Root-finding, Interpolation, Numerical Differentiation

Numerical Methods

Cezar Santos

FGV/EPGE

Acknowledgements

- This set of slides is heavily based on notes by Georg Dürnecker and Grey Gordon.

Root-finding

Two of the most common problems in economics:

$$f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n$$

and

$$g(x) = x, \quad g : \mathbb{R}^n \to \mathbb{R}^n$$

The first is called a root-finding problem, and the second is a fixed-point problem.
Note that the two are equivalent in the following sense:

- For any f, define $g = f + x$. Then $f = 0 \Leftrightarrow g = x$.
- For any g, define $f = g - x$. Then $g = x \Leftrightarrow f = 0$.

Often it is infeasible to solve this analytically; use numerical methods to approximate the solution.

Root-finding

One way to solve $f(x) = 0$ is to convert it to a minimization problem:

$$\arg\min_x \frac{1}{2} \sum_i f_i(x)^2.$$

The min will be equal to zero only at the root. You will learn many techniques for multi-dimensional minimization. However, there are good reasons not to do this:

1. When trying to solve $f(x) = 0$, we are often very demanding and want $f(x)$ very close to zero. When doing minimization, we are usually satisfied with just being fairly close to zero.
2. The minimization problem may have many local minima. These types of problems are very hard to solve.
3. If using comparison methods only, the root-problem has more information.

But, What is Zero?

Many ancient cultures didn't have a 0. For them, numbers were $1, 2, 3, \ldots$.
The computer has the opposite problem. It thinks many numbers are zero.

But, What is Zero?

Many ancient cultures didn't have a 0. For them, numbers were $1, 2, 3, \ldots$.
The computer has the opposite problem. It thinks many numbers are zero.

Numbers close to zero may be represented as zero.
Plus, this may vary with the operation being done.
So... BE CAREFUL!!!

Root-finding in One Dimension

We will begin with the simplest case, one-dimensional root-finding:

$$f(x) = 0, \quad f : \mathbb{R} \to \mathbb{R}.$$

Bisection Method

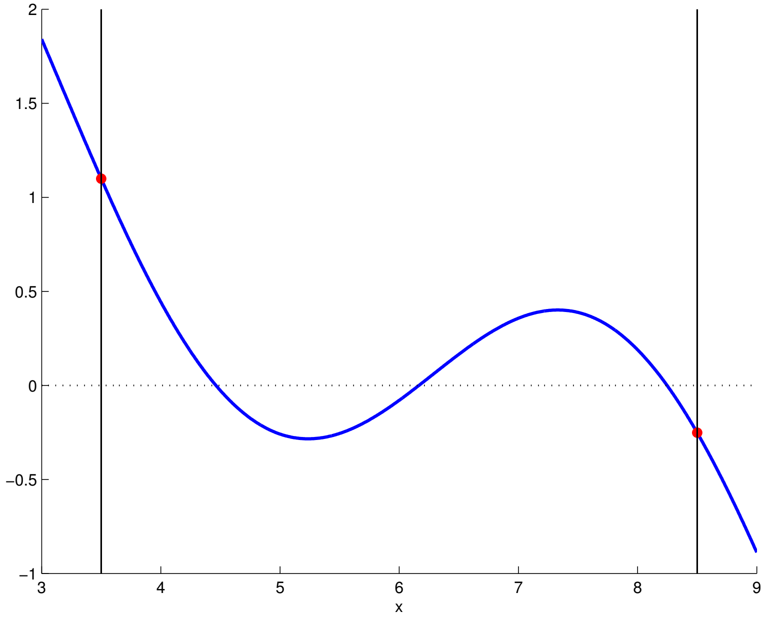
- Bisection: A simple and robust method for finding the root of a **univariate** continuous function $f(x)$ on a closed interval $[a, b]$
- Always converges to the solution - if one exists, and if the initial interval includes the solution
- Does not rely on the derivatives of the function $\Rightarrow$ can be used to find roots of non-smooth functions
- Basic idea: If f is continuous and $f(a)$ and $f(b)$ are of opposite sign $\Rightarrow$

  $$\exists x^* \in [a, b] \quad \text{for which} \quad f(x^*) = 0 \quad \text{(Intermediate Value Theorem)}$$

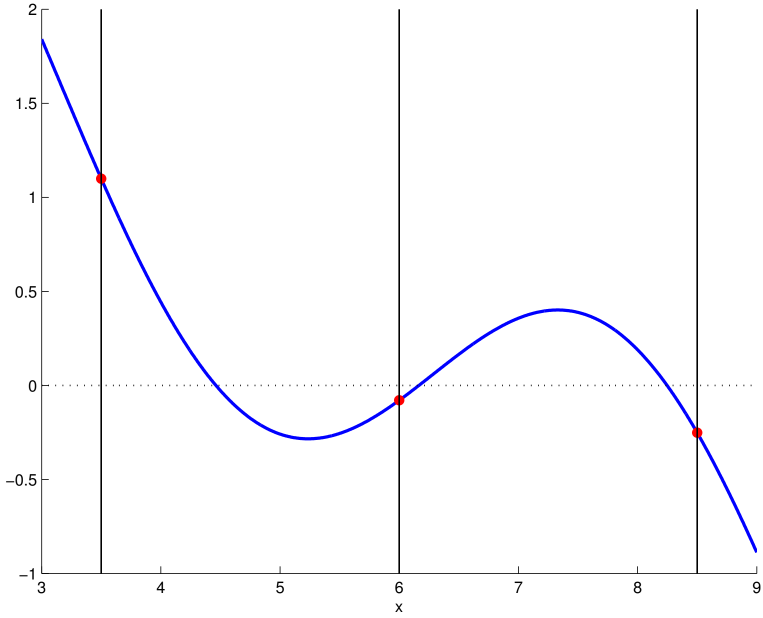
Bisection Method

- Algorithm (for $f(a) < 0$ and $f(b) > 0$)
  - (i) define lower $\underline{x}$ and upper bound $\bar{x}$ of interval: Use $\underline{x} = a$ and $\bar{x} = b$
  - (ii) compute midpoint of interval $c = \frac{\bar{x}+\underline{x}}{2}$ and evaluate $f(c)$
  - (iii) if $f(c) > 0$ set $\bar{x} = c$, otherwise if $f(c) < 0$ set $\underline{x} = c$
  - (iv) if $|\bar{x} - \underline{x}| \leq \varepsilon$ then stop and call c a root, otherwise go to step (ii)
  - (v) Check precision of solution $|f(c)| \leq \delta$

