Computational Physics (PHYS6350)

Lecture 7: Non-linear equations and root-finding: Part 2

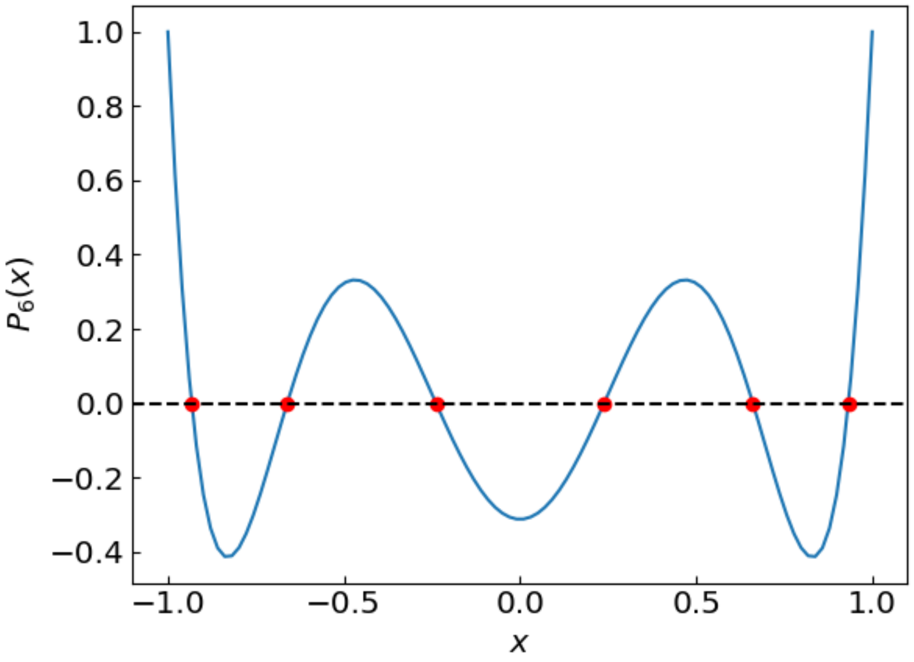
- Roots of polynomials
- Systems of non-linear equations
- Function extrema

February 6, 2025

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics/tree/spring2025

Roots of polynomials

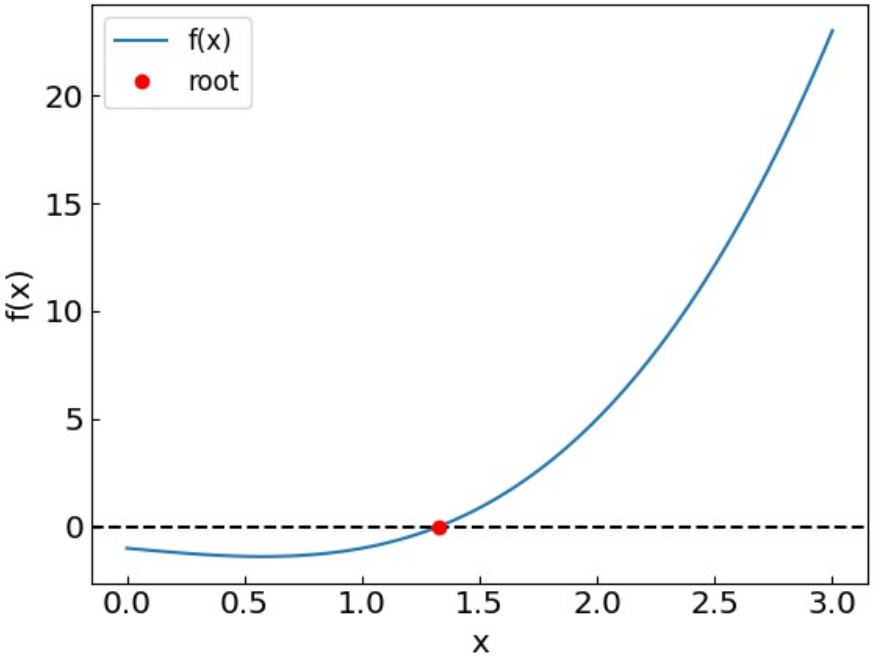


References: Chapters 5.1, 9.5 of *Numerical Recipes Third Edition* by W.H. Press et al.

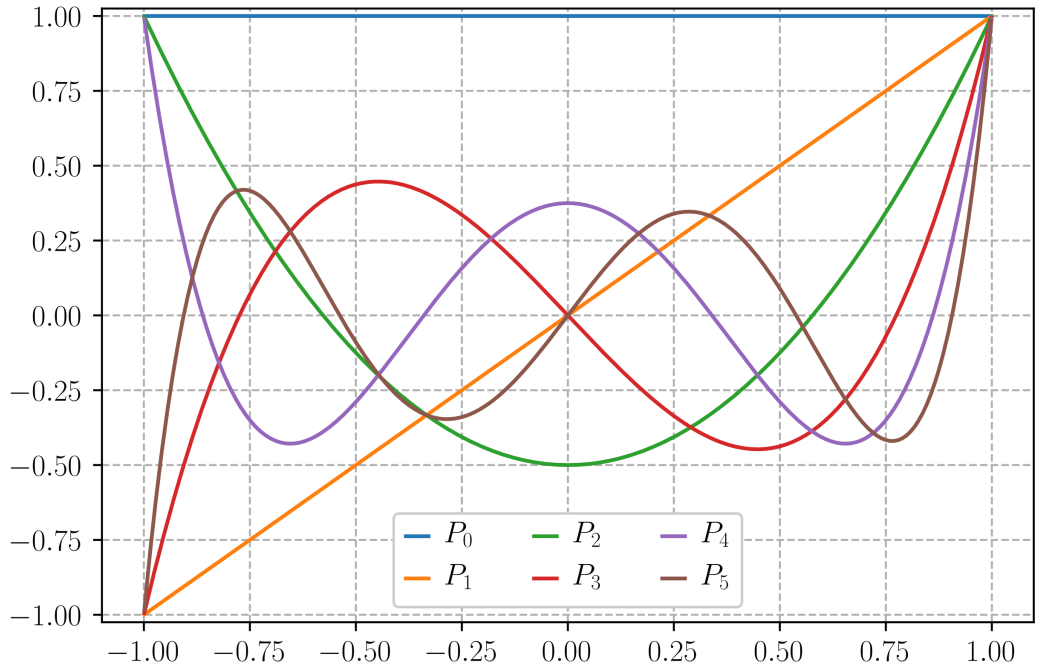
Roots of polynomials

So far we've dealt with polynomials with one real root, such as

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



Other polynomials (e.g. **Legendre polynomials**) have multiple real roots, and we need to calculate them all



Preliminaries: evaluating polynomials efficiently

A polynomial can typically be written as

$$P(x) = \sum_{j=0}^{n} a_j\, x^j$$

or, equivalently, as

$$P(x) = a_0 + x(a_1 + x(\ldots))$$

which allows one to evaluate both the polynomial and its derivative efficiently

```
def Poly(x,a):
    ret = a[len(a) - 1]
    for j in range(len(a) - 2, -1, -1):
        ret = ret * x + a[j]
    return ret
```

```
# Evaluate the derivative of a polynomial
# with coefficients a at a point x
def dPoly(x,a):
    p = a[len(a) - 1]
    dp = 0.
    for j in range(len(a) - 2, -1, -1):
        dp = dp * x + p
        p = p * x + a[j]
    return dp
```

Preliminaries: multiplying and dividing a polynomial

Multiplication:

Multiply $P(x) = \sum_{j=0}^{n} a_j\, x^j$ by $(x - c)$ to get $\tilde{P}(x) = (x - c)\, P(x) = \sum_{j=0}^{n+1} \tilde{a}_j x^j$.

