

PFE 3

Dr W Palubicki

Topics

- Numpy/Scipy
 - Linear Algebra
 - Functional Analysis
- Purpose
 - Understand basics of data science tools

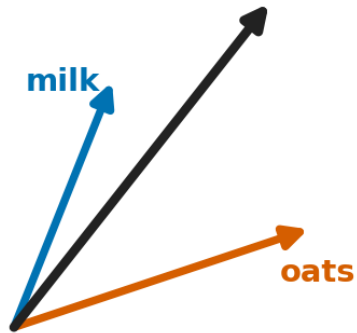
Linear algebra (numpy.linalg)

Matrix and vector products

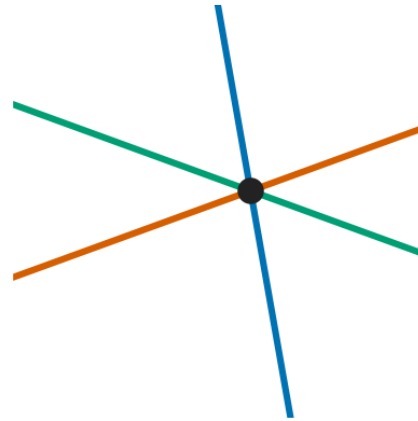
<code>dot(a, b[, out])</code>	Dot product of two arrays.
<code>linalg.multi_dot(arrays)</code>	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
<code>vdot(a, b)</code>	Return the dot product of two vectors.
<code>inner(a, b)</code>	Inner product of two arrays.
<code>outer(a, b[, out])</code>	Compute the outer product of two vectors.
<code>matmul(x1, x2, /[, out, casting, order, ...])</code>	Matrix product of two arrays.
<code>tensordot(a, b[, axes])</code>	Compute tensor dot product along specified axes for arrays ≥ 1 -D.
<code>einsum(subscripts, *operands[, out, dtype, ...])</code>	Evaluates the Einstein summation convention on the operands.
<code>einsum_path(subscripts, *operands[, optimize])</code>	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
<code>linalg.matrix_power(a, n)</code>	Raise a square matrix to the (integer) power n .
<code>kron(a, b)</code>	Kronecker product of two arrays.

Linear systems are everywhere

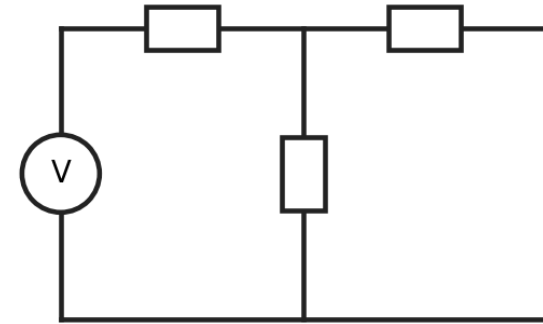
Mixing: a recipe



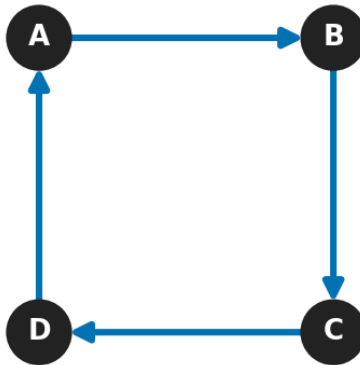
Positioning: lines meet



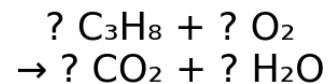
Circuits: currents balance



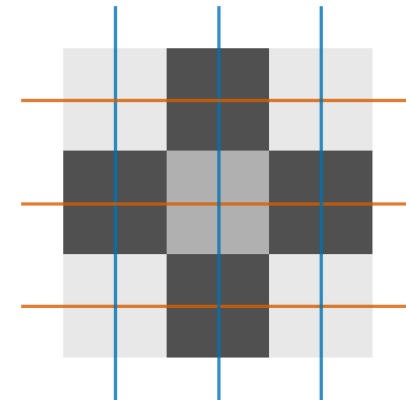
Traffic: in = out



Chemistry: atoms balance



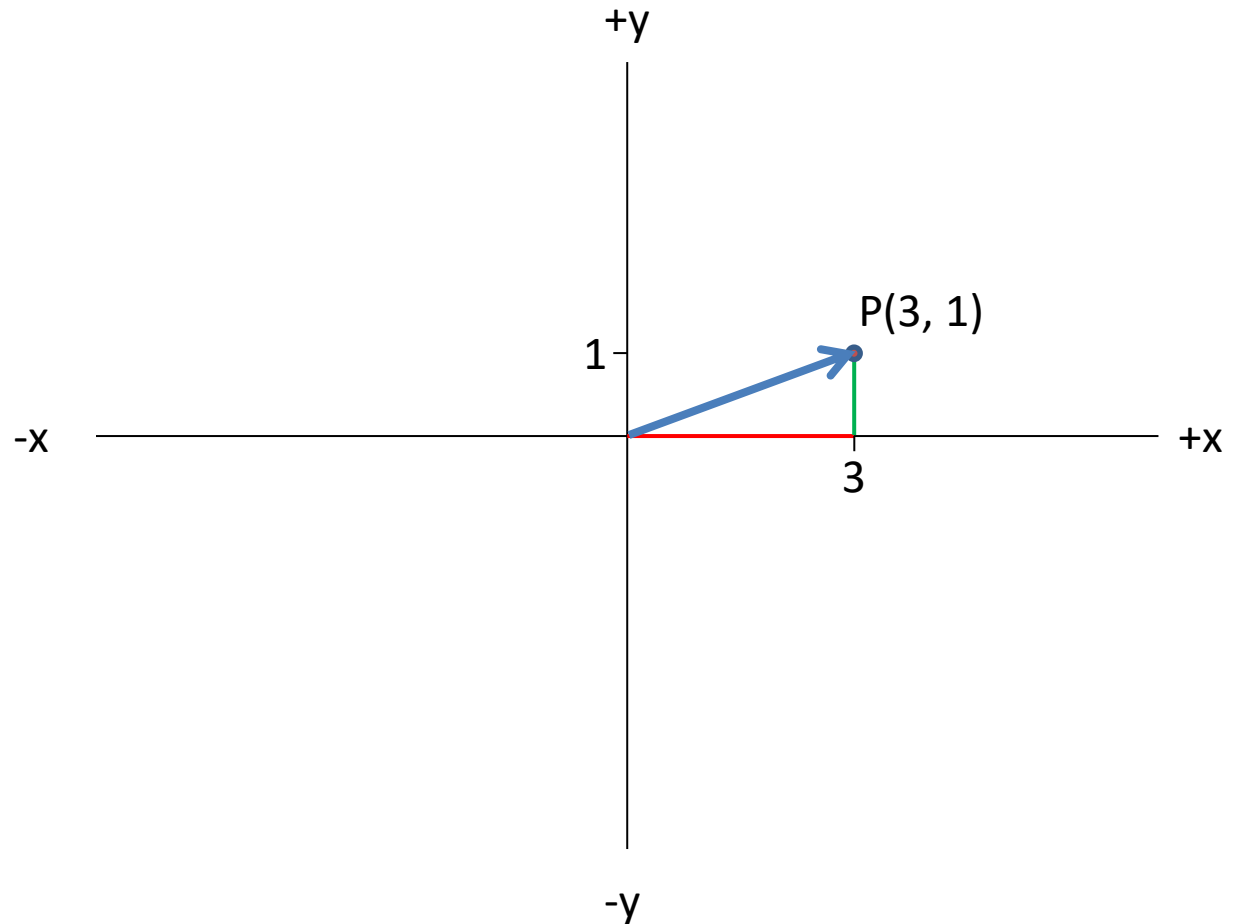
Imaging: rays add up



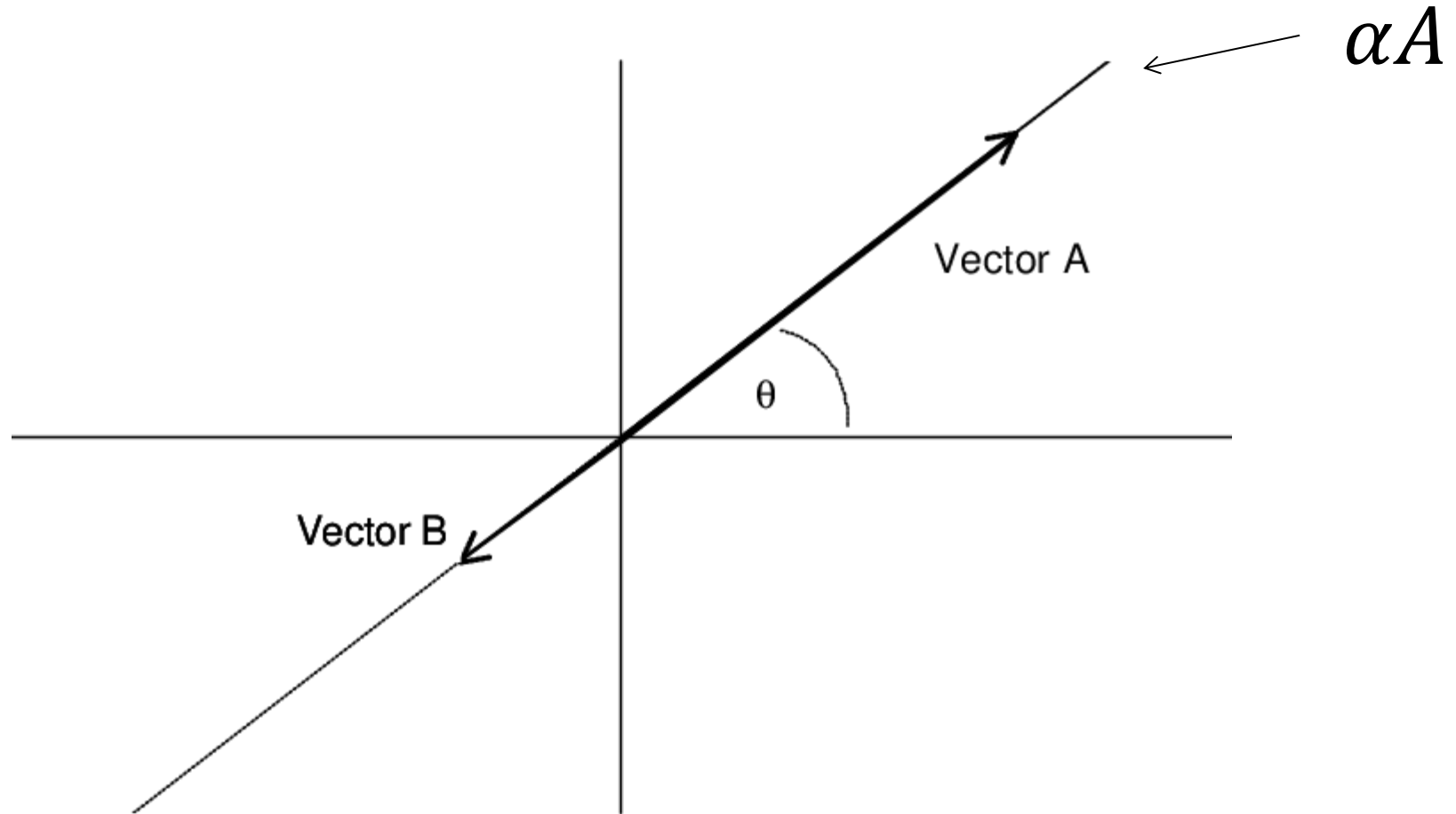
Six different problems, one mathematical shape: several unknowns, several linear conditions.

Vectors

- Physics
- Computer science
- Mathematics



Vector span

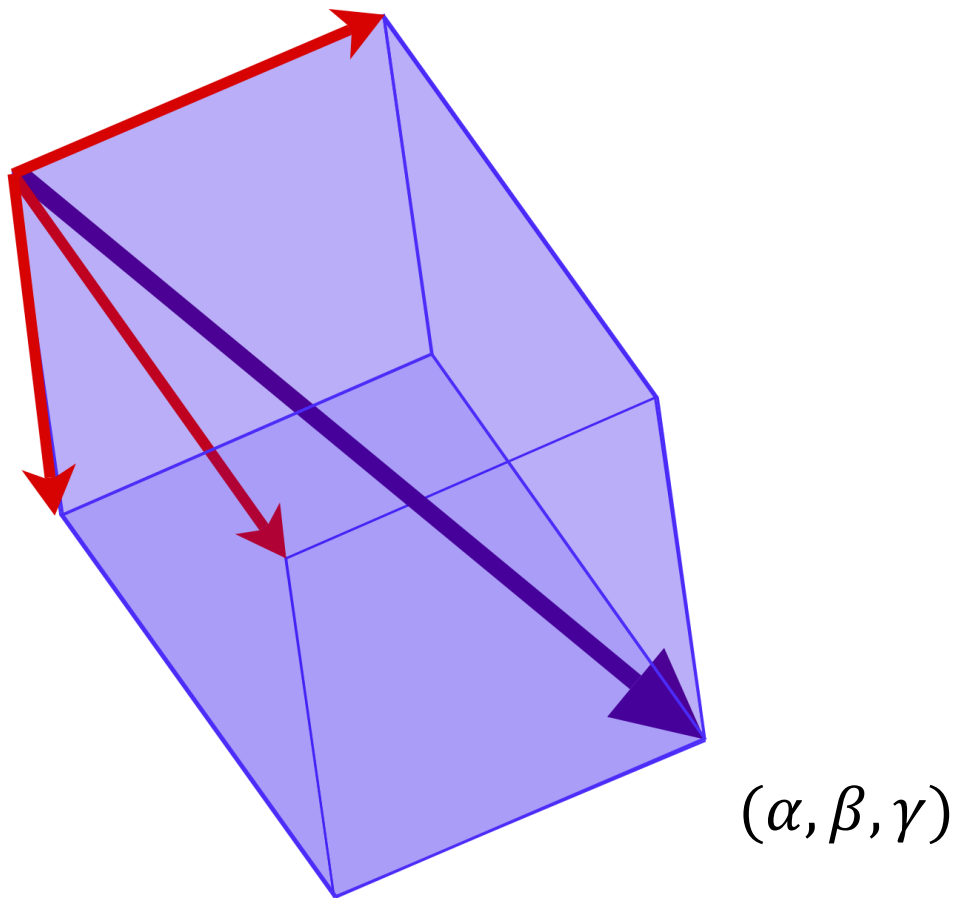


Linear Combination of Vectors

$$\alpha A + \beta B + \gamma C$$

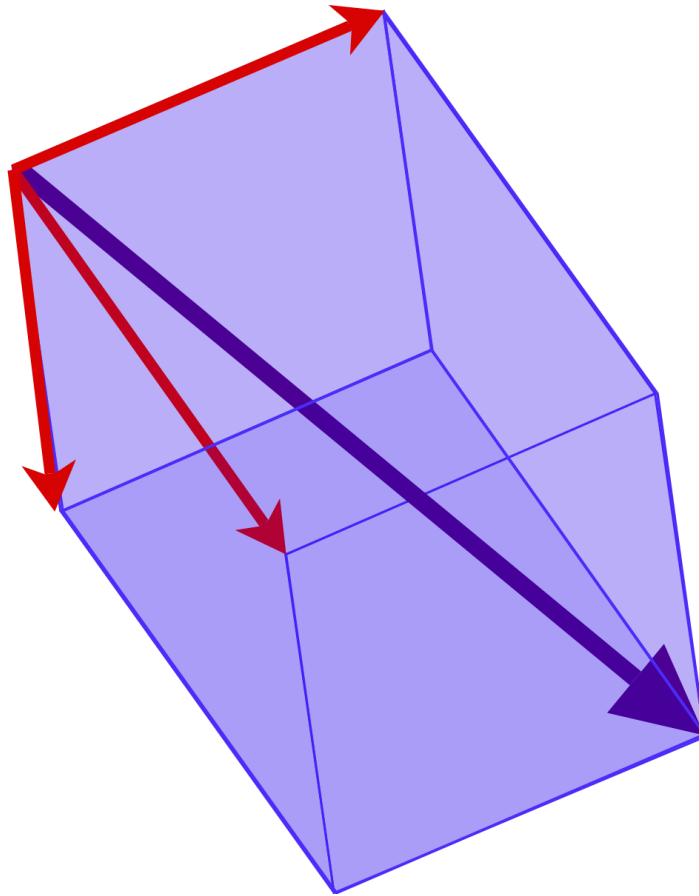
Linear Combination of Vectors

$$\alpha A + \beta B + \gamma C$$

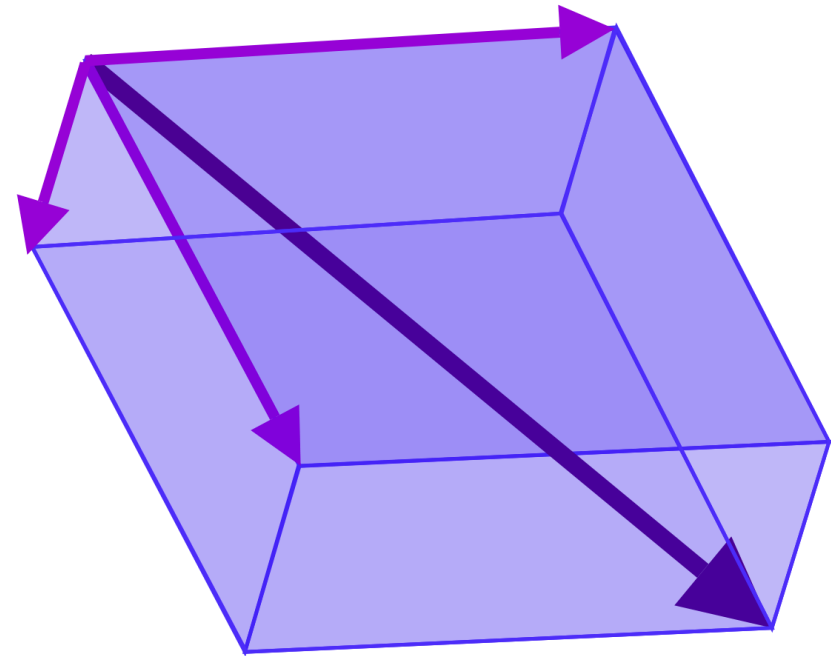


Linear Combination of Vectors

$$\alpha A + \beta B + \gamma C$$



$$(\alpha, \beta, \gamma)$$



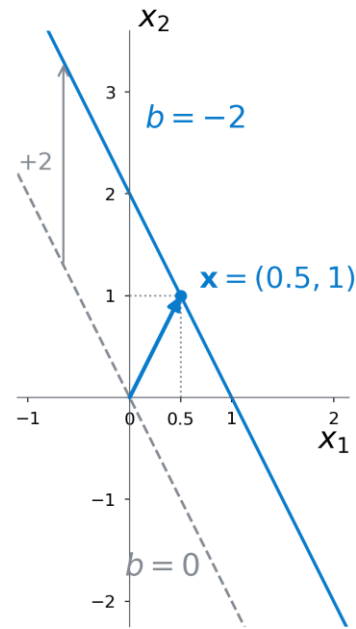
$$\left(\alpha, \frac{1}{2}\beta, \frac{3}{2}\gamma\right)$$

Linear Equation

$$a_1 x_1 + \cdots + a_n x_n + b = 0$$

Linear Equation

$$a_1 x_1 + \cdots + a_n x_n + b = 0$$



$$2x_1 + x_2 - 2 = 0$$

$$a_1 = 2, \quad a_2 = 1, \quad b = -2$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$$

$$2 \cdot 0.5 + 1 \cdot 1 - 2 = 0$$

Linear Equations Example

- You run **0.2 km** every minute.
- The horse runs **0.5 km** every minute, but it takes 6 minutes to saddle the horse.