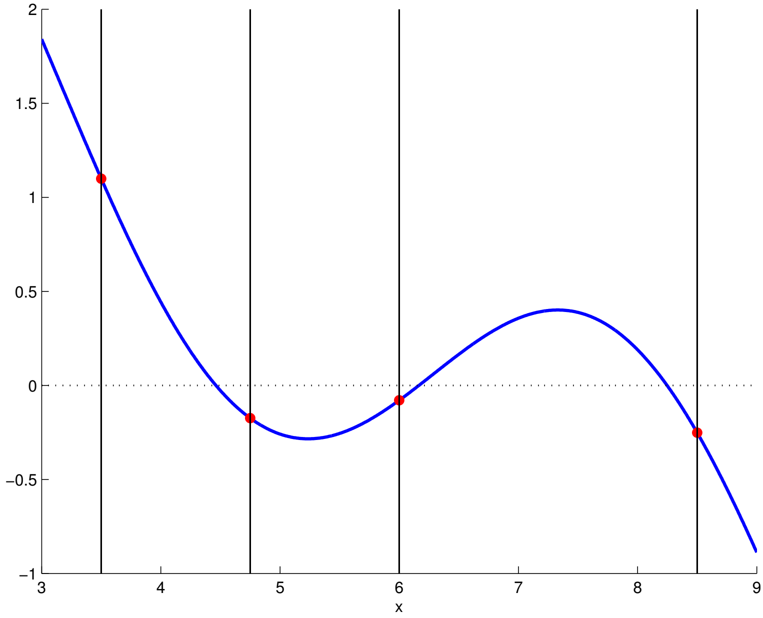
Bisection method



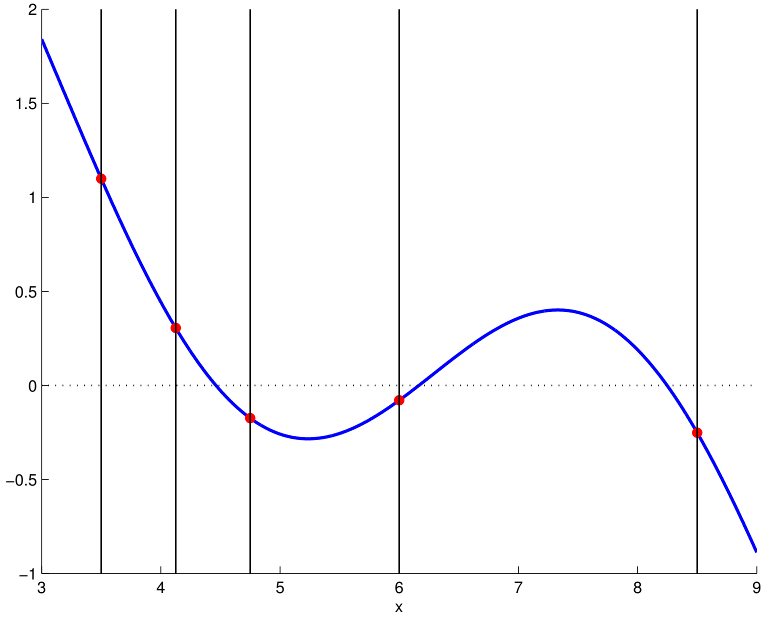
Bisection method



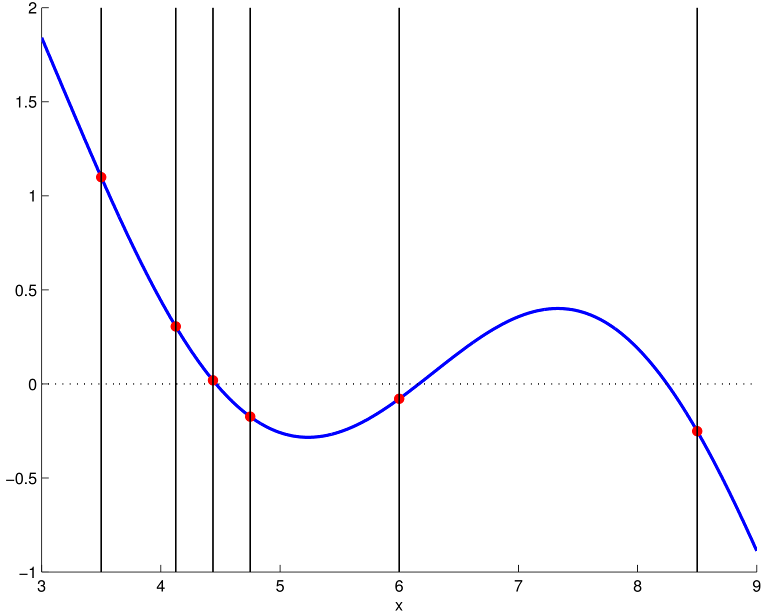
Bisection method



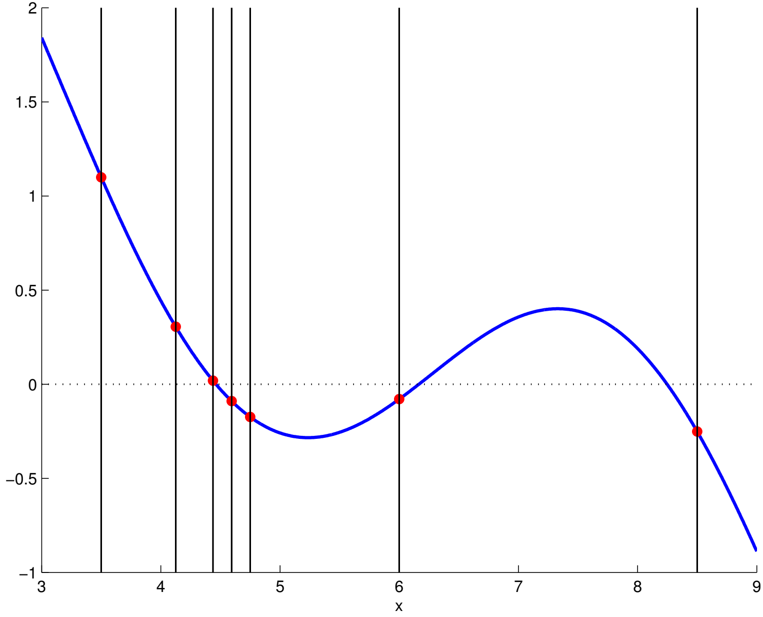
Bisection method



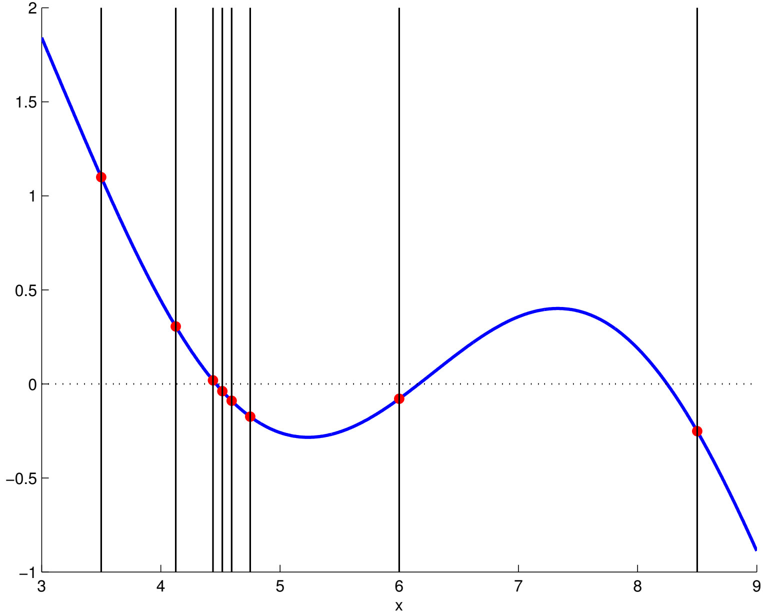
Bisection method



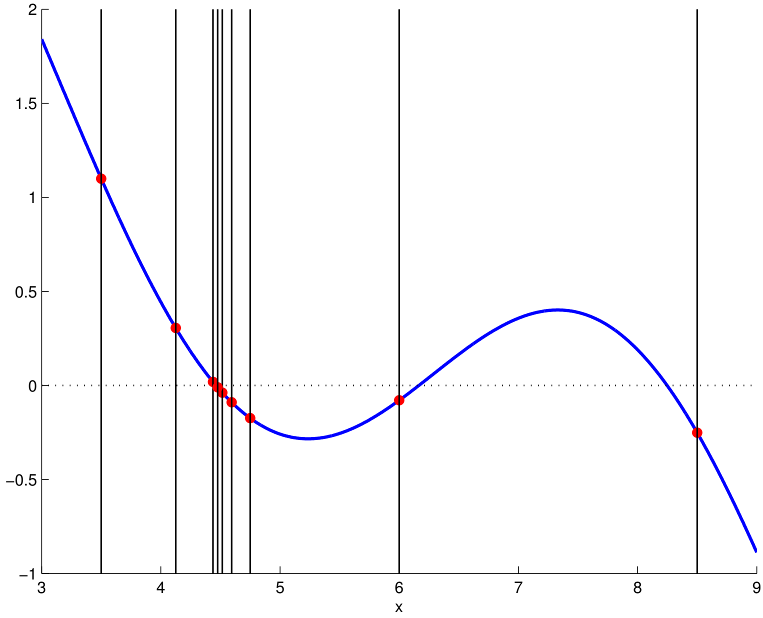
Bisection method



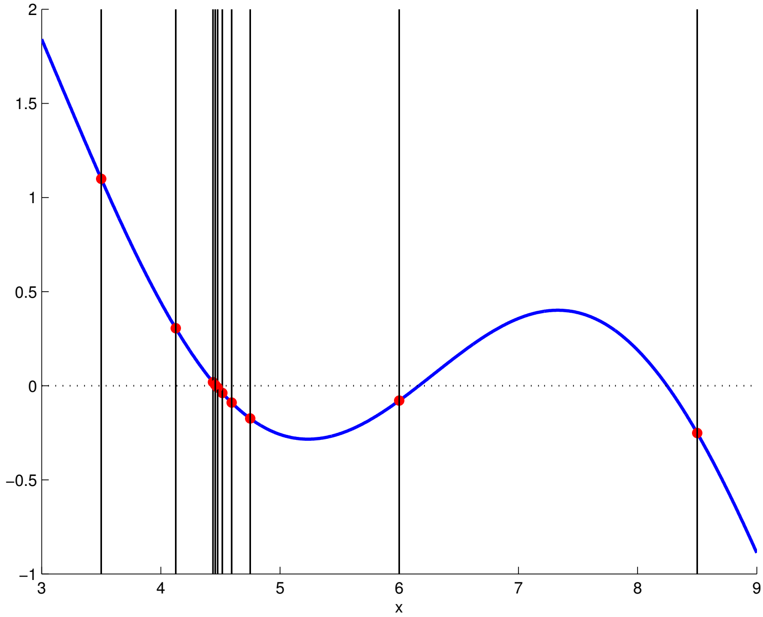
Bisection method



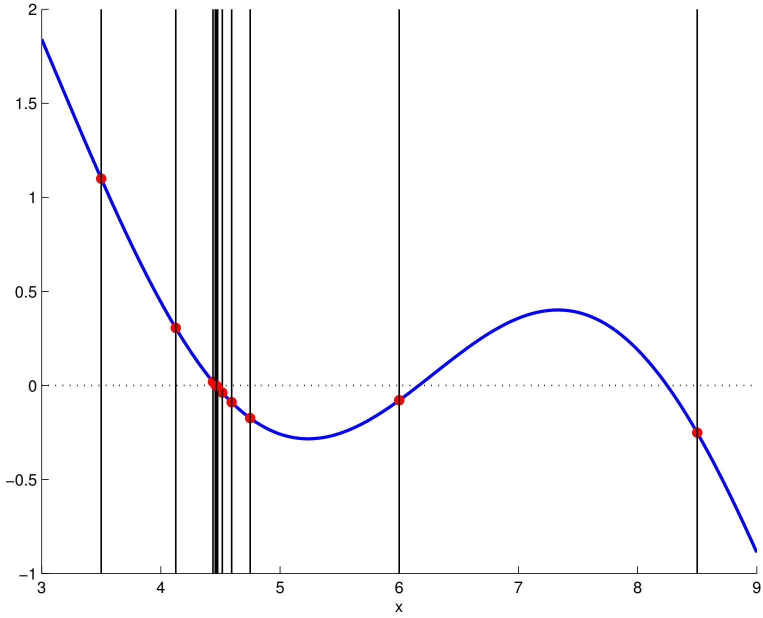
Bisection method



Bisection method



Bisection method



Bisection Method

Advantages

1. Finds a zero of any C^0 function.
2. Doesn't require even continuity, it always converges.
3. Extremely simple and lightweight.
4. Frequently used.

