Polynomial roots

Contents

Polynomial roots#

Polynomial roots are *nice* because we can solve for them directly. Polynomials of degree n have exactly n roots, including multiplicity (coincident roots). The roots may be complex, even in polynomials with real

coefficients. If the root is complex ($a+bi$), then its *complex conjugate* ($a-bi$) is also a root.

The roots of polynomials of order ≤ 4 can be solved analytically.

E.g.:

$a x^2 +b x + c = 0$

has roots:

$x = \frac{-b \pm \sqrt{b^2-4ac}} {2a}$

per the quadratic formula.

Certaint polynomials of order $\gt 4$ may have analytically solvable roots but there is no such general formula per the Abel-Ruffini theorem.

All the roots of polynomials may be found numerically by solving for the eigenvalues of the polynomial *companion matrix*. This is the basis for modern numerical methods which we will cover when we get to matrix eigenvalue calculations.

Example usage of numerical root finding tools.#

```
# prompt: Give me a 10th order polynomial with integer coefficients,, prin

import numpy as np
from scipy.linalg import companion

# Define the coefficients of the polynomial
coefficients = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

# Print the polynomial
print("Polynomial:")
print(np.poly1d(coefficients))

# Calculate the companion matrix
companion_matrix = companion(coefficients)

# Print the companion matrix
print("\nCompanion Matrix:")
print(np.round(companion_matrix, 2))

# Find the eigenvalues (roots) of the companion matrix
roots = np.linalg.eigvals(companion_matrix)

# Print the roots
print("\nRoots from eigenvalues of companion matrix:")
print(np.round(roots, 2))
```

```
print("\nRoots from 'roots' function:")
print(np.round(np.roots(coefficients),2))
```

```
Polynomial:
   10     9     8     7     6     5     4     3     2
1 x  + 2 x + 3 x + 4 x + 5 x + 6 x + 7 x + 8 x + 9 x + 10 x + 11

Companion Matrix:
[[ -2.  -3.  -4.  -5.  -6.  -7.  -8.  -9. -10. -11.]
 [  1.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   1.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   1.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   1.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   1.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   1.   0.]]

Roots from eigenvalues of companion matrix:
[-1.26+0.36j -1.26-0.36j -0.88+0.96j -0.88-0.96j -0.25+1.26j -0.25-1.26j
  0.44+1.17j  0.44-1.17j  0.95+0.73j  0.95-0.73j]

Roots from 'roots' function:
[-1.26+0.36j -1.26-0.36j -0.88+0.96j -0.88-0.96j -0.25+1.26j -0.25-1.26j
  0.44+1.17j  0.44-1.17j  0.95+0.73j  0.95-0.73j]
```

Note: in many software environments, \(j\) is used as the imaginary number instead of \(i\).

previous

Root finding

next

Closed root-finding methods

Contents

Skip to main content

Back to top

Ctrl+K

- .ipynb
- 

.pdf

Open methods

Contents

Open methods#

Another class of root finding algorithms do not require bracketting, and are therefore deemed *open*. This alleviates the issue of N-D dimensionalization at the expense of robustness. They are usually faster than bracketting methods since we are not constantly updating the brackets, but our method is now succeptible to divergence.

Secant method#

Let's reconsider the method of False Position but now we always disregard bracket updating and simply take \(c\) as our new guess. Our algorithm is now:

Take 2 initial guesses: \(x^i\)

Calculate the next guess:

\[x^{i+1} = {x^i} - f(x^{i}) \frac{x^i-x^{i-1}}{f(x^i)-f(x^{i-1})} \]

Check if tolerance is met

(Which tolerance?)

```
# prompt: Solve x^2-2 using the secant method, plotting each step with lin

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  return x**2 - 2

def secant_method(f, x0, x1, tolerance=1e-6, max_iterations=100):
  """
  Finds the root of a function using the secant method.

  Args:
    f: The function to find the root of.
    x0: The initial guess.
    x1: The second initial guess.
    tolerance: The desired tolerance for the root.
    max_iterations: The maximum number of iterations.

  Returns:
    The approximate root of the function.
  """
  print(f"Iteration 0: x = {x1}")
  x_values = [x0, x1]
  for i in range(max_iterations):
    x_new = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
    x_values.append(x_new)
```

```
    print(f"Iteration {i+1}: x = {x_new}")
    if abs(f(x_new)) < tolerance:
      return x_new, x_values
    x0 = x1
    x1 = x_new
  return None, x_values

# Initial guesses
x0 = 1
x1 = 2

# Find the root using the secant method
root, x_values = secant_method(f, x0, x1)


if root:
    print("Approximate root:", root)
    plt.plot(root, 0, 'go', label='Approximate root')
else:
    print("Secant method did not converge within the maximum number of ite

error = abs(np.array(x_values)-root)
print("\nThe sequence of errors is:")
for i,e in enumerate(error):
  print('Iteration, ', i, ' error in x is ', e)


# Plot the function and the secant method iterations
x = np.linspace(.9, 2.1, 100)
plt.plot(x, f(x), label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Secant Method')
plt.grid(True)

# Plot the initial guesses
plt.plot([x0, x1], [f(x0), f(x1)], 'o', label='Initial guesses')

# Plot each iteration of the secant method
for i in range(len(x_values) - 1):
    plt.plot([x_values[i], x_values[i+1]], [f(x_values[i]), f(x_values[i+1
    plt.plot(x_values[i+1], 0, 'ro', label=f'x_{i+2}' if i < 2 else None)


plt.legend()
plt.show()

Iteration 0: x = 2
Iteration 1: x = 1.3333333333333335
Iteration 2: x = 1.4000000000000001
Iteration 3: x = 1.4146341463414633
Iteration 4: x = 1.41421143847487
```

```
Iteration 5: x = 1.4142135620573204
Approximate root: 1.4142135620573204

The sequence of errors is:
Iteration,  0  error in x is  0.4142135620573204
Iteration,  1  error in x is  0.5857864379426796
Iteration,  2  error in x is  0.08088022872398692
Iteration,  3  error in x is  0.014213562057320273
Iteration,  4  error in x is  0.00042058428414293303
Iteration,  5  error in x is  2.12358245033073e-06
Iteration,  6  error in x is  0.0
```

../../ images/ 9d33dddfcd1f5157eacd3ea387398843266382af225ecc81143b1fc4e16b93a3.png

The Secant method maintains superlinear convergence … but we can do better with a little more information…

The Newton-Raphson method#

Take another look at the fraction in the Secant method update equation:

\[x^{i+1} = {x^i} - f(x^{i}) \frac{x^i-x^{i-1}}{f(x^i)-f(x^{i-1})} \]

