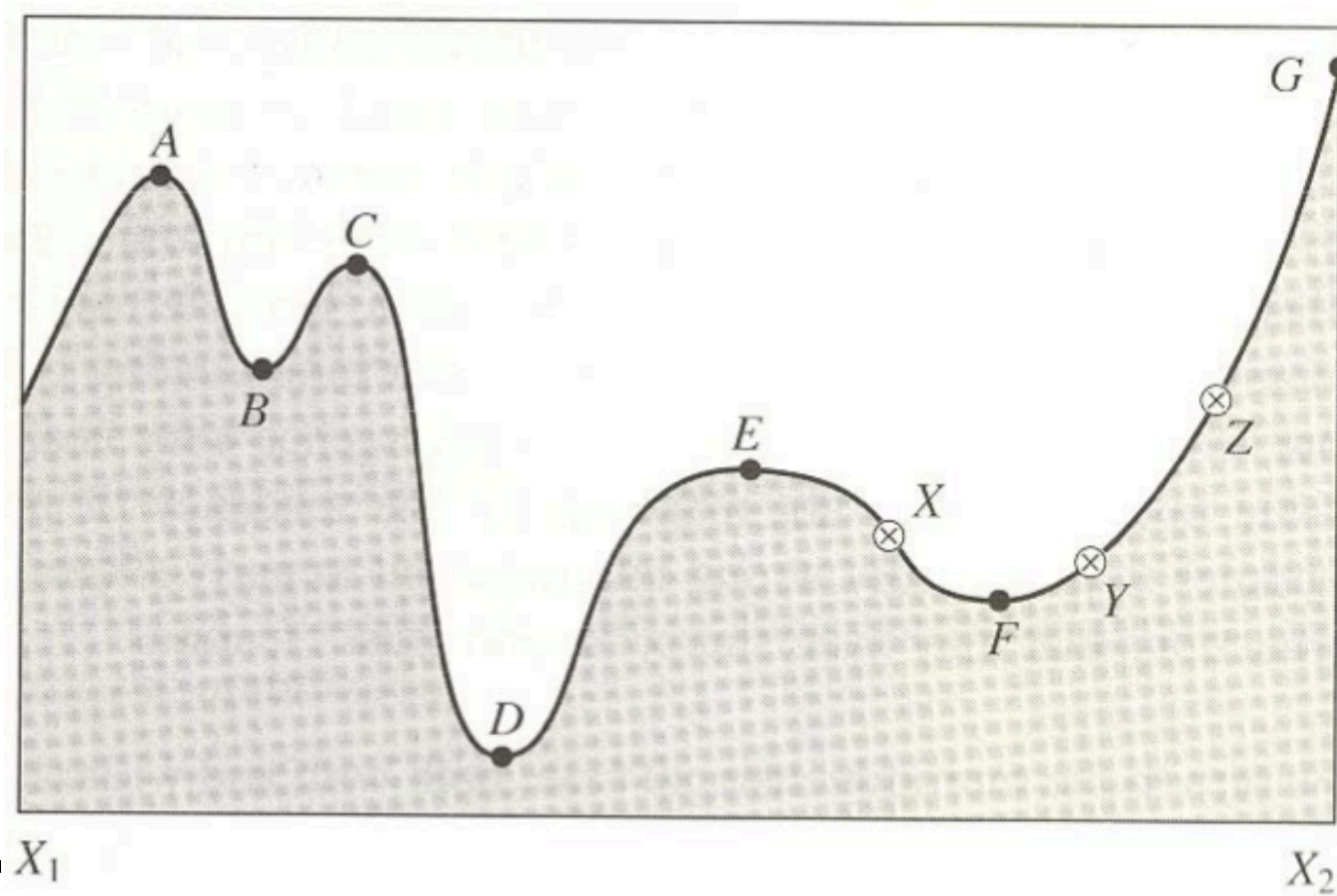
HMKANG@UMICH.EDU

MODULE 2.3

SINGLE-DIMENSIONAL OPTIMIZATION

The minimization problem



Specific objectives

- **Finding global minimum**
  - Obtain the lowest possible value of the function
  - A very hard problem to solve generally.

