Root finding, linear and nonlinear sets of equations

Kristina Kislyakova
University of Vienna - 22.10.2025

General problem we will consider today

one or more functions, depending on one or more variables

$=$

known values

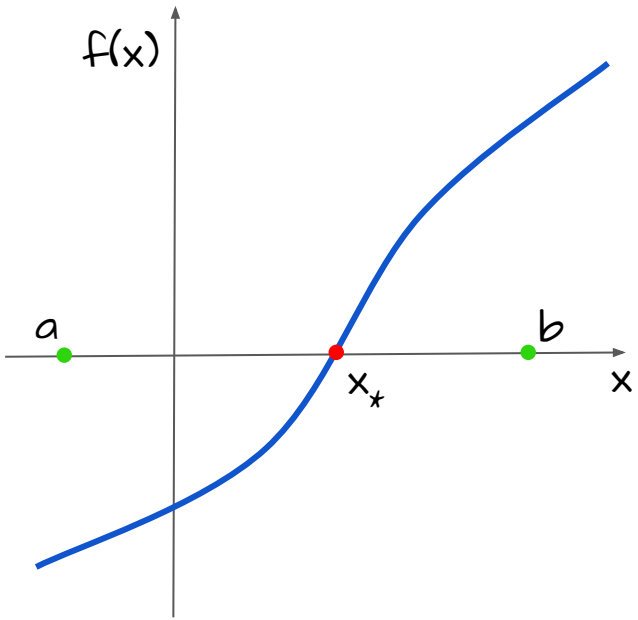
we want to find the values of the variables that make this true

Root finding methods

Introduction

The problem

We want to find the value x_* such that for a function $f(x)$, where $x \in (a,b)$, we have: $f(x_*) = 0$.



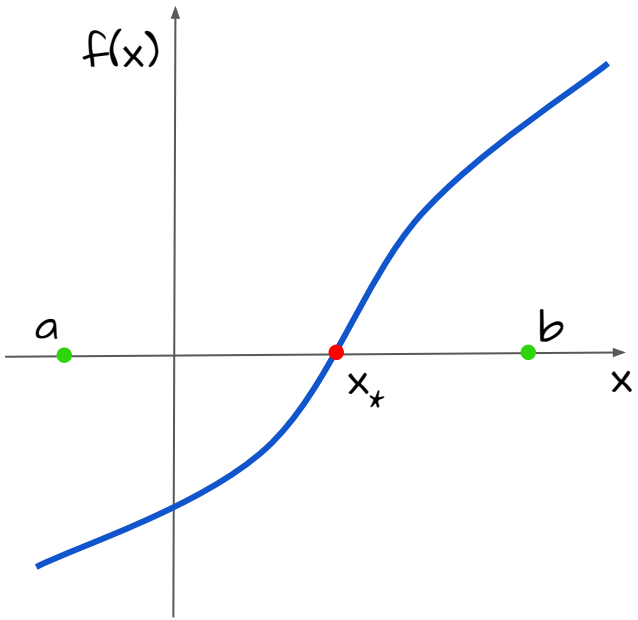
Introduction

The problem

We want to find the value x_* such that for a function $f(x)$, where $x \in (a,b)$, we have: $f(x_*) = 0$.

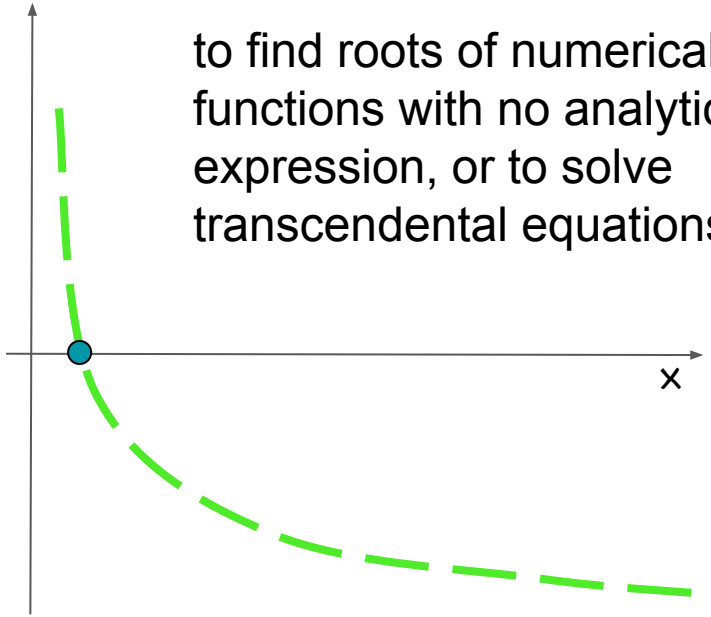
How do we solve this?

A typical procedure is to find a recurrence relation of the form $x_{i+1} = R(x_i)$ and to iterate. Therefore, we start from an *approximate trial solution*, and improve the solution by applying the *recurrence relation*, until some predetermined *convergence criterion* is satisfied.



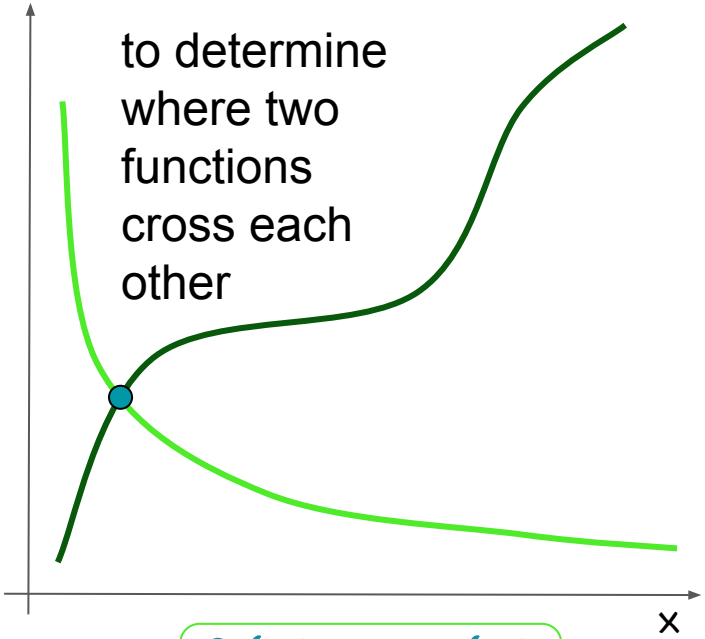
Introduction

Why is this useful?



to find roots of numerical functions with no analytic expression, or to solve transcendental equations

$$f(x) = 0$$



to determine where two functions cross each other

$$f(x) = g(x)$$

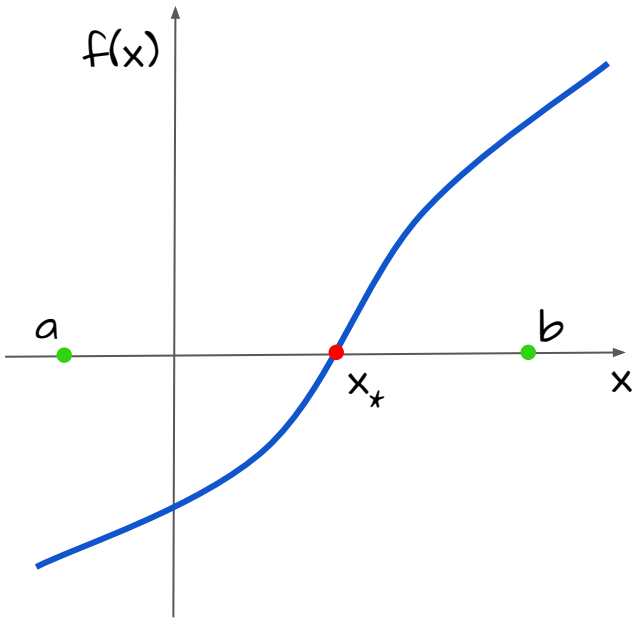
Introduction

We will look at the following methods:

1. **bisection method**
2. **secant method**
3. **Newton-Raphson method**

For each of them, we will consider the following points:

- under what conditions a method will converge
- if it converges, then how fast
- what is the best choice for the initial guess

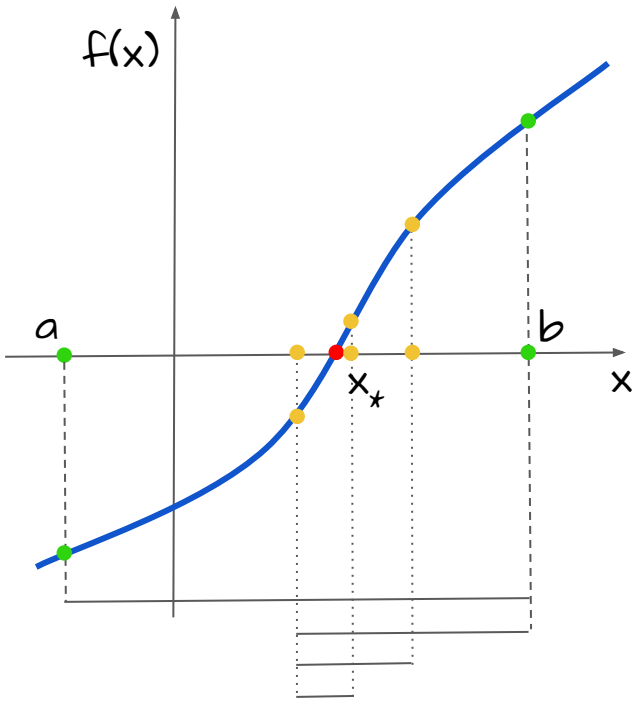


1. Bisection method

If $f(a)\cdot f(b) < 0$, then there exist a root in the interval $[a,b]$.

