PHYS 351
Topic 2: Root Finding

Dr. Daugherity
Abilene Christian University

Problem

Solve any gross equation for x:

$$\cosh(x^2 e^x) = \sin x + x!$$

Problem 2

Our approach: ROOT FINDING!

Find $\boldsymbol{x}$ where $\boldsymbol{f(x) = 0}$

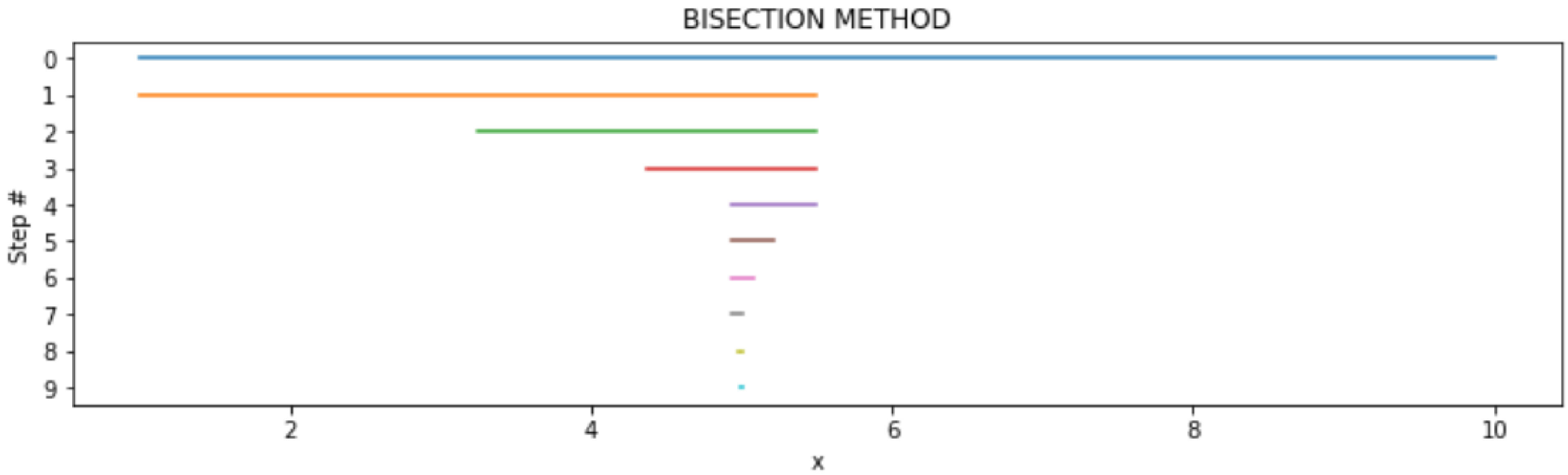
Example:

