Root Finding and Optimization

Ramses van Zon

SciNet, University of Toronto

Scientific Computing Lecture 11
February 11, 2014

Root Finding

- It is not uncommon in scientific computing to want solve an equation numerically.
- If there is one unknown and one equation only, we can always write the equation as

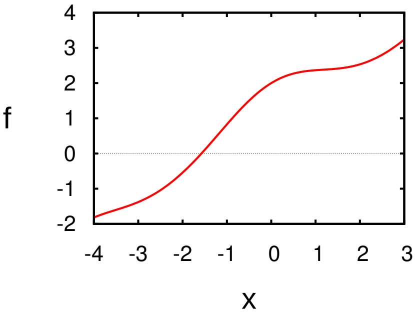
$$\mathbf{f(x) = 0}$$

- If $\mathbf{x}$ satisfies this equation, it is called a "root".
- If there's a set of equations, one can write:

$$\begin{aligned}
\mathbf{f_1(x_1, x_2, x_3, \dots)} &= \mathbf{0} \\
\mathbf{f_2(x_1, x_2, x_3, \dots)} &= \mathbf{0} \\
\mathbf{f_3(x_1, x_2, x_3, \dots)} &= \mathbf{0} \\
&\dots
\end{aligned}$$

The one-dimensional case is considerably easier to solve: First.

1D root finding



Algorithms always start from an initial guess and (usually) a bounding interval $\mathbf{[a,b]}$ in which the root is to be found.

What's so nice about 1D?

- If $\mathbf{f(a)}$ and $\mathbf{f(b)}$ have opposite signs, and $\mathbf{f}$ is continuous, there must be a root inside the interval: the root is *bracketed*.
- Consecutive refinement of the interval until $\mathbf{a}-\mathbf{b}<\varepsilon$ guarranteed to find the root.

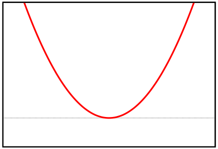
Bracketing

Must find bounding interval first.

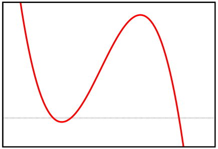
Strategies:

- *Plot* the function.
- Make a conservative guess for $[\mathbf{a}, \mathbf{b}]$ then *slice up* the interval, checking for sign change of $\mathbf{f}$.
- Make a wild guess and *expand* the interval until $\mathbf{f}$ exhibits a sign change.

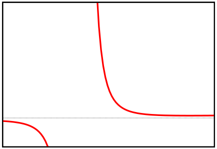
Troublemakers



No sign change



Easily missed



Singularity

Suppose we've bracketed a root, what's next?

Classic root finding algorithms

- Bisection
- Secant/False Position
- Ridders'/Brent
- Newton-Raphson (requires derivatives)

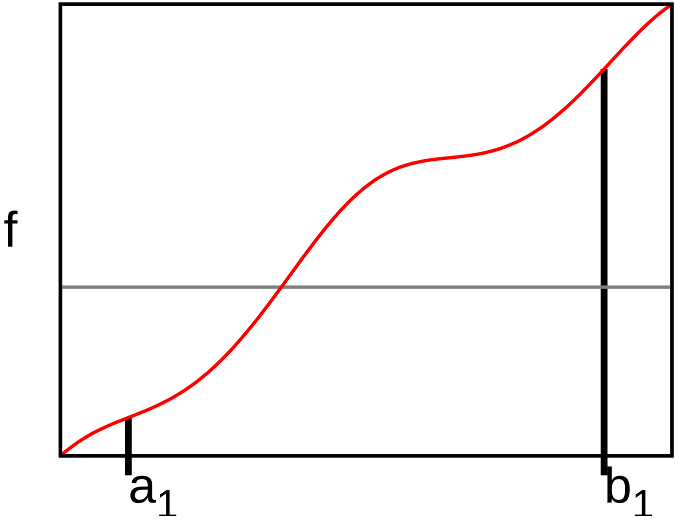
All of these zero in on the root, but have different convergence and stability characteristics.

Note:

- For polynomial $\mathbf{f}$: specialized routines (Muller, Laguerre)
- For eigenvalue problems: specialized routines (linear algebra)

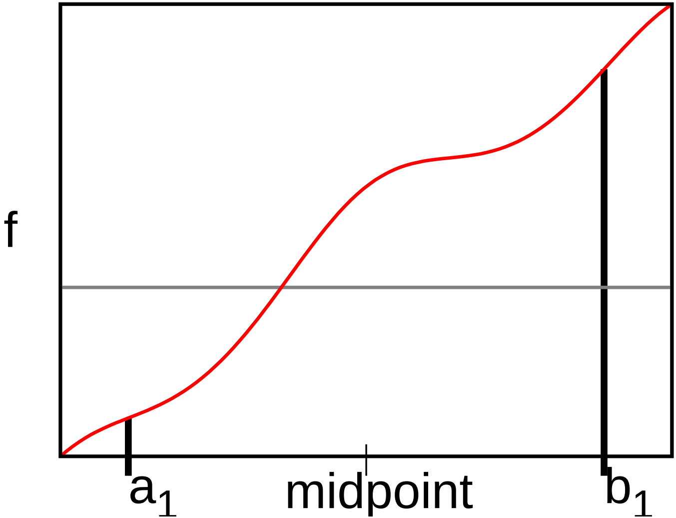
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