Easy to see that

$$\tilde{a}_0 = -c a_0, \qquad \text{and} \qquad \tilde{a}_j = a_{j-1} - c\, a_j, \qquad j = 1, \ldots, n+1$$

```python
# Multiply polynomial by (x - c)
def PolyMult(a,c):
    n = len(a)
    ret = a[:]
    ret.append(ret[-1])
    for j in range(n-1,0,-1):
        ret[j] = ret[j-1] - c * ret[j]
    ret[0] = -c * ret[0]
    return ret
```

Division:

Inverting these relations defines the division of $\tilde{P}(x)$ by (x-c)

$$a_j = \tilde{a}_{j+1} + c\, a_{j+1}, \quad j = 0, \ldots, n$$

Note that the division only makes sense when $x{=}c$ is a root of $\tilde{P}(x)$

```python
# Divide the polynomial by (x - c),
# assuming x = c is one of the roots
def PolyDiv(a,c):
    n = len(a) - 1
    ret = a[:]
    ret[-1] = 0.
    for j in range(n-1,-1,-1):
        ret[j] = a[j+1] + c * ret[j+1]
    ret.pop()
    return ret
```

Roots of Legendre polynomials

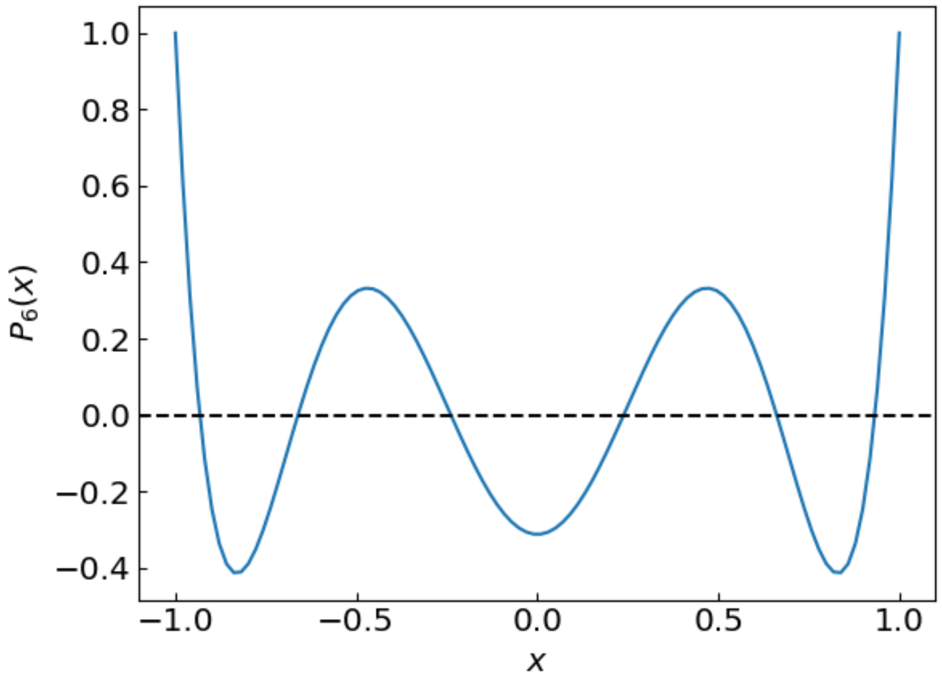
Roots of Legendre polynomials $P_n(x)$ play an important role e.g. for numerical integration using quadratures

Each $P_n(x)$ has n real roots in the interval x = -1...1

Consider

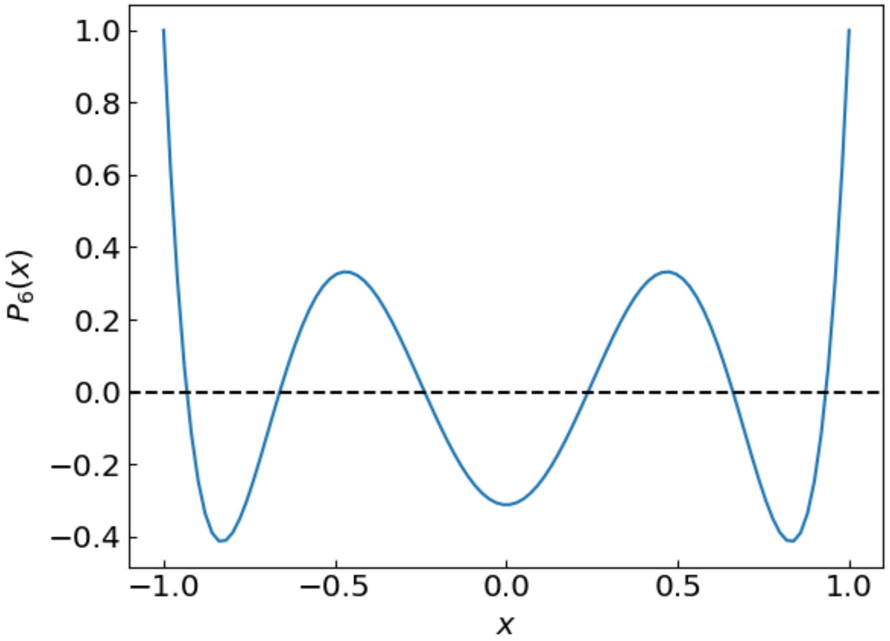
$$P_6(x) = \frac{1}{16}\left(231x^6 - 315x^4 + 105x^2 - 5\right)$$

How to evaluate its six roots accurately?



Roots of Legendre polynomials

Strategy 1: Bracket each root from visual analysis and use the bisection method for refinement

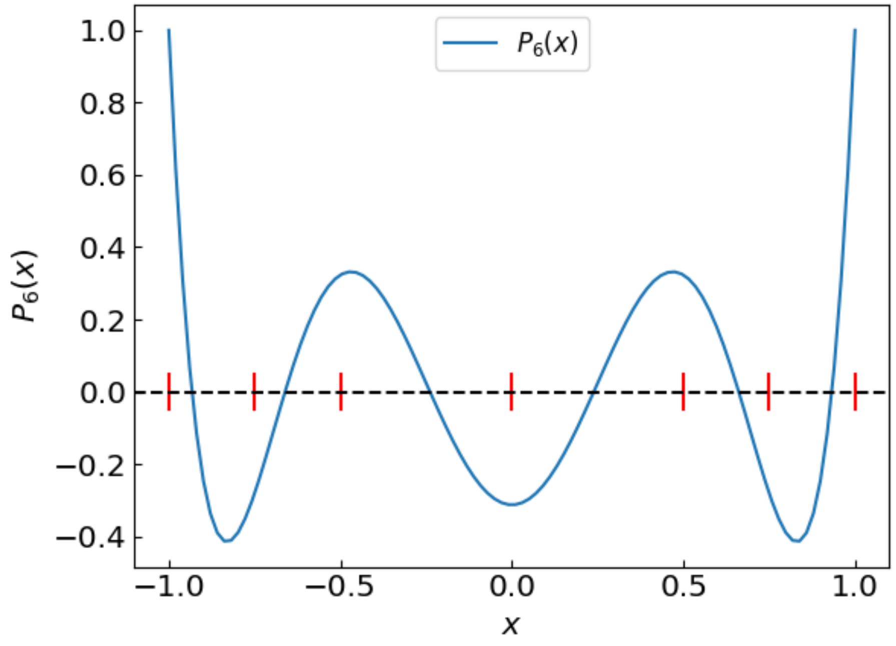


Roots of Legendre polynomials

Strategy 1: Bracket each root from visual analysis and use the bisection method for refinement

```
xroots = []

# Root 1
xleft = -1.
xright = -0.75
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 1 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = -0.75
xright = -0.5
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 2 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = -0.5
xright = 0.
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 3 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.
xright = 0.5
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 4 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.5
xright = 0.75
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 5 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.75
xright = 1.
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 6 between", xleft, "and", xright, "is x =",xroots[-1])
```