To solve: $\cosh(x^2 e^x) = \sin x + x!$

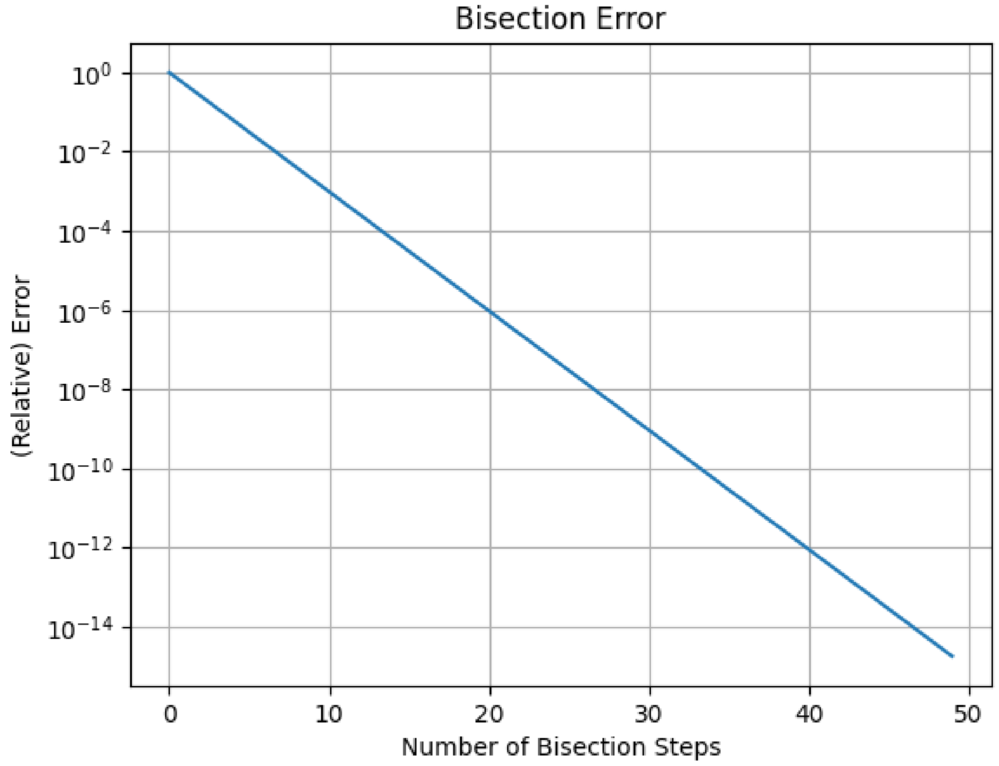
Find roots of

$$f(x) = \cosh(x^2 e^x) - \sin x + x!$$

BISECTION



BISECTION METHOD
Step #
x
0
1
2
3
4
5
6
7
8
9
2
4
6
8
10



Bisection Error
(Relative) Error
Number of Bisection Steps
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
0
10
20
30
40
50

NEWTON-RAPHSON

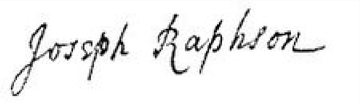
https://en.wikipedia.org/wiki/Joseph_Raphson

Joseph Raphson

From Wikipedia, the free encyclopedia

Joseph Raphson (c. 1648 – c. 1715) was an English mathematician known best for the Newton–Raphson method.

Joseph Raphson	
Born	c. 1648 Middlesex, England
Died	c. 1715 England
Nationality	English
Alma mater	University of Cambridge
Known for	Newton–Raphson method
Scientific career	
Fields	Mathematician
Signature	

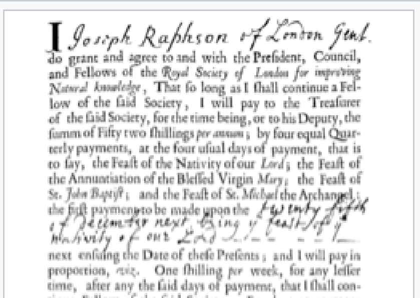


Contents [hide]

Biography [edit]