Linear Equations Example

- You run **0.2 km** every minute.
- The horse runs **0.5 km** every minute, but it takes 6 minutes to saddle the horse.
- We can make **two** equations (**d**=distance in km, **t**=time in minutes)

Linear Equations Example

- You run **0.2 km** every minute.
- The horse runs **0.5 km** every minute, but it takes 6 minutes to saddle the horse.
- We can make **two** equations (**d**=distance in km, **t**=time in minutes)

$$d = 0.2t$$

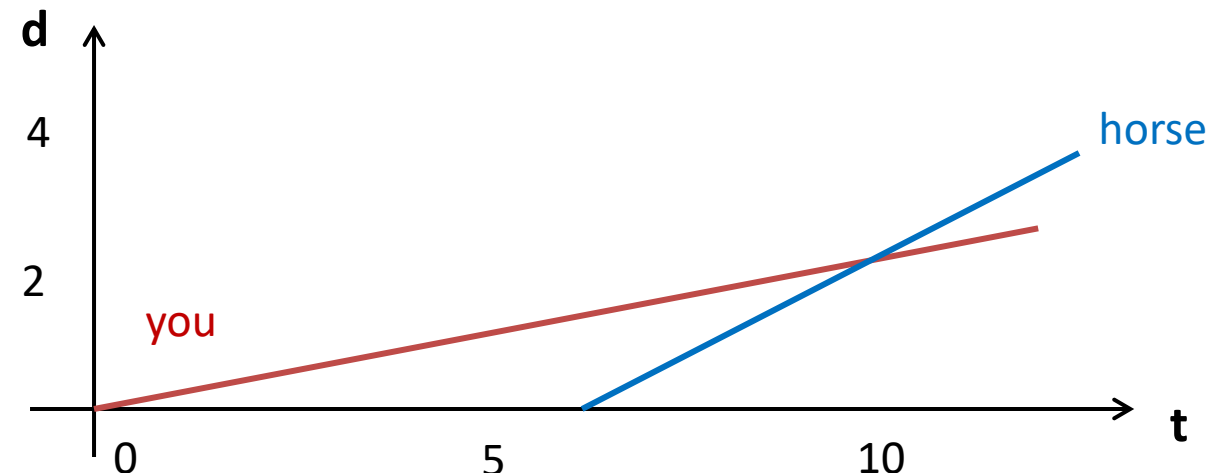
$$d = 0.5(t-6) = 0.5t-3$$

Linear Equations Example

- You run **0.2 km** every minute.
- The horse runs **0.5 km** every minute, but it takes 6 minutes to saddle the horse.
- We can make **two** equations (**d**=distance in km, **t**=time in minutes)

$$d = 0.2t$$

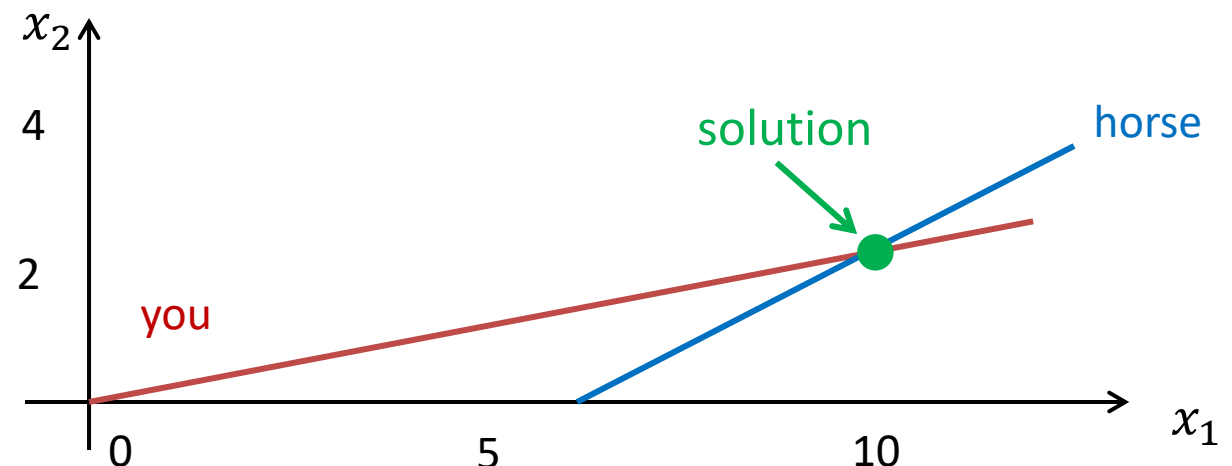
$$d = 0.5(t-6) = 0.5t-3$$



Linear Equations Example

- You run **0.2 km** every minute.
- The horse runs **0.5 km** every minute, but it takes 6 minutes to saddle the horse.
- We can make **two** equations (**d**=distance in km, **t**=time in minutes)

- $0.2x_1 + (-1)x_2 = 0,$
 $0.5x_1 + (-1)x_2 = 3.$



Linear Systems of Equations

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1,$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2,$$

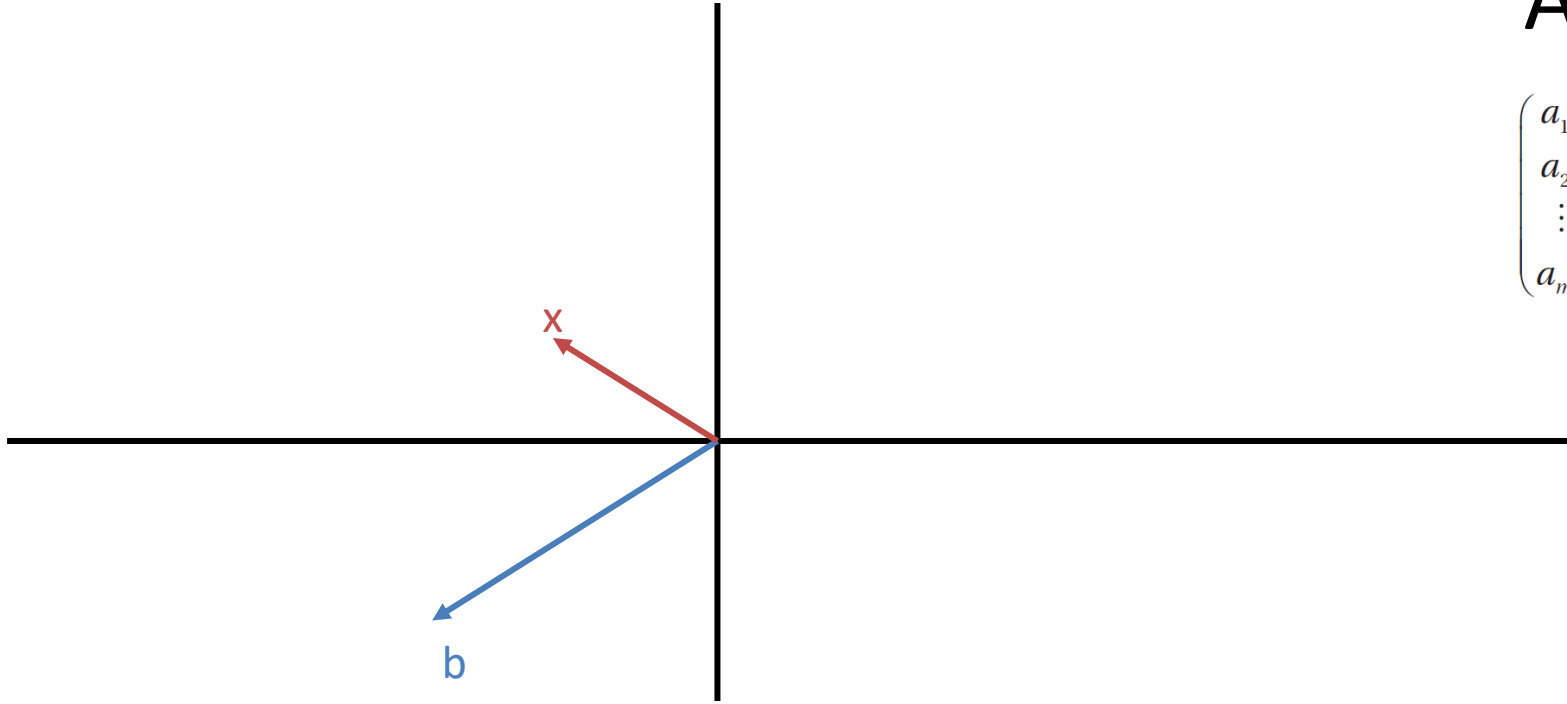
...

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m.$$

Linear Systems of Equations

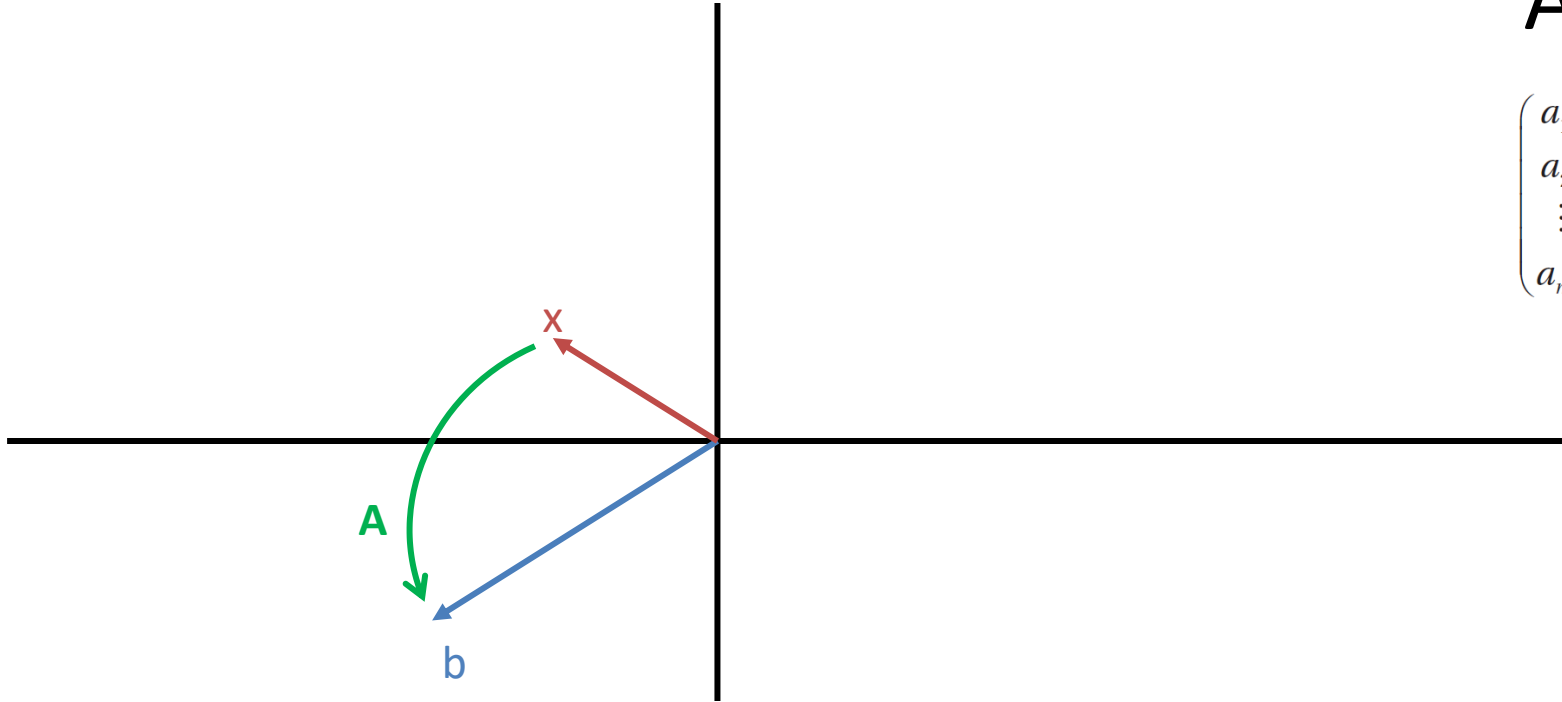
$$\begin{array}{ccc} & \mathbf{A} & \mathbf{x} = \mathbf{b} \\ \left(\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array} \right) & \left(\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right) & = \left(\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_m \end{array} \right) \end{array}$$

Geometric Interpretation



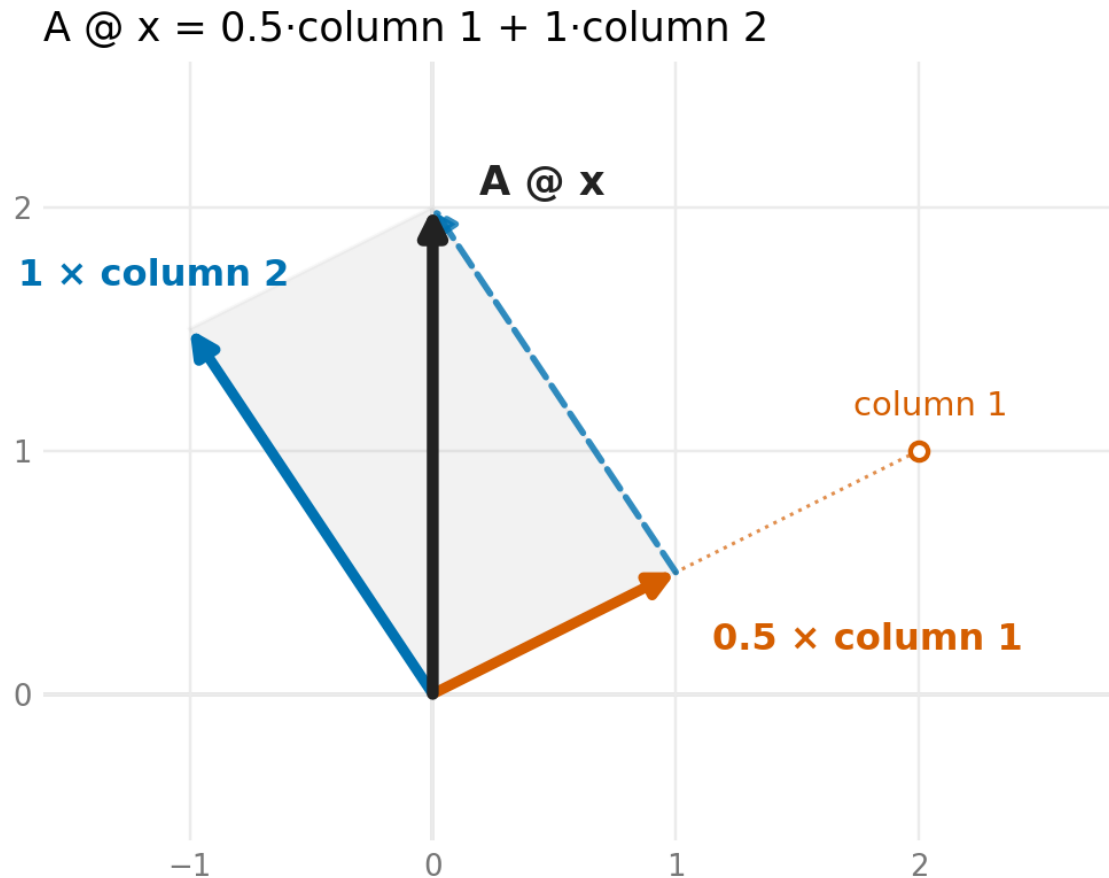
$$\begin{matrix} A & x & b \end{matrix}$$
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Geometric Interpretation



$$\begin{matrix} A & x & b \end{matrix}$$
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

A x: an interpolation of columns



Column picture: $Ax = x_1 \cdot (\text{column 1}) + x_2 \cdot (\text{column 2})$.

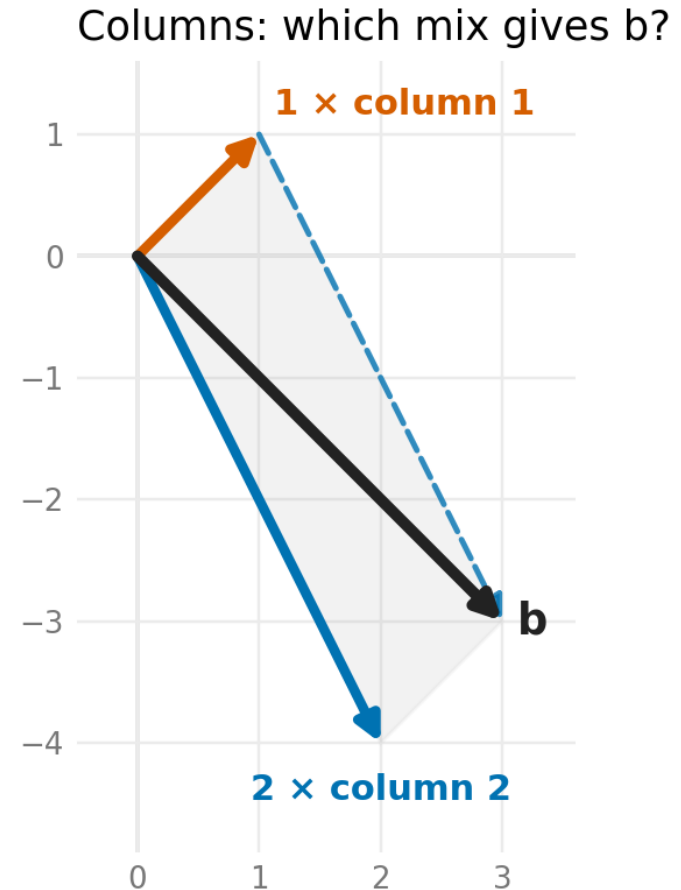
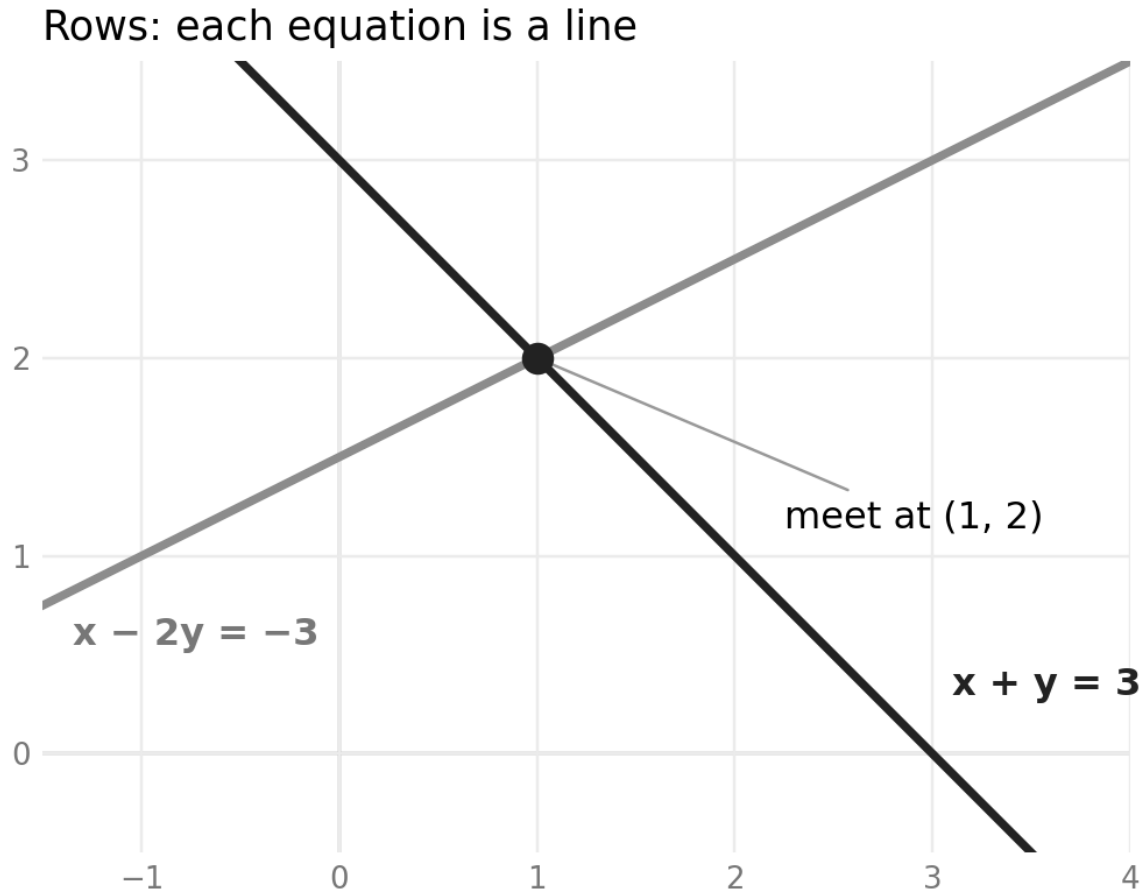
The columns are ingredients, x is the recipe, b is the dish.

