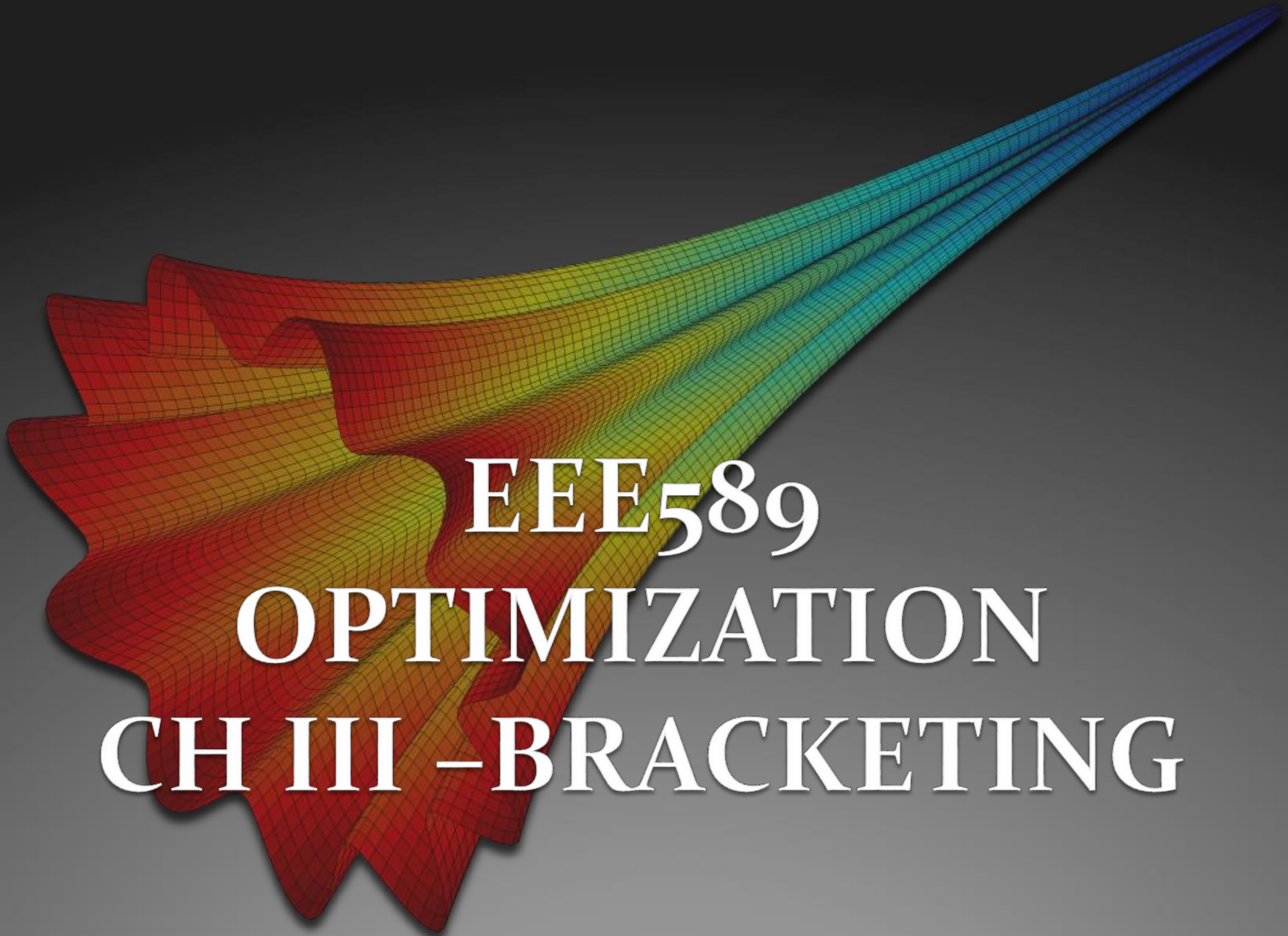


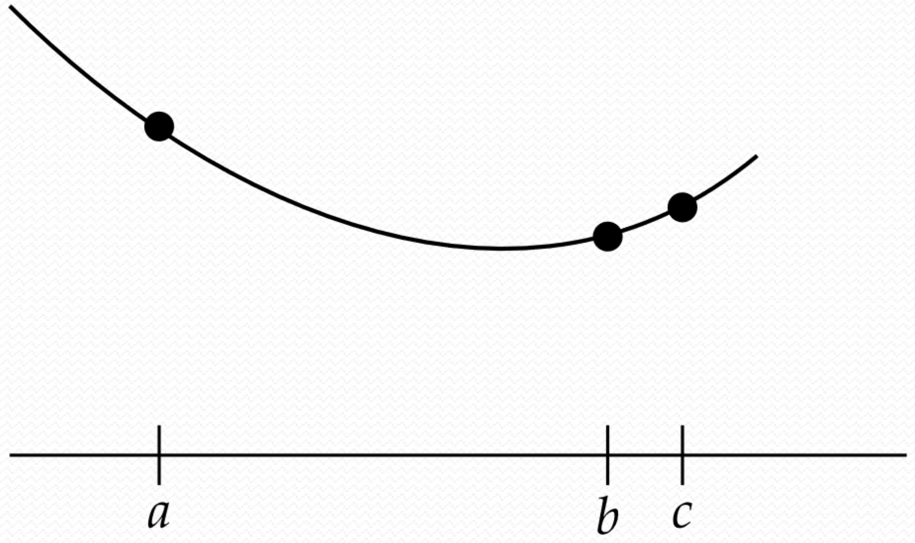
EEE589 OPTIMIZATION CH III –BRACKETING

Introduction

- This lecture presents a variety of *bracketing* methods for univariate functions, or functions involving a single variable.
- Bracketing is the process of identifying an interval in which a local minimum lies and then successively shrinking the interval.
- For many functions, derivative information can be helpful in directing the search for an optimum, but, for some functions, this information may not be available or might not exist.
- A wide variety of approaches that leverage different assumptions will be outlined.

Unimodality

- A *unimodal function* f is one where there is a *unique* x∗, such that f is monotonically decreasing for x ≤ x∗ and monotonically increasing for x ≥ x∗.
- It follows from this definition that the unique global minimum is at x∗, and there are no other local minima.
- Given a unimodal function, we can *bracket* an interval [a, c] containing the global minimum if we can find three points a < b < c, such that f (a) > f (b) < f (c).
- The global minimum is guaranteed to be inside the interval [a,c] if f (a) > f (b) < f (c)



Finding an Initial Bracket

- When optimizing a function, we often start by first bracketing an interval containing a local minimum.
- We then successively reduce the size of the bracketed interval to converge on the local minimum.
- A simple procedure (algorithm) can be used to find an initial bracket.

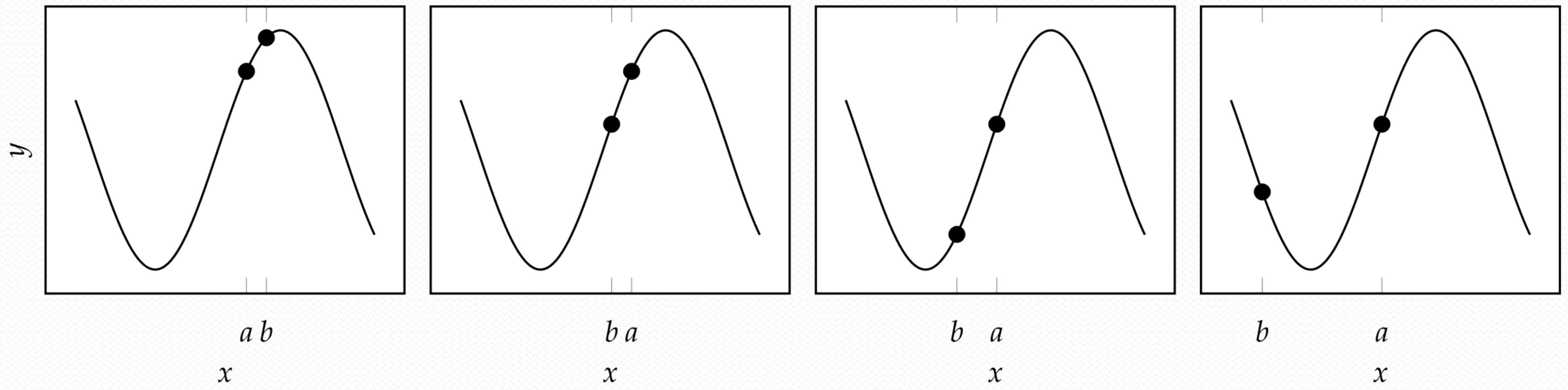
```julia
function bracket_minimum(f, x=0; s=1e-2, k=2.0)
    a, ya = x, f(x)
    b, yb = a + s, f(a + s)
    if yb > ya
        a, b = b, a
        ya, yb = yb, ya
        s = -s
    end
    while true
        c, yc = b + s, f(b + s)
        if yc > yb
            return a < c ? (a, c) : (c, a)
        end
        a, ya, b, yb = b, yb, c, yc
        s *= k
    end
end
```

Finding an Initial Bracket

- Starting at a given point, we take a step in the positive direction. The distance we take is a *hyperparameter* to this algorithm, but the algorithm provided defaults it to 1×10^{-2}.
- We then search in the downhill direction to find a new point that exceeds the lowest point.
- With each step, we expand the step size by some factor, which is another hyperparameter to this algorithm that is often set to 2.
- Functions without local minima, such as exp(x), cannot be bracketed and will cause bracket_minimum to fail.

