PFE 4

Dr W Palubicki

Topics

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

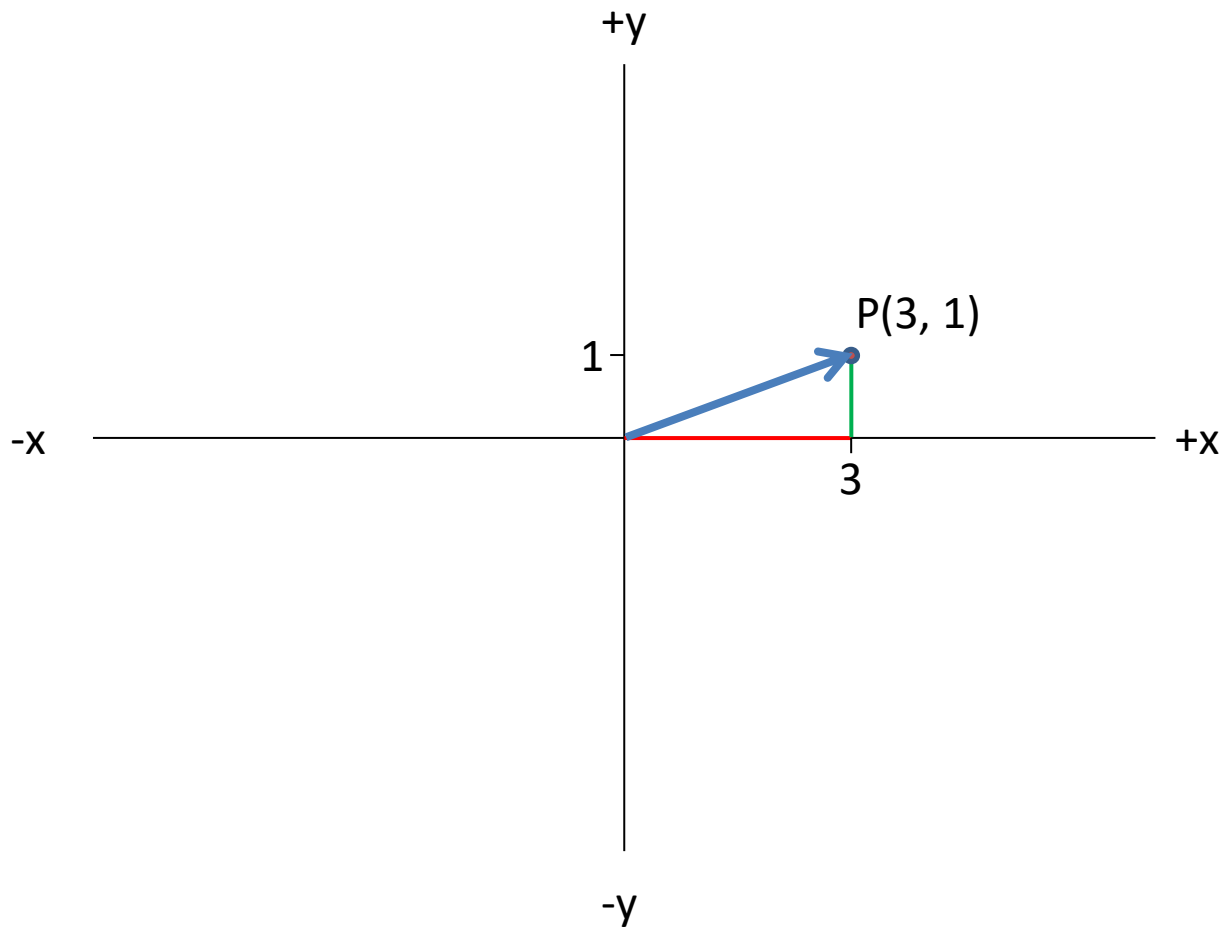
Linear algebra (numpy.linalg)

Matrix and vector products

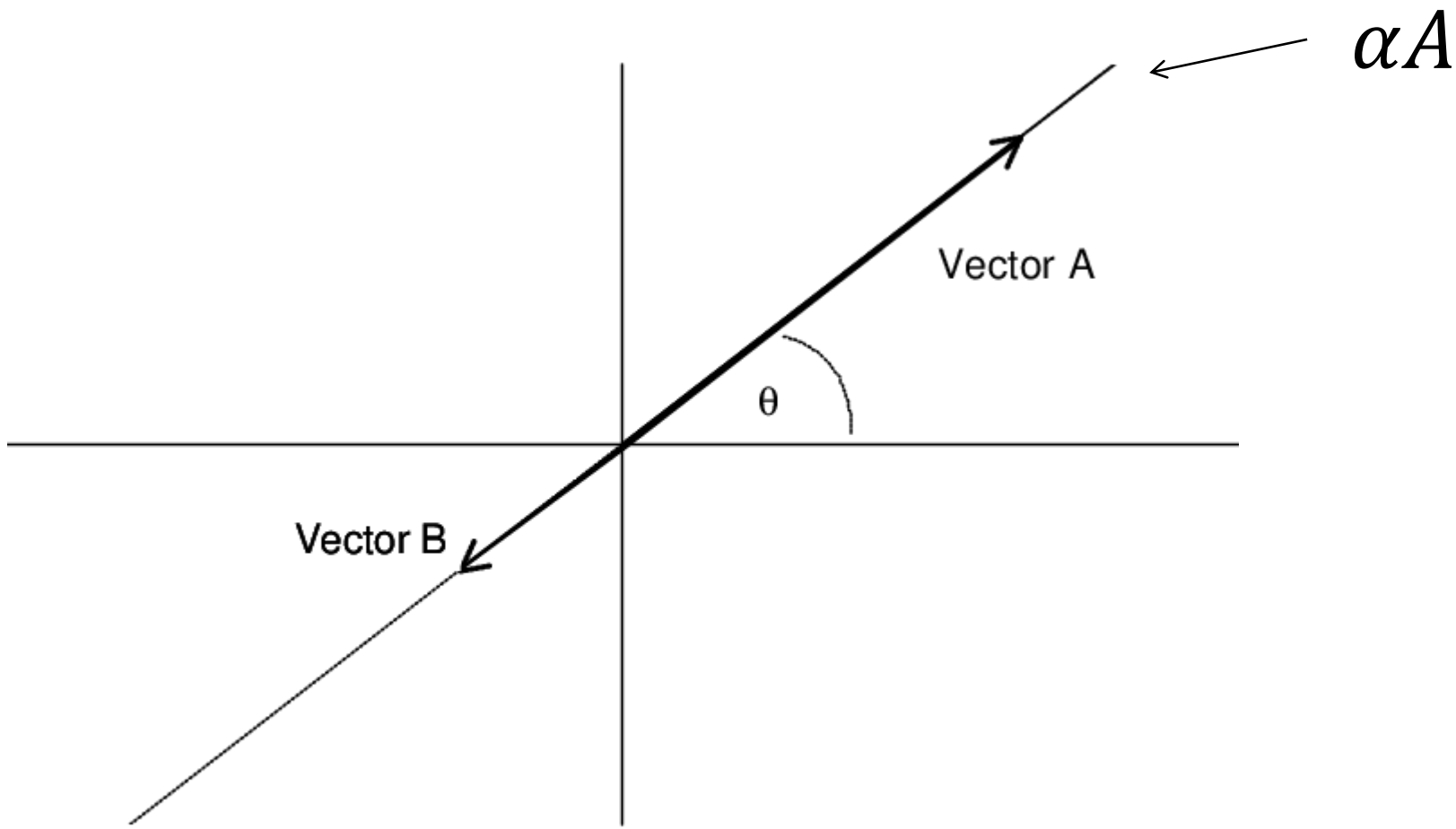
dot(a, b[, out])	Dot product of two arrays.
linalg.multi_dot(arrays)	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
vdot(a, b)	Return the dot product of two vectors.
inner(a, b)	Inner product of two arrays.
outer(a,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

Vectors

- Physics
- Computer science
- Mathematics



Vector span

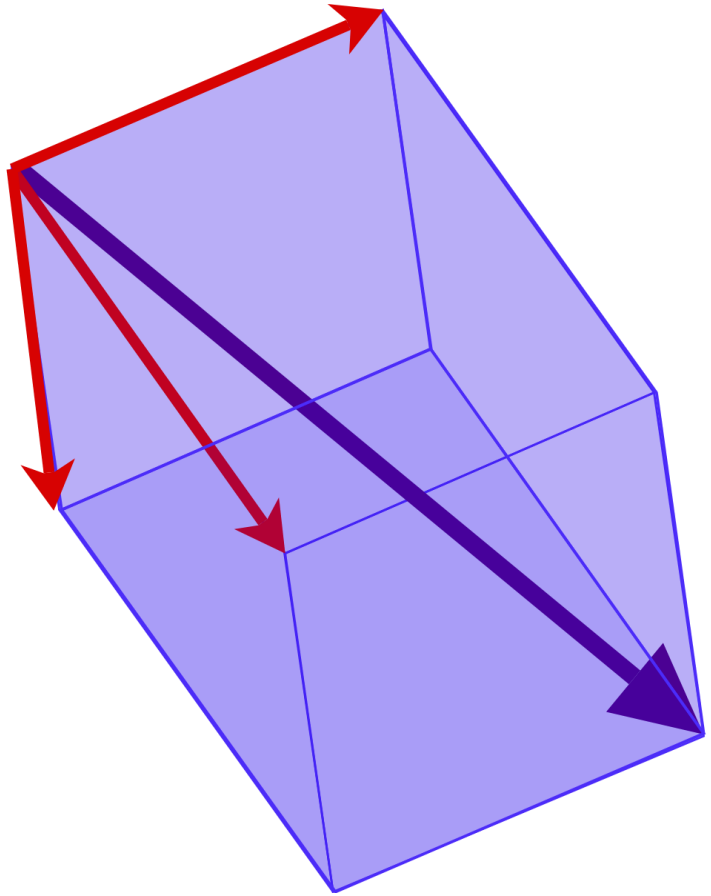


Linear Combination of Vectors

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

Linear Combination of Vectors

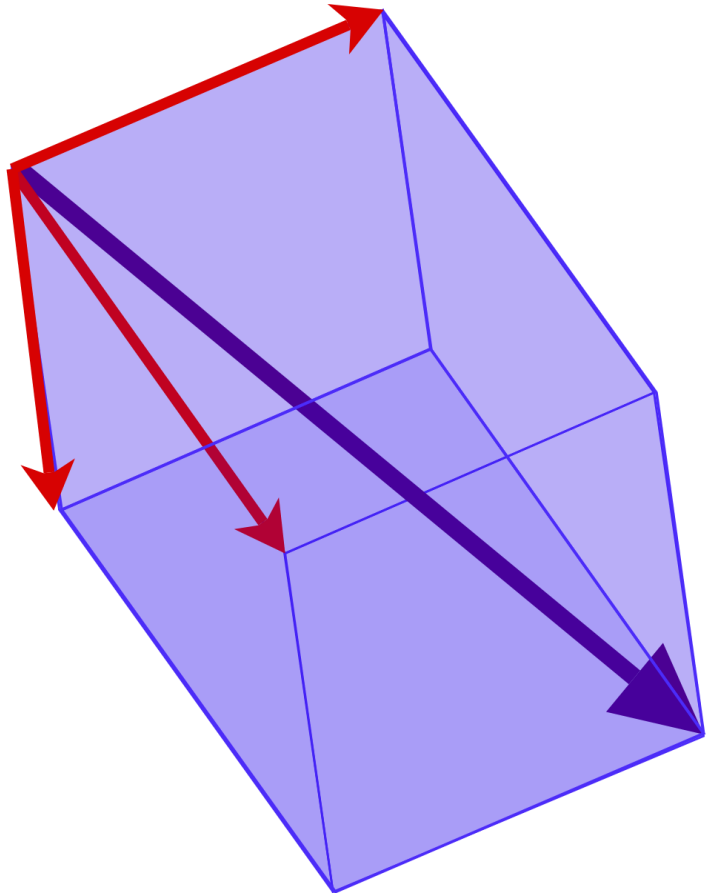
$\alpha A + \beta B + \gamma C$



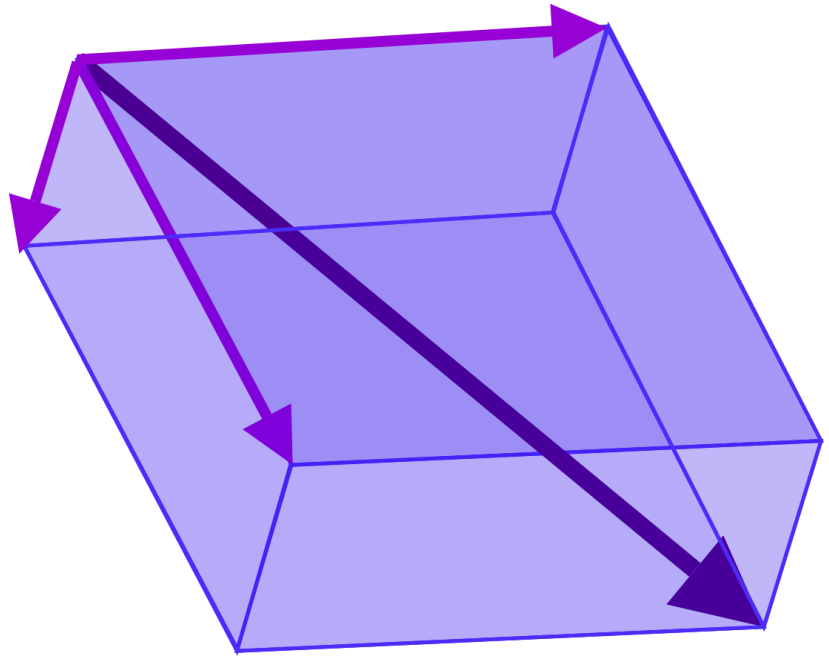
(α, β, γ)

Linear Combination of Vectors

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



(α, β, γ)



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

Linear Equation

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

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)