This is an (inverse) approximation of \(\frac{\partial f}{\partial x}\), the derivative of \(f\)! Typically, we are able to find this quantity, and the algorithm becomes:

\[x^{i+1} = {x^i} - \frac{f(x^{i})}{f'(x^i)} \]

or, solving for the increment \(\Delta x = x^{i+1}-x^i\) and dropping the indicies,

\[\begin{split} \begin{align} \Delta x &= - \frac{f(x)}{f'(x)} \\ f'(x) \Delta x &= - f(x) \end{align} \end{split}\]

```
# prompt: Repeat the root finding using Newton's method and scipy tools

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  return x**2 - 2

def df(x):
  return 2*x

def newton_raphson(f, df, x0, tolerance=1e-6, max_iterations=100):
  """
  Finds the root of a function using the Newton-Raphson method.

  Args:
```

```
    f: The function to find the root of.
    df: The derivative of the function.
    x0: The initial guess.
    tolerance: The desired tolerance for the root.
    max_iterations: The maximum number of iterations.

  Returns:
    The approximate root of the function.
  """

  x_values = [x0]
  print(f"Iteration 0: x = {x0}")
  for i in range(max_iterations):
    x_new = x0 - f(x0) / df(x0)
    x_values.append(x_new)
    print(f"Iteration {i+1}: x = {x_new}")
    if abs(f(x_new)) < tolerance:
      return x_new, x_values
    x0 = x_new
  return None, x_values

# Initial guess
x0 = 1

# Find the root using the Newton-Raphson method
root, x_values = newton_raphson(f, df, x0)

if root:
    print("Approximate root:", root)
    plt.plot(root, 0, 'go', label='Approximate root')
else:
    print("Newton-Raphson method did not converge within the maximum numbe

error = abs(np.array(x_values)-root)
print("\nThe sequence of errors is:")
for i,e in enumerate(error):
  print('Iteration, ', i, ' error in x is ', e)


# Plot the function and the Newton-Raphson method iterations
x = np.linspace(.9, 2.1, 100)
plt.plot(x, f(x), label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Newton-Raphson Method')
plt.grid(True)

# Plot the initial guess
plt.plot([x0], [f(x0)], 'o', label='Initial guess')

# Plot each iteration of the Newton-Raphson method
for i in range(len(x_values) - 1):
```

```
    plt.plot([x_values[i], x_values[i+1]], [f(x_values[i]), f(x_values[i+1
    plt.plot(x_values[i+1], 0, 'ro', label=f'x_{i+2}' if i < 2 else None)


plt.legend()
plt.show()

Iteration 0: x = 1
Iteration 1: x = 1.5
Iteration 2: x = 1.4166666666666667
Iteration 3: x = 1.4142156862745099
Iteration 4: x = 1.4142135623746899
Approximate root: 1.4142135623746899

The sequence of errors is:
Iteration,  0  error in x is  0.41421356237468987
Iteration,  1  error in x is  0.08578643762531013
Iteration,  2  error in x is  0.002453104291976871
Iteration,  3  error in x is  2.123899820016817e-06
Iteration,  4  error in x is  0.0
```

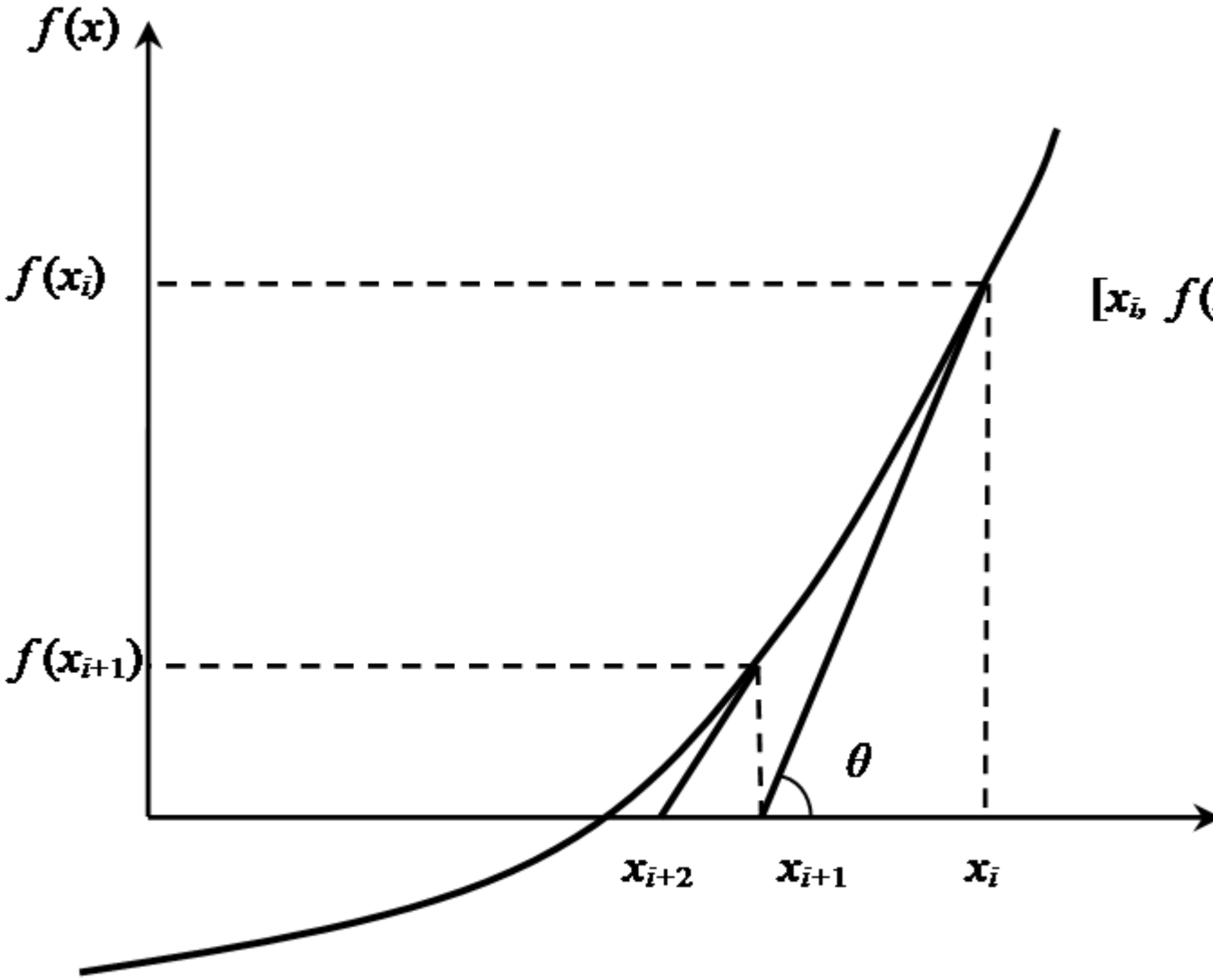
../../ images/
77c7b0a37494ad1056d1a04ac71412400e37185068775eaad4e1063ff314be0f.png

The Newton-Raphson method has quadratic convergence (\(k=2\)) *near the root* which is a great result! It does so, however at the cost of calculating the Jacobian and solving a linear system.

As we saw in the previous lecture, Newton's method amounts to usign the current position and the (true) tangent to estimate the next guess:

\[f'(x) \Delta x = - f(x)\]

Graphically:



Near the root, it converges quadratically, but there are some not-uncommon scenarios where it can fail:

The Newton-Raphson method only finds *local* roots, not all of them. Efficient and robust root finding **requires a good initial guess**.

Fortunatley, in Engineering, this is commonly the case!

Example of an initial guess#

If we need to solve for temperature \(T(x,y,z,t)\) as a nonlinear, time dependent, partial differential equation, we will be given an initial value for \(T(x,y,z, t=0)\).

When solving a nonlinear equation for \(T(x,y,z,t=1)\), what do you suppose the initial guess should be?

Answer: The initial guess for should be the solution at the preceeding time step!