Finding an Initial Bracket

- An example of running bracket_minimum on a function. The method reverses direction between the first and second iteration and then expands until a minimum is bracketed in the fourth iteration.



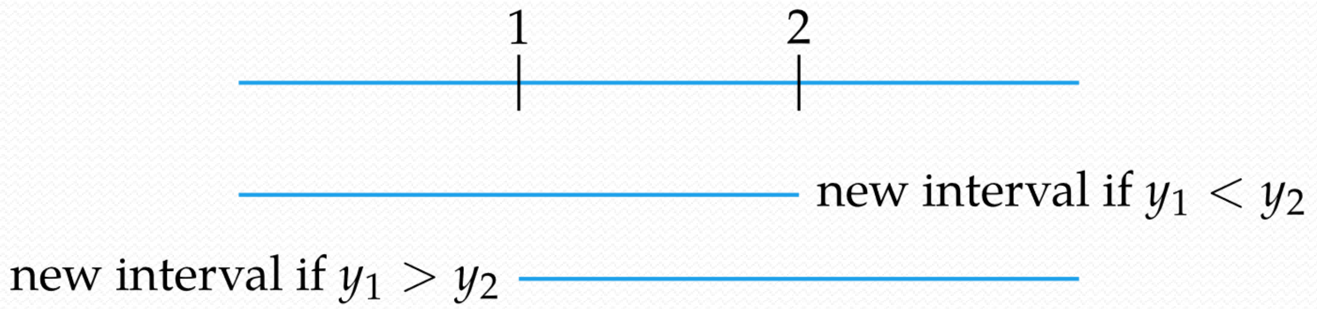
Fibonacci Search

- Suppose we have a unimodal f bracketed by the interval [a, b].
- Given a limit on the number of times we can query the objective function, *Fibonacci search* (algorithm) is guaranteed to maximally shrink the bracketed interval.
- When function evaluations are limited, the Fibonacci Search algorithm is guaranteed to maximally shrink the bracketed interval.

```julia
function fibonacci_search(f, a, b, n; ϵ=0.01)
    s = (1-√5)/(1+√5)
    ρ = 1 / (φ*(1-s^(n+1))/(1-s^n))
    d = ρ*b + (1-ρ)*a
    yd = f(d)
    for i in 1 : n-1
        if i == n-1
            c = ϵ*a + (1-ϵ)*d
        else
            c = ρ*a + (1-ρ)*b
        end
        yc = f(c)
        if yc < yd
            b, d, yd = d, c, yc
        else
            a, b = b, c
        end
        ρ = 1 / (φ*(1-s^(n-i+1))/(1-s^(n-i)))
    end
    return a < b ? (a, b) : (b, a)
end
```

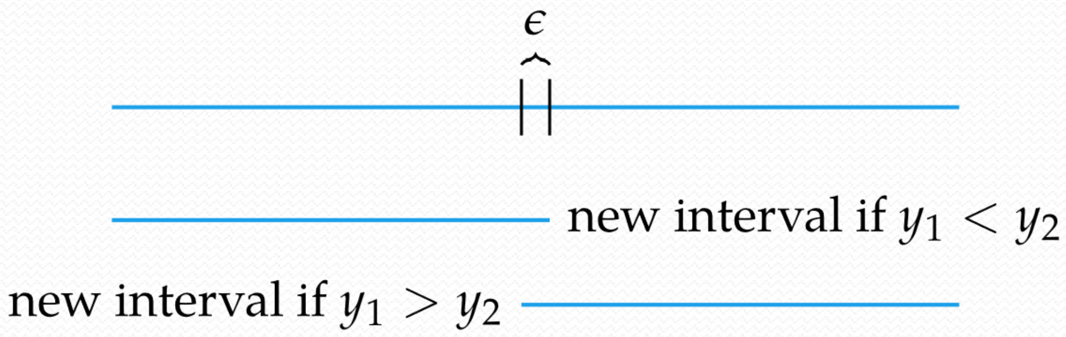
Fibonacci Search

- Suppose we can query f only twice. If we query f on the one-third and twothird points on the interval, then we are guaranteed to remove one-third of our interval, regardless of f , as shown in figure.
- When restricted to two function evaluations, compare



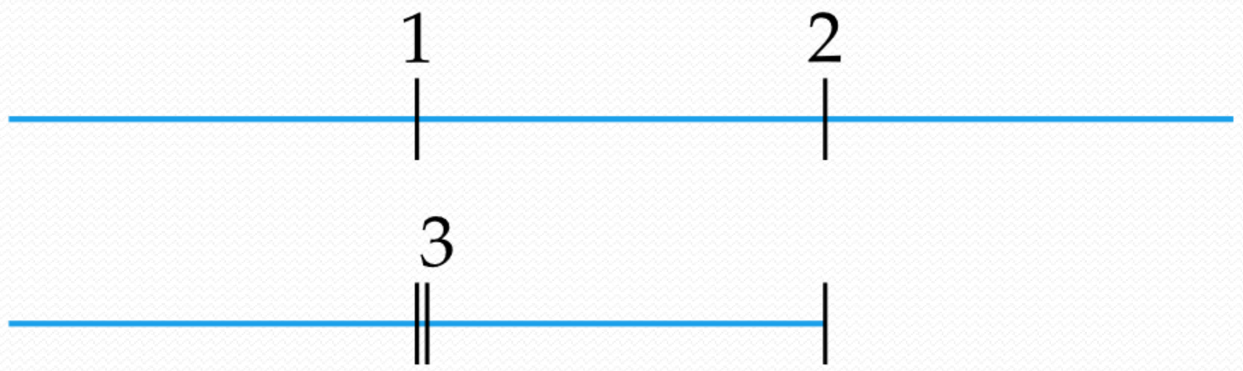
Fibonacci Search

- We can guarantee a tighter bracket by moving our guesses toward the center. In the limit as $\epsilon \to 0$, we are guaranteed to shrink our guess by a factor of two as shown in figure.



Fibonacci Search

- With three queries, we can shrink the interval by a factor of three. We first query f on the one-third and two-third points on the interval, discard one-third of the interval, and then sample just next to the better sample as shown in figure.
- When restricted to three function evaluations



Fibonacci Search

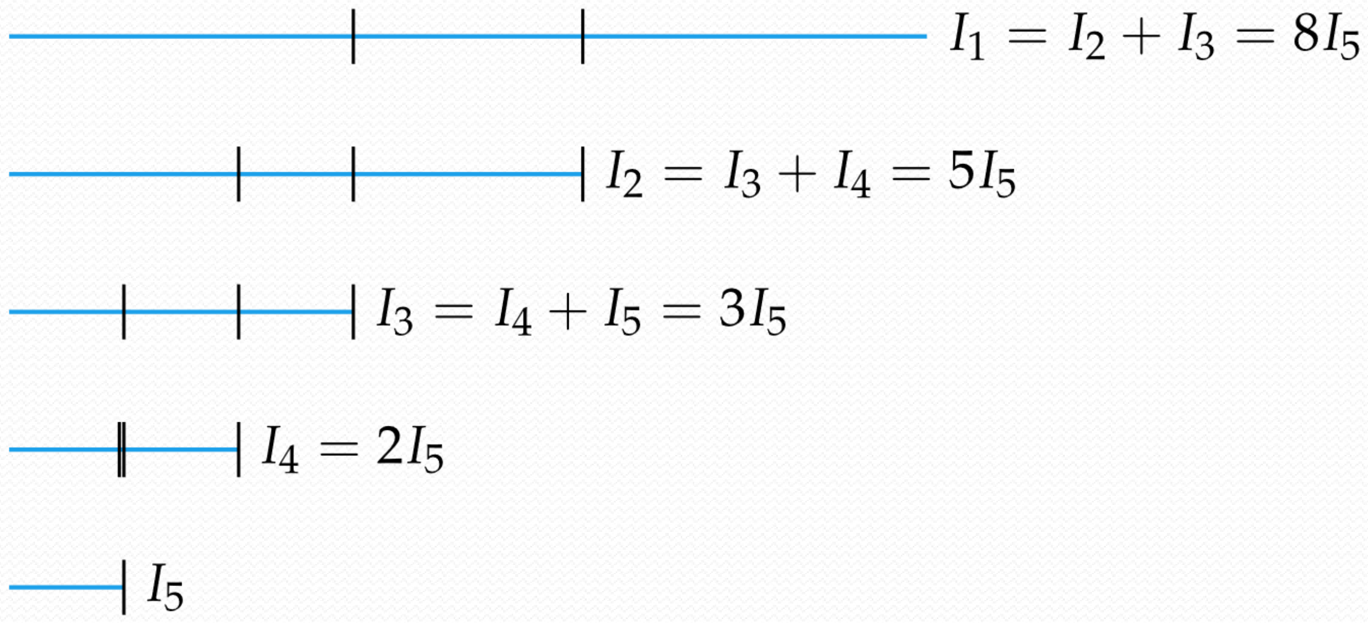
- For n queries, the length of the intervals are related to the Fibonacci sequence: 1, 1, 2, 3, 5, 8, and so forth. The first two terms are one, and the following terms are always the sum of the previous two:

- When restricted to n function evaluations

$$F_n = \begin{cases} 1 & \text{if } n \leq 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Fibonacci Search

- Figure shows the relationship between the intervals.