We compute the middle point $x_m = (a+b)/2$, then:

- if $f(x_m)\cdot f(a) < 0$, we consider the new interval $[a,x_m]$ and repeat...
- if $f(x_m)\cdot f(b) < 0$, we consider the new interval $[x_m,b]$ and repeat...
- if $f(x_m) = 0$ or $|a-b|<\varepsilon$, we have found our solution!



1. Bisection method

Error and convergence

The error is defined as $\varepsilon_i = |x_* - x_i|$.
For this method:

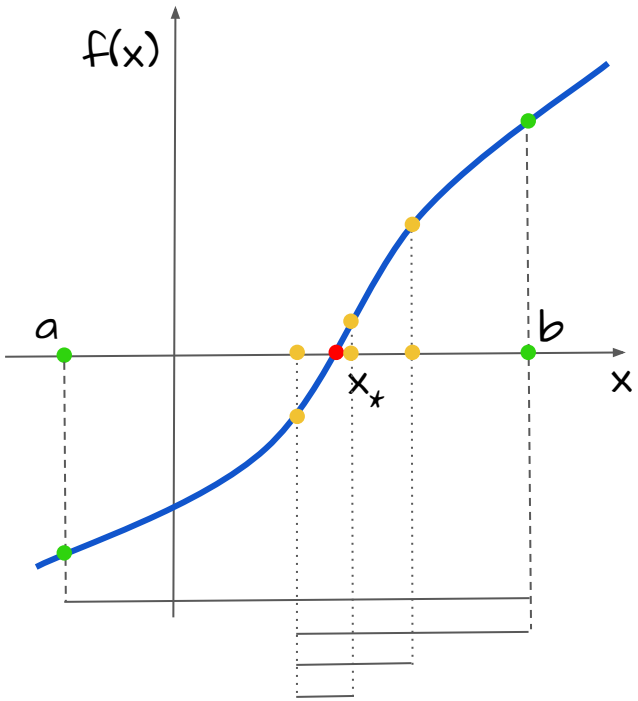
$$\varepsilon_i \leq \frac{|a_i - b_i|}{2}$$

and in every step the error is half of the previous step (linear convergence):

$$\varepsilon_{i+1} = \frac{\varepsilon_i}{2} = \frac{\varepsilon_{i-1}}{2^2} = \ldots = \frac{\varepsilon_0}{2^{(i+1)}}$$

If we demand an error ε, we can compute the number of steps needed to obtain it:

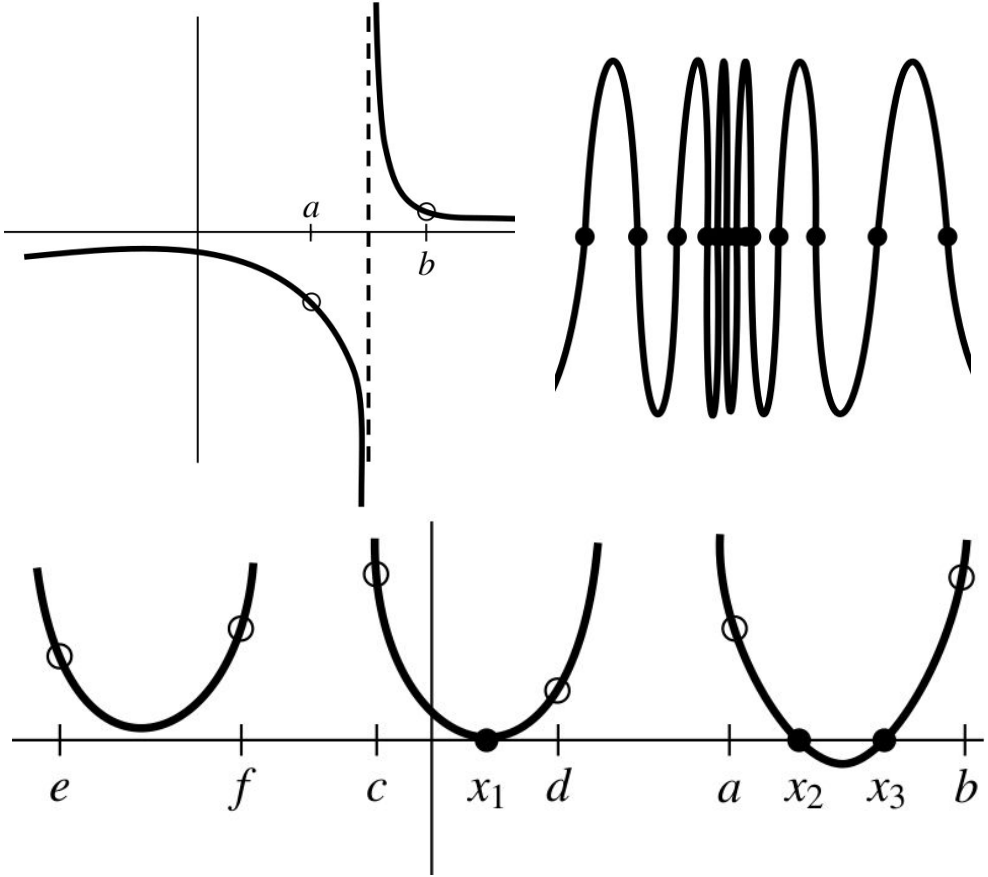
$$N = \log_2(\varepsilon_0/\varepsilon)$$



1. Bisection method

Problems and limitations

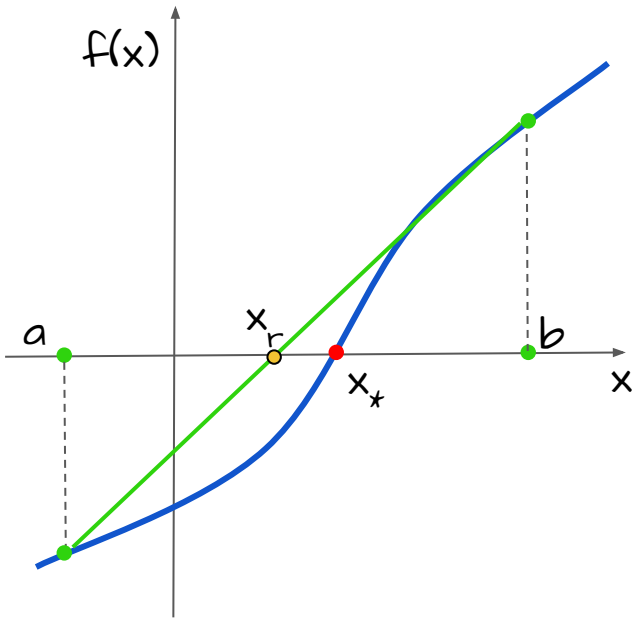
- you need two initial guesses that bracket the root
- if your interval contains more than one root or a discontinuity then you are in trouble…
  (BUT: if the interval contains two or more roots, bisection will find one of them, and if it contains a singularity, it will converge on it)
- slow convergence: each iteration reduces the error by a factor of 2



2. Secant method

If a function is continuous in the interval *[a,b]* and *f(a)·f(b) < 0*, then there will be a straight line connecting the points *(a,f(a))* and *(b,f(b))*, described by the equation:

$$y(x) = f(a) + \frac{f(a) - f(b)}{a - b}(x - a)$$



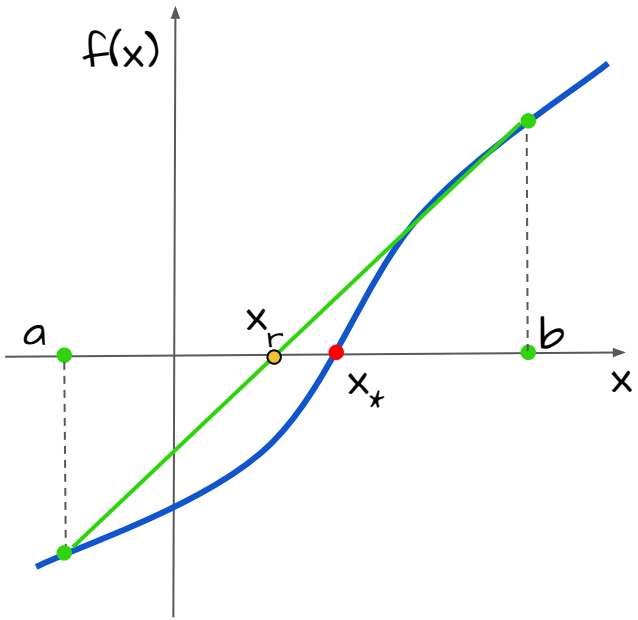
2. Secant method

If a function is continuous in the interval *[a,b]* and *f(a)·f(b) < 0*, then there will be a straight line connecting the points *(a,f(a))* and *(b,f(b))*, described by the equation:

$$y(x) = f(a) + \frac{f(a) - f(b)}{a - b}(x - a)$$

which crosses the x-axis in a point x_r [i.e., $y(x_r) = 0$], given by:

$$x_r = \frac{bf(a) - af(b)}{f(a) - f(b)} = b - \frac{f(b)}{f(b) - f(a)}(b - a)$$