- **Finding local minimum**
  - Smallest value with finite neighborhoold
  - Relative easier problem to solve

A detour – local root finding problem

- **Consider the problem of finding zeros for $f(x)$**

- **Assume that you know**
  - the point a where $f(a)$ is positive
  - the point b where $f(b)$ is positive
  - that $f(x)$ is continuous between a and b.

- **How would you proceed to find x such that $f(x) = 0$?**

Bisection Method

```python
## func : an arbitrary function
## lo   : x value where func(x) is negative
## hi   : x value where func(x) is positive
## e    : tolerance of errors in x
def binaryRoot(func, lo, hi, e):
    itr = 0       ## number of iteration
    while(True): ## loop until returns
        d = hi-lo              ## range of x
        point = (hi+lo)/2.0  ## midpoint in x
        fpoint = func(point) ## midpoint in y
        if ( fpoint < 0 ):   ## if y is negative..
            d = lo-point
            lo = point         ## set it as new lo
        else:
            d = point-hi
            hi = point  ## if not, set it as new hi
        ## return if range is small enough, or y is exactly zero
        if ( ( abs(d) < e ) or ( fpoint == 0 ) ):
            print("iteration:\t" + str(itr))
            print("point:\t" + str(point))
            print("fpoint:\t" + str(fpoint))
            print("d:\t" + str(d))
            return point;
        itr += 1
```

See at bios615_2_3.ipynb

Evaluating the performance

```python
import math
binaryRoot(math.sin, 0-math.pi/math.sqrt(2), math.pi/2, 1e-5)
```

```
iteration:       18
point:   1.44799433991e-06
fpoint:  1.4479943399e-06
d:       -7.23311957526e-06
```

Improving the bisection method

- **Approximation using linear interpolation**

$$f^*(x) = f(a) + (x - a)\frac{f(b) - f(a)}{b - a}$$

- **Root finding strategy**
  - Select a new trial point such that $f^*(x) = 0$

False Position Method

```
def linearRoot(func, lo, hi, e):
    itr = 0          ## number of iteration
    flo = func(lo) ## need y values at the bounaries
    fhi = func(hi)
    while(True): ## loop until returns
        d = hi-lo                  ## range of x
        point = lo + d * flo / (flo - fhi)
        fpoint = func(point) ## midpoint in y
        if ( fpoint < 0 ):    ## if y is negative..
            d = lo-point
            lo = point             ## set it as new lo
            flo = fpoint           ## replace the bounary value, too
        else:
            d = point-hi
            hi = point    ## if not, set it as new hi
            fhi = fpoint ## replace the bounary value, too
        ## return if range is small enough, or y is exactly zero
        if ( ( abs(d) < e ) or ( fpoint == 0 ) ):
            print("iteration:\t" + str(itr))
            print("point:\t" + str(point))
            print("fpoint:\t" + str(fpoint))
            print("d:\t" + str(d))
            return point;
        itr += 1
```

See at bios615_2_3.ipynb

Comparing the performance

```
import math
binaryRoot(math.sin, 0-math.pi/math.sqrt(2), math.pi/2, 1e-5)
```

```
iteration:      18
point:   1.44799433991e-06
fpoint:  1.4479943399e-06
d:       -7.23311957526e-06
```

```
linearRoot(math.sin, 0-math.pi/math.sqrt(2), math.pi/2, 1e-5)
```

```
iteration:      6
point:   2.58493941423e-26
fpoint:  2.58493941423e-26
d:       -2.63766959334e-20
```

Comparing with R's `uniroot()`

```
> uniroot(sin, c(0-base::pi/sqrt(2),base::pi/2), tol=1e-5)
$root
[1] -1.376062e-11

$f.root
[1] -1.376062e-11

$iter
[1] 6

$init.it
[1] NA

$estim.prec
[1] 5e-06
```

Summary - local root finding

- **Two simple methods**
  - Bisection method : `binaryRoot()`
  - False position method : `linearRoot()`

