

Lecture 06: Integration and Root Finding

Sergei V. Kalinin

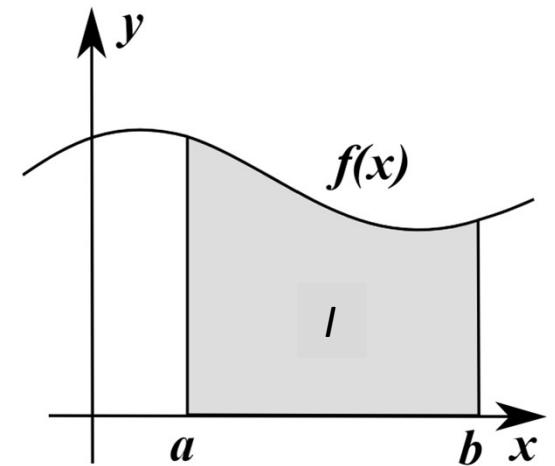
Numerical integration

Generic problem: evaluate

$$I = \int_a^b f(x)dx$$

We need numerical integration when

- Cannot/difficult integrate analytically
- Only know the integrand $f(x)$ at certain points



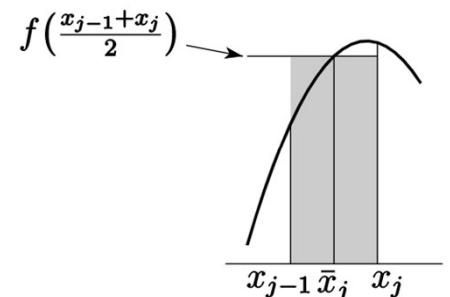
References: Chapter 5 of *Computational Physics* by Mark Newman
Chapter 4 of *Numerical Recipes Third Edition* by W.H. Press et al.

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: rectangular (midpoint) rule

Interpret the integral as the area under the curve and approximate by a rectangle evaluated at midpoint

$$\int_a^b f(x) dx \approx (b - a) f\left(\frac{a + b}{2}\right)$$



Error (from Euler-McLaurin formula):

$$\int_a^b f(x) dx - (b - a) f\left(\frac{a + b}{2}\right) \approx \frac{(b - a)^3}{24} f''(a)$$

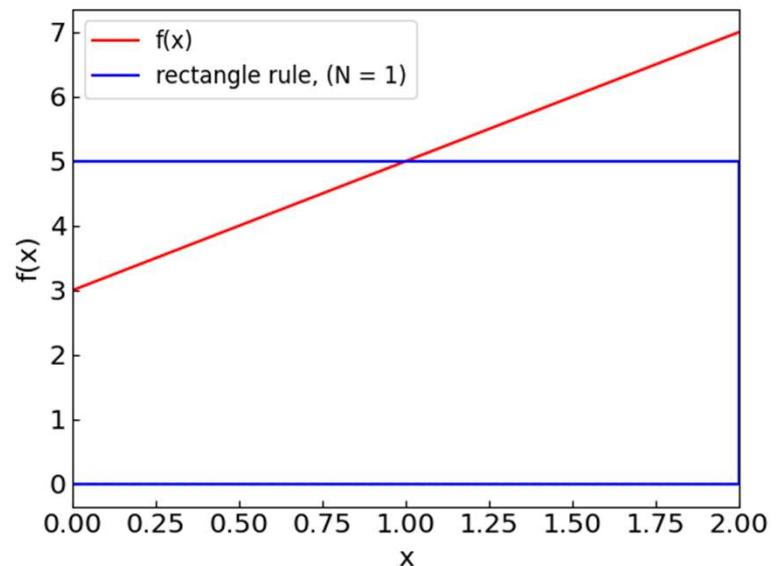
The rule is exact for the integration of linear functions

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: rectangular (midpoint) rule

Example:

$$I = \int_0^2 2x + 3 dx = 10$$



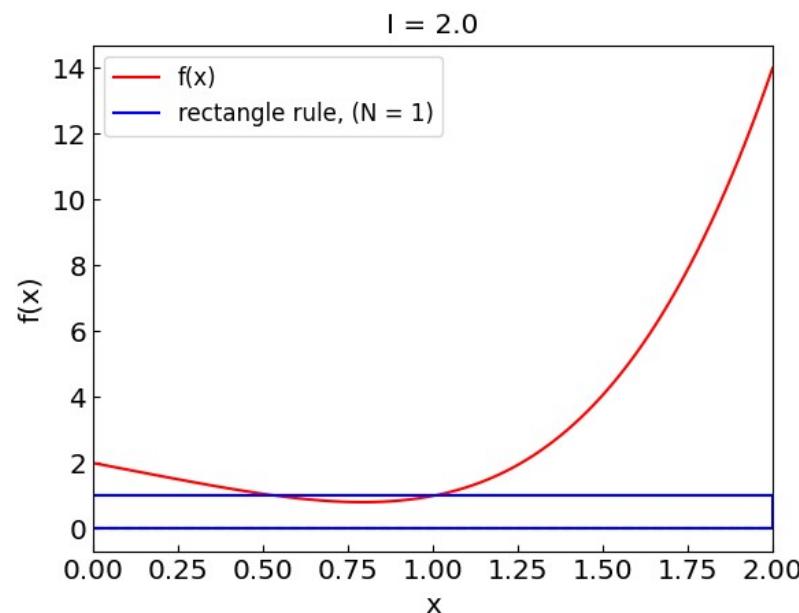
Although the rectangle is a poor approximate of the line (which is a trapezoid here), the errors cancel out

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: rectangular (midpoint) rule

Another example:

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Rectangle rule gives $I_{\text{rect}} = 2$ which is way off

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended (composite) rectangular rule

Split the integration interval into N sub-intervals and apply the rectangle rule separately to each one

$$\int_a^b f(x) \approx h \sum_{k=1}^N f(x_k), \quad k = 1, \dots, N$$

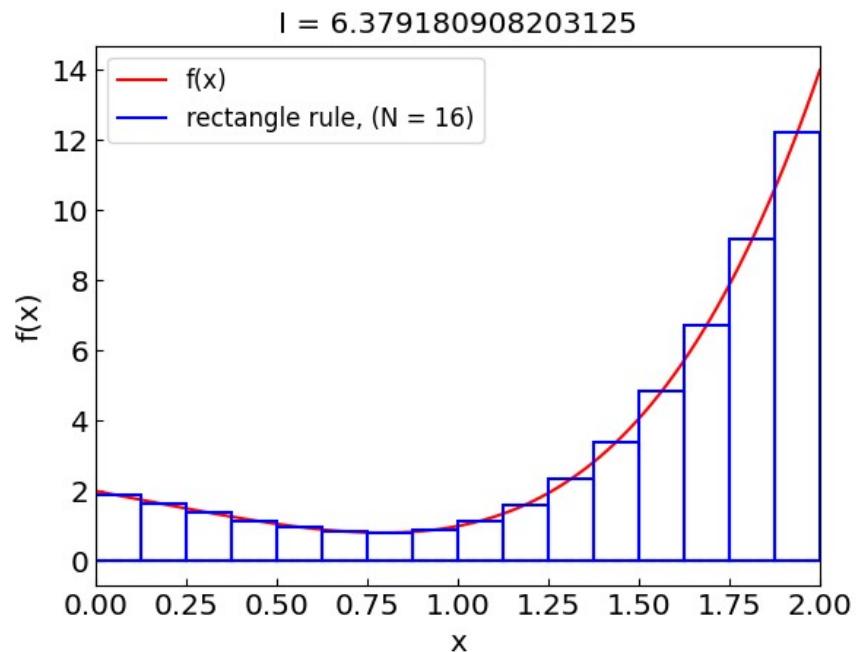
$$x_k = a + \frac{2k - 1}{2} h.$$

$$h = (b - a)/N$$

Error estimate:

$$I - I_{\text{rect}} = (b - a) \frac{h^2}{24} f''(a) + \mathcal{O}(h^4)$$

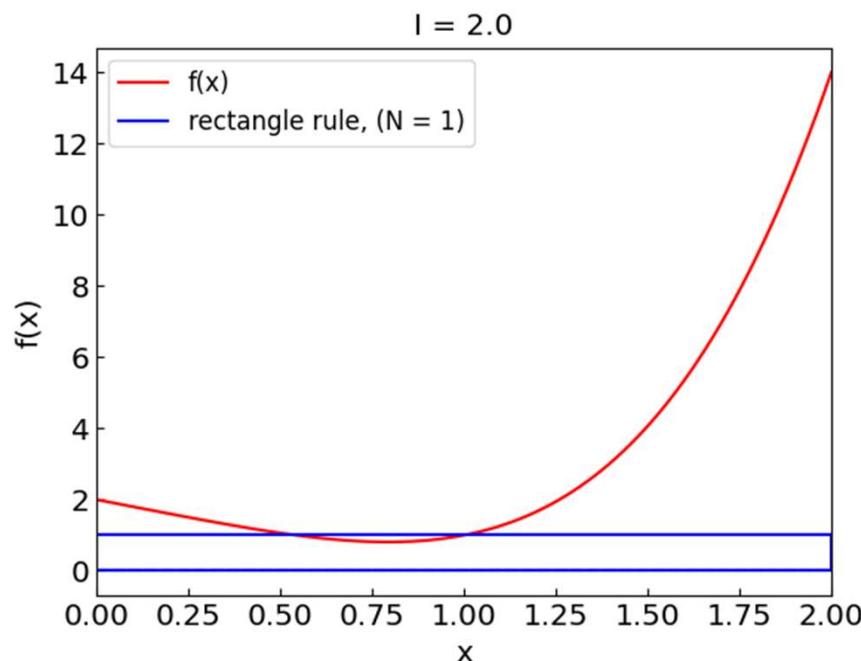
$$\int_0^2 x^4 - 2x + 2$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended (composite) rectangular rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$

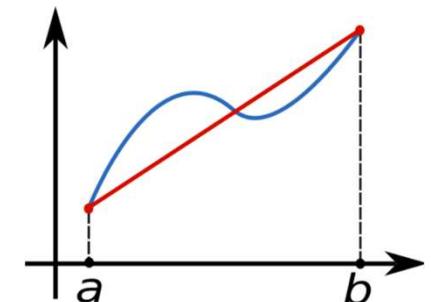


From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: trapezoidal rule

Approximate the integral by a trapezoid

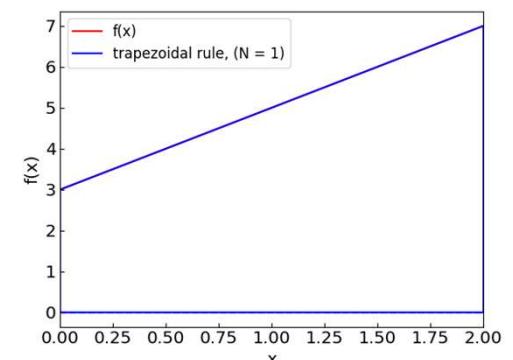
$$\int_a^b f(x) dx \approx (b - a) \frac{f(a) + f(b)}{2}$$



Error:

$$\int_a^b f(x) dx - (b - a) \frac{f(a) + f(b)}{2} \approx -\frac{(b - a)^3}{12} f''(a)$$

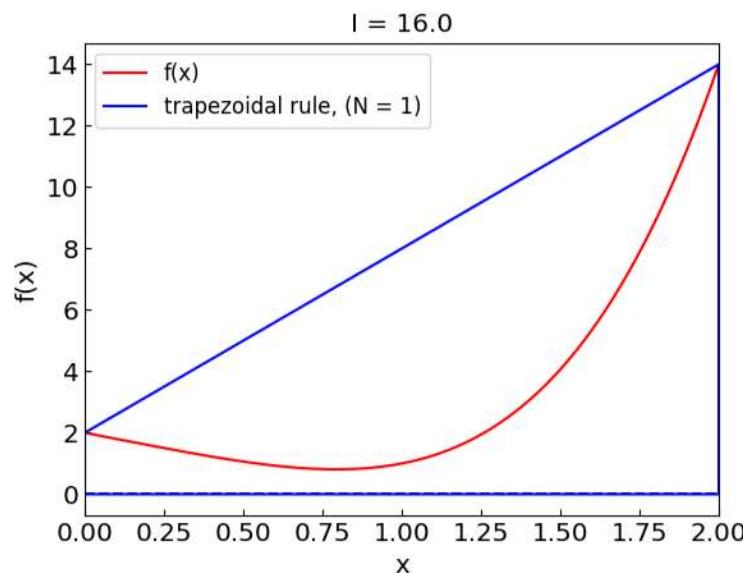
The rule is exact for the integration of linear functions



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: trapezoidal rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Trapezoidal rule gives $I_{\text{trap}} = 16$, way off and in the opposite direction relative to rectangle rule

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended trapezoidal rule

$$\int_a^b f(x) \approx h \sum_{k=0}^N \frac{f(x_k) + f(x_{k+1})}{2}, \quad i = 0, \dots, N$$

$$x_k = a + kh.$$

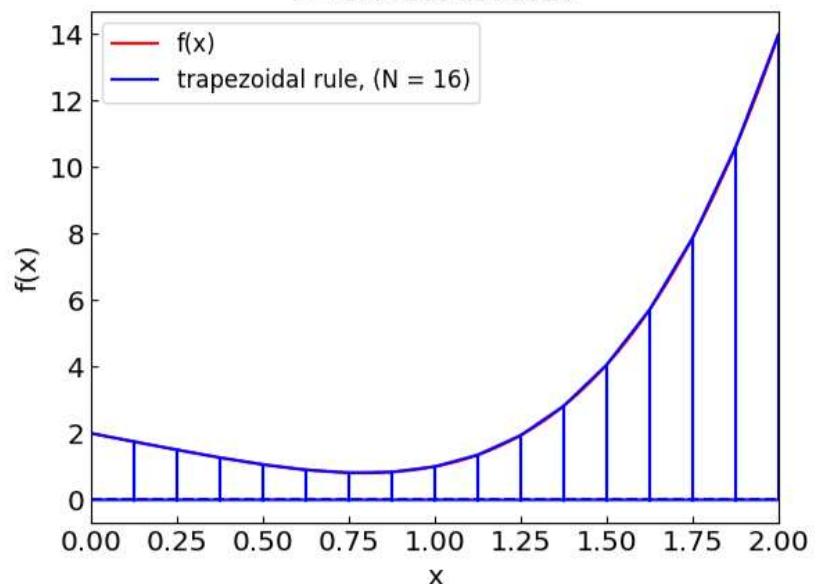
$$h = (b - a)/N$$

Error estimate:

$$I - I_{\text{trap}} = -(b - a) \frac{h^2}{12} f''(a) + \mathcal{O}(h^4)$$

$$\int_0^2 x^4 - 2x + 2$$

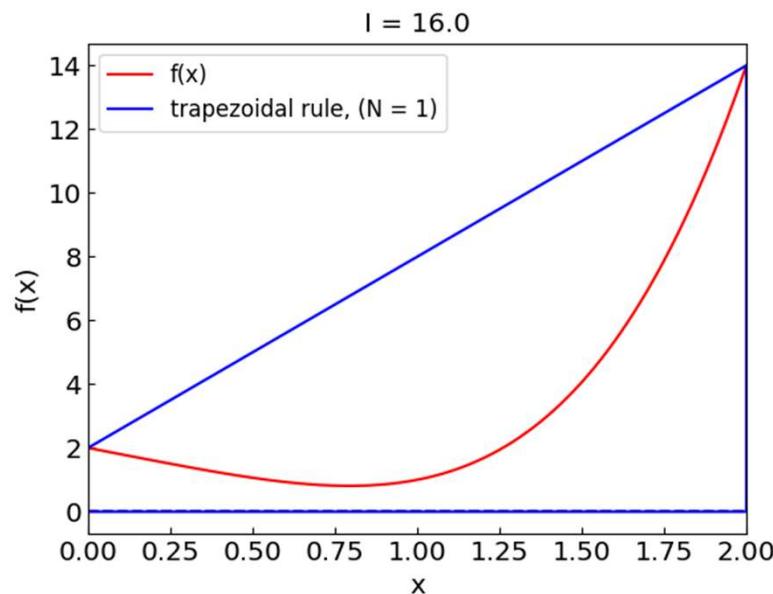
$I = 6.441650390625$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended (composite) trapezoidal rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: Simpson's rule