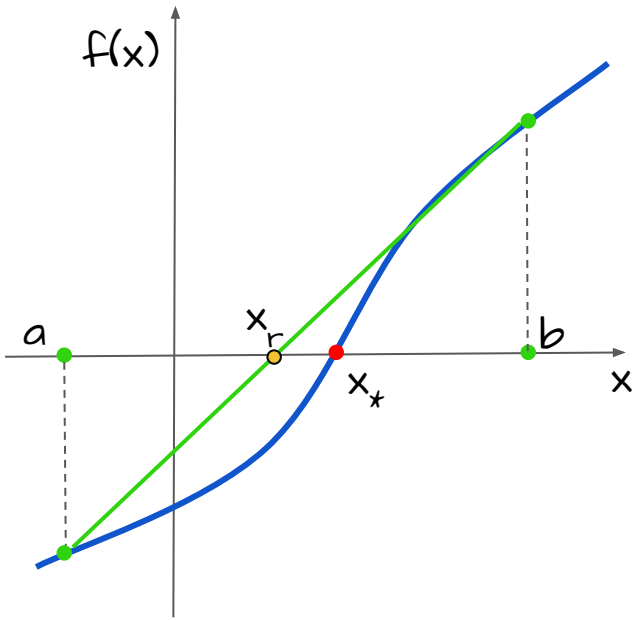
2. Secant method

If a function is continuous in the interval *[a,b]* and *f(a)·f(b) < 0*, then there will be a straight line connecting the points *(a,f(a))* and *(b,f(b))*, described by the equation:

$$y(x) = f(a) + \frac{f(a) - f(b)}{a - b}(x - a)$$

which crosses the x-axis in a point x_r [i.e., $y(x_r) = 0$], given by:

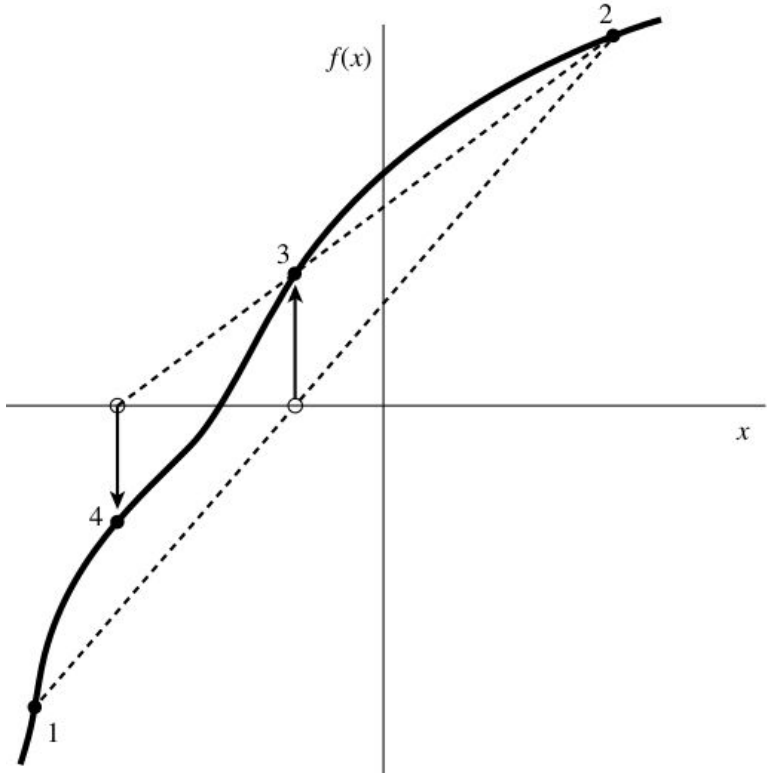
$$x_r = \frac{bf(a) - af(b)}{f(a) - f(b)} = b - \frac{f(b)}{f(b) - f(a)}(b - a)$$



With this method, we substitute the actual function with a straight line and we assume that the point where this line crosses the *x*-axis is a first approximation to the root of the equation.

2. Secant method

There are various options for the choice of the following estimate of the root.

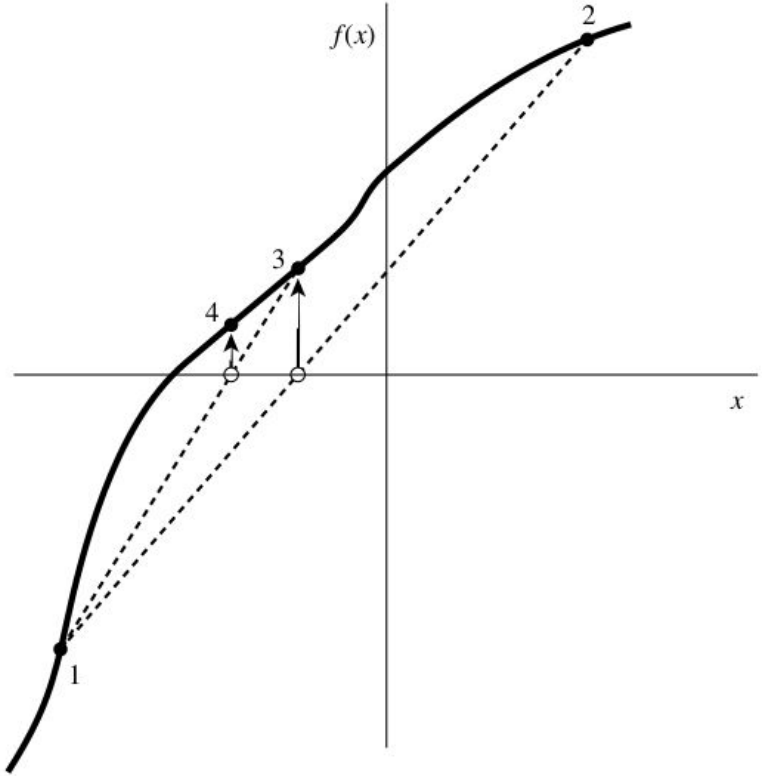


One option is to always retain the most recent evaluation, and to discard the oldest one (with an arbitrary choice in the first iteration):

- start from x_1 and x_2 and find x_3
- discard x_1, and use x_2 and x_3 to find x_4
- discard x_2, and use x_3 and x_4 to find x_5...

2. Secant method

There are various options for the choice of the following estimate of the root.



Another option is to always retain points bracketing the root:

- start from x_1 and x_2 and find x_3
- if $f(x_1)\cdot f(x_3) < 0$, use x_1 and x_3 to find x_4 (otherwise use x_2 and x_3 to find x_4)
- if $f(x_1)\cdot f(x_4) < 0$, use x_1 and x_4 to find x_5…

This is actually called *"false position method"*.

2. Secant method

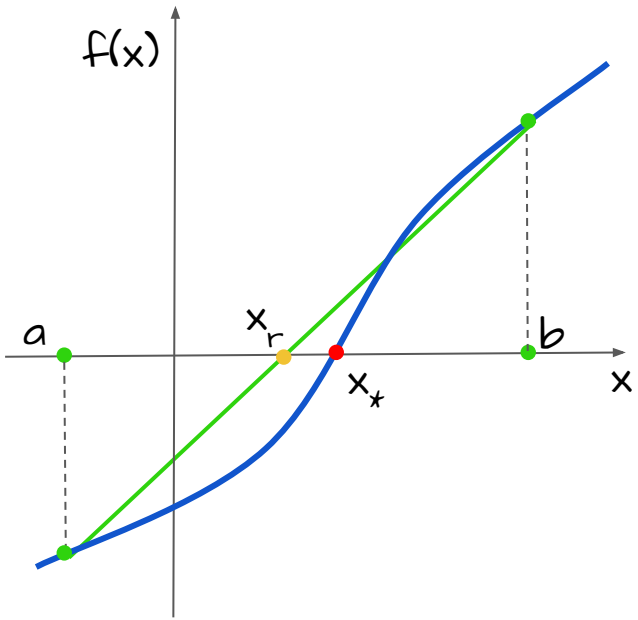
Error and convergence

If the error is defined as $\varepsilon_i = |x_* - x_i|$, the convergence of this method is

$$\varepsilon_{i+1} = K\,\varepsilon_i^{1.618}$$

This is a superlinear convergence, meaning that the *secant method* converges more quickly than the *bisection method*.