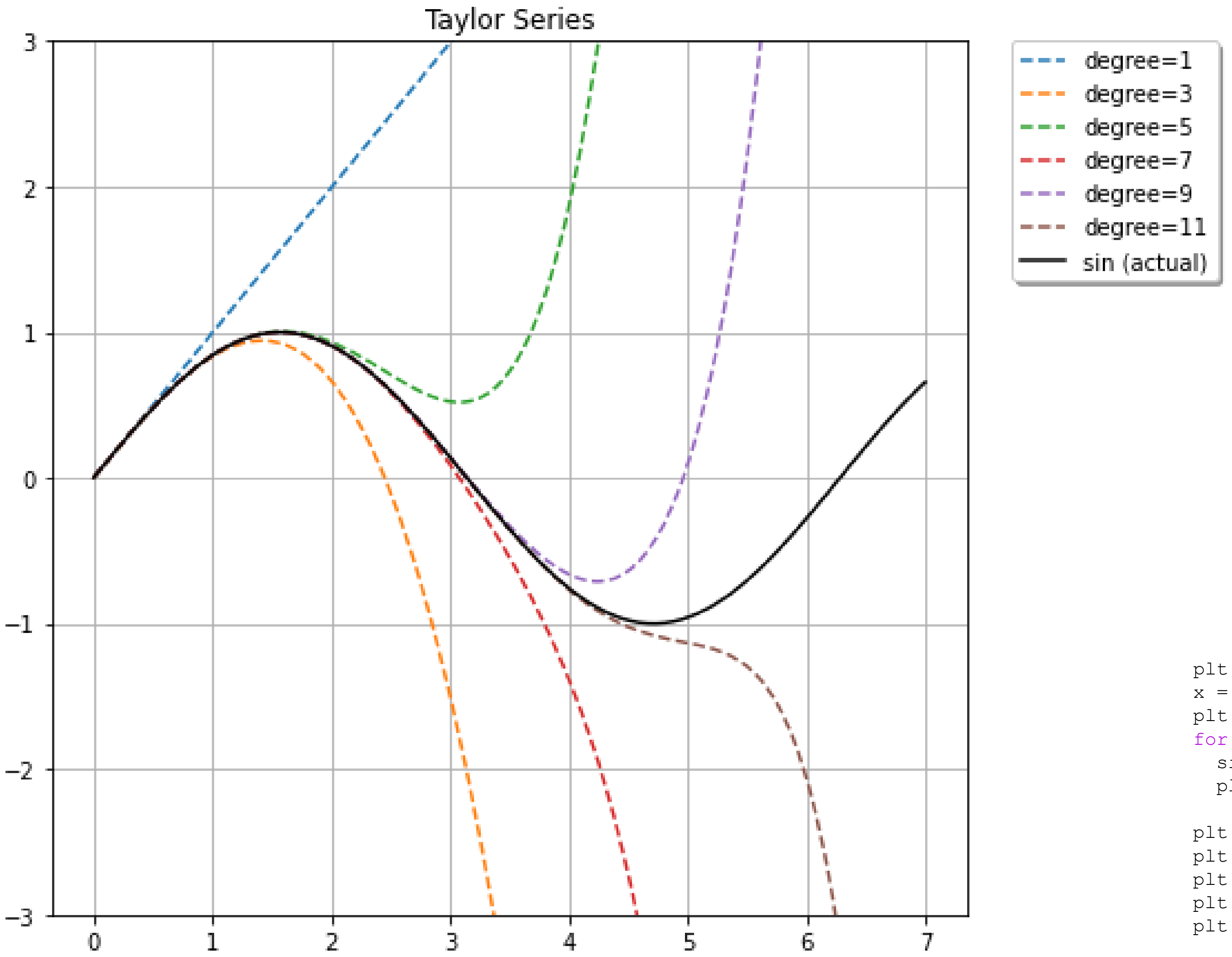
Little is known about Raphson's life, and even his exact years of birth and death are unknown, although the mathematical historian Florian Cajori provided the approximate dates 1648–1715. He was likely of Jewish and Irish descent.[1] Raphson attended Jesus College at Cambridge, graduating with an M.A. in 1692.[2] He was made a Fellow of the Royal Society on 30 November 1689, after being proposed for membership by Edmund Halley.

Raphson's most notable work is *Analysis Aequationum Universalis*, which was published in 1690. It contains a method, now known as the Newton–Raphson method, for approximating the roots of an equation. Isaac Newton had developed a very similar formula in his *Method of Fluxions*, written in 1671, but this work would not be published until 1736, nearly 50 years after Raphson's *Analysis*. However, Raphson's version of the method is simpler than Newton's, and is therefore generally considered superior. For this reason, it is Raphson's version of the method, rather than Newton's, that is to be found in textbooks today.


I Joseph Raphson of London Gent. do grant and agree to and with the President, Council, and Fellows of the *Royal Society of London for improving Natural knowledge*, That so long as I shall continue a Fellow of the said Society, I will pay to the Treasurer of the said Society, for the time being, or to his Deputy, the summ of Fifty two shillings *per annum*; by four equal Quarterly payments, at the four usual days of payment, that is to say, the Feast of the Nativity of our *Lord*; the Feast of the Annuntiation of the Blessed Virgin *Mary*; the Feast of St. *John Baptist*; and the Feast of St. Michael the Archangel; the first payment to be made upon the [illegible] of December next being ye feast of ye Nativity of our Lord next ensuing the Date of these Presents; and I will pay in proportion, *viz.* One shilling *per* week, for any lesser time, after any the said days of payment, that I shall con-