Example: find the root of \(x^3-2x+2\)#

There is a real root at x~-1.769.

```
# prompt: Use a newton-raphson method to find the root of x^3-2x+2 form an

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  """The function whose root we want to find."""
  return x**3 - 2*x + 2

def df(x):
  """The derivative of the function."""
  return 3*x**2 - 2

def newton_raphson(f, df, x0, max_iter=8, tolerance=1e-6):
  """
  Finds a root of the function f using the Newton-Raphson method.

  Args:
    f: The function whose root we want to find.
    df: The derivative of the function.
    x0: The initial guess for the root.
    max_iter: The maximum number of iterations.
    tolerance: The tolerance for the root.

  Returns:
    The root of the function, or None if the method fails to converge.
  """
  x = x0
  guesses = [x]
  for i in range(max_iter):
    x_new = x - f(x) / df(x)
    guesses.append(x_new)

    if abs(x_new - x) < tolerance:
      return x_new, guesses

    x = x_new
  return None, guesses

x0 = 2
root, guesses = newton_raphson(f, df, x0)
```

```
# Plot the function and the guesses
x_vals = np.linspace(-3, 3, 100)
y_vals = f(x_vals)
plt.plot(x_vals, y_vals, label="f(x) = x^3 - 2x + 2")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Newton-Raphson Method")

for i, guess in enumerate(guesses):
  plt.plot(guess, f(guess), 'ro' if i == len(guesses) - 1 else 'bo')
  if i > 0:
    plt.plot([guesses[i-1], guess], [f(guesses[i-1]), f(guess)], 'g--', al


plt.axhline(0, color='black', linestyle='--')  # Add horizontal line at y=
plt.legend()
plt.xlim([-3, 6]) # Set x-axis limits
plt.ylim([-10, 40]) # Set y-axis limits
plt.grid(True)
plt.show()

if root:
  print(f"Root found: {root:.6f}")
else:
  print("Newton-Raphson method failed to converge.")
```

../../ images/
ddebbff5e911c52e38de1574d801bb116d9fcd0625a33381c1b38c29b6647fd7.png

```
Newton-Raphson method failed to converge.
```

Example: Find the root of \(\sqrt(x)\)#

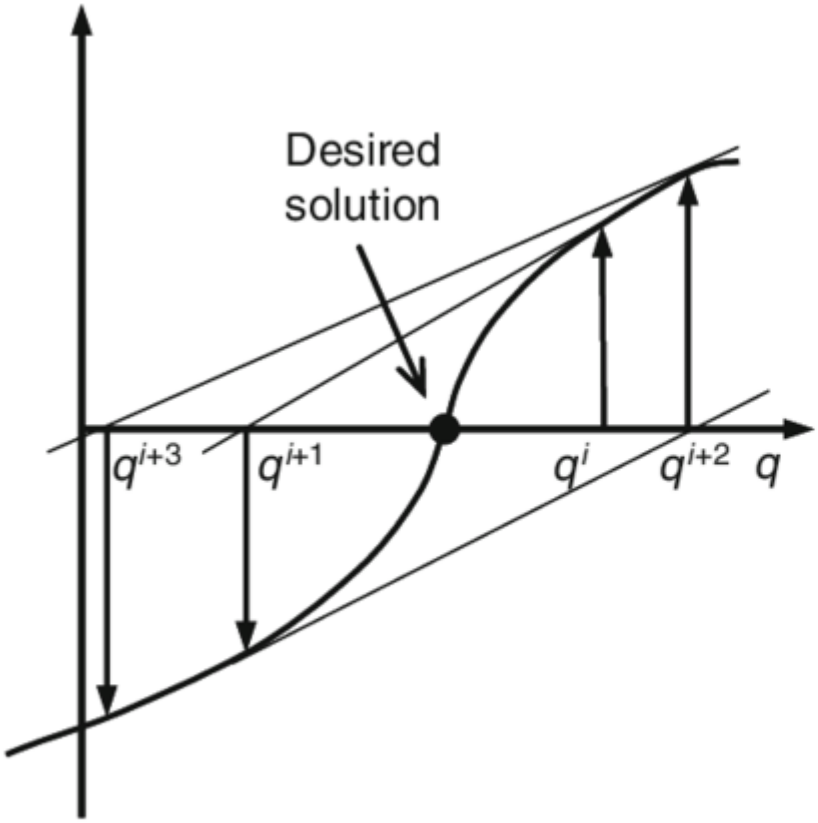
```
import numpy as np
def f(x):
  return np.sqrt(x)

def jacobian(x):
  return 1/3*x**(-2./3)

def newton_raphson(x0, tolerance=1e-6, max_iterations=10):
  """
  Newton-Raphson method for solving a system of nonlinear equations.
  """
  x = x0
  for iter in range(max_iterations):
    f_x = f(x)
    print("Iteration, ", iter, " the guess is ", np.round(x,3), " with res
    J_x = jacobian(x)
    delta_x = -f_x/J_x
    x = x + delta_x
```

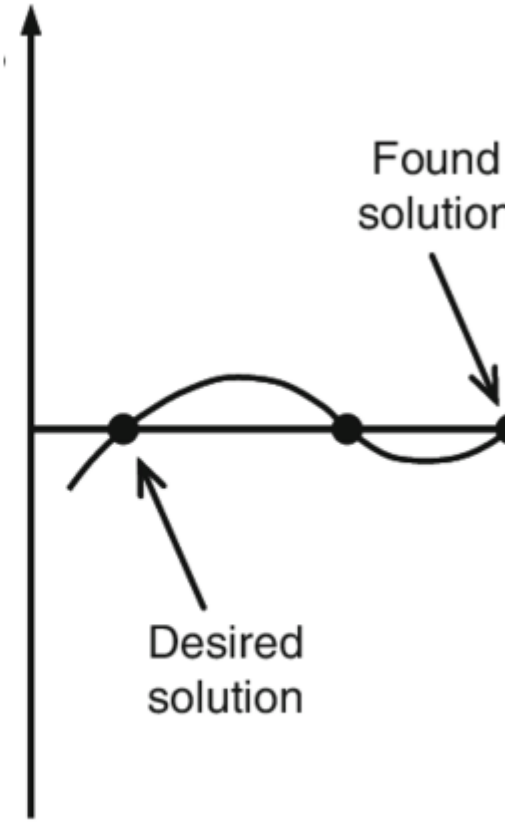
```
    if np.linalg.norm(f_x) < tolerance:
      return x
  return None  # No solution found within the maximum iterations

newton_raphson(x0 = 2)
#newton_raphson(x0 = 2+0j)
```

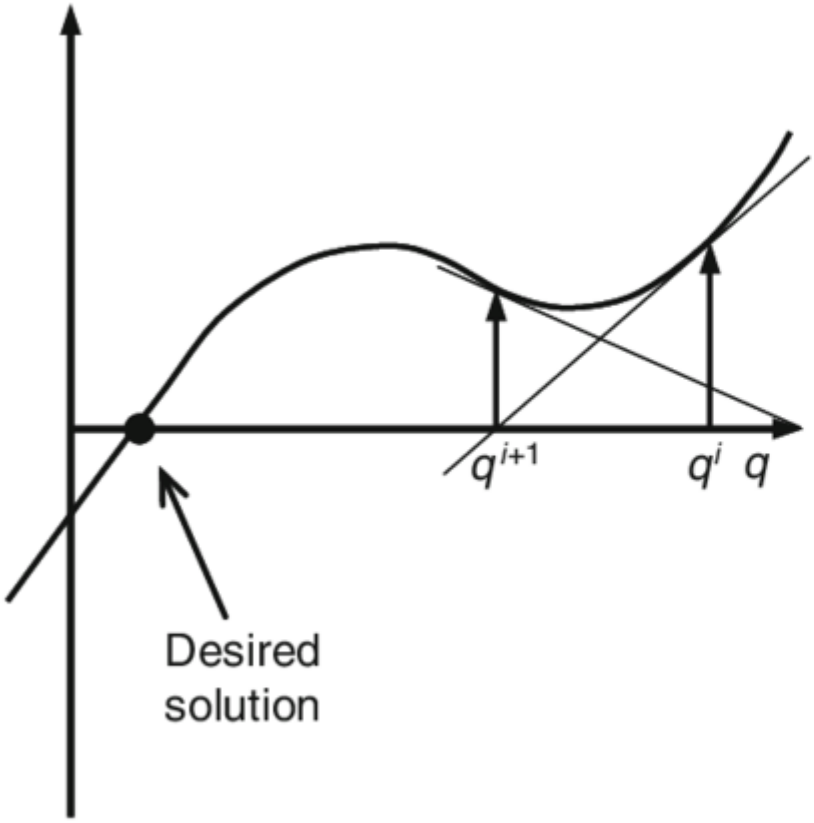
```
Iteration,  0  the guess is  2  with residual  1.4142135623730951
Iteration,  1  the guess is  -4.735  with residual  nan
Iteration,  2  the guess is  nan  with residual  nan
Iteration,  3  the guess is  nan  with residual  nan
Iteration,  4  the guess is  nan  with residual  nan
Iteration,  5  the guess is  nan  with residual  nan
Iteration,  6  the guess is  nan  with residual  nan
Iteration,  7  the guess is  nan  with residual  nan
Iteration,  8  the guess is  nan  with residual  nan
Iteration,  9  the guess is  nan  with residual  nan

<ipython-input-17-26a7d35b8674>:3: RuntimeWarning: invalid value encounter
  return np.sqrt(x)
<ipython-input-17-26a7d35b8674>:6: RuntimeWarning: invalid value encounter
  return 1/3*x**(-2./3)
```

