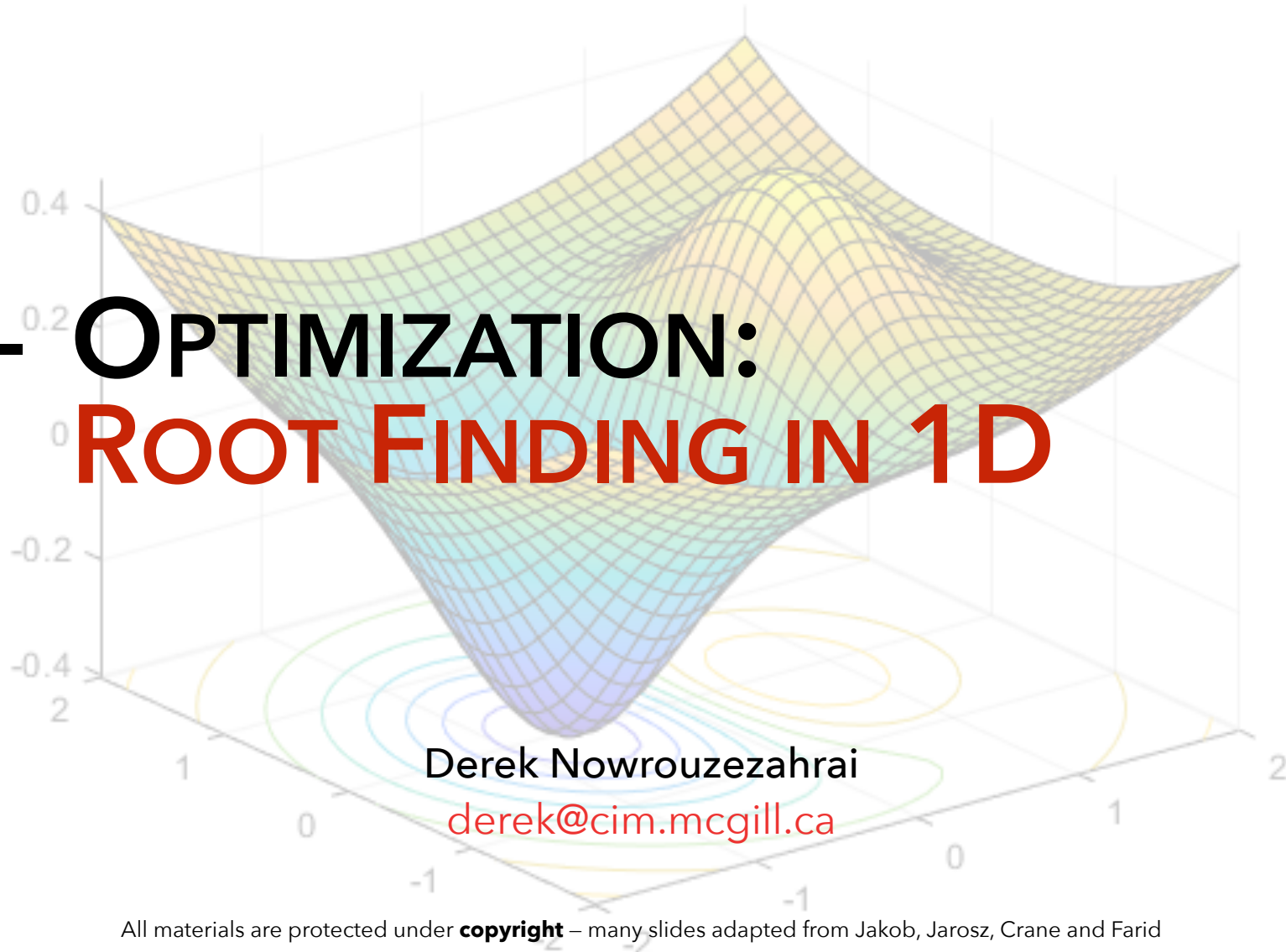


5A – OPTIMIZATION: ROOT FINDING IN 1D

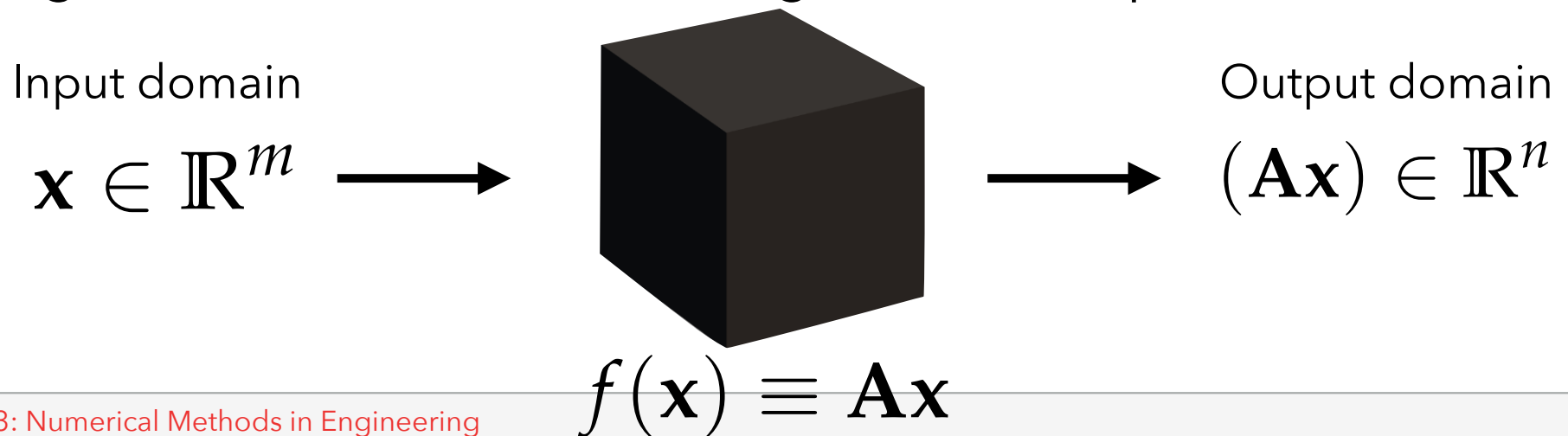


All materials are protected under **copyright** – many slides adapted from Jakob, Jarosz, Crane and Farid

Linear Models

We've only considered *linear functions*, where we saw:

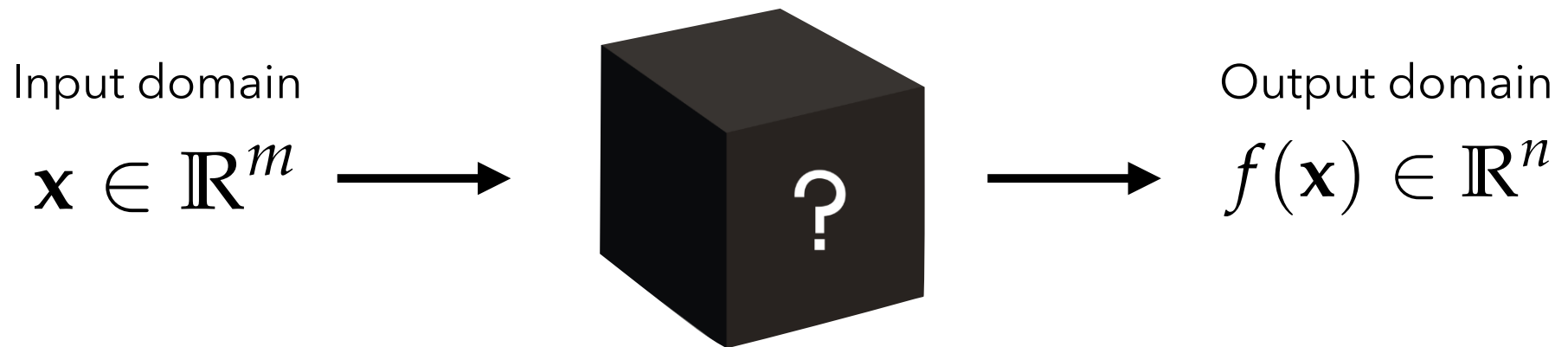
- that every linear map $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ of independent variables \mathbf{x} is fully specified by an $n \times m$ matrix \mathbf{A}
- how to solve linear problems using linear algebra and algorithms based on linear algebraic manipulations



Nonlinear Models

What happens if we can't assume that a model is linear?

- how do we treat functions about which we know very little?
- what are the minimal impositions we need to make on f before we can perform any meaningful analysis?



Let's start simple...

Consider a function that maps a (1D) real number to another (1D) real number:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

If f was linear, solving a system involving f would be trivial

- unlike the linear setting, solving *non-linear systems* of equations – with even a single input and output dimension – can be quite challenging

Let's start simple... or not?

Linear systems only have 0, 1 or infinitely many solutions

This is not the case for non-linear systems of equations

- consider some examples of non-linear $f : \mathbb{R} \rightarrow \mathbb{R}$

$$\exp(x) + 1 = 0$$

0 solutions

$$\exp(-x) - x = 0$$

1 solution

$$x^2 - 4 \sin(x) = 0$$

2 solutions

$$x^3 - 6x^2 + 11x - 6 = 0$$

3 solutions

$$\sin(x) = 0$$

Infinitely many solutions

[Heath]

Let's start simple... or not?

Also, it's not too soon to start thinking about how you might go about solving such an equation, e.g., $\sin(x) = 0$

- what if the function's form isn't as amenable to mathematical manipulation?