Taylor Series

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + h_k(x)(x-a)^k,$$



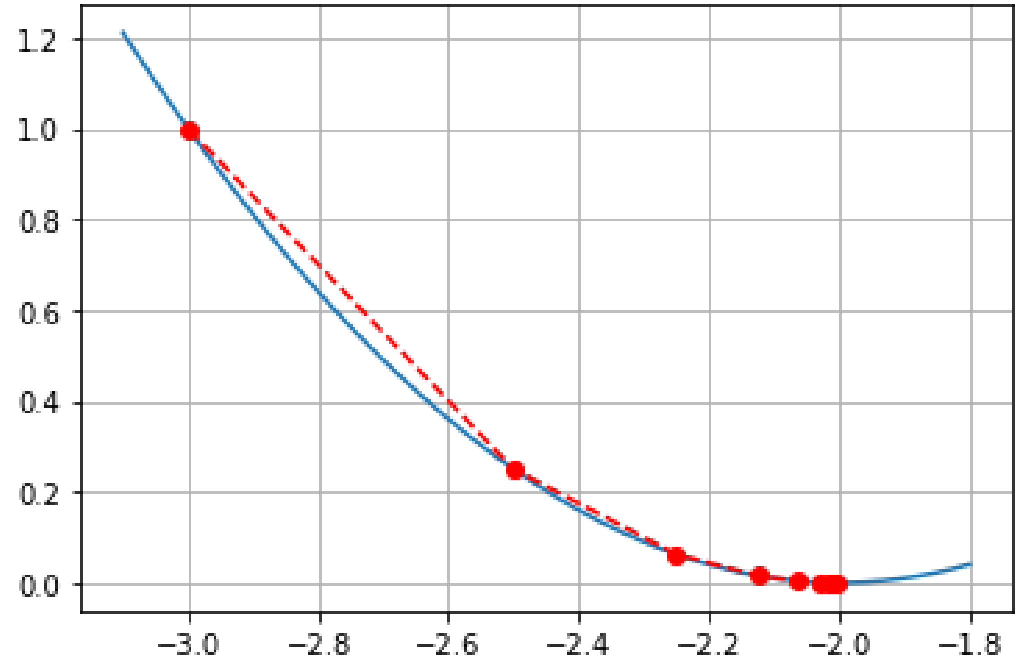

```
plt.figure(figsize=(8,6))
x = np.linspace(0, 7, num=100)
plt.plot(x, np.sin(x), label="sin curve")
for degree in np.arange(1, 13, step=2):
  sin_taylor = approximate_taylor_polynomial(np.sin, 0, degree, 1, order=degree + 2)
  plt.plot(x, sin_taylor(x), label=f"degree={degree}", ls='--')

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left',borderaxespad=0.0, shadow=True)
plt.tight_layout()
plt.ylim(-2,3)
plt.grid()
plt.show()
```

Newton-Raphson

Example: $f(x) = x^2 + 4x + 4$

```
Initial Guess =  -3
next guess is xnew = -2.5
next guess is xnew = -2.25
next guess is xnew = -2.125
next guess is xnew = -2.0625
next guess is xnew = -2.03125
next guess is xnew = -2.015625
next guess is xnew = -2.0078125
next guess is xnew = -2.00390625
```



Finally

ROOT_SCALAR

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

scipy.optimize.

root_scalar

`root_scalar(f, args=(), method=None, bracket=None, fprime=None, fprime2=None, x0=None, x1=None, xtol=None, rtol=None, maxiter=None, options=None)` [source]

Find a root of a scalar function.

Parameters:

f : *callable*

A function to find a root of.

args : *tuple, optional*

Extra arguments passed to the objective function and its derivative(s).

method : *str, optional*

Type of solver. Should be one of

- 'bisect' (see here)
- 'brentq' (see here)
- 'brenth' (see here)
- 'ridder' (see here)
- 'toms748' (see here)
- 'newton' (see here)
- 'secant' (see here)
- 'halley' (see here)

bracket: A sequence of 2 floats, optional

An interval bracketing a root. *f(x, *args)* must have different signs at the two endpoints.

x0 : *float, optional*

Initial guess.

x1 : *float, optional*

A second guess.

fprime : *bool or callable, optional*

If *fprime* is a boolean and is True, *f* is assumed to return the value of the objective function and of the derivative. *fprime* can also be a callable returning the derivative of *f*. In this case, it must accept the same arguments as *f*.

fprime2 : *bool or callable, optional*

If *fprime2* is a boolean and is True, *f* is assumed to return the value of the objective function and of the first and second derivatives. *fprime2* can also be a callable returning the second derivative of *f*. In this case, it must accept the same arguments as *f*.

xtol : *float, optional*

Tolerance (absolute) for termination.

rtol : *float, optional*

Tolerance (relative) for termination.

maxiter : *int, optional*

Maximum number of iterations.