  $d = 0.2t$

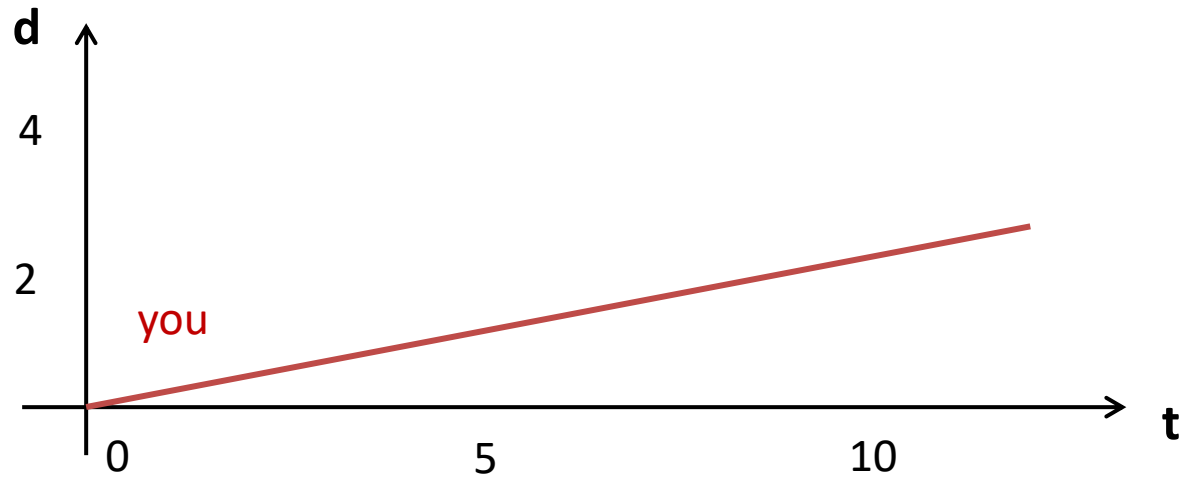
  $d = 0.5(t-6)$

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

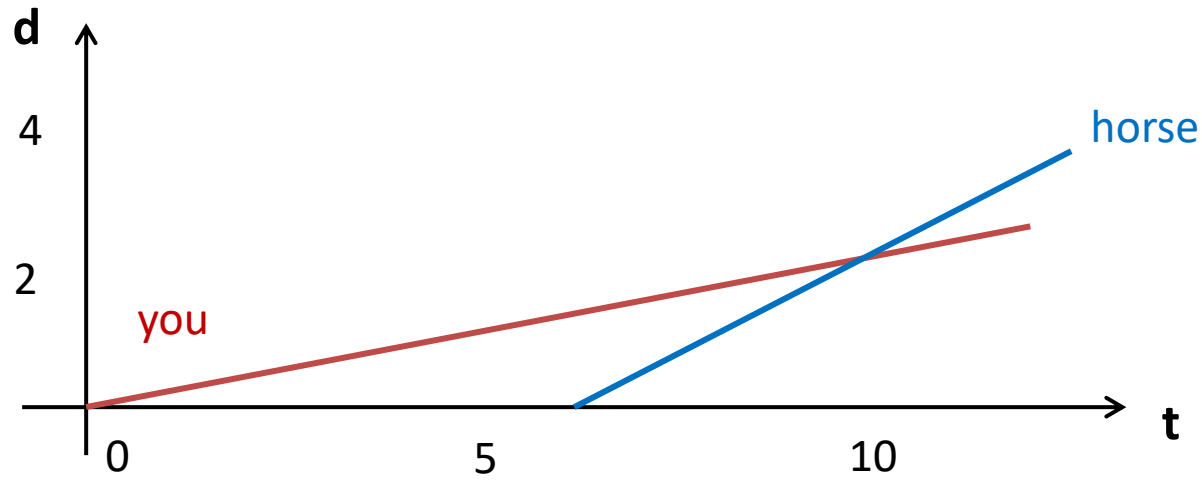


Linear Equations Example

- 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

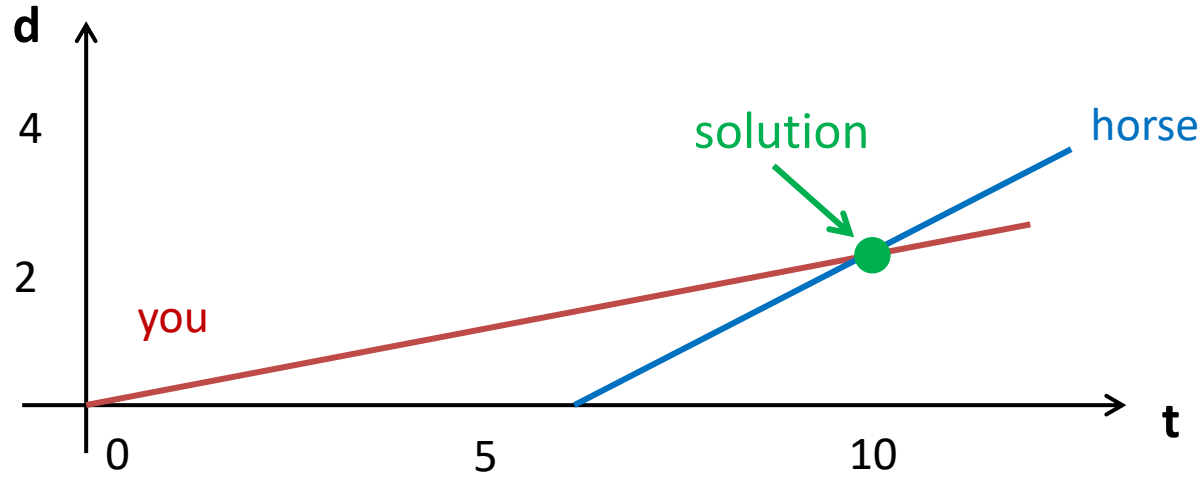


Linear Equations Example

- 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)

d = 0.2t

d = 0.5(t–6) = 0.5t-3



Linear Systems of Equations

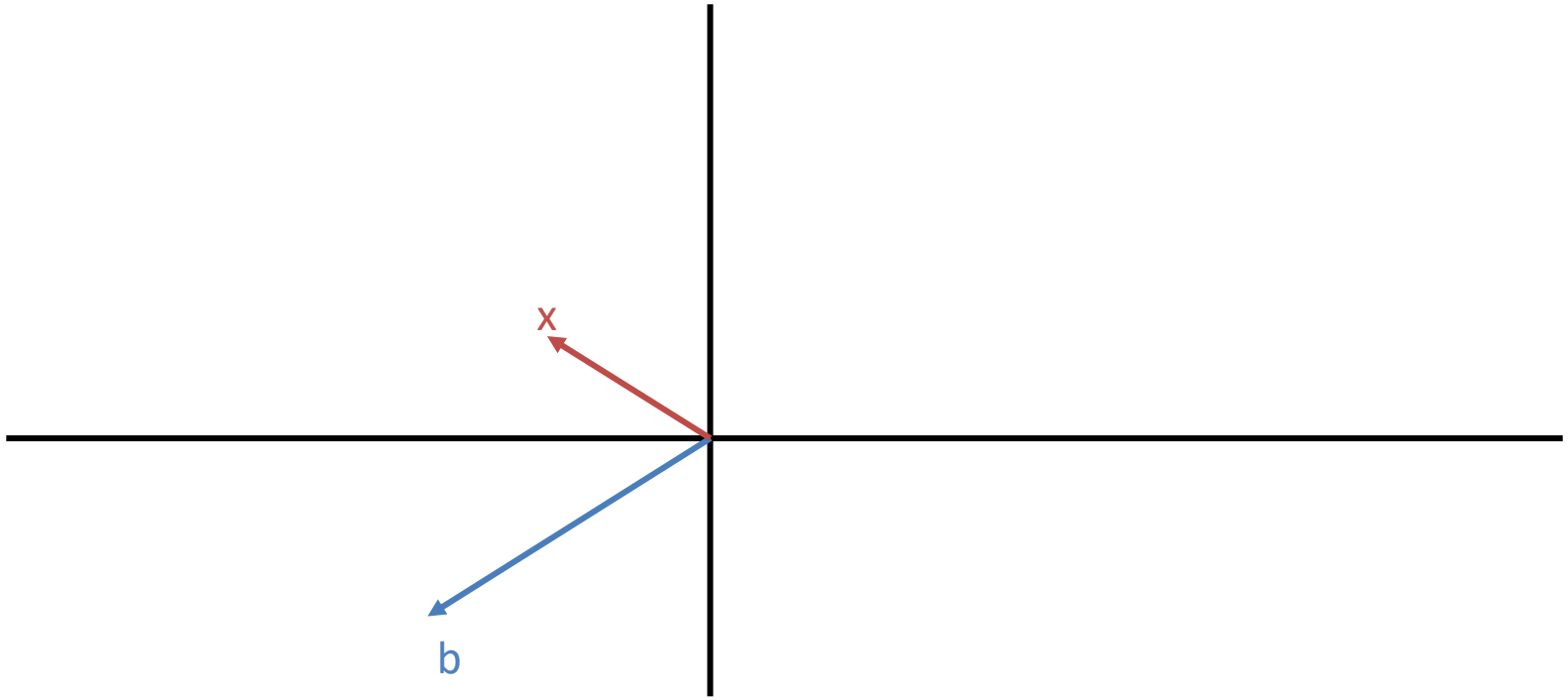
$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\
&\ldots \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m.
\end{aligned}
$$