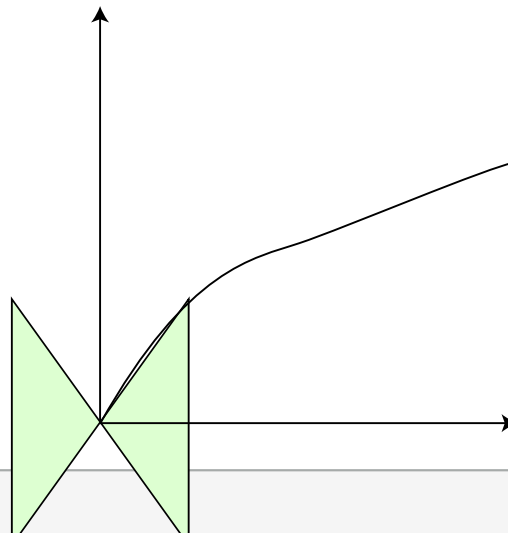
$$f_1(x) := \begin{cases} 1, & x \in \mathbb{Q} \\ -1, & x \in \mathbb{R} \setminus \mathbb{Q} \end{cases} \quad f_2(x) := \begin{cases} 1, & x \neq 0.13525634 \\ 0, & x = 0.13525634 \end{cases}$$

- maintaining generality, what impositions should we enforce/assume to avoid a "try all possible values"-type of solution?

Some Assumptions on f

Applying just a few impositions on f will allow us to perform many types of interesting analyses:

- continuity: $f(x) \rightarrow f(y)$ as $x \rightarrow y$
- Lipschitz continuity: $|f(x) - f(y)| \leq c|x - y|$ for some fixed $c \in \mathbb{R}$



Some Assumptions on f

Applying just a few impositions on f will allow us to perform many types of interesting analyses:

- continuity: $f(x) \rightarrow f(y)$ as $x \rightarrow y$
- Lipschitz continuity: $|f(x) - f(y)| \leq c|x - y|$ for some fixed $c \in \mathbb{R}$
- differentiability: $f'(x)$ exists for every x in the input domain
- C^k differentiability: $\frac{\partial^i}{\partial x^i} f \in C$ ($i = 1, \dots, k$)

Non-linear Equations – Root Finding

The first type of non-linear equations we will try to solve are so-called *root finding* problems, of the form:

$$\text{Solve for } \mathbf{x} \text{ in } f(\mathbf{x}) = \mathbf{y}$$

which we can always transform to the form:

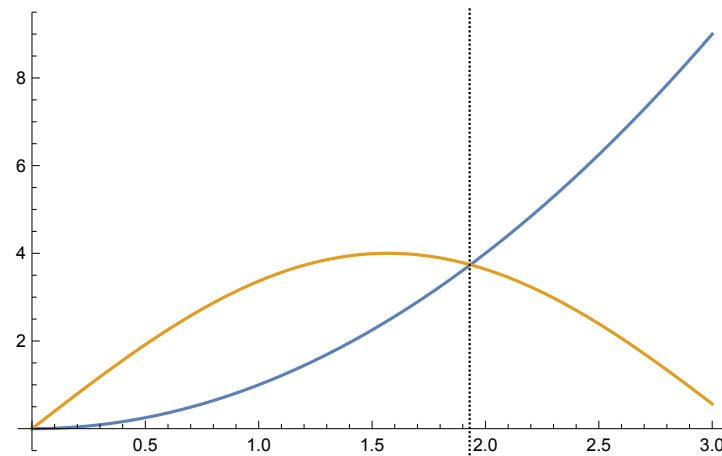
$$\text{Solve for } \mathbf{x} \text{ in } g(\mathbf{x}) = 0$$

Note that the linear setting is a special case of root finding, with $\mathbf{Ax} - \mathbf{b} = 0$ (or \approx in over/underdetermined cases)

Root-finding – Example

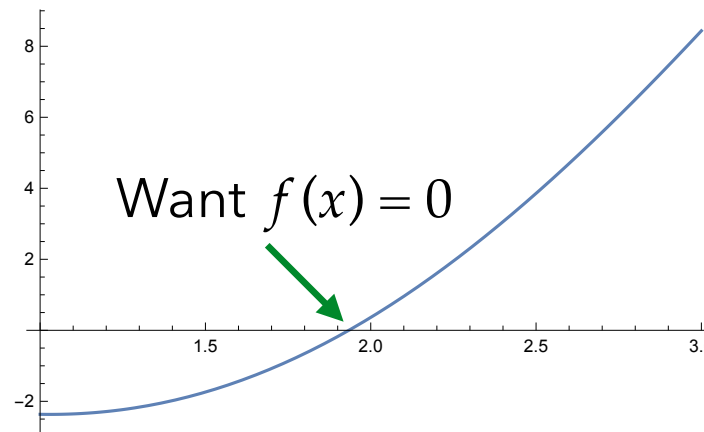
While some non-linear equations may admit analytic solutions, even seemingly innocuous equations may not

- take, for example, $x^2 = 4 \sin x$, which has no analytic solution
- this fact motivates the need for *numerical solutions*