Fibonacci Search

- Binet’s formula defines a Fibonacci number analytically where φ is the Golden Ratio, $\varphi=(1+\sqrt{5})/2 \approx 1.61803$

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}},$$

- The ratio between successive Fibonacci numbers is

$$\frac{F_n}{F_{n-1}} = \varphi \frac{1 - s^{n+1}}{1 - s^n}$$

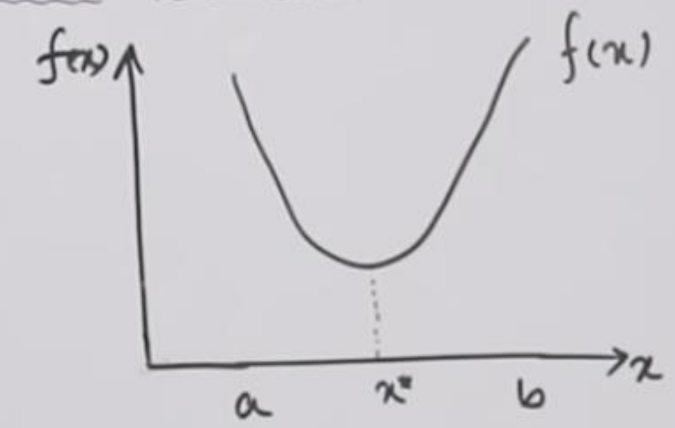
where $s = \frac{1-\sqrt{5}}{1+\sqrt{5}} \approx -0.382$

- Next example walks through an application to a univariate function.

Fibonacci Search Method

Algorithm

1. Choose a lower bound a and an upper bound b. Set $L = b - a$. Assume the desired number of function evaluations to be n. Set $k = 2$.
2. Compute $L_k^* = \left(\frac{F_{n-k+1}}{F_{n+1}}\right)L$
   Set $x_1 = a + L_k^*$ and $x_2 = b - L_k^*$
3. Compute $f(x_1)$ and $f(x_2)$
   Use fundamental region elimination rule to eliminate a region. Set new a and b.
4. If $k = n$ then stop. Else $k = k+1$ and go to step 2.



1, 1, 2, 3, 5, 8, 13, . . .

$\boxed{F_n = F_{n-1} + F_{n-2}}$, $F_0 = 1$, $F_1 = 1$
$n = 2, 3, \ldots$

$x_1 = a + L_k^*$
$x_2 = b - L_k^*$

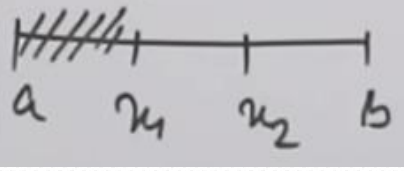
$L_k^* = \left(\frac{F_{n-k+1}}{F_{n+1}}\right)L$

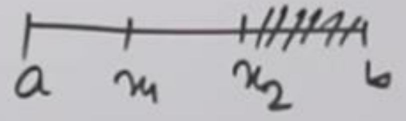
$f(x_1)$ & $f(x_2)$

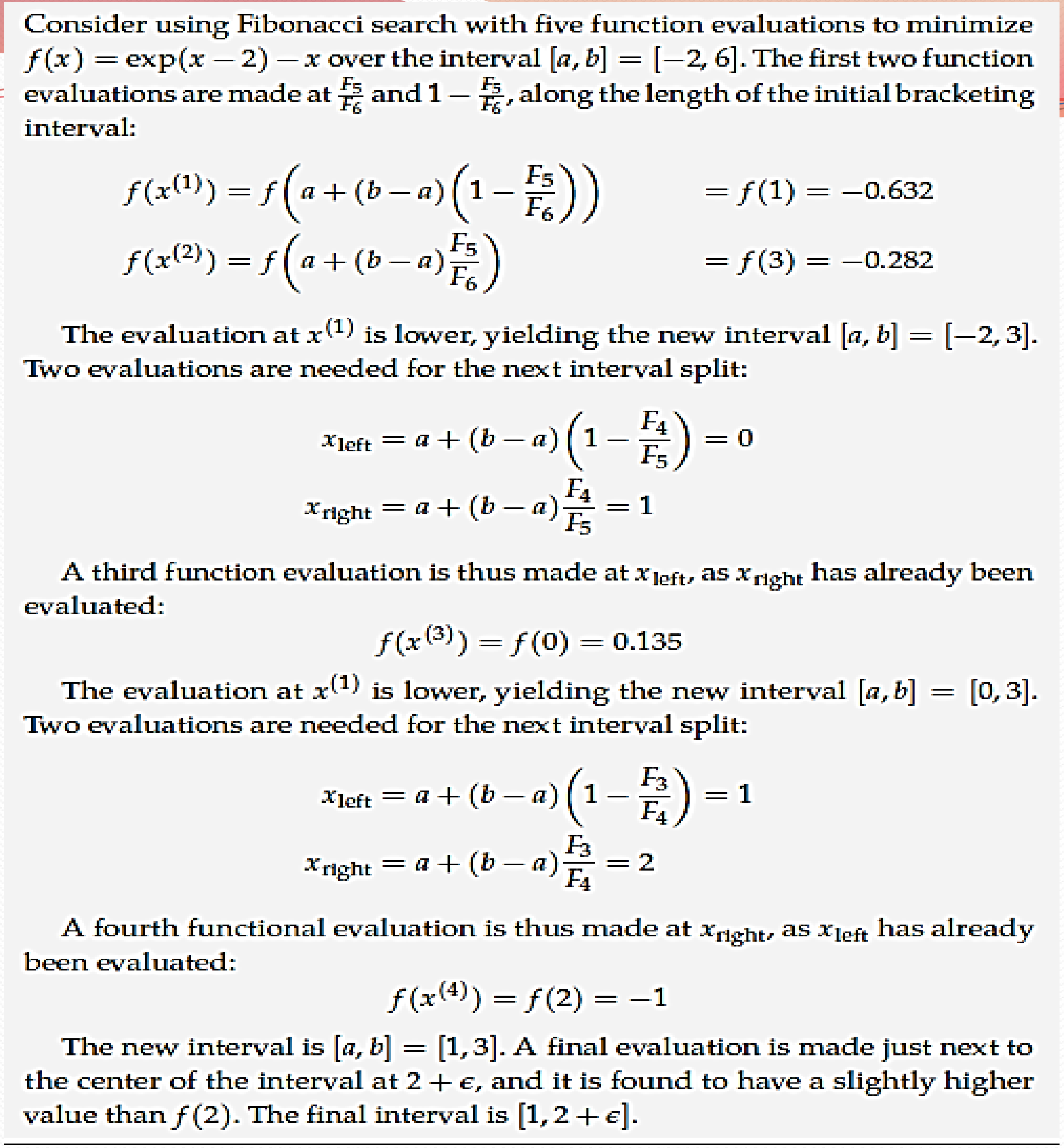
$\boxed{L = b - a}$

$f(x_1) > f(x_2)$

$f(x_1) < f(x_2)$







Consider using Fibonacci search with five function evaluations to minimize $f(x) = \exp(x-2) - x$ over the interval $[a, b] = [-2, 6]$. The first two function evaluations are made at $\frac{F_5}{F_6}$ and $1 - \frac{F_5}{F_6}$, along the length of the initial bracketing interval:

$$f(x^{(1)}) = f\left(a + (b-a)\left(1 - \frac{F_5}{F_6}\right)\right) \qquad = f(1) = -0.632$$

$$f(x^{(2)}) = f\left(a + (b-a)\frac{F_5}{F_6}\right) \qquad = f(3) = -0.282$$

The evaluation at $x^{(1)}$ is lower, yielding the new interval $[a, b] = [-2, 3]$. Two evaluations are needed for the next interval split:

$$x_{\text{left}} = a + (b-a)\left(1 - \frac{F_4}{F_5}\right) = 0$$

$$x_{\text{right}} = a + (b-a)\frac{F_4}{F_5} = 1$$

A third function evaluation is thus made at x_{left}, as x_{right} has already been evaluated:

$$f(x^{(3)}) = f(0) = 0.135$$

The evaluation at $x^{(1)}$ is lower, yielding the new interval $[a, b] = [0, 3]$. Two evaluations are needed for the next interval split:

$$x_{\text{left}} = a + (b-a)\left(1 - \frac{F_3}{F_4}\right) = 1$$

$$x_{\text{right}} = a + (b-a)\frac{F_3}{F_4} = 2$$

A fourth functional evaluation is thus made at x_{right}, as x_{left} has already been evaluated:

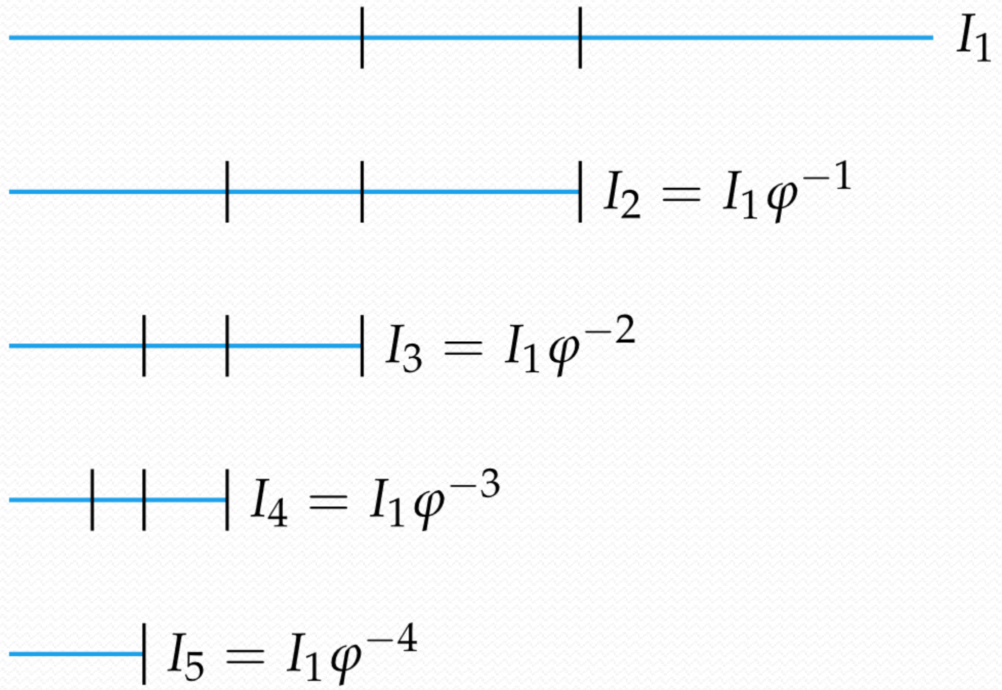
$$f(x^{(4)}) = f(2) = -1$$

The new interval is $[a, b] = [1, 3]$. A final evaluation is made just next to the center of the interval at $2 + \epsilon$, and it is found to have a slightly higher value than $f(2)$. The final interval is $[1, 2 + \epsilon]$.

Golden Section Search

- In the limit of large *N*, the ratio of successive Fibonacci numbers approaches the Golden Ratio, so φ can be used to perform approximate Fibonacci search

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \varphi$$



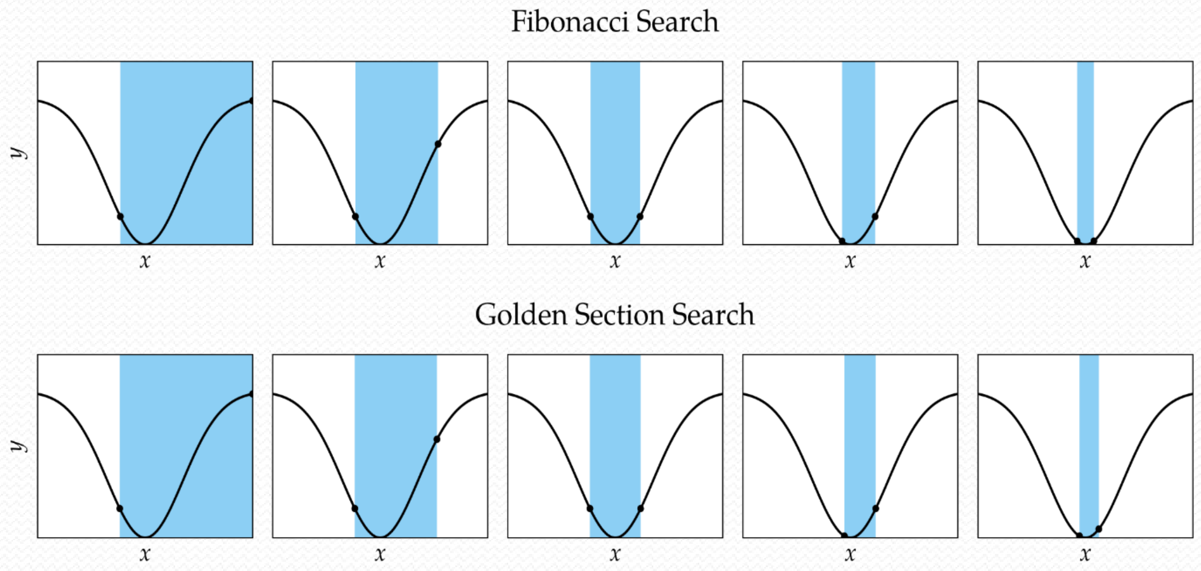
Golden Section Search

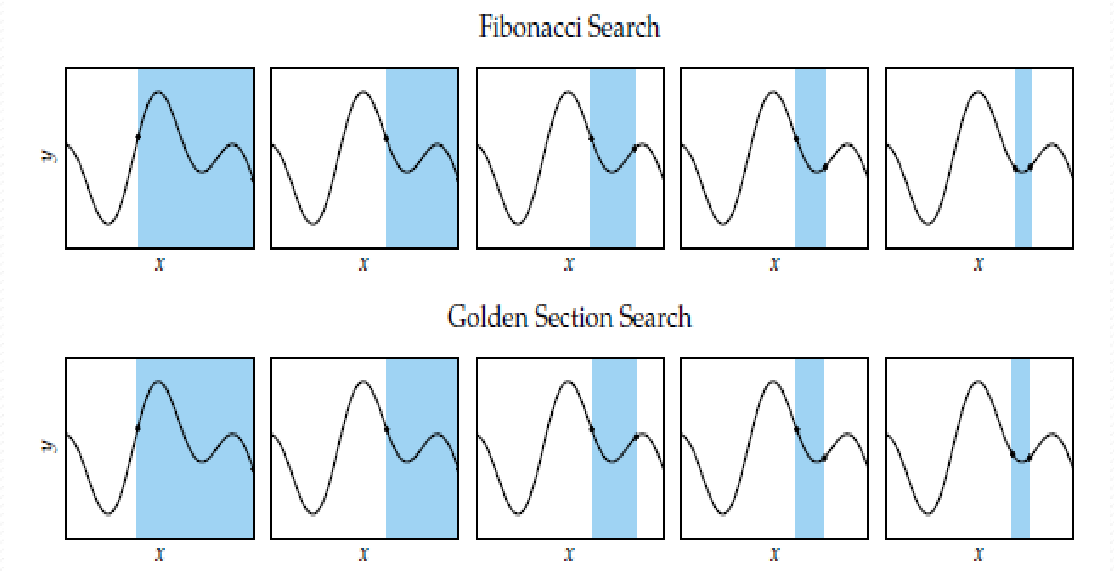
- *Golden section search* (algorithm) uses the golden ratio to approximate Fibonacci search.

```
function golden_section_search(f, a, b, n)
    ρ = φ-1
    d = ρ * b + (1 - ρ)*a
    yd = f(d)
    for i = 1 : n-1
        c = ρ*a + (1 - ρ)*b
        yc = f(c)
        if yc < yd
            b, d, yd = d, c, yc
        else
            a, b = b, c
        end
    end
    return a < b ? (a, b) : (b, a)
end
```

Fibonacci/Golden Section Search Comparison

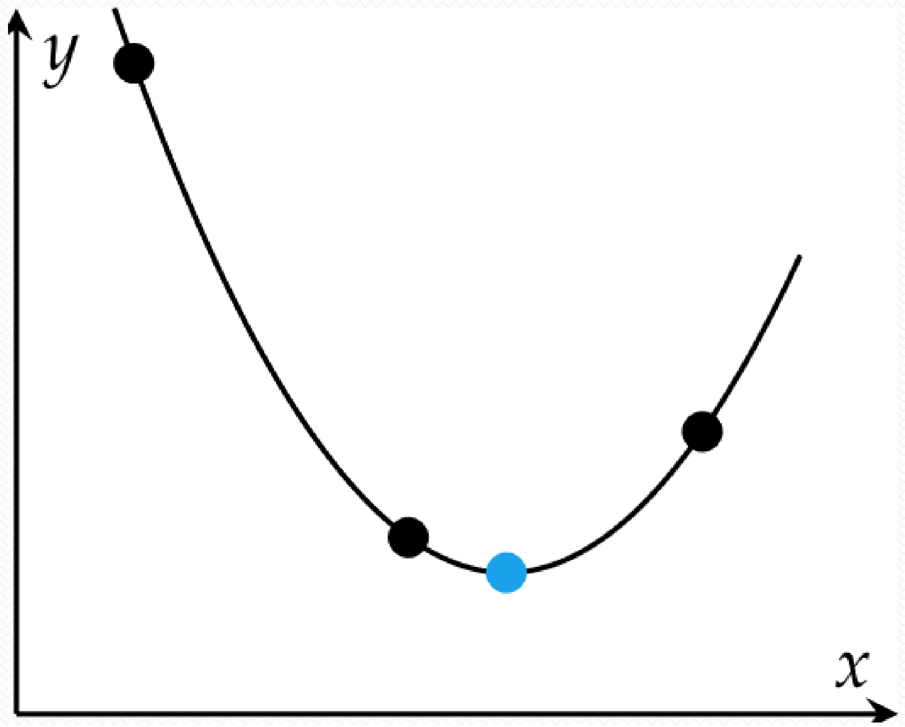
- Figures compare Fibonacci search with golden section search on unimodal and nonunimodal functions, respectively.





Quadratic Fit Search

- *Quadratic fit search* leverages our ability to analytically solve for the minimum of a quadratic function.
- Many local minima look quadratic when we zoom in close enough.
- Quadratic fit search iteratively fits a quadratic function to three bracketing points, solves for the minimum, chooses a new set of bracketing points, and repeats as shown in figure.