Solving $Ax = b$ asks: how much of each column reaches b ?

```
A = np.array([[2, -1],
              [1, 1.5]])
x = np.array([0.5, 1.0])
print(A @ x)
print(x[0]*A[:, 0] + x[1]*A[:, 1])
```

```
[0. 2.]
[0. 2.]
```

Two interpretations



Rows: each equation is a line; the solution is where the lines meet (the horse example).

Columns: the solution says how much of each column to mix to reach b. Both give $x = 1$, $y = 2$.

$$Ax = b?$$

$$Ax = b?$$

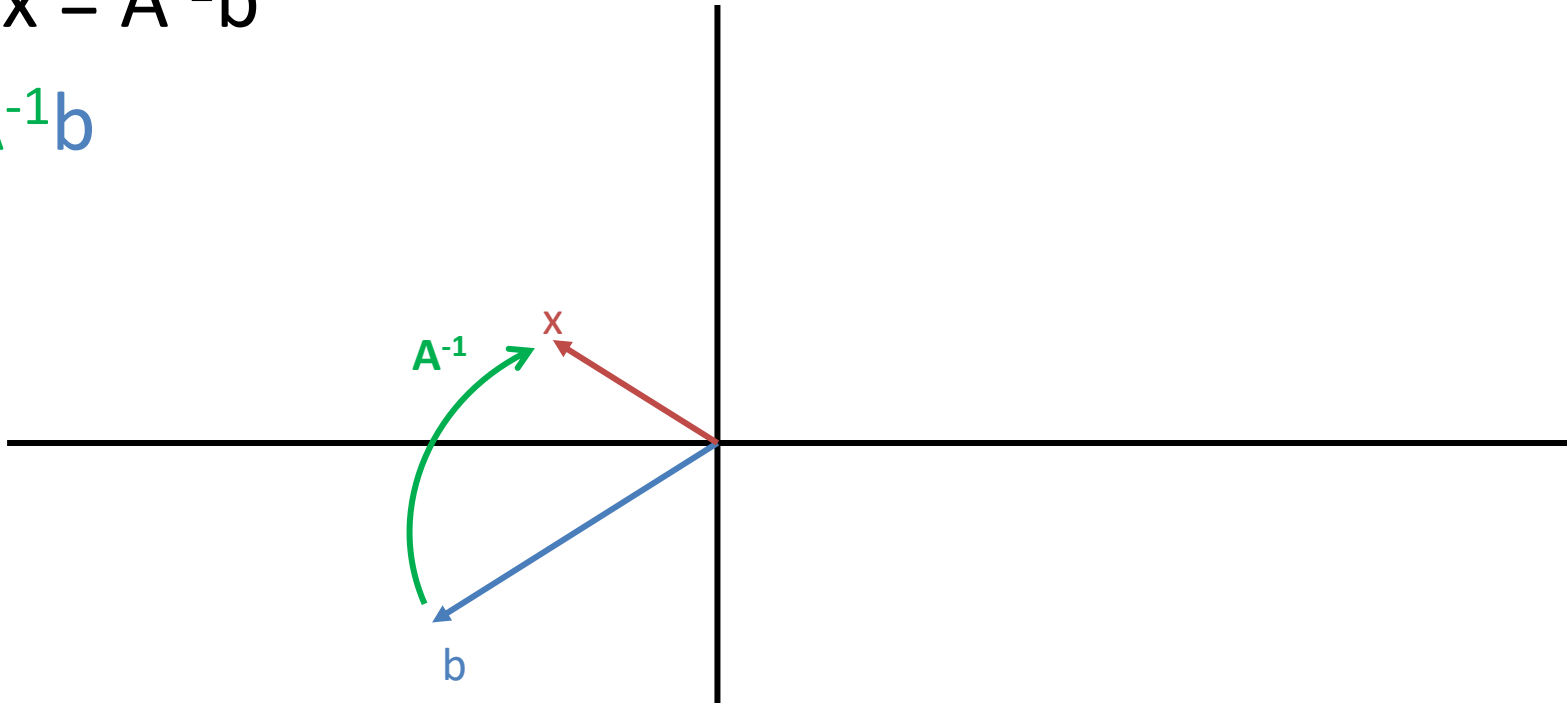
- $A^{-1}Ax = A^{-1}b$

$$Ax = b?$$

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

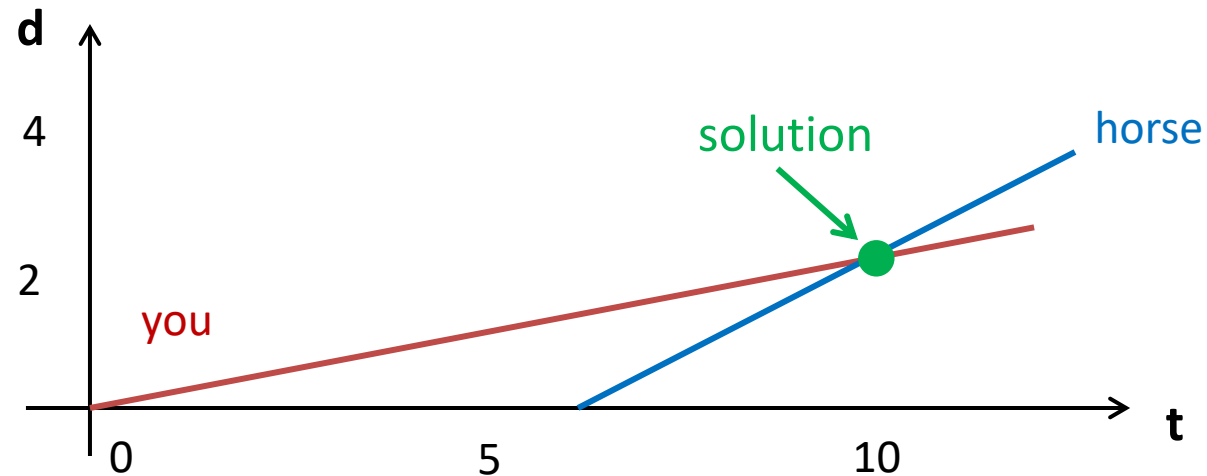
$$Ax = b?$$

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



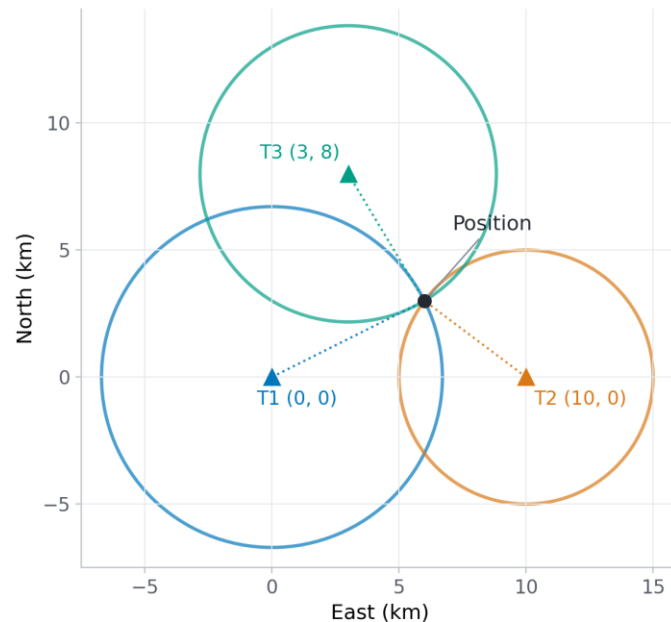
Exercise

- Draw the graph from the horse and human running example.
- What is A , x , and b in this example?
- Draw x and b onto new graphs.



Real life: where am I?

1. Distances define circles

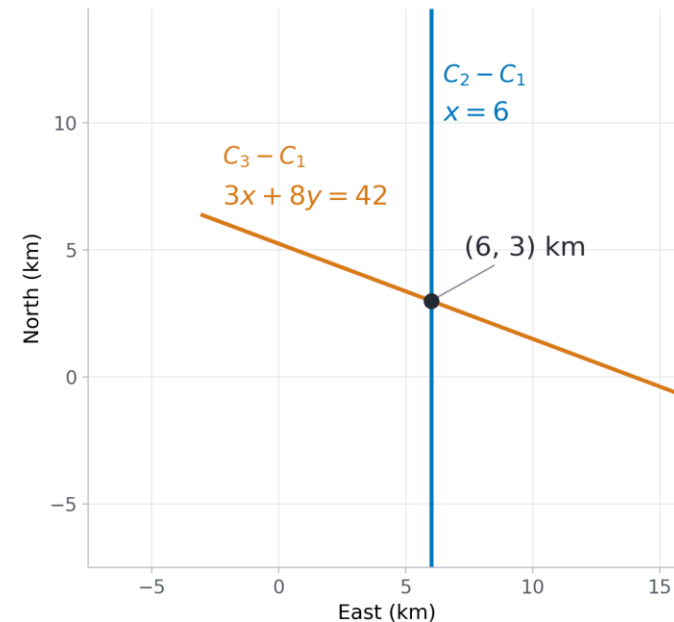


$$C_1: x^2 + y^2 = 45$$

$$C_2: (x - 10)^2 + y^2 = 25$$

$$C_3: (x - 3)^2 + (y - 8)^2 = 34$$

2. Subtract equations, then solve



$$-20x + 100 = -20 \Rightarrow x = 6$$

$$-6x - 16y + 73 = -11 \Rightarrow 3x + 8y = 42$$

- Distances to known towers put you on circles.
- Subtracting two circle equations cancels the squares: a straight line.
- Two such lines meet at your position: a 2×2 linear system.
- GPS does the same in 3D, with the clock error as a fourth unknown.

numpy.linalg.inv

`numpy.linalg.inv(a)`

[\[source\]](#)

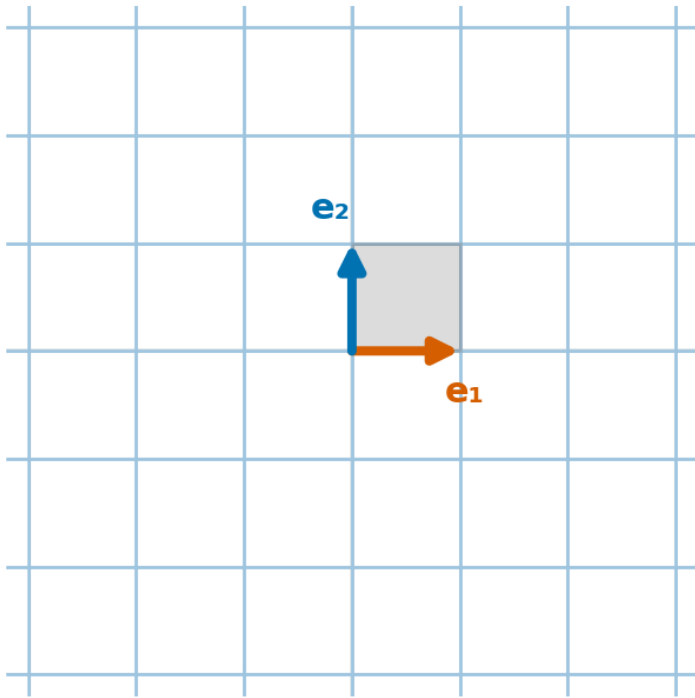
Compute the (multiplicative) inverse of a matrix.

Given a square matrix *a*, return the matrix *ainv* satisfying `dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])`.

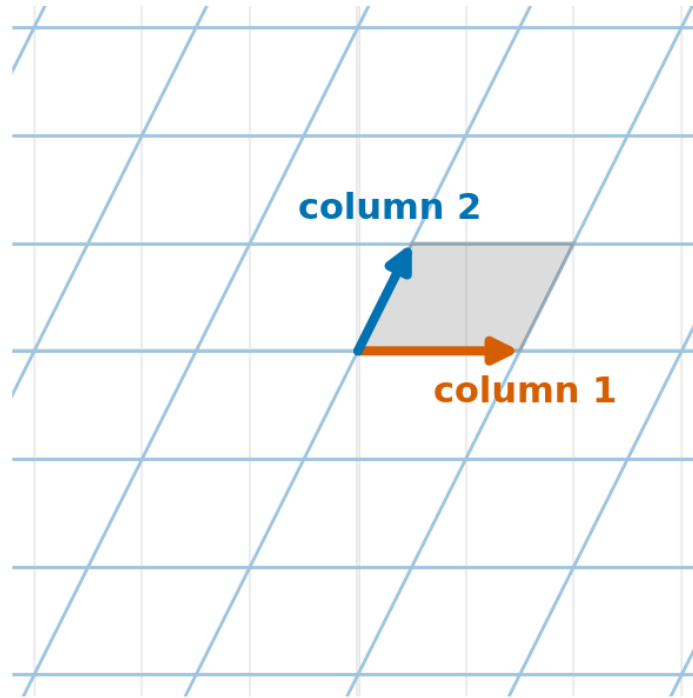
```
>>> a = np.array([[[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]],
      [[-5. ,  2. ],
       [ 3. , -1. ]])
```

A matrix transforms the whole plane

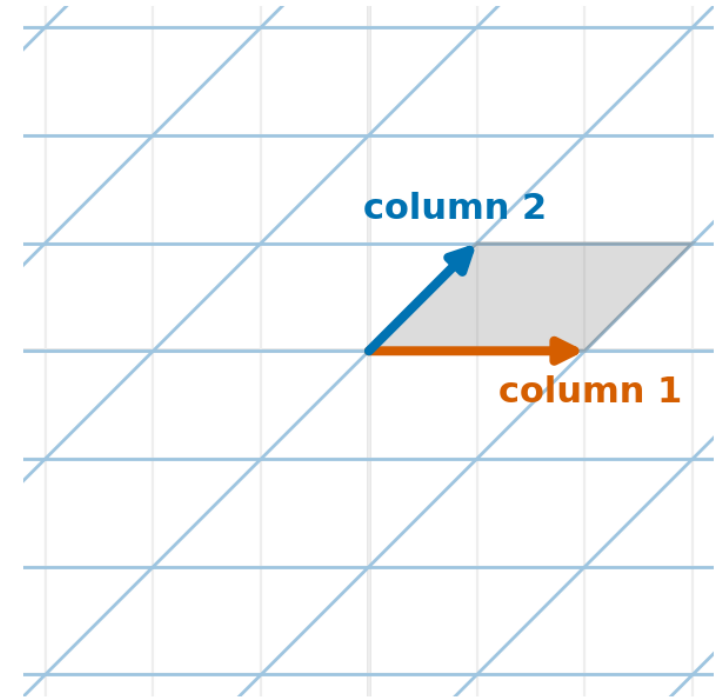
Before



Halfway



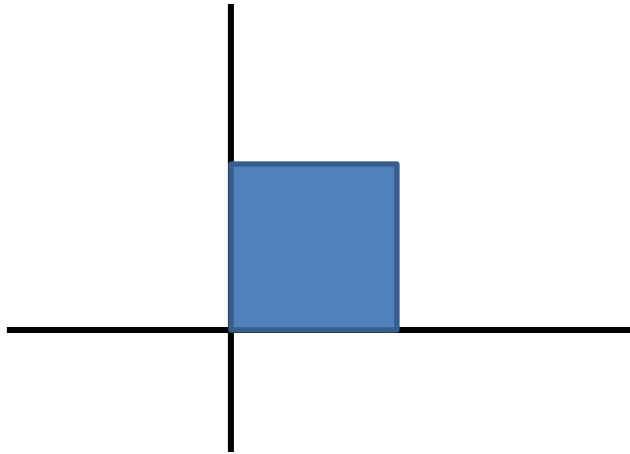
After $A = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$



Linear: grid lines stay straight, parallel and evenly spaced.

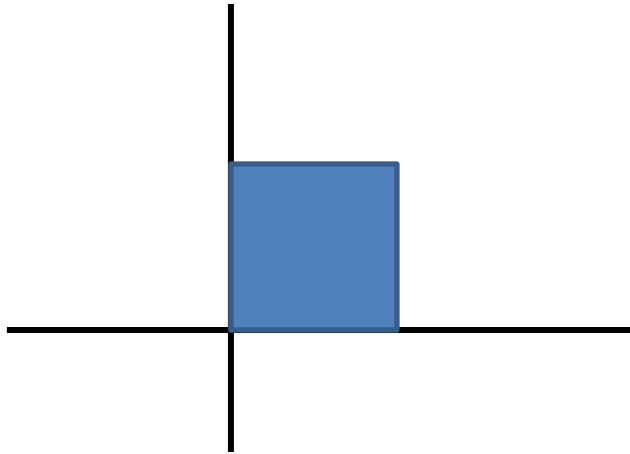
Read the columns: column 1 is where $(1, 0)$ lands, column 2 is where $(0, 1)$ lands. Every other point follows.

Measuring the area changes of transformations



$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

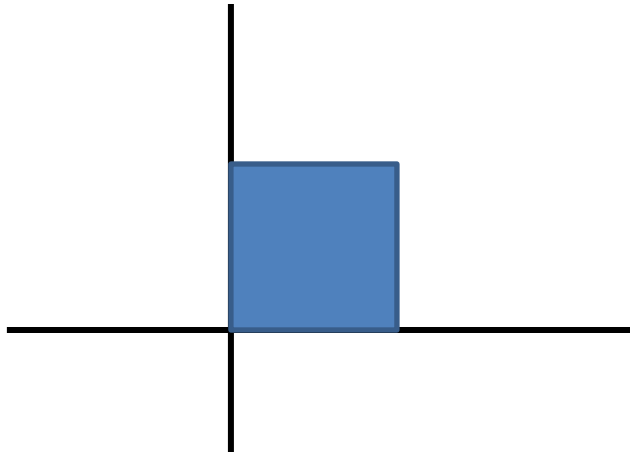
Measuring the area changes of transformations



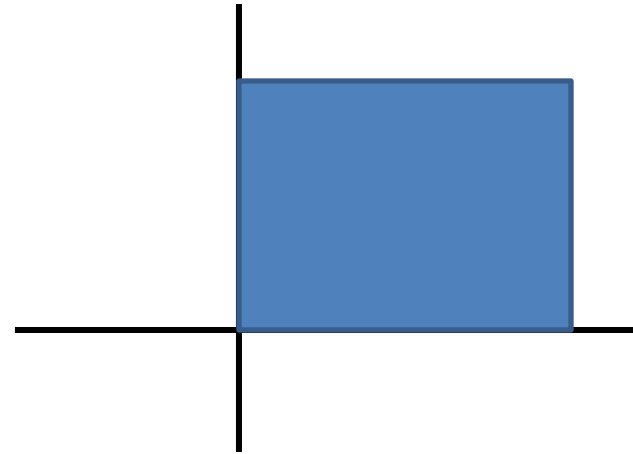
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix}$$