Recall the error estimates for rectangular and trapezoidal rules

$$I - I_{\text{rect}} = (b - a) \frac{h^2}{24} f''(a) + \mathcal{O}(h^4)$$

$$I - I_{\text{trap}} = -(b - a) \frac{h^2}{12} f''(a) + \mathcal{O}(h^4)$$

Combine them to eliminate the $\mathcal{O}(h^2)$ error term:

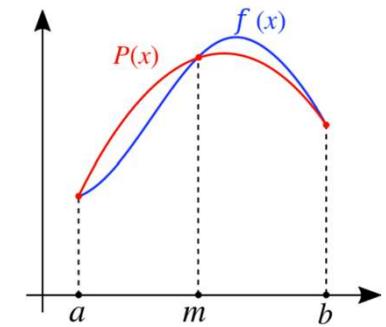
$$I_S = \frac{2I_{\text{rect}} + I_{\text{trap}}}{3}$$

i.e.

$$\int_a^b f(x) dx \approx \frac{(b - a)}{6} \left[f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right]$$

An equivalent way to obtain the rule: replace the integrand by the parabolic interpolation

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>



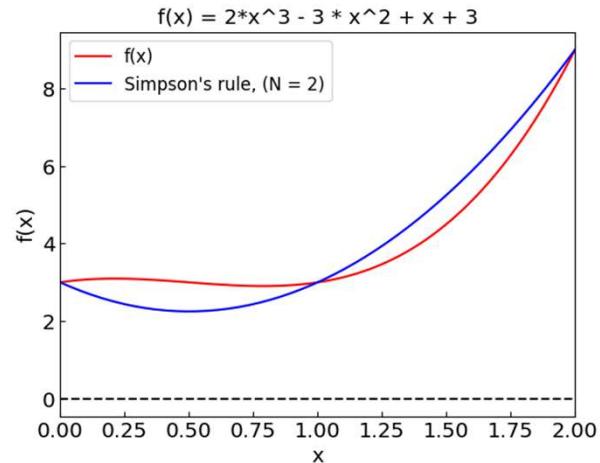
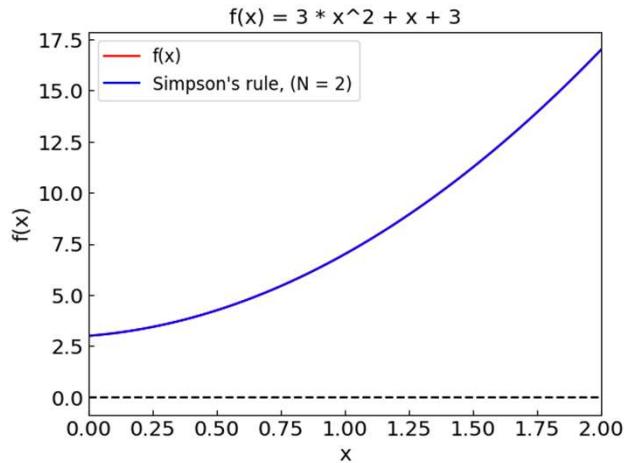
Numerical integration: Simpson's rule

$$\int_a^b f(x) dx \approx \frac{(b-a)}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

The error for the Simpson's rule is

$$I - I_S = C h^4 + \mathcal{O}(h^6)$$

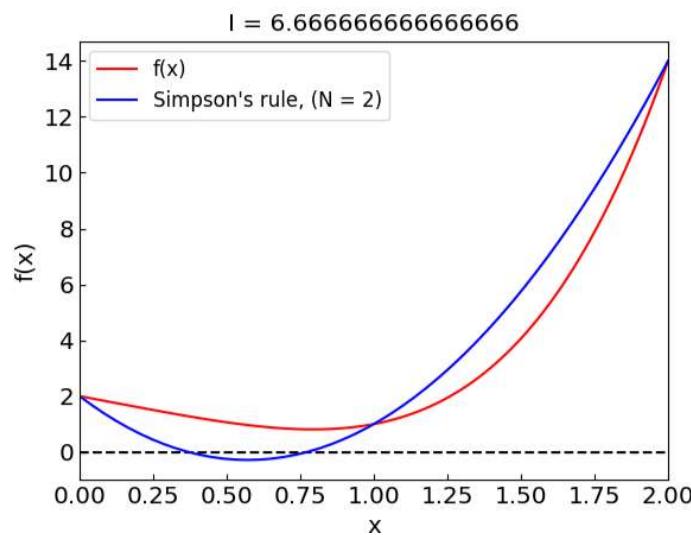
The method is exact for polynomials up to third order



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration: Simpson's rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Simpson's rule gives $I_{\text{trap}} = 6.66$ using three points, which is already not too bad!

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended Simpson's rule

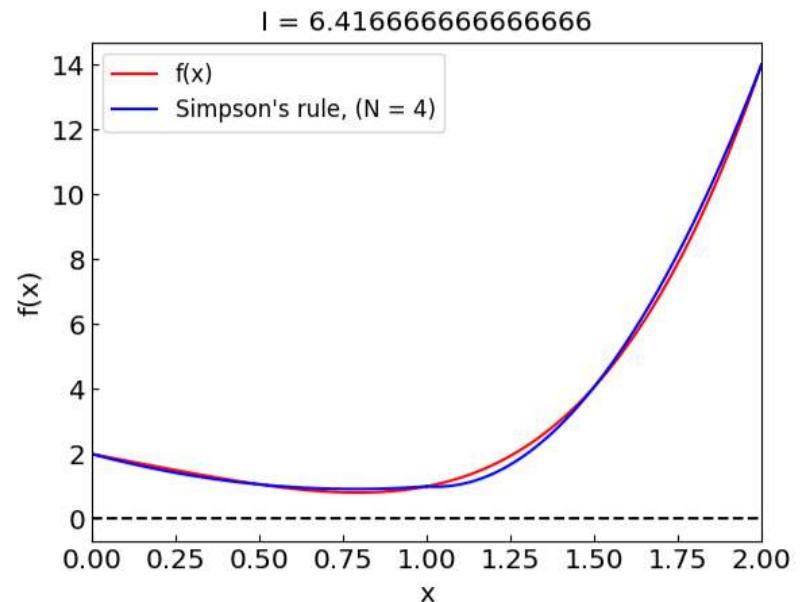
$$\int_a^b f(x) \approx \frac{h}{3} \left[f(x_0) + 4 \sum_{k=1}^{N/2} f(x_{2k-1}) + 2 \sum_{k=1}^{N/2-1} f(x_{2k}) + f(x_N) \right], \quad i = 0, \dots, N \quad \int_0^2 x^4 - 2x + 2$$

$$h = (b - a)/N$$

N must be even!

Error estimate:

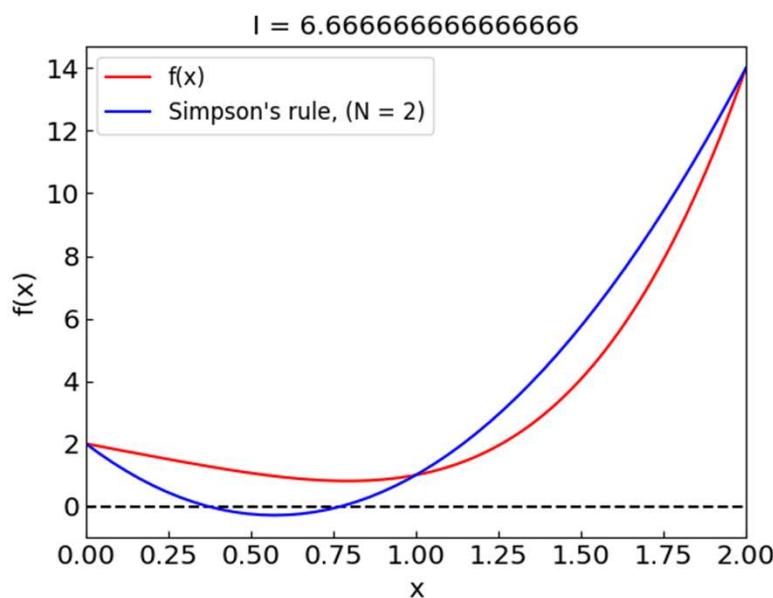
$$I - I_S = C h^4 + \mathcal{O}(h^6)$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Extended Simpson's rule

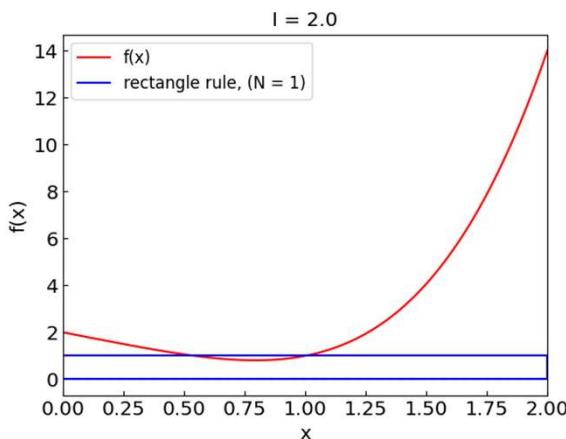
$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



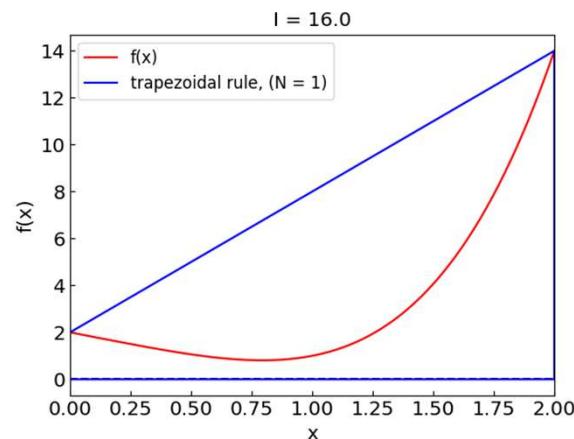
From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Comparing the methods

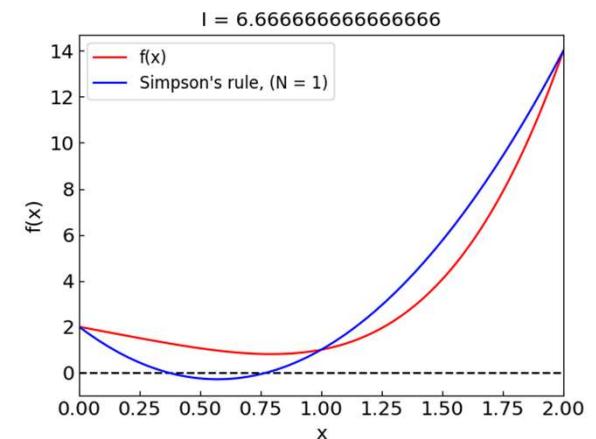
Rectangle



Trapezoid



Simpson



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Adaptive quadrature

We would like to control the error in our calculation

This can be achieved by doubling the number of subintervals
and keeping track of the error estimate

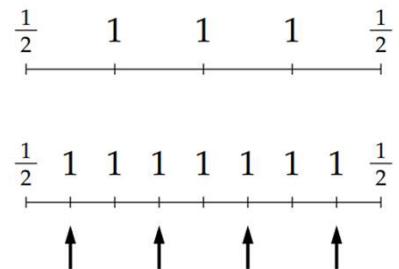
Recall that in the rectangle/trapezoidal rule the error is
proportional to h^2

$$I - I_{\text{trap}} \approx ch^2$$

At step k we have $h_k = h_{k-1}/2$ therefore $I - I_{\text{trap}}^k \approx ch_k^2$, $I - I_{\text{trap}}^{k-1} \approx 4ch_k^2$,
and the error at step k is estimated as

$$\varepsilon_k \simeq (I_{\text{trap}}^k - I_{\text{trap}}^{k-1})/3$$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>



Adaptive trapezoidal rule

```
Trapezoidal rule for numerical integration with adaptive step
if trapezoidal_rule_adaptive(f, a, b, nst = 1, tol = 1.e-8, max_iterations = 16):
    Iprev = 0.
    n = nst
    Iprev = trapezoidal_rule(f, a, b, n)
    print("Iteration: {0:5}, I = {1:20.15f}".format(1, Iprev))
    for k in range(1, max_iterations):
        n *= 2
        Inew = trapezoidal_rule(f, a, b, n)
        ek = (Inew - Iprev) / 3.
        print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(k+1, Inew, ek))
        if (abs(ek) < tol): Computing the integral of x^4 - 2x + 2 over the interval ( 0.0 , 2.0 ) using adaptive trapezoidal rule
            return Inew
        Iprev = Inew
        Iteration: 2, I = 9.00000000000000, error estimate = -2.33333333333333
        Iteration: 3, I = 7.06250000000000, error estimate = -0.64583333333333
        Iteration: 4, I = 6.55664062500000, error estimate = -0.16536458333333
        Iteration: 5, I = 6.44165039062500, error estimate = -0.04158528645833
        Iteration: 6, I = 6.41041564941406, error estimate = -0.01041158040364
        Iteration: 7, I = 6.402604103088379, error estimate = -0.002603848775228
        Iteration: 8, I = 6.400651037693024, error estimate = -0.000651021798452
        Iteration: 9, I = 6.400162760168314, error estimate = -0.000162759174903
        Iteration: 10, I = 6.400040690088645, error estimate = -0.000040690026556
        Iteration: 11, I = 6.400010172525072, error estimate = -0.000010172521191
        Iteration: 12, I = 6.400002543131352, error estimate = -0.000002543131240
        Iteration: 13, I = 6.400000635782950, error estimate = -0.000000635782801
        Iteration: 14, I = 6.400000158945742, error estimate = -0.000000158945736
        Iteration: 15, I = 6.400000039736406, error estimate = -0.000000039736446
        Iteration: 16, I = 6.40000009934106, error estimate = -0.00000009934100
Failed to achieve tolerance. Final result: 6.40000009934106
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Adaptive Simpson rule

For Simpson's rule $\varepsilon_k \simeq (I_S^k - I_S^{k-1})/15$ (understand why 15 and not 3?)