Quadratic Fit Search

- Given bracketing points a < b < c, we wish to find the coefficients p_1, p_2, and p_3 for the quadratic function q that goes through (a, y_a), (b, y_b), and (c, y_c):

$$
\begin{aligned}
q(x) &= p_1 + p_2 x + p_3 x^2 \\
y_a &= p_1 + p_2 a + p_3 a^2 \\
y_b &= p_1 + p_2 b + p_3 b^2 \\
y_c &= p_1 + p_2 c + p_3 c^2
\end{aligned}
$$

Quadratic Fit Search

In matrix form, we have

$$\begin{bmatrix} y_a \\ y_b \\ y_c \end{bmatrix} = \begin{bmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

We can solve for the coefficients through matrix inversion:

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{bmatrix}^{-1} \begin{bmatrix} y_a \\ y_b \\ y_c \end{bmatrix}$$

Our quadratic function is then

$$q(x) = y_a \frac{(x-b)(x-c)}{(a-b)(a-c)} + y_b \frac{(x-a)(x-c)}{(b-a)(b-c)} + y_c \frac{(x-a)(x-b)}{(c-a)(c-b)}$$

We can solve for the unique minimum by finding where the derivative is zero:

$$x^* = \frac{1}{2} \frac{y_a(b^2-c^2) + y_b(c^2-a^2) + y_c(a^2-b^2)}{y_a(b-c) + y_b(c-a) + y_c(a-b)}$$

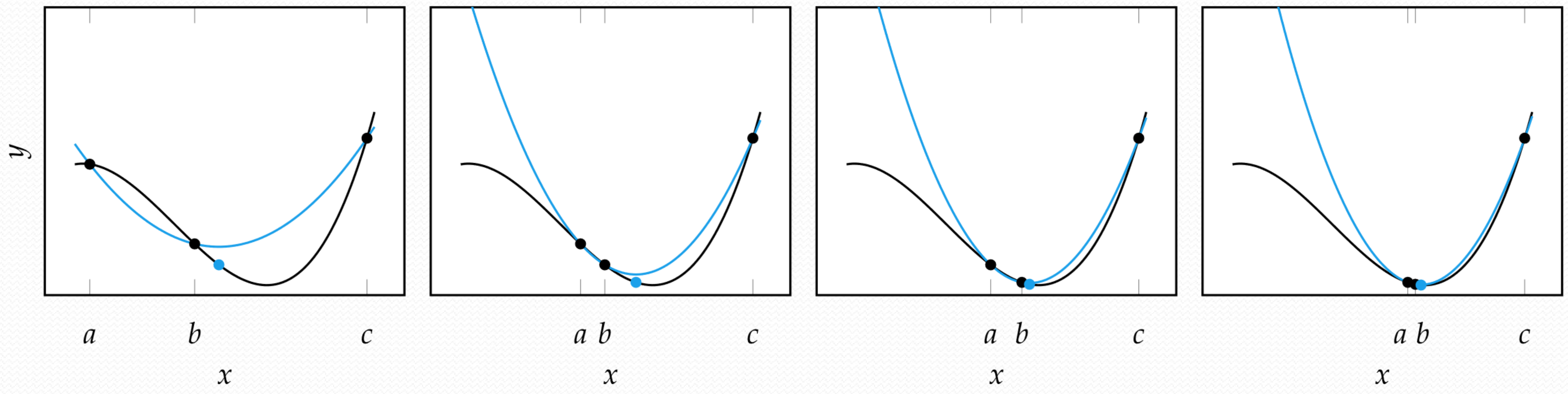
Quadratic Fit Search

- Quadratic fit search is typically faster than golden section search.
- It may need safeguards for cases where the next point is very close to other points.
- A basic implementation is provided in algorithm.

```
function quadratic_fit_search(f, a, b, c, n)
    ya, yb, yc = f(a), f(b), f(c)
    for i in 1:n-3
        x = 0.5*(ya*(b^2-c^2)+yb*(c^2-a^2)+yc*(a^2-b^2)) /
                (ya*(b-c)     +yb*(c-a)     +yc*(a-b))
        yx = f(x)
        if x > b
            if yx > yb
                c, yc = x, yx
            else
                a, ya, b, yb = b, yb, x, yx
            end
        elseif x < b
            if yx > yb
                a, ya = x, yx
            else
                c, yc, b, yb = b, yb, x, yx
            end
        end
    end
    return (a, b, c)
end
```

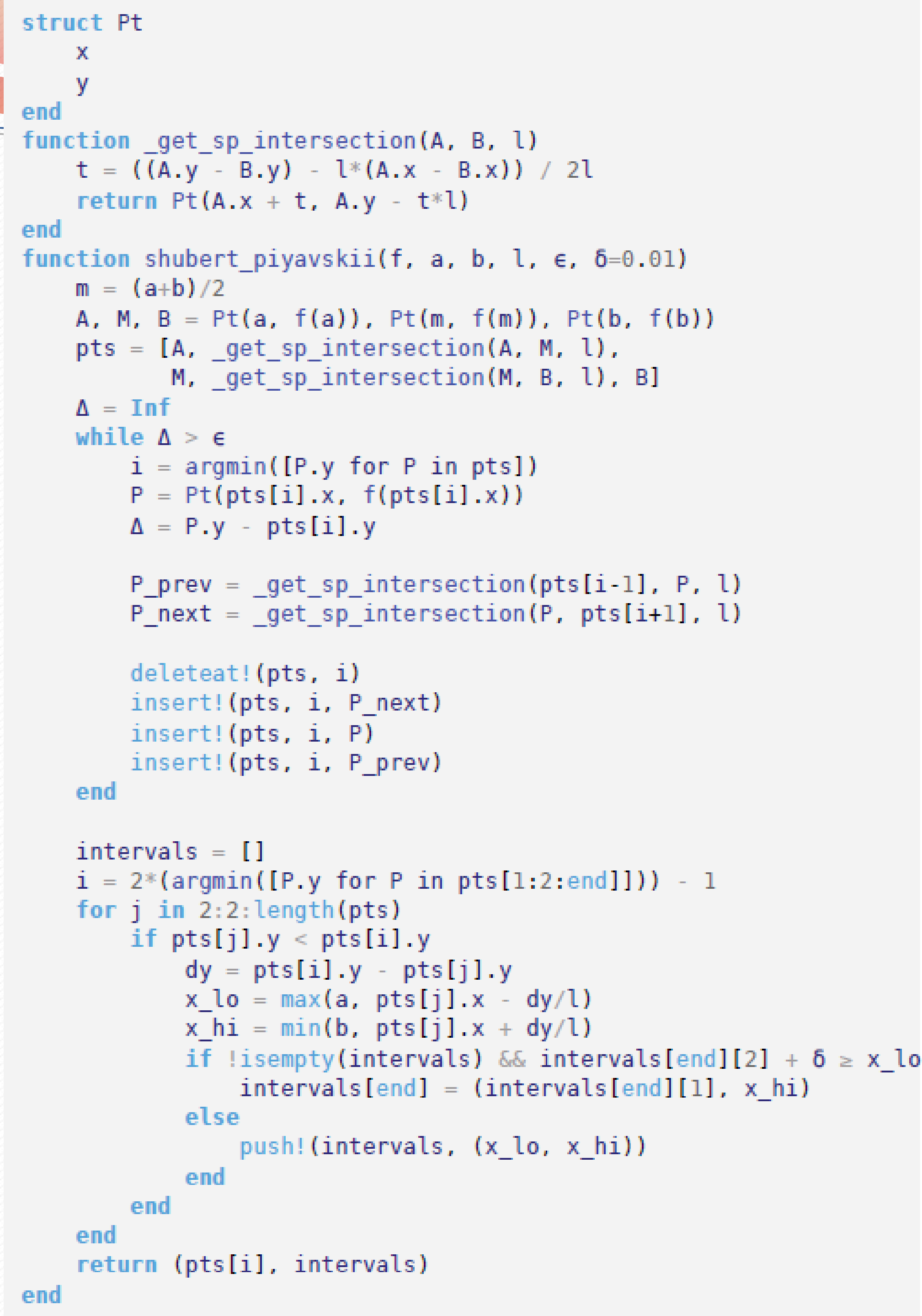
Quadratic Fit Search

- If a function is locally nearly quadratic, the minimum can be found after several steps. Figure shows several iterations of the algorithm.