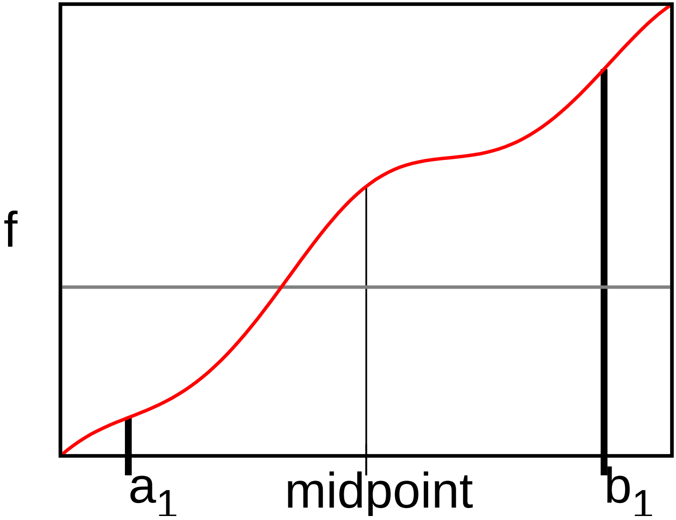
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



Bisection method

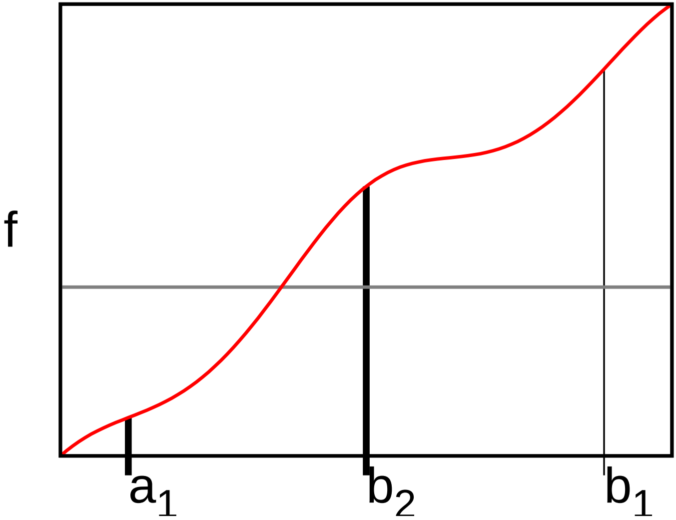
Split the interval. Compute **f(midpoint)**. Get new root bracket.



SciNet compute • calcul CANADA

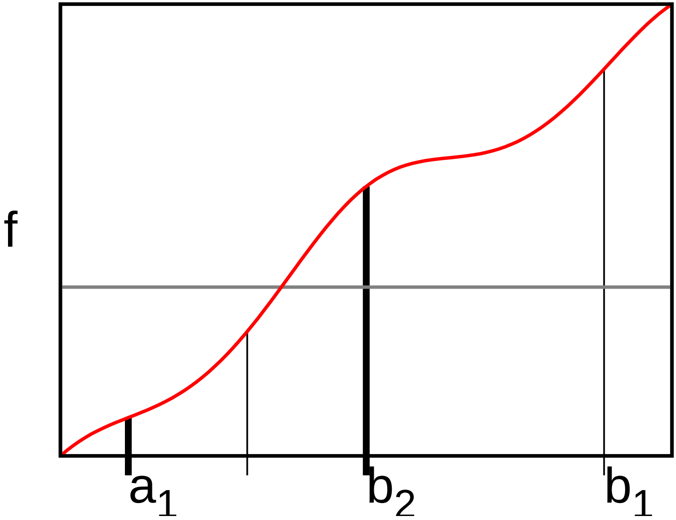
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



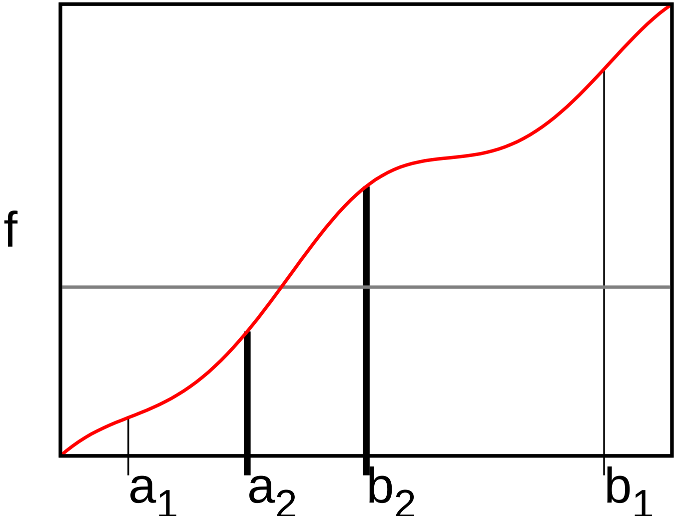
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



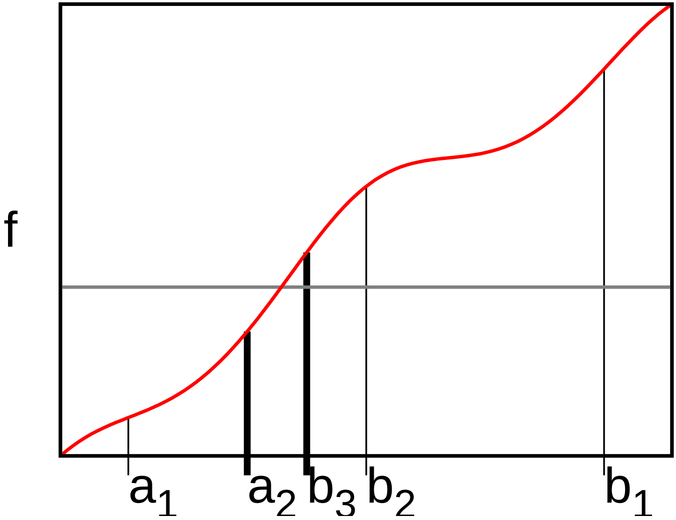
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



