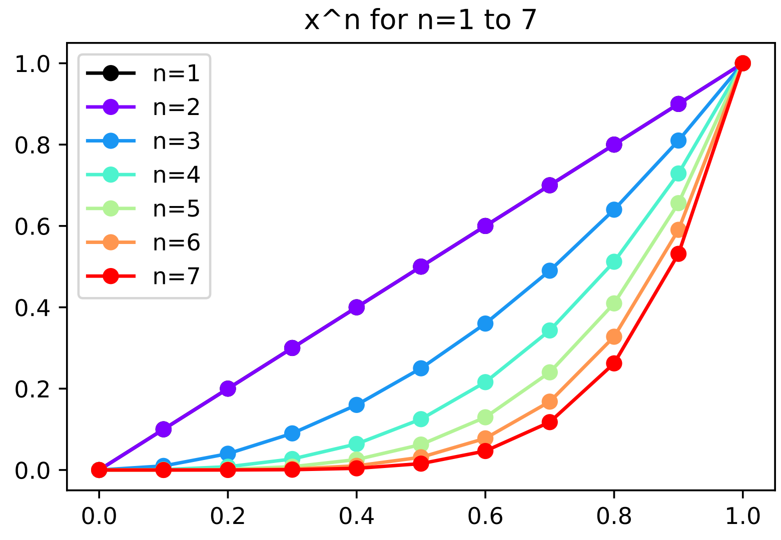
```
my_error = []
for i in range(1,21):
  my_error.append(error_norm(*approx(i)))
```

```
plt.clf()
plt.plot(my_error)
plt.xlabel("Number of interpolating points")
plt.ylabel("Maximum absolute error")
plt.show()
```



The problem: linear dependency

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 1.1, 0.1)
plt.plot(x, x, marker='o', linestyle='-', color='black', label='n=1')

colors = plt.cm.rainbow(np.linspace(0, 1, 6))

for i in range(1, 7):
  plt.plot(x, x**i, marker='o', linestyle='-', color=colors[i-1], label=f'n={i+1}')

plt.title("x^n for n=1 to 7")
plt.legend()
plt.show()
```



Wrap-up

- Gauss-Jordan elimination is a good *idea* (systematic row operations), but not always the best computational tool
- Roundoff can turn "should be zero" into "is not zero"
- Pivoting is a cheap way to make elimination behave much better
- Ill-conditioning is different: sometimes the system itself is sensitive