Measuring the area changes of transformations

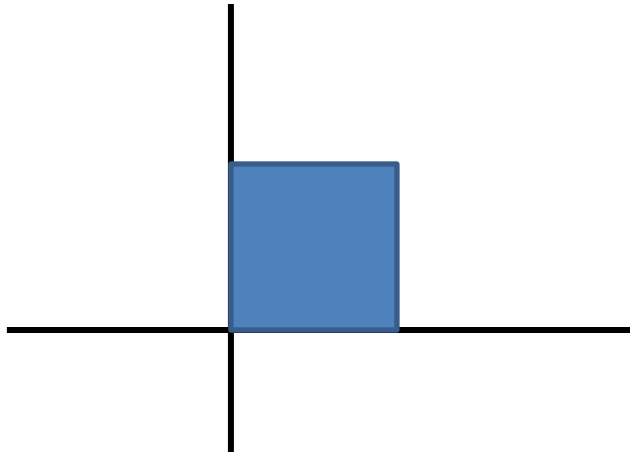


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

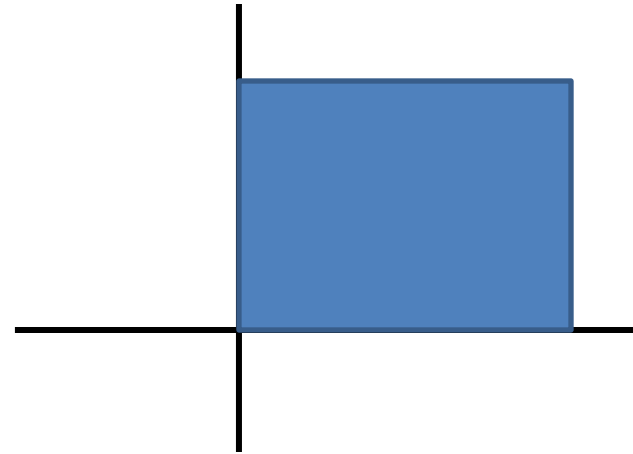


$$\begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix}$$

Measuring the area changes of transformations

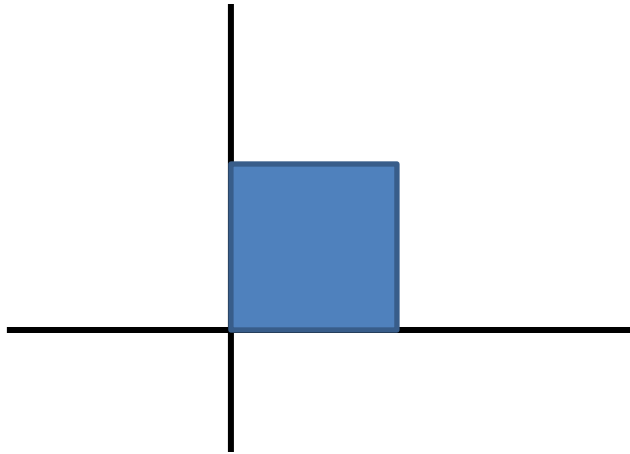


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

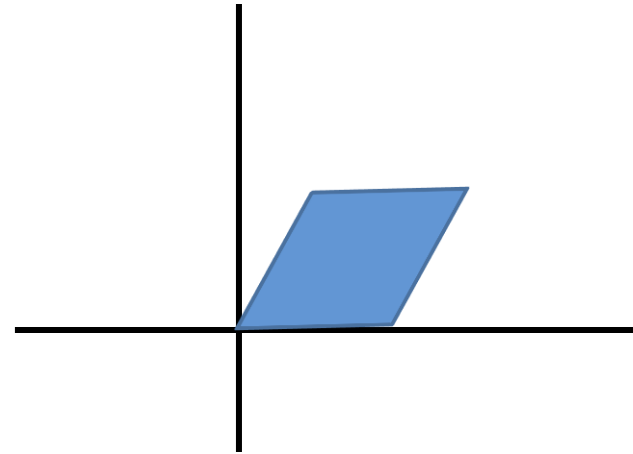


$$\begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix} \rightarrow \text{area } 3$$

Measuring the area changes of transformations

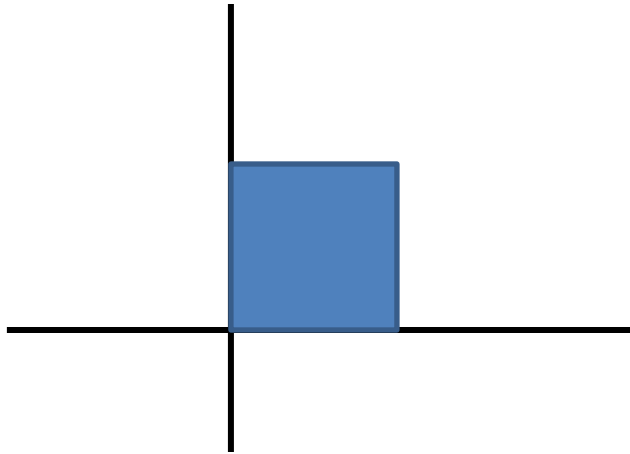


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

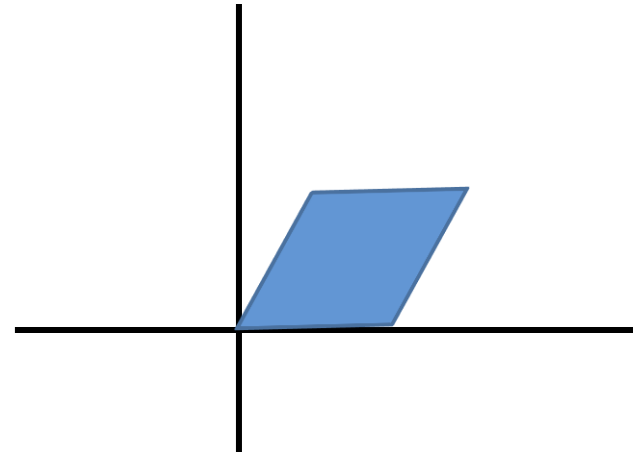


$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

Measuring the area changes of transformations



$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

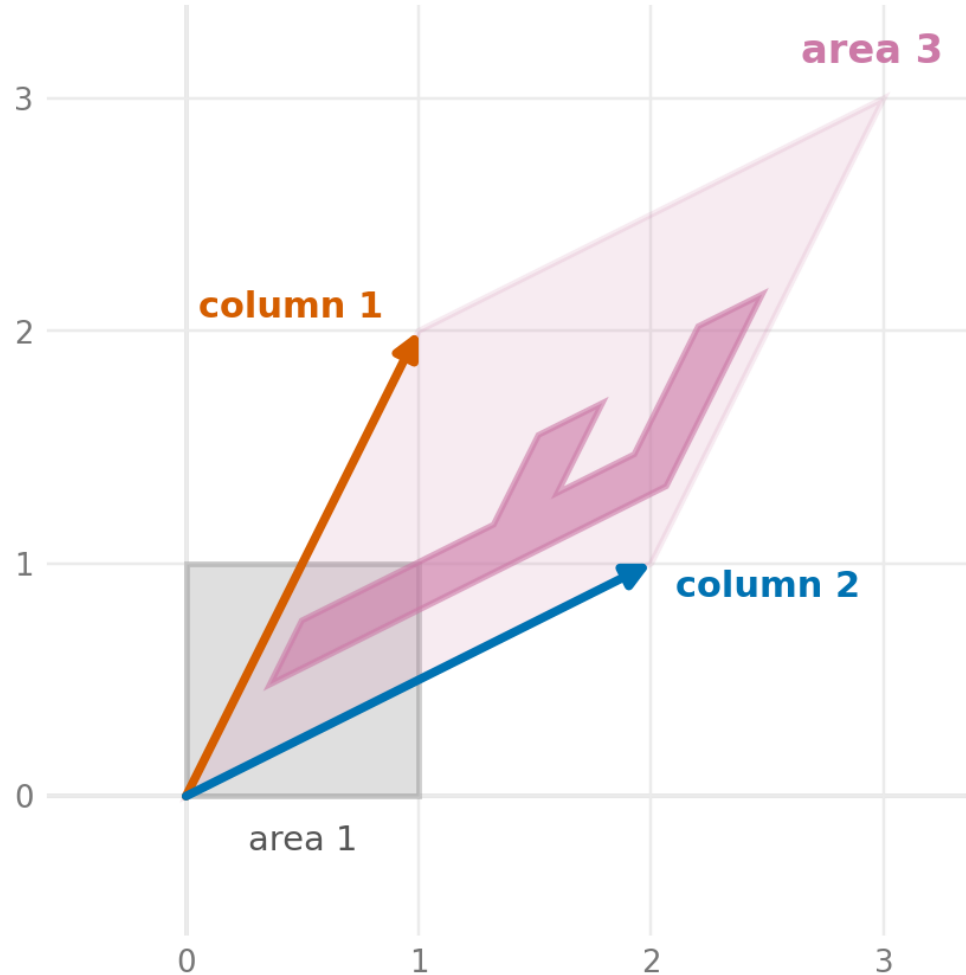


$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

This is the “determinant” of a matrix

Negative determinant: a mirror image

$\det = -3$: areas $\times 3$, F mirrored

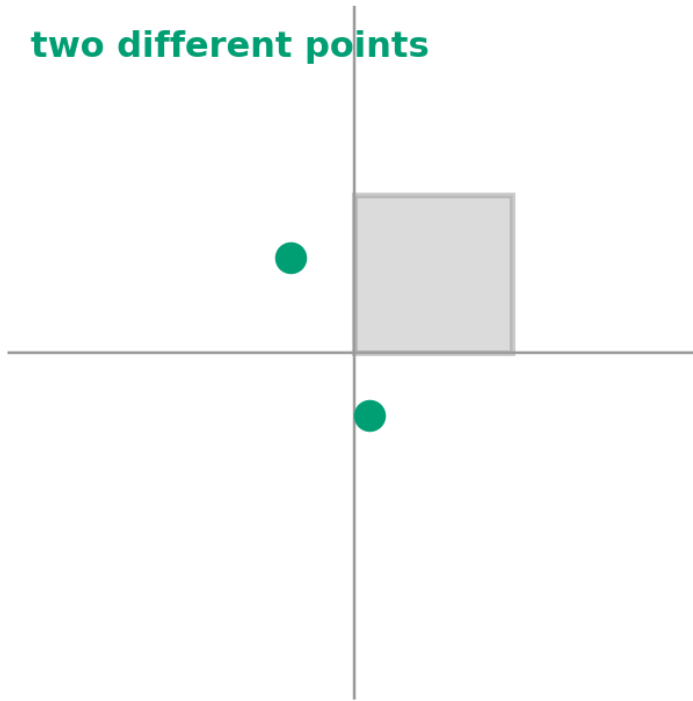


- Swap the order of the columns and the area stays 3, but the shape is mirrored.
- The sign of $\det$ says whether orientation is kept (+) or flipped (–): a left glove becomes a right glove.
- This is why `np.linalg.det([[1, 2], [3, 4]]) = -2.0` on the next slide.

det = 0: the shadow problem

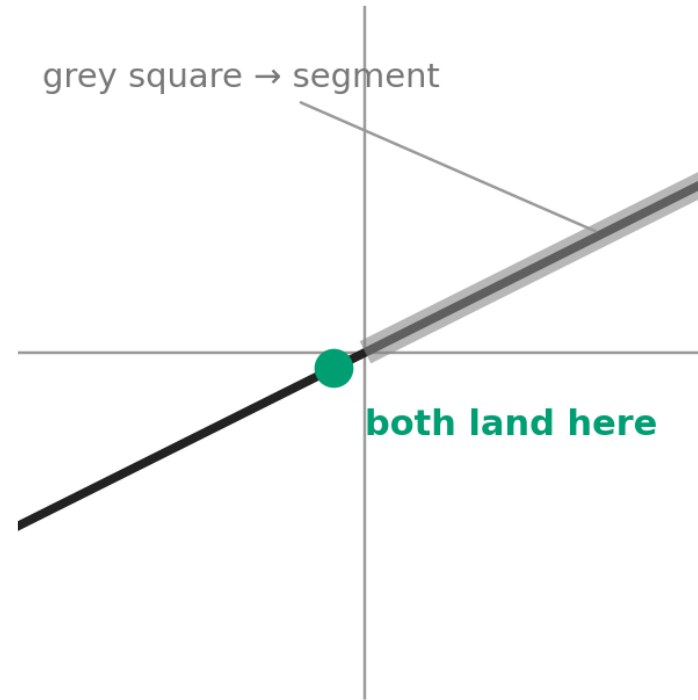
Before: the whole plane

two different points

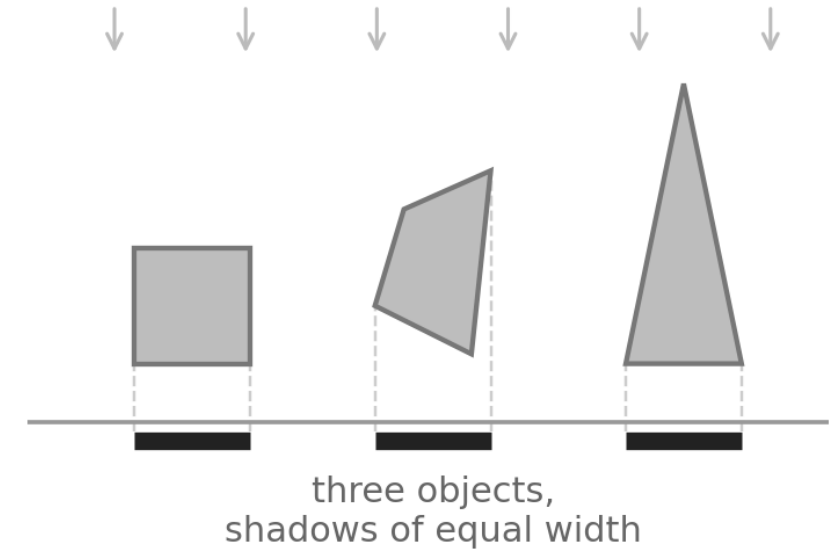


After (det = 0): one line

grey square → segment



Shadow: many objects, one shadow



det = 0 flattens the plane onto a line: different points land on the same spot.

No way back: from a shadow you cannot rebuild the object, so A^{-1} does not exist (compare with $x = A^{-1} b$).

numpy.linalg.det

`numpy.linalg.det(a)`

[\[source\]](#)

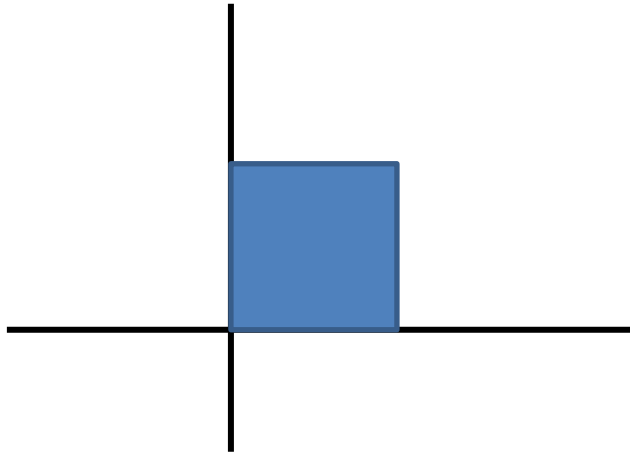
Compute the determinant of an array.

Parameters: *a* : (*..., M, M*) *array_like*
Input array to compute determinants for.

Returns: *det* : (*...*) *array_like*
Determinant of *a*.

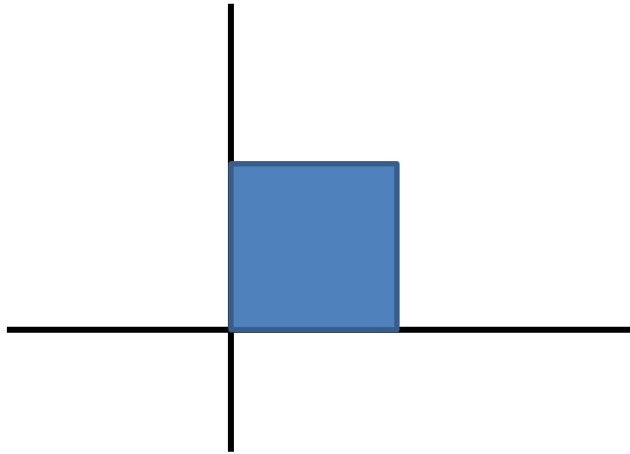
```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0
```

Eigenvalues and Eigenvectors

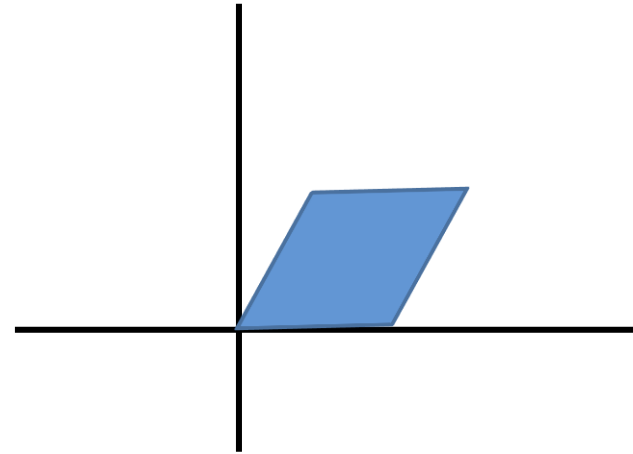


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

Eigenvalues and Eigenvectors

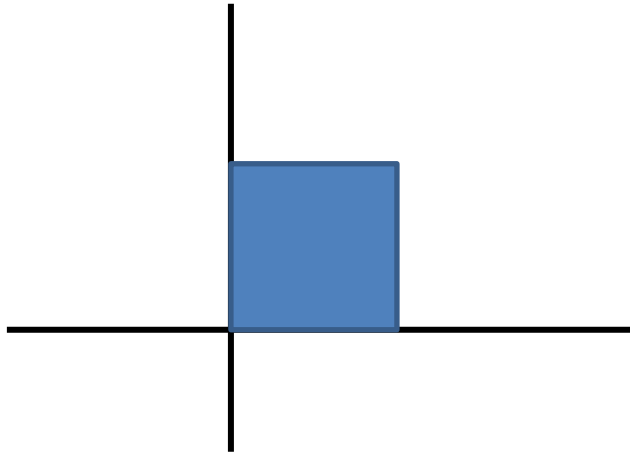


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

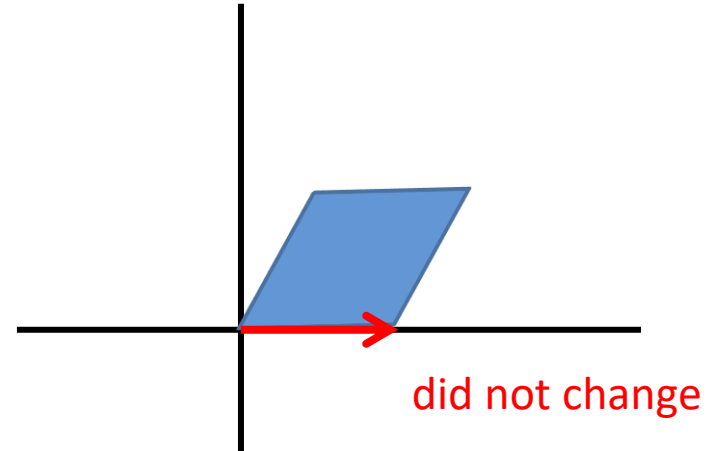


$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

Eigenvalues and Eigenvectors



$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$



$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \rightarrow \text{area } 1$$

Eigenvalues and Eigenvectors

$$A\mathbf{v} = \lambda\mathbf{v}$$

Eigenvalue
↓
λ
↑
Eigenvector
v

numpy.linalg.eig

Wartosci i wektory wlasne

`numpy.linalg.eig(a)`

[\[source\]](#)