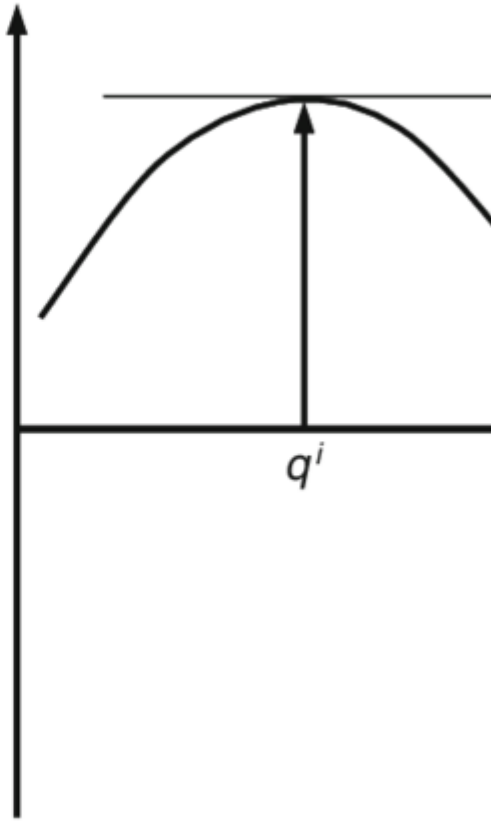
What went wrong here? What else could we try?

Some common failure situations#









Before we talk about mitigation strategies, let’s generalize the Newton-Raphson method to N-D

#The N-D Newton-Raphson method

The Newton-Raphson method thus far has been described for scalar functions or scalar arguments (i.e.: 1-D).

Consider a system of n unkowns $\vec{x}$ and a set of n nonlinear equations that we wish to solve simultaneously:

$$\begin{split}\begin{align} f_1(\vec{x}) &= 0 \\ f_2(\vec{x}) &= 0 \\ \vdots \\ f_n(\vec{x}) &= 0 \end{align} \end{split}$$

which may be summarized as a vector function $\vec{f}(\vec{x})=\vec{0}$ also of dimension n. Since we have n equations and n unkowns, we can (hopefully) find an exact solution of the simultaneous set of equations, i.e.: a root.

The Newton-Raphson method generalized quite readily except the derivative must be replaced by the vector-derivative of a vector function (called the *Jacobian*):

$$\begin{split} J = \frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \end{split}$$

where we can see that J is a square $n \times n$ matrix. The Newton-Raphson method takes the form: $$ J \Delta \vec{x} = - \vec{f} $$ which is …. (wait for it!)… a linear system solving for the vector of increments, $\Delta\vec{x}$!

This is an example of computational-thinking: we have broken down a muiltivariable non-linear vector function into a sequence of linear systems!

Example: solve a system of nonlinear equations:#

$$\begin{split}\begin{align} x^2 + y^2 - z &= 1 \\ x - y^2 + z^2 &= 1 \\ x y z &= 1 \end{align} \end{split}$$

#####Answer

Rewrite the equations as a system of nonlinear functions:

$$\begin{split}\begin{align} f_1(x, y, z) &= x^2 + y^2 - z - 1 \\ f_2(x, y, z) &= x - y^2 + z^2 -1 \\ f_3(x, y, z) &= x*y*z - 1 \end{align} \end{split}$$

or in vector form:

$$\begin{split}\vec{f}(\vec{x}) = \begin{bmatrix} f_1(x, y, z) \\ f_2(x, y, z) \\ f_3(x, y, z) \end{bmatrix} = \begin{bmatrix} x^2 + y^2 - z - 1 \\ x - y^2 + z^2 -1 \\ x y z - 1 \end{bmatrix} =\vec{0} \end{split}$$

The Jacobian is:

$$\begin{split} J = \frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} 2x & 2y & -1 \\ 1 & -2y & 2z \\ yz & xz & xy \end{bmatrix} \end{split}$$

Now for a given $\vec{x}$ we can solve for the increment.

```
# prompt: show newton's method to solve this system, using linalg.solve

import numpy as np

def f(x):
  """
  The system of nonlinear equations.
  """
  x, y, z = x
  return np.array([
      x**2 + y**2 - z - 1,
      x - y**2 + z**2 - 1,
      x * y * z - 1
  ])

def jacobian(x):
  """
  The Jacobian matrix.
  """
  x, y, z = x
  return np.array([
      [2 * x, 2 * y, -1],
      [1, -2 * y, 2 * z],
      [y * z, x * z, x * y]
  ])

def newton_raphson(x0, tolerance=1e-6, max_iterations=100):
  """
  Newton-Raphson method for solving a system of nonlinear equations.
  """
  x = x0
  for iter in range(max_iterations):
    f_x = f(x)
    print("Iteration, ", iter, " the guess is ", np.round(x,3), " with res
    J_x = jacobian(x)

    #~~~~ What now? ####


    #~~ Answer
```

```
    #delta_x = np.linalg.solve(J_x, -f_x)
    #~~~~
    x = x + delta_x
    if np.linalg.norm(f_x) < tolerance:
      return x
  return None  # No solution found within the maximum iterations

# Initial guess
x0 = np.array([2, 2, 2])

# Solve the system
solution = newton_raphson(x0)

if solution is not None:
  print("Solution found:", solution)
else:
  print("No solution found within the maximum iterations.")
```

```
Iteration,  0  the guess is  [2 2 2]  with residual  8.660254037844387
Iteration,  1  the guess is  [1.04 1.61 1.6 ]  with residual  1.9930051233
Iteration,  2  the guess is  [1.07  1.091 1.066]  with residual  0.3650315
Iteration,  3  the guess is  [1.001 1.008 1.007]  with residual  0.0198219
Iteration,  4  the guess is  [1. 1. 1.]  with residual  9.992137180440437e
Iteration,  5  the guess is  [1. 1. 1.]  with residual  2.103116571058853e
Solution found: [1. 1. 1.]
```

Let's try a different initial guess:

```
solution = newton_raphson(np.array([3, 3 , 3]))
```

```
Iteration,  0  the guess is  [3 3 3]  with residual  29.597297173897484
Iteration,  1  the guess is  [1.054 2.533 2.524]  with residual  6.9985754
Iteration,  2  the guess is  [0.889 1.641 1.66 ]  with residual  1.6424438
Iteration,  3  the guess is  [ 2.047  0.239 -0.06 ]  with residual  3.6032
Iteration,  4  the guess is  [1.707 1.028 2.231]  with residual  5.5179698
Iteration,  5  the guess is  [1.238 0.984 1.279]  with residual  1.0856646
Iteration,  6  the guess is  [1.031 0.998 1.016]  with residual  0.0930478
Iteration,  7  the guess is  [1. 1. 1.]  with residual  0.0010714827370243
Iteration,  8  the guess is  [1. 1. 1.]  with residual  1.8136147365183113
```