Root-finding – Example

$$f(x) := x^2 - 4 \sin x$$



If we plot the function, we can “simply” read the solution

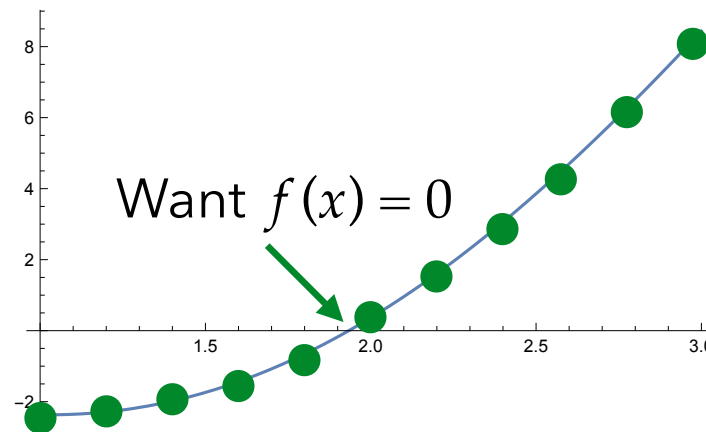
- why isn't this a good strategy for a general function?

Intermediate Value Theorem

With our assumptions, we don't need a lot of function evaluations to deduce important facts about its form

- even a few sparse evaluations can be revealing of f 's structure

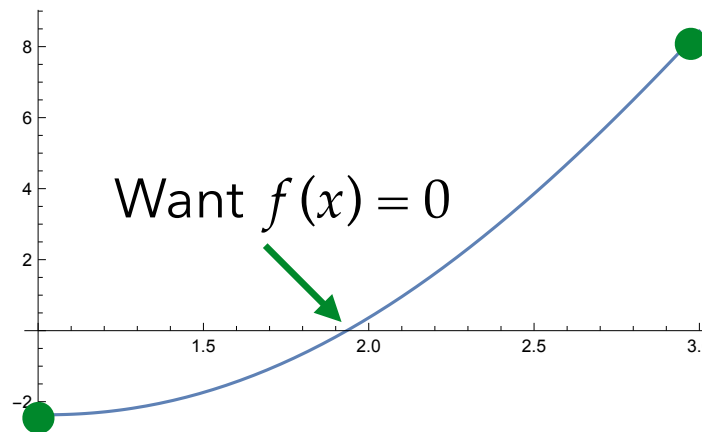
$$f(x) := x^2 - 4 \sin x$$



Intermediate Value Theorem

If f is continuous, the *intermediate value theorem* states that, given $f(x_0) = y_0$ and $f(x_1) = y_1$, then $f(x)$ must take on **every** value between y_0 and y_1 (for some $x \in [x_0, x_1]$)

$$f(x) := x^2 - 4 \sin x$$

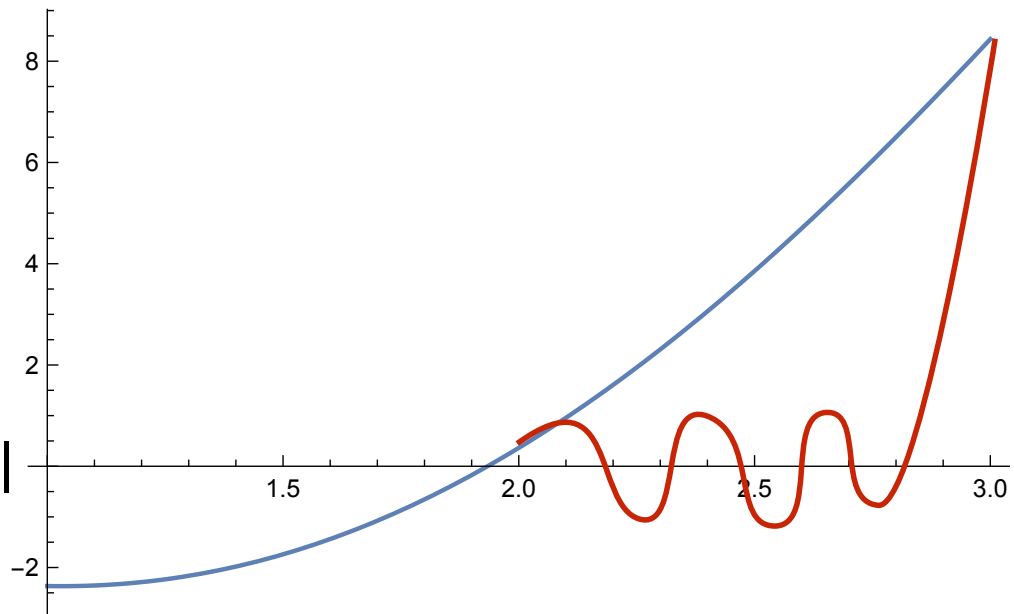


From IVT to a Binary Search Strategy

We can leverage the intermediate value theorem to devise a binary divide-and-conquer strategy to find a **single root** of our function