Compute the eigenvalues and right eigenvectors of a square array.

```
>>> w, v = LA.eig(np.diag((1, 2, 3)))
>>> w; v
array([ 1.,  2.,  3.])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

>>>

numpy.linalg.matrix_rank

numpy.linalg.matrix_rank(*M*, *tol=None*)

[\[source\]](#)

Return matrix rank of array using SVD method

Rank of the array is the number of SVD singular values of the array that are greater than *tol*.

Parameters: *M* : {(M,), (M, N)} array_like

array of <=2 dimensions

tol : {None, float}, optional

threshold below which SVD values are considered zero. If *tol* is None, and *S* is an array with singular values for *M*, and *eps* is the epsilon value for datatype of *S*, then *tol* is set to `S.max() * max(M.shape) * eps`.

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 6 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Predict, then check

Predict the eigenvalues and eigenvectors, then check with `np.linalg.eig`:

- Rotation by 90° : $[[0, -1], [1, 0]]$
- Shear: $[[1, 1], [0, 1]]$
- Projection onto the line $y = x$: $[[0.5, 0.5], [0.5, 0.5]]$
- Reflection in the line $y = x$: $[[0, 1], [1, 0]]$

Hint: one answer is complex, one matrix has only one eigen-direction, and 0, 1 and -1 appear. Explain each from the geometry.

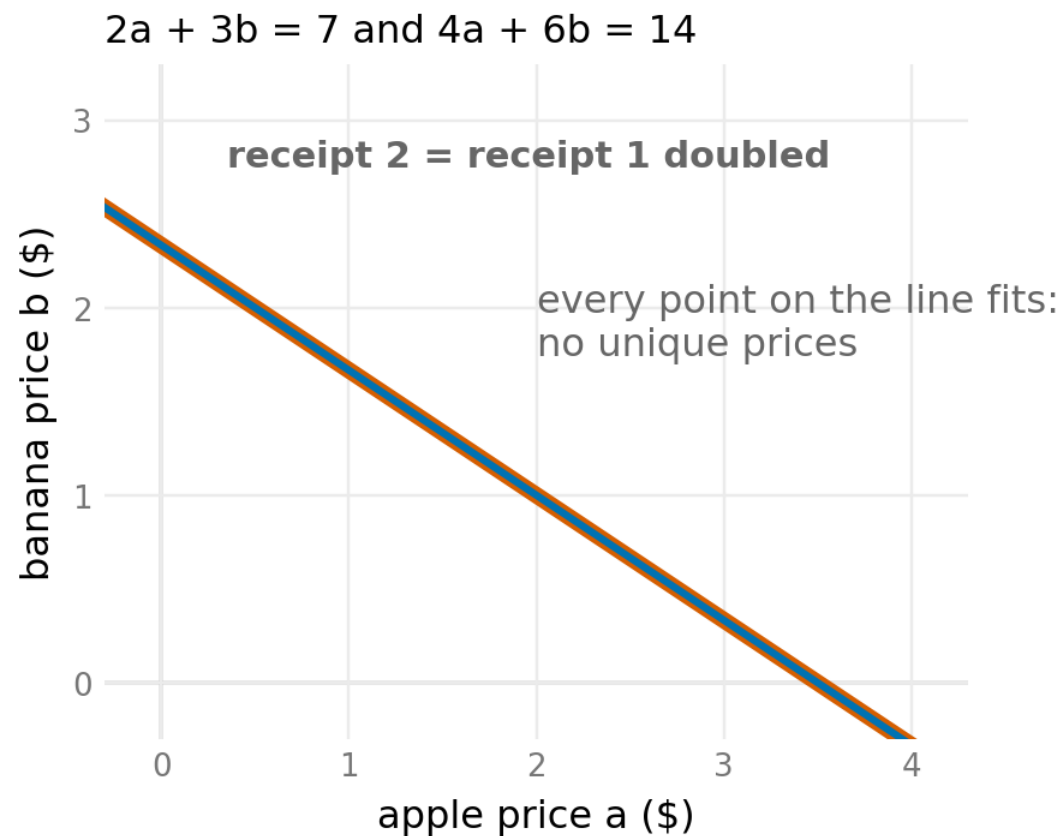
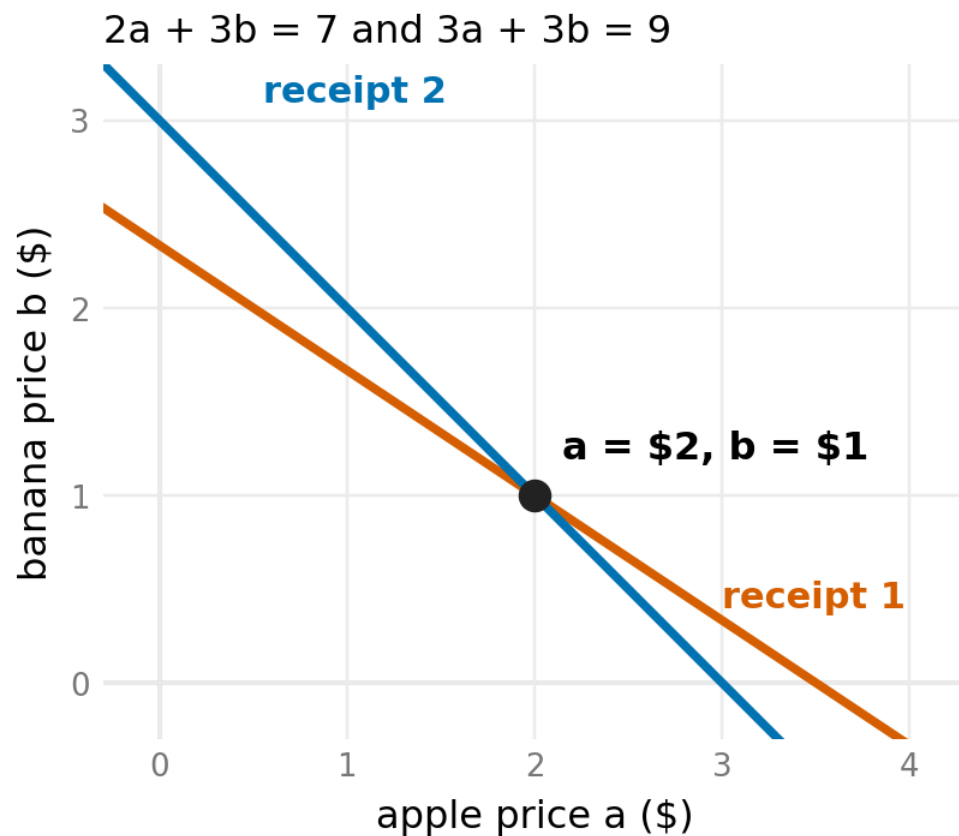
Matrix Rank Example

- If we know that
 - 2 apples and 3 bananas cost \$7
 - 3 apples and 3 bananas cost \$9
- Then we can figure out the extra apple must cost \$2, and so the bananas costs \$1 each. There are 2 variables and the rank is also 2.

Matrix Rank Example

- If we know that
 - 2 apples and 3 bananas cost \$7
 - 3 apples and 3 bananas cost \$9
- Then we can figure out the extra apple must cost \$2, and so the bananas costs \$1 each. There are 2 variables and the rank is also 2.
- But if we only know that
 - 2 apples and 3 bananas cost \$7
 - 4 apples and 6 bananas cost \$14
- We can't go any further because the second row of data is just twice the first and gives us no new information. There are 2 variables and the rank is only 1.

Matrix rank geometrically



Left: two different receipts, two lines, one crossing: apple \$2, banana \$1 (rank 2).

Right: the second receipt is the first one doubled; same line, no unique prices (rank 1). `np.linalg.matrix_rank` counts this.

Conditioning

- If a matrix has full rank there exists a solution.
- But there could be an approximate solution due to errors induced by floating point arithmetic.
- **Conditioning number** measures how well a given matrix A is conditioned

$$\text{cond}(A) \equiv \|A^{-1}\| \cdot \|A\|$$

numpy.linalg.cond¶

`numpy.linalg.` **cond** (*x*, *p=None*)

[\[source\]](#)

Compute the condition number of a matrix.

This function is capable of returning the condition number using one of seven different norms, depending on the value of *p* (see Parameters below).

Parameters: *x* : (*..., M, N*) array_like

The matrix whose condition number is sought.

p : {None, 1, -1, 2, -2, inf, -inf, 'fro'}, optional

Order of the norm:

<i>p</i>	norm for matrices
----------	-------------------

None	2-norm, computed directly using the <code>SVD</code>
------	--

'fro'	Frobenius norm
-------	----------------

inf	max(sum(abs(x), axis=1))
-----	--------------------------

-inf	min(sum(abs(x), axis=1))
------	--------------------------

1	max(sum(abs(x), axis=0))
---	--------------------------

-1	min(sum(abs(x), axis=0))
----	--------------------------

2	2-norm (largest sing. value)
---	------------------------------

-2	smallest singular value
----	-------------------------

inf means the numpy.inf object, and the Frobenius norm is the root-of-sum-of-squares norm.

Returns: *c* : {float, inf}

The condition number of the matrix. May be infinite.

Solving Linear Systems

- Direct methods
 - Transforms the equations into equivalent equations that have the same solution but are easier to solve
- Iterative methods
 - Start with an initial result near the solution and iteratively improve it
 - Usually, they are slower but have advantages if matrices are very big or sparse

Direct Methods

- Gauss elimination
- LU decomposition
- Gauss-Jordan elimination

scipy.linalg.lu

scipy.linalg.lu(*a*, *permute_l=False*, *overwrite_a=False*, *check_finite=True*)

[\[source\]](#)

Compute pivoted LU decomposition of a matrix.

The decomposition is:

$$A = P L U$$

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Parameters: *a* : (M, N) array_like

Array to decompose

permute_l : bool

Perform the multiplication P*L (Default: do not permute)

overwrite_a : bool

Whether to overwrite data in a (may improve performance)

check_finite : boolean, optional

Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns:

(If permute_l == False)

p : (M, M) ndarray

Permutation matrix

l : (M, K) ndarray

Lower triangular or trapezoidal matrix with unit diagonal. K = min(M, N)

u : (K, N) ndarray

Upper triangular or trapezoidal matrix

(If permute_l == True)

pl : (M, K) ndarray

Permuted L matrix. K = min(M, N)

u : (K, N) ndarray

Upper triangular or trapezoidal matrix

numpy.linalg.solve

`numpy.linalg.solve(a, b)`

[\[source\]](#)

Solve a linear matrix equation, or system of linear scalar equations.

Computes the “exact” solution, x , of the well-determined, i.e., full rank, linear matrix equation $ax = b$.

Examples

Solve the system of equations $3 * x_0 + x_1 = 9$ and $x_0 + 2 * x_1 = 8$:

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.]
```

```
>>>
```

<http://www.netlib.org/lapack/lug/node37.html#seccomp>

Iterative Methods

- Gauss-Seidel
- Jacobi
- Huge literature on this topic

Sparse linear algebra (scipy.sparse.linalg)

Solving linear problems

Direct methods for linear equation systems:

- | | |
|---|--|
| <code>spsolve(A, b[, permc_spec, use_umfpack])</code> | Solve the sparse linear system $Ax=b$, where b may be a vector or a matrix. |
| <code>factorized(A)</code> | Return a function for solving a sparse linear system, with A pre-factorized. |

Iterative methods for linear equation systems:

- | | |
|--|---|
| <code>bicg(A, b[, x0, tol, maxiter, xtype, M, ...])</code> | Use BiConjugate Gradient iteration to solve $Ax = b$ |
| <code>bicgstab(A, b[, x0, tol, maxiter, xtype, M, ...])</code> | Use BiConjugate Gradient STABILized iteration to solve $Ax = b$ |
| <code>cg(A, b[, x0, tol, maxiter, xtype, M, callback])</code> | Use Conjugate Gradient iteration to solve $Ax = b$ |
| <code>cgs(A, b[, x0, tol, maxiter, xtype, M, callback])</code> | Use Conjugate Gradient Squared iteration to solve $Ax = b$ |
| <code>gmres(A, b[, x0, tol, restart, maxiter, ...])</code> | Use Generalized Minimal RESidual iteration to solve $Ax = b$. |
| <code>lgmres(A, b[, x0, tol, maxiter, M, ...])</code> | Solve a matrix equation using the LGMRES algorithm. |
| <code>minres(A, b[, x0, shift, tol, maxiter, ...])</code> | Use MINimum RESidual iteration to solve $Ax=b$ |
| <code>qmr(A, b[, x0, tol, maxiter, xtype, M1, M2, ...])</code> | Use Quasi-Minimal Residual iteration to solve $Ax = b$ |

Iterative methods for least-squares problems:

- | | |
|--|---|
| <code>lsqr(A, b[, damp, atol, btol, conlim, ...])</code> | Find the least-squares solution to a large, sparse, linear system of equations. |
| <code>lsmr(A, b[, damp, atol, btol, conlim, ...])</code> | Iterative solver for least-squares problems. |

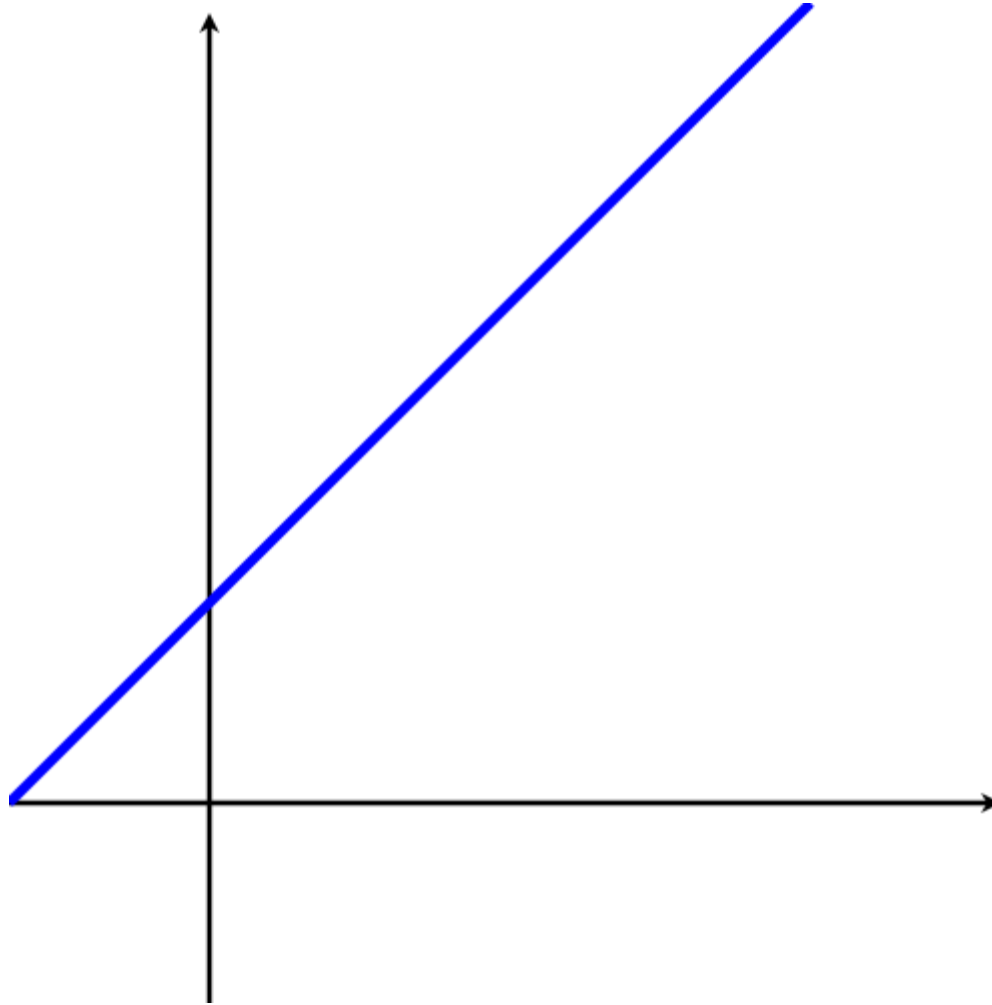
Solving Linear Systems of Equations

- $n = m$: can have an exact solution

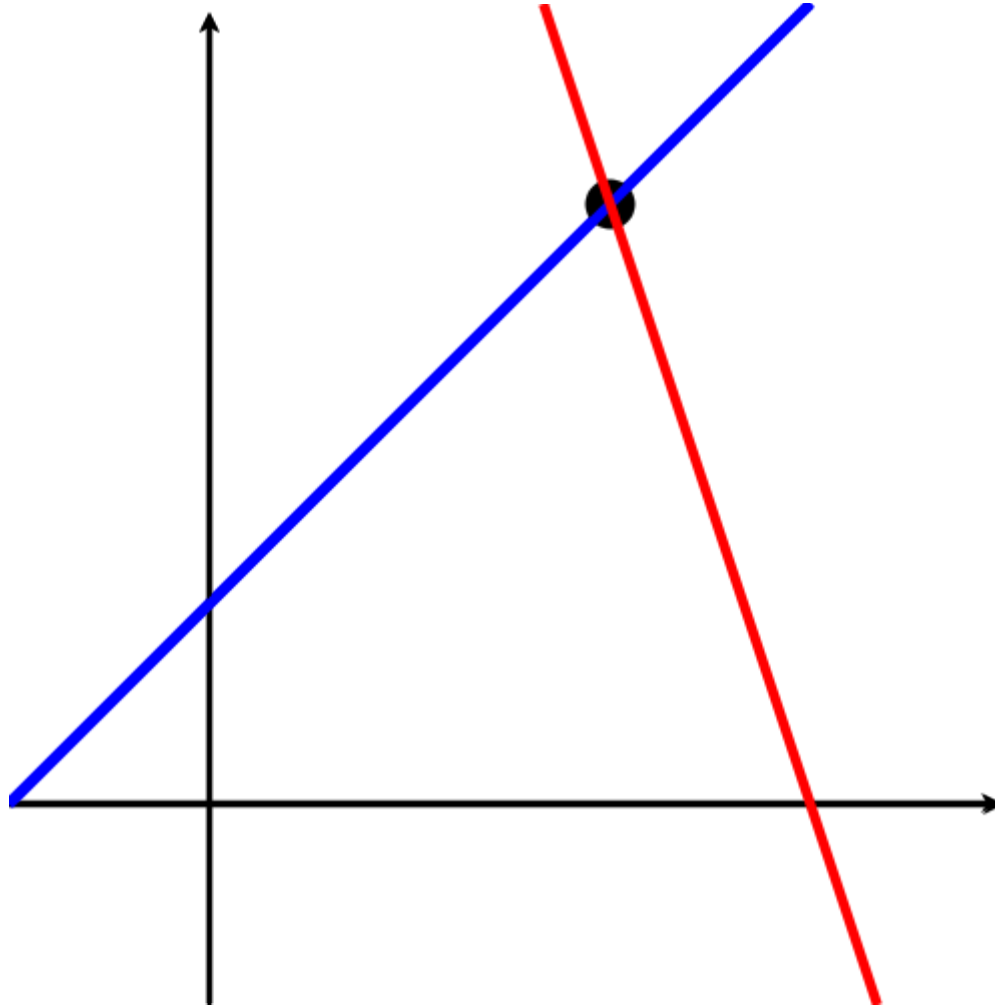
Solving Linear Systems of Equations

- $n = m$: can have an exact solution
- $n < m$: less equations than unknowns?
- $m > n$: more equations than unknowns?