```
# Simpson's rule for numerical integration with adaptive step
def simpson_rule_adaptive(f, a, b, nst = 2, tol = 1.e-8, max_iterations = 16):
    Iprev = 0.
    n = nst
    Iprev = simpson_rule(f, a, b, n)
    print("Iteration: {0:5}, I = {1:20.15f}".format(1, Iprev))
    for k in range(1, max_iterations):
        n *= 2
        Inew = simpson_rule(f, a, b, n)
        ek = (Inew - Iprev) / 15.

        print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(k+1, Inew, ek))
        if (abs(ek) < tol):
            return Inew
        Iprev = Inew

    print("Failed to achieve")
    return Inew
```

Computing the integral of $x^4 - 2x + 2$ over the interval (0.0 , 2.0) using adaptive Simpson's rule

Iteration:	1, I =	6.66666666666666
Iteration:	2, I =	6.41666666666666, error estimate = -0.0166666666666667
Iteration:	3, I =	6.40104166666666, error estimate = -0.001041666666667
Iteration:	4, I =	6.40006510416666, error estimate = -0.000065104166667
Iteration:	5, I =	6.400004069010416, error estimate = -0.000004069010417
Iteration:	6, I =	6.400000254313150, error estimate = -0.000000254313151
Iteration:	7, I =	6.400000015894571, error estimate = -0.000000015894572
Iteration:	8, I =	6.40000000993410, error estimate = -0.00000000993411

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Adaptive quadratures: Romberg method

Recall that we obtained error estimate for trapezoidal method at step k

$$\varepsilon_k \simeq (I_{\text{trap}}^k - I_{\text{trap}}^{k-1})/3$$

On the other hand, by definition, $\varepsilon_k = I - I_{\text{trap}}^k$

Therefore, we can improve our estimate of the integral as

$$I = R_{k,1} = I_{\text{trap}}^k + \frac{I_{\text{trap}}^k - I_{\text{trap}}^{k-1}}{3} + \mathcal{O}(h^4)$$

Romberg method: continue this procedure iteratively

$$R_{k,m+1} = R_{k,m} + \frac{R_{k,m} - R_{k-1,m}}{4^m - 1} .$$

until the desired accuracy is reached

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Romberg method

```
def romberg(
    f,
    a,
    b,
    accuracy=1e-8,
    max_order=10
):
    R = np.zeros((max_order, max_order))
    h = (b - a) / 2.
    R[0, 0] = h * (f(a) + f(b)) # The initial trapezoidal rule
    for n in range(1, max_order):
        trapezoid = 0.0
        for j in range(2**(n-1)):
            trapezoid += f(a + (2*j+1)*h)
        R[n, 0] = 0.5 * R[n-1, 0] + h * trapezoid # The trapezoidal rule
        l = 1
        # The Romberg iterations
        for m in range(1, n+1):
            l *= 4
            R[n, m] = (l * R[n, m-1] - R[n-1, m-1]) / (l-1)
        print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(n, R[n, m], abs(R[n, m] - R[n-1, m-1])))
        if abs(R[n, m] - R[n-1, m-1]) < accuracy:
            return R[n, m]
    h /= 2.
print("Romberg method did not converge to required accuracy")
return R[-1, -1]
```

Computing the integral of $x^4 - 2x + 2$ over the interval (0.0 , 2.0) using Romberg method

Iteration: 1, I = 6.666666666666667, error estimate = 9.333333333333332
Iteration: 2, I = 6.400000000000000, error estimate = 0.2666666666666667
Iteration: 3, I = 6.400000000000000, error estimate = 0.000000000000000

From the course by Volodymyr Vovchenko,

<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals

- Contain integrable singularities (typically at the endpoints)

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_0^1 = 2$$

- (Semi-)infinite integration range

$$\int_0^\infty e^{-x} dx = 1$$

$$\int_{-\infty}^\infty e^{-x^2} dx = \sqrt{\pi}$$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals: Singularities at endpoints

- Even though if the singularities at integration endpoints are integrable, the trapezoidal, Simpson, etc. methods will fail because they evaluate the integrand at the endpoints

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x} \Big|_0^1 = 2$$

```
def fsing1(x):
    return 1./np.sqrt(x)

trapezoidal_rule(fsing1,0.,1.,10)
/tmp/ipykernel_31240/847063500.py:2: RuntimeWarning: divide by zero encountered in double_scalars
    return 1./np.sqrt(x)
```

- Solution:** use method that does use the endpoints (e.g. rectangle rule)

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals: Singularities at endpoints

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_0^1 = 2$$

```
def fsing1(x):
    return 1./np.sqrt(x)
```

```
| print('Using rectangle rule to evaluate \int_0^1 1/\sqrt{x} dx')
| nst = 1
| rectangle_rule_adaptive(fsing1,0.,1.,1,1.e-3,20)
```

```
Using rectangle rule to evaluate \int_0^1 1/\sqrt{x} dx
Iteration: 1, I = 1.414213562373095
Iteration: 2, I = 1.577350269189626, error estimate = 0.054378902272177
Iteration: 3, I = 1.698844079579673, error estimate = 0.040497936796682
Iteration: 4, I = 1.786461001734842, error estimate = 0.029205640718390
Iteration: 5, I = 1.848856684639738, error estimate = 0.020798560968299
Iteration: 6, I = 1.893088359706383, error estimate = 0.014743891688882
Iteration: 7, I = 1.924392755699513, error estimate = 0.010434798664376
Iteration: 8, I = 1.946535279970520, error estimate = 0.007380841423669
Iteration: 9, I = 1.962194152677056, error estimate = 0.005219624235512
Iteration: 10, I = 1.973267083679453, error estimate = 0.003690977000799
Iteration: 11, I = 1.981096937261288, error estimate = 0.002609951193945
Iteration: 12, I = 1.986633507070365, error estimate = 0.001845523269692
Iteration: 13, I = 1.990548459938304, error estimate = 0.001304984289313
Iteration: 14, I = 1.993316751362098, error estimate = 0.000922763807931
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals: (Semi-)infinite intervals

Solution: map to a finite interval [e.g. (0,1)] by a change of variables

- Semi-infinite:

$$\int_a^{\infty} f(x)dx \quad \xrightarrow{\hspace{2cm}} \quad x = a + \frac{t}{1-t} \quad \xrightarrow{\hspace{2cm}} \quad \int_a^{\infty} f(x)dx = \int_0^1 f\left(a + \frac{t}{1-t}\right) \frac{dt}{1-t^2} = \int_0^1 g(t)dt$$

- Infinite:

$$\int_{-\infty}^{\infty} f(x)dx \quad \xrightarrow{\hspace{2cm}} \quad x = \frac{t}{1-t^2} \quad \xrightarrow{\hspace{2cm}} \quad \int_{-\infty}^{\infty} f(x)dx = \int_{-1}^1 f\left(\frac{t}{1-t^2}\right) \frac{1+t^2}{(1-t^2)^2} dt = \int_{-1}^1 g(t)dt$$

Then apply a standard method (e.g. rectangle rule to avoid endpoint singularities) to $g(t)$

NB: Other options for the change of variable are possible

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals: Semi-infinite intervals

$$\int_0^{\infty} e^{-x} dx = 1$$

```
def fexp(x):
    return np.exp(-x)

def g(t, f, a = 0.):
    return f(a + t / (1. - t)) / (1. - t)**2

a = 0.
def frect(x):
    return g(x, fexp, a)

print('Using change of variable and the rectangle rule to evaluate \int_0^{\infty} \exp(-x) dx')
rectangle_rule_adaptive(frect, 0., 1., 1.e-6, 20)
```

```
Using change of variable and the rectangle rule to evaluate \int_0^{\infty} \exp(-x) dx
Iteration: 1, I = 1.471517764685769
Iteration: 2, I = 1.035213267452946, error estimate = -0.145434832410941
Iteration: 3, I = 0.984670579385046, error estimate = -0.016847562689300
Iteration: 4, I = 1.001784913275257, error estimate = 0.005704777963404
Iteration: 5, I = 1.000155714391028, error estimate = -0.000543066294743
Iteration: 6, I = 1.000040642390661, error estimate = -0.000038357333456
Iteration: 7, I = 1.000010172618432, error estimate = -0.000010156590743
Iteration: 8, I = 1.000002543136036, error estimate = -0.000002543160799
Iteration: 9, I = 1.000000635783161, error estimate = -0.000000635784292
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Improper integrals: Infinite intervals

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi} = 1.772454 \dots$$

```
def fexp2(x):
    return np.exp(-x**2)

def g2(t, f):
    return f(t / (1. - t**2)) * (1.+t**2) / (1. - t**2)**2

def frect2(x):
    return g2(x, fexp2)

print('Using change of variable and the rectangle rule to evaluate \int_{-\infty}^{\infty} \exp(-x^2) dx')
rectangle_rule_adaptive(frect2, -1., 1., 1, 1.e-6, 20)

print('Expected value: \sqrt{\pi} = ', np.sqrt(np.pi))

Using change of variable and the rectangle rule to evaluate \int_{-\infty}^{\infty} \exp(-x^2) dx
Iteration:  1, I =  2.000000000000000
Iteration:  2, I =  2.849690615244243, error estimate = 0.283230205081414
Iteration:  3, I =  1.557994553948652, error estimate = -0.430565353765197
Iteration:  4, I =  1.808005109208286, error estimate = 0.083336851753211
Iteration:  5, I =  1.770118560572371, error estimate = -0.012628849545305
Iteration:  6, I =  1.772492101507391, error estimate = 0.000791180311673
Iteration:  7, I =  1.772453880915058, error estimate = -0.000012740197444
Iteration:  8, I =  1.772453850905505, error estimate = -0.000000010003185
Expected value: \sqrt{\pi} = 1.7724538509055159
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Numerical integration so far

- Rectangle rule

$$\int_a^b f(x) dx \approx (b - a) f\left(\frac{a + b}{2}\right)$$

- Trapezoidal rule

$$\int_a^b f(x) dx \approx (b - a) \frac{f(a) + f(b)}{2}$$

- Simpson's rule

$$\int_a^b f(x) dx \approx \frac{(b - a)}{6} \left[f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right]$$

All can be written as

$$\int_a^b f(x) dx \approx \sum_k w_k f(x_k)$$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Integrating the interpolating polynomial

There is a systematic way to derive a numerical integration scheme

$$\int_a^b f(x) dx \approx \sum_k w_k f(x_k)$$

which will give an exact result when $f(x)$ is a polynomial up to a certain degree.

Recall the interpolating polynomial through $N+1$ points where $f(x)$ can be evaluated

$$f(x) \approx p_N(x) = \sum_{k=0}^N f(x_k) L_{N,k}(x)$$

$$L_{N,k}(x) = \prod_{j \neq k} \frac{x - x_j}{x_k - x_j}$$

Then, the integral reads

$$\int_a^b f(x) dx \approx \int_a^b p_N(x) dx = \sum_{k=0}^N w_k f(x_k) \quad \text{where} \quad w_k = \int_a^b L_{N,k}(x) dx$$

This expression is exact when $f(x)$ is a polynomial up to degree N

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Quadrature methods

- Newton-Cotes: equidistant spacing
- Clenshaw-Curtis: minimize Runge phenomenon
- Gauss-Legendre quadrature: use Legendre polynomials

Summary of integration

- Rectangle/trapezoidal rule
 - Good for quick calculations not requiring great accuracy
 - Does not rely on the integrand being smooth; a good choice for noisy/singular integrands, equally spaced points
- Romberg method
 - Control over error
 - Good for relatively smooth functions evaluated at equidistant nodes
- Gaussian quadrature
 - Theoretically most accurate if the function is relatively smooth
 - Good for many repeated calculations of the same type of integral
 - Requires unequally spaced nodes
 - Error can be challenging to control, especially for non-smooth functions

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Non-linear equations

Suppose we have an equation $f(x) = 0$

We can evaluate $f(x)$, but we do not know how to solve it for x

Examples:

- Roots of high-order polynomials (physics example: Lagrange L₁ point)
- Transcendental equations
 - e.g. magnetization equation

$$M = \mu \tanh \frac{JM}{k_B T}$$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Root-finding techniques

Numerical root-finding method: iterative process to determine the root(s) of non-linear equation(s) to desired accuracy

Types:

- Two-point (bracketing)
 - Bisection method
 - False position method
- Local
 - Secant method
 - Newton-Raphson method (using the derivative)
 - Relaxation method
- Multi-dimensional
 - Newton method
 - Broyden method

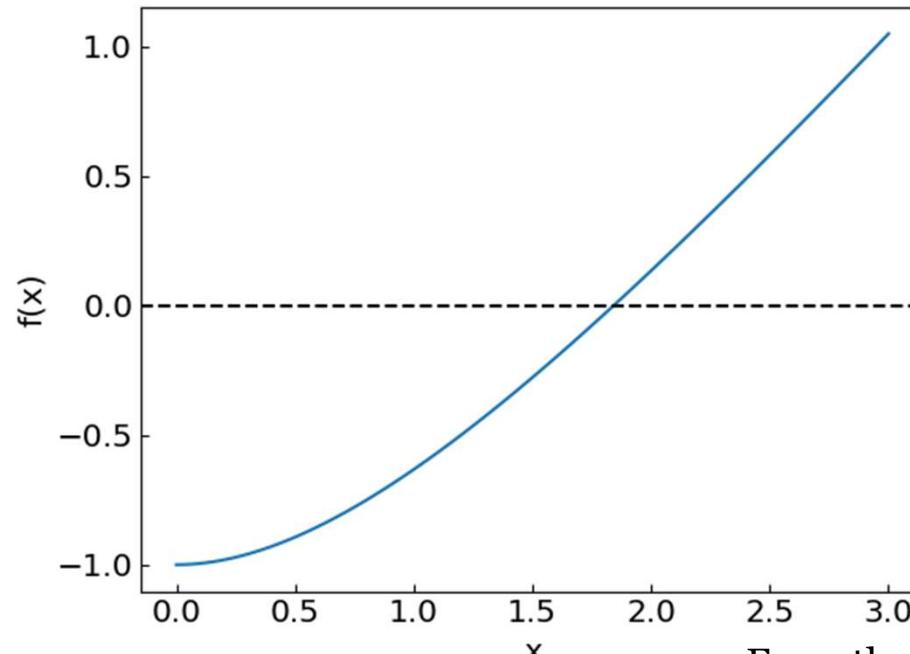
From the course by Volodymyr Vovchenko,

<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Non-linear equations

Consider an equation

$$x + e^{-x} - 2 = 0$$



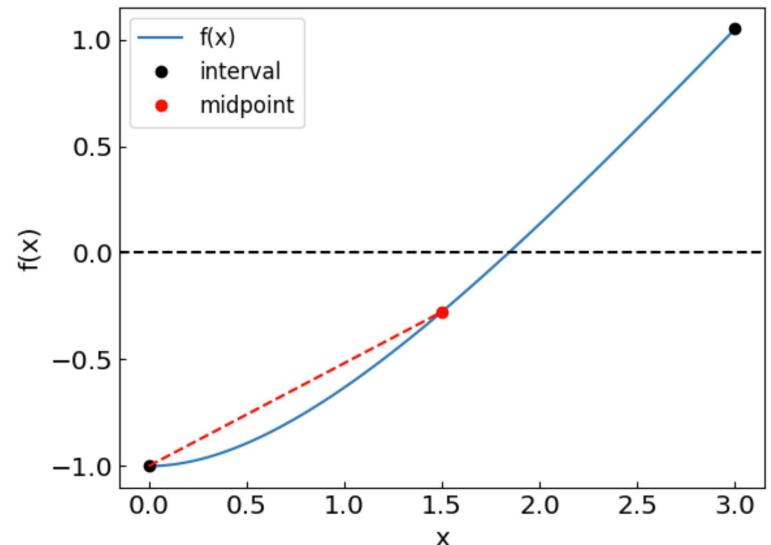
From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Bisection method

Bisection method:

1. Find an interval (a,b) which brackets the root x^*
 - $x^* \in (a,b)$
 - $f(a) & f(b)$ have opposite signs
2. Take the midpoint $c = (a+b)/2$ and halve the interval bracketing the root
3. Repeat the process until the desired precision is achieved

Method is guaranteed to converge to the root
The error is halved at each step – “linear” convergence