- **In the bisection method, the bracketing interval is halved at each step**

- **For well-behaved function, the false position method will converge faster, but there is no performance guarantee.**

Back to the minimization problem

- **Consider a complex function *f(x)* (e.g likelihood)**
- **Goal is to find *x* which *f(x)* is maximum or minimum value**
- **Maximization and minimization are equivalent**
  - Replace *f(x)* with *−f(x)*

Notes from root finding

- **Two approaches possibly applicable to minimization problems:**

- **Bracketing**
  - Keep track of intervals containing solution

- **Accuracy**
  - Recognize that the solution has limited precision

Considering the machine precision

- When estimating the minima and bracketing intervals, floating-point accuracy must be considered
- In general, if the machine precision is ϵ, the achievable accuracy is no more than $\sqrt{\epsilon}$
- In **`double`** precision floating-point number, it is ~8 decimal digits, since **`double`** has ~16 decimal digits of precision

More details on machine precision

- $\sqrt{\epsilon}$ comes from the second-order Taylor approximation around a local minima

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2$$

- For functions where higher order terms are important, accuracy could be even lower
  - For example, the minimum for $f(x) = 1 + x^4$ is only estimated to about $\epsilon^{\frac{1}{4}}$

Steps for minimization strategy

1. Bracket minimum
    - Search for regions that contains local minima

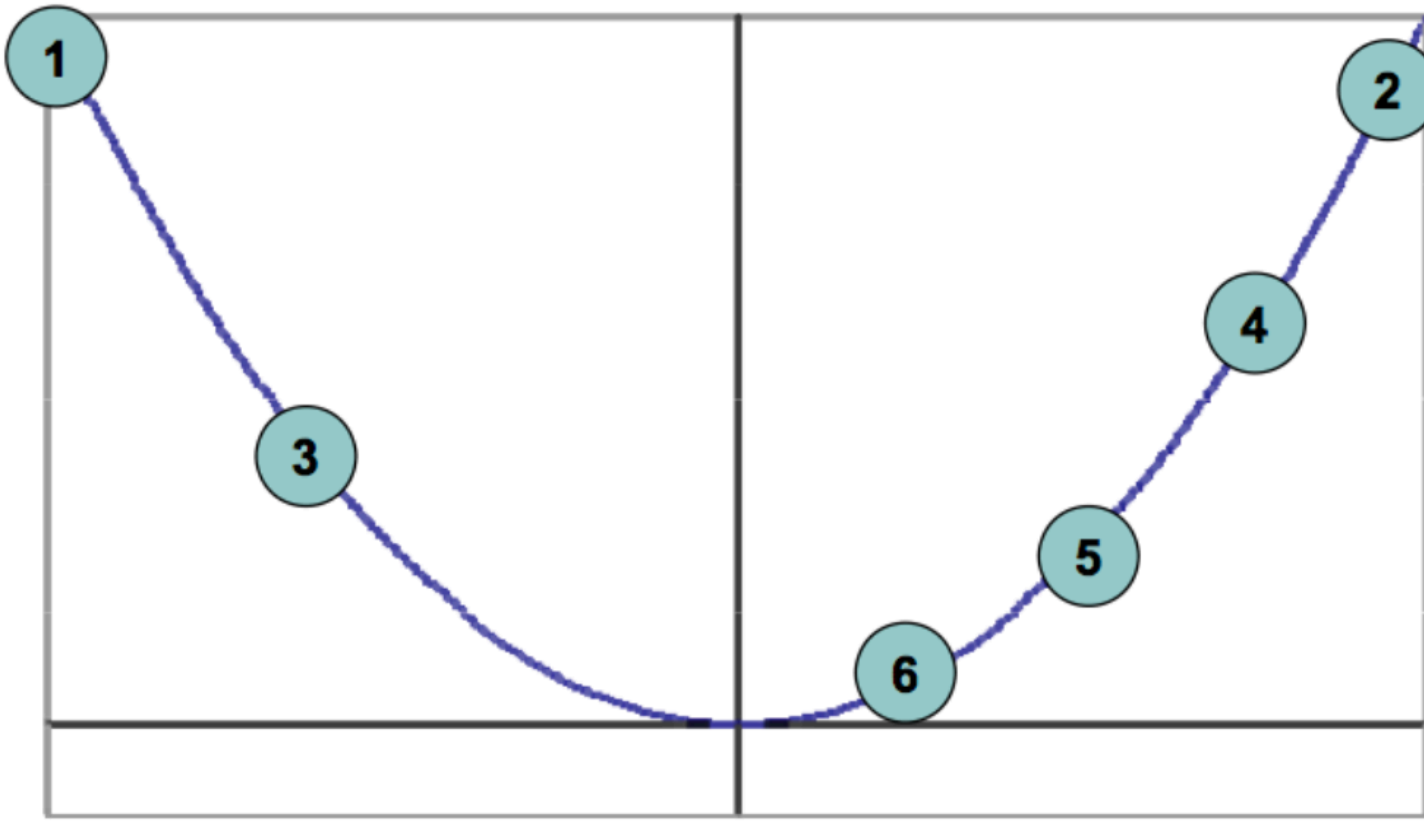
2. Successively tighten bracket interval
    - .. using local minimization