Disadvantages

1. Convergence is slow relative to other methods. It takes 3 iterations to reduce error by 1 decimal place ($.5^3 \approx 10\%$).
2. It does not exploit information about function curvature, i.e. slow.
3. Have to find initial bracket (true of all these methods).

Newton-Raphson Method

- Iterative scheme which uses the first-order approx. of f to find the zero
- Strategy: Start at x_0 ($f(x_0)$ must be defined) and approximate f linearly around x_0

$$g(x) = f(x_0) + f'(x_0)(x - x_0) \qquad \text{where} \quad f'(x) = \frac{\partial f(x)}{\partial x}$$

- Find the root of $g(x)$, i.e. find x_1 which solves

$$0 = f(x_0) + f'(x_0)(x_1 - x_0) \qquad \Rightarrow \qquad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

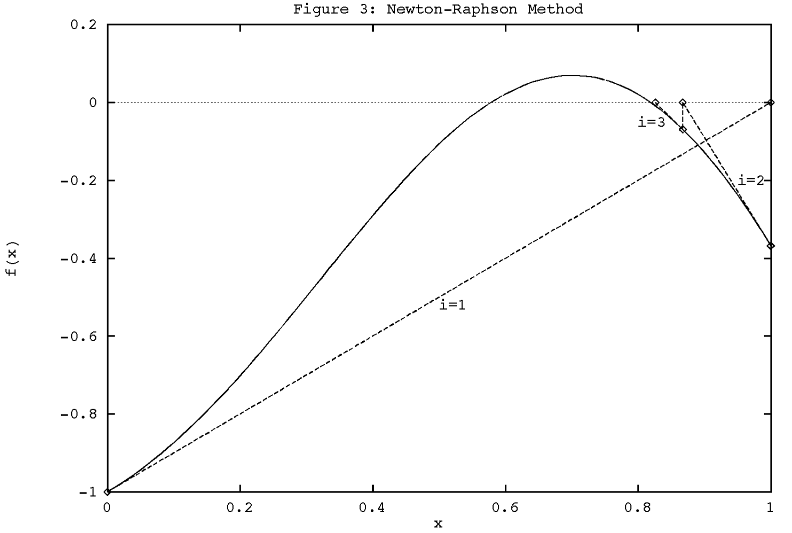
- Check if $f(x_1)$ is defined[1]. If yes, repeat the procedure, i.e. linearly approximate f at x_1. This gives rise to the following iterative scheme

$$x_{s+1} = x_s - \frac{f(x_s)}{f'(x_s)}$$

The solution x^* is found if $f(x_{s+1}) \approx 0$

[1] If it is not defined, choose a point between x_0 and x_1.

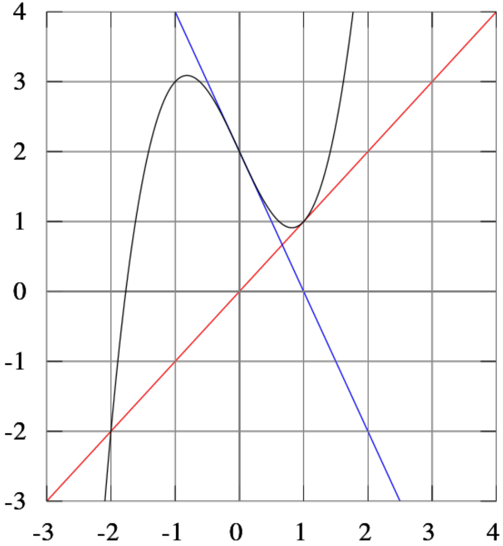
Newton-Raphson Method - Working :)



Newton-Raphson Method

- Essence: This method reduces a non-linear problem to a sequence of linear problems, where the zeros are easy to compute
- Converges quadratically - is generally faster than bisection
- Newton's method is fast when it works, but it may not always converge

Newton-Raphson Method - Not Working :(



Problems with Newton-Raphson

- Choice of initial conditions
- $f(x_{s+1})$ may not be defined. Remedy: Modified Newton-Raphson method: It backtracks from x_{s+1} along the direction $f'(x_s)$ to a point x'_{s+1} at which f can be evaluated.
- Computation of the derivative of f can be costly or impossible
- Example: In the Aiyagari-model we aim at finding the interest rate r^* which makes aggregate excess-supply of capital, $K^{s-d}(r)$ equal to zero. Problem: There is no analytical expression for $K^{s-d}(r) \Rightarrow$ impossible to compute the derivative

Using the Secant instead of the Tangent

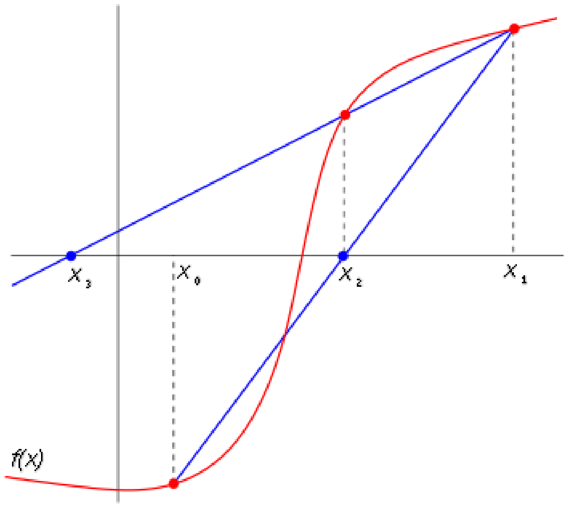
- What if we can't compute the derivative?
- Solution: Use the slope of the secant that connects two points $(x_s, f(x_s))$ and $x_{s+1}, f(x_{s+1})$ instead of $f'(x_s)$

$$\text{Secant: } g(x) = f(x_{s+1}) + \frac{f(x_{s+1}) - f(x_s)}{x_{s+1} - x_s}(x - x_{s+1})$$

- From the zero of the secant $g(x) = 0$ we get an iterative scheme

$$x_{s+2} = x_{s+1} - \frac{x_{s+1} - x_s}{f(x_{s+1}) - f(x_s)} f(x_{s+1})$$

Problems with the Secant



Notice: In this example, convergence is unlikely to occur.

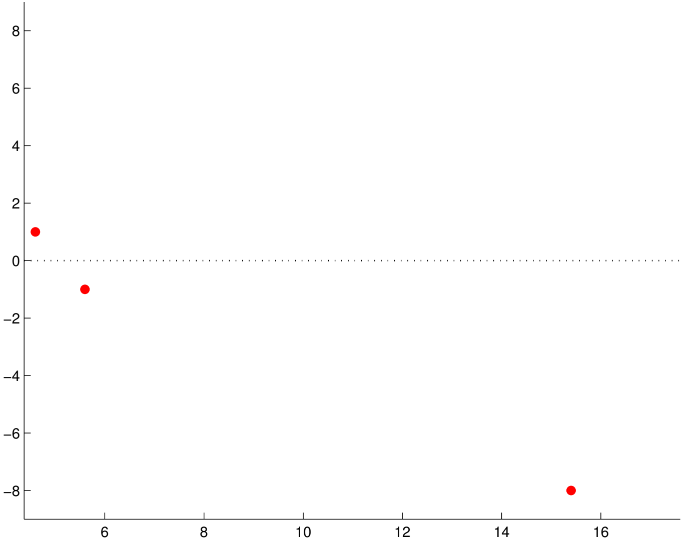
Brent's Method: A Hybrid

Mixing things:

- Inverse quadratic interpolation (IQI) if things are going well
- Bisection when they are not

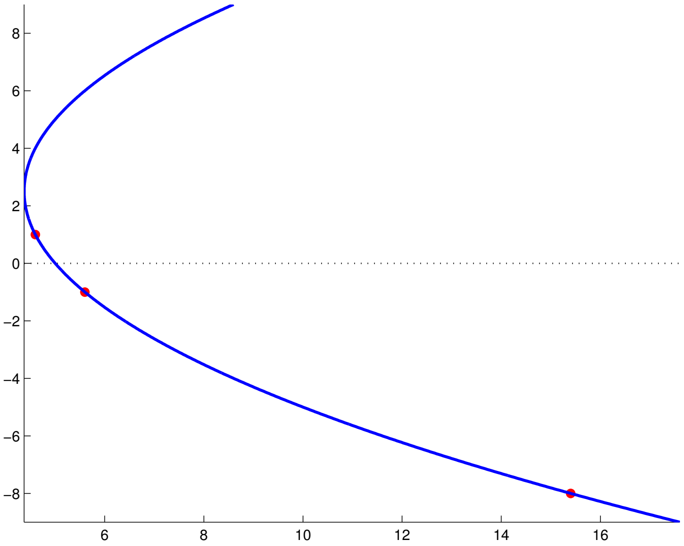
Brent's method

Any point in the algorithm has a triplet:



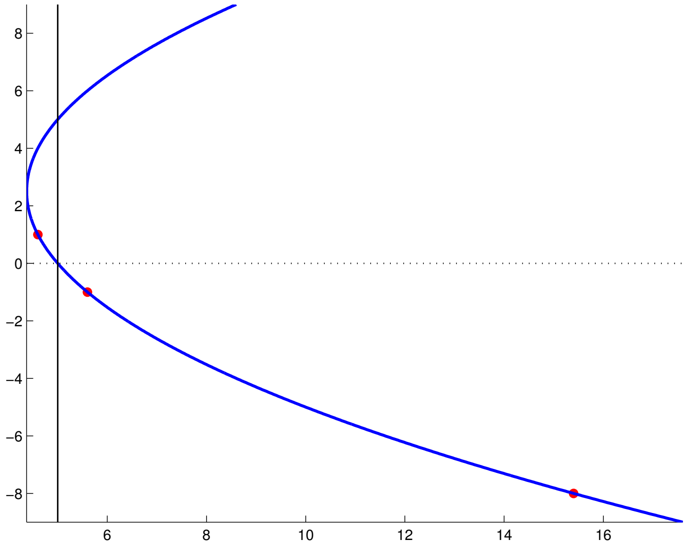
Brent's method

An inverse quadratic interpolation step is attempted



Brent's method

The trial point is the unique root of the IQI polynomial

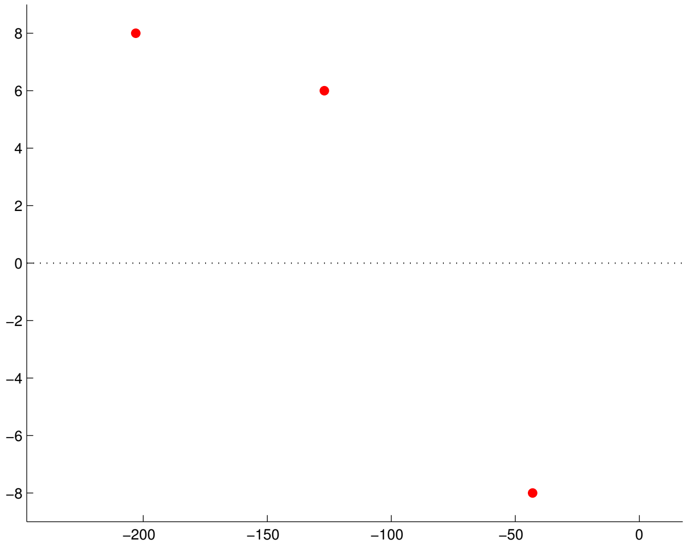


Brent's method

In the preceding example, the IQI worked fairly well in that it shrunk the bracket. Brent adds an additional check that the bracket must shrink *enough*.
IQI can completely fail as in the next example.

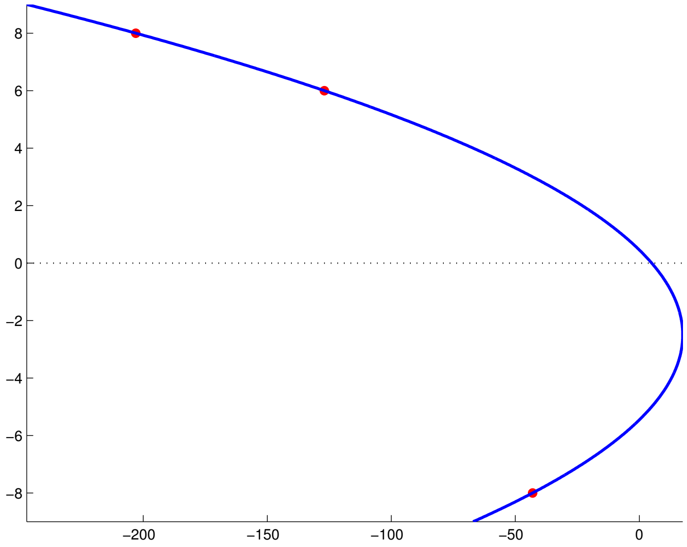
Brent's method

The triplet does not look so bad



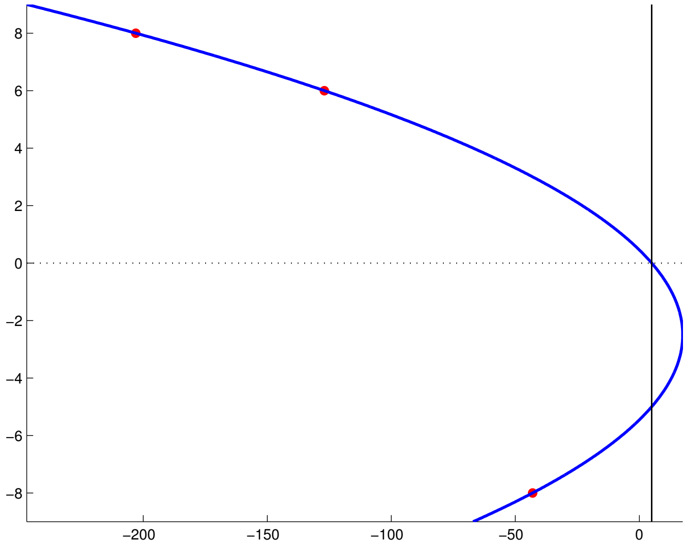
Brent's method

An inverse quadratic interpolation step is attempted



Brent's method

The new trial point is outside the bracket, that's bad



Brent's method

Some additional details about Brent's method:

- Sometimes a parabola does not exist (e.g., if $f(a) = f(b)$). In this case, the secant method is used to generate a trial point.
- With a trial point from either the secant method or IQI, it checks that the trial (1) falls within the bracket and (2) shrinks the bracket enough.
- If the trial point does not meet both these criteria, a bisection step is taken.
- "The devil is in the details," and this is complicated to program. Of course, someone already programmed it. (Grey Gordon has a Fortran version on his website)

Brent's method

Advantages

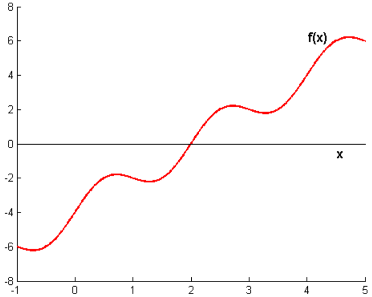
1. Finds a zero of any C^0 function.
2. Doesn't require even continuity, it always converges.
3. Converges very quickly for any differentiable functions.
4. Known and in fact Matlab implements fzero using Brent's method.

Disadvantages

1. Difficult to program
2. Not a lightweight program, there are several multiplications, additions, and if statements.

Homotopy continuation method

- No gradient-based method (Newton, Secant, etc.) is globally convergent
- Often, convergence requires a good initial guess about x^*
- Example: $f(x) = 2x - 4 + \sin(\pi x)$



- Possible remedy: Homotopy methods

Homotopy continuation method

- Idea:
  - Start with simpler function for which the solution (root) is known
  - Gradually deform the function until the function of interest is reached
  - In each step, solve for the root using the solution of previous step as starting point
- Essence: Homotopy deforms simple function into function of interest, computes series of zeros and ends with zero to function of interest
- $\Rightarrow$ Globally convergent way of finding zeros

- Problem of interest $f(x) = 0$ where $f : \mathbb{R}^n \to \mathbb{R}^n$
- Homotopy function $H(x,t) : \mathbb{R}^{n+1} \to \mathbb{R}^n$ continuously deforms a simple(r) function $g(x)$ into $f(x)$

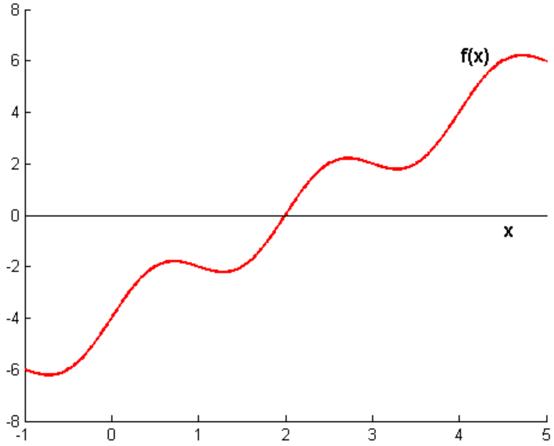
$$H(x,0) = g(x) \qquad H(x,1) = f(x)$$

- Example: Linear homotopy

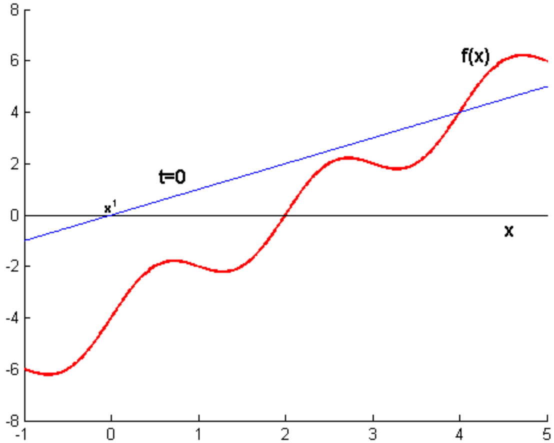
$$H(x,t) = tf(x) + (1-t)g(x)$$

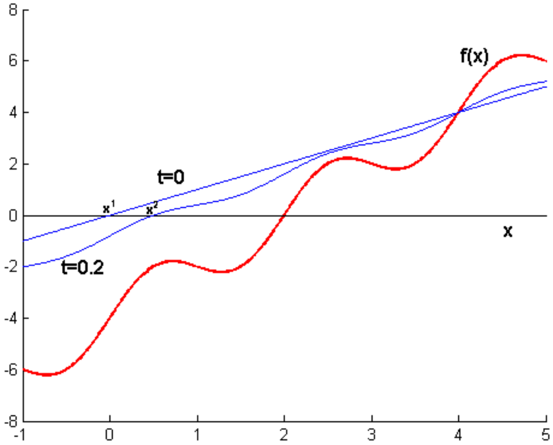
- Simple continuation method
  - Form the sequence t: $t^i \in \{t^1, t^2, ..., t^n\}$, with $t^1 = 0$ and $t^n = 1$
  - Set $i = 1$ and find roots of $H(x,t^1) = 0$ and set x^1 equal to the solution
  - For $i > 1$, solve $H(x,t^i) = 0$ using x^{i-1} as the initial guess