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Bisection method

```
def bisection_method(
    f,                      # The function whose root we are trying to find
    a,                      # The left boundary
    b,                      # The right boundary
    tolerance = 1.e-10,      # The desired accuracy of the solution
):
    fa = f(a)                # The value of the function at the left boundary
    fb = f(b)                # The value of the function at the right boundary
    if (fa * fb > 0.):
        return None            # Bisection method is not applicable

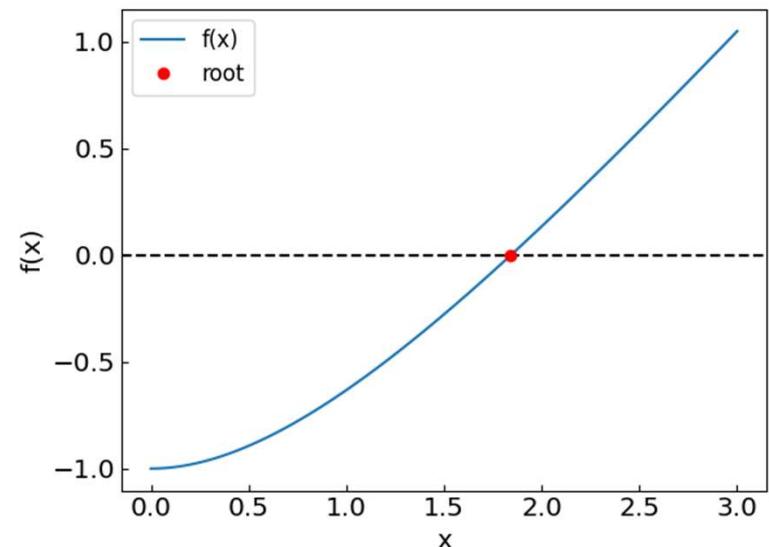
    global last_bisection_iterations
    last_bisection_iterations = 0

    while ((b-a) > tolerance):
        last_bisection_iterations += 1
        c = (a + b) / 2.        # Take the midpoint
        fc = f(c)                # Calculate the function at midpoint

        if (fc * fa < 0.):
            b = c                # The midpoint is the new right boundary
            fb = fc
        else:
            a = c                # The midpoint is the new left boundary
            fa = fc

    return (a+b) / 2.
```

$$x + e^{-x} - 2 = 0$$



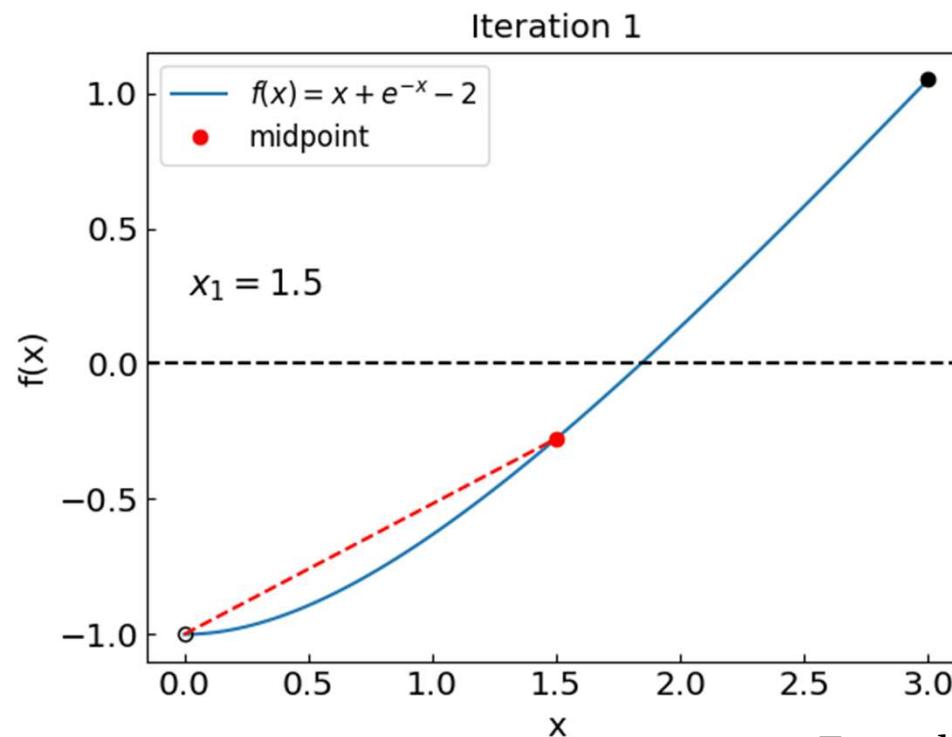
Solving the equation $x + e^{-x} - 2 = 0$ on an interval (0.0 , 3.0) using bisection method

The solution is $x = 1.8414056604233338$ obtained with 35 iterations

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Bisection method: how the iterations look like

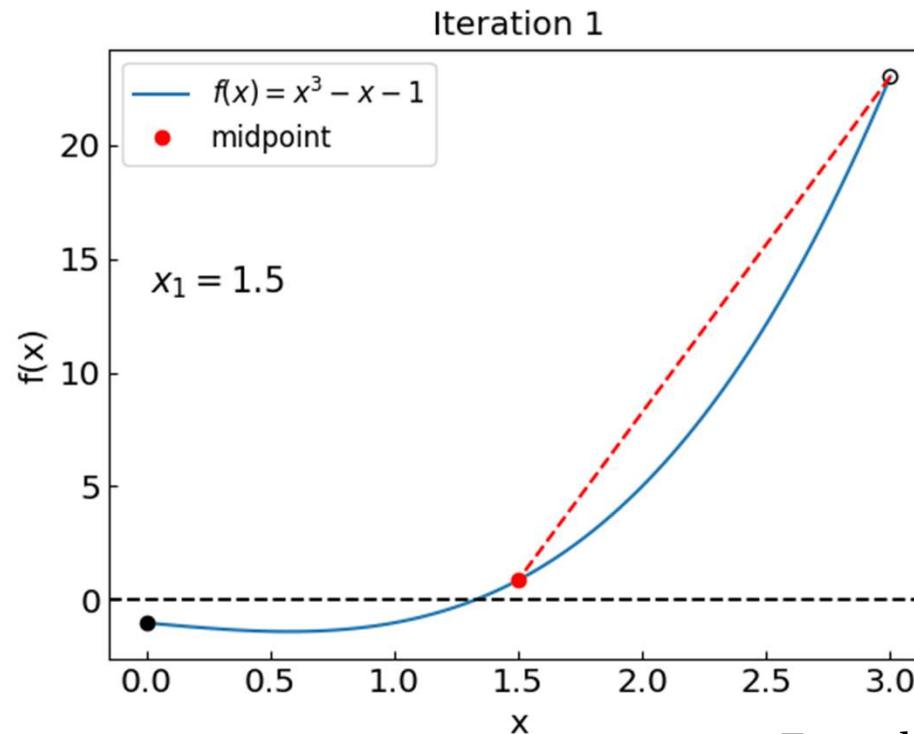
$$x + e^{-x} - 2 = 0$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Bisection method: another example

Let us consider another equation: $x^3 - x - 1 = 0$



35 iterations in both cases

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

False position method

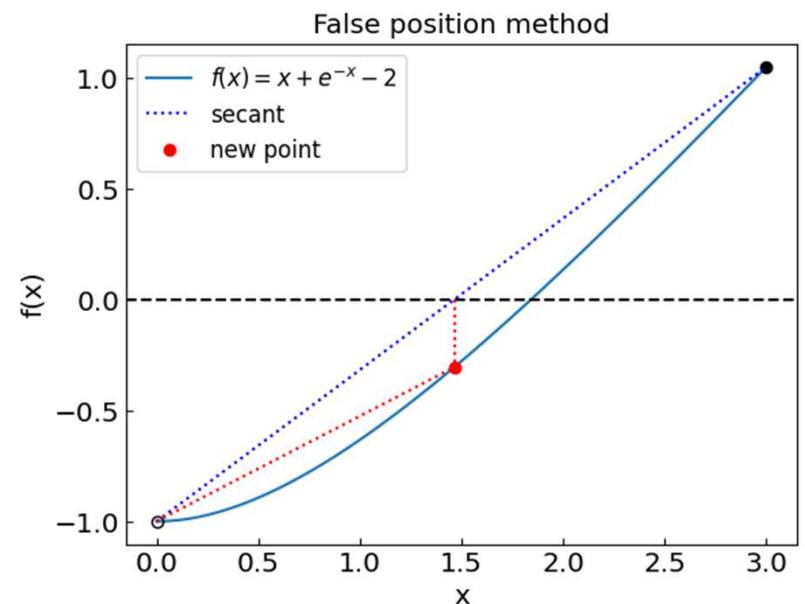
False position method:

1. Find an interval (a,b) which brackets the root x^* , same as bisection
2. Instead of midpoint take a point where the straight line between the endpoints crosses the $y = 0$ axis

$$c = a - f(a) \frac{b - a}{f(b) - f(a)}$$

3. Repeat the process until the desired precision is achieved

Method is guaranteed to converge to the root
“Linear” convergence; typically faster than bisection, but not always



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

False position method

```
def falseposition_method(
    f,                      # The function whose root we are trying to find
    a,                      # The left boundary
    b,                      # The right boundary
    tolerance = 1.e-10,     # The desired accuracy of the solution
    max_iterations = 100,   # Maximum number of iterations
):
    fa = f(a)              # The value of the function at the left boundary
    fb = f(b)              # The value of the function at the right boundary
    if (fa * fb > 0.):
        return None          # False position method is not applicable

    xprev = xnew = (a+b) / 2.           # Estimate of the solution from the previous step

    global last_falseposition_iterations
    last_falseposition_iterations = 0

    for i in range(max_iterations):
        last_falseposition_iterations += 1

        xprev = xnew
        xnew = a - fa * (b - a) / (fb - fa) # Take the point where straight line between a and b crosses y = 0
        fnew = f(xnew)                      # Calculate the function at midpoint

        if (fnew * fa < 0.):
            b = xnew                      # The intersection is the new right boundary
            fb = fnew
        else:
            a = xnew                      # The midpoint is the new left boundary
            fa = fnew

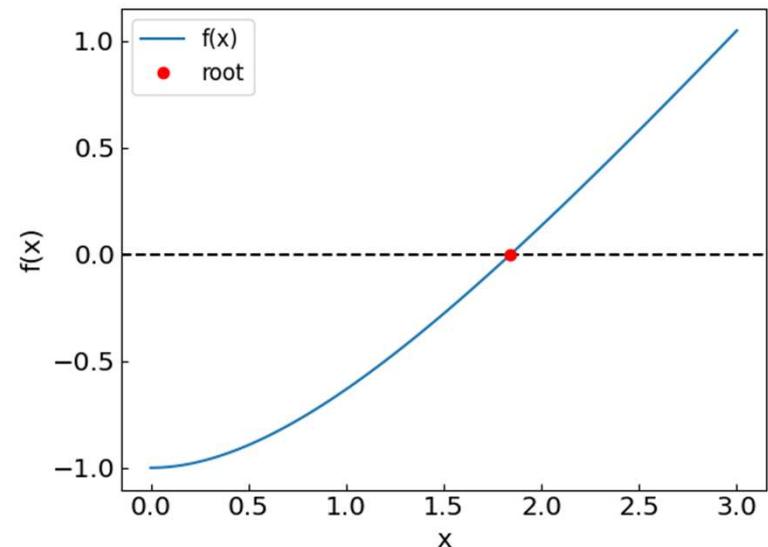
        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("False position method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

return xnew
```

Solving the equation $x + e^{-x} - 2 = 0$ on an interval (0.0 , 3.0) using the false position method
The solution is $x = 1.8414056604354012$ obtained after 11 iterations

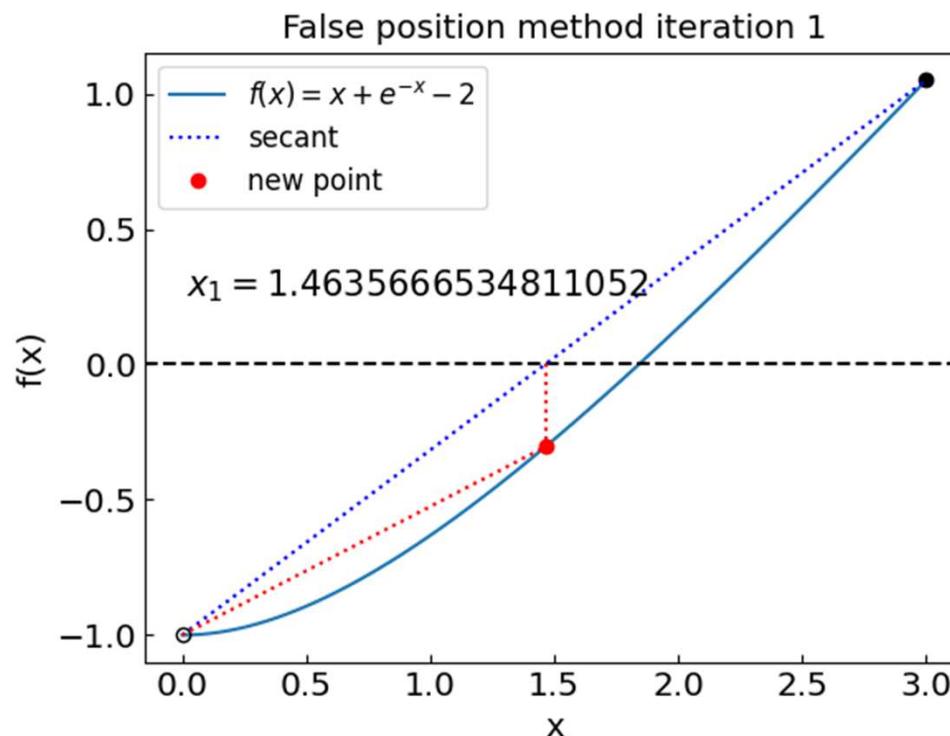
$$x + e^{-x} - 2 = 0$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

False position method

$$x + e^{-x} - 2 = 0$$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

False position vs bisection (to 10 decimal digits)

$$x + e^{-x} - 2 = 0$$

Bisection method:

```
Iteration: 1, c = 1.5000000000000000
Iteration: 2, c = 2.2500000000000000
Iteration: 3, c = 1.8750000000000000
Iteration: 4, c = 1.6875000000000000
Iteration: 5, c = 1.7812500000000000
Iteration: 6, c = 1.8281250000000000
Iteration: 7, c = 1.8515625000000000
Iteration: 8, c = 1.8398437500000000
Iteration: 9, c = 1.8457031250000000
Iteration: 10, c = 1.8427734375000000
Iteration: 11, c = 1.8413085937500000
Iteration: 12, c = 1.8420410156250000
Iteration: 13, c = 1.8416748046875000
Iteration: 14, c = 1.8414916992187500
Iteration: 15, c = 1.8414001464843750
Iteration: 16, c = 1.8414459228515625
Iteration: 17, c = 1.8414230346679690
Iteration: 18, c = 1.8414115905761720
Iteration: 19, c = 1.8414058685302730
Iteration: 20, c = 1.8414030075073240
...
Iteration: 35, c = 1.8414056604669900
```

False position method:

```
Iteration: 1, x = 1.463566653481105
Iteration: 2, x = 1.809481253839539
Iteration: 3, x = 1.839095511827520
Iteration: 4, x = 1.841240588240115
Iteration: 5, x = 1.841393875903701
Iteration: 6, x = 1.841404819191791
Iteration: 7, x = 1.841405600384506
Iteration: 8, x = 1.841405656150106
Iteration: 9, x = 1.841405660130943
Iteration: 10, x = 1.841405660415115
Iteration: 11, x = 1.841405660435401
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

False position vs bisection: not always clear who wins

$$x^3 - x - 1 = 0$$

Bisection method:

```
Iteration: 1, c = 1.500000000000000  
Iteration: 2, c = 0.750000000000000  
Iteration: 3, c = 1.125000000000000  
Iteration: 4, c = 1.312500000000000  
Iteration: 5, c = 1.406250000000000  
Iteration: 6, c = 1.359375000000000  
Iteration: 7, c = 1.335937500000000  
Iteration: 8, c = 1.324218750000000  
Iteration: 9, c = 1.330078125000000  
Iteration: 10, c = 1.327148437500000  
Iteration: 11, c = 1.325683593750000  
Iteration: 12, c = 1.324951171875000  
Iteration: 13, c = 1.324584960937500  
Iteration: 14, c = 1.324768066406250  
Iteration: 15, c = 1.324676513671875  
Iteration: 16, c = 1.324722290039062  
Iteration: 17, c = 1.324699401855469  
Iteration: 18, c = 1.324710845947266  
Iteration: 19, c = 1.324716567993164  
Iteration: 20, c = 1.324719429016113  
...  
Iteration: 35, c = 1.324717957206303
```

False position method:

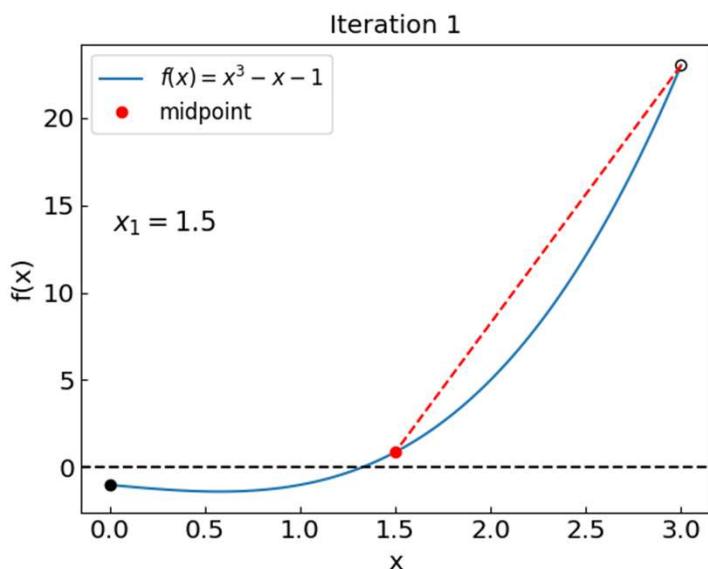
```
Iteration: 1, x = 0.125000000000000  
Iteration: 2, x = 0.258845437616387  
Iteration: 3, x = 0.399230727605107  
Iteration: 4, x = 0.541967526475374  
Iteration: 5, x = 0.681365453934702  
Iteration: 6, x = 0.811265467641601  
Iteration: 7, x = 0.926423756077868  
Iteration: 8, x = 1.023635980751716  
Iteration: 9, x = 1.102112700940041  
Iteration: 10, x = 1.163084623011103  
Iteration: 11, x = 1.209004461867383  
Iteration: 12, x = 1.242759715838447  
Iteration: 13, x = 1.267123755869329  
Iteration: 14, x = 1.284474915416815  
Iteration: 15, x = 1.296712725379603  
Iteration: 16, x = 1.305284823099690  
Iteration: 17, x = 1.311260149895704  
Iteration: 18, x = 1.315411216706803  
Iteration: 19, x = 1.318288144277179  
Iteration: 20, x = 1.320278742279728  
...  
Iteration: 66, x = 1.324717957079699
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlovch/PHYS6350-ComputationalPhysics>