Detailed minimization strategy

- **Find 3 points such that**
  - $a < b < c$
  - $f(b) < f(a)$ and $f(b) < f(c)$

- **Then search for minimum by**
  - Selecting trial points in the interval
  - Keep minimum values and flanking points

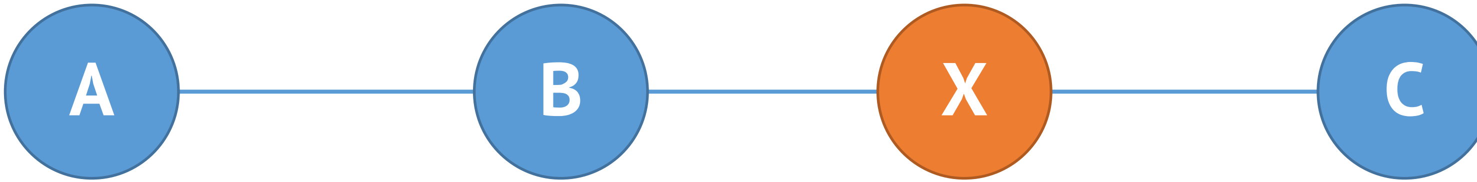
Minimization after bracketing



What is the best location of the new point X?

- Here, we cannot use a bisection method..

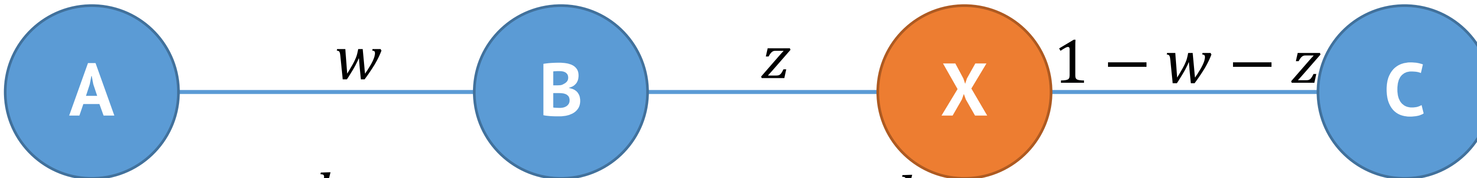
What we **want** is..



- **We want to minimize the size of the next interval which will be either (A, X) or (B, C)**
  - If $f(X) < f(B)$, the next search interval will be (B, C)
  - If $f(B) < f(X)$, the next search interval will be (A, X)