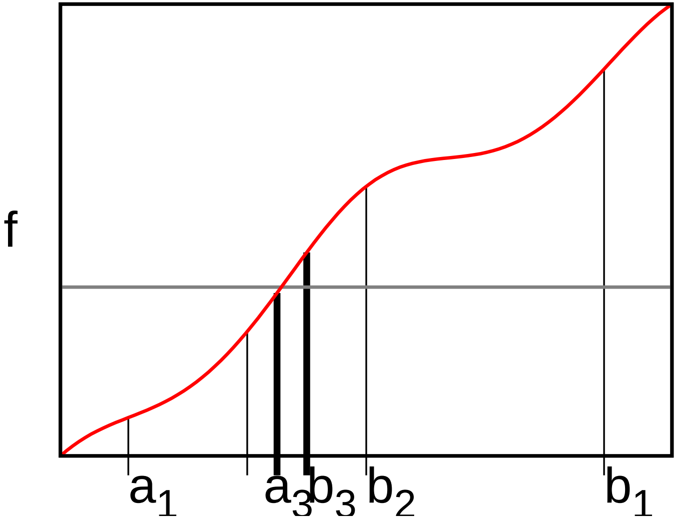
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



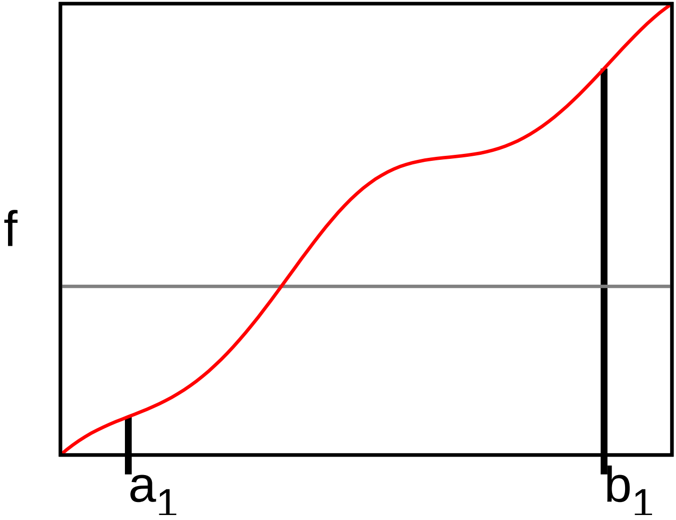
Bisection method

Split the interval. Compute **f(midpoint)**. Get new root bracket.



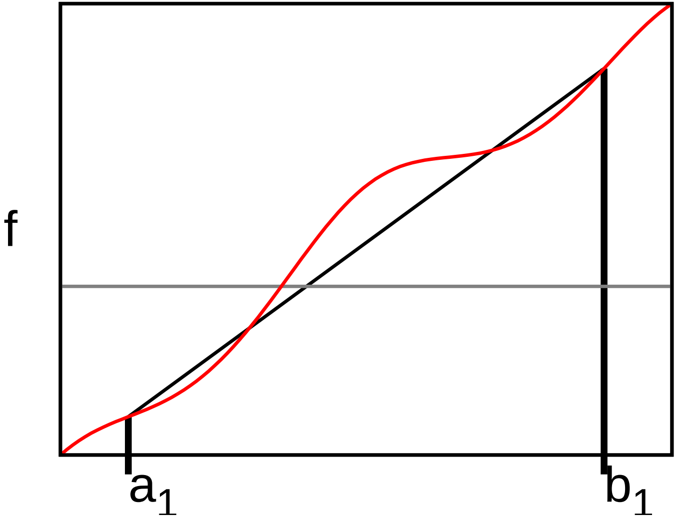
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



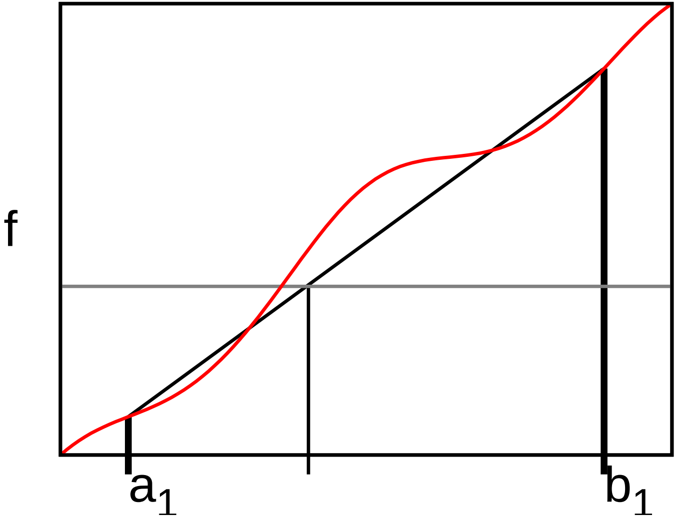
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



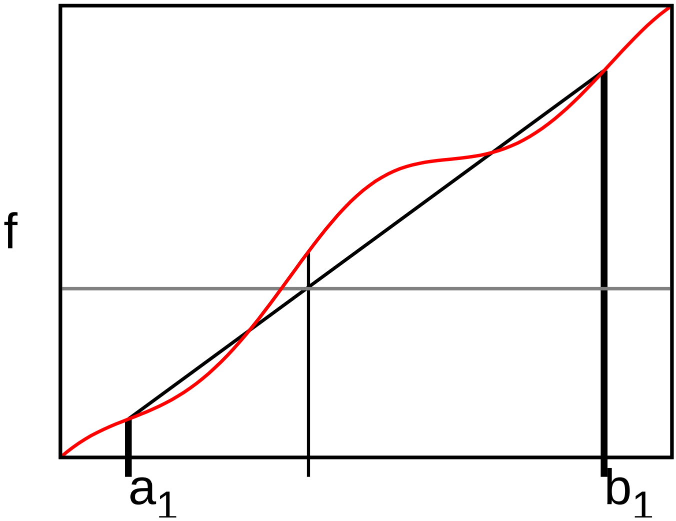
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