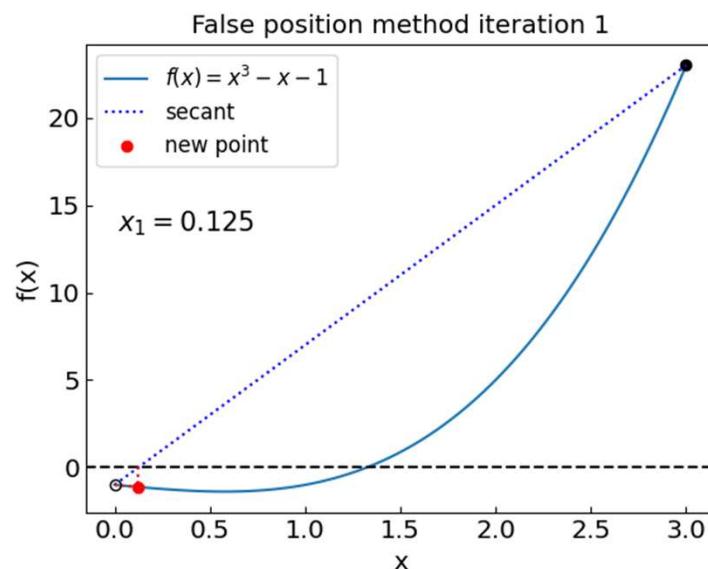
False position vs bisection: not always clear who wins

$$x^3 - x - 1 = 0$$

Bisection method:



False position method:



More advanced methods combine the two and add other refinements*

- Ridders' method
- Brent method

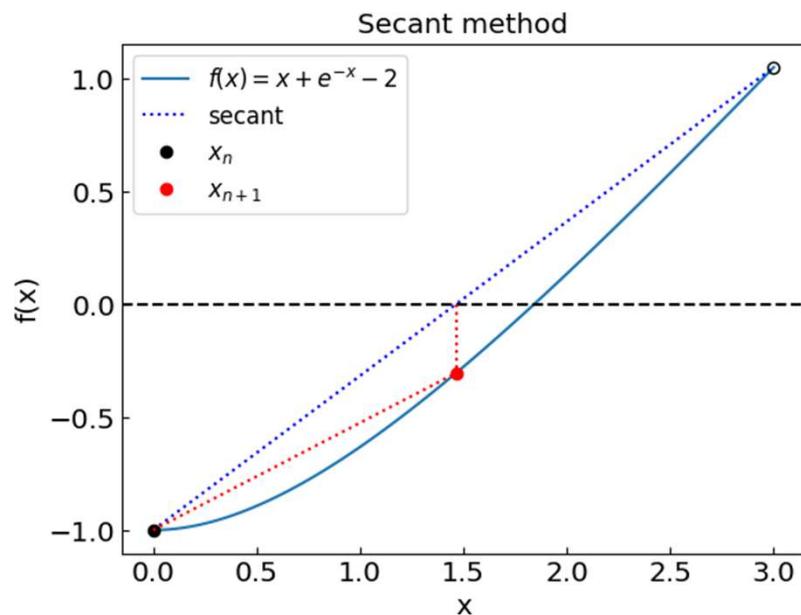
see chapters 9.2, 9.3 of *Numerical Recipes Third Edition* by W.H. Press et al.

From the course by Volodymyr Vovchenko,

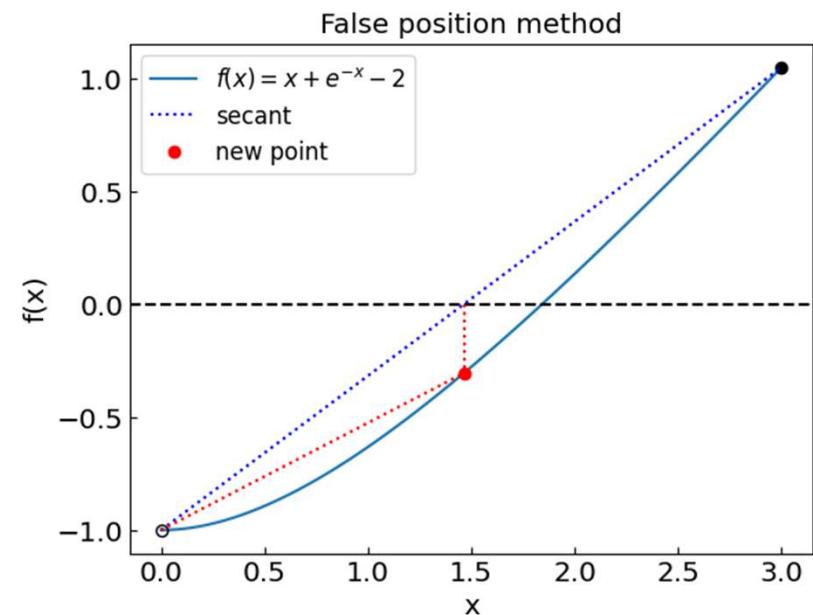
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Secant method

Secant method: similar to false position, but the interval *need not bracket the root*
Always uses the last two points



VS



Typically “superlinear” convergence when works

Can still be slower than bisection or not converge at all (e.g. secant is parallel to $y = 0$ axis)

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Secant method

```
def secant_method(
    f,                      # The function whose root we are trying to find
    a,                      # The left boundary
    b,                      # The right boundary
    tolerance = 1.e-10,      # The desired accuracy of the solution
    max_iterations = 100,    # Maximum number of iterations
):
    fa = f(a)                # The value of the function at the left boundary
    fb = f(b)                # The value of the function at the right boundary

    xprev = xnew = a          # Estimate of the solution from the previous step

    global last_secant_iterations
    last_secant_iterations = 0

    for i in range(max_iterations):
        last_secant_iterations += 1

        xprev = xnew
        xnew = a - fa * (b - a) / (fb - fa) # Take the point where straight line between a and b crosses y = 0
        fnew = f(xnew)                      # Calculate the function at midpoint

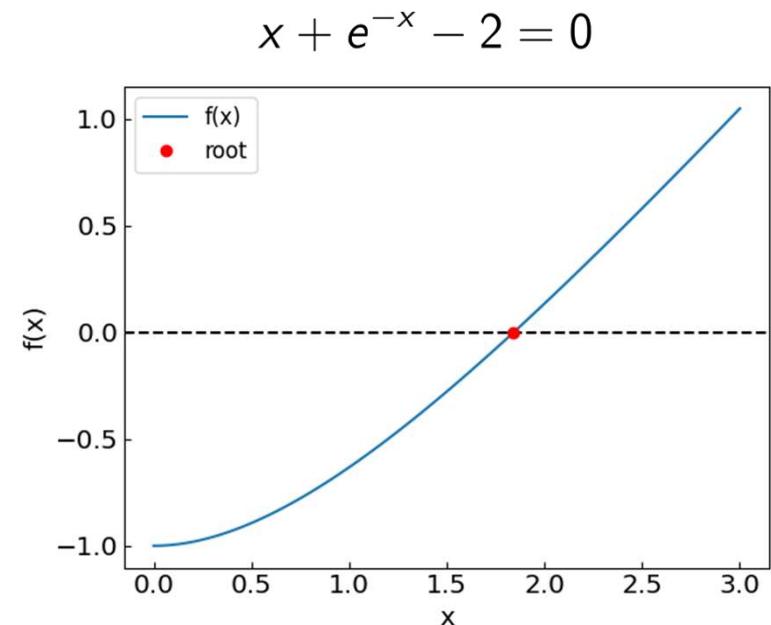
        b = a
        fb = fa
        a = xnew
        fa = fnew

        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("Secant method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

return xnew
```

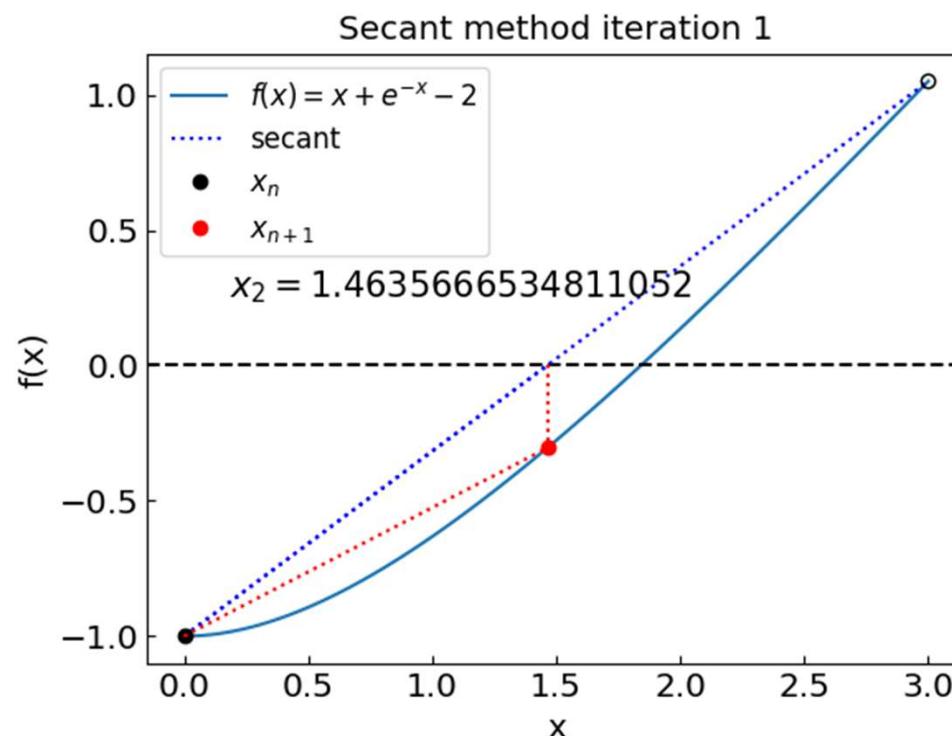
Solving the equation $x + e^{-x} - 2 = 0$ on an interval (0.0 , 3.0) using the secant method
The solution is $x = 1.8414056604369606$ obtained after 7 iterations



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Secant method

$$x + e^{-x} - 2 = 0$$

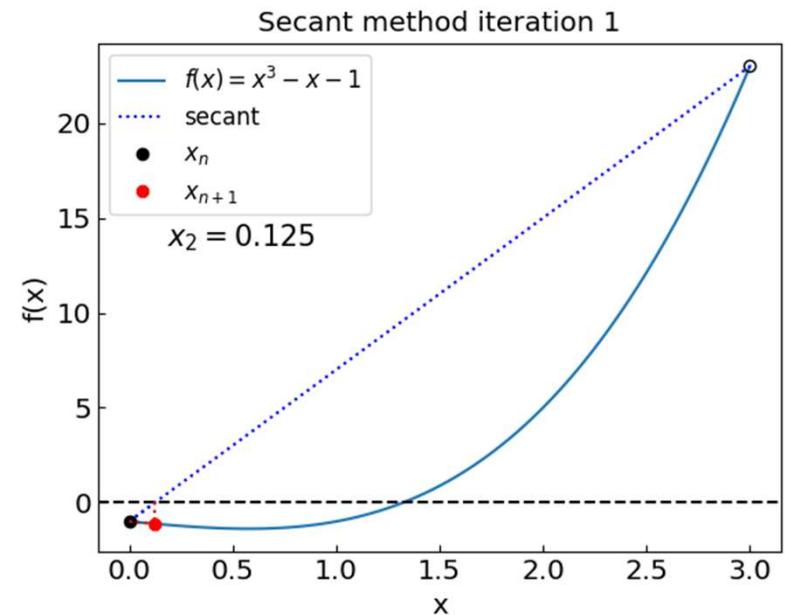


From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Secant method

$$x^3 - x - 1 = 0$$

```
Iteration: 1, x = 0.1250000000000000 Iteration: 17, x = -1.058303471905222  
Iteration: 2, x = -1.015873015873016 Iteration: 18, x = -0.643978481189561  
Iteration: 3, x = -14.026092564115256 Iteration: 19, x = -0.131674045244213  
Iteration: 4, x = -1.010979901305751 Iteration: 20, x = -1.933586024088406  
Iteration: 5, x = -1.006133240911884 Iteration: 21, x = 0.157497929951306  
Iteration: 6, x = -0.512666258317272 Iteration: 22, x = 0.626623389695762  
Iteration: 7, x = 0.273834681149844 Iteration: 23, x = -2.226715128003442  
Iteration: 8, x = -1.287767830907429 Iteration: 24, x = 1.093727500240917  
Iteration: 9, x = 3.565966235528240 Iteration: 25, x = 1.382563036703896  
Iteration: 10, x = -1.077368321415013 Iteration: 26, x = 1.310687668369503  
Iteration: 11, x = -0.947522156044583 Iteration: 27, x = 1.323983763313963  
Iteration: 12, x = -0.513174359589628 Iteration: 28, x = 1.324727653842468  
Iteration: 13, x = 0.447558454314033 Iteration: 29, x = 1.324717950607204  
Iteration: 14, x = -1.325124217388110 Iteration: 30, x = 1.324717957244686  
Iteration: 15, x = 4.186373891812861 Iteration: 31, x = 1.324717957244746  
Iteration: 16, x = -1.167930924631363
```



Because the method does not bracket the root, it is not guaranteed to converge

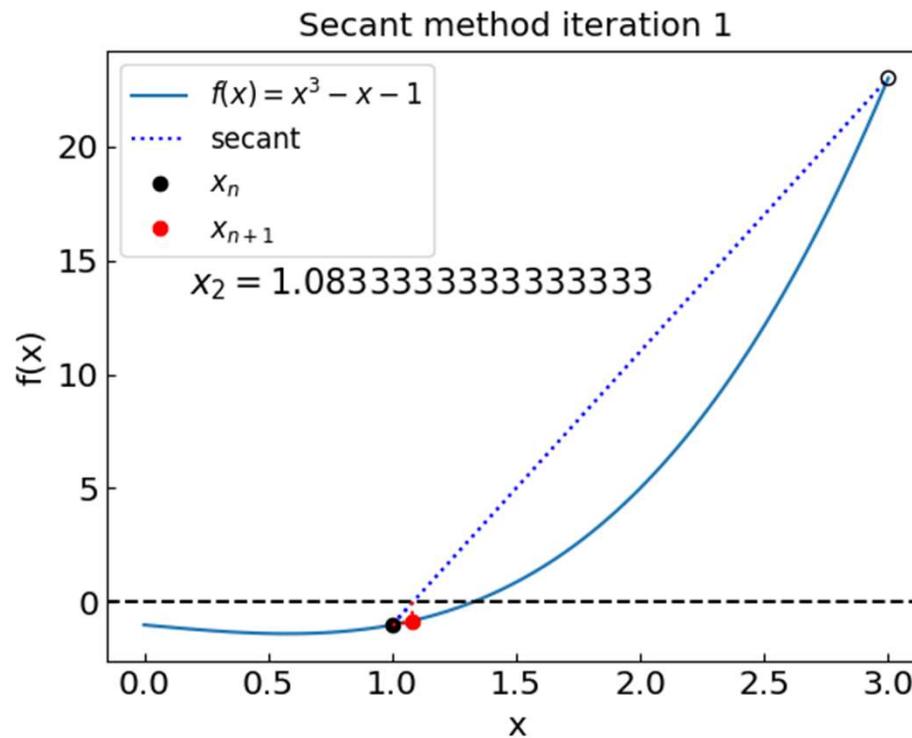
In this case, it managed to recover

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Secant method

$$x^3 - x - 1 = 0$$

Choose the initial interval as (1,3) instead of (0,3)