- **What would be a principled strategy to select X?**

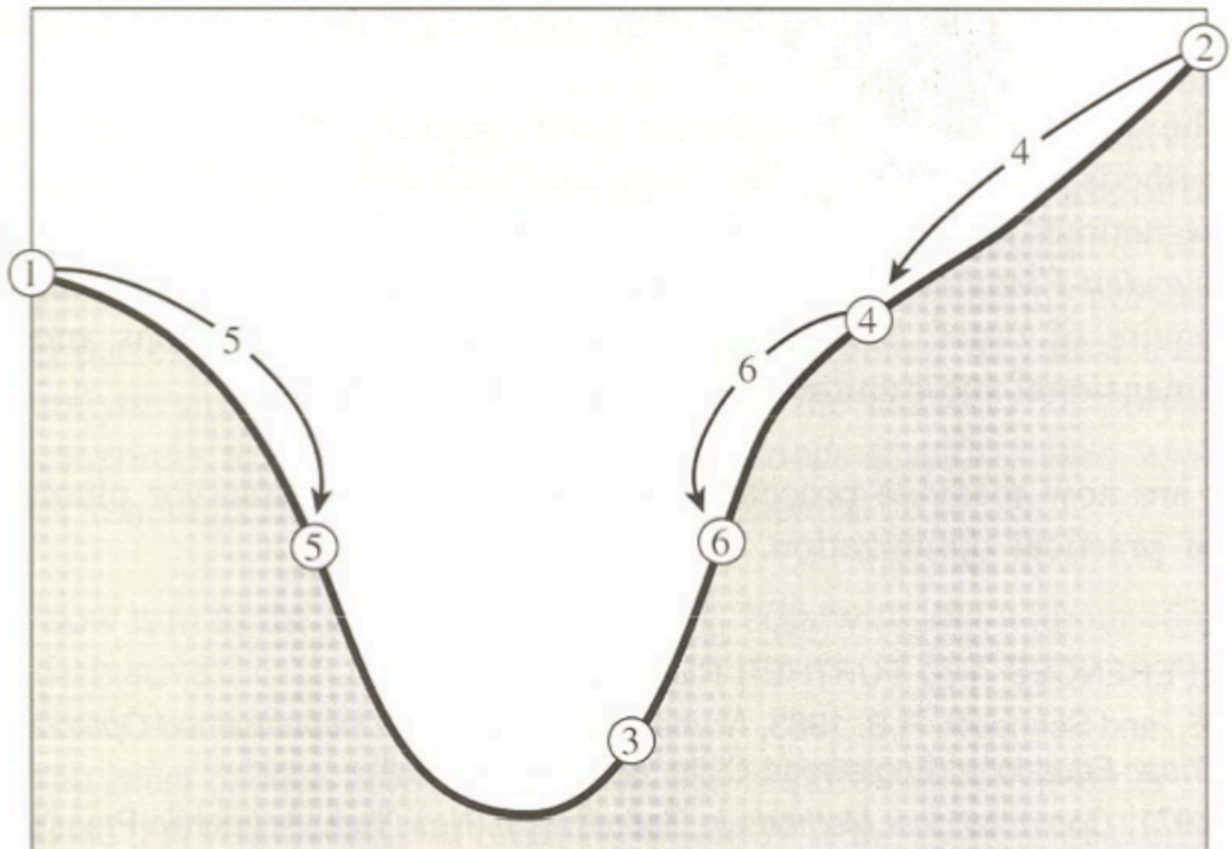
Minimizing the worst-case damage



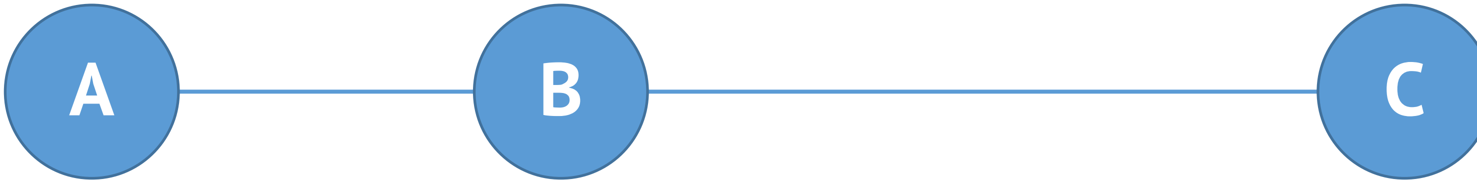
$$w = \frac{b-a}{c-a} \qquad z = \frac{x-b}{c-a}$$