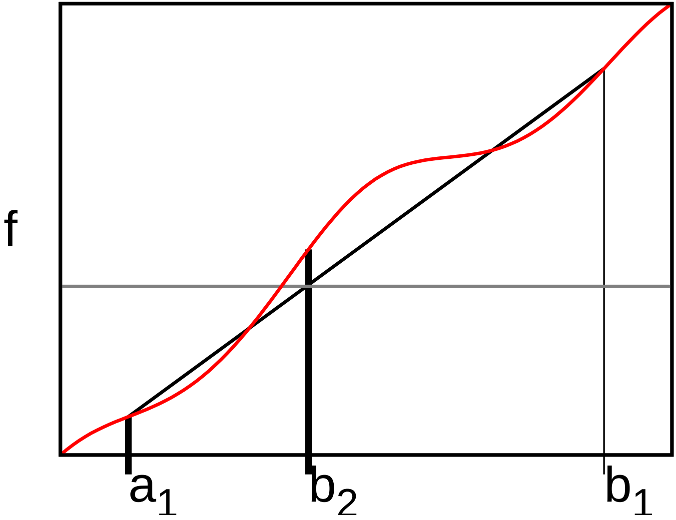
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



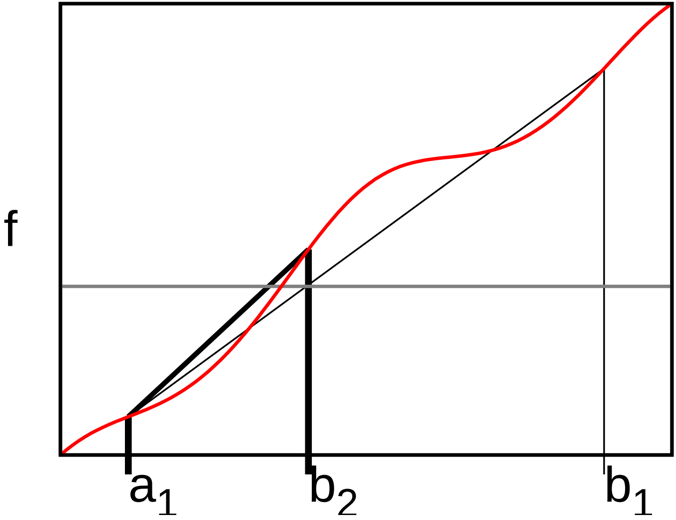
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



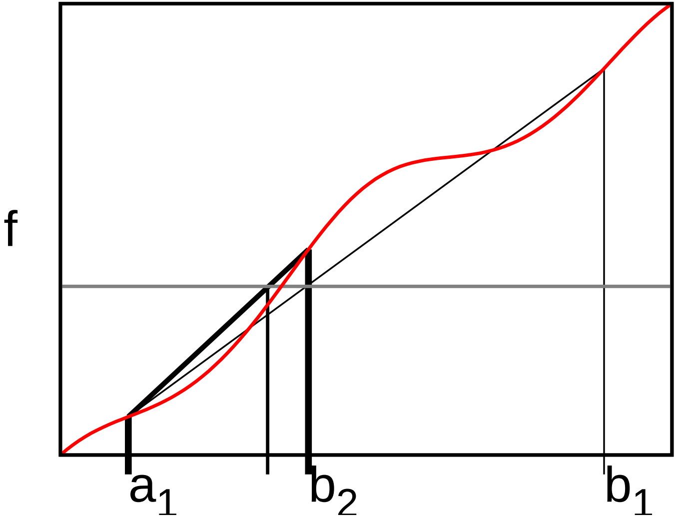
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



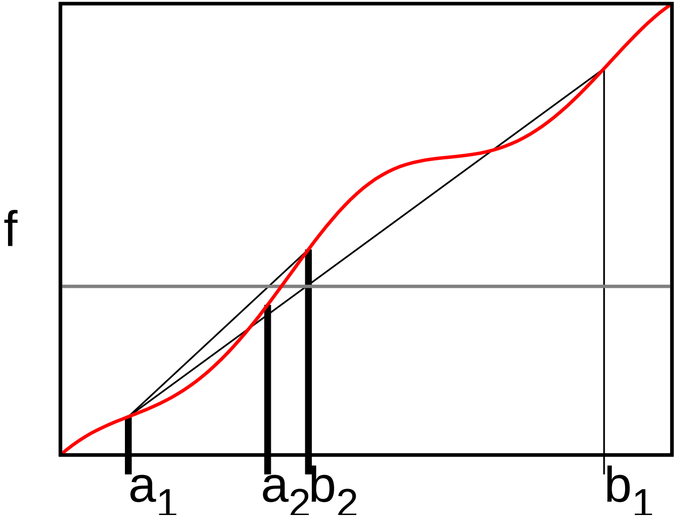
False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.



False Position Method

Linearly interpolate **f**. Solve for next **x**. Get new root bracket.

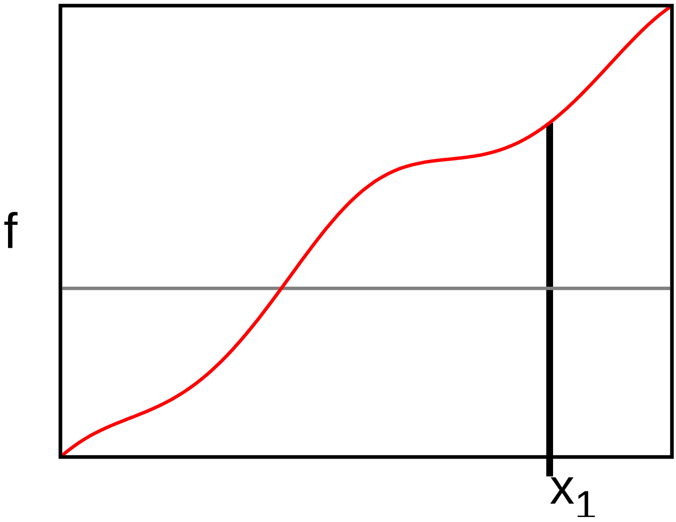


Other non-derivate methods

- Secant: Like false position, but keep most-recent point.
- Ridders' method: uses an exponential fit.
- Brent's method: uses an inverse quadratic fit.