Great, but something is funny with the residual... Let's keep going!

```
solution = newton_raphson(np.array([10, 10 , 10]))
```

```
Iteration,  0  the guess is  [10 10 10]  with residual  1016.7610338717747
Iteration,  1  the guess is  [1.037 9.488 9.486]  with residual  122.54179
Iteration,  2  the guess is  [0.99  5.009 5.009]  with residual  31.151332
Iteration,  3  the guess is  [0.91  2.803 2.812]  with residual  7.8633181
Iteration,  4  the guess is  [0.756 1.814 1.861]  with residual  1.8475446
Iteration,  5  the guess is  [0.273 1.768 1.965]  with residual  0.2402070
Iteration,  6  the guess is  [0.319 1.664 1.858]  with residual  0.0192416
Iteration,  7  the guess is  [0.327 1.656 1.848]  with residual  0.0002604
Iteration,  8  the guess is  [0.327 1.656 1.848]  with residual  4.9045009
```

Still converged but to a different root...

What about negatives?

```
solution = newton_raphson(np.array([-1, -1 ,-1]))
```

```
Iteration,  0  the guess is  [-1 -1 -1]  with residual  3.4641016151377544
Iteration,  1  the guess is  [-7.  5.  1.]  with residual  86.625631310830
Iteration,  2  the guess is  [-3.924  2.106  0.99 ]  with residual  21.741
Iteration,  3  the guess is  [-2.54   0.452  1.005]  with residual  5.8079
Iteration,  4  the guess is  [-1.921 -0.552  1.606]  with residual  1.6863
Iteration,  5  the guess is  [-1.619 -0.393  1.659]  with residual  0.1308
Iteration,  6  the guess is  [-1.583 -0.383  1.652]  with residual  0.0015
Iteration,  7  the guess is  [-1.583 -0.382  1.652]  with residual  2.8665
```

Another root?

```
solution = newton_raphson(np.array([-10, 0,-10]))
```

```
Iteration,  0  the guess is  [-10   0 -10]  with residual  140.72313242676
Iteration,  1  the guess is  [-4.786  0.01  -5.289]  with residual  35.104
Iteration,  2  the guess is  [-2.189  0.049 -2.946]  with residual  8.7200
Iteration,  3  the guess is  [-0.908  0.203 -1.8  ]  with residual  2.2103
Iteration,  4  the guess is  [-0.125  0.844 -1.296]  with residual  1.3486
Iteration,  5  the guess is  [-1.024  0.136 -1.243]  with residual  1.6270
Iteration,  6  the guess is  [-0.291  0.885 -1.23 ]  with residual  1.4089
Iteration,  7  the guess is  [-1.226 -0.019 -1.188]  with residual  2.1415
Iteration,  8  the guess is  [-0.531  0.676 -1.227]  with residual  1.2164
Iteration,  9  the guess is  [-3.688 -2.373 -1.03 ]  with residual  23.600
Iteration,  10  the guess is  [-2.12  -0.752 -1.027]  with residual  6.304
Iteration,  11  the guess is  [-1.242  0.241 -1.154]  with residual  2.109
Iteration,  12  the guess is  [-0.356  0.836 -1.314]  with residual  1.333
Iteration,  13  the guess is  [-1.391 -0.202 -1.172]  with residual  2.737
Iteration,  14  the guess is  [-0.711  0.521 -1.209]  with residual  1.243
Iteration,  15  the guess is  [ 1.97   3.045 -1.403]  with residual  17.68
Iteration,  16  the guess is  [ 0.43   1.799 -1.505]  with residual  4.740
Iteration,  17  the guess is  [-0.056  0.927 -1.135]  with residual  1.506
Iteration,  18  the guess is  [-0.984  0.258 -1.274]  with residual  1.534
Iteration,  19  the guess is  [-0.125  1.026 -1.26 ]  with residual  1.677
Iteration,  20  the guess is  [-0.865  0.314 -1.208]  with residual  1.348
Iteration,  21  the guess is  [ 0.135  1.306 -1.261]  with residual  2.528
Iteration,  22  the guess is  [-0.533  0.632 -1.217]  with residual  1.167
Iteration,  23  the guess is  [-5.092 -3.751 -0.999]  with residual  48.68
Iteration,  24  the guess is  [-2.826 -1.497 -0.992]  with residual  12.53
Iteration,  25  the guess is  [-1.703 -0.194 -1.021]  with residual  3.663
Iteration,  26  the guess is  [-0.837  0.528 -1.291]  with residual  1.413
Iteration,  27  the guess is  [ 0.892  1.976 -1.387]  with residual  6.488
Iteration,  28  the guess is  [ 0.262  1.056 -1.057]  with residual  1.938
Iteration,  29  the guess is  [-0.871  0.564 -1.45 ]  with residual  1.556
Iteration,  30  the guess is  [ 0.873  1.937 -1.413]  with residual  6.268
Iteration,  31  the guess is  [ 0.264  1.032 -1.055]  with residual  1.883
Iteration,  32  the guess is  [-0.899  0.554 -1.465]  with residual  1.604
Iteration,  33  the guess is  [ 0.689  1.762 -1.401]  with residual  5.025
```

```
Iteration,  34  the guess is  [ 0.033  0.961 -1.146]  with residual  1.598
Iteration,  35  the guess is  [-0.891  0.358 -1.295]  with residual  1.393
Iteration,  36  the guess is  [ 0.159  1.293 -1.28 ]  with residual  2.503
Iteration,  37  the guess is  [-0.516  0.635 -1.22 ]  with residual  1.155
Iteration,  38  the guess is  [-4.362 -3.057 -1.052]  with residual  34.90
Iteration,  39  the guess is  [-2.445 -1.144 -1.045]  with residual  9.087
Iteration,  40  the guess is  [-1.464 -0.016 -1.093]  with residual  2.768
Iteration,  41  the guess is  [-0.64   0.619 -1.288]  with residual  1.241
Iteration,  42  the guess is  [32.342 31.94  -3.679]  with residual  4436.
Iteration,  43  the guess is  [15.693 16.412 -3.681]  with residual  1107.
Iteration,  44  the guess is  [ 7.126  8.812 -3.71 ]  with residual  274.3
Iteration,  45  the guess is  [ 2.142  5.383 -4.023]  with residual  61.00
Iteration,  46  the guess is  [ 3.103  1.911 -0.705]  with residual  14.01
Iteration,  47  the guess is  [ 1.524  1.138 -0.475]  with residual  3.629
Iteration,  48  the guess is  [1.224 0.481 0.209]  with residual  1.020612
Iteration,  49  the guess is  [1.268 1.093 1.425]  with residual  1.518906
Iteration,  50  the guess is  [1.035 1.027 1.069]  with residual  0.192433
Iteration,  51  the guess is  [1.001 1.001 1.003]  with residual  0.006473
Iteration,  52  the guess is  [1. 1. 1.]  with residual  9.601721087105206
Iteration,  53  the guess is  [1. 1. 1.]  with residual  2.129378108229476
```

Yikes! Note what's happening to guesses and the residual... odd behaviour indeed!

##Basins of attraction

The previous examples show that depending on the initial guess, we may arrive at different roots! These are the so-called *basins of attraction* for each root. Newton's method *jumps* around the parameter space, and the increase in residual corresponds to a jump across a basin. This can lead to tempermental (but beautiful) behaviour.