Here, we ignore that it is challenging to:

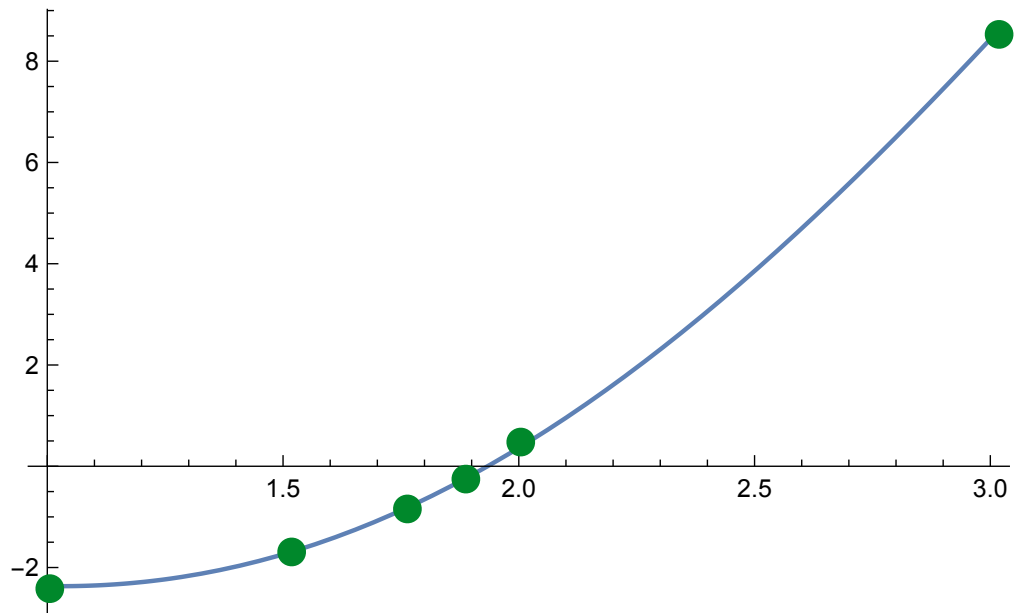
- know how many roots the function has, and
- whether we've found them all



From IVT to a Binary Search Strategy

We must additionally assume that we have access to a **bracket** within which we know there is **at least one** root

If so, we start from our bracket end points, and iteratively shrink the bracket (e.g., recursively) until it is sufficiently tight to the root



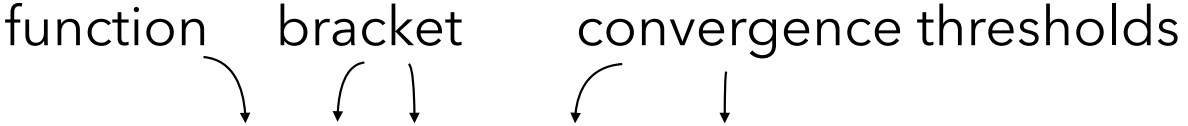
Algorithm #1 – Bisection Search

```
def bisectionRootFinding(f, l, r, eps1, eps2):  
    while True:  
        # find the midpoint  
        m = l + (r - l) / 2  
        # termination criteria  
        if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
            return m  
        # update bracket  
        if f(l) * f(m) < 0:  
            r = m  
        else:  
            l = m
```


Algorithm #1 – Bisection Search

function bracket convergence thresholds

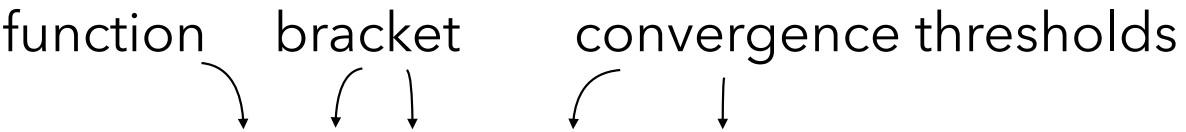
```
def bisectionRootFinding(f, l, r, eps1, eps2):  
    while True:  
        # find the midpoint  
        m = l + (r - l) / 2  
        # termination criteria  
        if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
            return m  
        # update bracket  
        if f(l) * f(m) < 0:  
            r = m  
        else:  
            l = m
```



Algorithm #1 – Bisection Search

function bracket convergence thresholds

```
def bisectionRootFinding(f, l, r, eps1, eps2):  
    while True:  
        # find the midpoint  
        m = l + (r - l) / 2  
        # termination criteria  
        if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
            return m  
        # update bracket  
        if f(l) * f(m) < 0:  
            r = m  
        else:  
            l = m
```