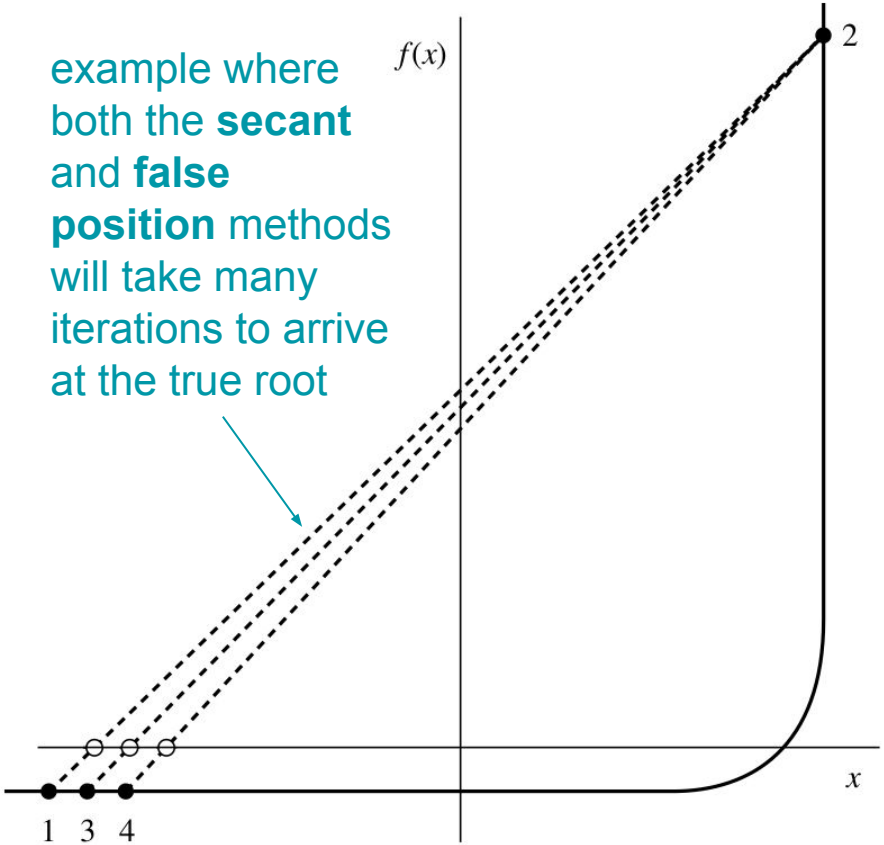
Attention: the *false position method* converges more slowly, because it sometimes retains the older function evaluation instead of the new one.



2. Secant method

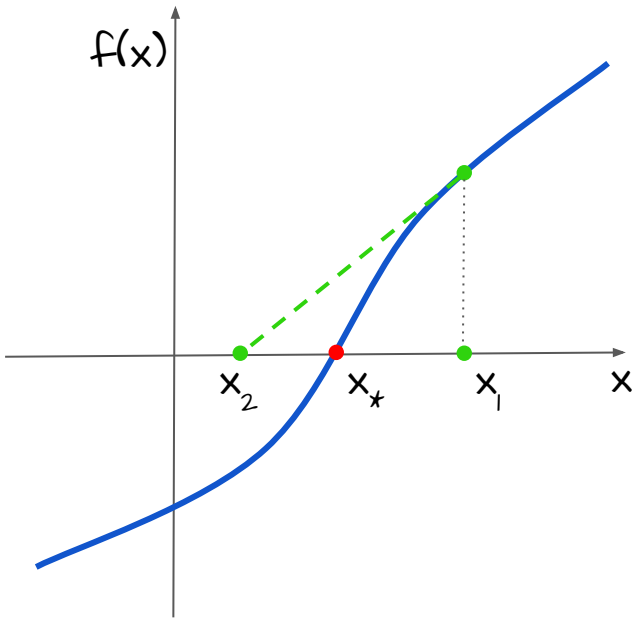
Problems and limitations

- you need two initial guesses (for false position method, they necessarily have to bracket the root)
- with the secant method, the root does not necessarily remain bracketed
- for functions that are not sufficiently continuous, the algorithm is **NOT** guaranteed to converge: local behavior might send it off to infinity



3. Newton-Raphson method

This method is different from the ones we discussed so far, because it requires the computation of both the function and its derivative at each step.



Geometrically, it extends the tangent line at the current point x_1 until it crosses zero, and it takes the abscissa of this crossing as the next guess, x_2.

3. Newton-Raphson method

Algebraically, this method comes from the Taylor series expansion of a function in the vicinity of a point:

$$f(x+\delta) \approx f(x) + \delta f'(x) + \frac{\delta^2}{2} f''(x) + \ldots$$

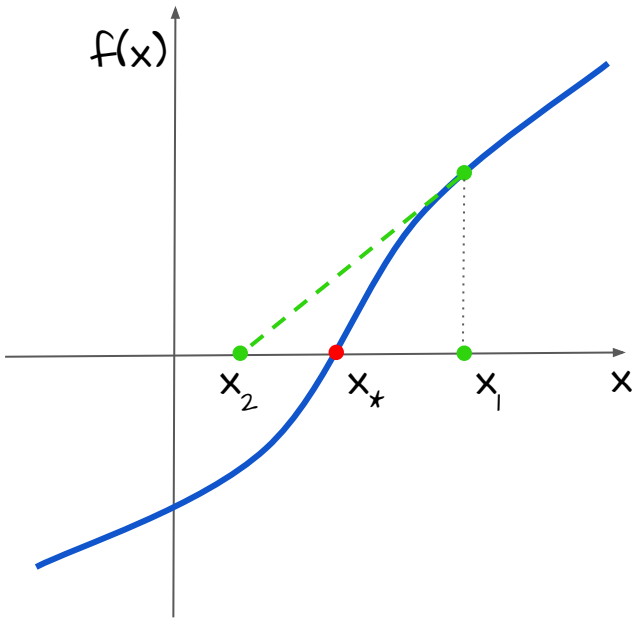
For small enough values of δ, the terms beyond linear are unimportant, and

$$f(x+\delta) = 0$$

implies:

$$\delta = -\frac{f(x)}{f'(x)}$$

The iteration is stopped when $|x_{i+1} - x_i| < \varepsilon$ or when $|f(x_i)| < \varepsilon$.



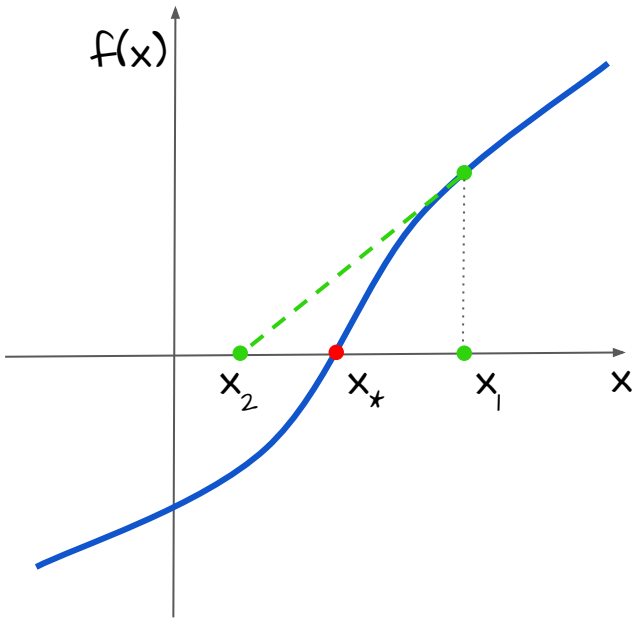
3. Newton-Raphson method

Error and convergence

This method converges quadratically:

$$\varepsilon_{i+1} = -\varepsilon_i^2 \frac{f''(x)}{2f'(x)}$$

and is therefore the preferred method for any function whose derivative can be computed efficiently.



3. Newton-Raphson method

Error and convergence

This method converges quadratically:

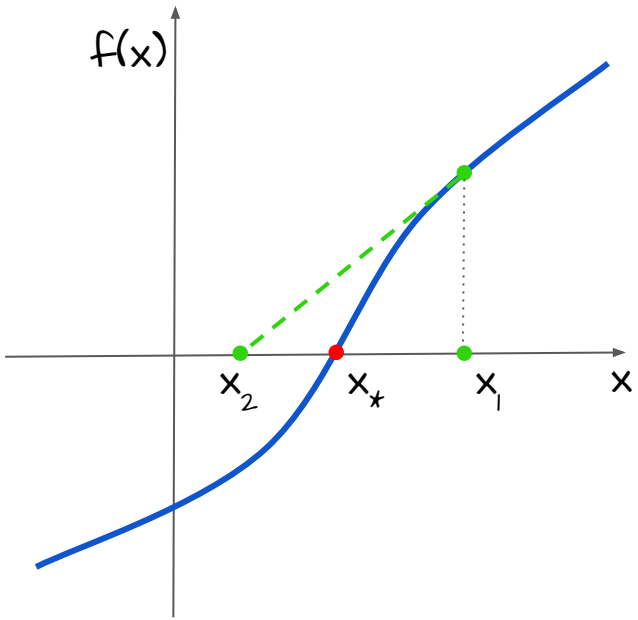
$$\varepsilon_{i+1} = -\varepsilon_i^2 \frac{f''(x)}{2f'(x)}$$

and is therefore the preferred method for any function whose derivative can be computed efficiently.