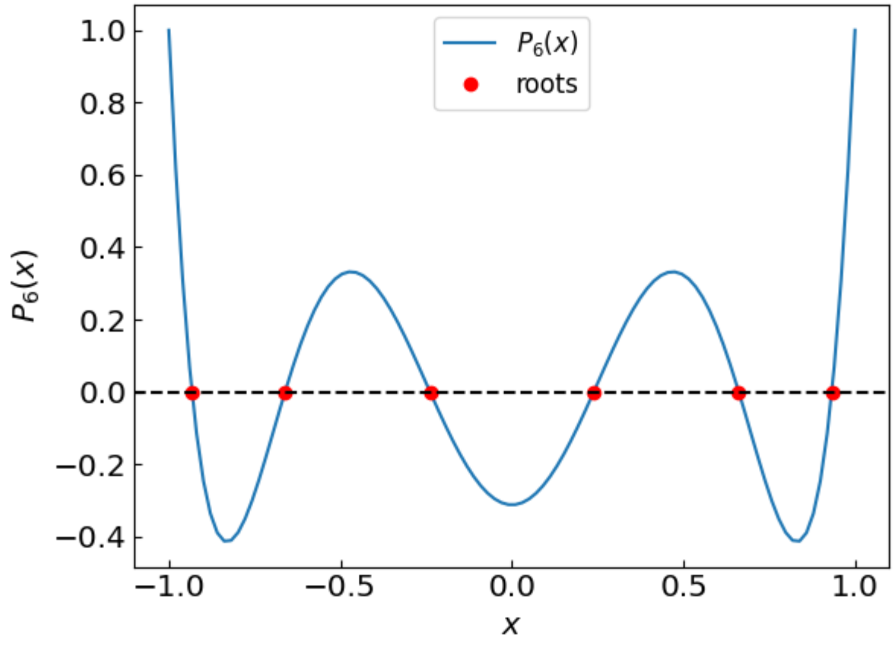


Roots of Legendre polynomials

Strategy 1: Bracket each root from visual analysis and use the bisection method for refinement

```
xroots = []

# Root 1
xleft = -1.
xright = -0.75
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 1 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = -0.75
xright = -0.5
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 2 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = -0.5
xright = 0.
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 3 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.
xright = 0.5
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 4 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.5
xright = 0.75
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 5 between", xleft, "and", xright, "is x =",xroots[-1])
xleft = 0.75
xright = 1.
xroots.append(bisection_method(fP6,xleft,xright))
print("Root 6 between", xleft, "and", xright, "is x =",xroots[-1])
```




```
Root 1 between -1.0 and -0.75 is x = -0.9324695142277051
Root 2 between -0.75 and -0.5 is x = -0.6612093864532653
Root 3 between -0.5 and 0.0 is x = -0.2386191860714461 7
Root 4 between 0.0 and 0.5 is x = 0.2386191860714461 7
Root 5 between 0.5 and 0.75 is x = 0.6612093864532653
Root 6 between 0.75 and 1.0 is x = 0.9324695142277051
```

Roots of Legendre polynomials

Strategy 1 is fairly fail-safe but requires significant manual pre-processing

Strategy 2:

1. Use one of the standard methods (e.g. Newton-Raphson) to find the first root x_1
2. Divide the polynomial by $(x-x_1)$
3. Apply Newton's method to the new polynomial to find x_2
4. Divide the polynomial by $(x-x_2)$ and repeat the above steps until all roots are found

Optional optimization:
Refine the roots by applying Newton-Raphson method again to the original polynomial, using the tentative roots as initial guesses
This helps to mitigate round-off error accumulation inherent in polynomial division

Roots of Legendre polynomials

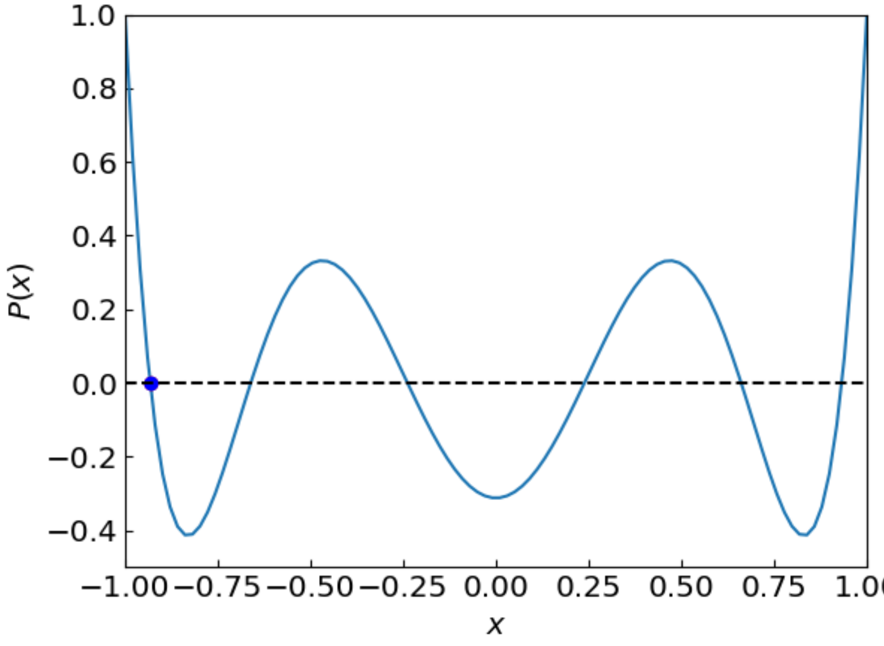
Using strategy 2, initial guess $x_0 = -1$

```python
def PolyRoots(
    a,                       # The coefficients of the polynomial that we are solving
    x0  = -1.,               # The initial guess for the first root
    accuracy = 1.e-10,       # The desired accuracy of the solution
    polishing = True,        # Whether to polish the roots further with Newton's method
    max_iterations = 100     # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
```

Roots of Legendre polynomials

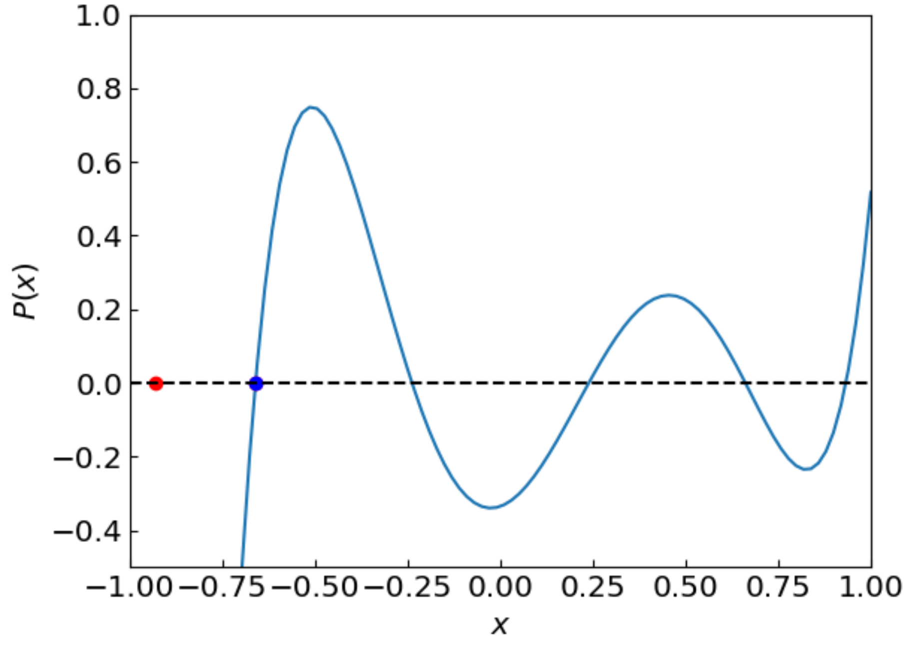
Using strategy 2, initial guess $x_0 = -1$

```
def PolyRoots(
    a,                      # The coefficients of the polynomial that we are solving
    x0  = -1.,              # The initial guess for the first root
    accuracy = 1.e-10,      # The desired accuracy of the solution
    polishing = True,       # Whether to polish the roots further with Newton's method
    max_iterations = 100    # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
```

Roots of Legendre polynomials

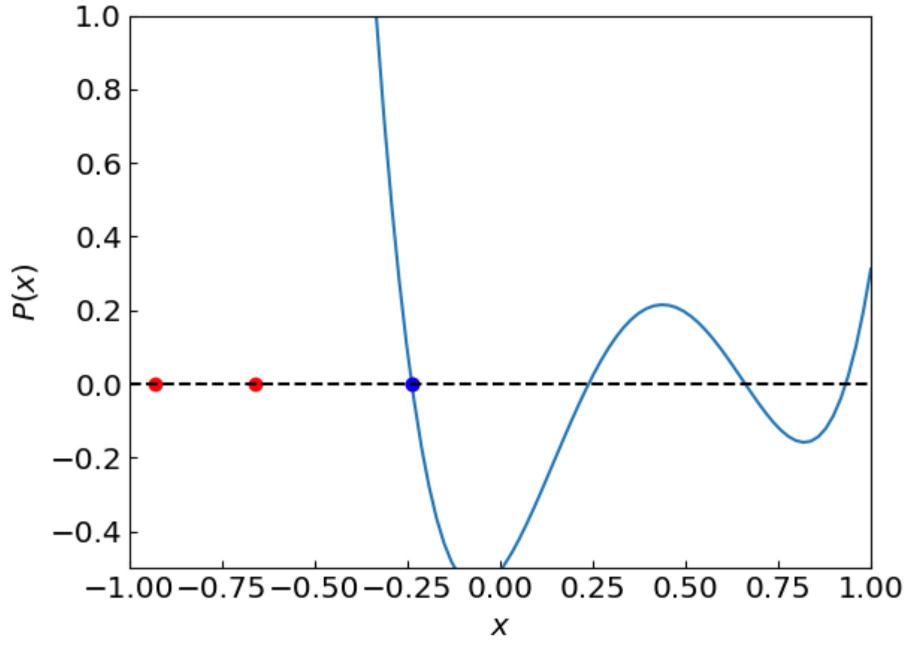
Using strategy 2, initial guess $x_0 = -1$

```
def PolyRoots(
    a,                      # The coefficients of the polynomial that we are solving
    x0  = -1.,              # The initial guess for the first root
    accuracy = 1.e-10,      # The desired accuracy of the solution
    polishing = True,       # Whether to polish the roots further with Newton's method
    max_iterations = 100    # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
Root  3 is x =  -0.23861918608319668
```