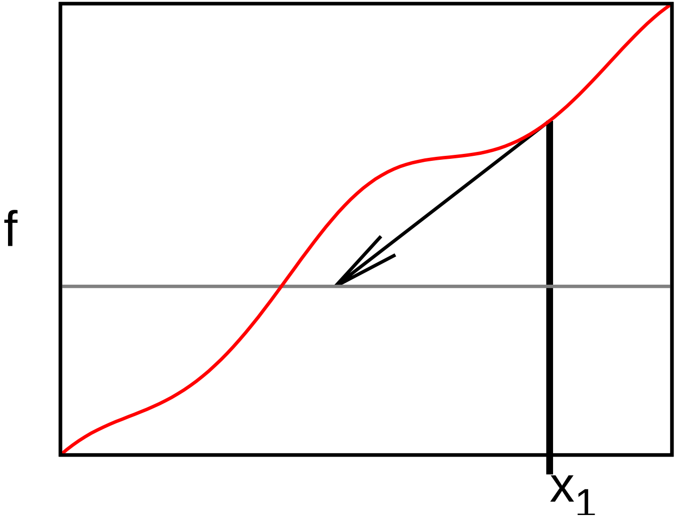
Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



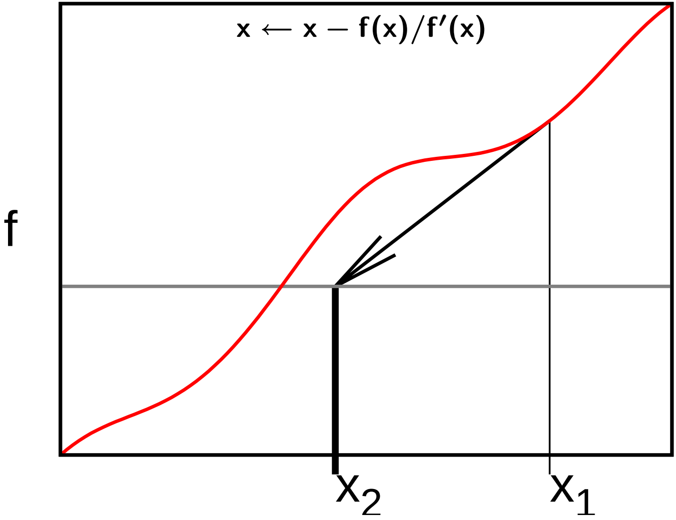
Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



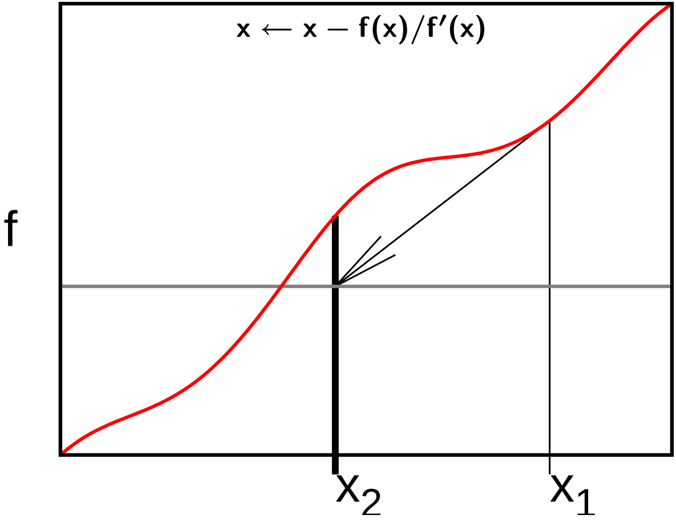
Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



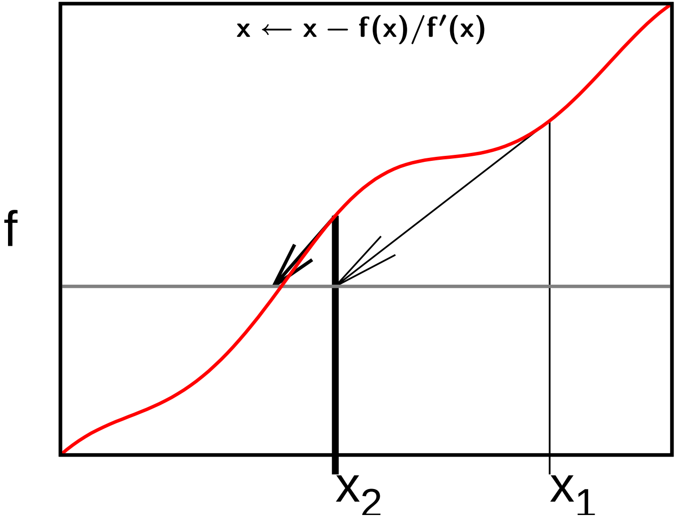
Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



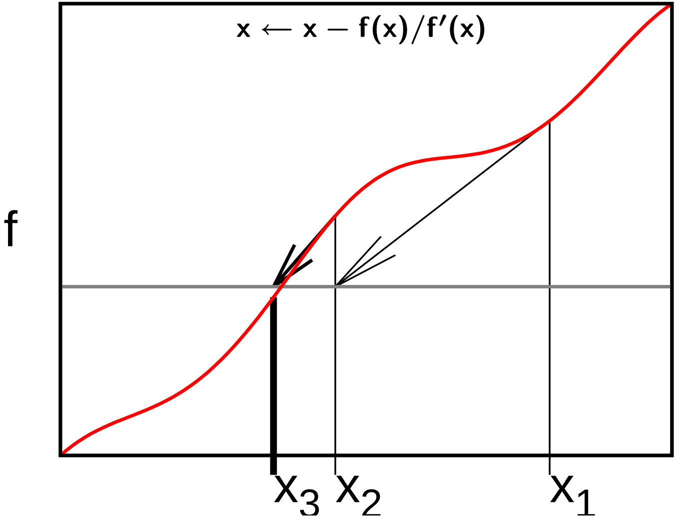
Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



Newton-Raphson method

Guess **x**. From function and derivate, approximate root. Repeat.