Linear Systems of Equations

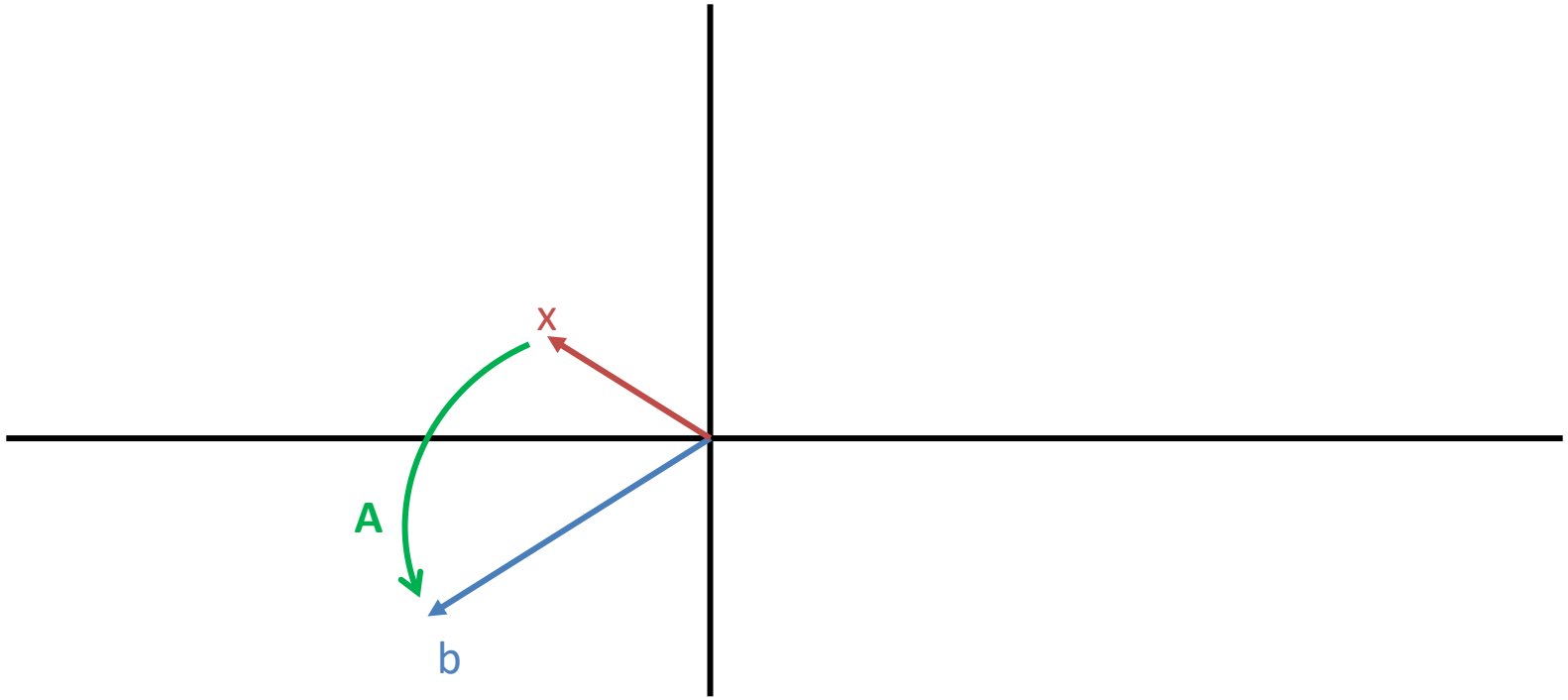
A x b

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

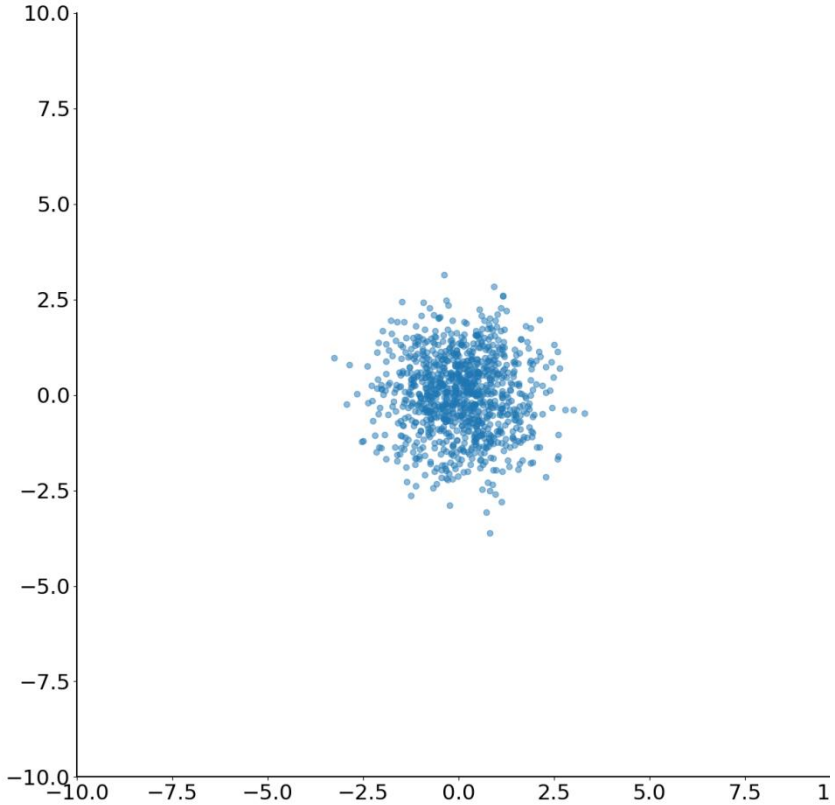
Geometric Interpretation

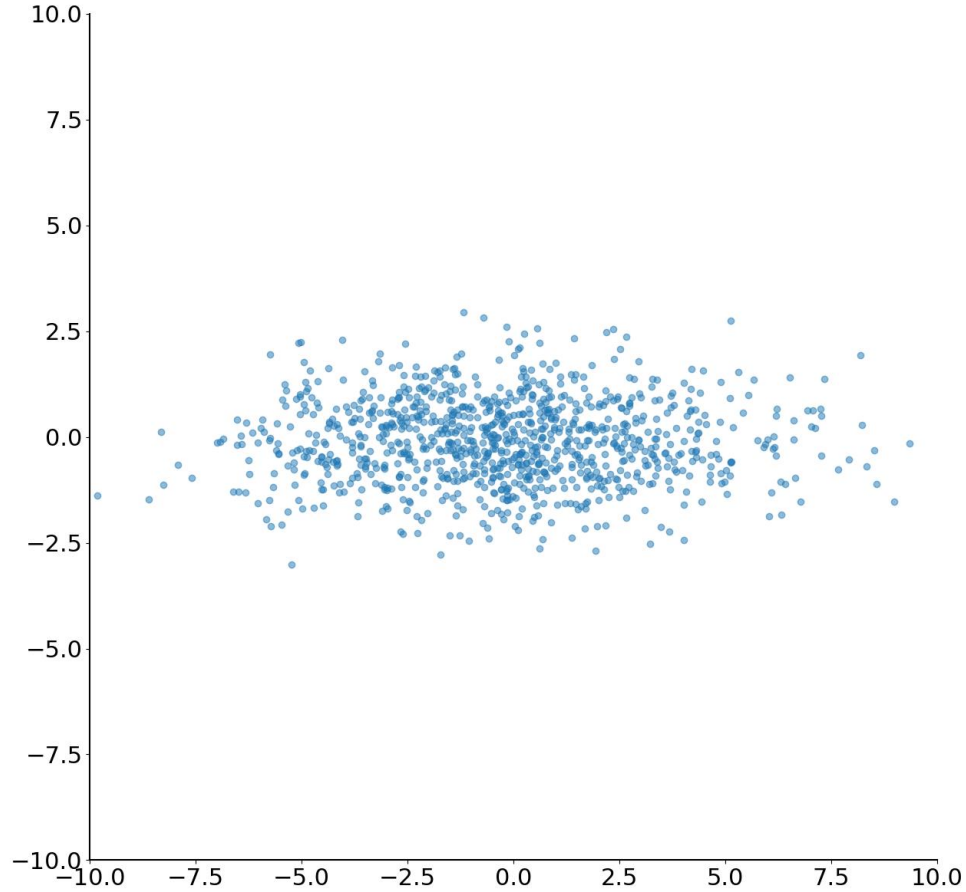


Geometric Interpretation

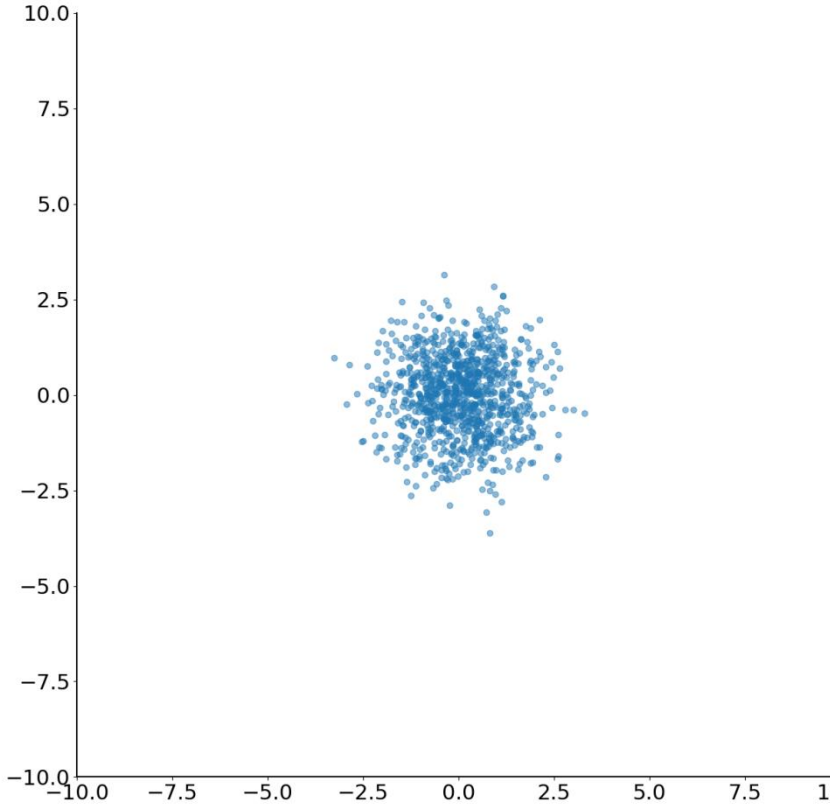


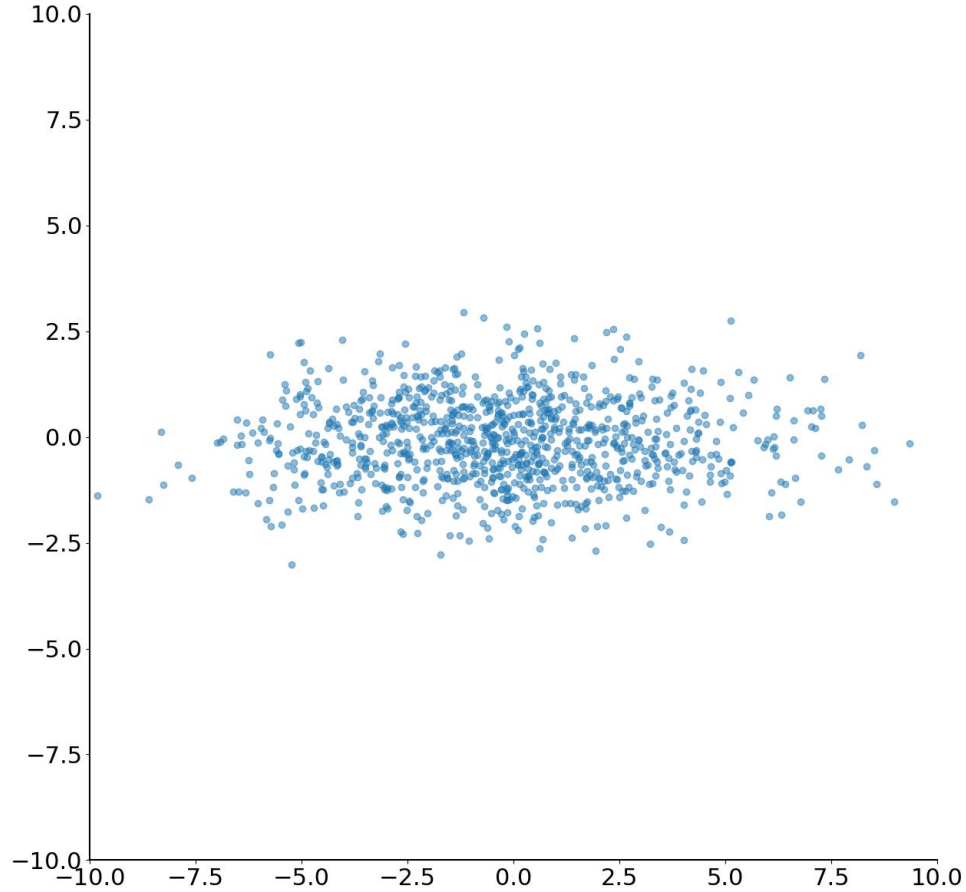
Example Transformations



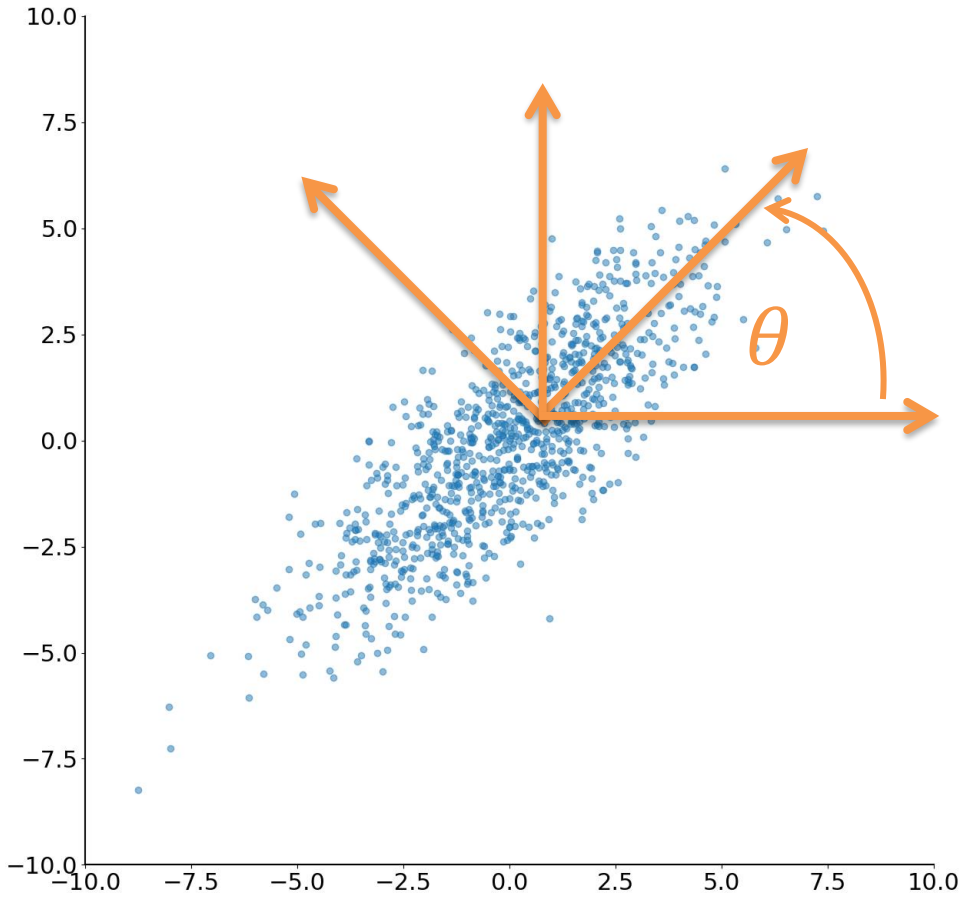


Example Transformations $S = \begin{bmatrix} x & 0 \\ 0 & y \end{bmatrix}$





Example Transformations



$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta & \cos(\theta) \end{bmatrix}$$

$$Ax = b?$$

$Ax = b$?

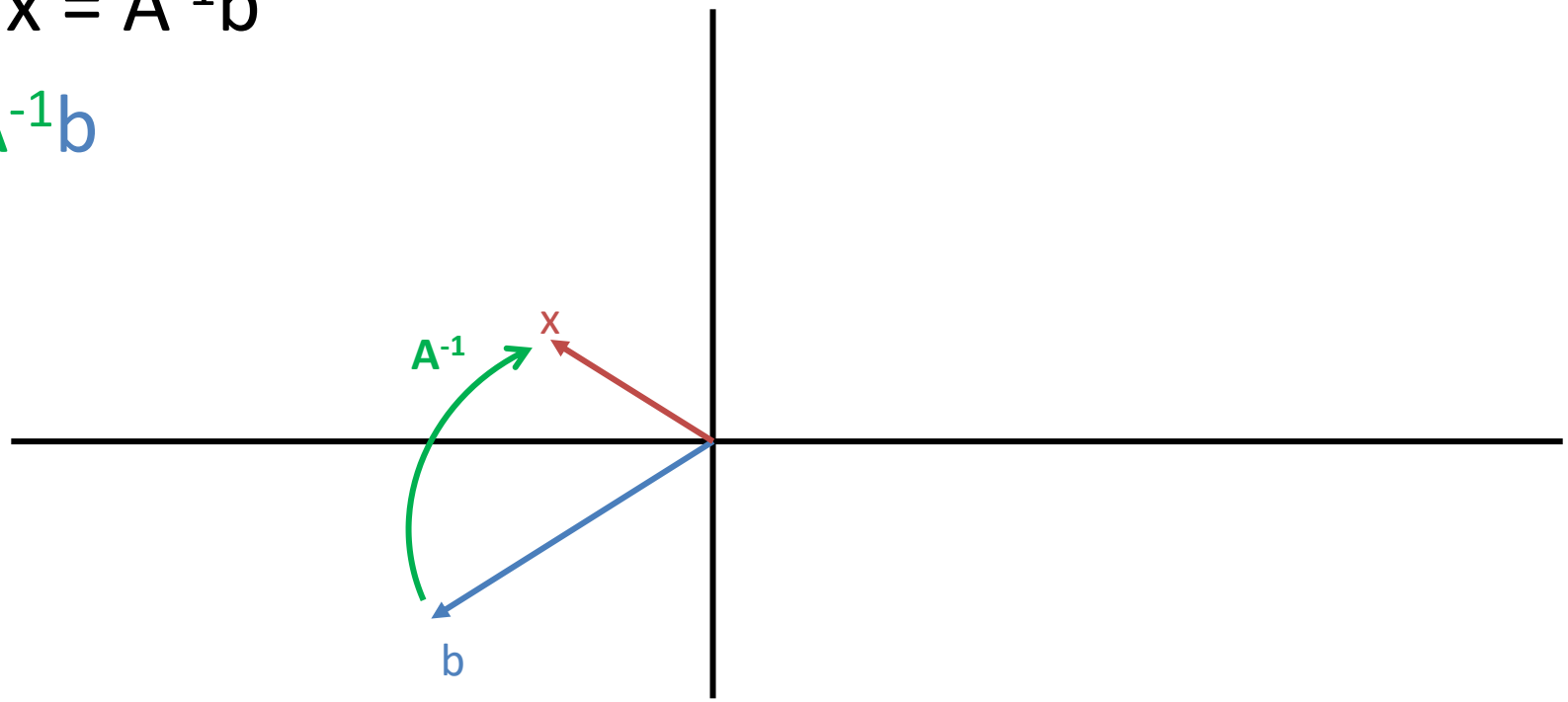
- $A^{-1}A\ x = A^{-1}b$

$Ax = b$?

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

$Ax = b$?