- **The next interval will have length either $1-w$ or $w+z$**
- **Optimal conditions are** $\begin{cases} 1-w = w+z \\ \dfrac{z}{1-w} = w \end{cases}$
- **Solving the equation leads to** $w = \dfrac{3-\sqrt{5}}{2} = 0.38197$

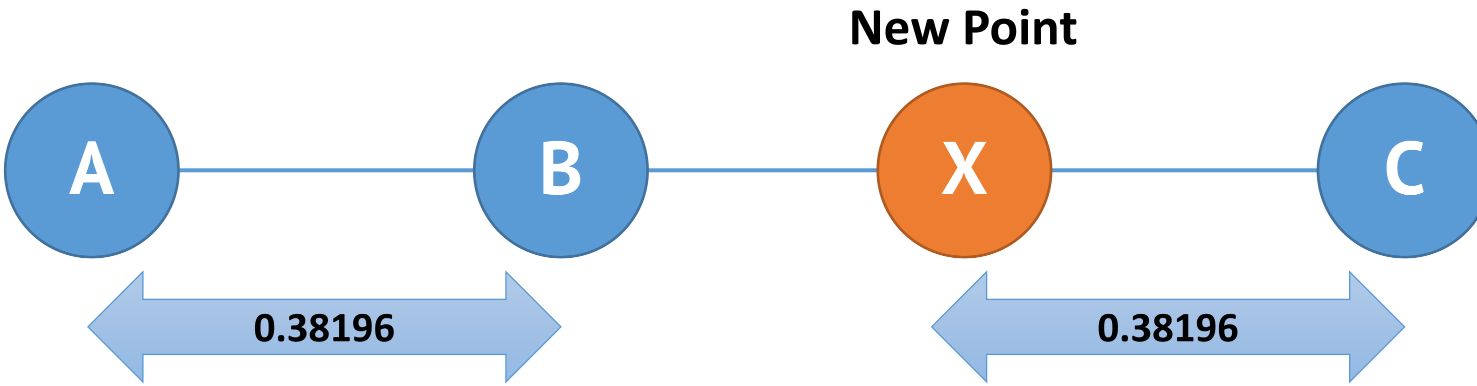
The golden search



The initial bracketing triplet

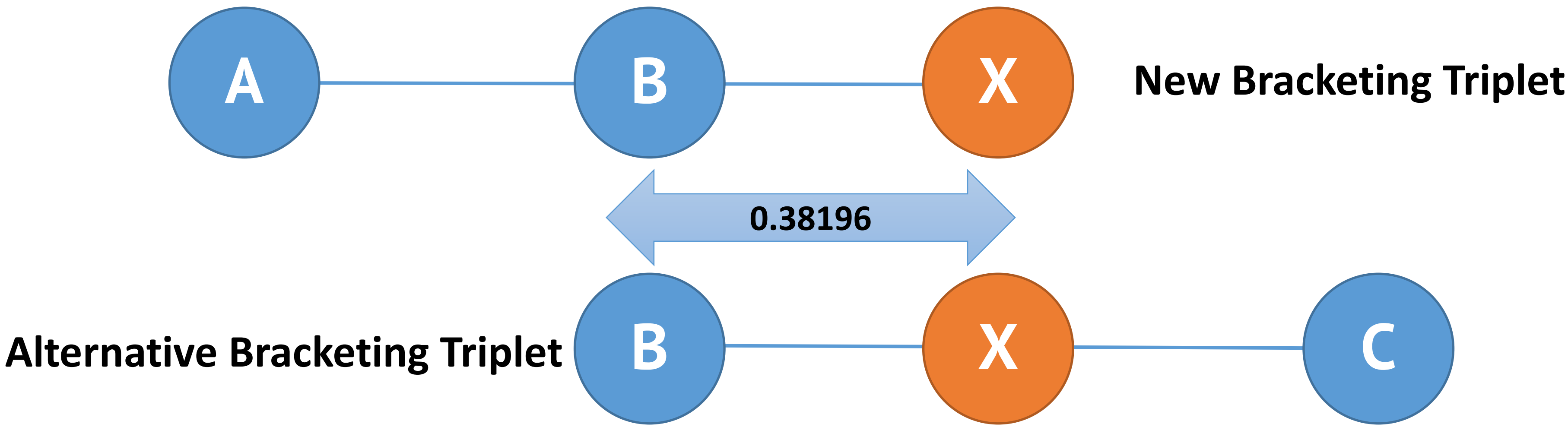


The golden ratio



The ratio between the new and old bracket size is
1 : 1.618, which is known as golden ratio