Convergence and Stability

method	convergence	stability
Bisection	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n$	Stable
Secant	$\epsilon_{n+1} = c\epsilon_n^{1.6}$	No bracket guarrantee
False position	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^{1.6}$	Stable
Ridders'	$\epsilon_{n+2} = c\epsilon_n^2$	Stable
Brent	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^2$	Stable
Newton-Raphson	$\epsilon_{n+1} = c\epsilon_n^2$	Can be unstable

SciNet compute • calcul CANADA

GNU Scientific Library (GSL)

Is a C library containing many useful scientific routines:

- Root finding
- Minimization
- Sorting
- Linear algebra, Eigen-systems
- Fast Fourier transforms
- Integration, differentiation, interpolation, approximation
- Random numbers
- Statistics, histograms, fitting
- Monte Carlo integration, simulated annealing
- ODEs
- Polynomials, permutations
- Special functions
- Vectors, matrices

GSL root finding example

```
#include <iostream>
#include <gsl/gsl_roots.h>
struct Params {
   double v, w, a, b, c;
};
double examplefunction(double x, void * param) {
   Params* p = (Params*)param;
   return p->a*cos(sin(p->v+p->w*x))+p->b*x-p->c*x*x;
}
int main() {
   double x_lo = -4.0;
   double x_hi = 5.0;
   Params args = {0.3, 2/3.0, 2.0, 1/1.3, 1/30.0};
   gsl_root_fsolver* solver;
   gsl_function fwrapper;
   solver=gsl_root_fsolver_alloc(gsl_root_fsolver_brent);
   fwrapper.function = examplefunction;
   fwrapper.params = &args;
   gsl_root_fsolver_set(solver, &fwrapper, x_lo, x_hi);
```

GSL root finding example

```
    std::cout << " iter [ lower, upper] root err\n";
    int status = 1;
    for (int iter=0; status and iter < 100; ++iter) {
        gsl_root_fsolver_iterate(solver);
        double x_rt = gsl_root_fsolver_root(solver);
        double x_lo = gsl_root_fsolver_x_lower(solver);
        double x_hi = gsl_root_fsolver_x_upper(solver);
        std::cout << iter <<' '<< x_lo <<' '<< x_hi
        <<' '<< x_rt <<' '<< x_hi - x_lo << "\n";
        status = gsl_root_test_interval(x_lo,x_hi,0,0.001);
    }
    gsl_root_fsolver_free(solver);
    return status;
}
```

GSL root finding example

```
   std::cout << " iter [ lower, upper] root err\n";
   int status = 1;
   for (int iter=0; status and iter < 100; ++iter) {
      gsl_root_fsolver_iterate(solver);
      double x_rt = gsl_root_fsolver_root(solver);
      double x_lo = gsl_root_fsolver_x_lower(solver);
      double x_hi = gsl_root_fsolver_x_upper(solver);
      std::cout << iter <<' '<< x_lo <<' '<< x_hi
      <<' '<< x_rt <<' '<< x_hi - x_lo << "\n";
      status = gsl_root_test_interval(x_lo,x_hi,0,0.001);
   }
   gsl_root_fsolver_free(solver);
   return status;
}
```

Compilation and likage:

```
$ g++ -c -O2 -I GSLINCDIR gslrx.cc -o gslrx.o
$ g++ gslrx.o -o gslrx -L GSLLIBDIR -lgsl -lgslcblas
```

(specify directories if installed in non-standard locations)

Multidimensional Root Finding

$$\vec{f}(\vec{x}) = \vec{0} \qquad \text{or} \qquad \begin{aligned} f_1(x_1, x_2, x_3, \ldots, x_D) &= 0 \\ f_2(x_1, x_2, x_3, \ldots, x_D) &= 0 \\ &\vdots \\ f_D(x_1, x_2, x_3, \ldots, x_D) &= 0 \end{aligned}$$

- Cannot bracket a root with a finite number of points.
- Roots of each equation define a $\mathbf{D-1}$ hypersurface.
- Looking for possible intersections of hypersurfaces.

Newton-Raphson for Multidimensional Root Finding

Given a good initial guess, Newton-Raphson can work in arbitrary dimensions:

$$
\begin{aligned}
\vec{\mathbf{f}}(\vec{\mathbf{x}}_0 &+ \delta\vec{\mathbf{x}}) = \vec{\mathbf{0}} \\
\vec{\mathbf{f}}(\vec{\mathbf{x}}_0) &+ \frac{\partial\vec{\mathbf{f}}}{\partial\vec{\mathbf{x}}}\cdot\delta\vec{\mathbf{x}} = \vec{\mathbf{0}} \\
\vec{\mathbf{f}}(\vec{\mathbf{x}}_0) &= -\frac{\partial\vec{\mathbf{f}}}{\partial\vec{\mathbf{x}}}\cdot\delta\vec{\mathbf{x}} \\
\vec{\mathbf{f}}(\vec{\mathbf{x}}_0) &= -\mathbf{J}\cdot\delta\vec{\mathbf{x}} \\
\delta\vec{\mathbf{x}} &= -\mathbf{J}^{-1}\cdot\vec{\mathbf{f}}(\vec{\mathbf{x}}_0)
\end{aligned}
$$

Requires inverting a $\mathbf{D}\times\mathbf{D}$ matrix, or at least, solving a linear set of equations: see lecture on linear algebra.