From the course by
Volodymyr Vovchenko,
<https://github.com/vlvo/vch/PHYS6350-ComputationalPhysics>

Newton-Raphson method

Newton-Raphson method:

- Local method (uses only the current estimate to get the next one)
- Requires the evaluation of derivative

Idea: Assume that a given point x is close to the root x^* [$f(x^*)=0$]

Then

$$f(x^*) \approx f(x) + f'(x)(x^* - x)$$

and since $f(x^*) = 0$ we have

$$x^* \approx x - \frac{f(x)}{f'(x)}$$

Iterative procedure:

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

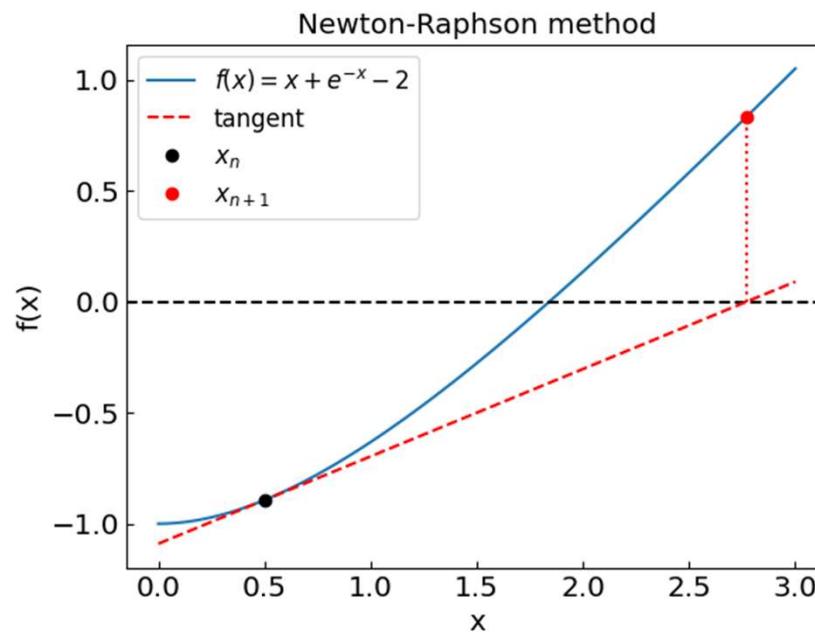
starting from an initial guess x_0

From the course by Volodymyr Vovchenko,

<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton-Raphson method

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



“Quadratic” convergence when works

However, when we are close to $f' = 0$, we have a problem

From the course by
Volodymyr Vovchenko,
[https://github.com/vch/PHYS6350-
ComputationalPhysics](https://github.com/vch/PHYS6350-ComputationalPhysics)

Newton-Raphson method

```
def newton_method(
    f,                      # The function whose root we are trying to find
    df,                     # The derivative of the function
    x0,                     # The initial guess
    tolerance = 1.e-10,      # The desired accuracy of the solution
    max_iterations = 100,    # Maximum number of iterations
):
    xprev = xnew = x0

    global last_newton_iterations
    last_newton_iterations = 0
    diff = 0.

    for i in range(max_iterations):
        last_newton_iterations += 1

        xprev = xnew
        fval = f(xprev)           # The current function value
        dfval = df(xprev)         # The current function derivative value

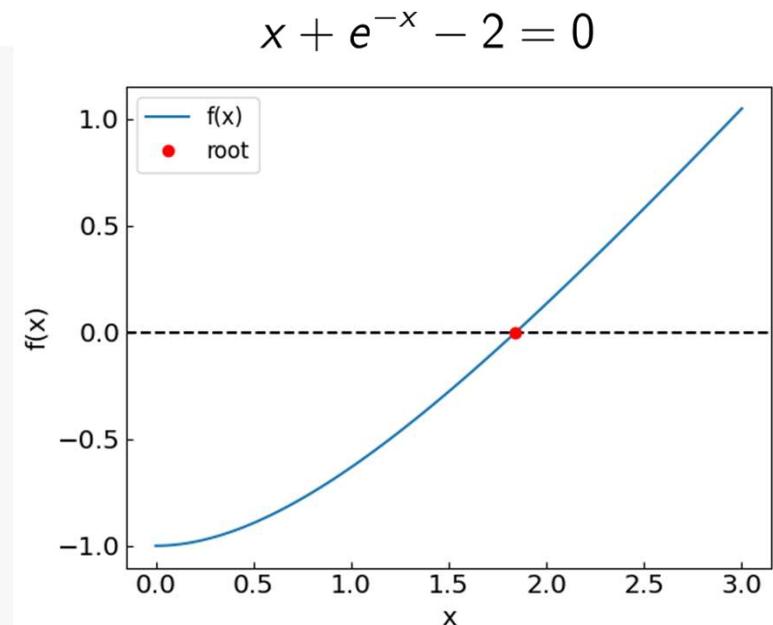
        xnew = xprev - fval / dfval # The next iteration

        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("Newton-Raphson method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew-xprev))

return xnew
```

Solving the equation $x + e^{-x} - 2 = 0$ with an initial guess of $x_0 = 0.5$
The solution is $x = 1.8414056604369606$ obtained after 6 iterations

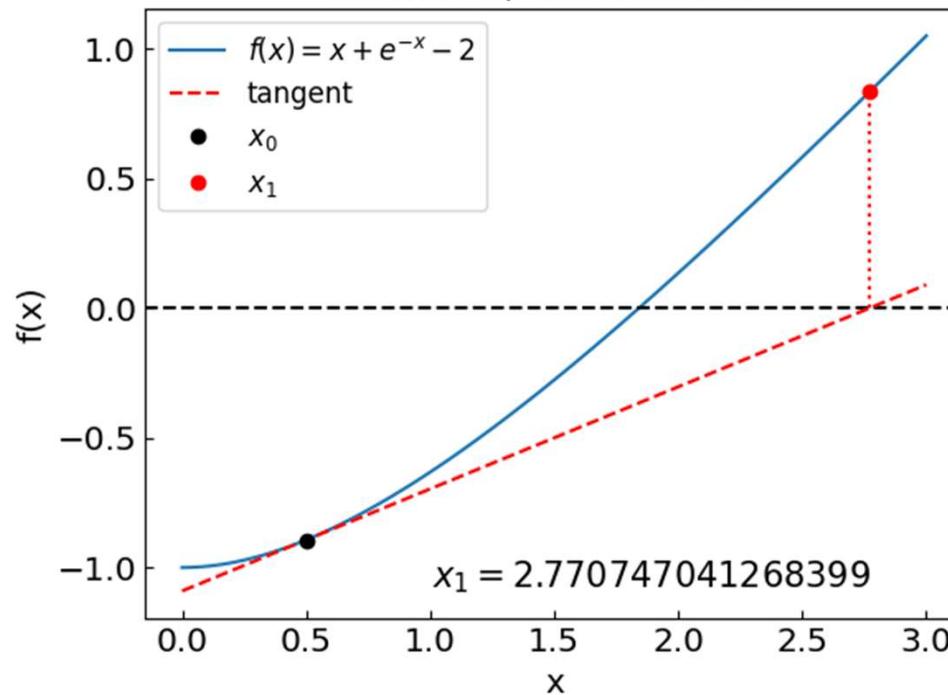


From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton-Raphson method

$$x + e^{-x} - 2 = 0$$

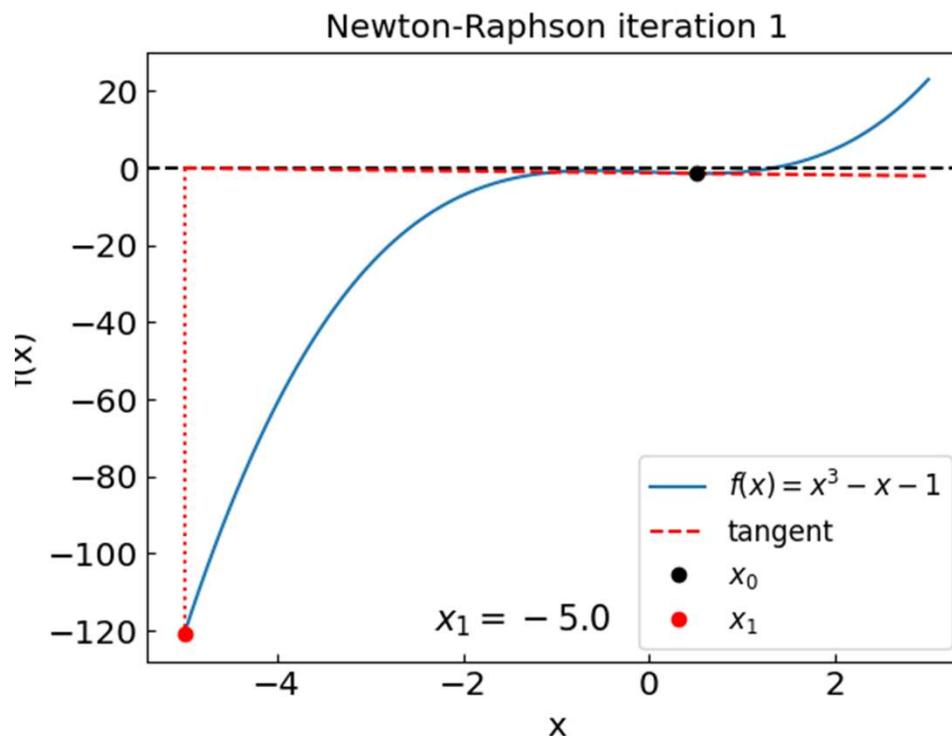
Newton-Raphson iteration 1



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton-Raphson method: issues

$$x^3 - x - 1 = 0$$



From the course by
Volodymyr
Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Similar issue as with the secant method; the reason: $f' = 0$ at $x = 0.577\dots$

Newton-Raphson method: issues

Try finding the root of $f(x) = x^3 - 2x + 2$ with an initial guess of $x_0 = 0$

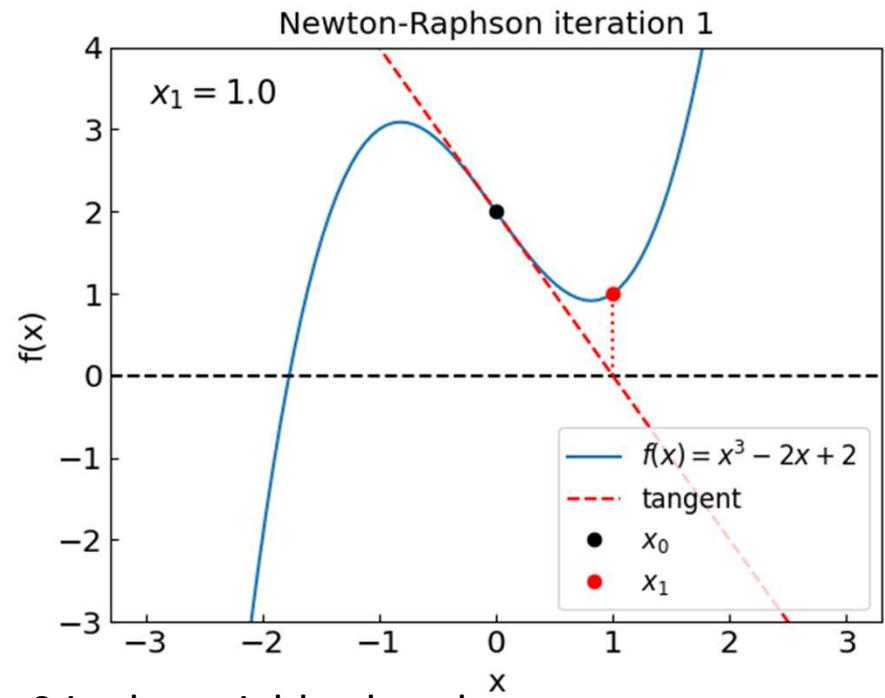
Iteration 1: $f(x_0) = 2$, $f'(x_0) = -2$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 1$$

Iteration 2: $f(x_1) = 1$ $f'(x_1) = 1$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 0$$

We are back to x_0 !



The main issue is, again, we have points with $f' = 0$ in the neighborhood

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Relaxation method

Relaxation method:

- Cast the equation $f(x) = 0$ in a form

$$x = \varphi(x)$$

- For example $\varphi(x) = f(x) + x$ but this choice is not unique
- The root is approximated by an iterative procedure

$$x_{n+1} = \varphi(x_n)$$

Convergence criterion:

$$|\varphi'(x_n)| < 1, \quad \text{for all } x_n$$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Relaxation method

```
def relaxation_method(
    phi,                      # The function from the equation x = phi(x)
    x0,                       # The initial guess
    tolerance = 1.e-10,        # The desired accuracy of the solution
    max_iterations = 100       # Maximum number of iterations
):
    xprev = xnew = x0

    global last_relaxation_iterations
    last_relaxation_iterations = 0

    for i in range(max_iterations):
        last_relaxation_iterations += 1

        xprev = xnew
        xnew = phi(xprev) # The next iteration

        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("The relaxation method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

return xnew
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Relaxation method

$$x + e^{-x} - 2 = 0 \quad \text{as} \quad x = 2 - e^{-x} \quad \text{i.e.} \quad \phi(x) = 2 - e^{-x}$$

Starting with $x_0=0.5$ we have