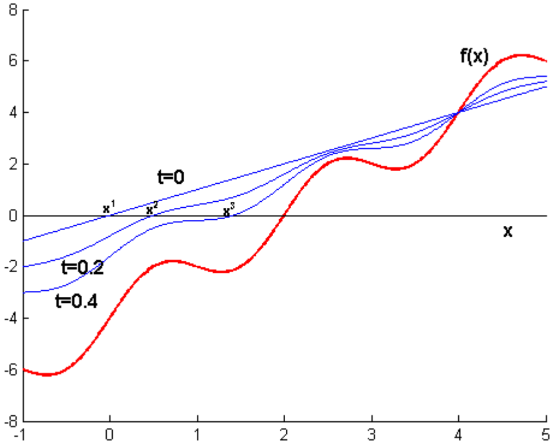
- Example: $f(x) = 2x - 4 + \sin(\pi x)$
  - Want to solve for $f(x) = 0$
  - Homotopy: $H(x,t) = [1-t]x + t[2x - 4 + \sin(\pi x)]$, where $H(x,0) = x$



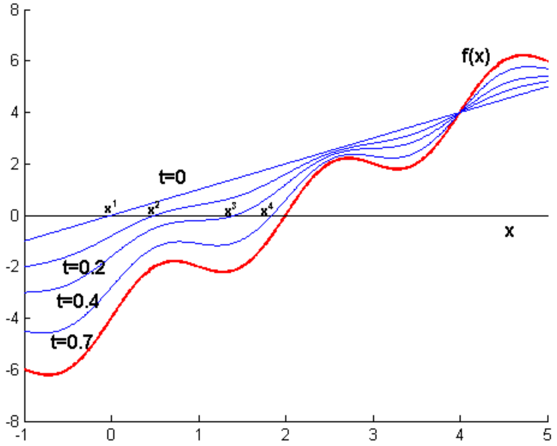
f(x)
x
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5


f(x)
t=0
x^1
x
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5

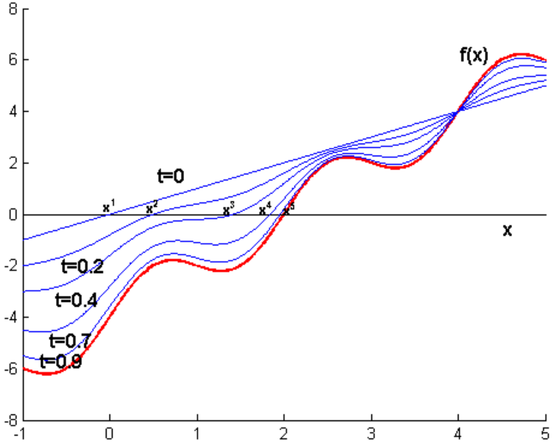

f(x)
t=0
t=0.2
x^1
x^2
x



f(x)
t=0
t=0.2
t=0.4
x^1
x^2
x^3
x
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5



f(x)
t=0
t=0.2
t=0.4
t=0.7
x^1
x^2
x^3
x^4
x

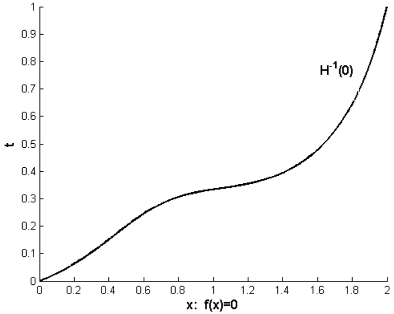


f(x)
t=0
t=0.2
t=0.4
t=0.7
t=0.9
x^1
x^2
x^3
x^4
x^5
x
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5

- The homotopy is a function of x and t. The object of interest is the set
$$H^{-1}(0) = \{(x,t) | H(x,t) = 0\}$$

- In general, for the homotopy to work, there has to be a continuous path in $H^{-1}(0)$ which connects the zeros of $H(x,0) = g(x)$ to the zeros of $H(x,1) = f(x)$

- $H^{-1}(0)$ for example from above:



Root-finding in n Dimensions

We'll now consider the case of root finding in n dimensions, i.e., finding an x such that

$$f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n.$$

This is a very hard problem.

- Unlike in the one-dimensional case, no guaranteed way to find a root.
- It is important to know a few techniques that you can try.
- The cost of many derivative methods grows quickly in n.
- You should try to avoid these types of problems if you can.

Gauss-Seidel Method

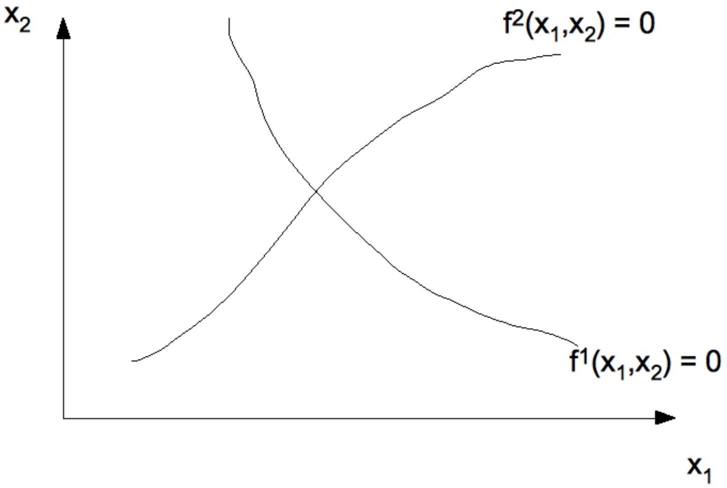
- This method starts with a point $\mathbf{x}^s = [x_1^s, x_2^s, ..., x_n^s]$ and obtains a new point by solving

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) \Longleftrightarrow \begin{cases} 0 = f^1(x_1^{s+1}, x_2^s, ..., x_n^s), \\ \\ 0 = f^2(x_1^{s+1}, x_2^{s+1}, ..., x_n^s), \\ \quad \vdots \\ 0 = f^n(x_1^{s+1}, x_2^{s+1}, ..., x_n^{s+1}) \end{cases}$$

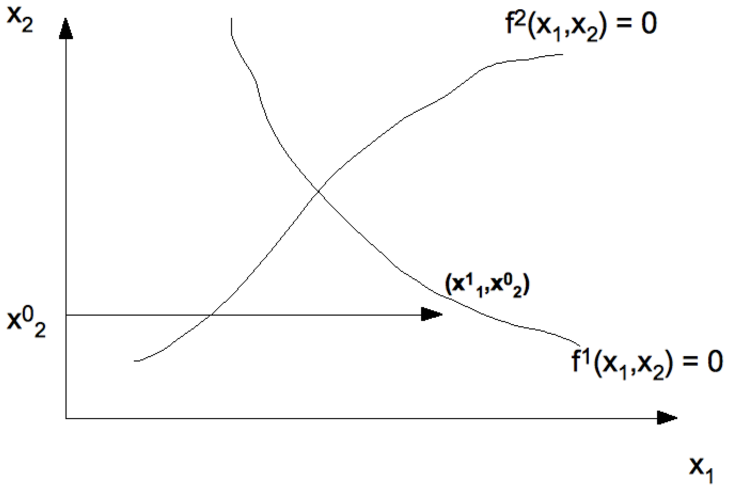
Gauss-Seidel Method

- Process is continued until two successive solutions $\mathbf{x}^s$ and $\mathbf{x}^{s+1}$ are sufficiently close together
- The problem of solving n equations simultaneously is reduced to solving n single equations in one variable x_i^{s+1}
- Convergence is not guaranteed.
- Possible remedy: Do not use the new value of x_i^{s+1} when solving for x_{i+1}^{s+1} but a combination of x_i^{s+1} and x_i^s