- $A^{-1}A\,x = A^{-1}b$
- $x = A^{-1}b$



numpy.linalg.inv

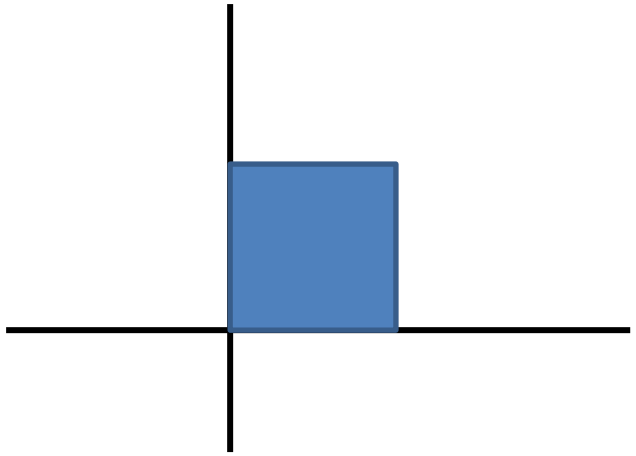
numpy.linalg.inv(*a*) [source]

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])`.

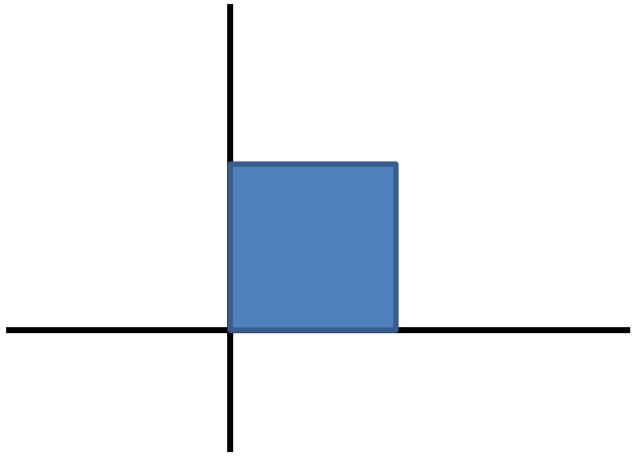
```
>>> a = np.array([[[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
>>> inv(a)
array([[[-2. ,  1. ],
        [ 1.5, -0.5]],
       [[-5. ,  2. ],
        [ 3. , -1. ]]])
```

Measuring the area changes of transformations



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

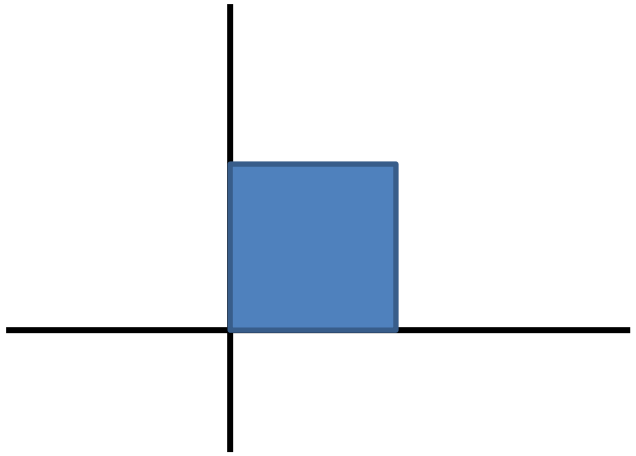
Measuring the area changes of transformations



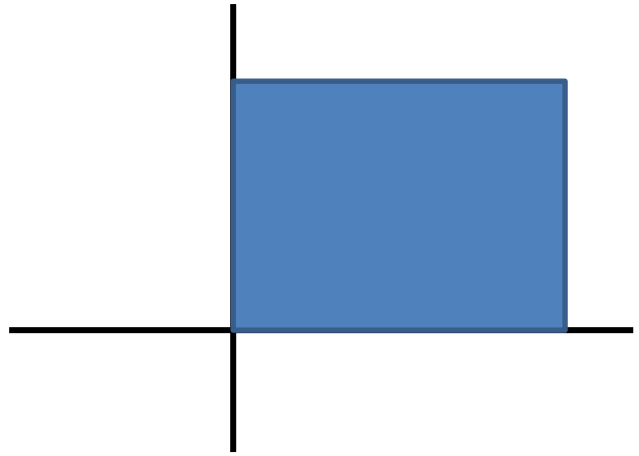
$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ → area 1

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

Measuring the area changes of transformations

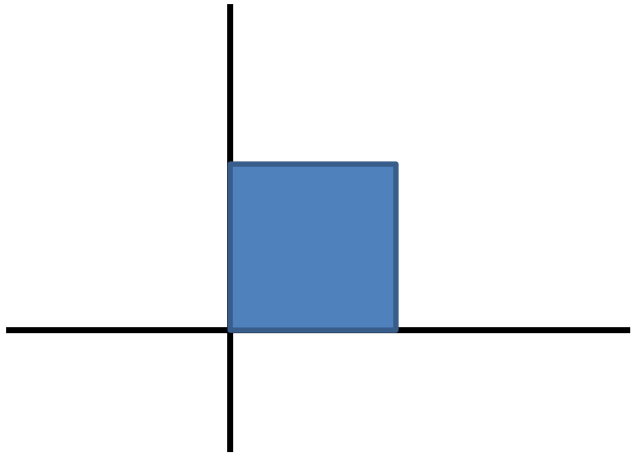


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow$ area 1

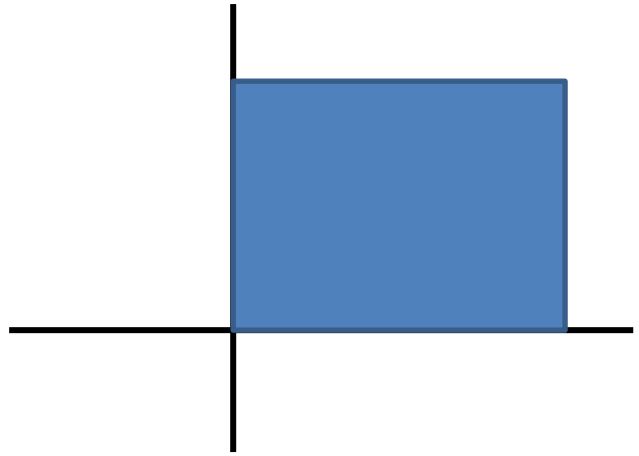


$\begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix}$

Measuring the area changes of transformations

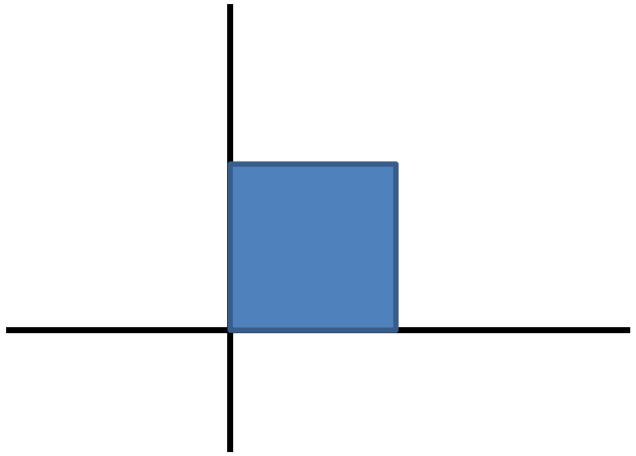


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow$ area 1

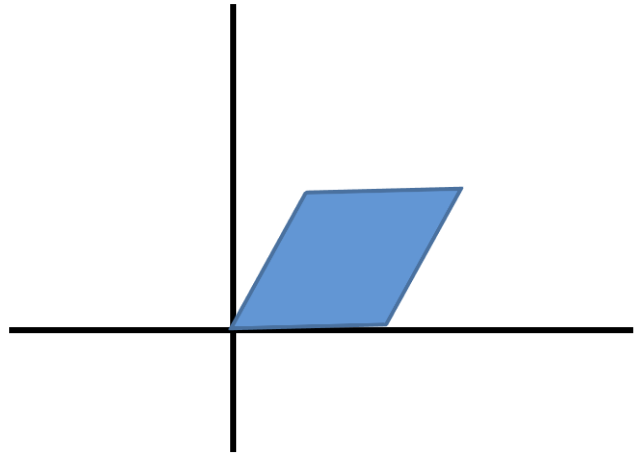


$\begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix} \rightarrow$ area 3

Measuring the area changes of transformations

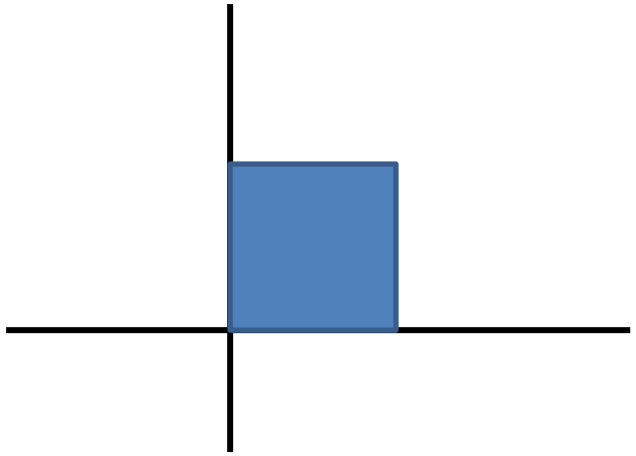


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ → area 1

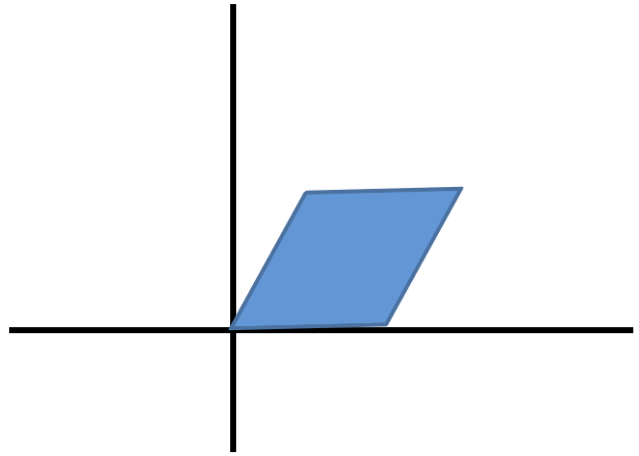


$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ → area 1

Measuring the area changes of transformations



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



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

This is the “determinant” of a matrix

numpy.linalg.det

numpy.linalg.**det**(*a*) [source]

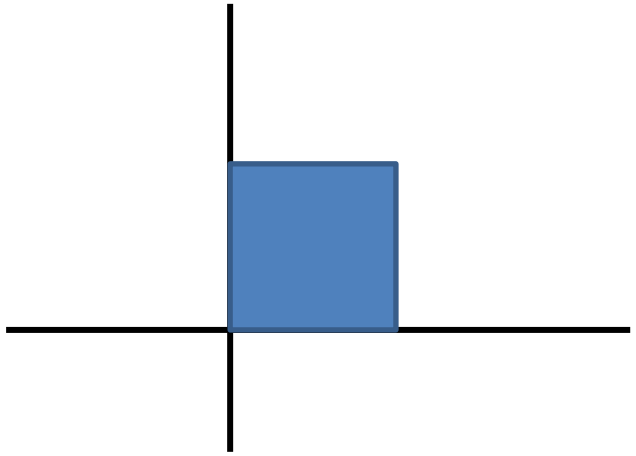
Compute the determinant of an array.

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

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

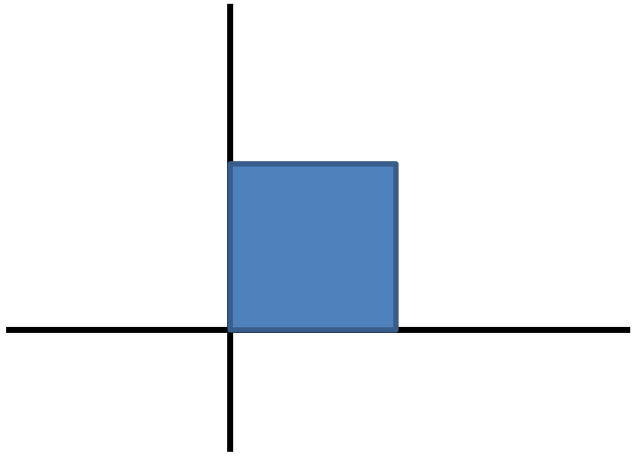
```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0
```

Eigenvalues and Eigenvectors

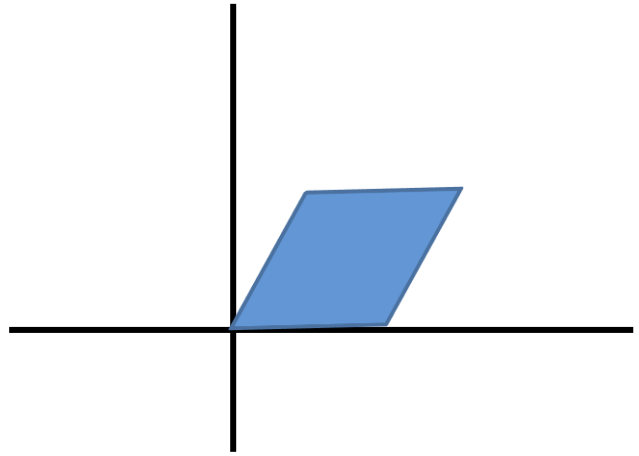


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ → area 1

Eigenvalues and Eigenvectors

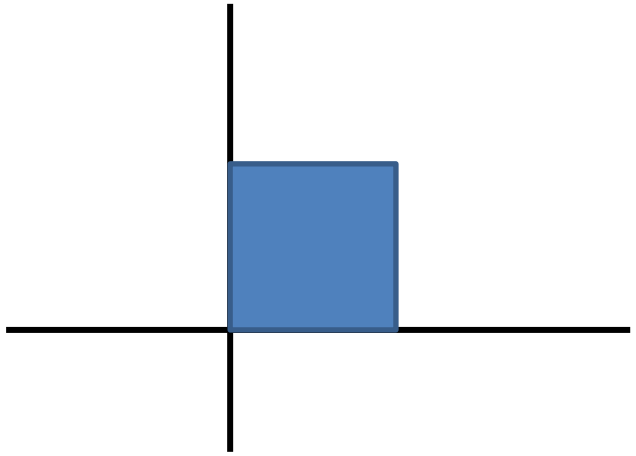


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ → area 1

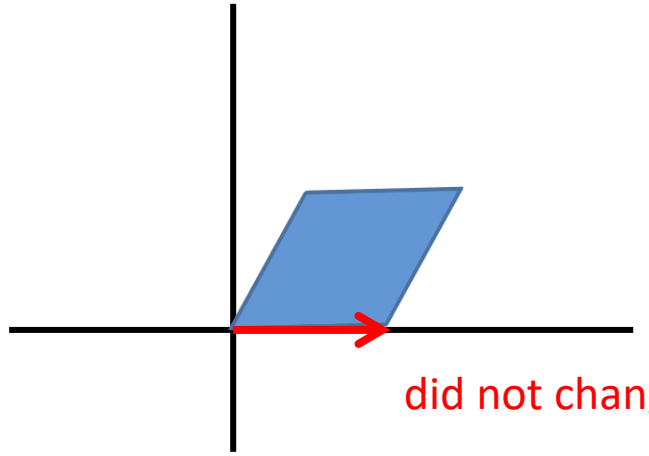


$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ → area 1

Eigenvalues and Eigenvectors

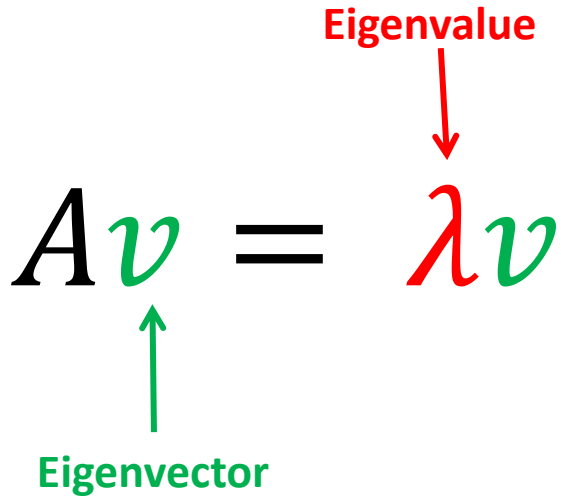


$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ → area 1



$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ → area 1

Eigenvalues and Eigenvectors



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`.