Shubert-Piyavskii Method

- In contrast with previous methods in this lecture, the *Shubert-Piyavskii method* is a *global optimization method* over a domain [a, b], meaning it is guaranteed to converge on the global minimum of a function irrespective of any local minima or whether the function is unimodal.
- A basic implementation is provided by algorithm.



```
struct Pt
    x
    y
end
function _get_sp_intersection(A, B, l)
    t = ((A.y - B.y) - l*(A.x - B.x)) / 2l
    return Pt(A.x + t, A.y - t*l)
end
function shubert_piyavskii(f, a, b, l, ϵ, δ=0.01)
    m = (a+b)/2
    A, M, B = Pt(a, f(a)), Pt(m, f(m)), Pt(b, f(b))
    pts = [A, _get_sp_intersection(A, M, l),
           M, _get_sp_intersection(M, B, l), B]
    Δ = Inf
    while Δ > ϵ
        i = argmin([P.y for P in pts])
        P = Pt(pts[i].x, f(pts[i].x))
        Δ = P.y - pts[i].y

        P_prev = _get_sp_intersection(pts[i-1], P, l)
        P_next = _get_sp_intersection(P, pts[i+1], l)

        deleteat!(pts, i)
        insert!(pts, i, P_next)
        insert!(pts, i, P)
        insert!(pts, i, P_prev)
    end

    intervals = []
    i = 2*(argmin([P.y for P in pts[1:2:end]])) - 1
    for j in 2:2:length(pts)
        if pts[j].y < pts[i].y
            dy = pts[i].y - pts[j].y
            x_lo = max(a, pts[j].x - dy/l)
            x_hi = min(b, pts[j].x + dy/l)
            if !isempty(intervals) && intervals[end][2] + δ ≥ x_lo
                intervals[end] = (intervals[end][1], x_hi)
            else
                push!(intervals, (x_lo, x_hi))
            end
        end
    end
    return (pts[i], intervals)
end
```

Shubert-Piyavskii Method

- The Shubert-Piyavskii method requires that the function be *Lipschitz continuous*, meaning that it is continuous and there is an upper bound on the magnitude of its derivative. A function f is Lipschitz continuous on [a, b] if there exists an $\ell > 0$ such that:

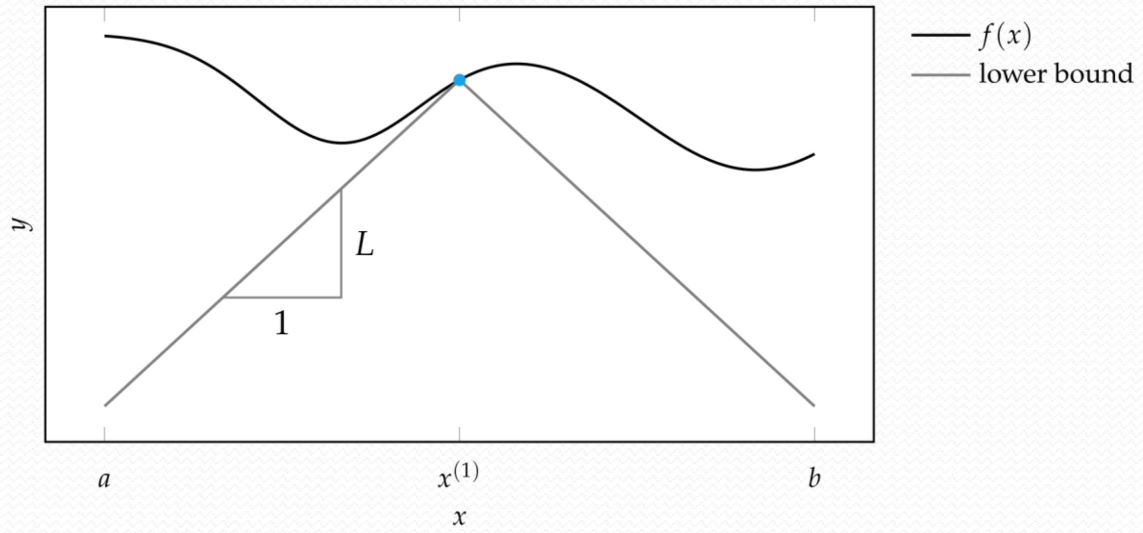
$$|f(x) - f(y)| \leq \ell |x - y| \text{ for all } x, y \in [a, b]$$

- Intuitively, ℓ is as large as the largest unsigned instantaneous rate of change the function attains on [a, b].
- Given a point $(x_0, f(x_0))$, we know that the lines $f(x_0) - \ell(x - x_0)$ for $x > x_0$ and $f(x_0) + \ell(x - x_0)$ for $x < x_0$ form a lower bound of f.

Shubert-Piyavskii Method

- The Shubert-Piyavskii method iteratively builds a tighter and tighter lower bound on the function. Given a valid Lipschitz constant ℓ, the algorithm begins by sampling the midpoint, $x^{(1)} = (a + b)/2$.
- A sawtooth lower bound is constructed using lines of slope $\pm\ell$ from this point. These lines will always lie below f if ℓ is a valid Lipschitz constant as shown in figure.

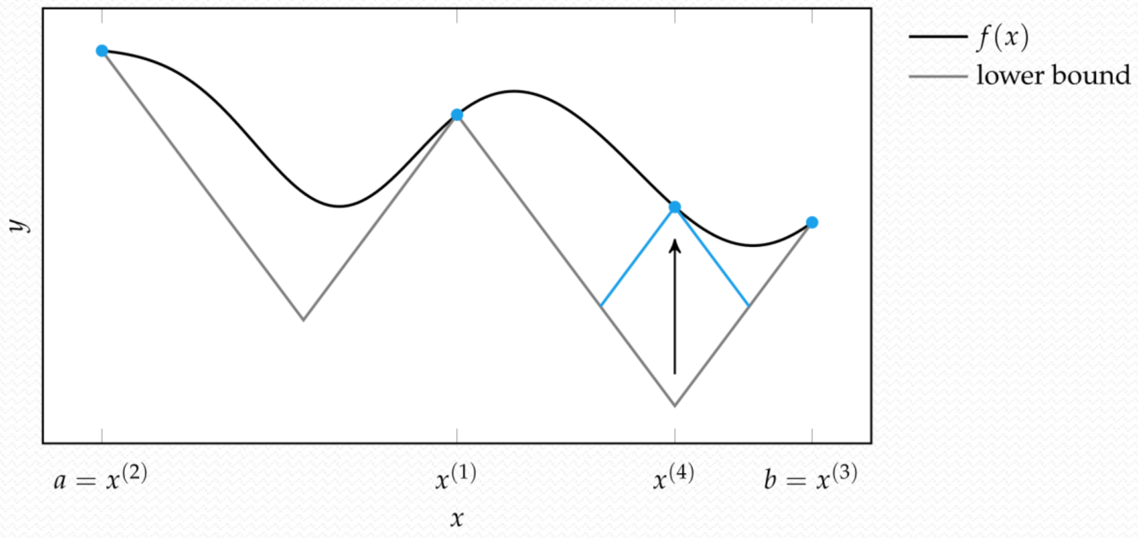
The first iteration of the Shubert-Piyavskii method.



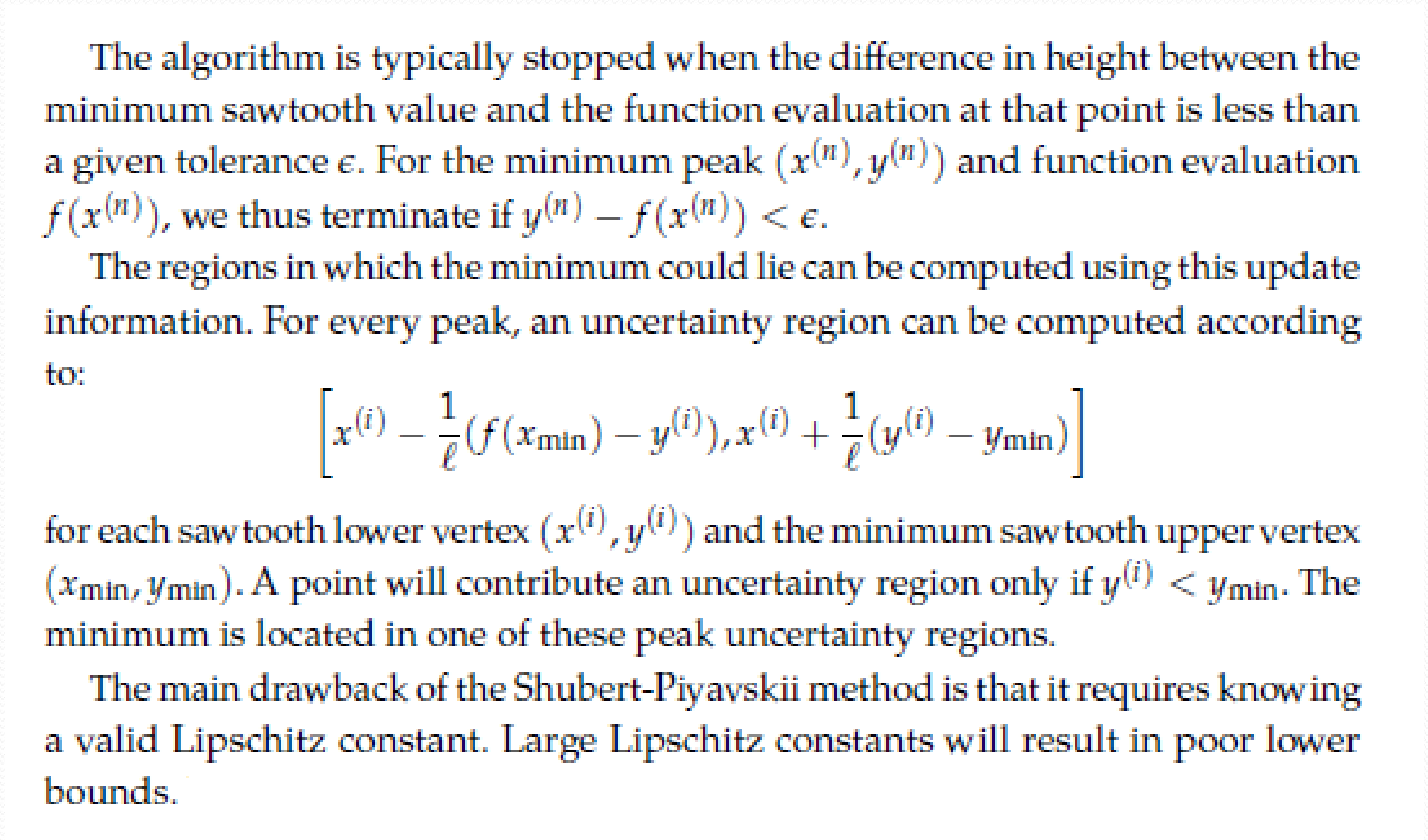
Shubert-Piyavskii Method

- Upper vertices in the sawtooth correspond to sampled points. Lower vertices correspond to intersections between the Lipschitz lines originating from each sampled point. Further iterations find the minimum point in the sawtooth, evaluate the function at that x value, and then use the result to update the sawtooth. Figure illustrates this process.