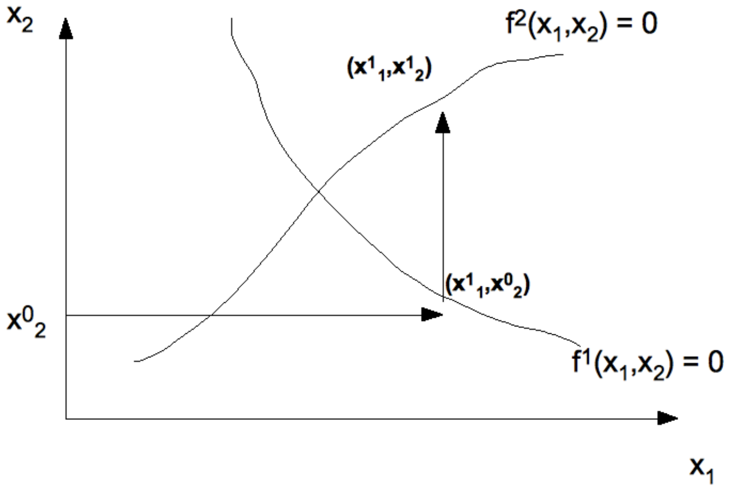
Example for Gauss-Seidel, with $n = 2$



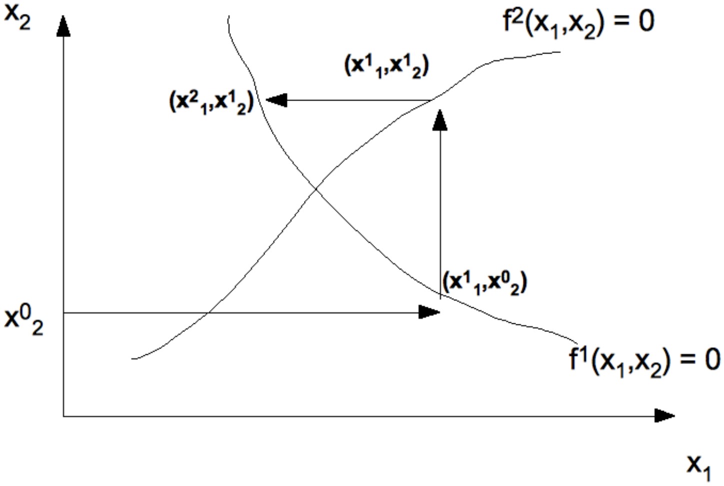
Example for Gauss-Seidel, with $n = 2$



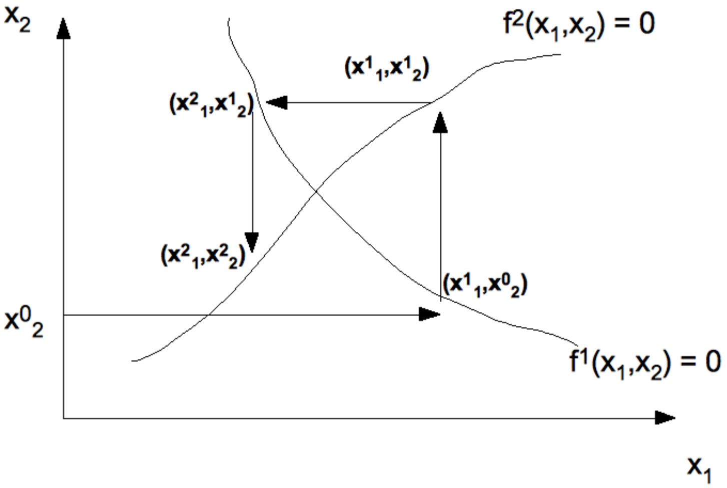
Example for Gauss-Seidel, with $n = 2$



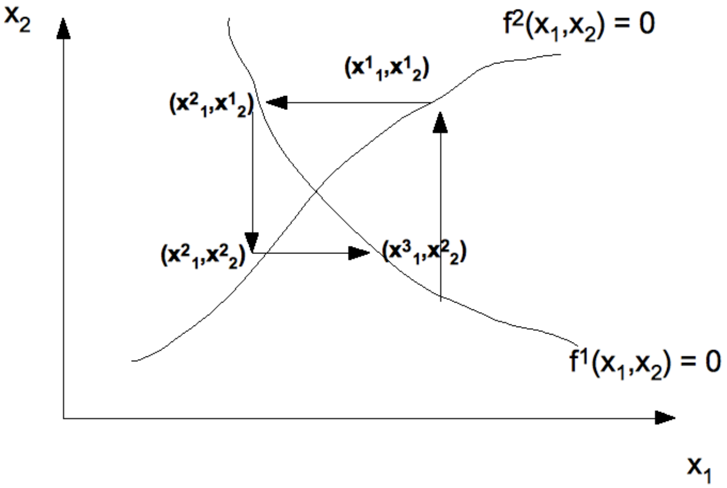
Example for Gauss-Seidel, with $n = 2$



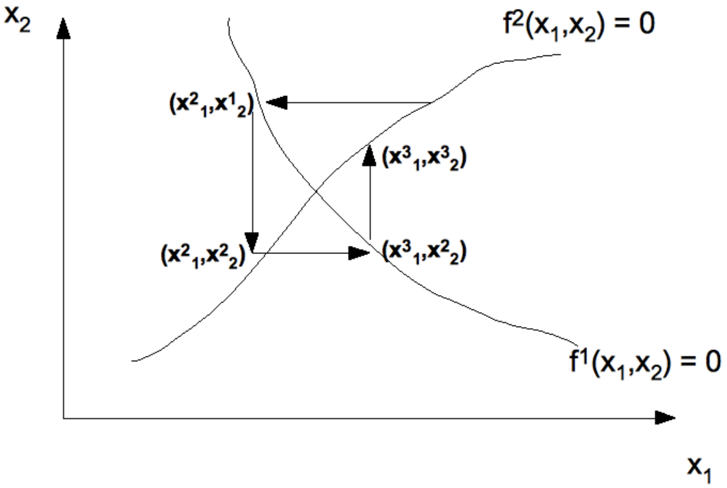
Example for Gauss-Seidel, with $n = 2$



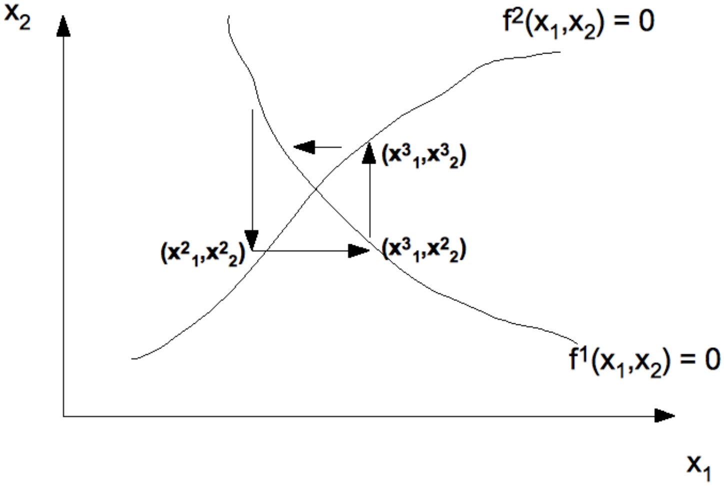
Example for Gauss-Seidel, with $n = 2$



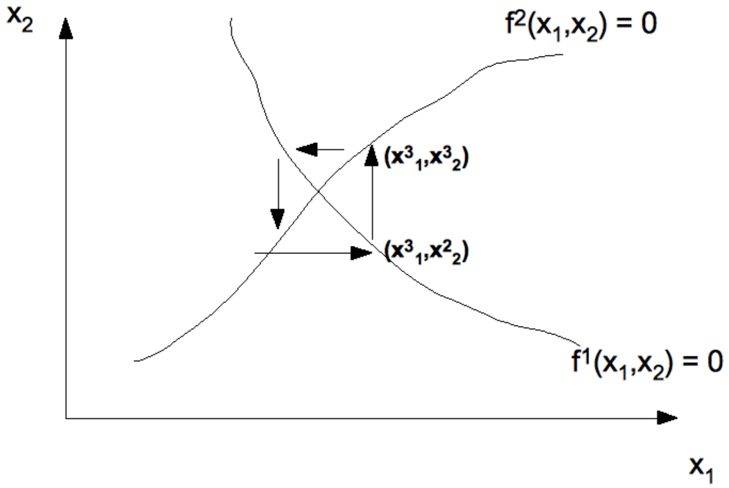
Example for Gauss-Seidel, with $n = 2$



Example for Gauss-Seidel, with $n = 2$



Example for Gauss-Seidel, with $n = 2$



Newton's Method in the Multi-variable Case

- What if $f(x) : R^n \to R^n$?
- Linear approximation of $g(x)$ in the n-dimensional case:

  $\mathbf{g}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$ where $\mathbf{J}$ is the Jacobian matrix
- from which we get

$$\mathbf{g}(\mathbf{x}) = 0 \quad \Rightarrow \quad \mathbf{x} = \mathbf{x}_0 - \mathbf{J}(\mathbf{x}_0)^{-1}\mathbf{f}(\mathbf{x}_0)$$

- The computation of $\mathbf{J}$ can be computationally costly, especially for large n
- Broyden's method approximates $\mathbf{J}$ with a matrix which is updated after each step

Aside: (almost) Never Compute Inverses!

In general, you should never try to invert a matrix.

- Computing matrix inverses has a very expensive computational cost. It is also error prone if the matrix is not very well-conditioned.
- Further, it can usually be avoided. Often, what one wants is not the matrix inverse itself, say A^{-1}, but the inverse times some b. I.e., one wants $x = A^{-1}b$.
- To compute this, it is much more accurate and fast to solve $Ax = b$ using Lapack.