```
Solving the equation x = 2 - e^-x with relaxation method an initial guess of x0 = 0.5
Iteration: 0, x = 0.500000000000000, phi(x) = 1.393469340287367
Iteration: 1, x = 1.393469340287367, phi(x) = 1.751787325113973
Iteration: 2, x = 1.751787325113973, phi(x) = 1.826536369684999
Iteration: 3, x = 1.826536369684999, phi(x) = 1.839029855597129
Iteration: 4, x = 1.839029855597129, phi(x) = 1.841028423293983
Iteration: 5, x = 1.841028423293983, phi(x) = 1.841345821475382
Iteration: 6, x = 1.841345821475382, phi(x) = 1.841396170032424
Iteration: 7, x = 1.841396170032424, phi(x) = 1.841404155305379
Iteration: 8, x = 1.841404155305379, phi(x) = 1.841405421731432
Iteration: 9, x = 1.841405421731432, phi(x) = 1.841405622579610
Iteration: 10, x = 1.841405622579610, phi(x) = 1.841405654432999
Iteration: 11, x = 1.841405654432999, phi(x) = 1.841405659484766
Iteration: 12, x = 1.841405659484766, phi(x) = 1.841405660285948
Iteration: 13, x = 1.841405660285948, phi(x) = 1.841405660413011
Iteration: 14, x = 1.841405660413011, phi(x) = 1.841405660433162
Iteration: 15, x = 1.841405660433162, phi(x) = 1.841405660436358
The solution is x = 1.8414056604331623 obtained after 15 iterations
```

Not as fast as Newton-Raphson but does not require evaluation of the derivative

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Relaxation method

$$x^3 - x - 1 = 0 \quad \text{as} \quad x = x^3 - 1 \quad \text{i.e.} \quad \varphi(x) = x^3 - 1$$

Starting with $x_0=0.5$ we have

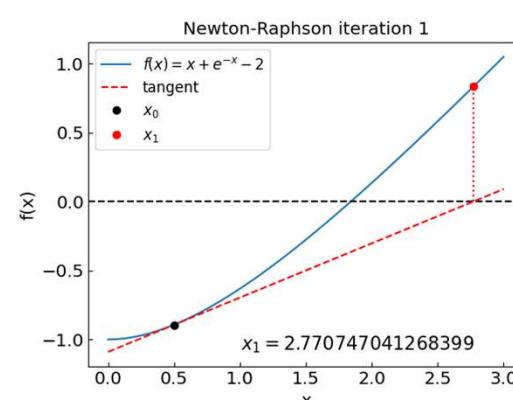
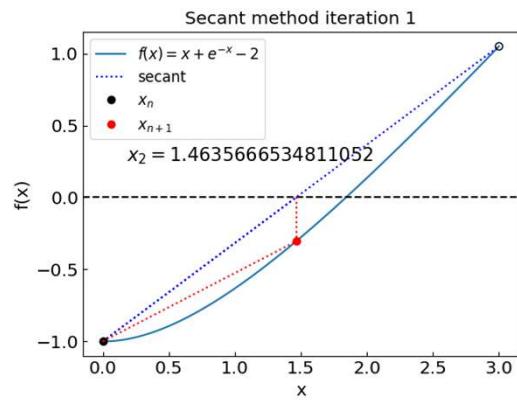
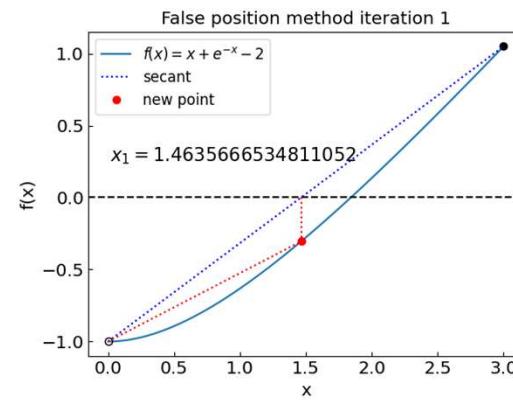
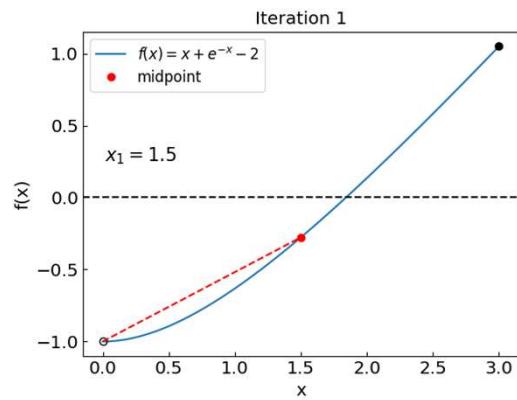
```
Solving the equation x = x^3 - 1 with relaxation method an initial guess of x0 = 0.0
Iteration: 0, x = 0.000000000000000, phi(x) = -1.000000000000000
Iteration: 1, x = -1.000000000000000, phi(x) = -2.000000000000000
Iteration: 2, x = -2.000000000000000, phi(x) = -9.000000000000000
Iteration: 3, x = -9.000000000000000, phi(x) = -730.0000000000000
Iteration: 4, x = -730.0000000000000, phi(x) = -389017001.000000000000000
Iteration: 5, x = -389017001.000000000000000, phi(x) = -58871587162270591457689600.000000000000000
Iteration: 6, x = -58871587162270591457689600.000000000000000, phi(x) = -20404090132275264698947825968051310952675782605
6202557355691431285390611316736.000000000000000
Iteration: 7, x = -204040901322752646989478259680513109526757826056202557355691431285390611316736.000000000000000, phi
(x) = -84947714722373876912426115385994721993304503407088864329587058315002861225858314510130211954336728493261609772281413
1127104275290993706669943943557518825041720139256751756296514363510463501782805696167407096791414943273033163341824.0000000
0000000
```

Divergent!

Reason: $|\varphi'(x_n)| < 1$ violated [try to come up with a better form of $\varphi(x)$?]

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Summary



Summary

Bisection method:

- Guaranteed to converge with a fixed rate
- Need to bracket the root

False position method:

- Guaranteed to converge
- Can be faster than bisection but not always
- Need to bracket the root

Secant method:

- Typically faster than bisection and false position
- May not always converge

Newton-Raphson method:

- Very fast when converges
- Can be sensitive to initial guess
- May not converge if $f'(x)=0$
- Requires evaluation of the derivative at each step

Relaxation method:

- Simple to implement
- Does not require derivative
- Often does not converge

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Systems of non-linear equations

$$\begin{aligned}f_1(x_1, \dots, x_N) &= 0, \\f_2(x_1, \dots, x_N) &= 0, \\&\dots \\f_N(x_1, \dots, x_N) &= 0\end{aligned}$$

References: Chapter 9.6 of *Numerical Recipes Third Edition* by W.H. Press et al.

Systems of non-linear equations

Sometimes we need to solve a system of non-linear equations, e.g.

$$\begin{aligned}f_1(x_1, \dots, x_N) &= 0, \\f_2(x_1, \dots, x_N) &= 0, \\&\dots \\f_N(x_1, \dots, x_N) &= 0\end{aligned}$$

Denoting $\mathbf{f} = (f_1, \dots, f_N)$ and $\mathbf{x} = (x_1, \dots, x_N)$ this can be written in compact form as

$$\mathbf{f}(\mathbf{x}) = 0.$$

For example:

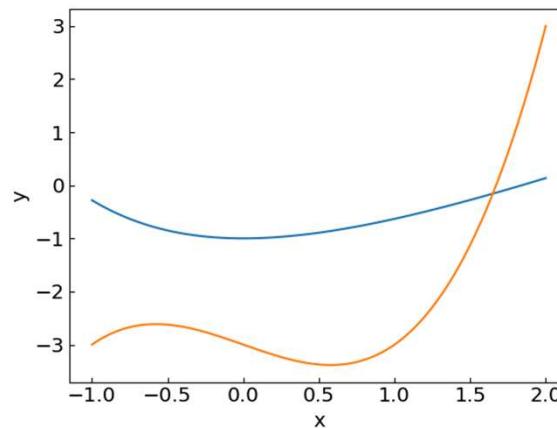
$$x + \exp(-x) - 2 - y = 0,$$

$$x^3 - x - 3 - y = 0.$$

i.e.

$$f_1(x_1, x_2) = x_1 + \exp(-x_1) - 2 - x_2$$

$$f_2(x_1, x_2) = x_1^3 - x_1 - 3 - x_2$$



From the course by
Volodymyr Vovchenko,
<https://github.com/vlvoch/PHYS6350-ComputationalPhysics>

Newton-Raphson method in multiple dimensions

Recall the Taylor expansion of function f around the root x^* in one-dimensional case:

$$f(x^*) \approx f(x) + f'(x)(x^* - x)$$

The multi-dimensional version of this expansion reads

$$\mathbf{f}(\mathbf{x}^*) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})(\mathbf{x}^* - \mathbf{x})$$

Here $\mathbf{J}(\mathbf{x})$ is the Jacobian, i.e. a $N \times N$ matrix of derivatives evaluated at \mathbf{x}

$$J_{ij}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j} .$$

Given that $\mathbf{f}(\mathbf{x}^*) = 0$, we have

$$\mathbf{J}(\mathbf{x})(\mathbf{x}^* - \mathbf{x}) \approx -\mathbf{f}(\mathbf{x}),$$

which is a system of linear equations for $\mathbf{x}^* - \mathbf{x}$. Solving this system yields

$$\mathbf{x}^* \approx \mathbf{x} - \mathbf{J}^{-1}(\mathbf{x}) \mathbf{f}(\mathbf{x}) .$$

Here $\mathbf{J}^{-1}(\mathbf{x})$ is the inverse Jacobian matrix.

The multi-dimensional Newton's method is an iterative procedure

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n) \mathbf{f}(\mathbf{x}_n) .$$

Reduces to Newton's method in 1D

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

From the course by
Volodymyr Vovchenko,
<https://github.com/vlvovch>
[/PHYS6350-](#)
[ComputationalPhysics](#)

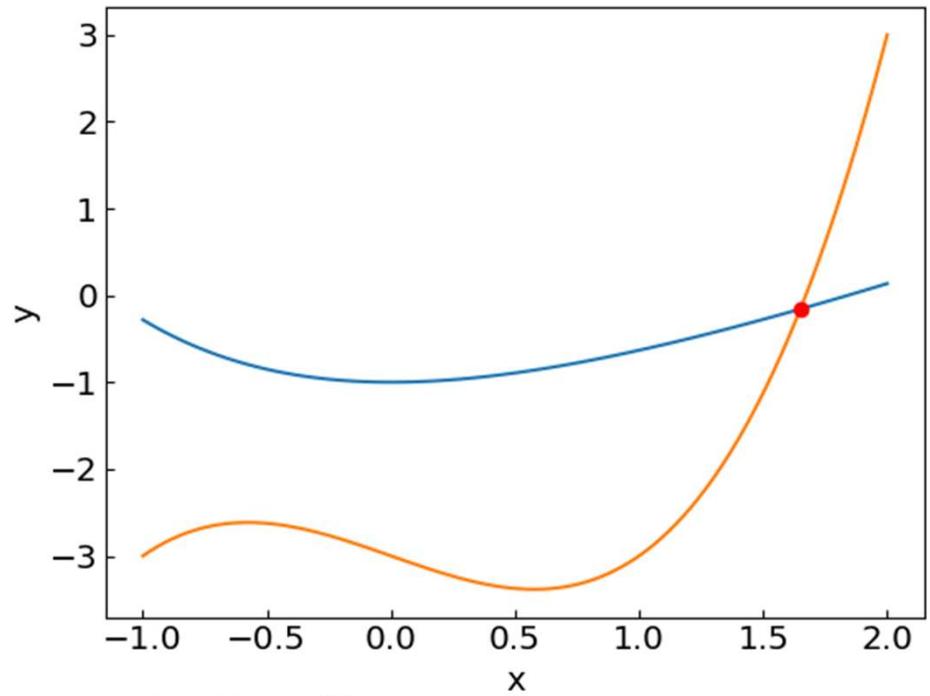
Newton-Raphson method in multiple dimensions

```
def newton_method_multi(
    f,
    jacobian,
    x0,
    accuracy=1e-8,
    max_iterations=100):
    x = x0
    global last_newton_iterations
    last_newton_iterations = 0

    if newton_verbose:
        print("Iteration: ", last_newton_iterations)
        print("x = ", x0)
        print("f = ", f(x0))
        print("|f| = ", ftil(f(x0)))

    for i in range(max_iterations):
        last_newton_iterations += 1
        f_val = f(x)
        jac = jacobian(x)
        jinv = np.linalg.inv(jac)
        delta = np.dot(jinv, -f_val)
        x = x + delta

        if np.linalg.norm(delta, ord=2) < accuracy:
            return x
    return x
```



```
Iteration: 12
x = [ 1.64998819 -0.15795963]
f = [ 0.0000000e+00 -6.66133815e-16]
|f| = 2.2186712959340957e-31
```

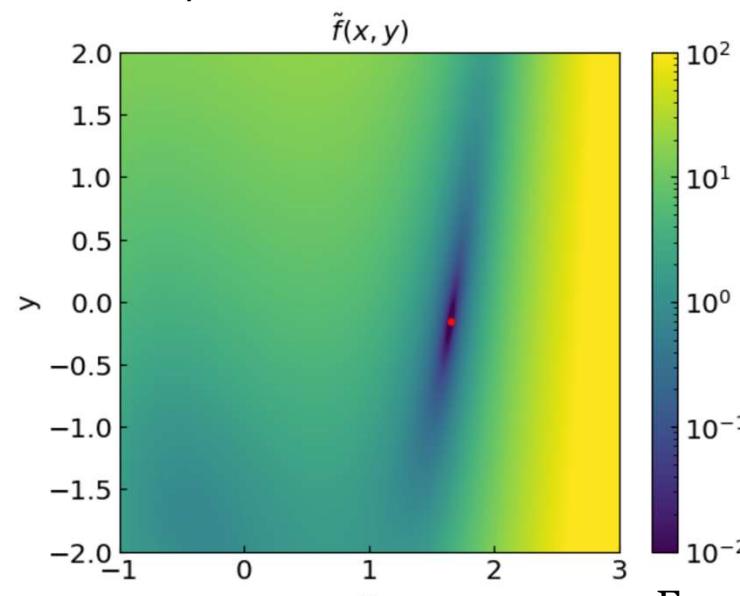
From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton-Raphson method in multiple dimensions

Introduce an objective function

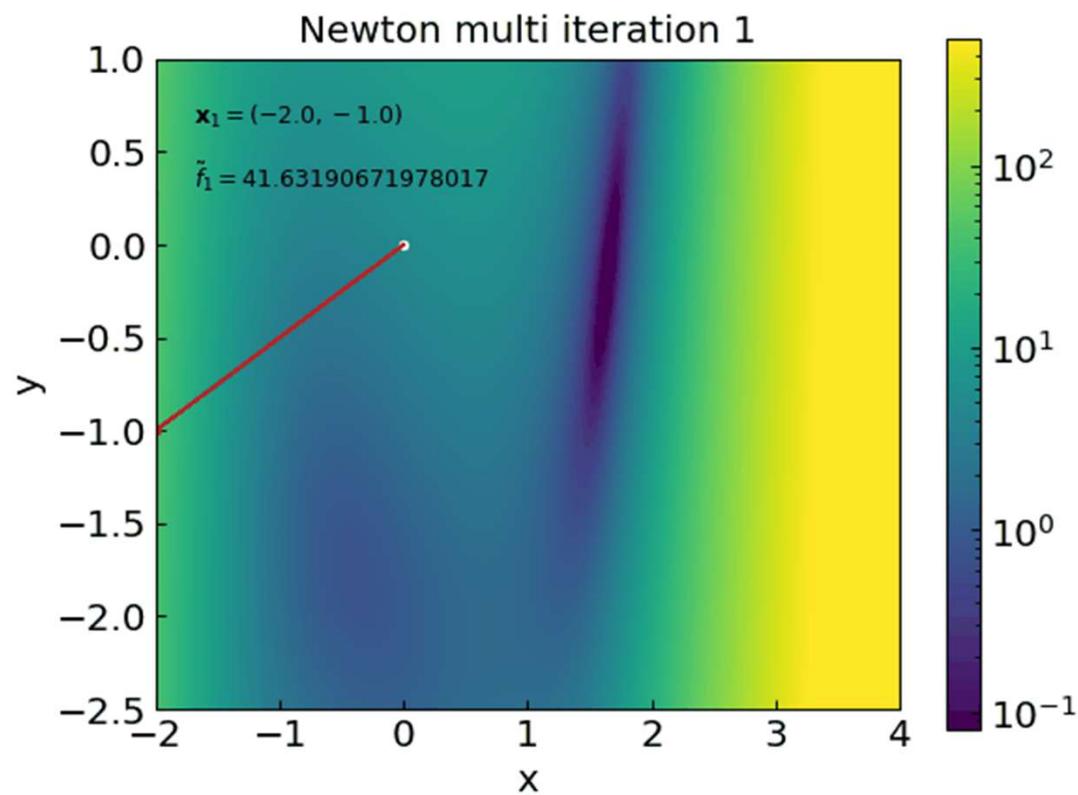
$$\tilde{f}(\mathbf{x}) = \frac{\mathbf{f}(\mathbf{x}) \cdot \mathbf{f}(\mathbf{x})}{2}$$

Its value is equal to zero (minimized) at the root



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton-Raphson method in multiple dimensions



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method

Broyden method is a multi-dimensional generalization of the secant method

Secant method: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ with $f'(x_n) \simeq \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$

Broyden method: $\mathbf{x}_{n+1} = \mathbf{x}_n - J^{-1}(\mathbf{x}_n) \mathbf{f}(\mathbf{x}_n)$ with

The solution to $J(\mathbf{x}_n)(\mathbf{x}_n - \mathbf{x}_{n-1}) \simeq \mathbf{f}(\mathbf{x}_n) - \mathbf{f}(\mathbf{x}_{n-1})$ is not unique

Broyden: $\mathbf{J}_n = \mathbf{J}_{n-1} + \frac{\Delta \mathbf{f}_n - \mathbf{J}_{n-1} \Delta \mathbf{x}_n}{\|\Delta \mathbf{x}_n\|^2} \Delta \mathbf{x}_n^T$ with $\mathbf{f}_n = \mathbf{f}(\mathbf{x}_n),$
 $\Delta \mathbf{x}_n = \mathbf{x}_n - \mathbf{x}_{n-1},$
 $\Delta \mathbf{f}_n = \mathbf{f}_n - \mathbf{f}_{n-1},$

For J_0 one can take the identity matrix or full Jacobian calculated once

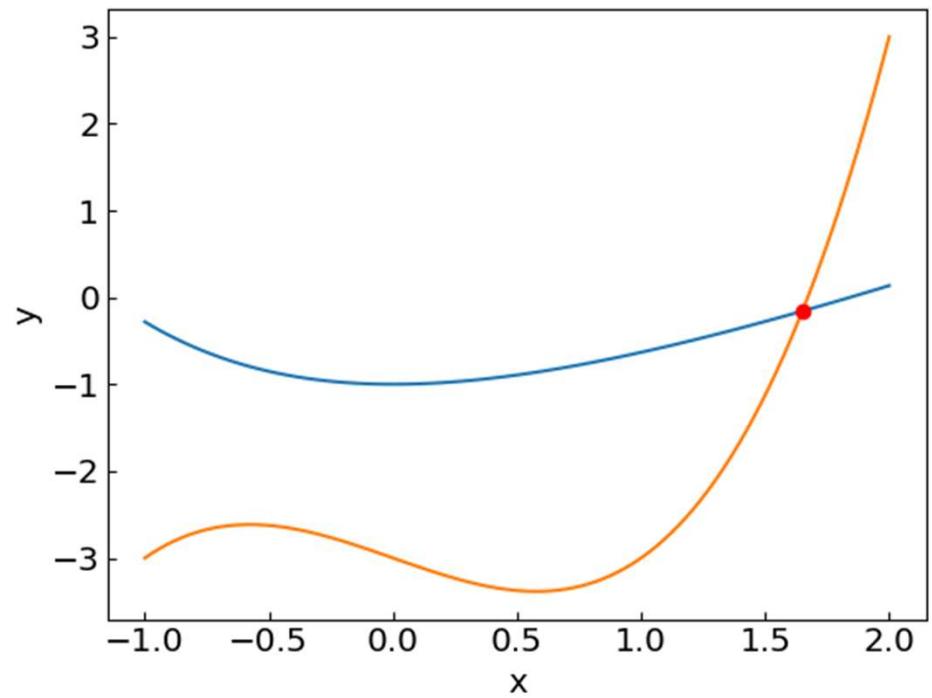
From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method (direct)