The derivative can also be computed numerically (not recommended):

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx}$$

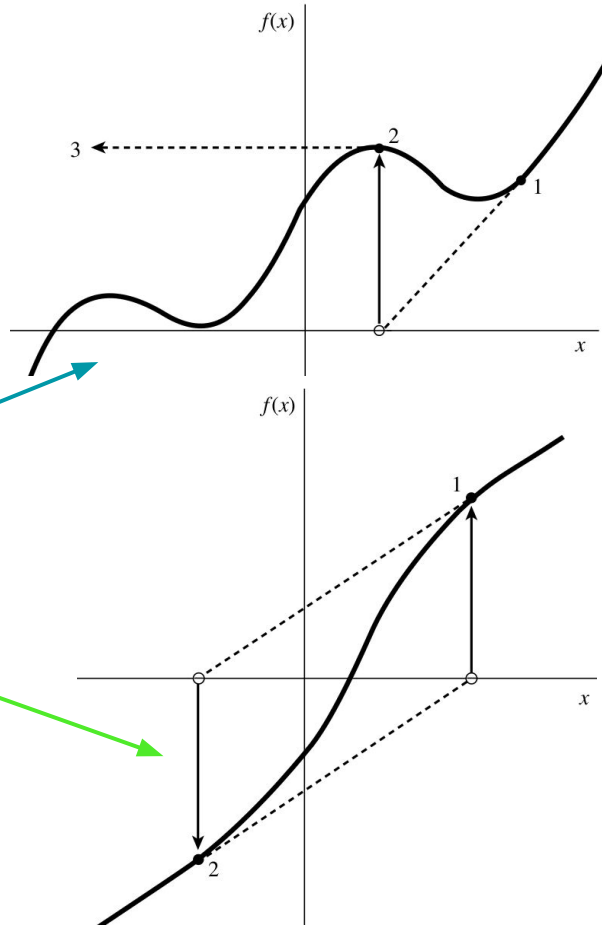
but in this case the superlinear order of convergence is at best only $\sqrt{2}$.



3. Newton-Raphson method

Problems and limitations

- you need to know the derivative of your function (you could compute it numerically, but you will lose efficiency)
- you only need one initial guess, but it needs to be close to the root you are looking for (far from a root, where the higher-order terms in the series are important, you can have troubles with this method!)
- it can be destructive if used in inappropriate circumstances
- for it to converge, *f'(x)* must exist and be non zero near the root, and *f''(x)* must be finite



3. Newton-Raphson method

Starting values

It is interesting to investigate the set of starting values for which this method does or does not converge to a root.

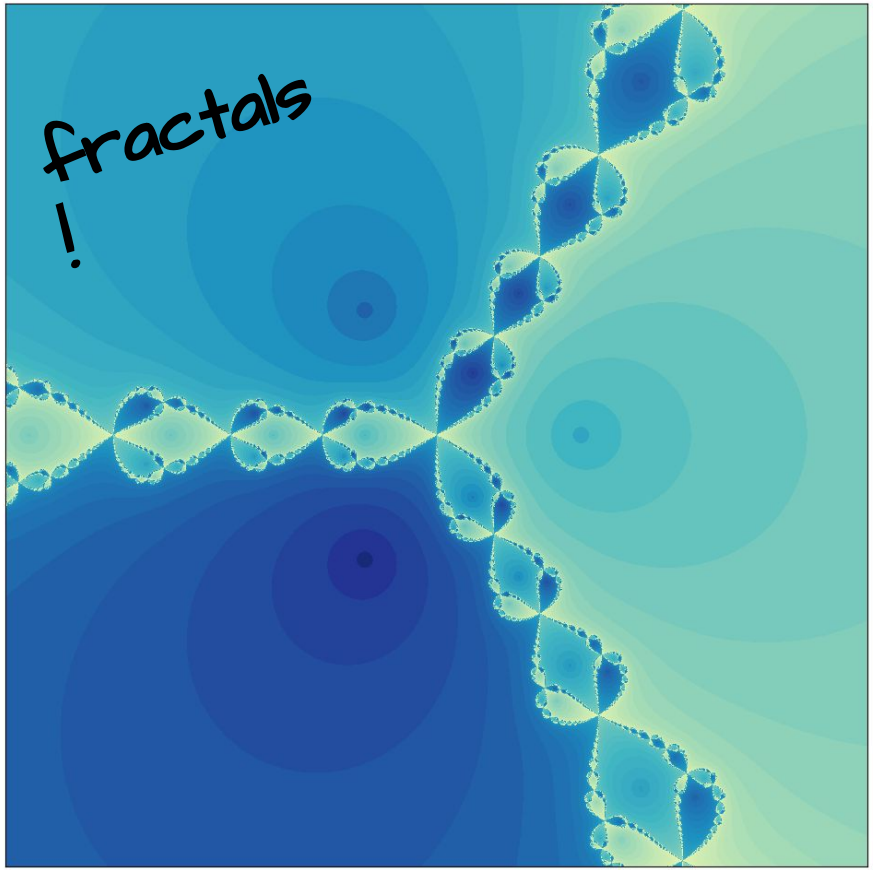
As an example, consider this equation:

$$z^3 - 1 = 0$$

With the Newton-Raphson method, the iteration is:

$$z_{i+1} = z_i - \frac{z_i^3 - 1}{3z_i^2}$$

The equation above has 1 real solution and 2 complex solutions. What happens if we use a starting point in the complex plane?



Newton-Raphson method in python

scipy.optimize.

newton

newton(*func*, *x0*, *fprime*=*None*, *args*=*()*, *tol*=*1.48e-08*, *maxiter*=*50*, *fprime2*=*None*, *x1*=*None*, *rtol*=*0.0*, *full_output*=*False*, *disp*=*True*) [source]

Find a root of a real or complex function using the Newton-Raphson (or secant or Halley's) method.

Find a root of the scalar-valued function *func* given a nearby scalar starting point *x0*. The Newton-Raphson method is used if the derivative *fprime* of *func* is provided, otherwise the secant method is used. If the second order derivative *fprime2* of *func* is also provided, then Halley's method is used.

If *x0* is a sequence with more than one item, `newton` returns an array: the roots of the function from each (scalar) starting point in *x0*. In this case, *func* must be vectorized to return a sequence or array of the same shape as its first argument. If *fprime* (*fprime2*) is given, then its return must also have the same shape: each element is the first (second) derivative of *func* with respect to its only variable evaluated at each element of its first argument.

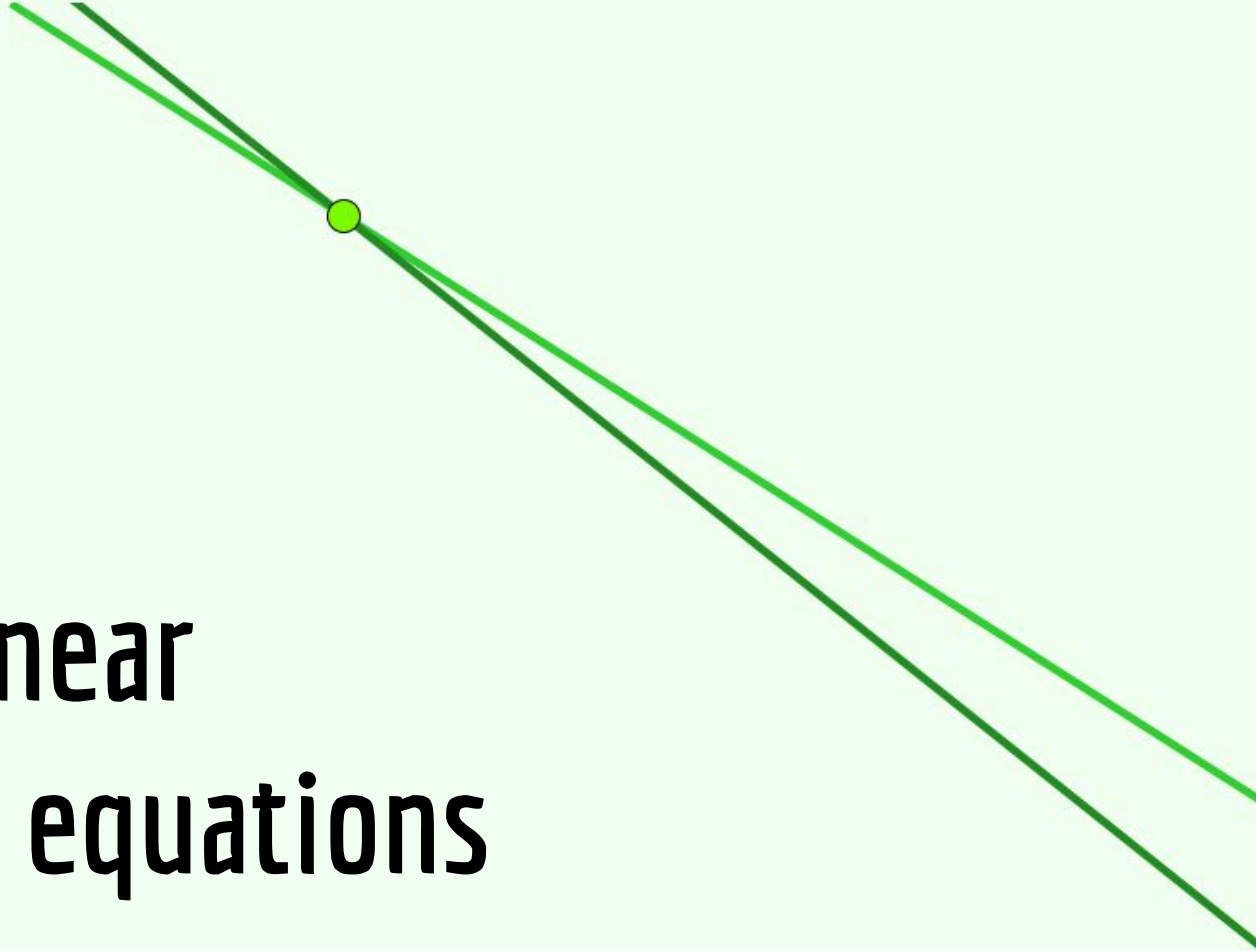
`newton` is for finding roots of a scalar-valued functions of a single variable. For problems involving several variables, see `root`.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html

One of the parameters:

fprime (callable, optional)

The derivative of the function when available and convenient. If it is None (default), then the secant method is used.