Convergence and Stability

- As in 1D, Newton-Raphson can be unstable.
- Need some safe guard that our iteration steps do not spin out of control.
- Several ways, e.g. make sure that $||\vec{\mathbf{f}}(\vec{\mathbf{x}})||^2$ gets smaller in each time step.
- This can potentially still fail, but usually does the trick.

Optimization

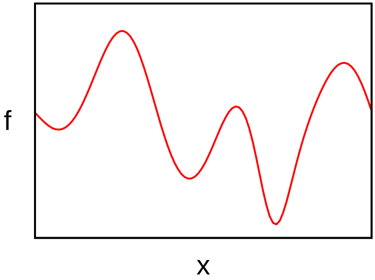
Optimization

Optimization=Minimization=Maximization

There's a function $\mathbf{f(x_1, x_2, , \ldots, x_D)}$.

- Want to know for which set of $\mathbf{x_1, x_2, \ldots}$ the function is at its minimum.
- Maximizing $\mathbf{f}$ is minimizing $-\mathbf{f}$.
- Different from solving $\partial \mathbf{f} / \partial \mathbf{x_i} = \mathbf{0}$ because of integration properties.
- Once more, $\mathbf{D = 1}$ is substantially different from higher dimensional cases.
- There's no guarantee that we find the global minimum, methods return local minimums.

1D Minimization



- Local minimum: smallest function value in its neighbourhood.
- Global minimum: smallest such function value for all **x**.

Strategy

- Generalize bracketing to minimization.
- **a**, **b**, **c** bracket a minimum of **f** if

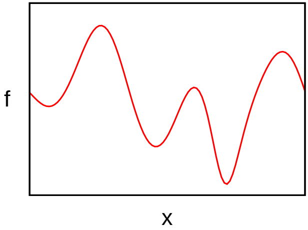
$$\mathbf{f(a)} > \mathbf{f(b)} < \mathbf{f(c)}$$

- Once a minimum is bracketed, we can successively refine the brackets.

Minimization Methods for 1D

Golden Section Method

- Choose the larger of the intervals $[\mathbf{a}, \mathbf{b}]$ and $[\mathbf{b}, \mathbf{c}]$.
- Place a new point **d** a fraction **w** into the larger interval.
- New triplet is succession of points bracketing the minimum.
- Optimal choice for $\mathbf{w} = \frac{1}{2}(3 - \sqrt{5}) \approx 0.38197$



Brent's Method

- Uses a quadratic interpolation using the three points, to find a new point.

All of these are in the GSL library.

Finding the global minimum

- There's no guarantee that we find the global minimum, methods return local minima.
- If we're not sure: Essentially try again from a different starting points, and check which of the found minima is the lowest.
- Some methods, especially those using random numbers like simulated annealing, have ways of escaping local minima.

Multidimensional Minimization

- No bracketing method available
- Not as dire as multi-dimensional root finding:
  down is down, so as local as the function is bounded from below, you're likely to find at least a local minimum.

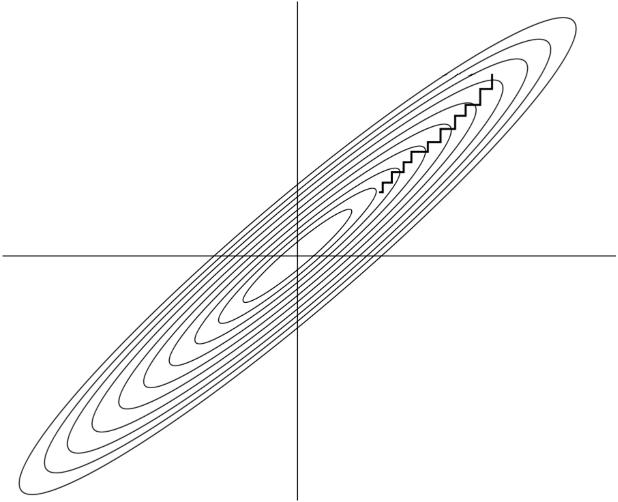
Common strategy

- Most methods are built upon 1D methods.
- Start from a trial point **P**
- Perform one dimensional minimizations in **D** 'independent' directions
- ~~Hope for the be~~st. Test for convergence.

SciNet compute • calcul CANADA

Finding ‘independent’ directions

Just choosing **D** directions will lead to staircase scenario.



It might get to the minimum eventually, but requires a lot of small 1d minimizations to do so.

Finding 'independent' directions

Non-gradient based direction sets

- Use previous 1D minimization to guide the choice of direction sets. *Powell's Methods*

Gradient based direction sets

- This one uses the derivatives to determine in which **D** directions to go next.
- Steepest Descent
  It may seem like a good idea just to go in the direction of the gradient: steepest descent. But you would have to take many straight turns.
- Conjugate Gradient.
  Essentially, you're solving the system as if it is quadratic. Better.

All of these are in the GSL library.

Simulated Annealing

- All of these methods have trouble with functions that have (many) local minima.
- Let's look for inspiration in Nature.
- Why? Nature is a very good minimizer.
- For instance, when temperature drops below 0^oC, water molecules find a new low energy configuration, which any of our previous minimization methods would have a tough time finding: $\mathbf{D} = \mathcal{O}(10^{23})$.
- The mechanism behind this is that while the temperature drops, the molecules are constantly moving in and out of local mimima, thus exploring much more of configurational space.

Simulated Annealing

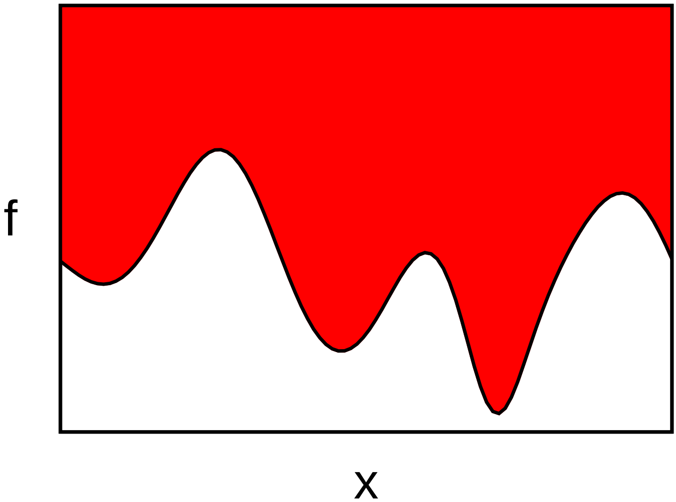
Ok, so how does that help us?