Algorithm #1 – Bisection Search

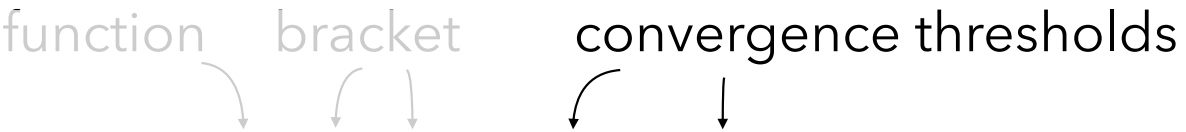
function bracket convergence thresholds

```
def bisectionRootFinding(f, l, r, eps1, eps2):  
    while True:  
        # find the midpoint  
        m = l + (r - l) / 2  
        # termination criteria  
        if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
            return m  
        # update bracket  
        if f(l) * f(m) < 0:  
            r = m  
        else:  
            l = m
```

Algorithm #1 – Bisection Search

function bracket convergence thresholds

```
def bisectionRootFinding(f, l, r, eps1, eps2):  
    while True:  
        # find the midpoint  
        m = l + (r - l) / 2  
        # termination criteria  
        if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
            return m  
        # update bracket  
        if f(l) * f(m) < 0:  
            r = m  
        else:  
            l = m
```



Bisection Termination Criteria

There are many termination heuristics we can employ

Our implementation implements the following criteria:

- return a root when the function evaluation is "*sufficiently*" small
 - as with many heuristics, depending on a user parameter eps_1

```
# termination criteria
if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:
    return m
```

Bisection Termination Criteria

There are many termination heuristics we can employ

Our implementation implements the following criteria:

- return a root when the function evaluation is "*sufficiently*" small
- return a root when the bracket is "*sufficiently*" tight
 - again, depending on a user defined tightness parameter eps_2

```
# termination criteria
if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:
    return m
```

- can, e.g., similarly consider a sliding window over the iterates

Bisection Termination Criteria

```
# termination criteria  
if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
    return m
```

We can also interpret these two criteria according to backward and forward error

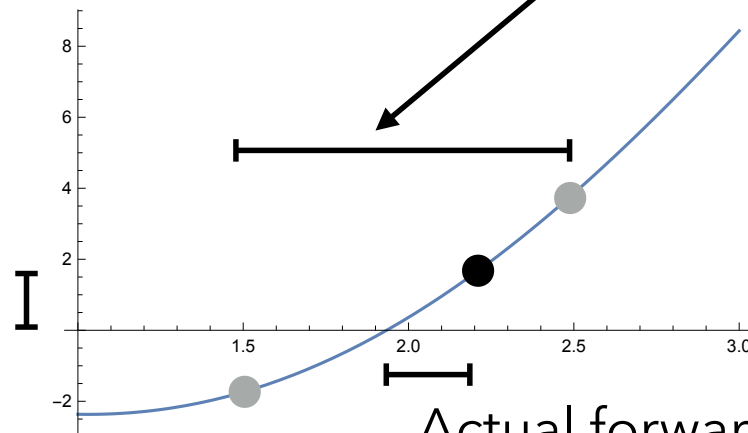
- $\text{np.abs}(f(m))$ is the backward error of our root estimate
- $\text{np.abs}(l-r)$ bounds the forward error of estimate
 - recall - forward error bounds are not typically easy to come by

Bisection Termination Criteria

```
# termination criteria  
if np.abs(f(m)) < eps1 or np.abs(l-r) < eps2:  
    return m
```

Backward error

Bound on forward error
(we don't always have this)



Actual forward error (unknown)

Bisection Search vs. Binary Search

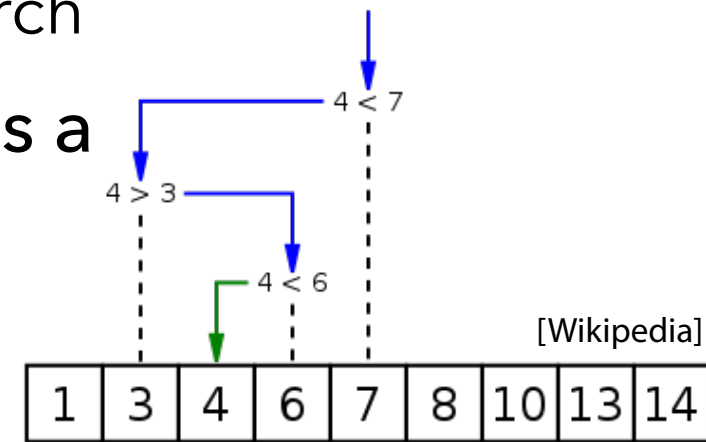
Having flashbacks to your DS&A course?