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

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

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.

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

$$\operatorname{cond}(A) \equiv \left\| A^{-1} \right\| \cdot \| A \|$$

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:

p	norm for matrices
None	2-norm, computed directly using the `SVD`
'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 decompostion 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 * x0 + x1 = 9` and `x0 + 2 * x1 = 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:

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

Iterative methods for linear equation systems:

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

Iterative methods for least-squares problems:

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

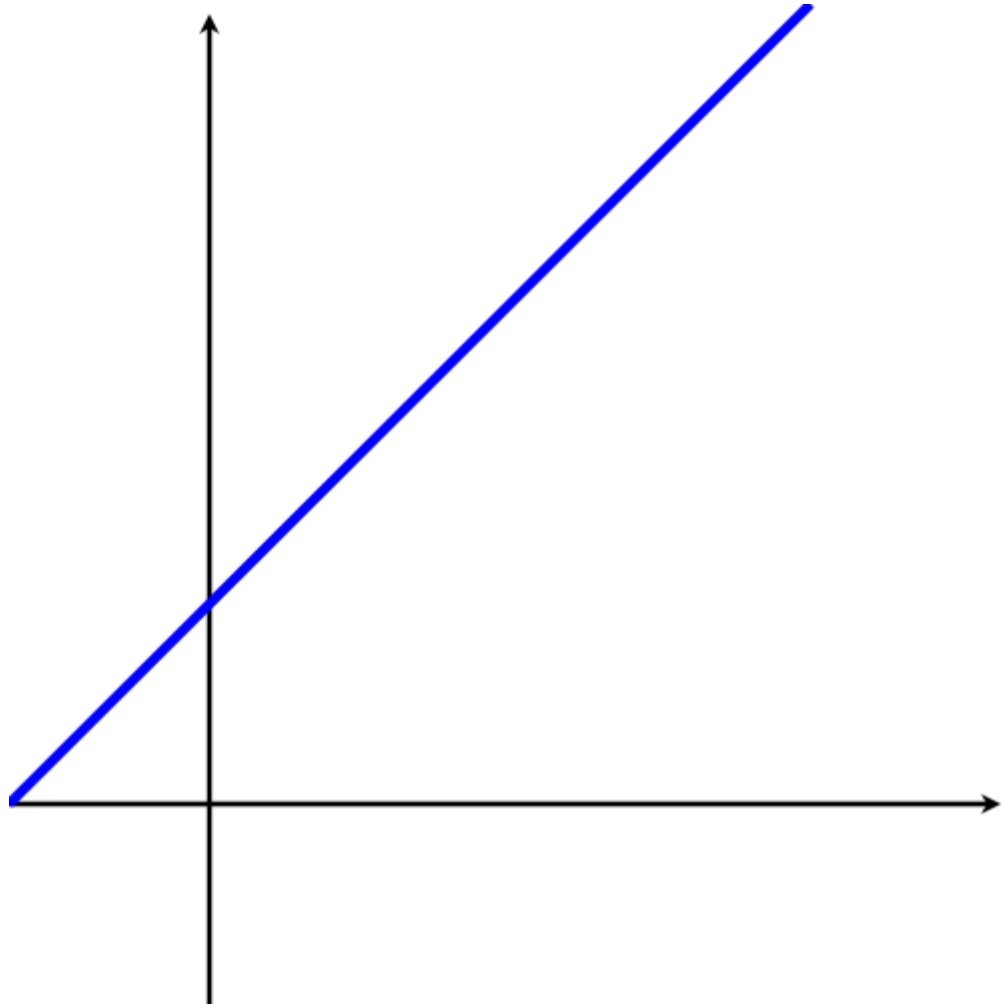
Solving Linear Systems of Equations

- n = m: can have an exact solution

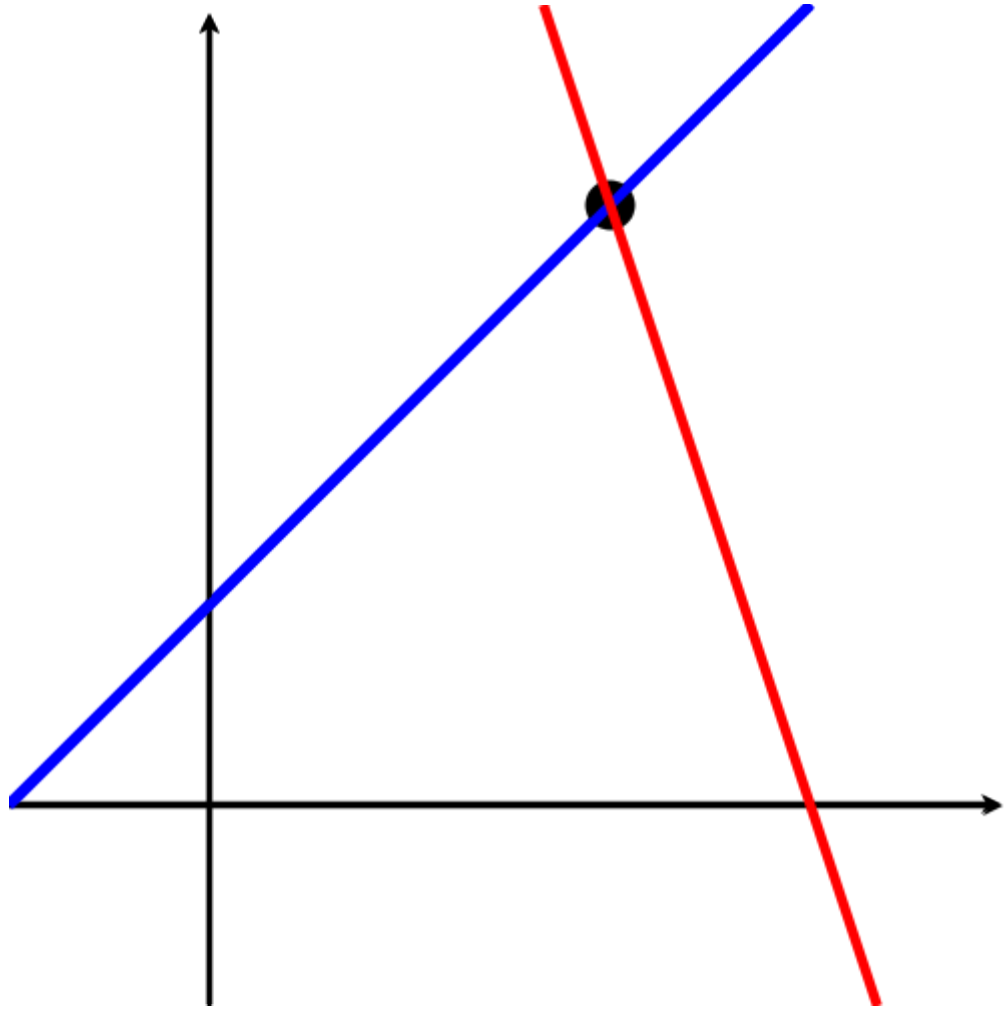
Solving Linear Systems of Equations

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

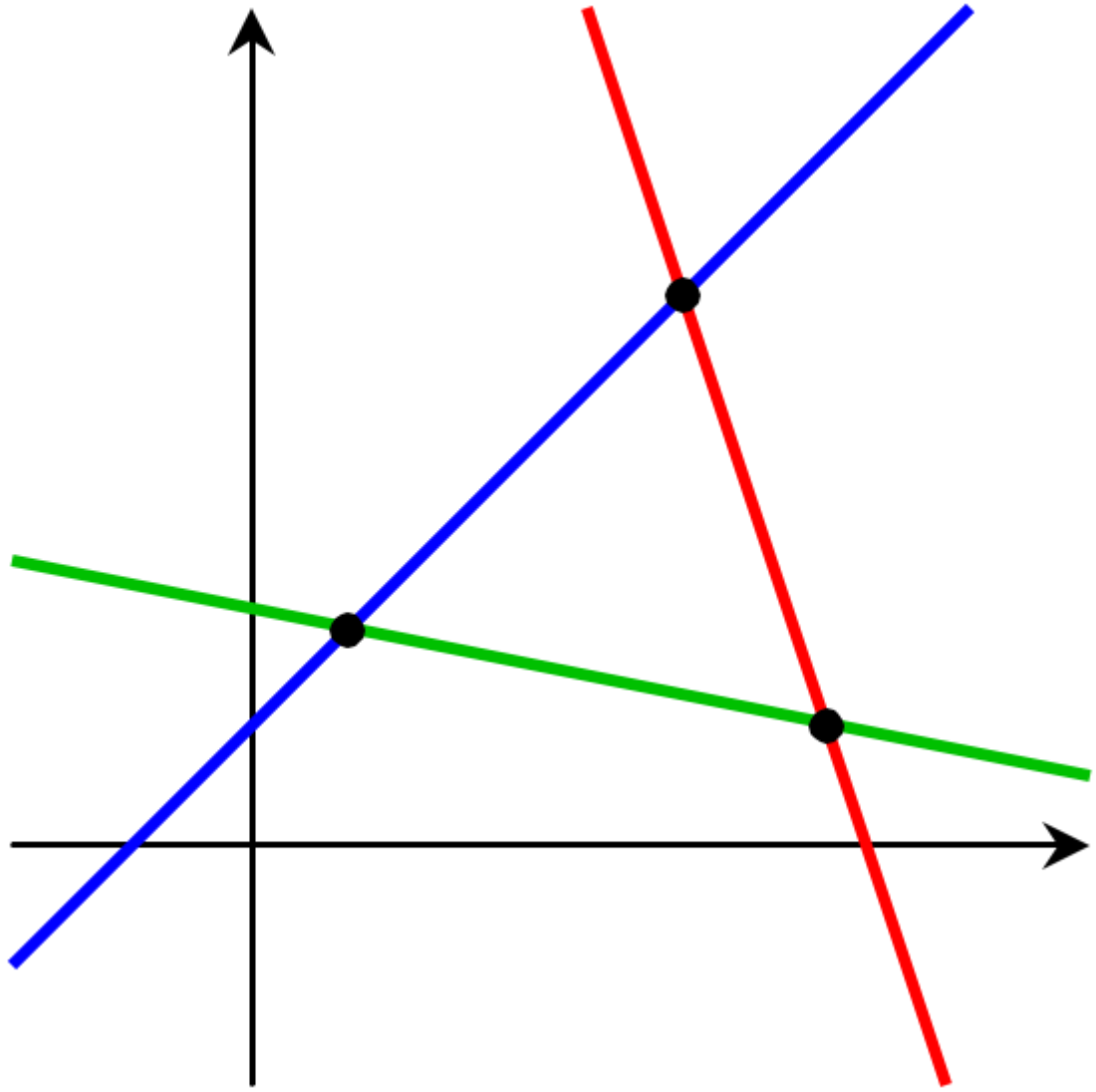