The golden ratio



The ratio between the new and old bracket size is
1 : 1.618, which is known as golden ratio

Summay - golden search

- Reduces bracketing by ~40%

- Performance is independent of the function that is being minimized

- In many cases, better schemes are available.

Implementation of golden search

```
## goldenSearch() for general minimiztion
## func : an arbitrary function
## a : smallest value in bracket triplet
## b : middle value in bracket triplet
## c : largest value in bracket triplet
## e : relative precision tolerance
def goldenSearch(func, a, b, c, e):
    itr = 0
    fb = func(b)
    ## when b is very small, need a higher precision
    while( abs(c-a) > abs(b*e) ):
        if ( b > (a+c)/2 ): ## as illustrated in the lecture
            x = a + 0.38196 * (c-a)
        else:                ## x is on the left side
            x = c - 0.38196 * (c-a)
        fx = func(x)     ## evaulate function
```

```
        if ( fx < fb ): ## if found a new minimum
            if (x > b):
                a = b
            else:
                c = b
            b = x
            fb = fx
        else:
            if ( x < b ):
                a = x
            else:
                c = x;
        itr += 1
    print("itr:\t" + str(itr))
    print("x:\t" + str(b))
    print("y:\t" + str(fb))
    print("d:\t" + str(c-a))
    return b;
```

A running example

```
def foo(x):
    return 0-math.cos(x)

goldenSearch(foo, 0-math.pi/math.sqrt(2),math.pi/4,math.pi/2,1e-5)
```

```
itr:     69
x:       -3.57189823006e-09
y:       -1.0
d:       2.32857231782e-14
```

Using existing function in R

Global variable for tracking the # of function calls

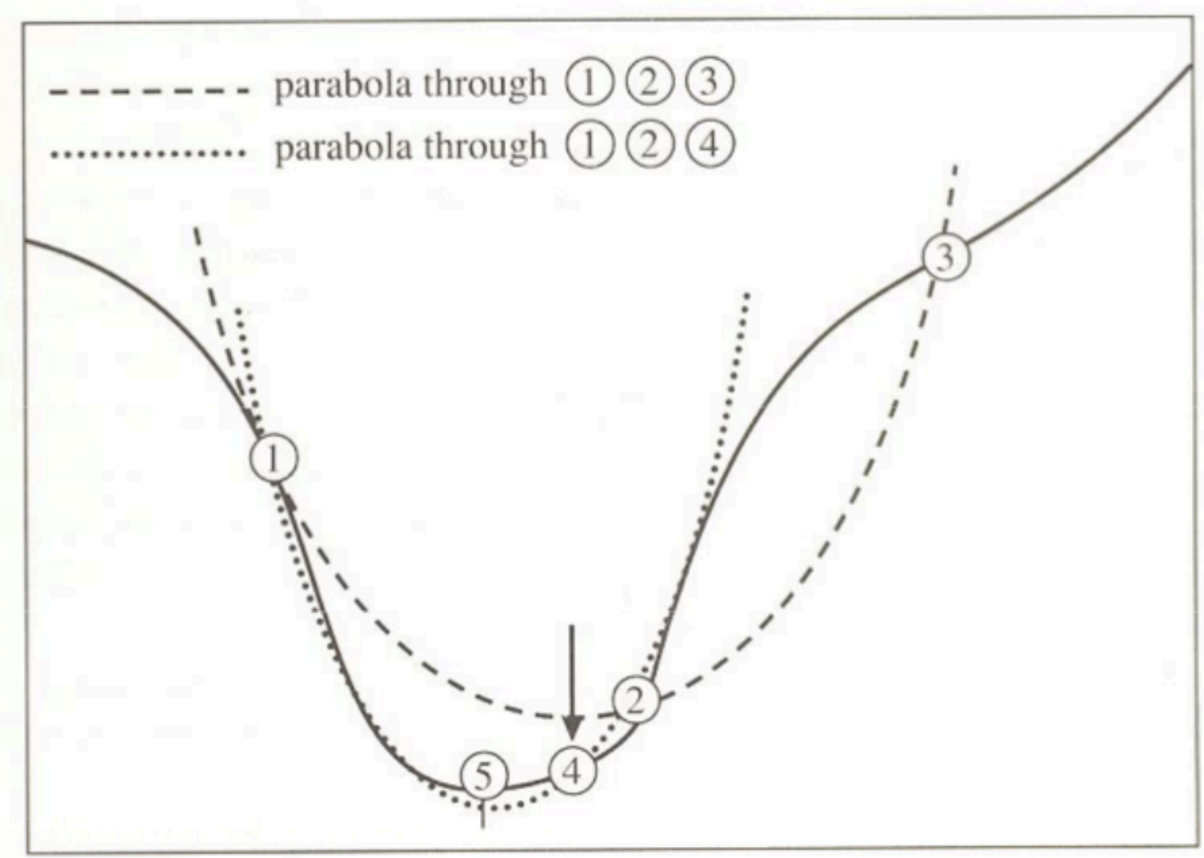
```
> itr <<- 0
> foo <- function(x) { itr <<- itr +1 ; return (-cos(x)) }
> optimize(foo,c(-base::pi/sqrt(2),base::pi/2),tol=1e-5)
$minimum
[1] -3.995917e-07

$objective
[1] -1

> itr
[1] 9
```