METHOD	INPUT	Always Converges?	Breaks if	Notes
bisection	bracket	YES	$f \geq 0$ (or opposite) since we need opposite signed brackets	-slow and steady -sometimes hard to find bracket
Newton	derivatives, initial guess	NO	$f'(x_i) = 0$ at any time	-need equations for derivatives -FAST! -can get lost
Secant	two guesses	NO	$f(x_0) = f(x_1)$ at any time	-easiest to start -can make second guess as $x_0 + \Delta x$ -can also get lost

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

Fancier Methods

Domain of f	Bracket?	Derivatives? fprime	fprime2	Solvers	Convergence Guaranteed?	Rate(s)(*)
R	Yes	N/A	N/A	• bisection • brentq • brenth • ridder • toms748	• Yes • Yes • Yes • Yes • Yes	• 1 "Linear" • >=1, <= 1.62 • >=1, <= 1.62 • 2.0 (1.41) • 2.7 (1.65)
R or *C*	No	No	No	secant	No	1.62 (1.62)
R or *C*	No	Yes	No	newton	No	2.00 (1.41)
R or *C*	No	Yes	Yes	halley	No	3.00 (1.44)

per iter. (per fun. eval)

Arguments for each method are as follows (x=required, o=optional).

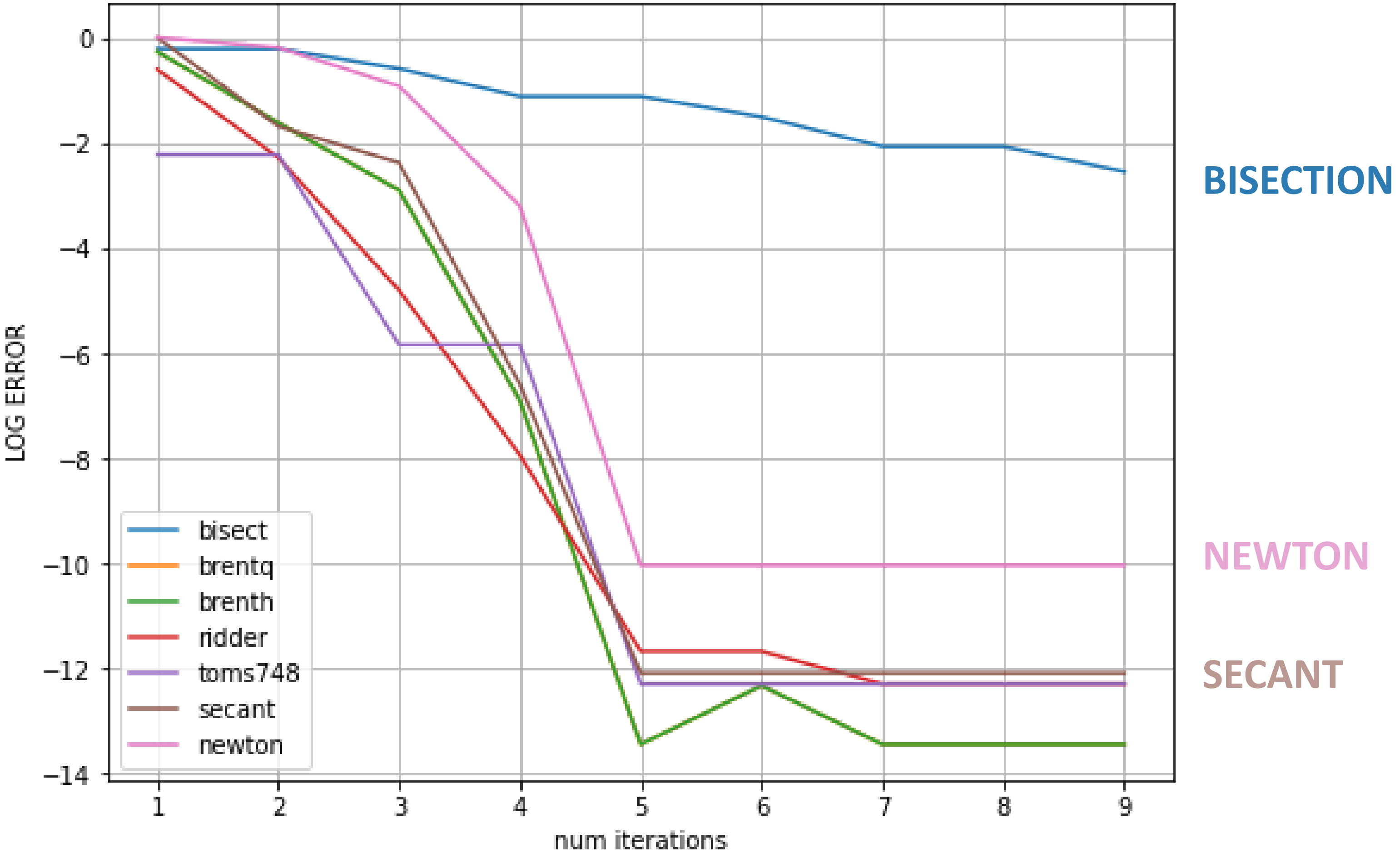
method	f	args	bracket	x0	x1	fprime	fprime2	xtol	rtol	maxiter	options
bisect	x	o	x					o	o	o	o
brentq	x	o	x					o	o	o	o
brenth	x	o	x					o	o	o	o
ridder	x	o	x					o	o	o	o
toms748	x	o	x					o	o	o	o
secant	x	o		x	o			o	o	o	o
newton	x	o		x		o		o	o	o	o
halley	x	o		x		x	x	o	o	o	o