Tempermental but beautiful behaviour#

The Newton-Raphson method naturally can handle complex numbers of the form \(x + y i\).

Example: \(x^3-x\)#

```
# prompt: Show the basins of attraction for x**3-x in the complex plane

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton

def f(z):
  return z**3 - z

def df(z):
  return 3 * z**2 - 1

# Create a grid of complex numbers
```

```
real_range = (-2, 2)
imag_range = (-2, 2)
grid_size = 500
real_values = np.linspace(real_range[0], real_range[1], grid_size)
imag_values = np.linspace(imag_range[0], imag_range[1], grid_size)
z_grid = np.array([[complex(r, i) for r in real_values] for i in imag_valu

# Apply Newton-Raphson to each point in the grid
#roots = np.array([[newton(f, z, fprime=df) for z in row] for row in z_gri
roots = newton(f, z_grid, fprime=df)

# Assign colors based on the root found
colors = np.zeros((grid_size, grid_size))
for i in range(grid_size):
  for j in range(grid_size):
    if roots[i, j] is None:
      colors[i, j] = 0
    elif abs(roots[i, j] - 1) < 0.5:
      colors[i, j] = 1
    elif abs(roots[i, j] - (-1)) < 0.5:
      colors[i, j] = 2
    elif abs(roots[i, j] - 0) < 0.5:
      colors[i, j] = 3
    else:
      colors[i, j] = 0  # Assign a default color


# Plot the basins of attraction
plt.imshow(colors, extent=[real_range[0], real_range[1], imag_range[0], im
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.title('Basins of Attraction for z^3 - z')
plt.colorbar()
plt.show()
```

../../ images/
0df8de9c026724bf8718df34adf6e58aca574d20c3b242a517812774378a5d29.png

Example: \(x^3-1\)#

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton

def f(z):
  return z**3 - 1

def df(z):
  return 3 * z**2

# Create a grid of complex numbers
real_range = (-1.5, 1.5)
```

```
imag_range = (-1.5, 1.5)
grid_size = 1000
real_values = np.linspace(real_range[0], real_range[1], grid_size)
imag_values = np.linspace(imag_range[0], imag_range[1], grid_size)
z_grid = np.array([[complex(r, i) for r in real_values] for i in imag_valu

# Apply Newton-Raphson to each point in the grid
#roots = np.array([[newton(f, z, fprime=df) for z in row] for row in z_gri
roots = newton(f, z_grid, fprime=df)

# Assign colors based on the root found
th = 1e-3
rs = np.roots([1,0,0,-1])
colors = np.zeros((grid_size, grid_size))
for i in range(grid_size):
  for j in range(grid_size):
    if roots[i, j] is None:
      colors[i, j] = 0
    elif abs(roots[i, j] - rs[0]) < th:
      colors[i, j] = 1
    elif abs(roots[i, j] - rs[1]) < th:
      colors[i, j] = 2
    elif abs(roots[i, j] - rs[2]) < th:
      colors[i, j] = 3
    else:
      colors[i, j] = 0  # Assign a default color


# Plot the basins of attraction
plt.imshow(colors, extent=[real_range[0], real_range[1], imag_range[0], im
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.title('Basins of Attraction for z^3 - 1')
plt.colorbar()
plt.show()
```

../../ images/
b32e734645705928c09ca624901383b9a56e4e0e1a08e6a826b070c65f3c7b56.png

Beautiful, but like most beautiful things… often problematic… It implies small changes in initial guesses can find dramatically different roots, and numerical methods are prone to ‘small changes’ due to roundoff error…

[previous](#)

Closed root-finding methods

[next](#)

‘Global’ convergence

Contents

By The Jupyter Book community

Root finding

Contents

Goals:

- Understand the nature of the root finding problem.
- Use standard tools for rootfinding of polynomials
- Understand bracketting root finding methods
- Understand open root finding methods

Root finding#

The roots (*aka zeros*) of a function are values of function arguments forwhich the function is zero:

Find \(x\) such that:

\[f(x) = 0 \]

It can become complicated when we consider vector \(\vec{x}\) and even \(\vec{f}\), which may seem complicated at first, but consider a special case of finding the roots of \(\vec{f}(\vec{x})\) is our familiar linear system, \(A \vec{x} -\vec{b} = \vec{0}\). This topic is mearly the generalization to nonlinear functions.

Roots of some nonlinear functions#

Let's build some intuition by exploring some type of roots in 1D functions using the *graphical method*: Plot the function and examine where it crosses the x-axis.

NB: Note the structure of the code below - Since we don't know *a priori* where the roots will be, we have to take a series of initial guesses and cross our finger.... and even then we may fail to find them all!

```
# prompt: Define a function that takes a function, plots it with xrange -1
# NB: Modified from original output
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root

def plot_and_find_roots(func):
  """Plots a function and finds its roots using fsolve.

  Args:
    func: The function to plot and find roots for.
  """

  x = np.linspace(-10, 10, 400)
  y = func(x)

  plt.figure(figsize=(8, 6))
  plt.plot(x, y, label='f(x)')
  plt.axhline(y=0, color='black', linestyle='--')  # Plot the x-axis
  plt.xlabel('x')
  plt.ylabel('f(x)')
  plt.title('Plot of f(x) and its Roots')
  plt.xlim([-10, 10])
  plt.ylim([-10, 10])

  x0s = np.arange(-10,10,1)
  for x0 in x0s:
    r = root(func, x0=x0)
    if r.success:
      plt.plot(r.x, r.fun, 'ro', markersize=8)  # Plot root with a red dot
    plt.legend(['f(x)', 'Roots'])

  plt.grid(True)
  plt.show()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last
Cell In[1], line 4
      1 # prompt: Define a function that takes a function, plots it with x
      2 # NB: Modified from original output
----> 4 import numpy as np
      5 import matplotlib.pyplot as plt
      6 from scipy.optimize import root

ModuleNotFoundError: No module named 'numpy'
```

Example 1: Real roots - \(x^2-4\)#

```
plot_and_find_roots(lambda x: x**2-4)
```

../../ images/
949768f6d65f61c90ccff47eec729c0a8ca88c6ee10ec4c71dcd56673cd3b955.png

Example 2: No roots - \(1/x\)#

```
plot_and_find_roots(lambda x: 1/x)
```

```
<ipython-input-4-f7fd98d7a758>:1: RuntimeWarning: divide by zero encounter
  plot_and_find_roots(lambda x: 1/x)
```


../../ images/
e02144920c19c296fdb899d831e335283477eeac181331d23353735c5e7b082a.png

Noting that the vertical line is a plotting artifact.

Example 3: Infinite roots \(sin(x^2)\)#

```
plot_and_find_roots(lambda x: np.sin(x**2))
```


../../ images/
b7b522b60b7acb25a7d9ec9a75dcc2eeffc5f8d0076a3de5dd046f4aa5fbe95c.png

Only the roots closest to the initial guesses are found!

Complex roots - \(x^2+1\)#

Even the graphical method is not completely reliable due to the existence of *complex roots*

```
def fun(x):
  return x**2 + 1

# Wrong!
plot_and_find_roots(fun)
```


../../ images/
26f406e2a26dc116b6da9729e12fb60280f2c36c0328052f22fdeca36a45cf4d.png

But this is wrong! The quadratic has 2 roots but we need to use a different method:

```
root(lambda x: x**2+1, x0 = [1+1j, 1-1j], method = "krylov")

# prompt: Do a complex plot of x**2+1 and add points at the roots

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root

def complex_plot(func):
  """Plots a complex function and finds its roots using root.