- bisection search is similar to binary search

Asymptotic time complexity analysis is a useful tool in the context of DS&A

- recall – the time complexity of *binary search* is $O(\log n)$ for an n -element list

Such an analysis cannot, unfortunately, apply as-is to our problem (i.e., $n = \infty$)



e.g., binary search for 4 in sorted list

Bisection– Convergence Analysis

DS&A asymptotic complexity analyses aim to bound the *compute* (or *memory*) as a function of *input size*

In contrast, one type of **complexity analysis for numerical methods** aims to bound *error* as a function of *compute* (e.g., the number of *iterations*)

- in the context of iterative solvers, like bisection search, *compute* is proportional to the *number of iterations*

We call this a convergence analysis

Bisection– Convergence Analysis

In the case of bisection, the error analysis isn't too complicated:

- at the start (i.e., before running an iteration of the search), we can bound the *maximum error* of our root estimate as $r - l$
- after one iteration, the *maximum error* bound is $(r - l)/2$
- in general for bisection search, $E_{k+1} \leq \frac{1}{2}E_k$ with $E_0 \leq r - l$
 - gain one decimal digit of precision every $\frac{\log 10}{\log 2} \approx 3$ iterations

Order of Convergence

When performing a convergence analysis, one telling measure is the *convergence rate*: the relative progress (in terms of error reduction) we make with every iteration

- considering the limit behaviour of the ratio between the error at iteration $k + 1$ and the error at the previous iteration k , raised to an exponent r ,

$$\text{if } \lim_{k \rightarrow \infty} \frac{E_{k+1}}{E_k^r} = C \text{ holds, for some } C \in \mathbb{R},$$

then r is the *convergence rate* of the method

Order of Convergence

$$\lim_{k \rightarrow \infty} \frac{E_{k+1}}{E_k^r} = C \text{ for some } C \in \mathbb{R}$$

A convergence rate of $r = 1$ is called *linear convergence*

- effectively gain 1 mantissa bit of precision per iteration
 - with bisection search, we had $r = 1$ and $C = 1/2$

A convergence rate of $r = 2$ is called *quadratic convergence*

- number of correct digits *doubles* every iteration

A convergence rate of $r = 3$ is called *cubic convergence*

- number of correct digits *triples* every iteration

Order of Convergence – Example

Revisiting our example of applying bisection search to the root-finding problem with $f(x) := x^2 - 4 \sin x$

- we observe the following iterative behaviour
 - recall: the root solution must lie between the brackets

l	$f(l)$	r	$f(r)$	l	$f(l)$	r	$f(r)$
1.000000	-2.365884	3.000000	8.435520	1.933594	-0.000846	1.937500	0.019849
1.000000	-2.365884	2.000000	0.362810	1.933594	-0.000846	1.935547	0.009491
1.500000	-1.739980	2.000000	0.362810	1.933594	-0.000846	1.934570	0.004320
1.750000	-0.873444	2.000000	0.362810	1.933594	-0.000846	1.934082	0.001736
1.875000	-0.300718	2.000000	0.362810				
1.875000	-0.300718	1.937500	0.019849				
1.906250	-0.143255	1.937500	0.019849				
1.921875	-0.062406	1.937500	0.019849				
1.929688	-0.021454	1.937500	0.019849				

Try tabulating the *error*,
across iterations...

Bisection Search – Summary

Some take aways about bisection root finding:

- linear convergence rate
 - could be worse (i.e., sub-linear), but we may hope to do better
- of all our impositions, bisection only exploits **continuity**
- bisection only considers the function's *sign*
- requires a bracket for at least one root
 - this begs at least one question: how do we *find* valid brackets?
- under these circumstances, bisection is **guaranteed** to (eventually) converge to **a** root

What's Next?

Next, we'll consider root finding approaches that:

- exploit the **derivative** impositions we've made on our function, and
- use more than just the sign of the function, at evaluation points

But, first...



Review – Taylor Series Expansion

Series Expansions of Functions

There are many ways to re-write continuous functions as a sum of elements in a series, such as with:

- Fourier series (coming soon!), or
- Maclaurin and Taylor series (coming even sooner)

Such expansions differ in – at a high-level – according to:

- the method with which we obtain a function's reformulation,
- the form of the terms in the series, and
- the potential benefits of the reformulation (both theoretical and practical)

Series Expansions – Maclaurin

Maclaurin series expansions of functions are a simple, indeed limited, starting point

- a Maclaurin series expansion of a function *assumes* that:

1. the function f can be represented as a power series in x

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots = \sum_{p=0}^{\infty} a_p x^p$$

from which we seek expressions for the power coefficients, a_p

- *not all functions can be represented as a power series*

2. the function f is C^k differentiable (sound familiar?)

Series Expansions – Maclaurin

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots = \sum_{p=0}^{\infty} a_p x^p$$

To obtain the a_p , we leverage the form of the power series (and its derivatives), and our underlying assumptions on f

- first, consider evaluating $f(0)$
 - clearly, $f(0) = a_0$
- now, consider evaluating $f'(0) = f^{(1)}(0)$ and arriving at $f^{(1)}(0) = a_1$
- how about $f''(0) = f^{(2)}(0)$?
 - Careful! $f^{(2)}(0) = 2a_2$
- how about $f'''(0) = f^{(3)}(0)$?