The fancier methods are bracketed to guarantee convergence https://github.com/scipy/scipy/blob/v1.14.1/scipy/optimize/_root_scalar.py#L62

Logic for choosing method

```python
# Pick a method if not specified.
# Use the "best" method available for the situation.
if not method:
    if bracket:
        method = 'brentq'
    elif x0 is not None:
        if fprime:
            if fprime2:
                method = 'halley'
            else:
                method = 'newton'
        elif x1 is not None:
            method = 'secant'
        else:
            method = 'newton'
```

The default bracketed method is brentq, which is Algorithm M in: https://dl.acm.org/doi/10.1145/355656.355659
This is a “safe” version (always bracketed) of secant using hyperbolic extrapolation



Simple root finding method comparison. (The results are very sensitive to the problem and initial guesses)

- Bisection converges very slowly
- Newton is still bad
- The fancier bisection methods are fastest, but secant is almost as good

Dr. D’s Advice

- Plot the function
- Use brackets to know which root you’re getting
- If you just want an answer, use secant (x0=initial guess, x1=x0+1e-4), but know that you might not get the root closest to x0

Practice

Find all roots where $x > 0$ of the function:

$$f(x) = \sin x - 0.1\,x$$

Practice

The thermodynamic efficiency η of a certain engine is given by:

$$\eta = \frac{\ln(T_2/T_1) - (1 - T_1/T_2)}{\ln(T_2/T_1) + (1 - T_1/T_2)/(\gamma - 1)}$$

where $\gamma = \frac{5}{3}$. Find the temperature ratio $\frac{T_2}{T_1}$ where $\eta = 0.3$

(Be careful with the fractions here…)

Non-Linear Systems

We can handle systems of equations in a similar way using **root** instead of **root_scalar**

`scipy.optimize.root(fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None, options=None)` [source]

Find a root of a vector function.

Parameters:

fun : *callable*

A vector function to find a root of.

x0 : *ndarray*

Initial guess.

args : *tuple, optional*

Extra arguments passed to the objective function and its Jacobian.

```python
from scipy.optimize import root
```

```python
def f(x):
    return [x[0]-3, x[1]-4]
```

```python
root(f, [1,2])
```

```
    fjac: array([[-1.00000000e+00, -1.09079412e-12],
       [ 1.09079412e-12, -1.00000000e+00]])
     fun: array([0., 0.])
 message: 'The solution converged.'
    nfev: 5
     qtf: array([-4.36228831e-12, -4.36273240e-12])
       r: array([-1.00000000e+00, -2.18131069e-12, -1.00000000e+00])
  status: 1
 success: True
       x: array([3., 4.])
```

Things to Know

- Why root finding lets us solve for x
- How to use `root_scalar` in 3 different ways
- Jacobian = matrix of derivatives
- How to do multi-dimensional cases