Roots of Legendre polynomials

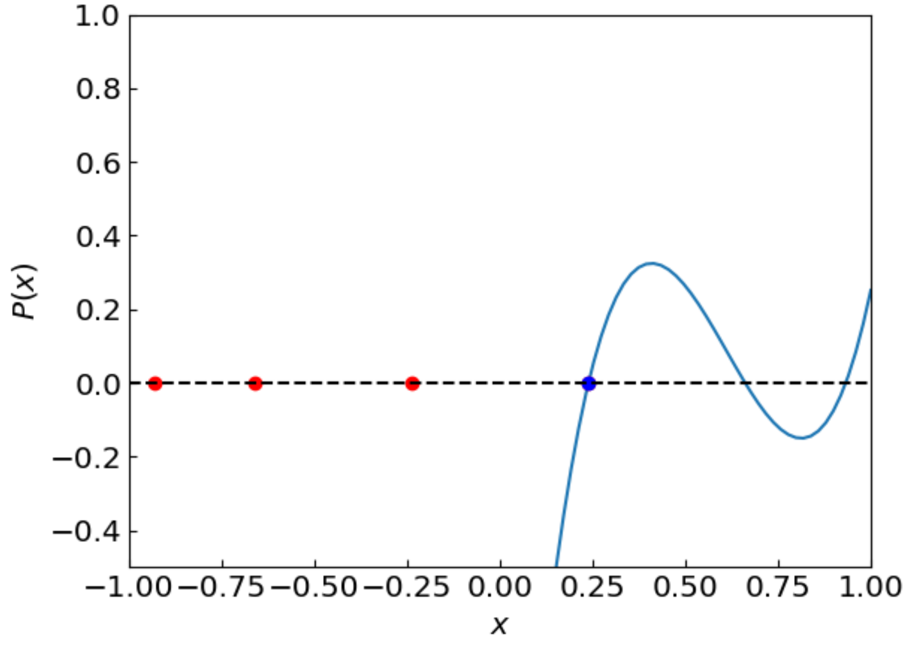
Using strategy 2, initial guess $x_0 = $ -1

```
def PolyRoots(
    a,                      # The coefficients of the polynomial that we are solving
    x0  = -1.,              # The initial guess for the first root
    accuracy = 1.e-10,      # The desired accuracy of the solution
    polishing = True,       # Whether to polish the roots further with Newton's method
    max_iterations = 100    # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
Root  3 is x =  -0.23861918608319668
Root  4 is x =  0.23861918608319652
```

Roots of Legendre polynomials

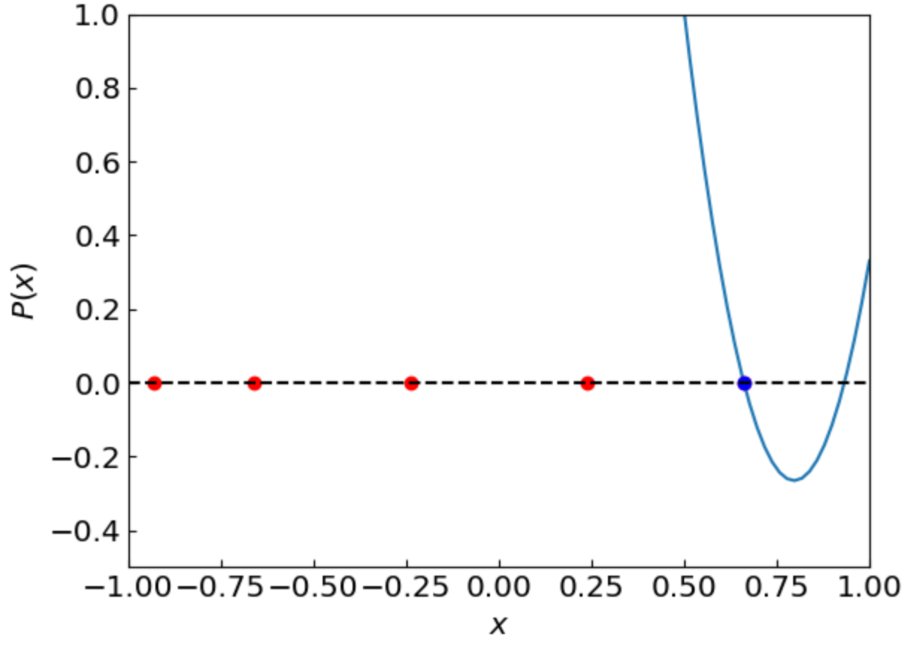
Using strategy 2, initial guess $x_0 = -1$

```
def PolyRoots(
    a,                      # The coefficients of the polynomial that we are solving
    x0  = -1.,              # The initial guess for the first root
    accuracy = 1.e-10,      # The desired accuracy of the solution
    polishing = True,       # Whether to polish the roots further with Newton's method
    max_iterations = 100    # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
Root  3 is x =  -0.23861918608319668
Root  4 is x =  0.23861918608319652
Root  5 is x =  0.6612093864662646
```

Roots of Legendre polynomials

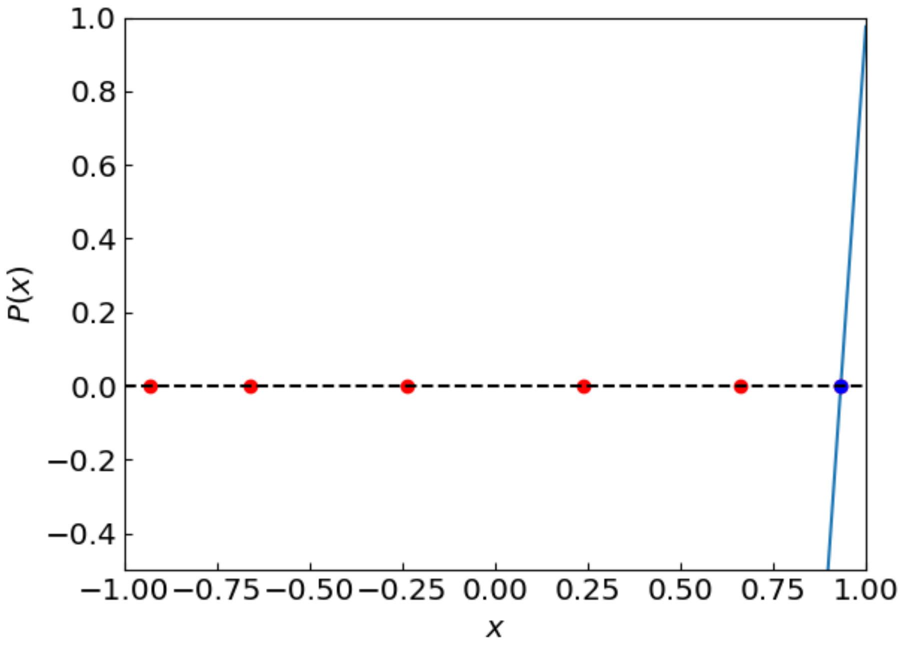
Using strategy 2, initial guess $x_0 = -1$

```
def PolyRoots(
    a,                      # The coefficients of the polynomial that we are solving
    x0  = -1.,              # The initial guess for the first root
    accuracy = 1.e-10,      # The desired accuracy of the solution
    polishing = True,       # Whether to polish the roots further with Newton's method
    max_iterations = 100    # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```