Sets of linear algebraic equations

Set of linear algebraic equations

In a set of linear algebraic equations, the N unknowns x_j (j = 1, 2, … , N) are related by M equations. The coefficients a_{ij} (with i = 1, 2, …, M and j = 1, 2, …, N) are known numbers, as are the right-hand side quantities b_i (i = 1, 2, … , M).

M equations

$$\left\{\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N & = & b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N & = & b_3 \\ \cdots & & \cdots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N & = & b_M \end{array}\right.$$

N unknowns

Set of linear algebraic equations

If $N = M$: there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j.

Set of linear algebraic equations

If $N = M$: there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j.

Analytically, there can be no unique solution if one or more of the M equations is a linear combination of the others (*row degeneracy*), or if all equations contain certain variables only in exactly the same linear combination (*column degeneracy*). A set of equations that is degenerate is called *singular*.

Set of linear algebraic equations

If $N = M$: there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j.

Analytically, there can be no unique solution if one or more of the M equations is a linear combination of the others (*row degeneracy*), or if all equations contain certain variables only in exactly the same linear combination (*column degeneracy*). A set of equations that is degenerate is called *singular*.

Numerically, at least two additional things can go wrong:

- Some of the equations may be so close to linearly dependent that roundoff errors render them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail.
- Accumulated roundoff errors in the solution process can swamp the true solution (especially if N is too large). The numerical procedure does not fail, but it returns a set of wrong x_j (direct substitution into original equations).

Set of linear algebraic equations

We can rewrite our equations in matrix form as:

$$\boxed{\mathbf{A}} \cdot \boxed{\mathbf{x}} = \boxed{\mathbf{b}}$$

where **A** is the matrix of coefficients and **b** is the right-hand side (as column vector)

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ & \dots & & \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{bmatrix}$$

LU decomposition

Suppose we are able to write the matrix **A** as a product of two matrices, where **L** is ***lower triangular*** (has elements only on the diagonal and below) and **U** is ***upper triangular*** (has elements only on the diagonal and above):

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

For the case of a 4 × 4 matrix **A**, for example, we would have:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

LU decomposition

We can use a decomposition like this:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b}$$

By first solving for the vector **y** such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

and then, finally, solving for the vector **x**

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

LU decomposition

The advantage of breaking up one linear set into two successive ones is that the solution of a triangular set of equations is quite trivial. Our first equation

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

can be solved by ***forward substitution*** as follows:

$$y_1 = \frac{b_1}{\alpha_{11}}$$

$$y_i = \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N$$

LU decomposition

And our second equation

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

can then be solved by ***back-substitution***:

$$x_N = \frac{y_N}{\beta_{NN}}$$

$$x_i = \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1$$

Notice that, once we have the **LU** decomposition of **A**, we can solve the equation with as many right-hand sides as we then care to, one at a time.

LU decomposition in python

scipy.linalg.lu_factor

```
scipy.linalg.lu_factor(a, 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*

Matrix to decompose

overwrite_a : *bool, optional*

Whether to overwrite data in A (may increase performance)

check_finite : *bool, 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:

lu : *(M, N) ndarray*

Matrix containing U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored.

piv : *(N,) ndarray*

Pivot indices representing the permutation matrix P: row i of matrix was interchanged with row piv[i].

https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html#scipy.linalg.lu_factor

This function can be used to carry out a LU decomposition of a matrix.

LU decomposition in python

scipy.linalg.lu_solve

```
scipy.linalg.lu_solve(lu_and_piv, b, trans=0, overwrite_b=False,
check_finite=True)
```
[source]

Solve an equation system, a x = b, given the LU factorization of a

Parameters: **(lu, piv)**

Factorization of the coefficient matrix a, as given by lu_factor

b : *array*

Right-hand side

trans : *{0, 1, 2}, optional*

Type of system to solve:

trans	system
0	a x = b
1	a^T x = b
2	a^H x = b

overwrite_b : *bool, optional*

Whether to overwrite data in b (may increase performance)

check_finite : *bool, optional*

Whether to check that the input matrices contain 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: **x** : *array*

Solution to the system

https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html

This function can be used to solve a set of linear equations expressed in matrix form, and for which the LU decomposition has been carried out.

(Also see the version scipy.linalg-solve: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html)

Solving linear equations in Python

numpy.linalg.solve

`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*.

Parameters:

a : *(..., M, M) array_like*
Coefficient matrix.

b : *{(M,), (..., M, K)}, array_like*
Ordinate or "dependent variable" values.

Returns:

x : *{(..., M,), (..., M, K)} ndarray*
Solution to the system a x = b. Returned shape is (..., M) if b is shape (M,) and (..., M, K) if b is (..., M, K), where the "..." part is broadcasted between a and b.

Raises:

https://numpy.org/doc/2.3/reference/generated/numpy.linalg.solve.html

This function solves a linear matrix equation, or system of linear scalar equations.

Solving linear equations in Python

scipy.linalg.lu_solve

```
scipy.linalg.lu_solve(lu_and_piv, b, trans=0, overwrite_b=False,
check_finite=True)                                                [source]
```

Solve an equation system, a x = b, given the LU factorization of a

Parameters: **(lu, piv)**

Factorization of the coefficient matrix a, as given by lu_factor

b : *array*

Right-hand side

trans : *{0, 1, 2}, optional*

Type of system to solve:

trans	system
0	a x = b
1	a^T x = b
2	a^H x = b

overwrite_b : *bool, optional*

Whether to overwrite data in b (may increase performance)

check_finite : *bool, optional*

Whether to check that the input matrices contain 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: **x** : *array*

Solution to the system

https://numpy.org/doc/2.3/reference/generated/numpy.linalg.solve.html

This function solves a linear matrix equation, or system of linear scalar equations.

More solutions for linear equations / working with matrixes

scipy.linalg.

solve