  Args:
    func: The function to plot and find roots for.
  """
```

```
  real_range = np.linspace(-3, 3, 100)
  imag_range = np.linspace(-3, 3, 100)

  real_part = np.empty((len(real_range), len(imag_range)))
  imag_part = np.empty((len(real_range), len(imag_range)))

  for i, real in enumerate(real_range):
    for j, imag in enumerate(imag_range):
      z = complex(real, imag)
      result = func(z)
      real_part[i, j] = result.real
      imag_part[i, j] = result.imag

  plt.figure(figsize=(8, 6))
  plt.contourf(real_range, imag_range, real_part, cmap='viridis')
  plt.colorbar(label='Real Part')
  plt.xlabel('Real')
  plt.ylabel('Imaginary')
  plt.title('Complex Plot of f(z)')
  plt.grid(True)

  # Find roots and plot them
  r = root(lambda x: x**2 + 1, x0=[1 + 1j, 1 - 1j], method="krylov")
  if r.success:
    for root_val in r.x:
      plt.plot(root_val.real, root_val.imag, 'ro', markersize=8)  # Plot r

  plt.show()


# Use the function to plot x**2 + 1
complex_plot(lambda z: z**2 + 1)
```

../../ images/
5e6861a0248c5daa91e3ccac2eb0148f97e212551016cd1a2658c515da4bf3e0.png

DON’T WORRY - We won’t be dealing with complex numbers in general in this course :-)

previous

Example: An overdetermined, inconsistent linear system

next

Polynomial roots

Contents

By The Jupyter Book community

‘Global’ convergence

Contents

Goals:

- See the pitfalls of Newton’s method
- Understand the ND Newton-Raphson method
- Awareness of linesearch algorithms for improved convergence.

‘Global’ convergence#

There are several options to modify the Newton-Raphson method in order to enhance the robustness of root finding, but the improvement in robustness has to be weighed against the computational expense.

We **have** to assume our initial guess is reasonable, so the goal is to ensure the solution doesn’t *wander*.

Line search#

If we trust that \(\Delta \vec{x}\) is pointing in the right direction, then we should make sure it doesn’t *step too far*.

These approaches are called *line search* alrogithms since we are moving along the direction prescribed by \(\Delta \vec{x}\).

Damped Newton-Raphson#

The easiest modification is adding a damping term. Calculate \(\Delta \vec{x}\) as usual from \(J \Delta \vec{x} = - \vec{f}\) but update the step a scalar factor of the increment: $\(\vec{x}^{i+1} = \vec{x}^i+\alpha \Delta \vec{x} \)$

For \(\alpha =1\) we recover the method.

For $0 < \alpha < 1$ the method is underdamped, and the solver forced to take small steps, keeping it closer to the guess but at the cost of convergence speed.

For $\alpha > 1$ the method is overdamped, and solver steps exagerated. This may seem like a good idea to speed things up but if you are not careful you will constantly overshoot your root or produce frenetic behaviour!

Optimal step size#

From the *damped* approach, we can also attempt to find a 'good' step size algorithmically. Near the root, we know we want $\alpha=1$ to get quadratic convergence, so this is often a starting point.

There are several approaches which levarage the information we have:

- $\vec{f}(\vec{x})$
- $J(\vec{x})$
- $\vec{f}(\vec{x}+\Delta \vec{x})$

Fit a quadratic#

With 3 pieces of information, we can fit a quadratic in the search direction and choose the minimum for the update.

Backtracking#

The backtracking routine starts with $\alpha=1$ and then subdivides it until

$$\vec{f}(\vec{x}+\alpha \Delta \vec{x}) - \vec{f}(\vec{x}) \approx J(\vec{x}) \alpha \Delta \vec{x}$$

Bounded minimization#

Find α in the range (0,1] that minimizes some measure of residual, e.g. $\| \vec{f}(\vec{x}+\alpha \Delta \vec{x}) \|$.

This is a 1D minimizaqtion problem (similar to our 1D root finding) and can use similar tools e.g.: Secant method. The tradeoff here is the efficiency of this minimzation vs just taking another Newton step.

previous

Open methods

Contents

By The Jupyter Book community

Closed root-finding methods

Contents

Closed root-finding methods#

The roots of nonlinear functions are more complicated than linear functions. We can seldom solve for the roots directly (notable exception being polynomial functions), and instead will need to *search* for them iteratively.

Iterative search methods are characterized by their *order of convergence*, which measure how successive guesses approach the true root. Given the true root x, we can check how successive guesses approach it by calculating the error:

$$ x^{i+1}-x \propto [x^{i}-x]^k $$

where k is the order.

Bracketting methods#

Bracketing methods exploits the fact that functions change sign across the roots of a 1-D function (a simple applicaiton of the intermediate value theorem). This does not work for certain cases (eg: $1/x$).

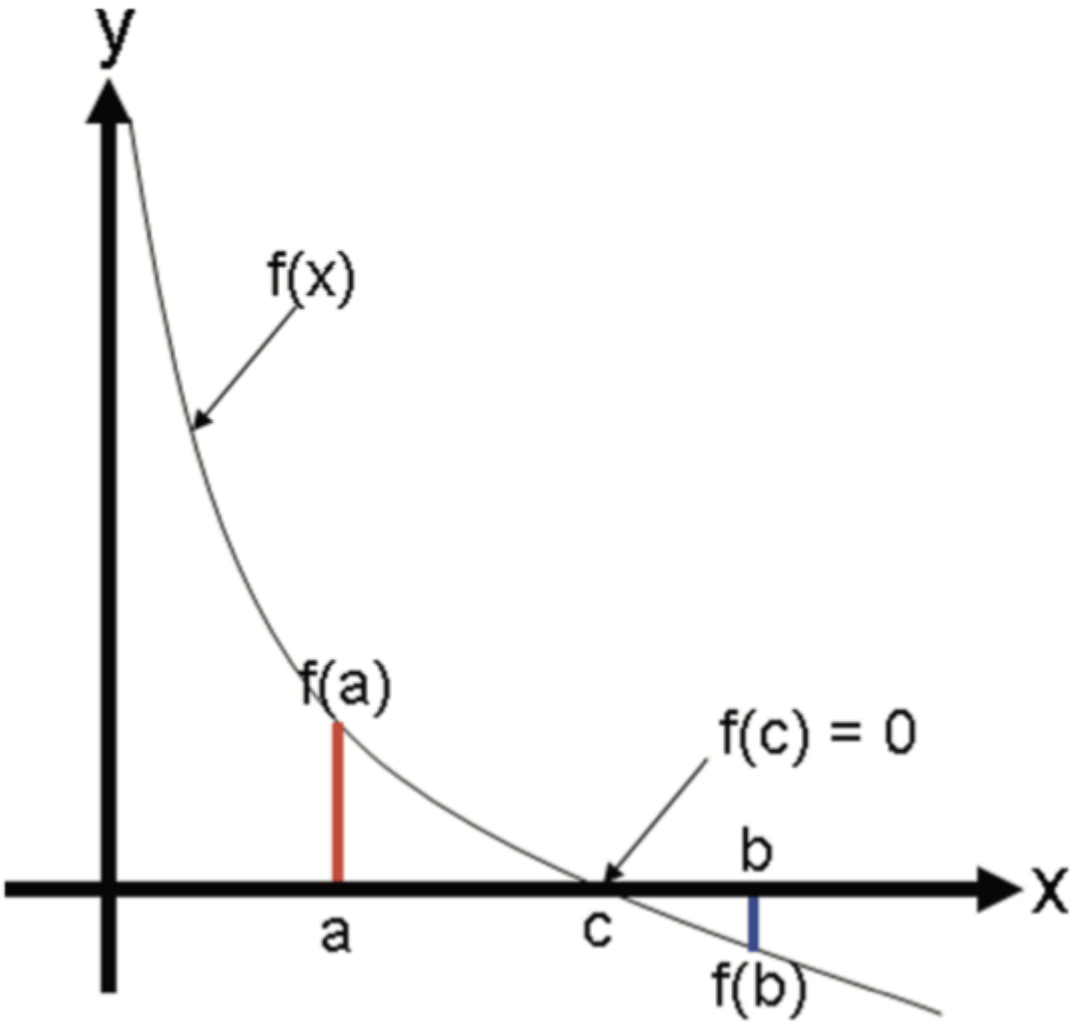
Bisection methods#

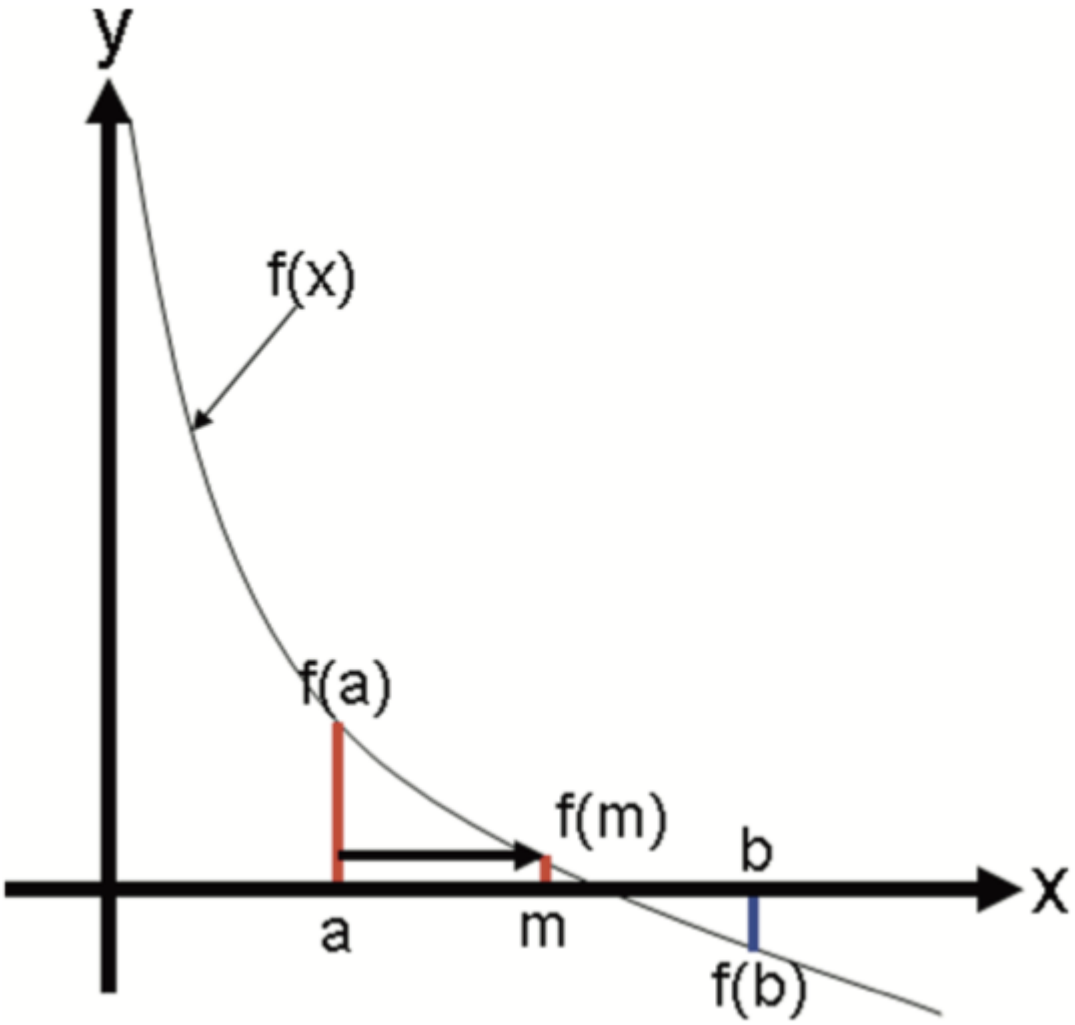
Bisection methods are essentially a binary search for the root. If $f(x)$ is continuous between bounds a and $b>a$ and $f(a)$ and $f(b)$ are opposite signs, there must be a point c where $f(c)=0$.

The algorithm is:

> Given brackets a and b
>
> Calculate the midpoint $c = (b-a)/2$.
>
> If $f(c) \approx 0$ or $a \approx b$: exit.
>
> If $f(c)>0$: set $a = c$
>
> If $f(c)<0$: set $b = c$
>
> repeat

Graphically this is:


y
f(x)
f(a)
f(c) = 0
b
x
a
c
f(b)