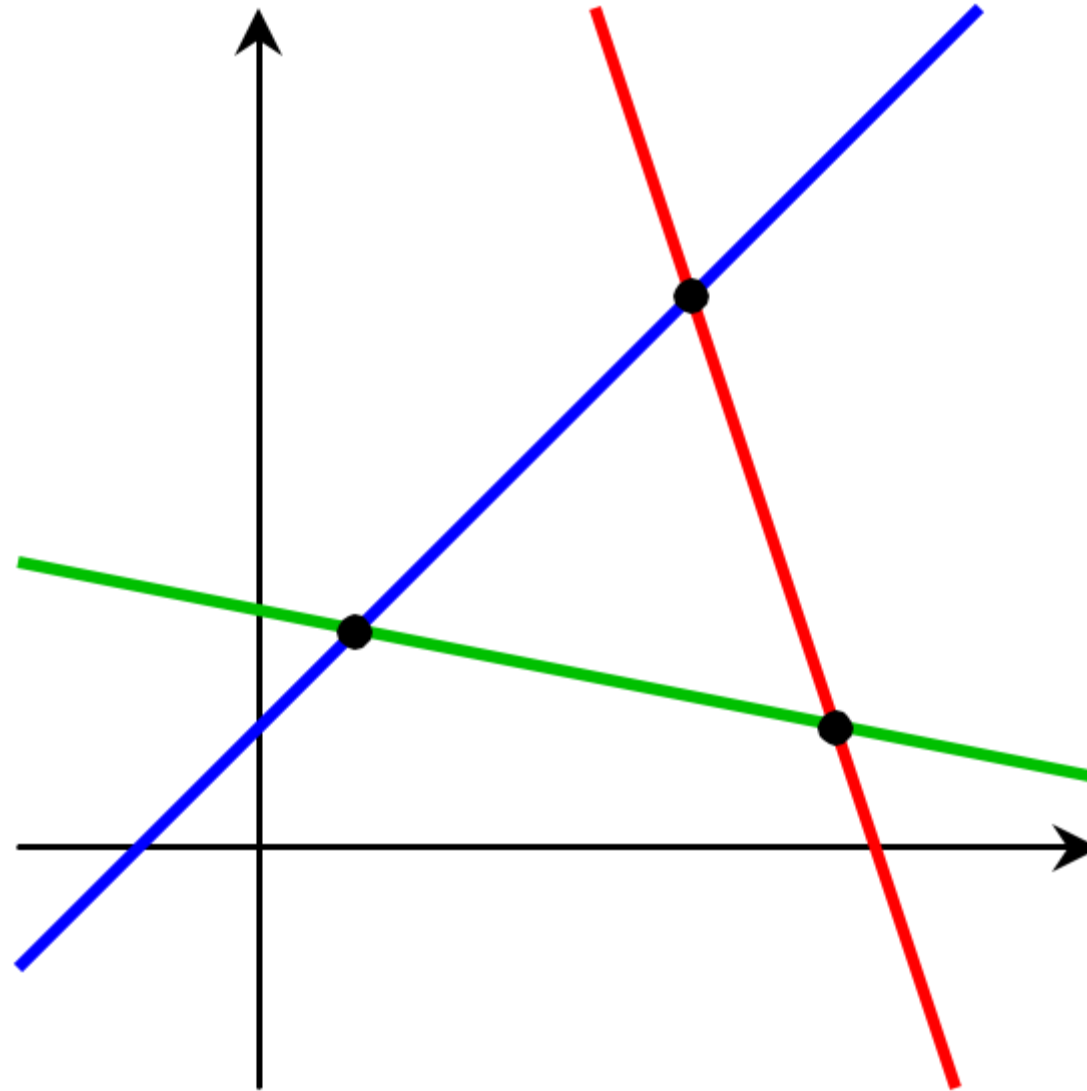
1 Equation



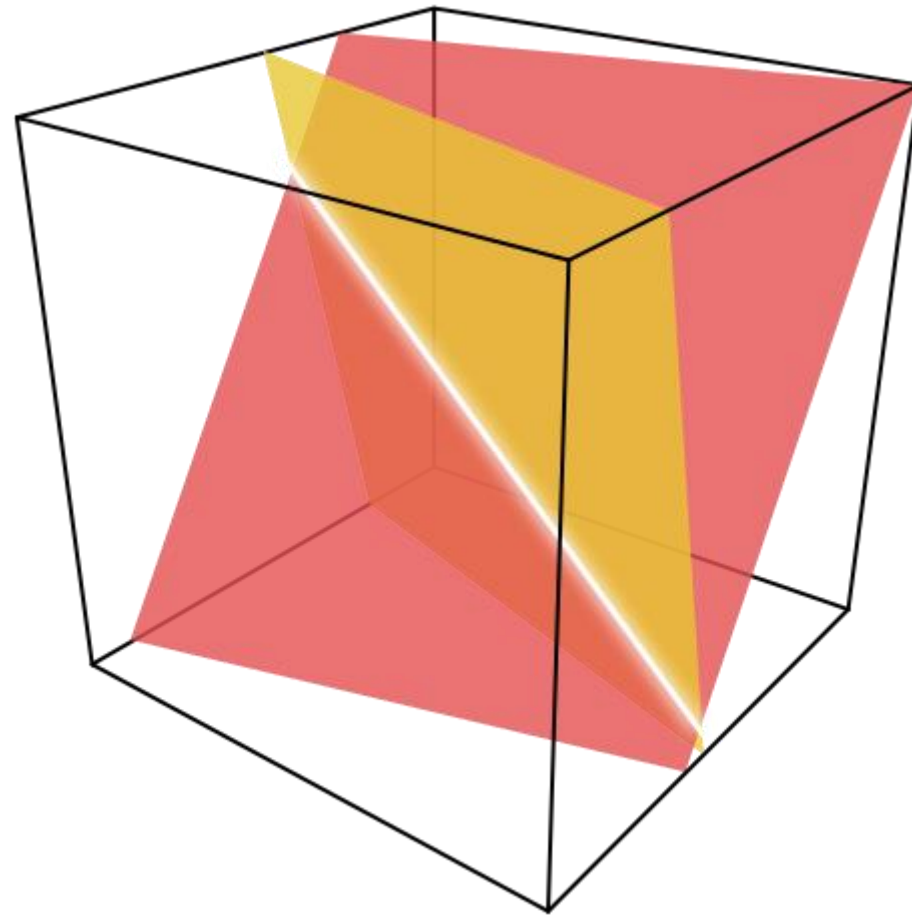
2 Equations



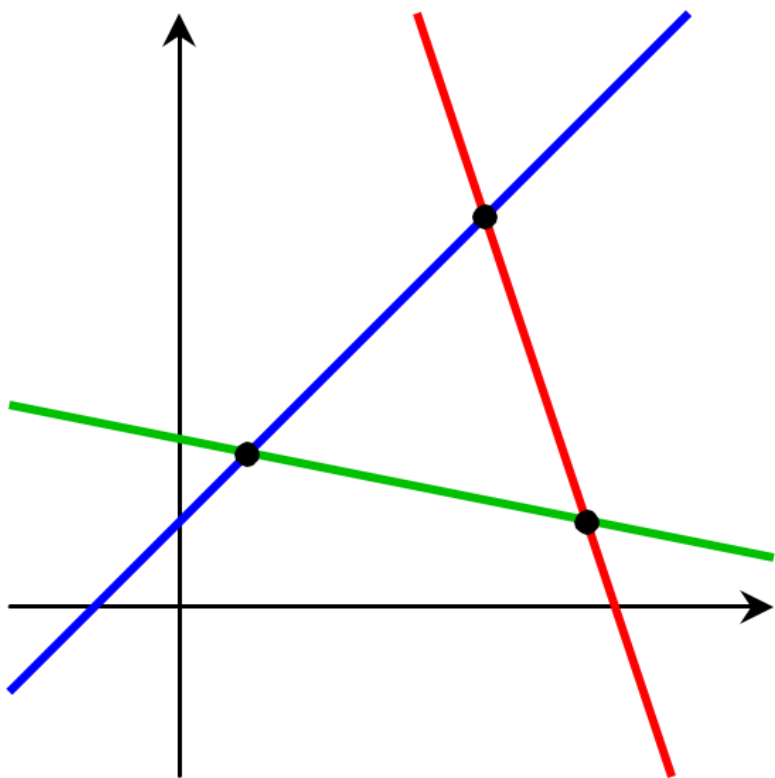
3 Equations



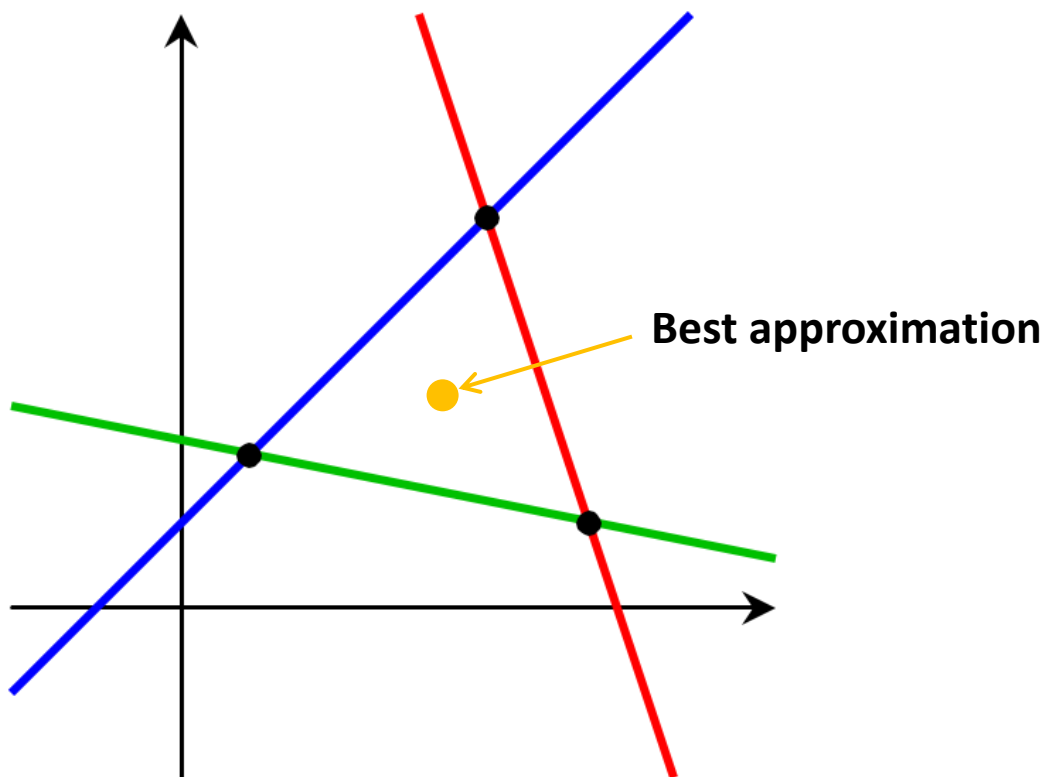
2 Equations 3 Unknowns



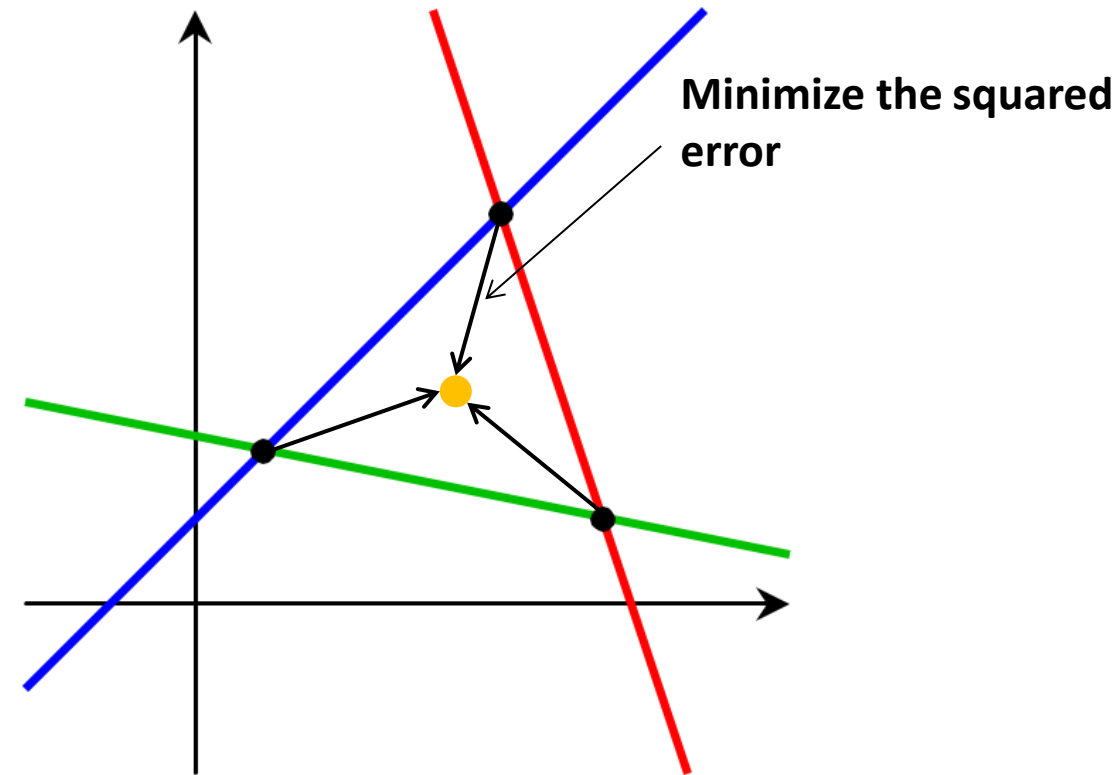
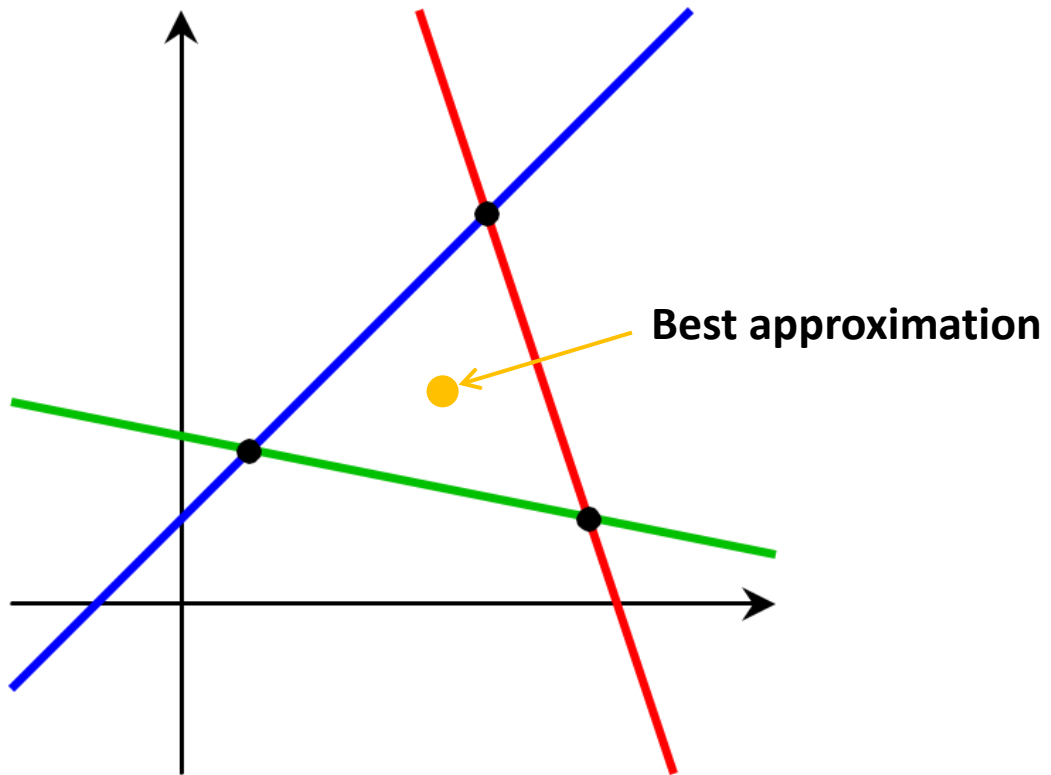
Approximation?



Approximation?



Least-squares



numpy.linalg.lstsq

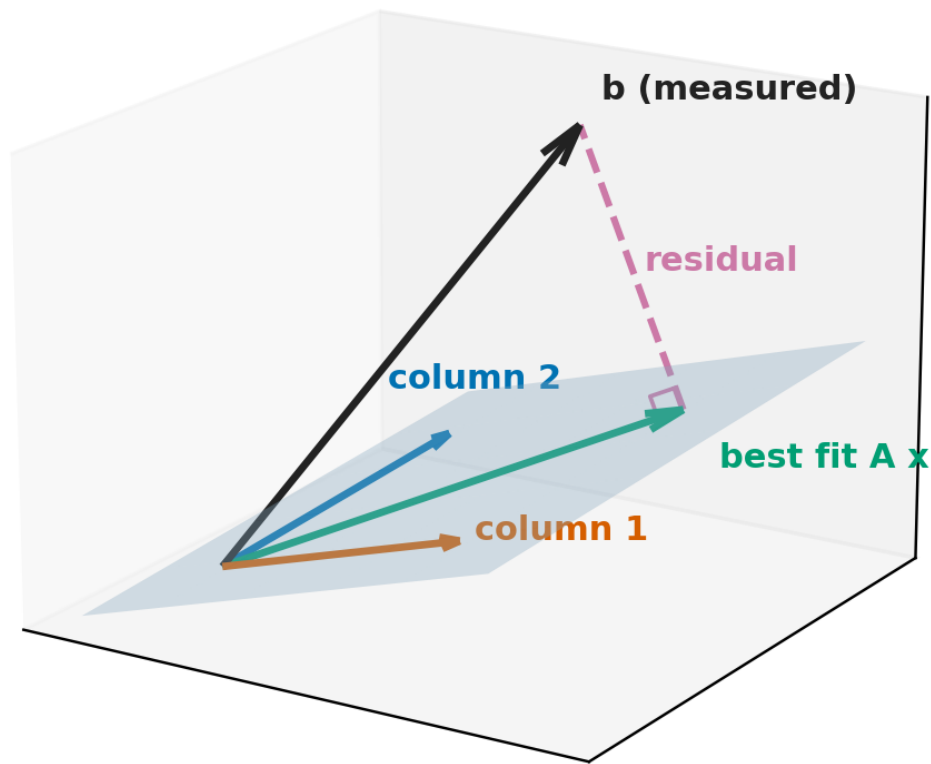
`numpy.linalg.lstsq(a, b, rcond=-1)`

[\[source\]](#)

Return the least-squares solution to a linear matrix equation.

Solves the equation $ax = b$ by computing a vector x that minimizes the Euclidean 2-norm $\|b - ax\|^2$. The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of a can be less than, equal to, or greater than its number of linearly independent columns). If a is square and of full rank, then x (but for round-off error) is the “exact” solution of the equation.

Least squares is a projection



- Ax can only reach the plane spanned by the columns; b lies off it.
- The best reachable point is the shadow of b on that plane: the residual is perpendicular to the columns, $A^T(Ax - b) = 0$.

```
A = np.array([[1, 1], [1, -1], [1, 0.2]])  
b = np.array([2, 0, 1.6])  
x, *_ = np.linalg.lstsq(A, b)  
print(x)  
print(A.T @ (A @ x - b))
```

```
[1.1316 1.0263]  
[0. 0.]
```

numpy.vstack

`numpy.vstack`(*tup*)

[\[source\]](#)

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

This function continues to be supported for backward compatibility, but you should prefer `np.concatenate` or `np.stack`.

The `np.stack` function was added in NumPy 1.10.

Parameters: *tup* : *sequence of ndarrays*

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns: *stacked* : *ndarray*

The array formed by stacking the given arrays.