Newton's Method

Advantages

- For a good enough initial guess, very quick.
- For small n, computational cost not very significant.
- Classic method, everyone knows of it.

Disadvantages

- Not guaranteed to converge
- Computing the Jacobian (even if analytic) for large n is costly.
- If one needs to use finite-differences to compute the Jacobian, is extra costly and error prone.

Broyden's Method

- What if we don't have the Jacobian?
- In the univariate version of Newton, we reverted to the secant method.
- Here, it's not as easy, but Broyden suggested a method to approximate the Jacobian as iterations go.

Broyden's Method

- As with the secant method, we need two points as an initial guess: x_0 and x_1
- Choose some initial guess for the Jacobian J_0. It can be the identity matrix if we don't have anything better.
- Update the Jacobian matrix with

$$J_i = J_{i-1} + \frac{(\Delta_i - J_{i-1}\delta_i)\delta_i^T}{\delta_i^T \delta_i},$$

  where $\delta_i = x_i - x_{i-1}$ and $\Delta_i = f(x_i) - f(x_{i-1})$.
- The guess is updated in a Newton-type way:

$$x_{i+1} = x_i - J_i^{-1} f(x_i)$$

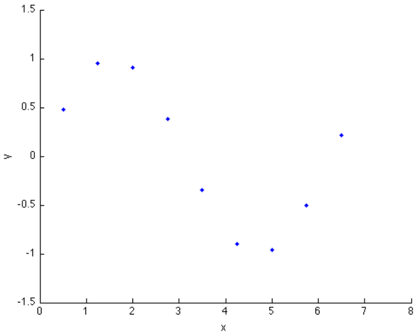
- As with any gradient-type method, there's no guarantee it converges.

Interpolation

- Suppose: $y = g(x)$ but $g(\cdot)$ is unknown. What is observed is a collection of n points in R^2, $D = \{(x_0, y_0), \ldots, (x_n, y_n)\}$ where $y_i = g(x_i)$, $x_i \neq x_j \; \forall i \neq j$
- Task: Find a function $\hat{g}(x)$ that approximates $g(x)$ as closely as possible
- Interpolation: Construct $\hat{g}$ so that $y_i = \hat{g}(x_i) \rightarrow$ Interpolant and the underlying function must agree at a finite number of points. Additional restrictions may be imposed (first derivatives, smoothness, etc)

Interpolation

With $\hat{g}$ at hand we can evaluate $g(\cdot)$ at any $x \neq x_i$



Interpolation Schemes

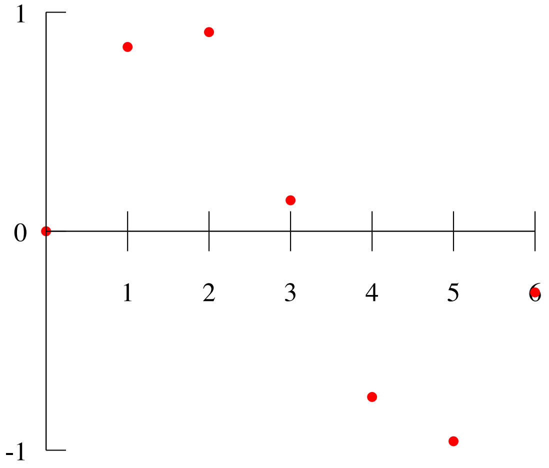
There are many different types of interpolation. Listed by popularity,

1. Linear interpolation
2. Spline interpolation
3. Polynomial interpolation (including Chebyshev collocation)
4. Nearest-neighbor interpolation

The next slides give examples from Wikipedia.

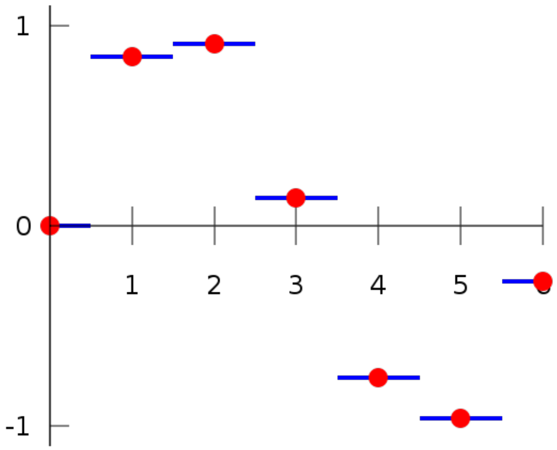
Interpolation Examples

Raw data:



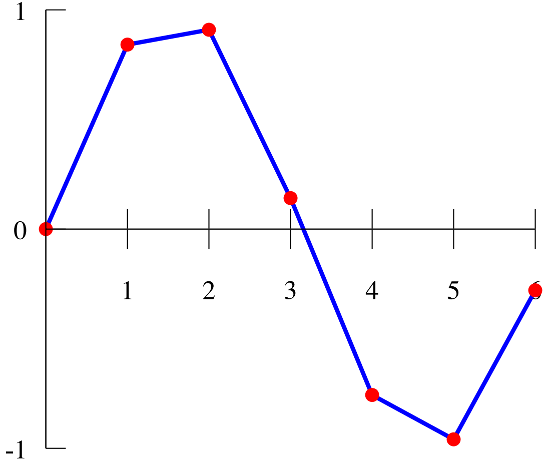
Interpolation Examples

Nearest neighbor:



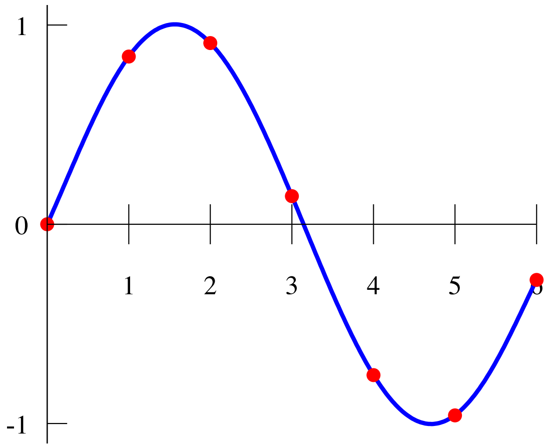
Interpolation Examples

Linear:



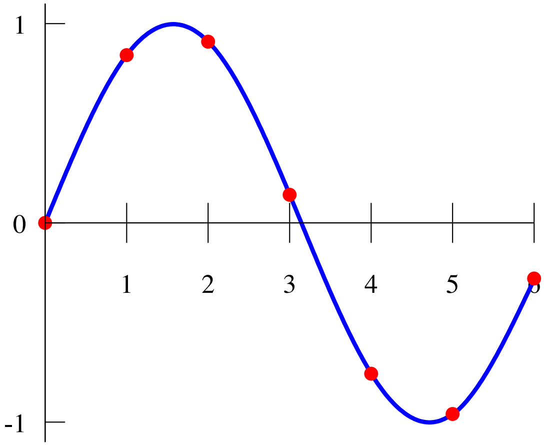
Interpolation Examples

Polynomial:



Interpolation Examples

Spline:



Spline and polynomial interpolation differ from each other.

Interpolation Overview

All the interpolation methods have strengths and weaknesses which we will learn about (except for nearest neighbor interpolation).

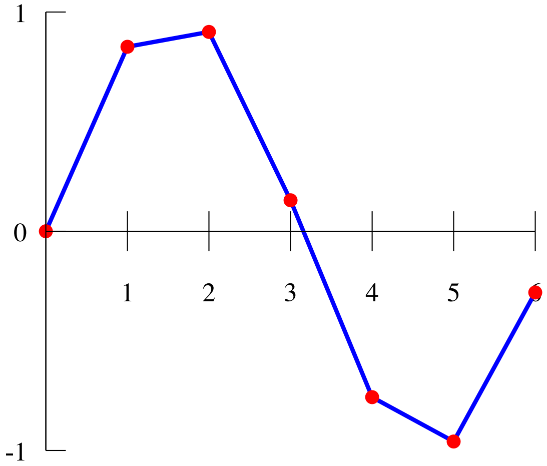
All the interpolation methods can be used to avoid discretization of the choice space:

$$V(a,e) = \max_{a' \in [\underline{A}, \overline{A}]} u(c) + \beta \mathbb{E}_e \hat{V}(a', e')$$

where $\hat{V}(\cdot, e)$ is the interpolant going through data $\{a_i, V(a_i, e)\}_{i=1}^{\#\mathscr{A}}$. Note that the state space is still discretized.

Linear Interpolation

We begin with the simplest method, linear interpolation:



Piecewise Linear Interpolation

- Simplest way to interpolate: Connecting the points
- Draw straight lines between successive data points
- Piecewise-linear interpolant $\hat{g}$ is given by

$$\hat{g}(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) \qquad \text{for} \quad x \in [x_i, x_{i+1}]$$

Linear Interpolation

While humble, linear interpolation has significant advantages which account for its popularity.

Advantages

- Very fast.
- Preserves weak concavity, monotonicity, and "positivity."
- Extrapolation properties are well-behaved.
- Well-known.
- Simple formula that is, e.g., easy to integrate.
- Easily generalizes to multi-dimensional case.
- Local: only the nearest values need to be known to interpolate.