```
solve(a, b, lower=False, overwrite_a=False, overwrite_b=False,
check_finite=True, assume_a=None, transposed=False)
```
[source]

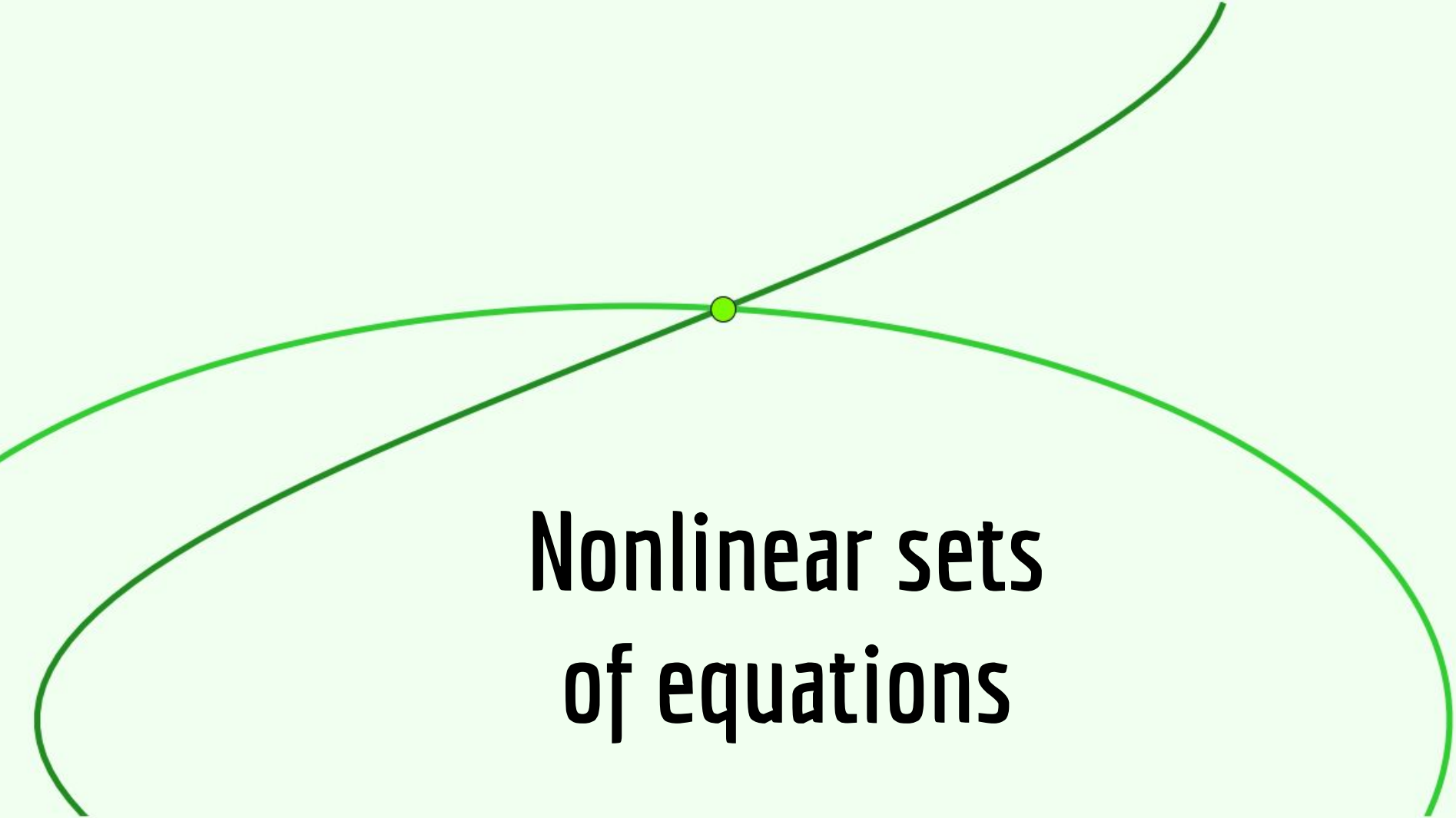
Solve the equation `a @ x = b` for `x`, where *a* is a square matrix.

If the data matrix is known to be a particular type then supplying the corresponding string to `assume_a` key chooses the dedicated solver. The available options are

diagonal	'diagonal'
tridiagonal	'tridiagonal'
banded	'banded'
upper triangular	'upper triangular'
lower triangular	'lower triangular'
symmetric	'symmetric' (or 'sym')
hermitian	'hermitian' (or 'her')
symmetric positive definite	'positive definite' (or 'pos')
general	'general' (or 'gen')

https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html

This function solves the equation a @ x = b for x, where a is a square matrix.a linear matrix equation, or system of linear scalar equations.



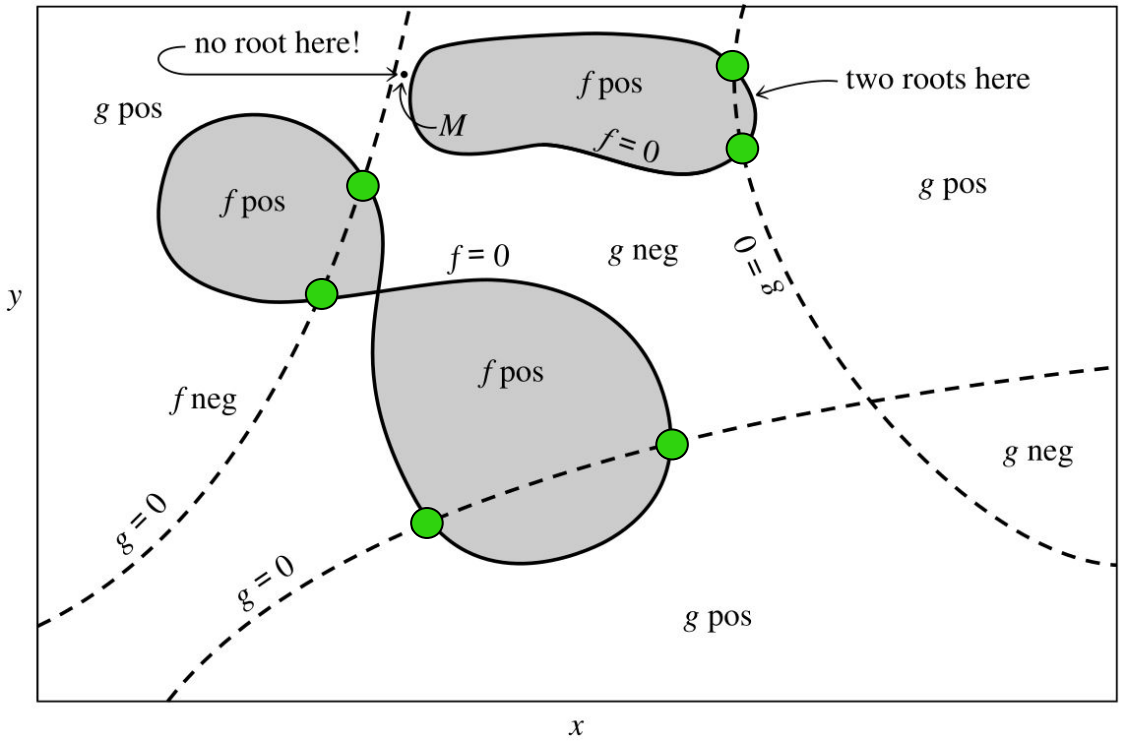
Nonlinear sets of equations

Nonlinear Systems of Equations

Let's consider here, for simplicity, the case of two dimensions, where we want to solve simultaneously:

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases}$$

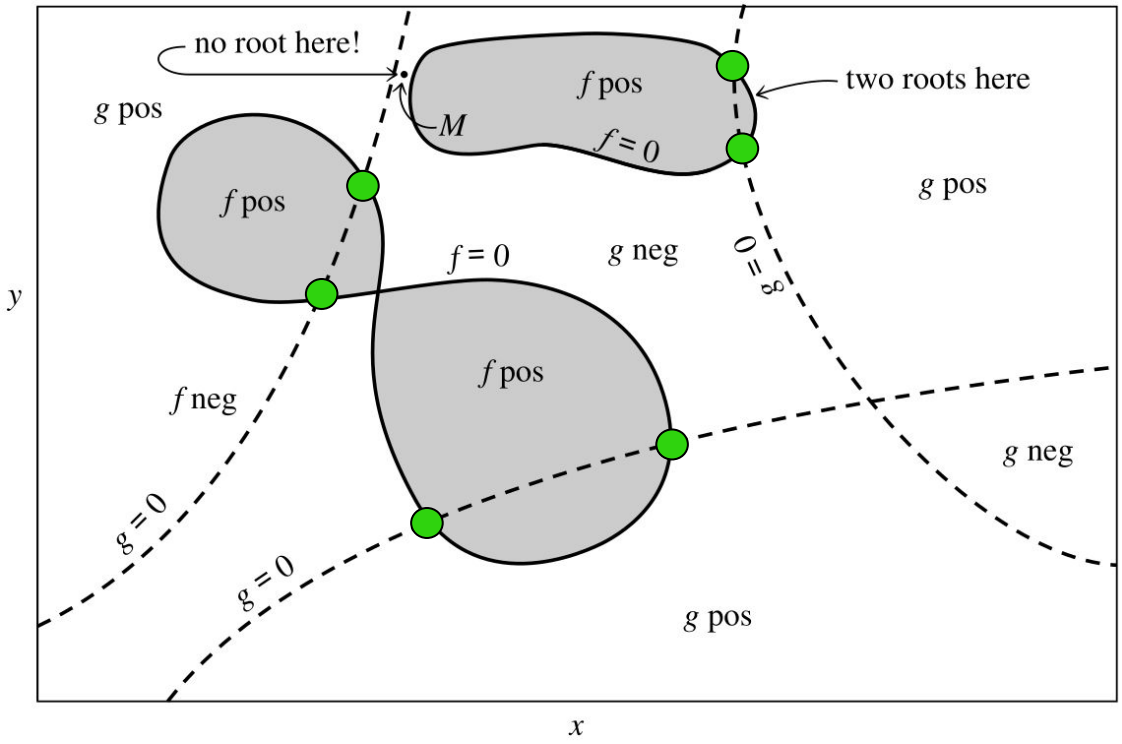
The main problem we will have to face is that ***there are no good, general methods*** for solving systems of more than one nonlinear equation!



Nonlinear Systems of Equations

The functions *f* and *g* are two arbitrary functions, each having ***zero contour lines*** that divide the *(x,y)* plane into regions where their respective function is positive or negative. We are interested into the zero contour boundaries, and we look for the ***points*** (if any) that are common to the zero contours of *f* and *g*.

Unfortunately, the functions *f* and *g* have, in general, no relation to each other at all!



Newton-Raphson Method for Nonlinear Systems of Equations

We consider here a typical problem, where we have *N* functional relations to be zeroed, involving variables x_i ($i = 1, 2, \ldots, N$):

$$F_i(x_1, x_2, \ldots, x_N) = 0 \qquad i = 1, 2, \ldots, N$$

We indicate with **x** the vector of values x_i and with **F** the vector of functions F_i. In the neighborhood of **x**, each of the functions F_i can be expanded in Taylor series:

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \boxed{\frac{\partial F_i}{\partial x_j}} \delta x_j + O(\delta\mathbf{x}^2)$$

Jacobian matrix **J** $\longrightarrow$ $J_{ij} \equiv \dfrac{\partial F_i}{\partial x_j}$

Newton-Raphson Method for Nonlinear Systems of Equations

In matrix notation, we can write this as:

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2)$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, we obtain a set of linear equations for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely:

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}$$

This is a matrix equation with the same form of those we have seen previously, and can be solved by LU decomposition!
The corrections are then added to the solutions vector:

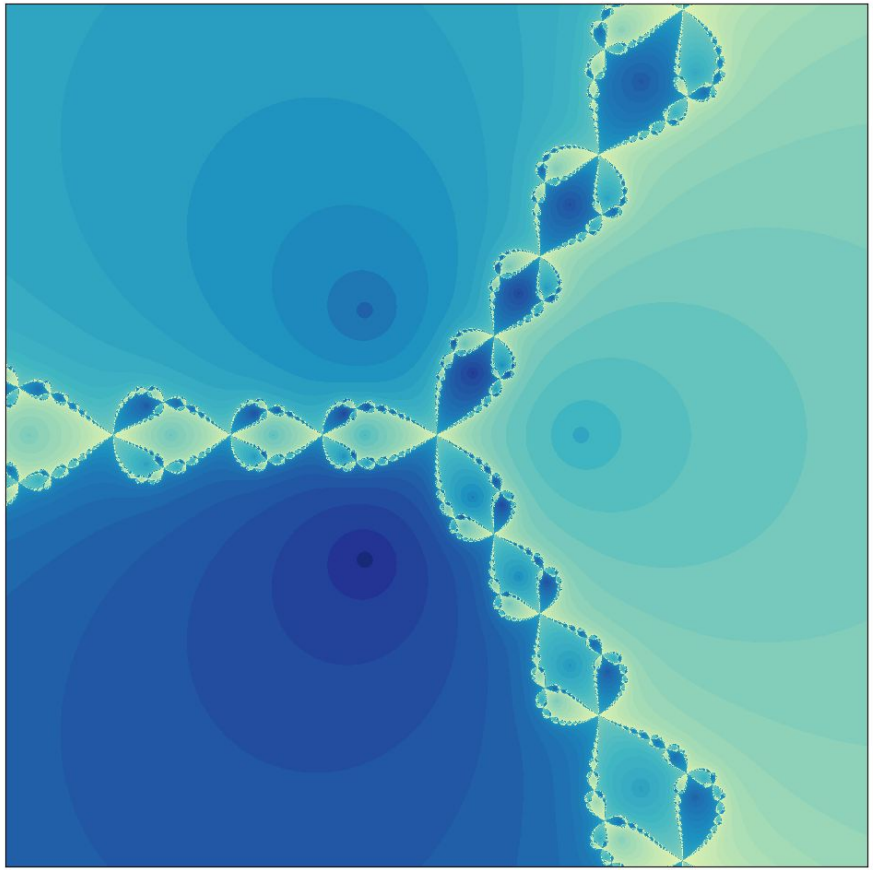
$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x}$$

and the process is iterated to convergence.

Newton-Raphson Method for Nonlinear Systems of Equations

Newton's method for solving nonlinear equations has an unfortunate tendency to wander off if the initial guess is not sufficiently close to the root, and is therefore not guaranteed to succeed.

A *global* method is one that converges to a solution from almost any starting point. We can develop an algorithm that combines the rapid local convergence of Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration.



Newton-Raphson Method for Nonlinear Systems of Equations

How do we decide whether to accept the Newton step $\delta\mathbf{x}$?
A reasonable strategy is to require that the step decrease $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$. This is the same requirement we would impose if we were trying to minimize

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$$

We note that the Newton step is a *descent direction* for *f*:

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0$$

Thus our strategy is quite simple: we always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence.
However, we check at each iteration that the proposed step reduces *f*.
If not, we *backtrack* along the Newton direction until we have an acceptable step.
Because the Newton step is a descent direction for f, we are guaranteed to find an acceptable step by backtracking.

Multidimensional Secant Methods: Broyden's Method

Newton's method as implemented above is quite powerful, but it still has several disadvantages. One drawback is that the Jacobian matrix is needed. In many problems analytic derivatives are unavailable. If function evaluation is expensive, then the cost of finite-difference determination of the Jacobian can be prohibitive.

There are quasi-Newton methods that provide cheap approximations to the Jacobian for zero finding. These methods are often called *secant methods*, since they reduce to the secant method in one dimension. The best of these methods still seems to be the first one introduced, ***Broyden's method***.

Like the secant method in one dimension, Broyden's method *converges superlinearly once you get close enough to the root*. Embedded in a global strategy, it is almost as robust as Newton's method, and often needs far fewer function evaluations to determine a zero. The final value of the approximate Jacobian is not always close to the true Jacobian at the root, even when the method converges.

Solving nonlinear sets of equations with python

scipy.optimize.newton_krylov