```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
Root  3 is x =  -0.23861918608319668
Root  4 is x =  0.23861918608319652
Root  5 is x =  0.6612093864662646
Root  6 is x =  0.9324695142031523
```

Roots of Legendre polynomials

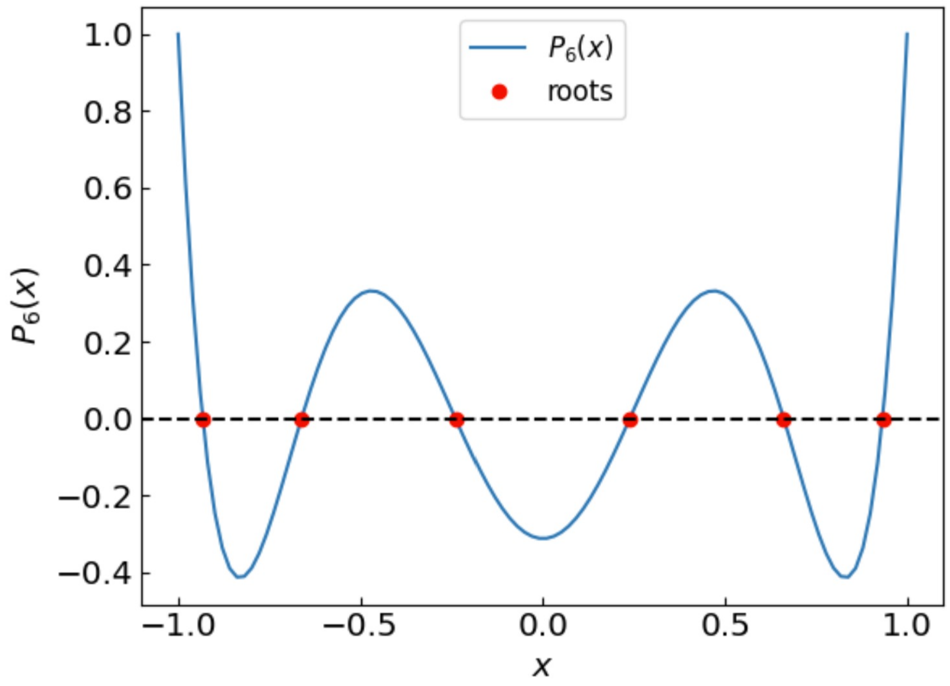
Using strategy 2, initial guess $x_0 = -1$

```python
def PolyRoots(
    a,                       # The coefficients of the polynomial that we are solving
    x0  = -1.,               # The initial guess for the first root
    accuracy = 1.e-10,       # The desired accuracy of the solution
    polishing = True,        # Whether to polish the roots further with Newton's method
    max_iterations = 100     # Maximum number of iterations in Newton's method
):
    ret = []
    n = len(a)
    apoly = a[:]
    current_root = x0

    def f(x):
        return Poly(x,apoly)
    def df(x):
        return dPoly(x,apoly)

    print("Searching all the roots using deflation and the Newton's method")
    # Loop over all the roots
    for k in range(0,n-1,1):
        current_root = newton_method(f,df,current_root,accuracy,max_iterations)
        if (current_root == None):
            print("Failed to find the next root!")
            break
        ret.append(current_root)
        print("Root ", k+1, "is x = ",current_root)
        # Deflate the polynomial
        apoly = PolyDiv(apoly, current_root)

    return ret
```

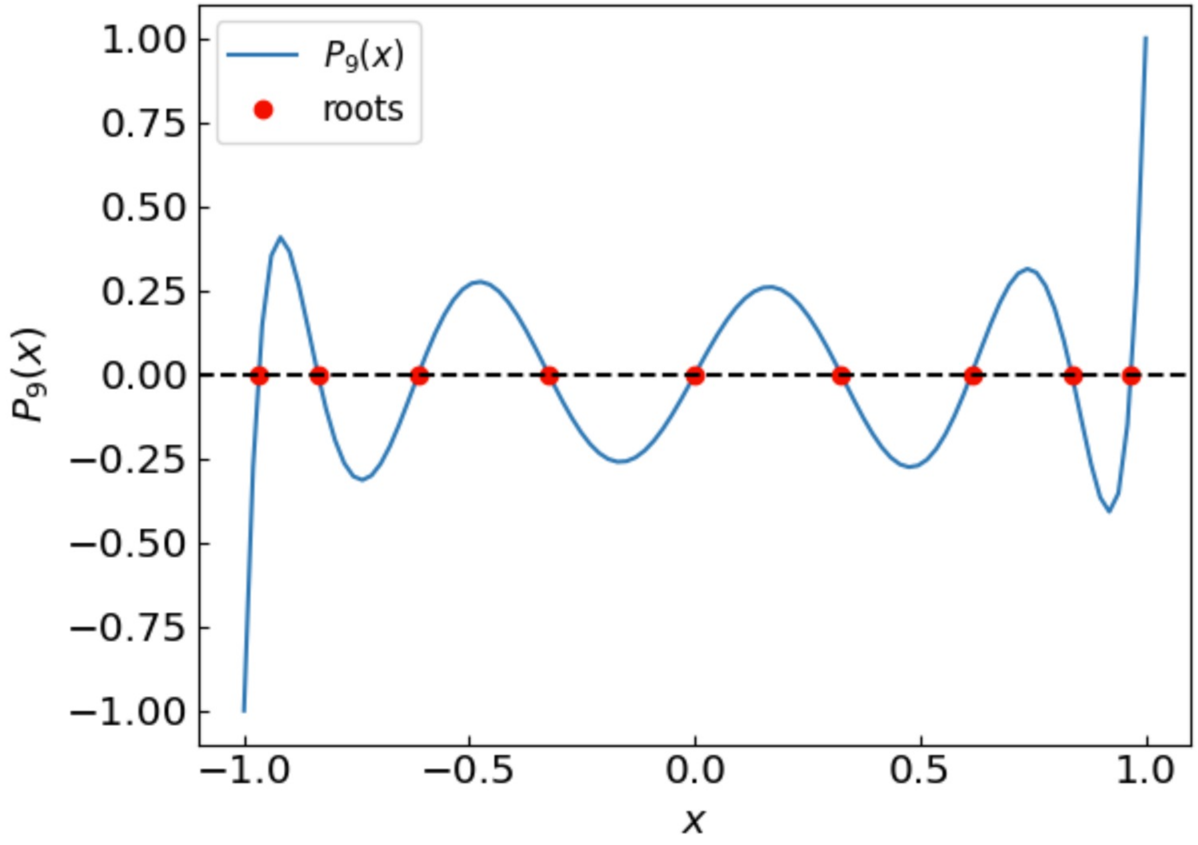



```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.932469514203152
Root  2 is x =  -0.6612093864662645
Root  3 is x =  -0.23861918608319668
Root  4 is x =  0.23861918608319652
Root  5 is x =  0.6612093864662646
Root  6 is x =  0.9324695142031523
```

Roots of Legendre polynomials

Same procedure for $P_9(x)$