1 Equation



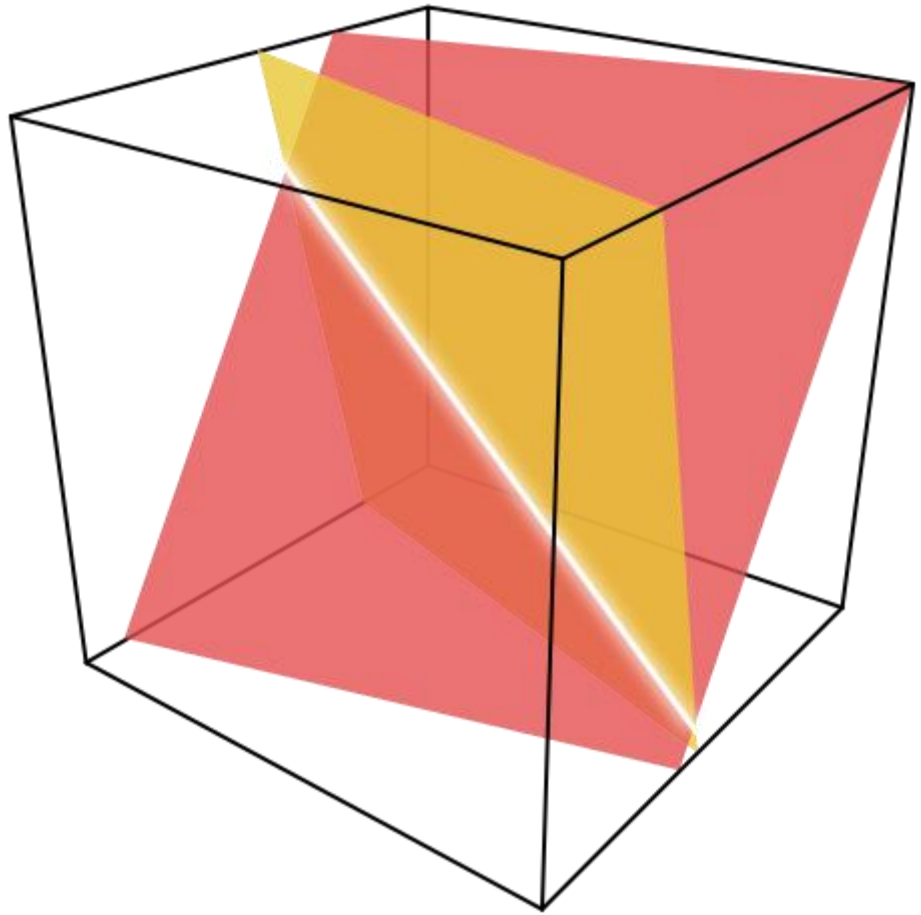
2 Equations



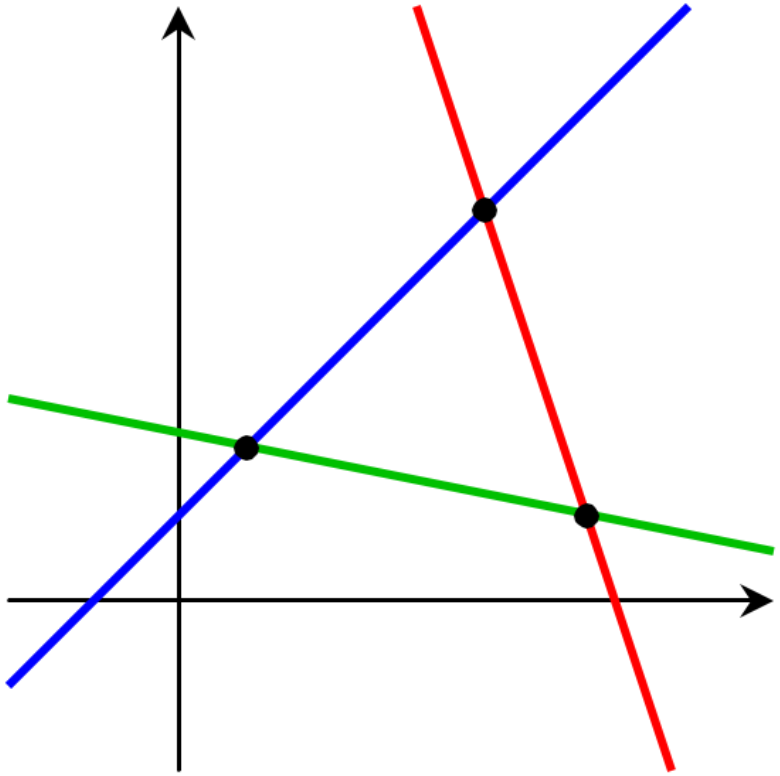
3 Equations



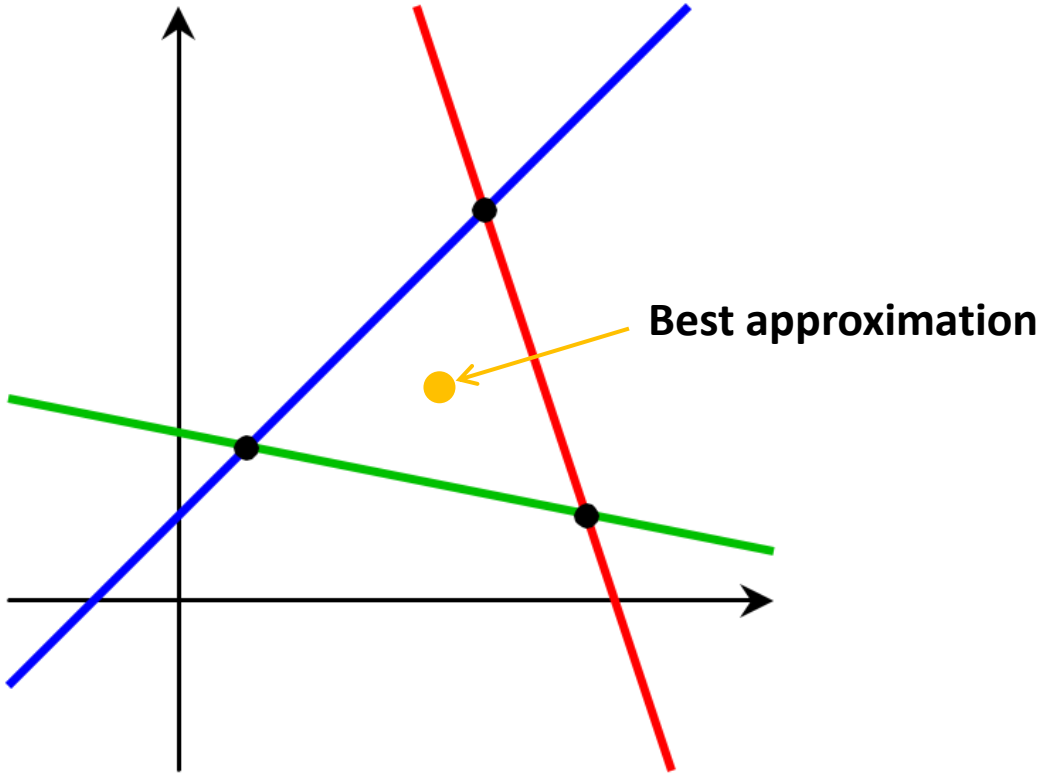
2 Equations 3 Unknowns



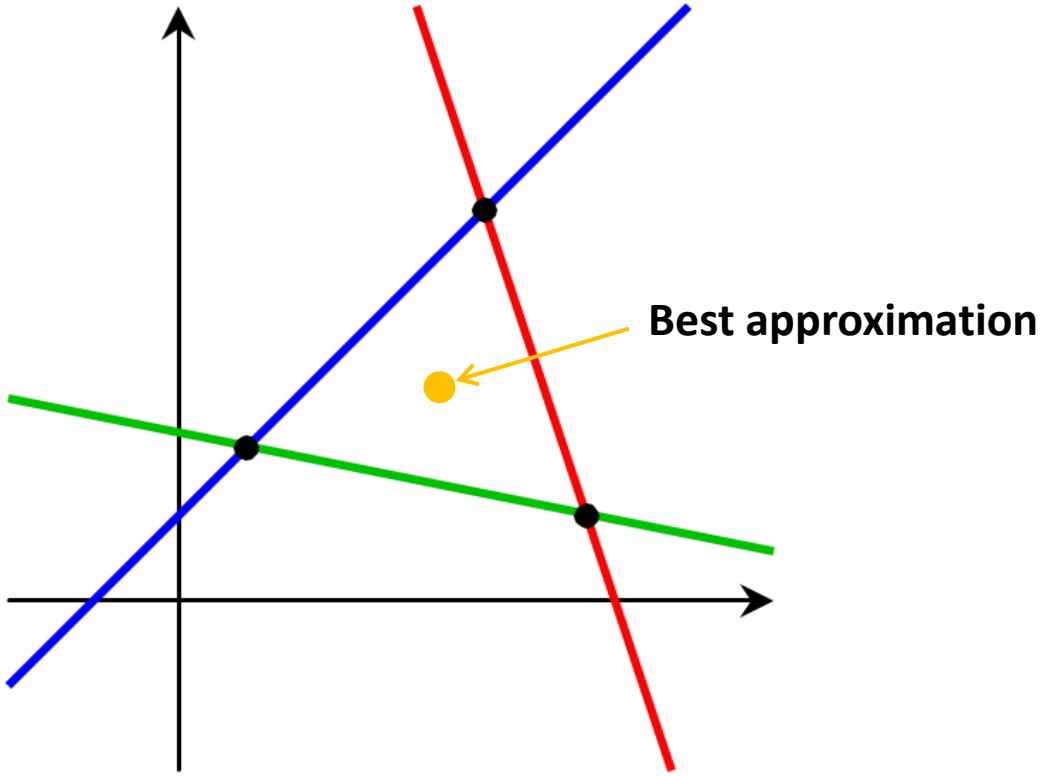
Approximation?

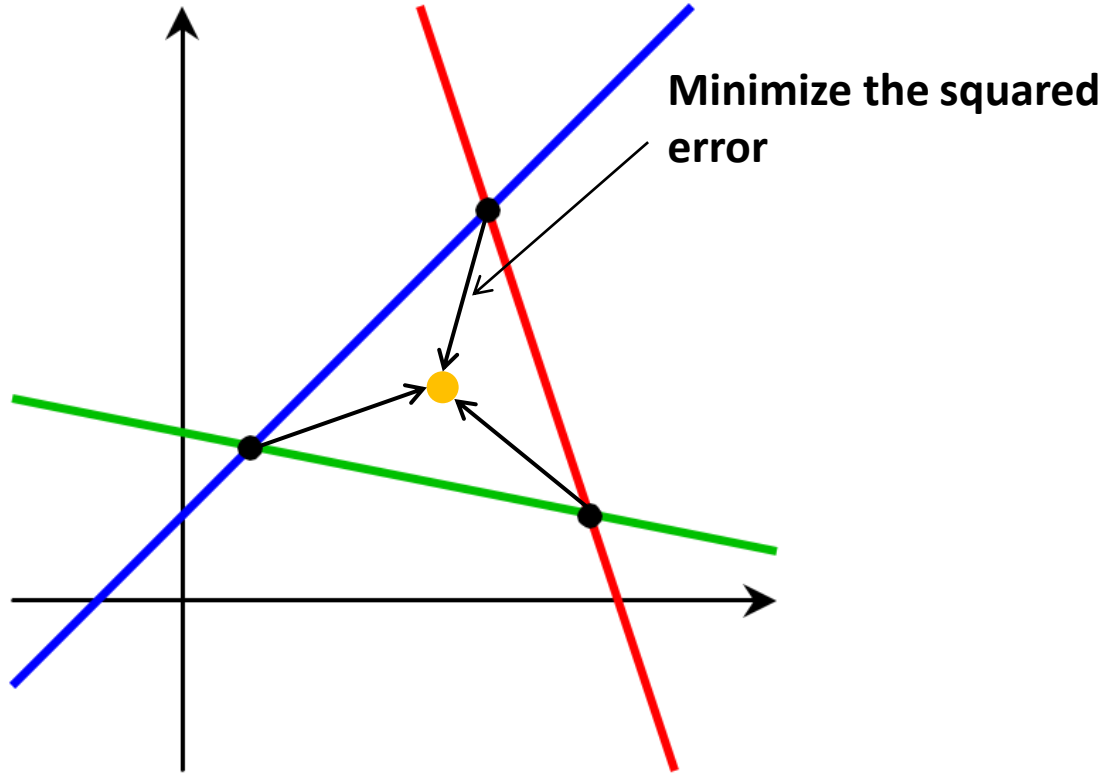


Approximation?



Least-squares





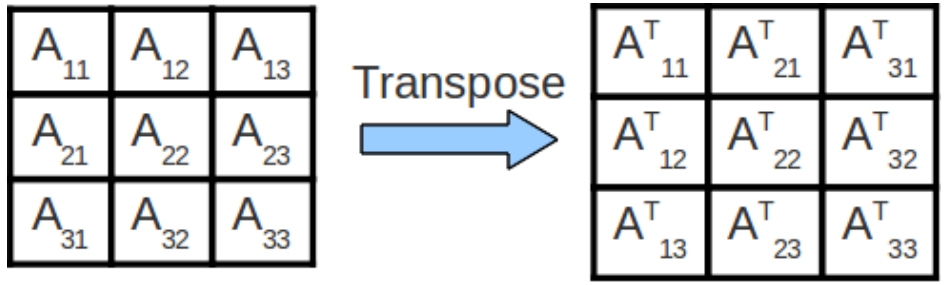
numpy.linalg.lstsq

numpy.linalg.**lstsq**(*a*, *b*, *rcond=-1*) [source]

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

Solves the equation *a x = b* by computing a vector *x* that minimizes the Euclidean 2-norm *|| b - a x ||^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.

Matrix Transposition A → A.T



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

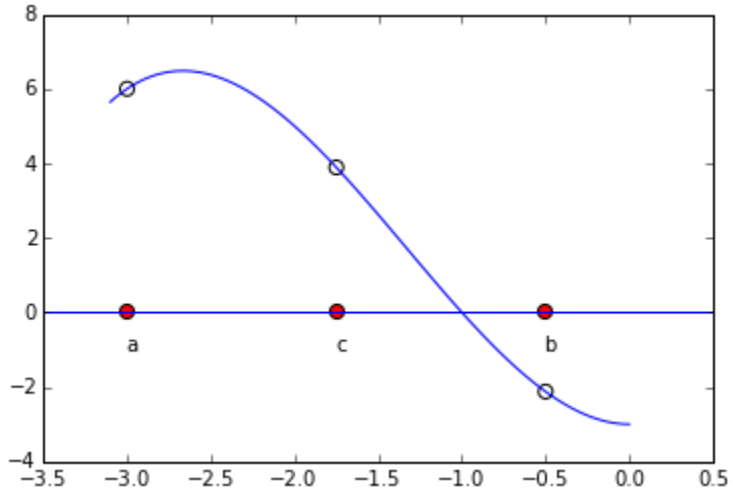
- $if\ f(x) = ax^2 + bx + c\ then$:

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

- For many functions an analytical solution cannot be found

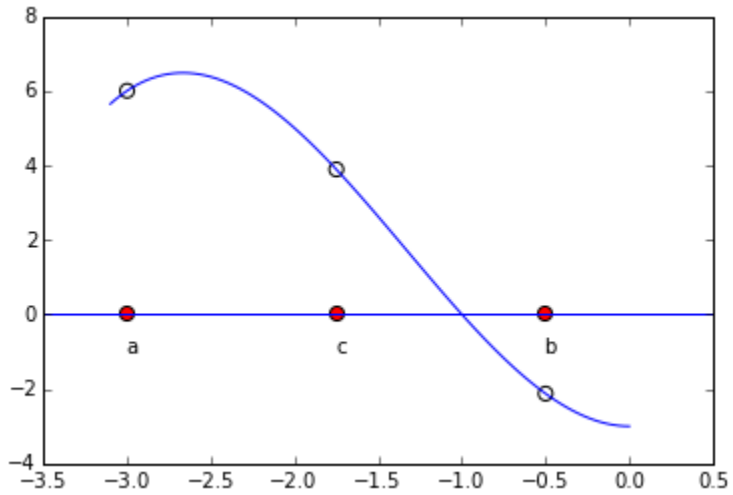
Bisection method

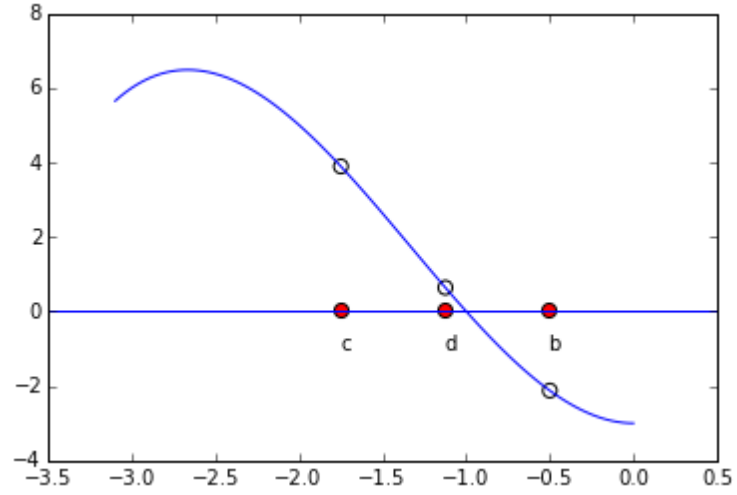
- Find points a and b such that `f(a)f(b) < 0`
- Calculate `c = (a+b)/2`, if `b-c < ε`: end
- If `f(a)f(c) < 0 : b = c`, else: `a = c`