Can we improve the golden search?

- As with root finding, performance can improve substantially when interpolation methods are used

- However, a linear approximation won’t work in this case, why?

Approximation using parabola



Parabolic interpolation

- Often converges faster than other algorithms
- However, there is no guarantee that this interpolation always works for an arbitrary function.
- Because golden search provides worst-case performance guarantee, it can be used as a fall-back for uncooperative functions.

State-of-the-art : Brent's algorithm

- **Track 6 points (not all distinct)**
  - The bracket boundaries (a,b)
  - The current minimum x
  - The second and third smallest value (w, v)
  - The new points to be examined u
- **Use parabolic interpretation**
  - Using (x, w, v) to propose new value for u,
  - Additional case is required to ensure u falls between a and b
- **Implemented as in R functions**
  - `uniroot` for 1-dimensional root finding
  - `optimize` for 1-dimensional optimization

Newton-Raphson method (1669)

- **Key idea**
  - Assume that the derivative function is available.
  - Use the derivative to find the next point with a linear interpolation
- **For root-finding problem, use the first derivative of *f(x)***

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

- **For singled-dimensional optimization, use the second derivative**

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

Properties of the Newton-Raphson method

- **Pros**
  - Easy to implement
  - Works for high-dimensional cases, too
  - Quadradic convergence

- **Cons**
  - Requires derivatives
  - Convergence is not guaranteed
  - Requires a big Jacobian or Hessian matrix for high-dimensional problems.
    - Quasi-newton method can be a fix

Examples of the Newton-Raphson method

- **Finding the reciprocal without division**

$$f(x) = a - \frac{1}{x} \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{a - \frac{1}{x_n}}{\frac{1}{x_n^2}} = x_n(2 - ax_n).$$

- **Finding the square root with basic arithmetic operations**

$$f(x) = x^2 - a \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n}{2} + \frac{a}{2x_n}.$$

Summary

- **Root finding algorithms**
  - Bisection Method : Simple but likely less efficient
  - False Position Method : More efficient for well-behaved functions
  -
- **Single-dimensional minimization**
  - Golden Search: ~38% reduction of interval per iteration
  - Parabola Method: More efficient for well-behaved functions
  - Brent's Method: Combination of above two. State-of-the-art
  - Newton-Raphson Method: Quadratic convergence w/derivatives.