```
Searching all the roots using deflation and Newton's method
Root  1 is x =  -0.9681602395076263
Root  2 is x =  -0.8360311073266355
Root  3 is x =  -0.6133714327005905
Root  4 is x =  -0.3242534234038087
Root  5 is x =  -2.9050086835705924e-16
Root  6 is x =  0.32425342340380925
Root  7 is x =  0.6133714327005846
Root  8 is x =  0.8360311073266581
Root  9 is x =  0.968160239507609
```



Try to go higher order? How high can we go?

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

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

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

Vector notation: $\mathbf{f} = (f_1,\dots,f_N)$ and $\mathbf{x} = (x_1,\dots,x_N)$

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

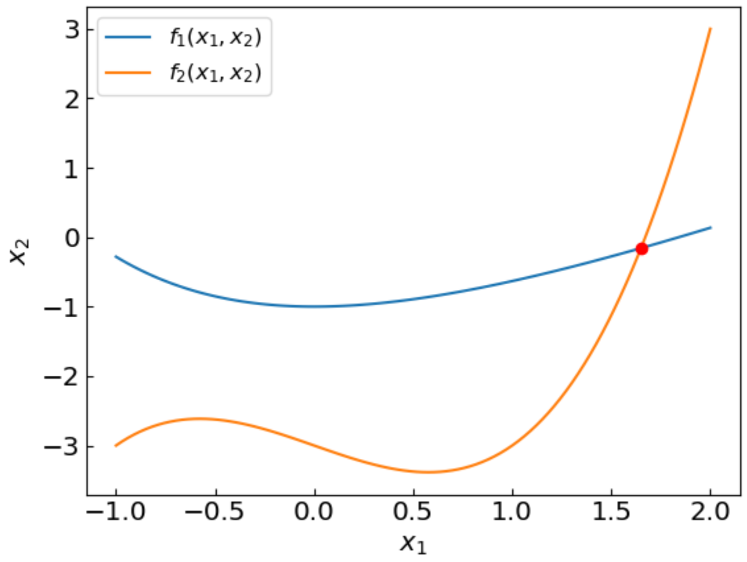
For example:

$$\begin{aligned} x + \exp(-x) - 2 - y &= 0, \\ x^3 - x - 3 - y &= 0. \end{aligned}$$

i.e.

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

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



Newton-Raphson method in multiple dimensions

We have a system on non-linear equations $\mathbf{f(x)} = \mathbf{0}$

Taylor expansion around the root $\mathbf{x^*}$ reads (multi-variate calculus)

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

1D: $f(x^*) \approx f(x) + f'(x)(x^* - x)$

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

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

$$\mathbf{f}(\boldsymbol{x}^*) = \mathbf{0} \quad \Rightarrow \quad \boldsymbol{J}(\boldsymbol{x})(\boldsymbol{x}^* - \boldsymbol{x}) \approx -\boldsymbol{f}(\boldsymbol{x}) \quad \Rightarrow \quad \boldsymbol{x}^* \approx x - \boldsymbol{J}^{-1}(\boldsymbol{x})\, \boldsymbol{f}(\boldsymbol{x})$$

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

$$x_{n+1} = x_n - \boldsymbol{J}^{-1}(\boldsymbol{x_n})\, \boldsymbol{f}(\boldsymbol{x_n})$$

In 1D reduces to Newton-Raphson: $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$

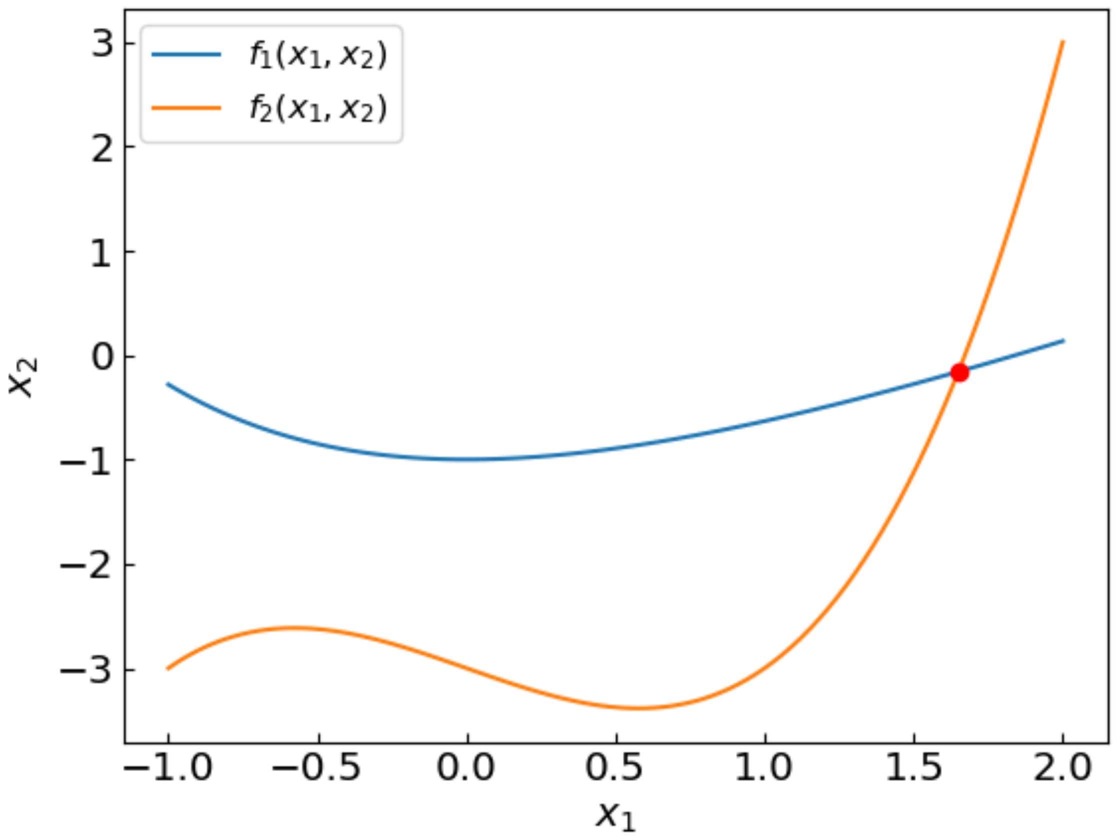
Newton-Raphson method in multiple dimensions

```python
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
```



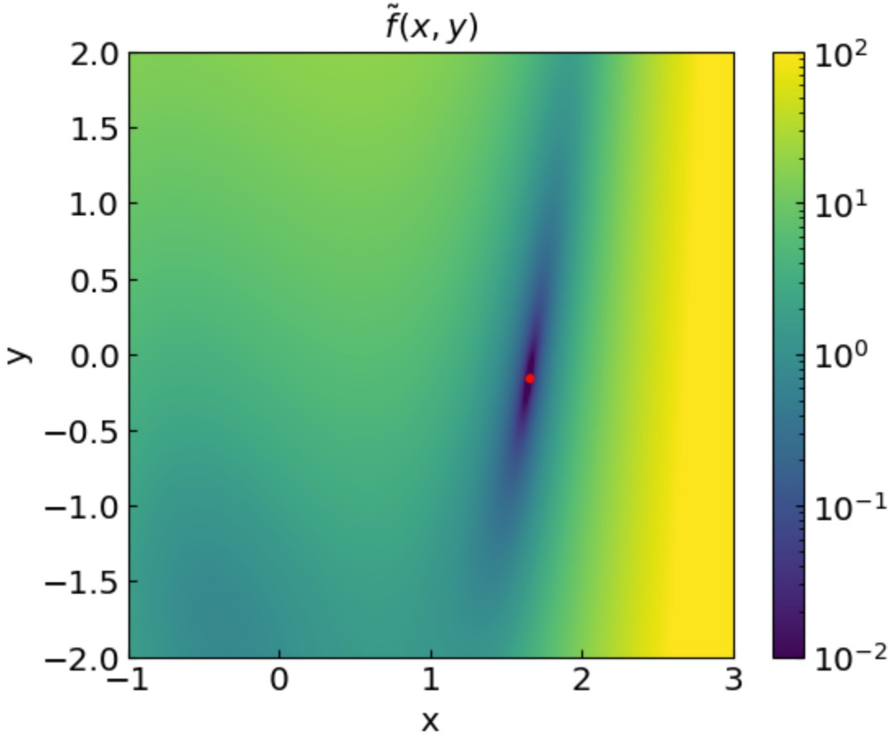