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

Root finding: : $f(x) = 0$

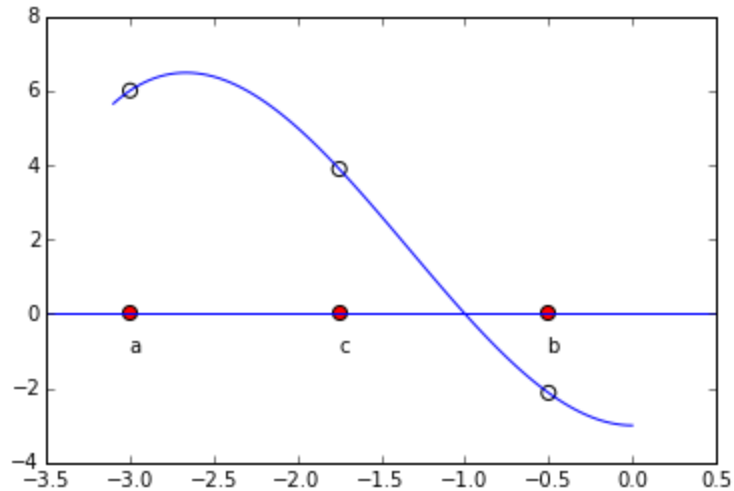
- *if $f(x) = ax^2 + bx + c$ then:*

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- For many functions an analytical solution cannot be found

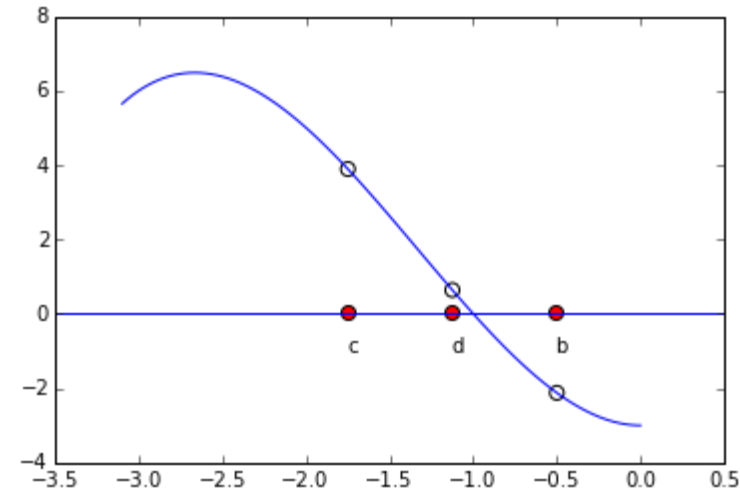
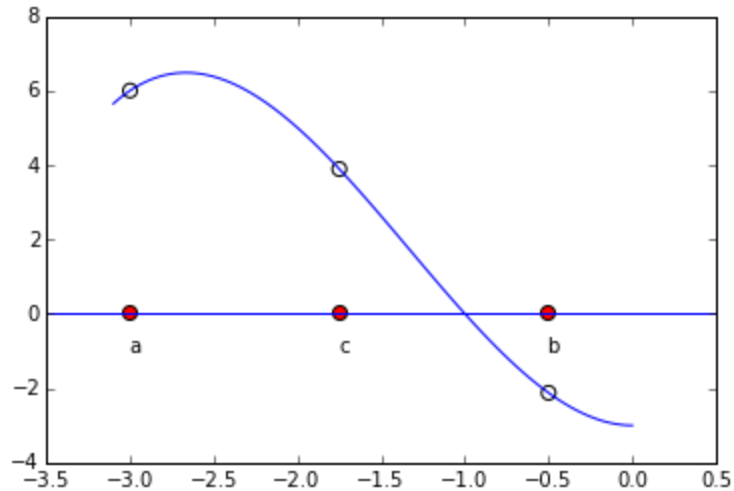
Bisection method

- Find points a and b such that $f(a)f(b) < 0$
- Calculate $c = (a+b)/2$, if $b-c < \varepsilon$: end
- If $f(a)f(c) < 0$: $b = c$, else: $a = c$



Bisection method

- Find points a and b such that $f(a)f(b) < 0$
- Calculate $c = (a+b)/2$, if $b-c < \varepsilon$: end
- If $f(a)f(c) < 0$: $b = c$, else: $a = c$



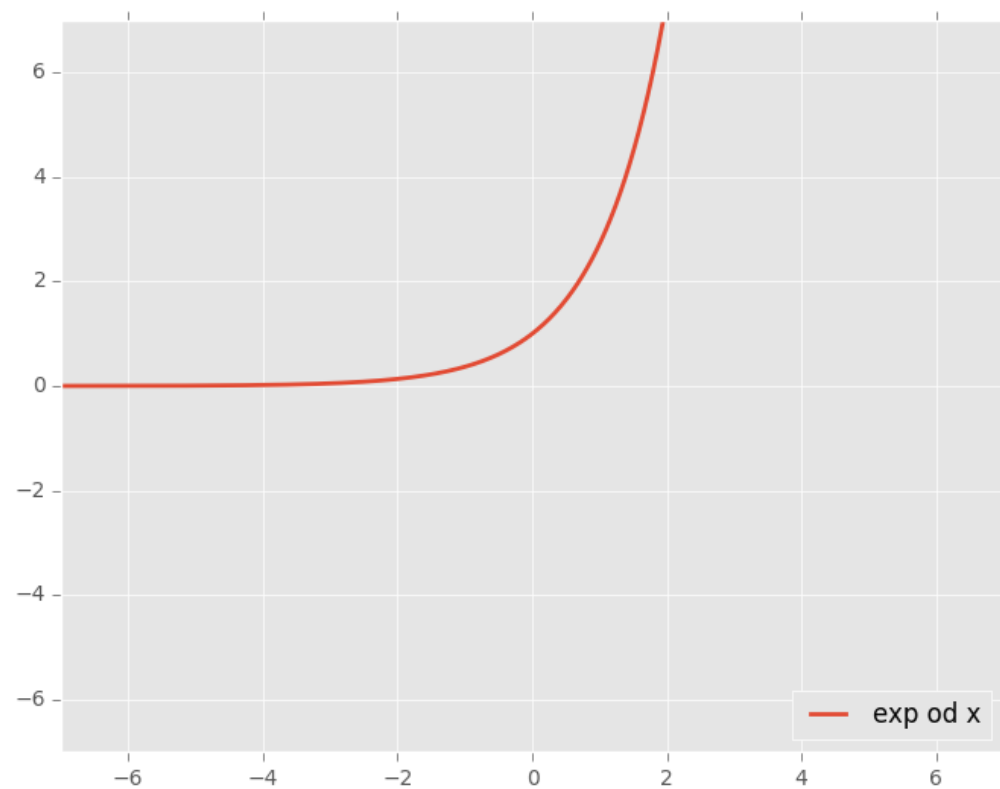
Taylor Series

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n$$

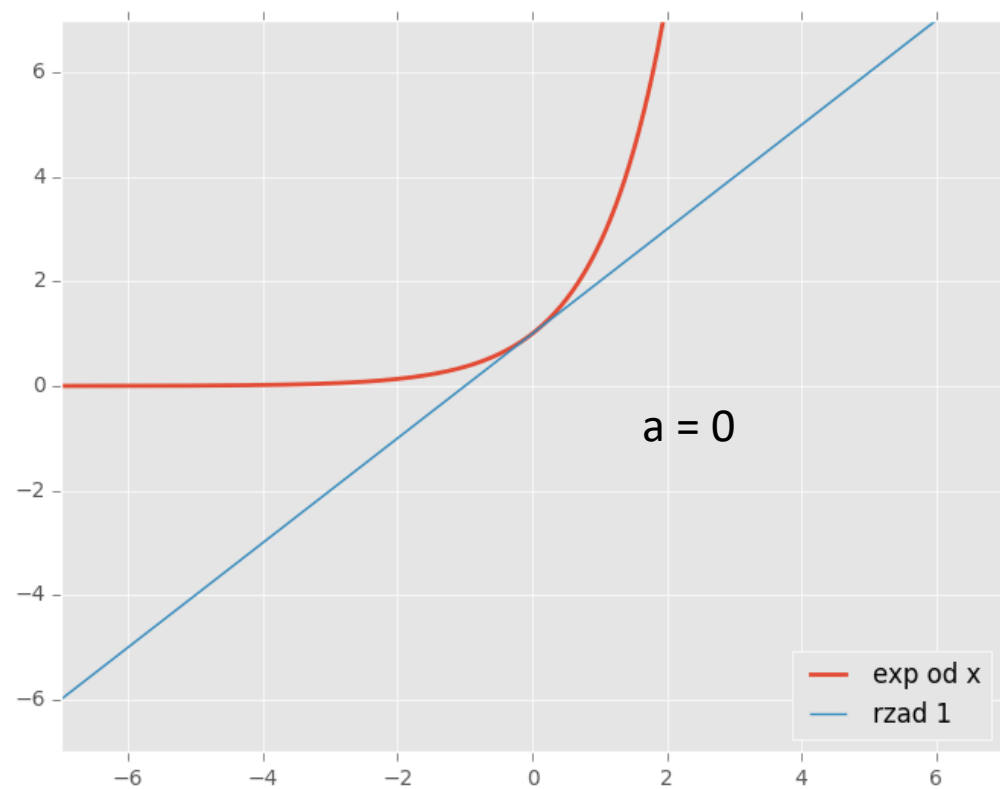
Taylor's theorem: there exists $b \in [x, a]$, such that

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(b)(x-a)^2$$

Taylor Series

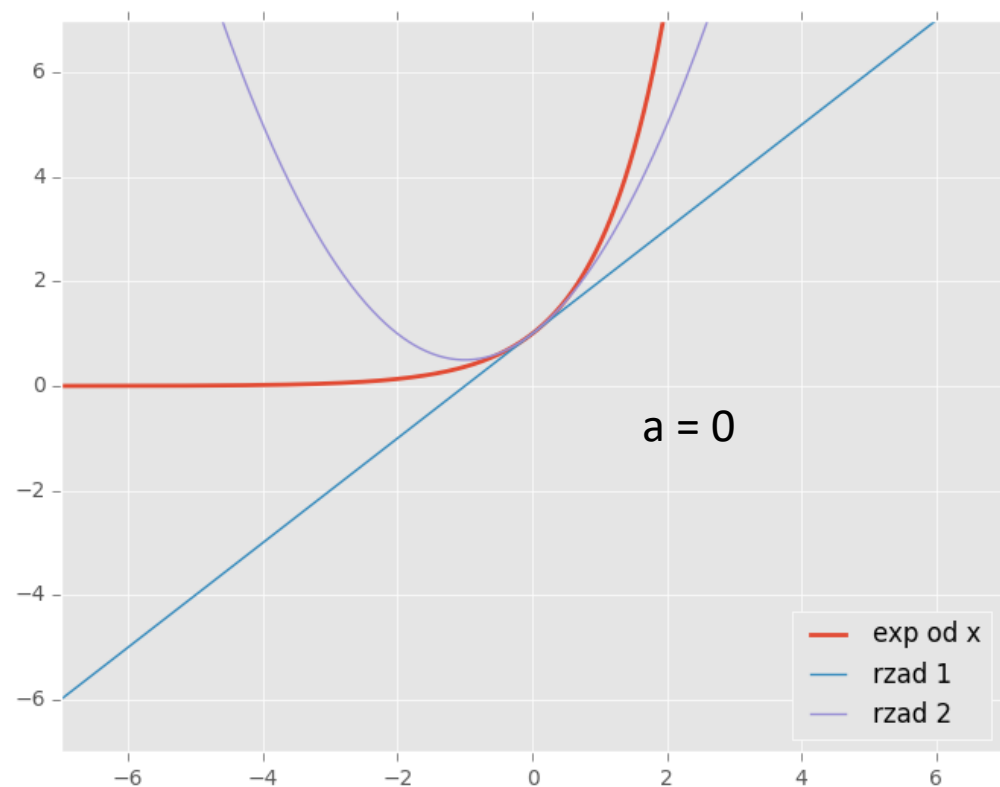


Taylor Series



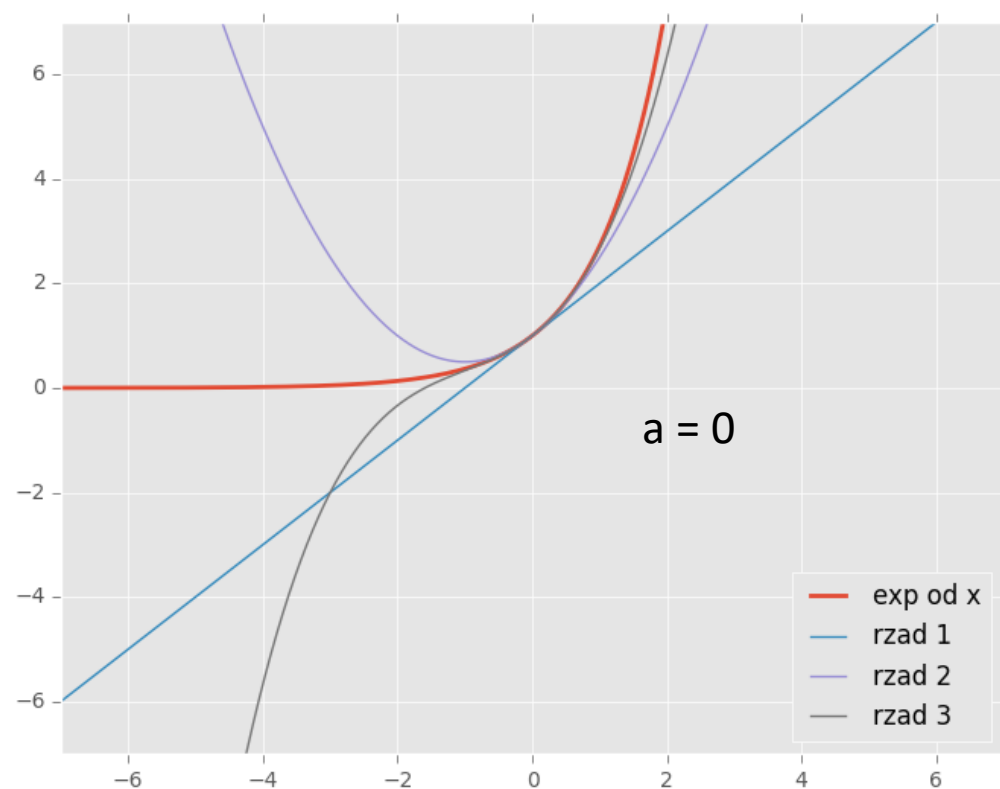
$$f(a) + \frac{f'(a)}{1!}(x - a)$$

Taylor Series



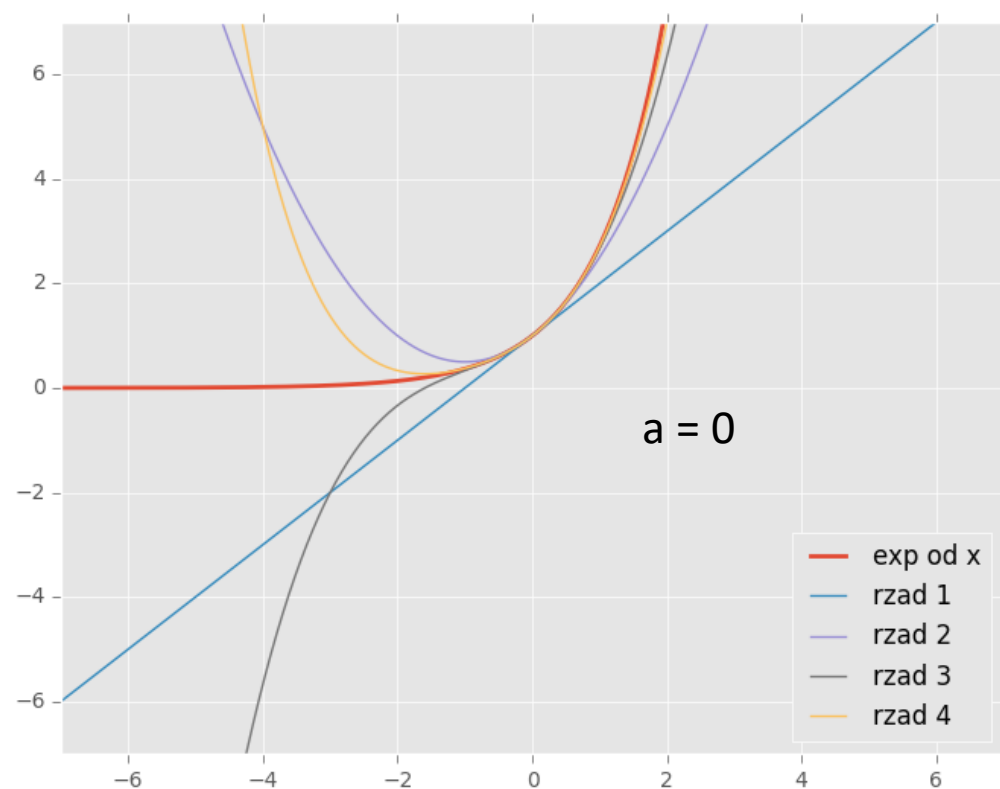
$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2$$

Taylor Series



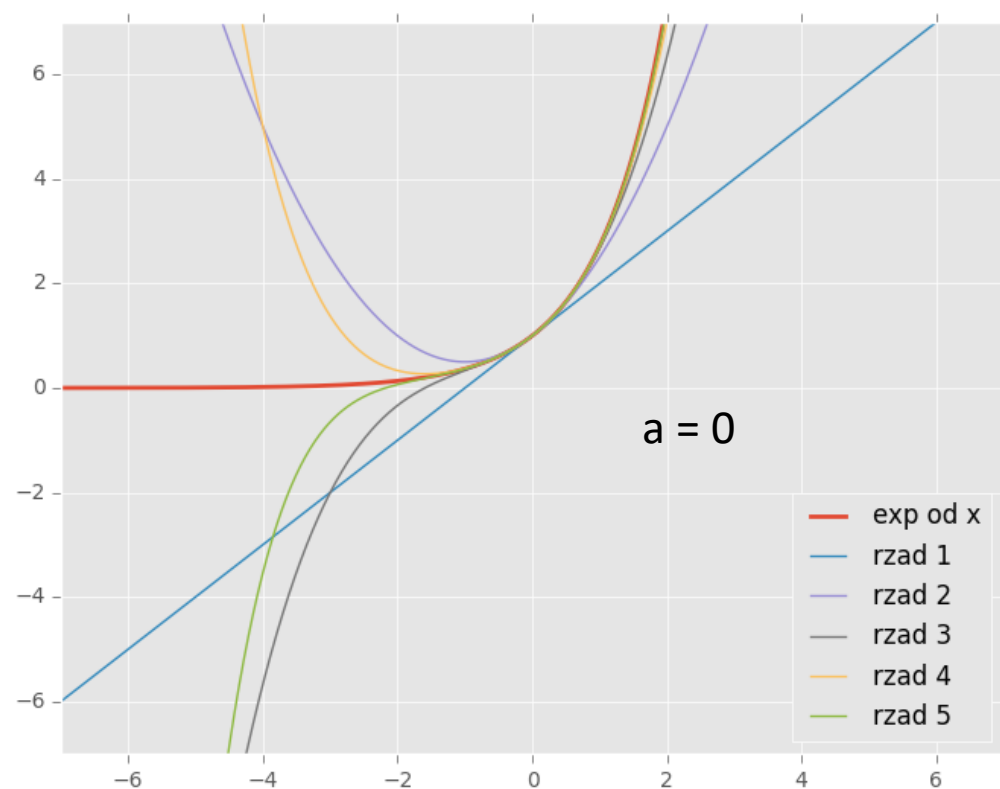
$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3$$

Taylor Series



$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots$$

Taylor Series



$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots$$

Taylor series $\rightarrow$ Newton's method

- $f(z) = f(x_0) + (z - x_0)f'(x_0) + \frac{1}{2}(z - x_0)^2 f''(c_0)$
- Because $f(z) = 0$ assuming that $|z - x_0|$ is small we approximate:

$$0 \approx f(x_0) + (z - x_0)f'(x_0)$$

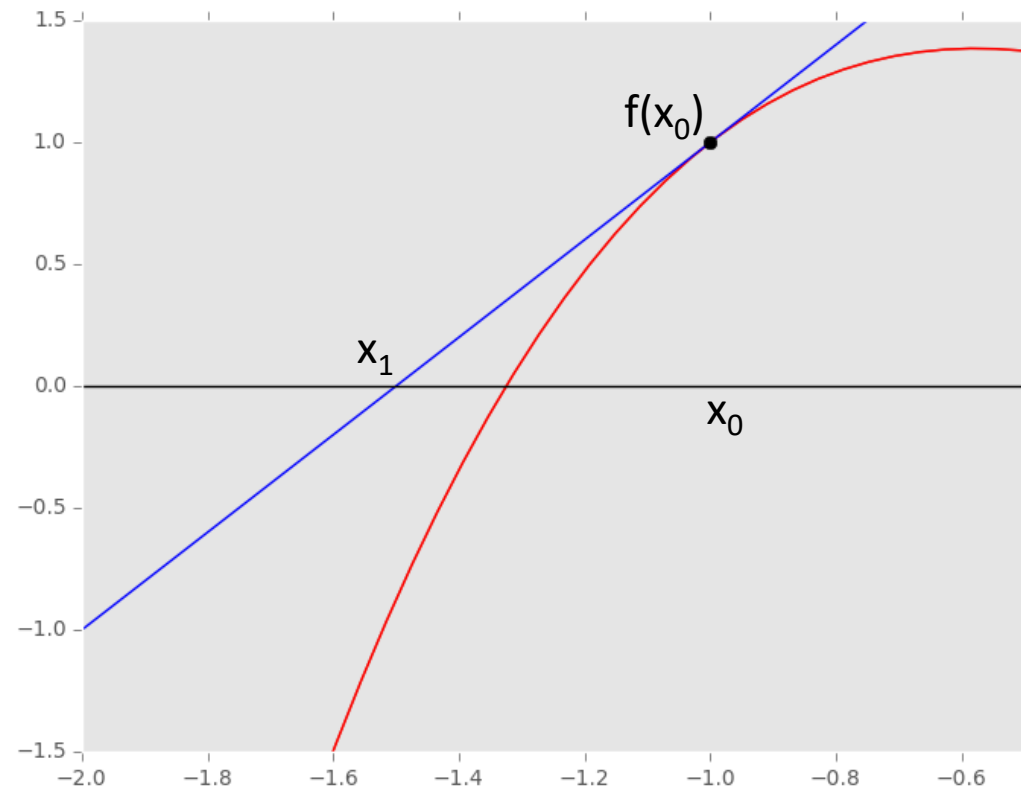
$$\text{which results in: } z \approx x_0 - \frac{f(x_0)}{f'(x_0)} \equiv x_1$$