```
# prompt: Find the root of x^2-2 using bisect

from scipy.optimize import bisect

def f(x):
  return x**2 - 2

print('Setting x tolerance: \n')
xtols = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5]
for xtol in xtols:
  print(bisect(f, 1, 2, maxiter = 100, xtol = xtol))


print('\nSetting relative tolerance \n', bisect(f, 1, 2, maxiter = 100, rt

Setting x tolerance:

1.4375
1.4140625
1.4150390625
1.41424560546875
1.4142074584960938
```

```
Setting relative tolerance
 1.4142135623715149
```

The error of the Bisection Method generaly follows:

$$ x^{i+1}-x = \frac{1}{2} [x^{i}-x] $$

and therefore has a linear order of convergence ($k=1$).

Method of False Position (Regula falsi)#

Let's use more information! In bisection we are only interested in $f(a,b)$ switching signs, but it stands to reason the larger $f(b)$ is compared to $f(a)$, the further the root is from b!

The method of false position uses this information to determine the next candidate solution:

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

The same algorithm as for bisection is then applied to replace either a or b with c so that the root remains bracketted.

```
# prompt: Give me a plot that illustrates the method of false position usi

import numpy as np
import matplotlib.pyplot as plt


def f(x):
  return x**2 - 2

# Example values for a and b
a = 1
b = 2

# Calculate f(a) and f(b)
fa = f(a)
fb = f(b)

# Calculate the next candidate solution c using the method of false positi
c = b - fb * (b - a) / (fb - fa)
fc = f(c)

# Generate x values for the plot
x = np.linspace(a - 0.5, b + 0.5, 100)

# Plot the function
plt.plot(x, f(x), label='f(x)')

# Plot the initial bracket
plt.plot([a, b, a, b], [fa, fb, 0, 0], 'o', label='Initial Bracket')
```

```
# Plot the new candidate solution c
plt.plot([c, c], [fc, 0], 'ro', label='New Candidate (c)')


# Plot the line connecting (a, f(a)) and (b, f(b))
plt.plot([a, b], [fa, fb], '--', color='gray', label='Linear interpolation

# Add labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Method of False Position (Regula falsi) - One Iteration')
plt.legend()
plt.grid(True)
plt.show()
```

../../ images/
705b81001dbdebb1a53427429c5094e2234dabbd7c942b9c990911004ef4d202.png

Note that the line joining the brackets approximates the tangent of the curve.

The method of False Position has an order of convergence of 1.618 (the Golden Ratio!):

\[x^{i+1}-x \propto [x^i-x]^{1.618} \]

This is considered *superlinear* and a good thing!

Summary of bracketting methods#

Bracketting methods are usually robust but slow to converge and generlization to N-D is not trivial.

Through incorporating the additional information of linear interpolation, the method of Flase Position achieves superlinear convergence.

But bracketting is complicated in N-D, so let's try to remove it.

previous

Polynomial roots

next

Open methods

Contents

By The Jupyter Book community