Updating the lower bound involves sampling a new point and intersecting the new lines with the existing sawtooth.



Shubert-Piyavskii Method



The algorithm is typically stopped when the difference in height between the minimum sawtooth value and the function evaluation at that point is less than a given tolerance ϵ. For the minimum peak $(x^{(n)}, y^{(n)})$ and function evaluation $f(x^{(n)})$, we thus terminate if $y^{(n)} - f(x^{(n)}) < \epsilon$.

The regions in which the minimum could lie can be computed using this update information. For every peak, an uncertainty region can be computed according to:

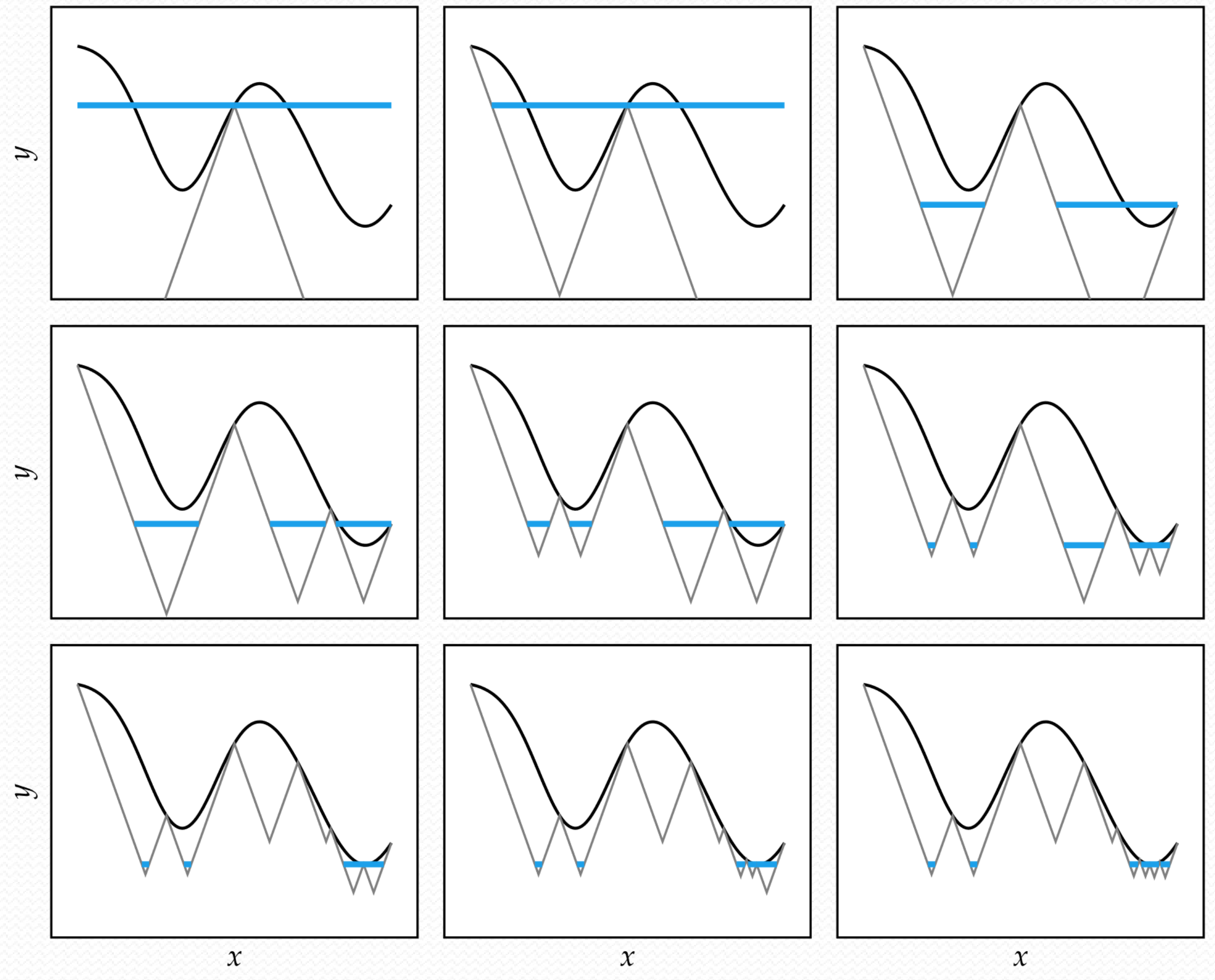
$$\left[x^{(i)} - \frac{1}{\ell}(f(x_{\min}) - y^{(i)}), x^{(i)} + \frac{1}{\ell}(y^{(i)} - y_{\min})\right]$$

for each sawtooth lower vertex $(x^{(i)}, y^{(i)})$ and the minimum sawtooth upper vertex $(x_{\min}, y_{\min})$. A point will contribute an uncertainty region only if $y^{(i)} < y_{\min}$. The minimum is located in one of these peak uncertainty regions.

The main drawback of the Shubert-Piyavskii method is that it requires knowing a valid Lipschitz constant. Large Lipschitz constants will result in poor lower bounds.

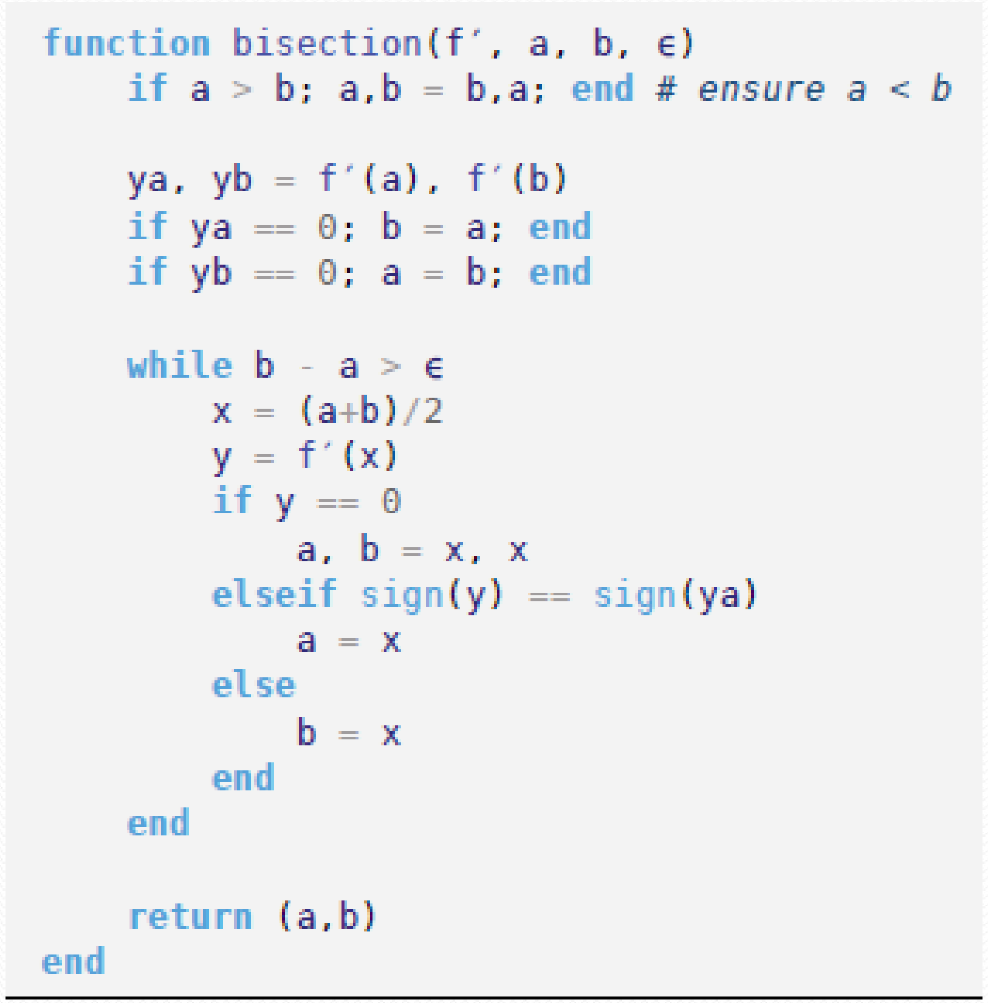
Shubert-Piyavskii Method

- Figure shows nine iterations of the Shubert-Piyavskii method proceeding left to right and top to bottom.
- The blue lines are uncertainty regions in which the global minimum could lie.