Bisection method

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



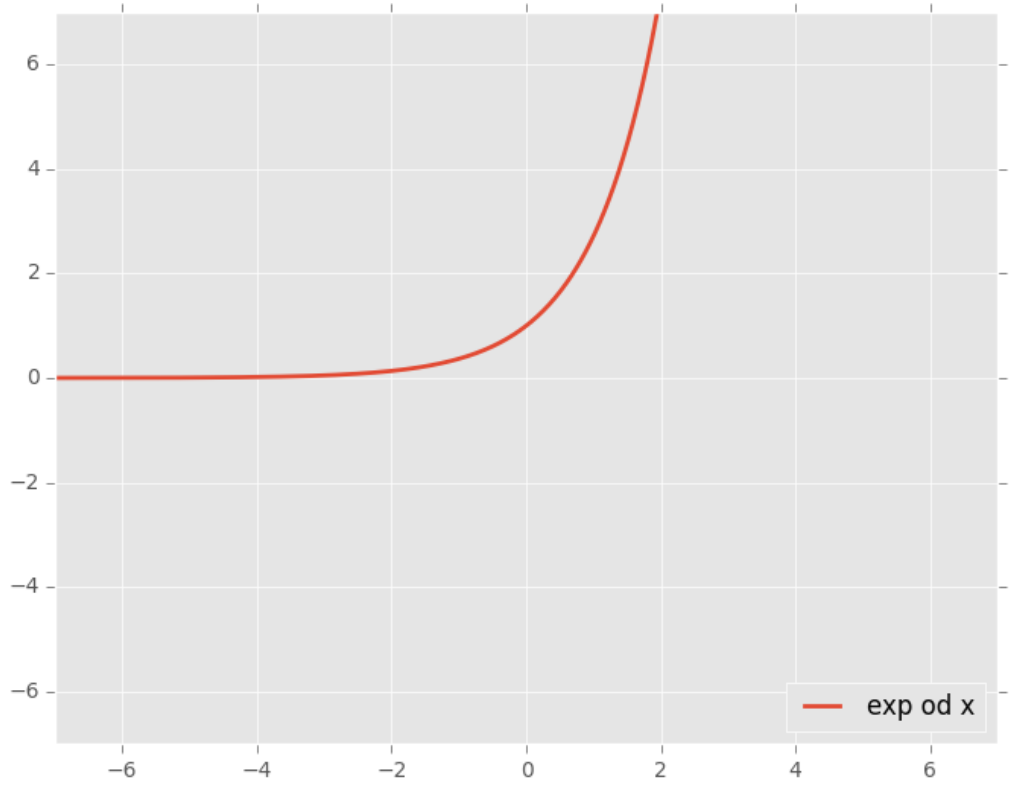


Taylor Series

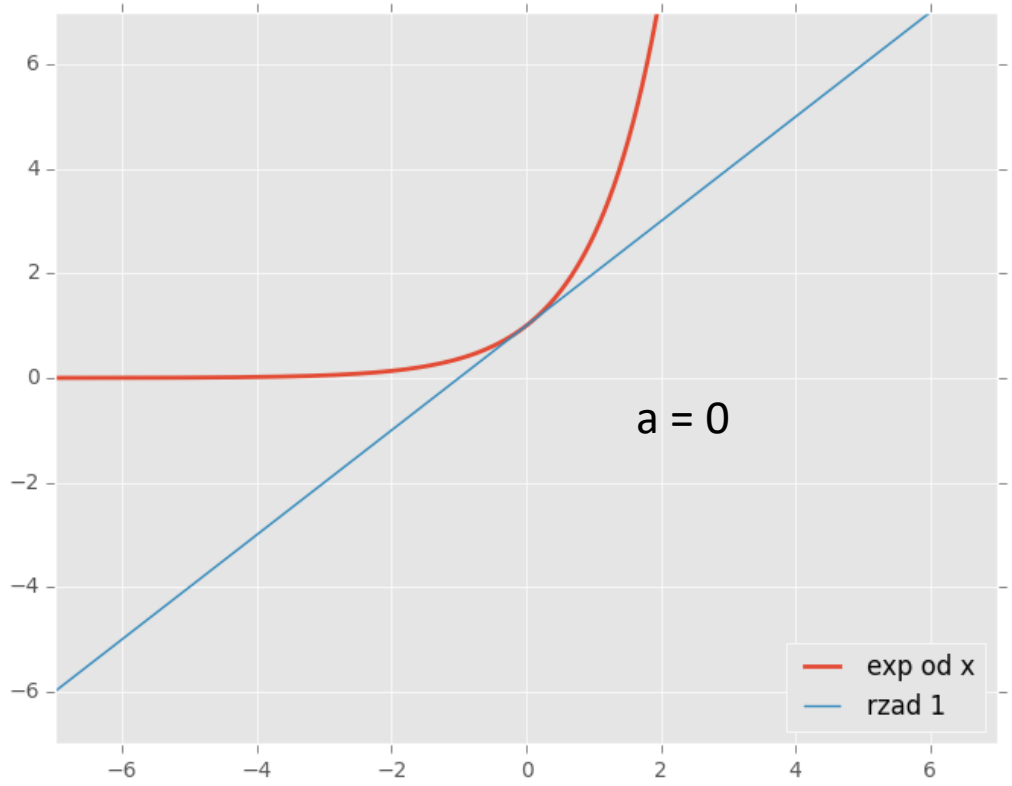
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Taylors theorem: there exists b∈[x,a], such that $f(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(b)(x-a)^2$

Taylor Series

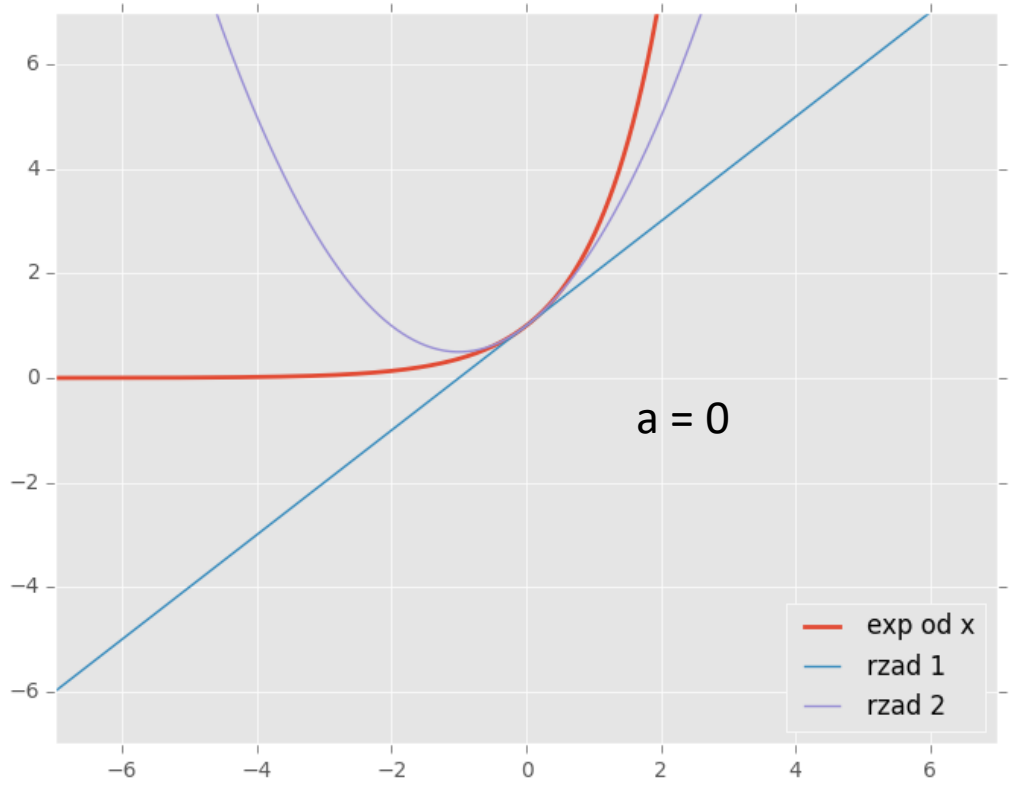


Taylor Series



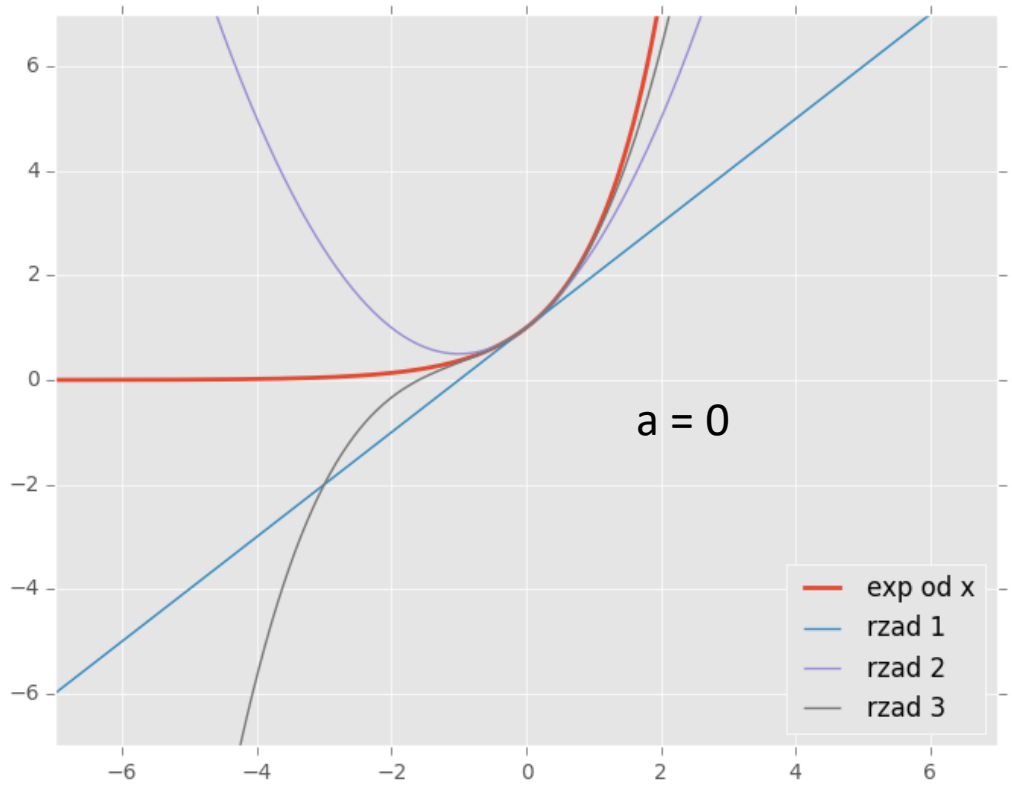
$$f(a) + \frac{f'(a)}{1!}(x-a)$$

Taylor Series



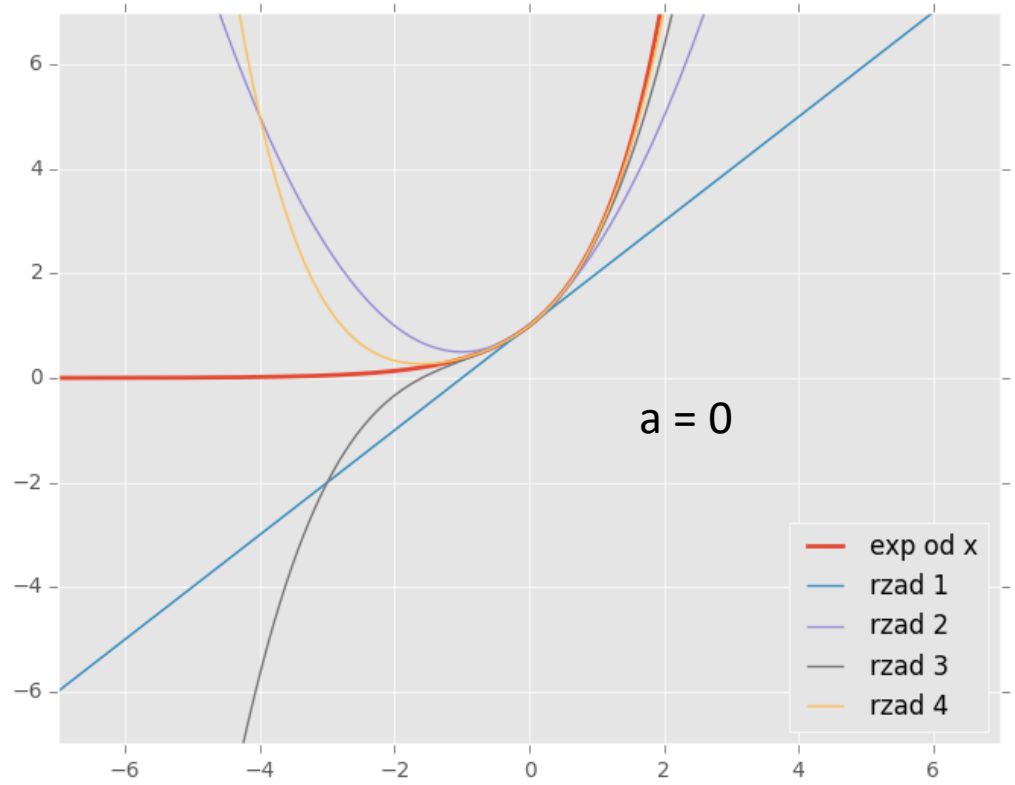
$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2$$