- **D** dimensional space becomes a configuration space.
- **f** becomes an energy function for the system.
- We assign the system a temperature **T**.
- We assign the system a dynamics that is such that the equilibrium distribution of configurations is

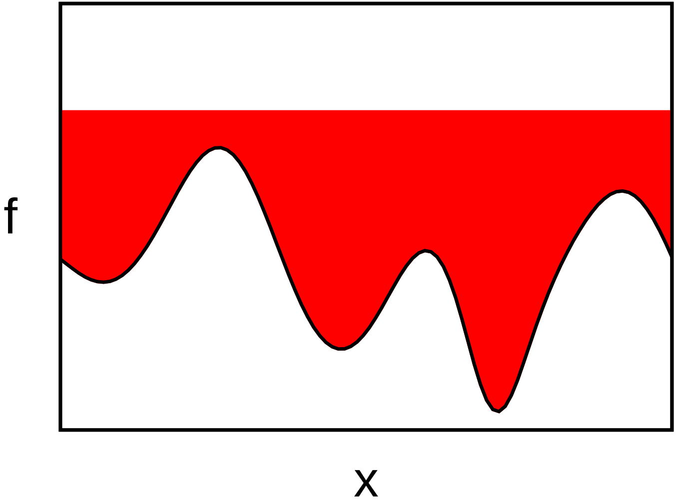
  $$\mathbf{P(\vec{x}) \propto e^{-f(\vec{x})/T}}$$

  E.g. use the Monte Carlo method from last week's lecture!
- Evolve this system with given dynamics, but from time to time, we reduce the temperature **T**.
- As $\mathbf{T} \longrightarrow \mathbf{0}$ the system will get trapped in a minimum. If we anneal slow enough, there's a good chance it's the global minimum.

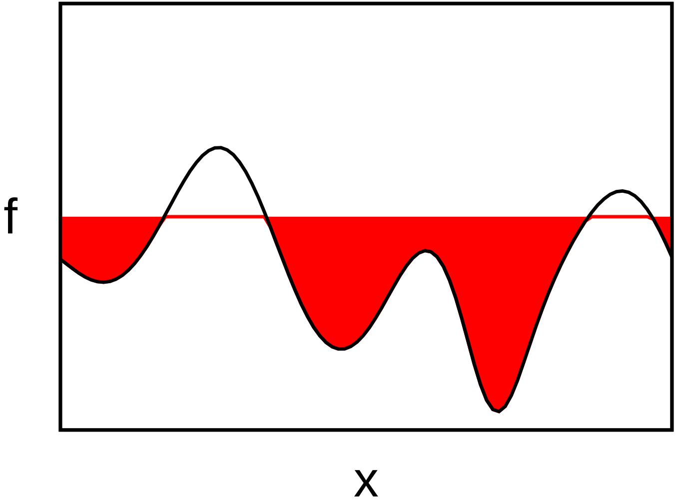
Simulated Annealing



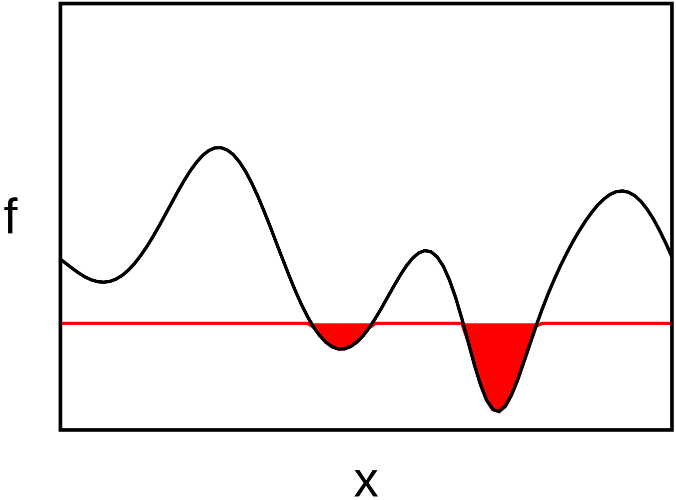
Simulated Annealing



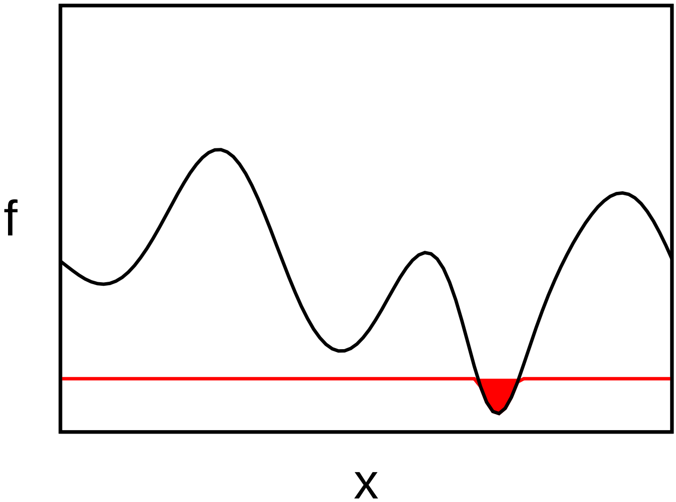
Simulated Annealing



Simulated Annealing



Simulated Annealing



Simulated Annealing

Final notes

- Typically does not converge as fast as the deterministic ones.
- But applicable in many 'hard' cases (travelling salesman).
- GSL has it too.