Bisection Method

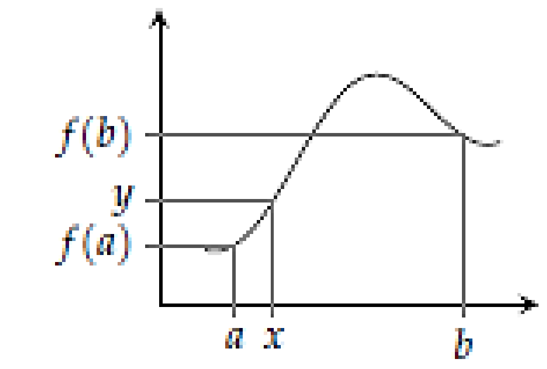
- The *bisection method* (algorithm) can be used to find *roots* of a function, or points where the function is zero.
- Such *root-finding methods* can be used for optimization by applying them to the derivative of the objective, locating where f ′(x) = 0.
- In general, we must ensure that the resulting points are indeed local minima.



```
function bisection(f′, a, b, ϵ)
    if a > b; a,b = b,a; end # ensure a < b

    ya, yb = f′(a), f′(b)
    if ya == 0; b = a; end
    if yb == 0; a = b; end

    while b - a > ϵ
        x = (a+b)/2
        y = f′(x)
        if y == 0
            a, b = x, x
        elseif sign(y) == sign(ya)
            a = x
        else
            b = x
        end
    end

    return (a,b)
end
```

Bisection Method

- The bisection method maintains a bracket [a, b] in which at least one root is known to exist.
- If f is continuous on [a, b], and there is some y ∈ [f (a), f (b)], then the *intermediate value theorem* stipulates that there exists at least one x ∈ [a, b], such that f (x) = y as shown in figure.
- It follows that a bracket [a, b] is guaranteed to contain a zero if f (a) and f (b) have opposite signs.



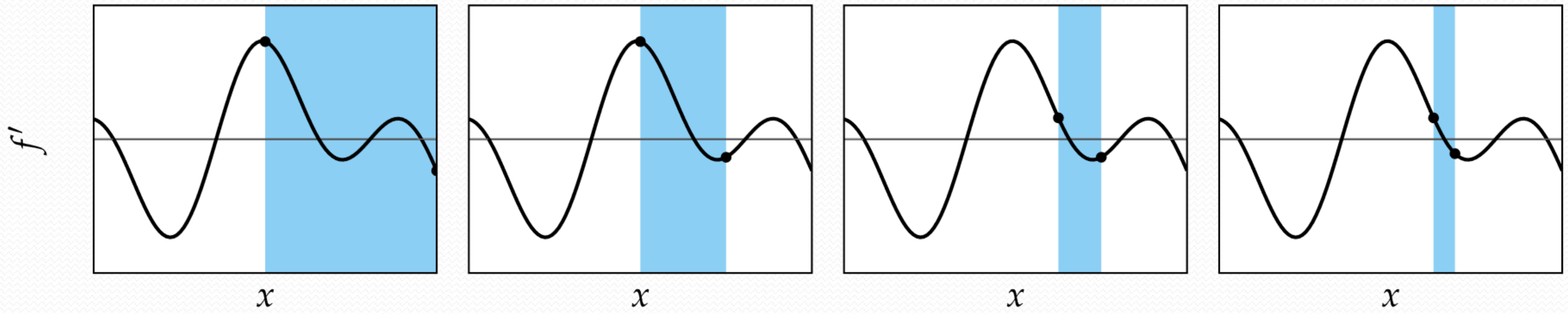
A horizontal line drawn from any y ∈ [f (a), f (b)] must intersect the graph at least once.

Bisection Method

- The bisection method cuts the bracketed region in half with every iteration.
- The midpoint (a + b)/2 is evaluated, and the new bracket is formed from the midpoint and whichever side that continues to bracket a zero.
- We can terminate immediately if the midpoint evaluates to zero.
- Otherwise we can terminate after a fixed number of iterations.
- This method is guaranteed to converge within ϵ of x* within $\lg\left(\frac{|b-a|}{\epsilon}\right)$ iterations, where lg denotes the base 2 logarithm.

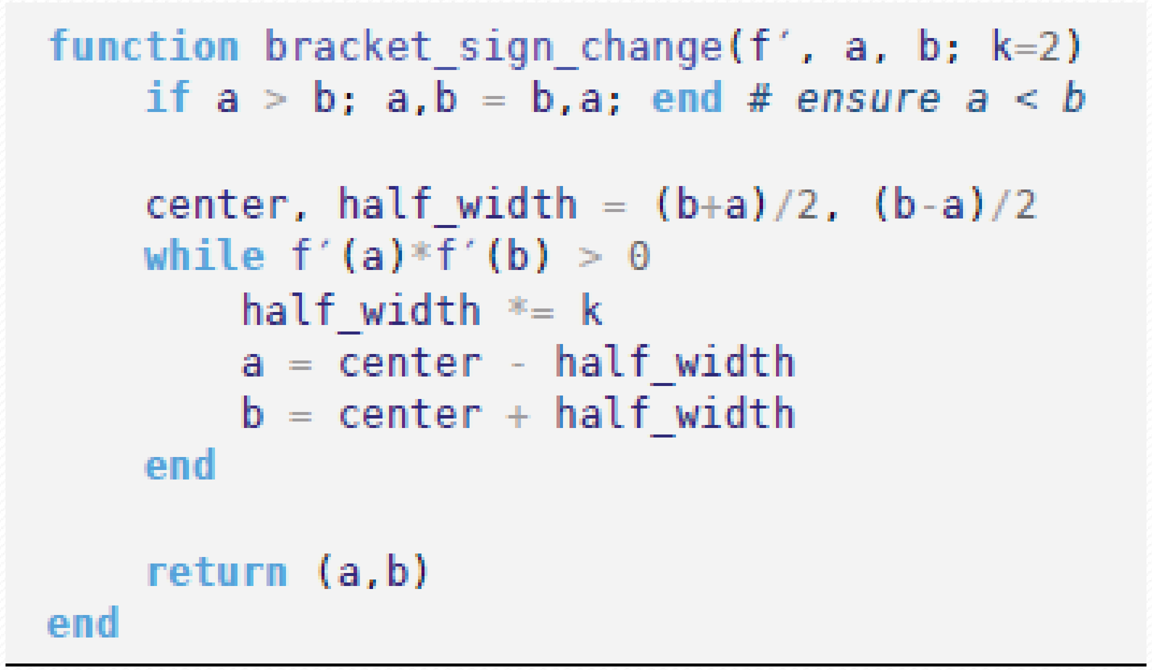
Bisection Method

- Figure shows four iterations of the bisection method. The horizontal line corresponds to f ′(x) = 0. Note that multiple roots exist within the initial bracket.



Bisection Method

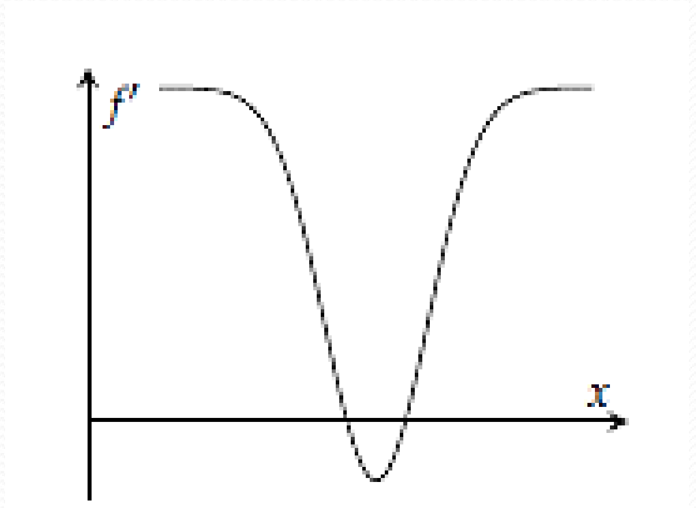
- Root-finding algorithms like the bisection method require starting intervals [a, b] on opposite sides of a zero. That is, sign(f ′(a)) ≠ sign(f ′(b)), or equivalently, f ′(a) f ′(b) ≤ 0.
- Algorithm provides a method for automatically determining such an interval.
- It starts with a guess interval [a, b]. So long as the interval is invalid, its width is increased by a constant factor.



```
function bracket_sign_change(f′, a, b; k=2)
    if a > b; a,b = b,a; end # ensure a < b

    center, half_width = (b+a)/2, (b-a)/2
    while f′(a)*f′(b) > 0
        half_width *= k
        a = center - half_width
        b = center + half_width
    end

    return (a,b)
end
```

Bisection Method

- Doubling the interval size is a common choice. This method will not always succeed as shown in figure.
- Functions that have two nearby roots can be missed, causing the interval to infinitely increase without termination.



A bracketing method initialized such that it straddles the two roots in this figure will expand forever, never to find a sign change. Also, if the initial interval is between the two roots, doubling the interval can cause both ends of the interval to simultaneously pass the two roots.

Summary

- Many optimization methods shrink a bracketing interval, including Fibonacci search, golden section search, and quadratic fit search
- The Shubert-Piyavskii method outputs a set of bracketed intervals containing the global minima, given the Lipschitz constant
- Root-finding methods like the bisection method can be used to find where the derivative of a function is zero