Taylor Series



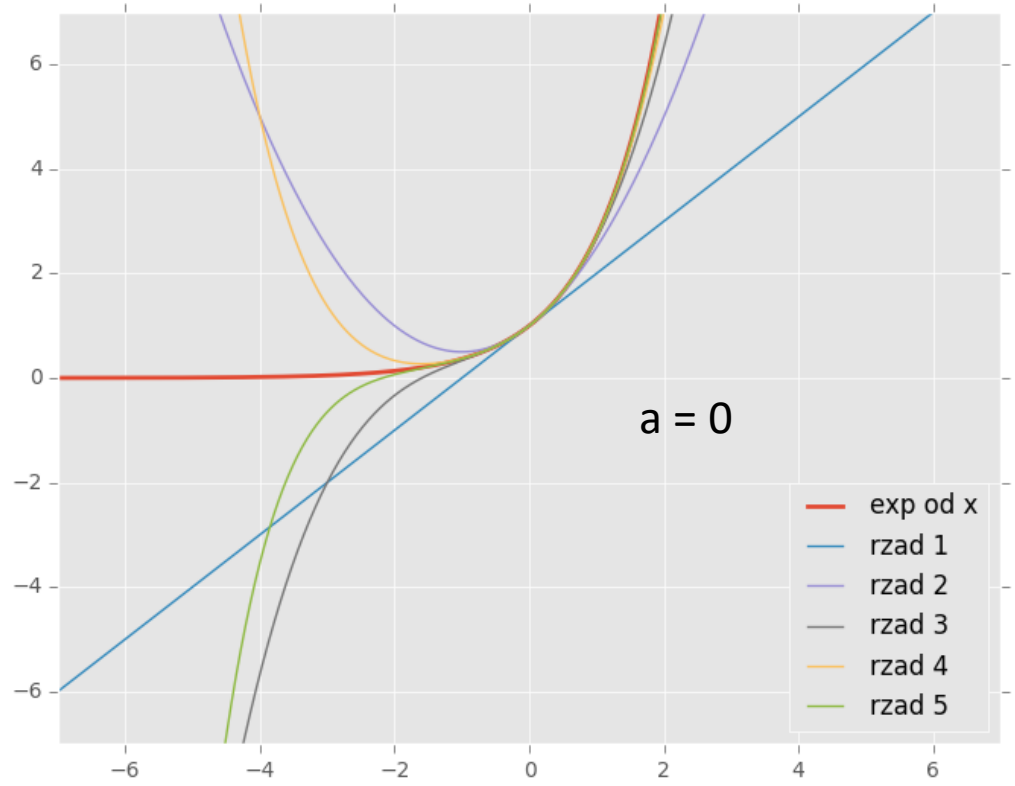
$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3$$

Taylor Series



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

Taylor Series



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

Taylor series → 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:

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

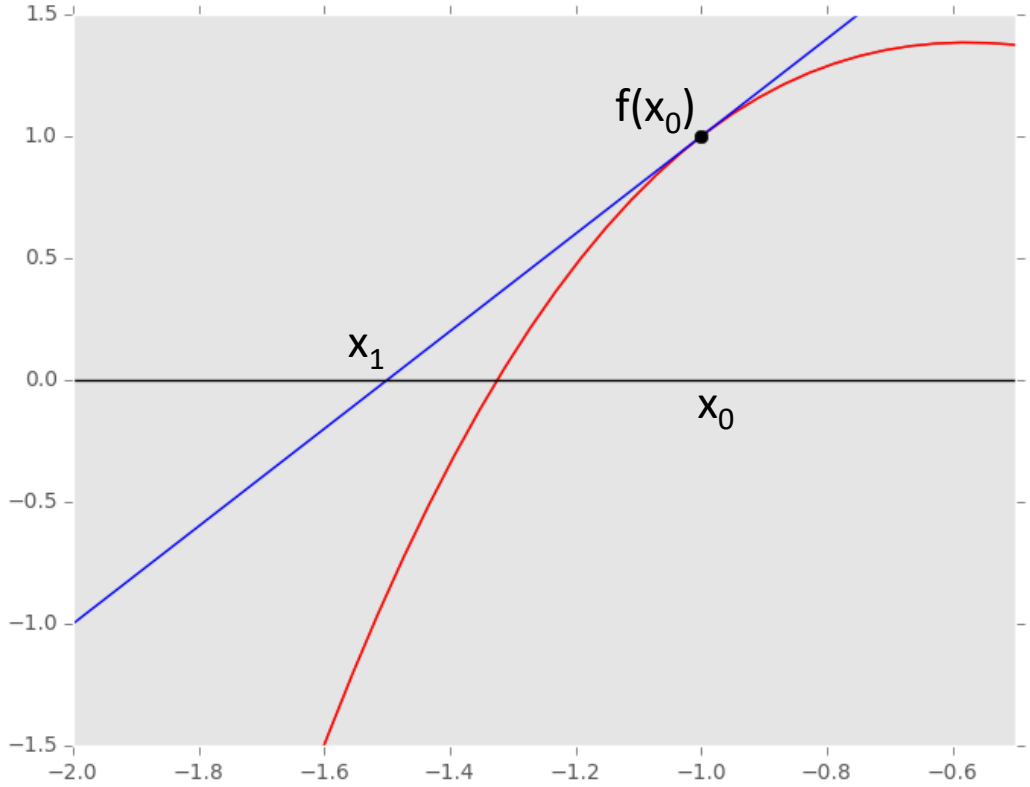
which results in: $z \approx x_0 - \frac{f(x_0)}{f'(x_0)} \equiv x_1$

Newton's method

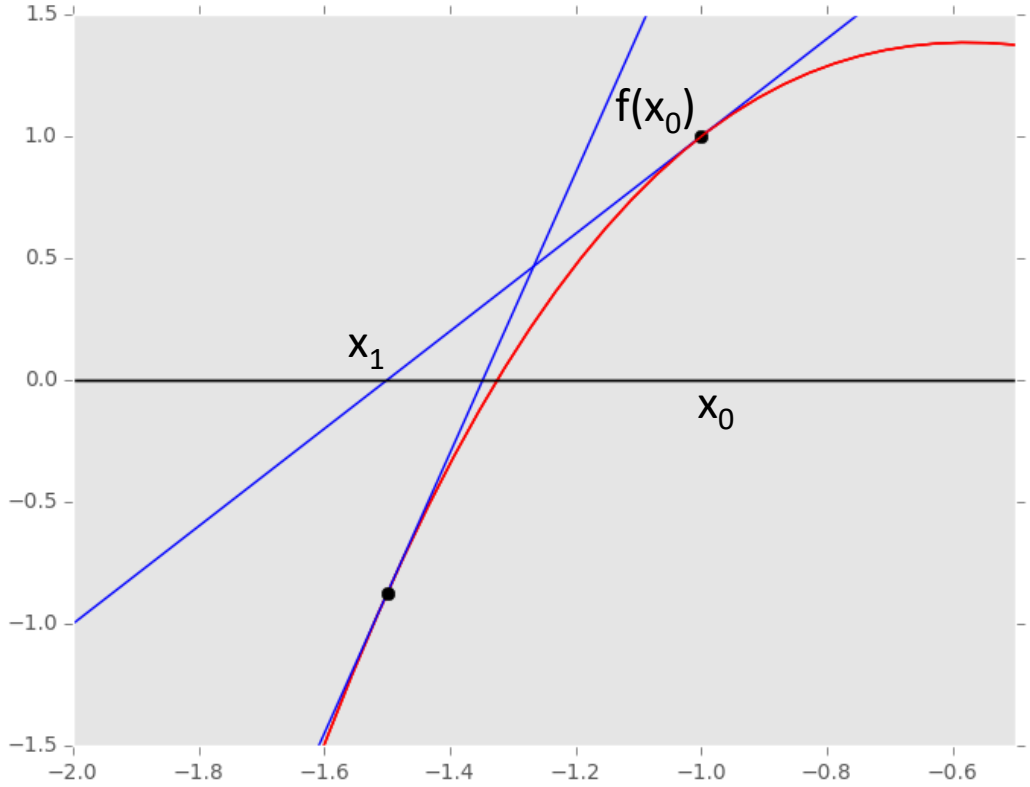
Iteration:

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

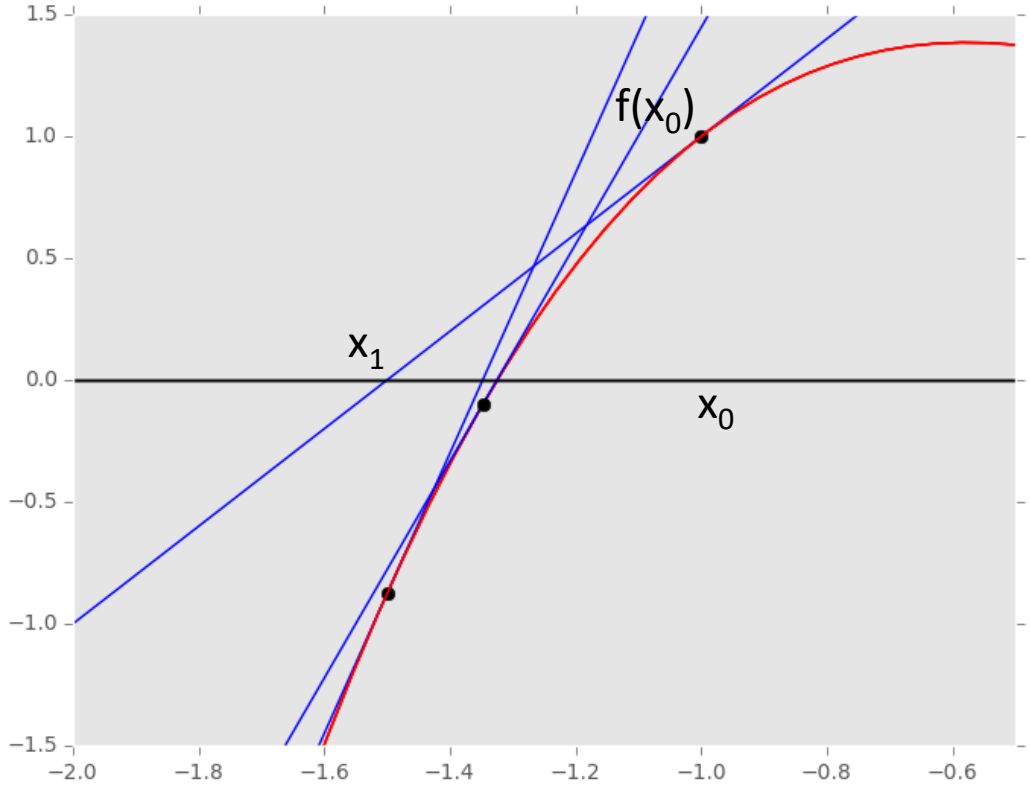
Newton's method



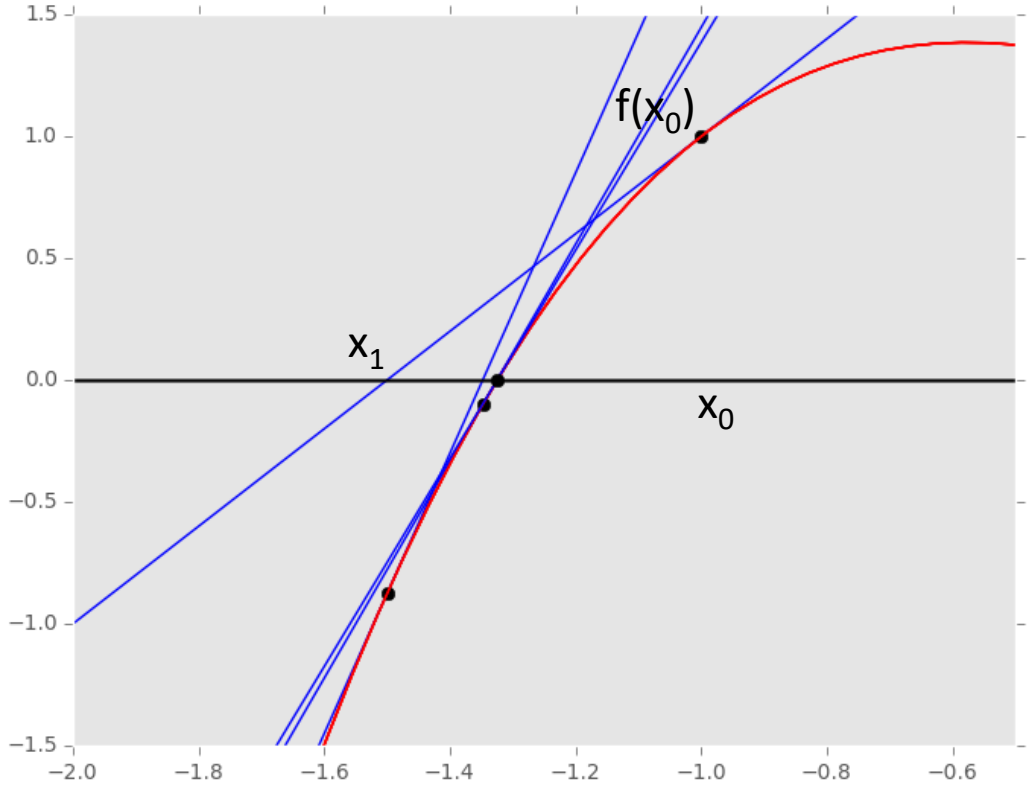
Newton's method



Newton's method



Newton's method



Scipy.optimize

Root finding

Scalar functions

root_scalar(f[, args, method, bracket, ...])	Find a root of a scalar function.
brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method with hyperbolic extrapolation.
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval using Ridder's method.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval using bisection.
newton(func, x0[, fprime, args, tol, ...])	Find a zero of a real or complex function using the Newton-Raphson (or secant or Halley's) method.
toms748(f, a, b[, args, k, xtol, rtol, ...])	Find a zero using TOMS Algorithm 748 method.
RootResults(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?
- Compute the rank and conditioning number of matrix A which expresses the system of equations.
- Solve the system of equations and give the LU decomposition of matrix A. 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 nupy). 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 1^{st} 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 1^{st} and 2^{nd} orders.

Vandermonde Matrix

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

Exercise

- Plot the conditioning number as a function of p for the following linear system (p ϵ [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.

Vectorizing Functions

SCALAR SINC FUNCTION

```
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```
# attempt
>>> x = array((1.3, 1.5))
>>> sinc(x)
ValueError: The truth value of an array with
more than one element is ambiguous. Use
a.any() or a.all()
>>> x = r_[-5:5:100j]
>>> y = vsinc(x)
>>> plot(x, y)
```

SOLUTION

```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc(x)
array([-0.1981, -0.2122])

>>> x2 = linspace(-5, 5, 101)
>>> plot(x2, sinc(x2))
```