```
Iteration:  12
x =  [ 1.64998819 -0.15795963]
f =  [ 0.00000000e+00 -6.66133815e-16]
|f| =  2.2186712959340957e-31
```

Newton-Raphson method in multiple dimensions

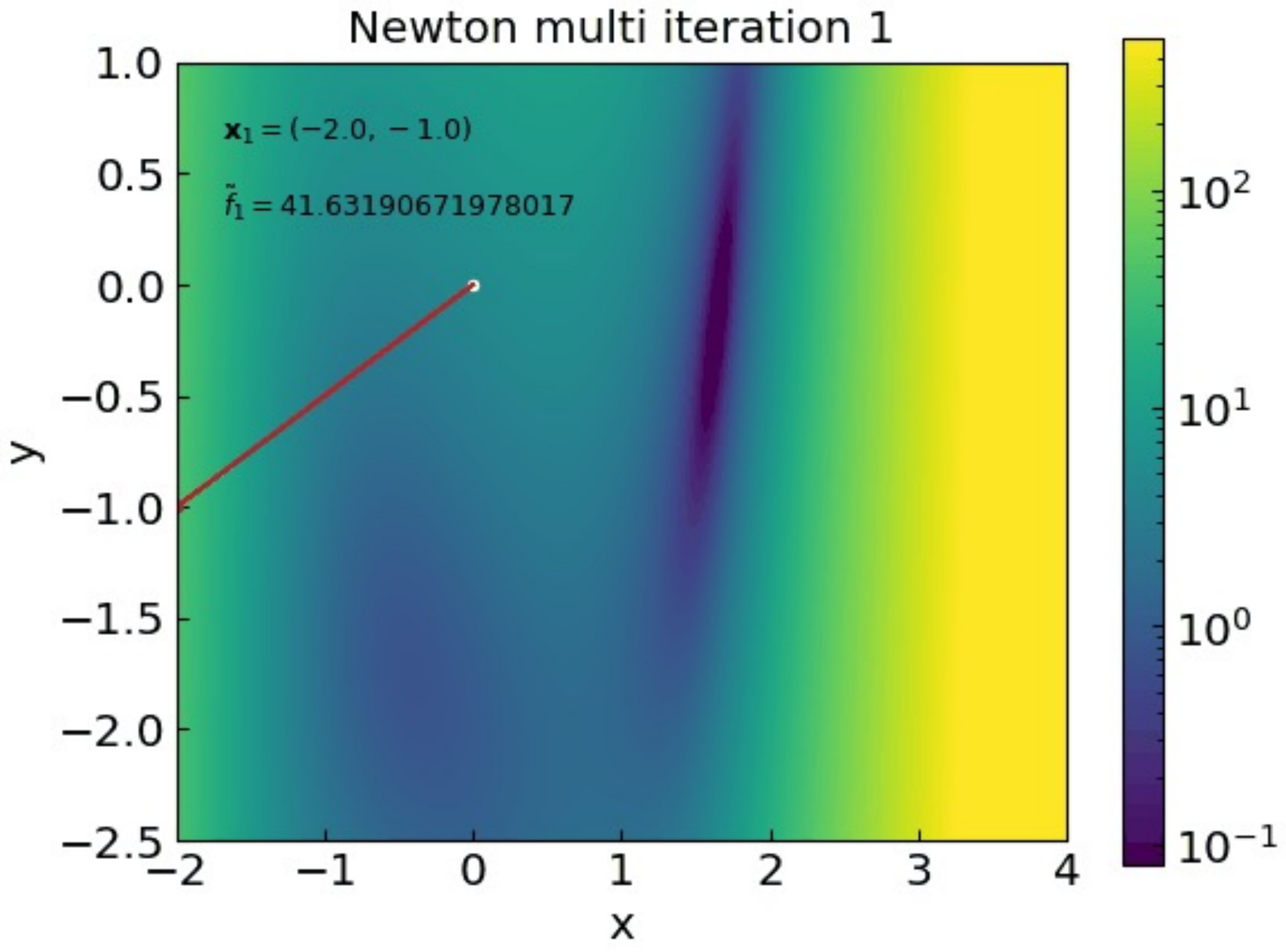
Introduce an objective function

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

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



Newton-Raphson method in multiple dimensions



Broyden method

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

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

The solution for $J(x_n)$ is not unique

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

Initial Jacobian J_0:

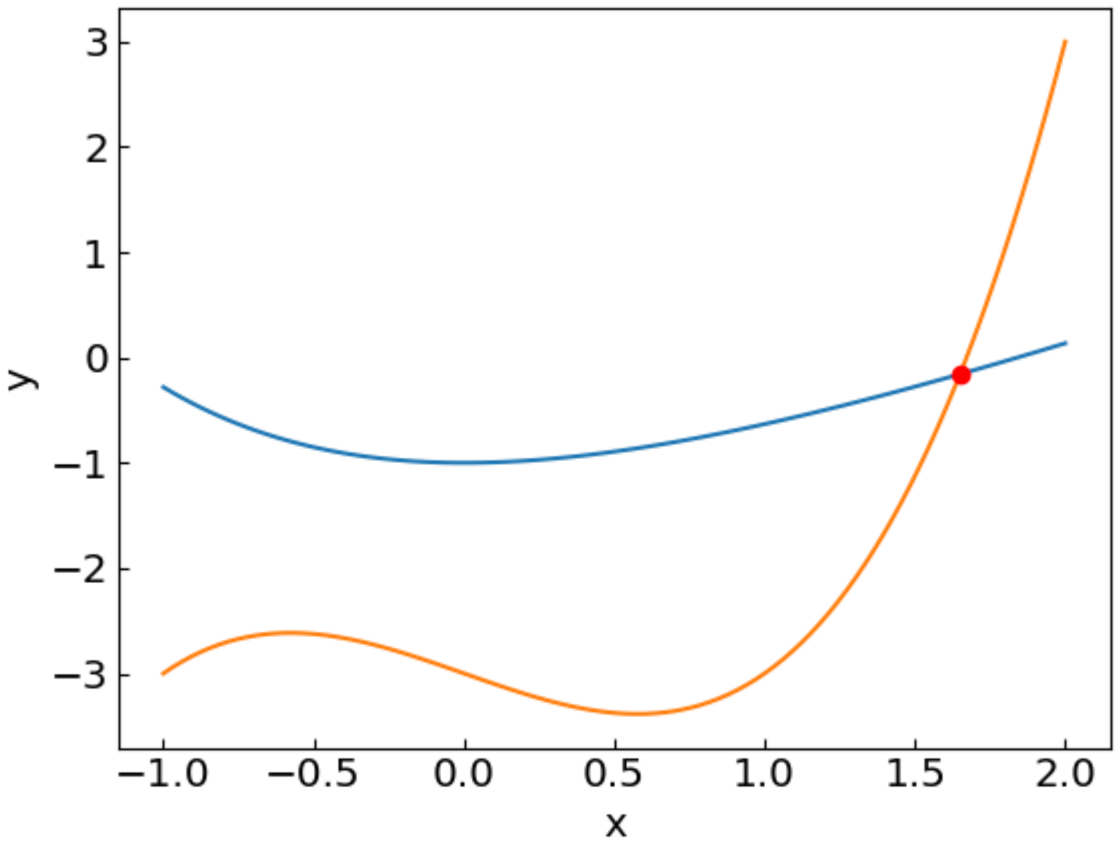
- Calculate the Jacobian $\mathbf{J(x_0)}$ — requires derivative but more accurate
- Initialize with Identity matrix $\mathbf{J(x_0) = I}$ — no derivative but can converge slower

Broyden method (direct)

```python
# 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
```

Broyden method: avoid matrix inversion

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

Sherman-Morrison formula:

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

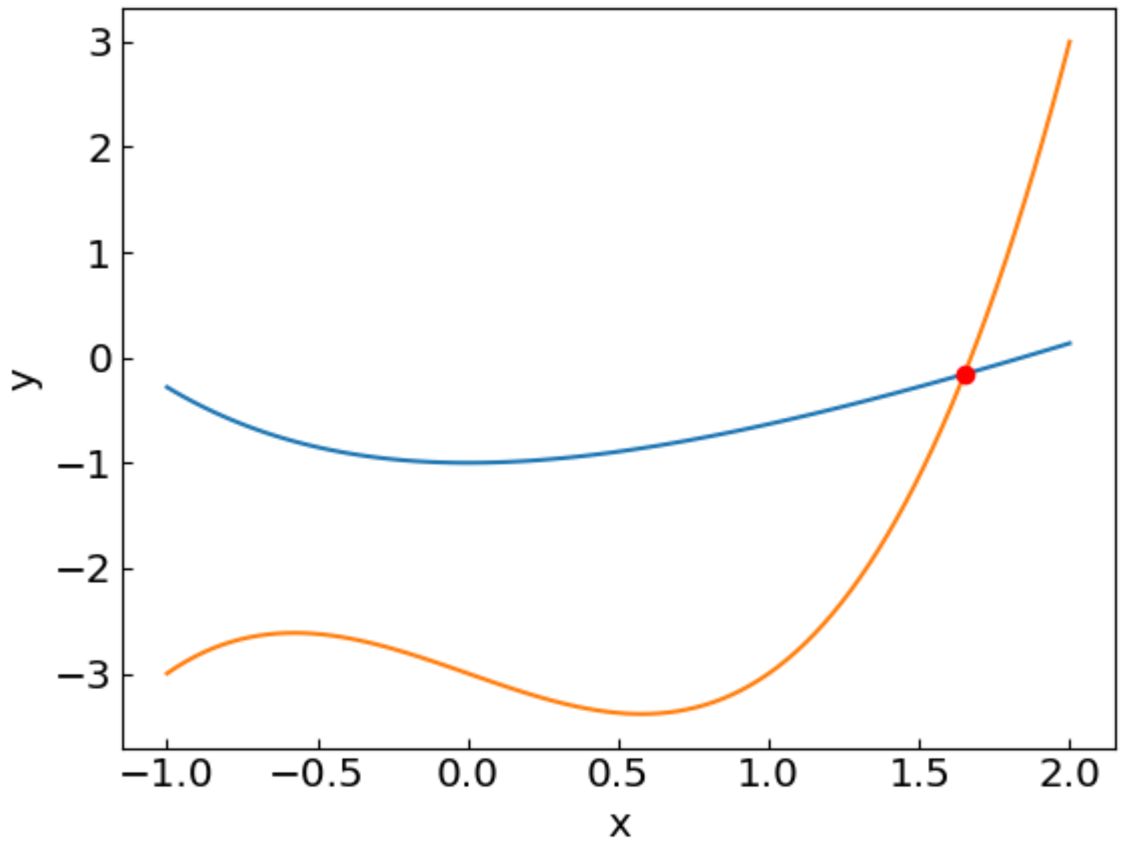
Update the inverse Jacobian directly!

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
```