```
# Direct implementation of Broyden's method
# (using matrix inversion at each step)
def broyden_method_direct(
    f,
    x0,
    accuracy=1e-8,
    max_iterations=100):
    global last_broyden_iterations
    last_broyden_iterations = 0
    x = x0
    n = x0.shape[0]
    J = np.eye(n)

    for i in range(max_iterations):
        last_broyden_iterations += 1
        f_val = f(x)
        Jinv = np.linalg.inv(J)
        delta = np.dot(Jinv, -f_val)
        x = x + delta
        if np.linalg.norm(delta, ord=2) < accuracy:
            return x
        f_new = f(x)
        u = f_new - f_val
        v = delta
        J = J + np.outer(u - J.dot(v), v) / np.dot(v, v)

    return x
```



```
Iteration: 54
x = [ 1.64998819 -0.15795963]
f = [ 2.97817326e-14 -4.50097265e-10]
|f| = 1.0129377443026415e-19
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method: avoid matrix inversion

$$\mathbf{J}_n = \mathbf{J}_{n-1} + \frac{\Delta \mathbf{f}_n - \mathbf{J}_{n-1} \Delta \mathbf{x}_n}{\|\Delta \mathbf{x}_n\|^2} \Delta \mathbf{x}_n^T$$

Sherman-Morrison formula:

$$\mathbf{J}_n^{-1} = \mathbf{J}_{n-1}^{-1} + \frac{\Delta \mathbf{x}_n - \mathbf{J}_{n-1}^{-1} \Delta \mathbf{f}_n}{\Delta \mathbf{x}_n^T \mathbf{J}_{n-1}^{-1} \Delta \mathbf{f}_n} \Delta \mathbf{x}_n^T \mathbf{J}_{n-1}^{-1}$$

Update the inverse Jacobian directly!

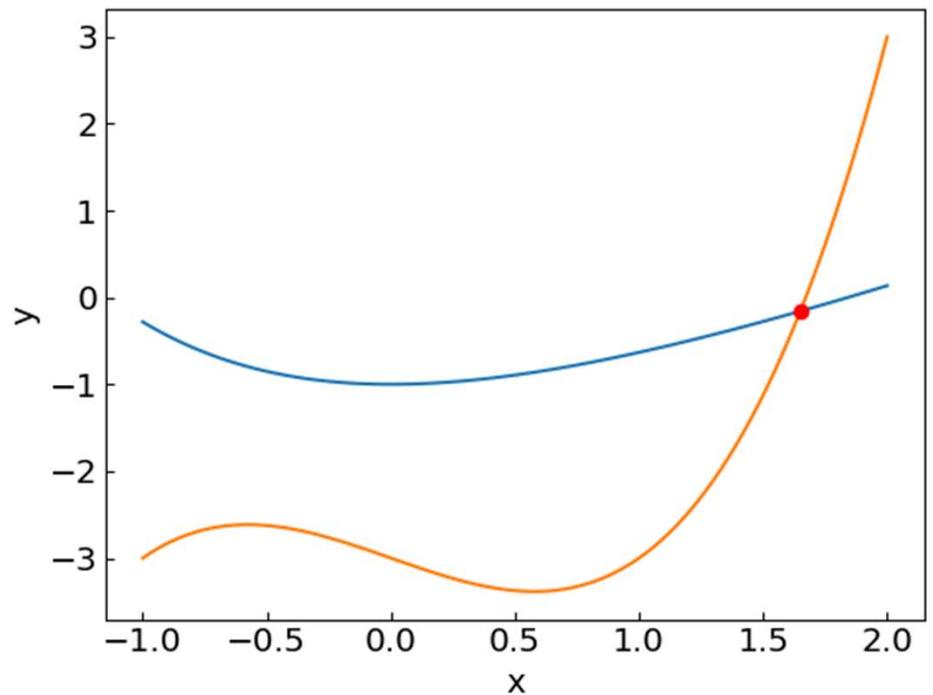
From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method (Sherman-Morrison)

```
def broyden_method(
    f,
    x0,
    accuracy=1e-8,
    max_iterations=100):
    global last_broyden_iterations
    last_broyden_iterations = 0
    x = x0
    n = x0.shape[0]
    Jinv = np.eye(n)

    for i in range(max_iterations):
        last_broyden_iterations += 1
        f_val = f(x)
        delta = -Jinv.dot(f_val)
        x = x + delta
        if np.linalg.norm(delta, ord=2) < accuracy:
            return x
        f_new = f(x)
        df = f_new - f_val
        dx = delta
        Jinv = Jinv + np.outer(dx - Jinv.dot(df), dx.T.dot(Jinv)) / np.dot(dx.T, Jinv.dot(df))

    return x
```



```
Iteration: 54
x = [ 1.64998819 -0.15795963]
f = [ 2.8255176e-14 -3.8877096e-10]
|f| = 7.557143001803891e-20
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method vs Newton-Raphson method

Broyden method converges somewhat slower (e.g. 54 vs 12 iterations in our example) but:

- Does not involve the calculation of Jacobian
- Does not involve matrix inversion

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Broyden method vs Newton-Raphson method

Broyden method converges somewhat slower (e.g. 54 vs 12 iterations in our example) but:

- Does not involve the calculation of Jacobian
- Does not involve matrix inversion

Possible refinement: improve the initial estimate for the Jacobian

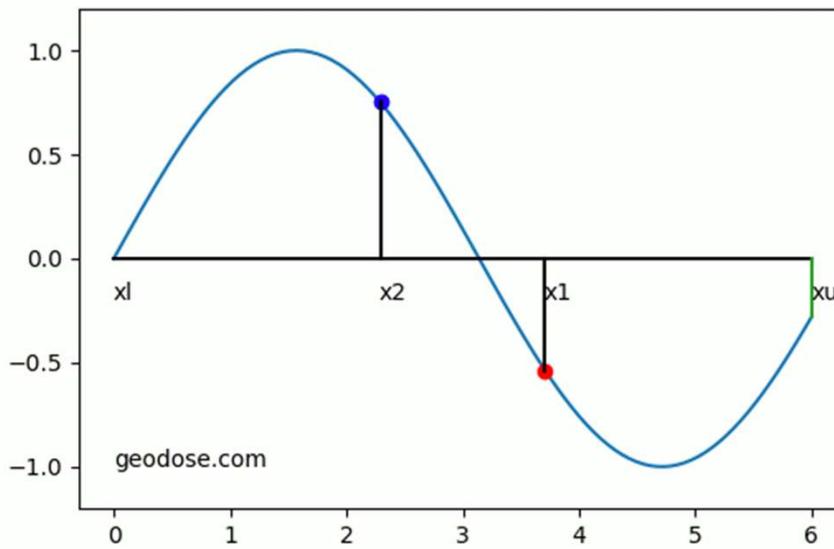
```
Iteration: 54
x = [ 1.64998819 -0.15795963]
f = [ 2.8255176e-14 -3.8877096e-10]
|f| = 7.557143001803891e-20
```



```
Iteration: 15
x = [ 1.64998819 -0.15795963]
f = [ 1.02683695e-11 1.31871458e-11]
|f| = 1.3967011340731408e-22
```

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Function minimization/maximization



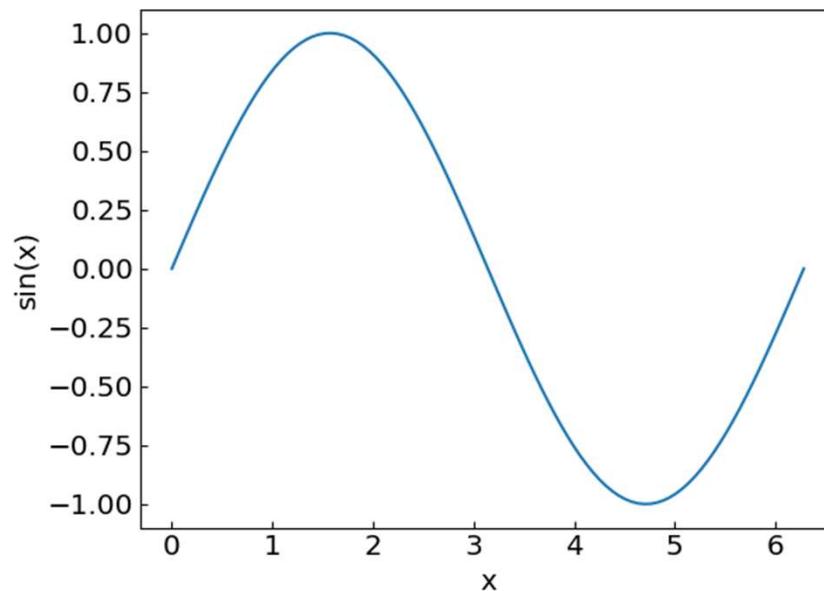
References: Chapter 6.4 of *Computational Physics* by Mark Newman
Chapter 10 of *Numerical Recipes Third Edition* by W.H. Press et al.

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Function extrema

Often we are interested to find the minimum of a function (e.g. energy minimization)

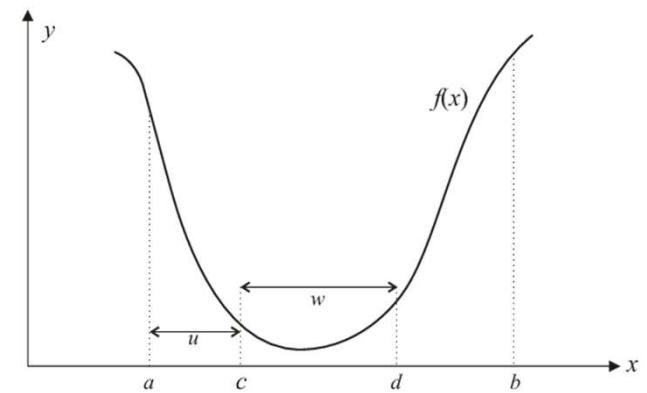
Consider the minimum of $f(x)=\sin(x)$ on interval $0..2\pi$



From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Golden section search

- Bracket the minimum x_{\min} in (a,b)
- Take $c = b - (b-a)/\varphi$ and $d = a + (b-a)/\varphi$
- if $f(c) < f(d)$, take $b = d$ as new right endpoint
- Otherwise, take $a = c$ as new left endpoint
- Repeat over the new interval (a,b) until the desired accuracy is reached



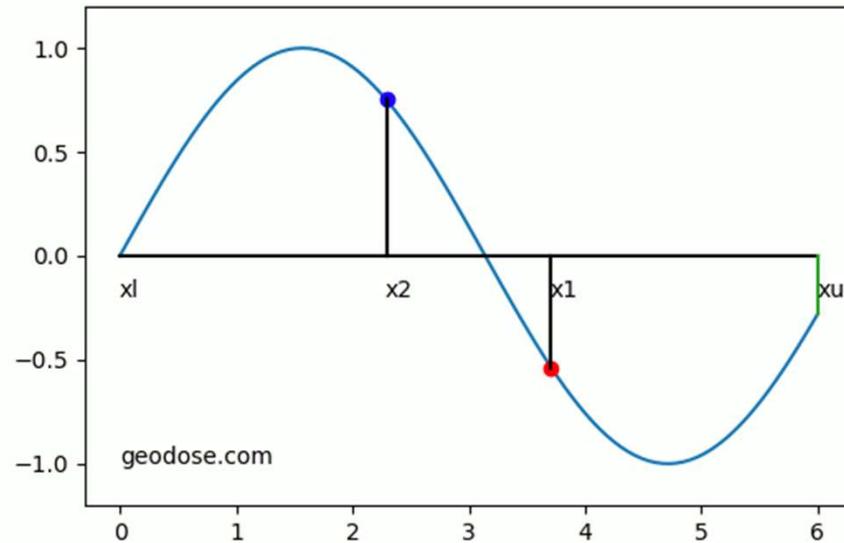
$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.618 \dots \quad \text{is the \b{golden ratio}}$$

This value ensures that the interval decreases by factor φ no matter what

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Golden section search

```
def gss(f, a, b, accuracy=1e-7):
    c = b - (b - a) / phi
    d = a + (b - a) / phi
    while abs(b - a) > accuracy:
        if f(c) < f(d):
            b = d
        else:
            a = c
        c = b - (b - a) / phi
        d = a + (b - a) / phi
    return (b + a) / 2
```



The minimum of $\sin(x)$ over the interval (0.0 , 6.283185307179586) is 4.712388990891052

To search for maximum of $f(x)$ look for minimum of $-f(x)$

From the course by Volodymyr Vovchenko,
<https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Newton method

The extremum of $f(x)$ is the root of the derivative, $f'(x) = 0$

Simply apply Newton-Raphson method for finding the root of $f'(x)$

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

$f''(x) > 0$, \rightarrow minimum

$f''(x) < 0$, \rightarrow maximum

```
def newton_extremum(f, df, d2f, x0, accuracy=1e-7, max_iterations=100):
    xprev = xnew = x0
    for i in range(max_iterations):
        xnew = xprev - df(xprev) / d2f(xprev)

        if (abs(xnew-xprev) < accuracy):
            return xnew

    xprev = xnew
return xnew
```

An extremum of $\sin(x)$ using Newton's method starting from $x_0 = 5.0$ is (0.0 , 6.283185307179586) is 4.71238898038469

Gradient descent method

Replace, $f''(x)$ by a descent factor $1/\gamma_n$

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

```
def gradient_descent(f, df, x0, gam = 0.01, accuracy=1e-7, max_iterations=100):
    xprev = x0
    for i in range(max_iterations):
        xnew = xprev - gam * df(xprev)

        if (abs(xnew-xprev) < accuracy):
            return xnew

        xprev2 = xprev
        xprev = xnew
    return xnew
```

From the course by
Volodymyr Vovchenko,
<https://github.com/vlvoch/PHYS6350-ComputationalPhysics>

Freedom in choosing γ_n

Can be generalized to multi-variable function $F(x_1, x_2, \dots)$

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$