Newton's method

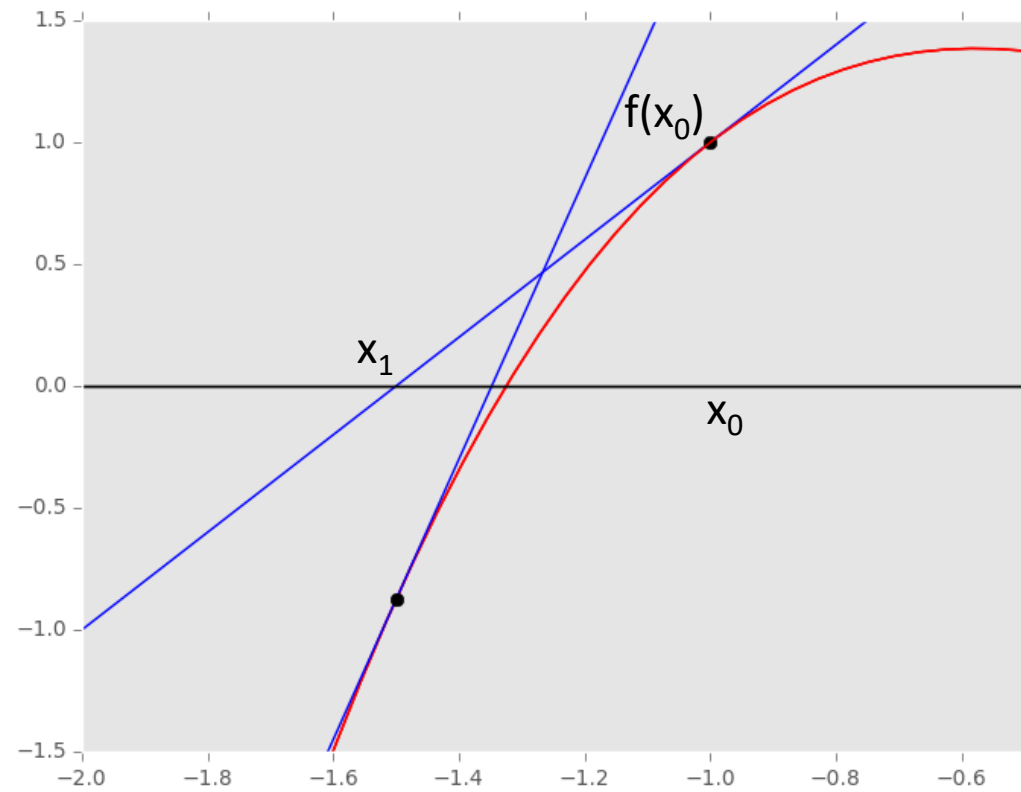
Iteration:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

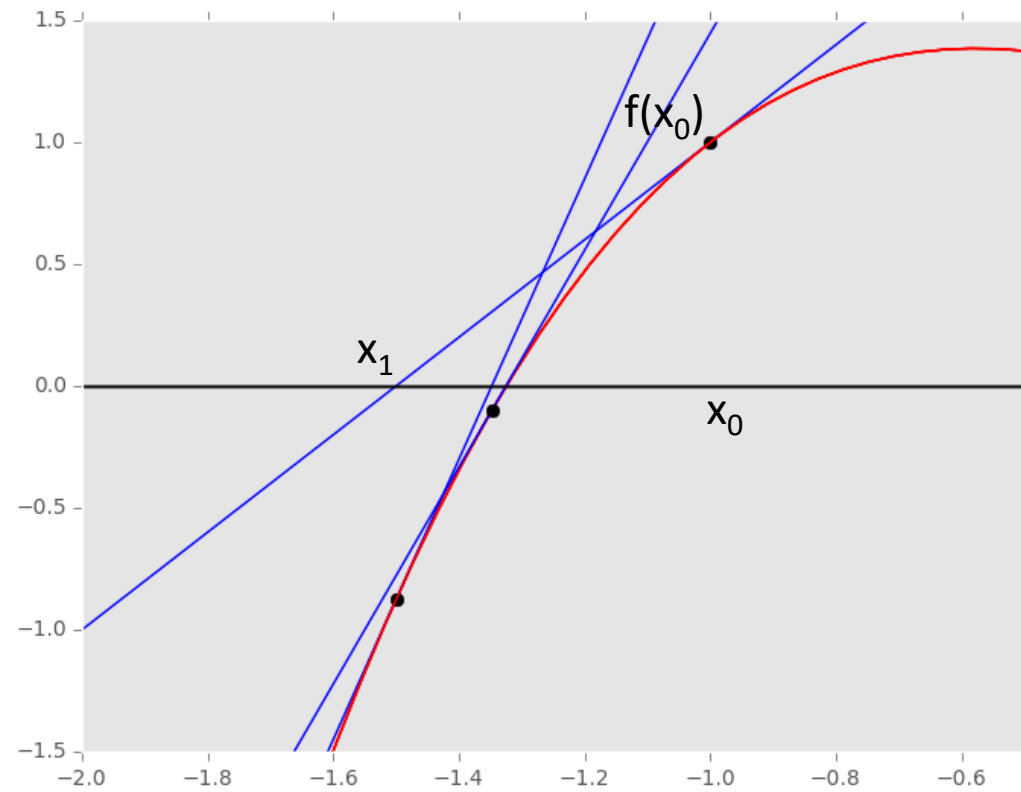
Newton's method



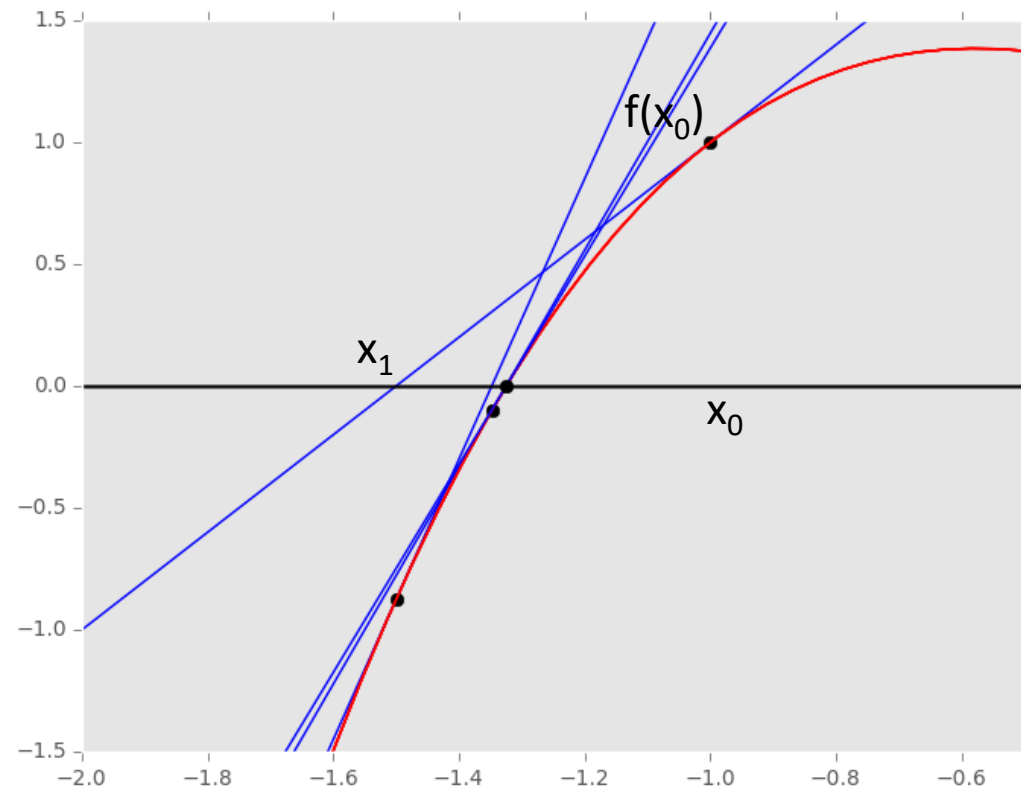
Newton's method



Newton's method



Newton's method



Scipy.optimize

Root finding

Scalar functions

<code>root_scalar</code> (f[, args, method, bracket, ...])	Find a root of a scalar function.
<code>brentq</code> (f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
<code>brenth</code> (f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method with hyperbolic extrapolation.
<code>ridder</code> (f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval using Ridder's method.
<code>bisect</code> (f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval using bisection.
<code>newton</code> (func, x0[, fprime, args, tol, ...])	Find a zero of a real or complex function using the Newton-Raphson (or secant or Halley's) method.
<code>toms748</code> (f, a, b[, args, k, xtol, rtol, ...])	Find a zero using TOMS Algorithm 748 method.
<code>RootResults</code> (root, iterations, ...)	Represents the root finding result.

Exercise

- Visualize the following system of equations as a 2D plot:

$$2x_1 + 3x_2 = 4$$

$$5x_1 + 4x_2 = 23$$

- What can you observe?
- Solve the system of equations using a linalg solver. You can verify the correctness of your solution with the graph you have drawn – is this always possible?

Exercise

- Add another equation $5x_2 = 18$ to the system of equations of the previous exercise and visualize it on the plot.
- Does a solution exist?
- Compute/approximate the solution by minimizing the squared error and draw it as a point on the plot.

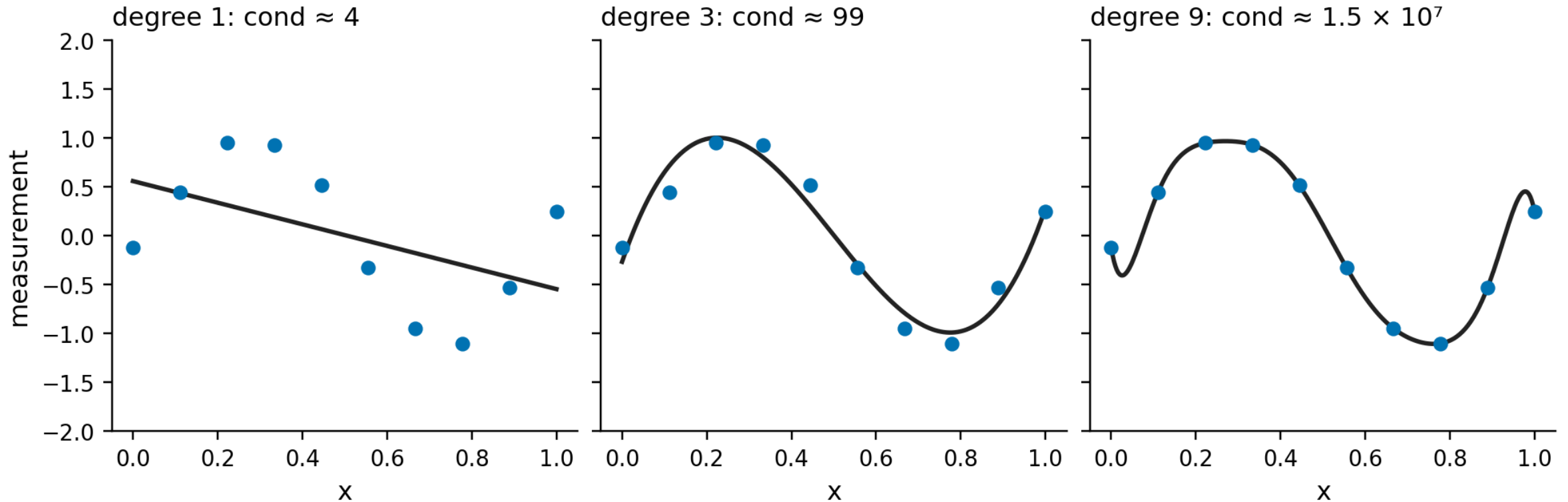
Exercise

- Create 10 random 2D data points (x, y) as a single ndarray in the range of 0 to 10. Add to each x and y value of a point a number between 0 and 10 in such a way that the first point is not increased but the last point is increased by 9 and the remaining number in an increasing sequence (use `arrange` from `numpy`). Scale each point by 3 units in x -direction. Then rotate each point around the origin by 45 degrees. Create the Vandermonde matrix A of polynomials of 1st order for the data sample x (just the first polynomial, the matrix will have 2 columns). Compute the least squares approximation of the linear system $Ax = y$. Visualize the data as points x, y and draw the solution of the approximation (the solution is a straight line).
- Read the documentation of the `numpy.polyfit` function. Redo the approximation using `polyfit` for 1st and 2nd orders.

Vandermonde Matrix

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

More unknowns, more fragile



Polynomial fitting is a least-squares problem with a Vandermonde matrix (`np.vander`). Higher degree fits the points better, but the condition number explodes and the curve follows the noise.

Exercise

- Plot the conditioning number as a function of p for the following linear system ($p \in [0.9, 1.1]$):

$$\begin{pmatrix} 1 & \sqrt{p} \\ 1 & \frac{1}{\sqrt{p}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Interpret the result of your numerical analysis.

Exercise

- Download the file [solve.npz](#) which contains arrays of the linear system $Ax=b$. Compute the vector x which solves this linear system or give the best approximation if there is none such vector.

Exercise: a boat on a river

A boat travels 24 km downstream in 2 hours.

The return trip upstream takes 3 hours.

How fast is the boat in still water, and how fast is the current?

What are the unknowns? Draw the graphs on a coordinate system where one axis describes the speed of the current and the other the speed of the boat.

Exercise: theatre tickets

A theatre sold 200 tickets and took 4600 zł.

Adults pay 30 zł, students 20 zł, seniors 15 zł.

There were twice as many students as seniors. How many tickets of each kind were sold?

Formulate this as a linear system of equations and solve it using `linalg.solve`

Exercise: two cities

Every year 10 % of the people in city A move to city B, and 5 % of the people in B move to A. Together they have 3 million people.

Eventually the populations stop changing. How many people live in each city then?

- What does 'stop changing' mean as an equation?
- Why is one of the balance equations not enough on its own?
- Solve, then check by simulating 50 years with a loop.

Exercise: the other direction

So far: story $\rightarrow$ equations. Now the reverse.

- Invent a story whose equations are $x + y = 10$ and $2x - y = 2$. Swap stories with your neighbour and translate theirs back.
- Invent a story with no solution, and one with infinitely many. What changes in the matrix?
- Invent a story that leads to a 3×3 system with a meaningful answer.

Project: from words to equations

Find a phenomenon, translate it into a linear system, solve it, defend it

The task

Find a real phenomenon (physics, engineering, biology, economics, computer science ...) that a published paper or an open-source project describes and that leads to a linear system $Ax = b$ (or $Ax = 0$, or a least-squares problem).

Translate the description from the source into unknowns and equations yourself, sentence by sentence.

Solve it with NumPy / SciPy and reproduce at least one number or figure from the source.

Defend the translation: why these unknowns, why these equations, how trustworthy is the answer?

Each student works on a different phenomenon (first come, first served). Using an LLM is allowed; you must be able to explain every equation without it.

What you hand in

#	Deliverable	Content
1	The source	Paper (DOI) or GitHub repository; quote the passage that describes the phenomenon (max. ½ page).
2	Translation table	'The source says' → 'equation', unknowns with units, assumptions you had to add.
3	Classification	Number of unknowns and independent equations, rank: one, none or many solutions? Square, under- or overdetermined?
4	Code	A short, readable script or notebook: build A and b (with loops for larger systems), solve with <code>np.linalg</code> / <code>scipy.linalg</code> .
5	Validation	Reproduce a number or figure from the source; plug the solution back into the story (residual, units, signs).
6	Sensitivity	<code>np.linalg.cond(A)</code> ; perturb b by a realistic measurement error; how much does x change?
7	One figure	Following the rules of lecture 1: one message, labels with units, purposeful colour.
8	Presentation	5 minutes.

Assessment

Criterion	Weight	Excellent means
Translation	20 %	Every equation traced to a sentence or a physical/biological/chemical law; unknowns and units explicit; hidden assumptions named.
Correctness and validation	20 %	Reproduces the source; residual and plausibility checks; classification (rank) correct.
Judgement	10 %	Conditioning and sensitivity discussed; limits of the linear model stated (what would make it nonlinear?).
Code	10 %	Readable, vectorised where sensible, uses the right routine (solve / lstsq / null_space / sparse).
Communication	40 %	Clear figure and presentation; answers the 'explain this equation' question.

Suggested timing: 2 weeks.

How to find a phenomenon

- Look for words that hide linear equations: balance, conservation, equilibrium, steady state, mixture, network, average of neighbors, superposition, calibration, best fit.
- Search Google Scholar or GitHub for these words together with your interest: 'steady state' + 'enzyme', 'nodal analysis' + 'circuit', 'least squares' + 'calibration' + 'sensor'.
- Good sign: the source contains a matrix, a table of coefficients, or a sentence like 'this leads to a system of equations'.
- Keep it small enough to understand completely: 3–20 unknowns, or a larger system you build with a loop from one rule.

Example topics (1/3): physics, engineering

Phenomenon	Linear system	Reference
Electric circuits (nodal analysis)	Kirchhoff's laws → square system	Ho, Ruehli, Brennan (1975), IEEE Trans. Circuits Syst. 22(6):504–509; github.com/PySpice-org/PySpice
Truss bridge statics	Force balance at each joint	github.com/ritchie46/anaStruct
Steady heat in a plate / room	Average of neighbours (discrete Laplace)	LeVeque (2007), Finite Difference Methods for ODEs and PDEs, SIAM
GPS positioning	Differences of distance equations	Bancroft (1985), IEEE Trans. Aerosp. Electron. Syst. AES-21(1):56–59
Spectral unmixing of materials	Mixture of known spectra (least squares)	Keshava, Mustard (2002), IEEE Signal Process. Mag. 19(1):44–57
CT reconstruction	Ray sums through pixels (ART / least squares)	Gordon, Bender, Herman (1970), J. Theor. Biol. 29:471–481; github.com/astra-toolbox/astra-toolbox

Example topics (2/3): biology, economy

Phenomenon	Linear system	Reference
Metabolic flux balance	$S v = 0$ (steady state of a reaction network)	Orth, Thiele, Palsson (2010), Nat. Biotechnol. 28:245–248; github.com/opencobra/cobrapy
Balancing chemical equations	Atom conservation $\rightarrow$ null space	Blakley (1982), J. Chem. Educ. 59(9):728
Age-structured population	Leslie matrix; stable age distribution	Leslie (1945), Biometrika 33(3):183–212
Economy: input–output model	$(I - A) x = d$	Leontief (1936), Rev. Econ. Stat. 18(3):105–125
Web page ranking	$(I - \alpha M) x = (1 - \alpha)/n \cdot 1$	Brin, Page (1998), Comput. Netw. ISDN Syst. 30:107–117
Cheapest healthy diet	Nutrient constraints (linear programme)	Stigler (1945), J. Farm Econ. 27(2):303–314

Example topics (3/3): computing, graphics

Phenomenon	Linear system	Reference
Seamless image cloning	Discrete Poisson equation per pixel	Pérez, Gangnet, Blake (2003), ACM Trans. Graph. 22(3):313–318
Mesh deformation	Laplacian coordinates (least squares)	Sorkine et al. (2004), Laplacian surface editing, Proc. SGP
Shape from shading	Photometric stereo: normals from 3+ lights	Woodham (1980), Opt. Eng. 19(1):139–144
Camera calibration	Homography / DLT from point pairs	Zhang (2000), IEEE TPAMI 22(11):1330–1334; github.com/opencv/opencv
Random walks and resistor networks	Graph Laplacian $L v = i$	Doyle, Snell (1984), Random Walks and Electric Networks, MAA
'Lights Out' puzzle	Linear system over $GF(2)$	Anderson, Feil (1998), Math. Mag. 71(4):300–303