```
scipy.optimize.newton_krylov(F, xin, iter=None, rdiff=None, method='lgmres',
inner_maxiter=20, inner_M=None, outer_k=10, verbose=False, maxiter=None,
f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, tol_norm=None,
line_search='armijo', callback=None, **kw)
```

Find a root of a function, using Krylov approximation for inverse Jacobian.

This method is suitable for solving large-scale problems.

Parameters: **F** : *function(x) -> f*

Function whose root to find; should take and return an array-like object.

xin : *array_like*

Initial guess for the solution

rdiff : *float, optional*

Relative step size to use in numerical differentiation.

method : *str or callable, optional*

Krylov method to use to approximate the Jacobian. Can be a string, or a function implementing the same interface as the iterative solvers in `scipy.sparse.linalg`. If a string, needs to be one of: `'lgmres'`, `'gmres'`, `'bicgstab'`, `'cgs'`, `'minres'`, `'tfqmr'`. The default is `scipy.sparse.linalg.lgmres`.

inner_maxiter : *int, optional*

Parameter to pass to the "inner" Krylov solver: maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

inner_M : *LinearOperator or InverseJacobian*

Preconditioner for the inner Krylov iteration. Note that you can use also inverse Jacobians as (adaptive) preconditioners. For example,

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton_krylov.html#scipy.optimize.newton_krylov

Solving nonlinear sets of equations with python

scipy.optimize.broyden1

```
scipy.optimize.broyden1(F, xin, iter=None, alpha=None,
reduction_method='restart', max_rank=None, verbose=False, maxiter=None,
f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, tol_norm=None,
line_search='armijo', callback=None, **kw)
```

Find a root of a function, using Broyden's first Jacobian approximation.

This method is also known as "Broyden's good method".

Parameters: **F** : ***function(x) -> f***

Function whose root to find; should take and return an array-like object.

xin : ***array_like***

Initial guess for the solution

alpha : ***float, optional***

Initial guess for the Jacobian is `(-1/alpha)`.

reduction_method : ***str or tuple, optional***

Method used in ensuring that the rank of the Broyden matrix stays low. Can either be a string giving the name of the method, or a tuple of the form `(method, param1, param2, ...)` that gives the name of the method and values for additional parameters. Methods available:

- `restart`: drop all matrix columns. Has no extra parameters.
- `simple`: drop oldest matrix column. Has no extra parameters.
- `svd`: keep only the most significant SVD components. Takes an extra parameter, `to_retain`, which determines the number of SVD components to retain when rank reduction is done. Default is `max_rank - 2`.

max_rank : ***int, optional***

Maximum rank for the Broyden matrix. Default is infinity (i.e., no rank reduction).

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.broyden1.html

Solving nonlinear sets of equations with python

scipy.optimize.fsolve

```
scipy.optimize.fsolve(func, x0, args=(), fprime=None, full_output=0, col_deriv=0,
xtol=1.49012e-08, maxfev=0, band=None, epsfcn=None, factor=100,
diag=None)
```
[source]

Find the roots of a function.

Return the roots of the (non-linear) equations defined by `func(x) = 0` given a starting estimate.

Parameters:

func : *callable* `f(x, *args)`

A function that takes at least one (possibly vector) argument, and returns a value of the same length.

x0 : *ndarray*

The starting estimate for the roots of `func(x) = 0`.

args : *tuple, optional*

Any extra arguments to *func*.

fprime : *callable* `f(x, *args)`, *optional*

A function to compute the Jacobian of *func* with derivatives across the rows. By default, the Jacobian will be estimated.

full_output : *bool, optional*

If True, return optional outputs.

col_deriv : *bool, optional*

Specify whether the Jacobian function computes derivatives down the columns (faster, because there is no transpose operation).

xtol : *float, optional*

The calculation will terminate if the relative error between two consecutive iterates is at most *xtol*.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html#scipy.optimize.fsolve

Summary

Let's summarize

Root finding

method	pros	cons
bisection **false position** (bracketing methods)	• robust, i.e. always converge	• require initial bracketing • slow convergence
secant **Newton-Raphson** (non-bracketing methods)	• faster • no need to bracket (just need reasonable starting point)	• may not converge • for NR, need to know the derivative

Let's summarize

Linear and non-linear equations

Solving a set of linear equations: one can always find a solution.

Typical methods: LU decomposition + various tools available in Python

Solving a set of non-linear equations: no general solution available

Typical methods: Newton-Rapson method, Broyden method, etc.