Series Expansions – Maclaurin

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots = \sum_{p=0}^{\infty} a_p x^p$$

In general, we see a pattern forming:

$$a_0 = f(0) \quad a_1 = f^{(1)}(0) \quad a_2 = \frac{1}{2}f^{(2)}(0) \quad a_3 = \frac{1}{6}f^{(3)}(0) \quad a_4 = \frac{1}{24}f^{(4)}(0)$$

$$a_p = \frac{1}{p!} f^{(p)}(0)$$

$$f(x) = \sum_{p=0}^{\infty} \frac{1}{p!} f^{(p)}(0) x^p$$

Series Expansions – Maclaurin

$$a_p = \frac{1}{p!} f^{(p)}(0) \quad f(x) = \sum_{p=0}^{\infty} \frac{1}{p!} f^{(p)}(0) x^p$$

We can arrive at another perspective on how Maclaurin coefficients “control” properties of our polynomial fit:

- we solved for an expression for a_0 by evaluating f at $x = 0$
 - this also “forces” the series expansion to match/reproduce the function’s value at $x = 0$
- we similarly solved for a_1 by evaluating $f^{(1)}$ at $x = 0$
 - this instead “forces” the series expansion’s **1st derivative** to match/reproduce the function’s **1st derivative** at $x = 0$

Series Expansions – Maclaurin

$$a_p = \frac{1}{p!} f^{(p)}(0) \quad f(x) = \sum_{p=0}^{\infty} \frac{1}{p!} f^{(p)}(0) x^p$$

We can arrive at another perspective on how Maclaurin coefficients “control” properties of our polynomial fit:

- generally, the Maclaurin coefficient a_i sets a constraint for our polynomial's i^{th} derivative at $x = 0$

Maclaurin Series Expansion – Example

Determine the Maclaurin expansion for $\cos(x)$

- here, you can assume that such an expansion exists

$$f(x) = \sum_{p=0}^{\infty} \frac{1}{p!} f^{(p)}(0) x^p \quad \cos(x) = \sum_{p=0}^{\infty} \frac{1}{p!} \frac{d^p}{dx^p} [\cos(x)] \Big|_0 x^p$$

$$\begin{array}{llll} \frac{d^0}{dx^0} \cos(x) = \cos(x) & \frac{d^2}{dx^2} \cos(x) = -\cos(x) & \frac{d^4}{dx^4} \cos(x) = \cos(x) & a_p \\ \frac{d^1}{dx^1} \cos(x) = -\sin(x) & \frac{d^3}{dx^3} \cos(x) = \sin(x) & & \end{array}$$

- note how the odd-numbered derivatives evaluate to 0:

$$a_0 = 1 \quad a_1 = 0 \quad a_2 = -1/(2!) \quad a_3 = 0 \quad a_4 = 1/(4!)$$

Series Expansions – Taylor

A Taylor series expansion generalizes Maclaurin, adopting its assumptions, but now treating function representation as a power series in $x - x_0$, instead of x

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + a_3(x - x_0)^3 + \dots$$

$$f(x) = \sum_{p=0}^{\infty} a_p(x - x_0)^p$$

where the expansion is said to be *centered* at a user-chosen point x_0

- here, the connection to Maclaurin is clear when $x_0 = 0$

Series Expansions – Taylor

$$f(x) = \sum_{p=0}^{\infty} a_p (x - x_0)^p$$

We similarly seek expressions for the series coefficients a_p and assume we have access to any derivative of f

- instead of evaluating (derivatives of) f at 0 to zero out the series' terms as we did for Maclaurin, we evaluate at them at x_0
- I leave the mechanics of these derivations as a take home exercise, and provide the final expression, here:

$$a_p = \frac{1}{p!} f^{(p)}(x_0) \quad f(x) = \sum_{p=0}^{\infty} \frac{1}{p!} f^{(p)}(x_0) (x - x_0)^p$$

Series Expansions – Taylor

Clamping the expansion to a fixed number of terms, we can approximate the original function to varying degrees of accuracy (i.e., as we increase the number of terms)

- an n^{th} -order *approximation* of a function typically refers to a clamped Taylor expansion using only the terms a_i for $i \in [0, n]$
 - e.g., a 1st-order expansion – sometimes called a *linear approximation* – includes the constant (a_0) & linear (a_1) terms

As with Maclaurin, we can arrive at the Taylor coefficient a_i by constraining the expansion's i^{th} derivative at $x = x_0$

Take-home Math & Coding Exercise

Derive the 1^{st} - and 2^{nd} -order Taylor expansion of e^x and visually compare the accuracy of these two approximations to the original function

- vary the the expansion's centering point
- bonus: use the sympy package to *programmatically derive* the expressions and then generalize to any order

What observations do you make regarding the accuracy (and inaccuracy) of the approximation, as you increase the order?

Take-home Math & Coding Exercise