Broyden method vs Newton-Raphson method

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

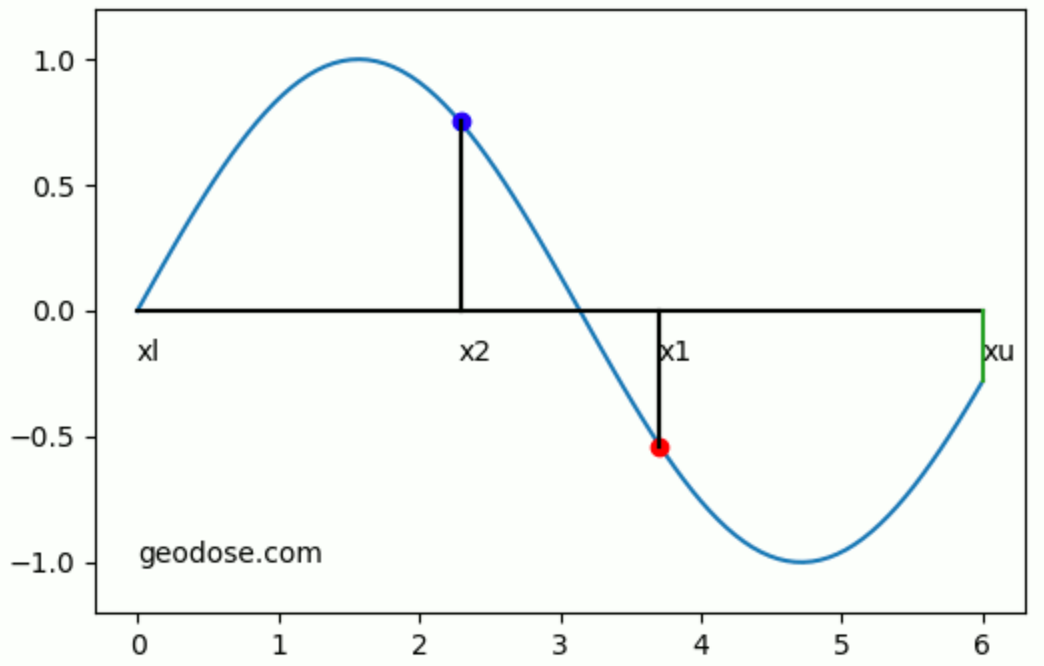
- Does not involve the calculation of Jacobian
- Does not involve matrix inversion

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
```

Function minimization/maximization

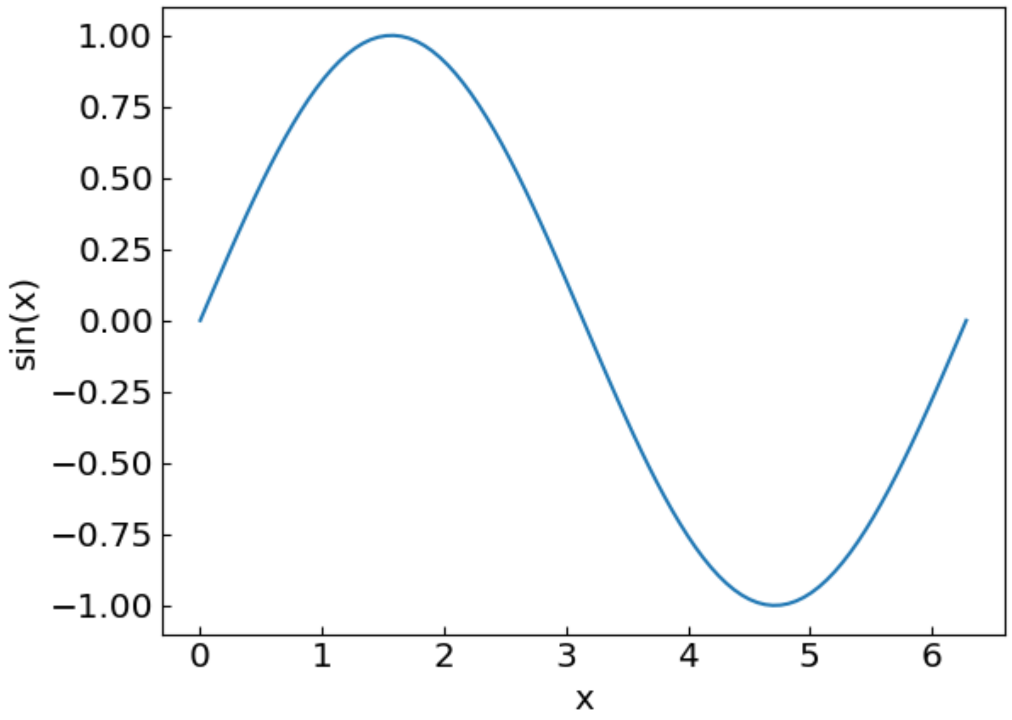


References: Chapter 6.4 of *Computational Physics* by Mark Newman
Chapter 10 of *Numerical Recipes Third Edition* by W.H. Press et al.

Function extrema

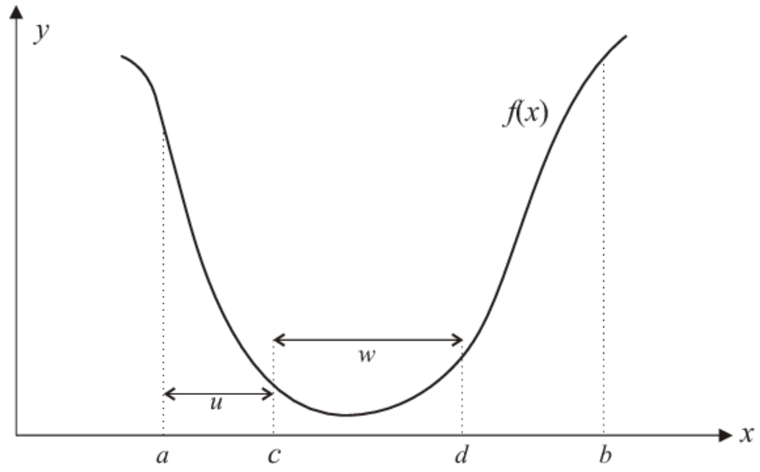
Often we are interested to find the minimum of a function (e.g. energy minimization)

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



Golden section search

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



$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618\ldots$$ is the **golden ratio**

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

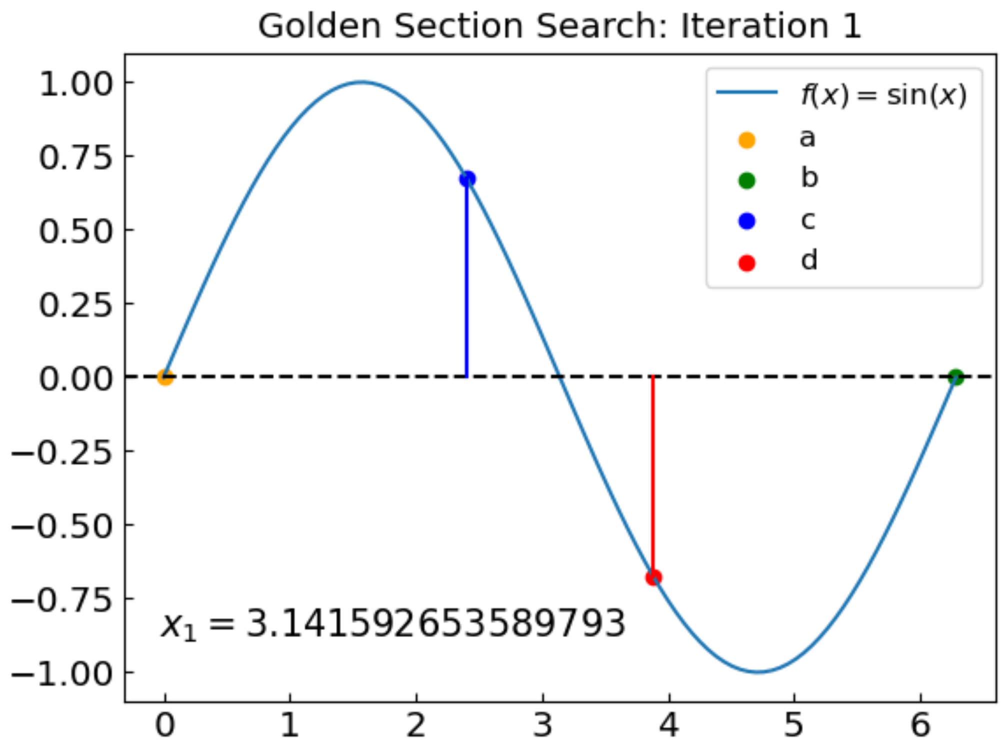
The method works when the function is **unimodal**

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 4.71238890891052
```

To search for a maximum of f(x) look for a minimum of –f(x)

Newton-Raphson method

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

Simply apply Newton-Raphson method (or one other standard methods) for finding the root of $f'(x)$

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

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

$f''(x) < 0, \quad \rightarrow \quad$ 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 x0 =  5.0  is ( 0.0 , 6.283185307179586 ) is 4.71238898038469
```

Gradient descent method

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

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

$\gamma_n > 0$ (minimum)

$\gamma_n < 0$ (minimum)

```
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
```

Freedom in choosing γ_n

Can be generalized to multi-variable function $F(x_1,x_2,\ldots)$ *final project idea(?)*

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