Linear Interpolation

Of course, linear interpolation has significant disadvantages.

Disadvantages

- Doesn't preserve differentiability.
- Biased for strictly concave/convex functions.

Binary Search

Once you have the interpolant, to evaluate it an x value, one must know which interval x falls in. E.g.,

$$\hat{f}(x) = \begin{cases} y_1 + (x - x_1)m_1 & \text{if } x < x_1 \\ y_i + (x - x_i)m_i & \text{if } x \in [x_i, x_{i+1}] \\ y_{n-1} + (x - x_{n-1})m_n & \text{if } x > x_n \end{cases}$$

Suppose that the interpolation data has $\{x_i\}_{i=1}^{100{,}000}$.

One way would be to go through and check from $i = 1$ to $99{,}999$ whether $x \in [x_i, x_{i+1}]$. Obviously, this is slow.

Binary Search

Once you have the interpolant, to evaluate it an x value, one must know which interval x falls in. E.g.,

$$\hat{f}(x) = \begin{cases} y_1 + (x - x_1)m_1 & \text{if } x < x_1 \\ y_i + (x - x_i)m_i & \text{if } x \in [x_i, x_{i+1}] \\ y_{n-1} + (x - x_{n-1})m_n & \text{if } x > x_n \end{cases}$$

Suppose that the interpolation data has $\{x_i\}_{i=1}^{100{,}000}$.

One way would be to go through and check from $i = 1$ to $99{,}999$ whether $x \in [x_i, x_{i+1}]$. Obviously, this is slow.

Suppose the data is **sorted**.
How many times must the grid $\{x_i\}$ be checked to find the location of x?
Answer:

Binary Search

Once you have the interpolant, to evaluate it an x value, one must know which interval x falls in. E.g.,

$$\hat{f}(x) = \begin{cases} y_1 + (x - x_1)m_1 & \text{if } x < x_1 \\ y_i + (x - x_i)m_i & \text{if } x \in [x_i, x_{i+1}] \\ y_{n-1} + (x - x_{n-1})m_n & \text{if } x > x_n \end{cases}$$

Suppose that the interpolation data has $\{x_i\}_{i=1}^{100{,}000}$.

One way would be to go through and check from $i = 1$ to $99,999$ whether $x \in [x_i, x_{i+1}]$. Obviously, this is slow.

Suppose the data is **sorted**.
How many times must the grid $\{x_i\}$ be checked to find the location of x?
Answer: no more than $\log_2(100,000) + 1 \approx 18$ times.

Binary Search

Binary search is a “divide-and-conquer” technique that is similar to the advanced algorithm for exploiting concavity. It only works for sorted data.

Algorithm: given $x \in [x_a, x_b)$, find i^* s.t. $x \in [x_{i^*}, x_{i^*+1})$ for $\{x_i\}_{i=a}^{b}$.

1. Let $i = \lfloor \frac{b-a}{2} \rfloor$.
2. If $b - a = 1$, stop. $i^* = a$. O/w, go to 3.
3. If $x < x_i$, then redefine b as i, o/w redefine a as i. Go to 1.

Note: at each step, the search space is being cut in half.

Alternatives to Binary Search

Even though binary search is very efficient, there are sometimes faster ways. For instance, if you have a known formula for the x_i values, such as linearly spaced $x_i = a + ih$, one can sometimes get an analytical representation:

$$i = \left\lfloor \frac{x_i - a}{h} \right\rfloor$$

However, one must be very careful with numerical error here.

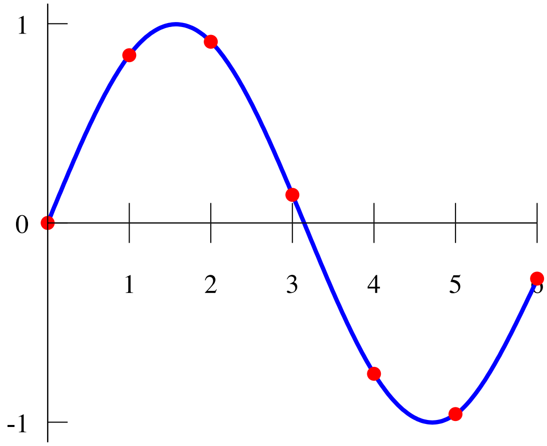
Spline interpolation

The biggest deficiency of linear interpolation is that it produces a non-differentiable interpolant. Splines are a different way to

interpolate that can produce an interpolant that is continuously differentiable up to a given order. In particular, they take piecewise-polynomials and combine them into a globally differentiable object.

Interpolation Examples

Spline:



Spline interpolation

Formally, for a function $s : [a, b] \to \mathbb{R}$, we have

> *$s(x)$ is a spline of order n if s is C^{n-2} on $[a, b]$ and there is a grid of points (called nodes [or knots]) $a = x_0 < x_1 < \ldots < x_m = b$ s.t. $s(x)$ is a polynomial of degree $n-1$ on each subinterval $[x_i, x_{i+1}]$.* —*Judd (1998)*

This is different from polynomial interpolation which uses one high-order polynomial for the entire domain $[a, b]$.

Spline interpolation

Spline of different orders go by different names:

- **Linear spline/interpolant** (order 2): linear interpolant (a spline of order 2 since it is C^0 globally and uses polynomials of degree 1 (linear) in each interval)
- **Quadratic spline** (order 3): each interval is a parabola and the entire spline is C^1.
- **Cubic spline** (order 4): each interval is a cubic and the entire spline is C^2.

There are also something called a a "pchip" in Matlab for a **piecewise cubic hermite polynomial** that is C^1 but has a cubic in each interval.

Cubic Splines

- Let's focus on cubic splines (for now)
- On each interval $[x_{i-1}, x_i]$ the spline is a cubic

$$s(x) = a_i + b_i x + c_i x^2 + d_i x^3 \qquad x \in [x_{i-1}, x_i]$$

- For each interval i there is a separate set of coefficients (a_i, b_i, c_i, d_i)
- In total there are $n + 1$ data points, n intervals and $4 * n$ unknown coefficients
- How to find the coefficients? Use conditions/restrictions on the spline

- Condition 1: $s(x_i) = g(x_i) = y_i$

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad i = 1, ..., n$$

- Condition 1: $s(x_i) = g(x_i) = y_i$

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad i = 1, ..., n$$

- Condition 2: We require the polynomial pieces to connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad i = 0, ..., n-1$$

- Condition 1: $s(x_i) = g(x_i) = y_i$

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad i = 1, ..., n$$

- Condition 2: We require the polynomial pieces to connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad i = 0, ..., n-1$$

- Condition 3: First and second derivative have to agree at x_i

$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad i = 1, ..., n-1$$

$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad i = 1, ..., n-1$$

- Condition 1: $s(x_i) = g(x_i) = y_i$

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad i = 1, ..., n$$

- Condition 2: We require the polynomial pieces to connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad i = 0, ..., n-1$$

- Condition 3: First and second derivative have to agree at x_i

$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad i = 1, ..., n-1$$

$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad i = 1, ..., n-1$$

- These are $4 * n - 2$ linear equations in $4 * n$ unknown a, b, c, d
- Two more conditions are needed

- Condition 1: $s(x_i) = g(x_i) = y_i$

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad i = 1, ..., n$$

- Condition 2: We require the polynomial pieces to connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad i = 0, ..., n-1$$

- Condition 3: First and second derivative have to agree at x_i

$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad i = 1, ..., n-1$$

$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad i = 1, ..., n-1$$

- These are $4 * n - 2$ linear equations in $4 * n$ unknown a, b, c, d
- Two more conditions are needed
- It's common to use $s'(x_0) = 0 = s'(x_n)$ (choice should depend on problem)

Cubic Splines

Some comments on cubic splines (not pchips)

- Cubic splines have very good convergence properties for smooth functions as the number of knots increase.
- Unfortunately, they respond to extreme data by introducing extra oscillation.
- Because of this, they are generally unsuitable for interpolating $V(k, e)$ in the k dimension because the extra oscillation introduces local minima/maxima.
- However, they may be very good for interpolating in other dimensions.

Spline Interpolation

When using splines in economics, the biggest problem is that we often want an interpolation method to guarantee that the interpolant is

1. positive
2. increasing
3. concave

whenever the data are.

Spline Interpolation

For this reason, the following two methods may be useful:

1. Schumaker's (1983) shape-preserving quadratic spline
2. A piecewise cubic hermite interpolating polynomial

The first guarantees that if the data are concave and/or increasing, then the interpolant will be. The second guarantees that if the data are increasing, then the interpolant will be (concavity is not guaranteed).

Projection Methods

Another way to accomplish similar things is through the use of projection methods. We'll get back to these soon.

Numerical Differentiation

Many methods we have discussed use derivatives. These can either be given analytically or computed by finite-differences.

Analytic derivatives have advantages and disadvantages:

Advantages

- Fast for the computer.
- Much more accurate than using finite-differences.

Disadvantages

- Extra work for the programmer.
- Error prone.

Numerical Differentiation

We can compute the derivative by finite differences. Recall the definition of the derivative:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

To approximate the derivative, we just take

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

for h small. The question is, how small should h be?

Numerical Differentiation

Taking h very small, for instance much less than εx where ε is machine precision, means that $x = x + h$! Very bad.

So, ensure $h \geq x\varepsilon$.

In fact, it's worse than this because $f(x+h) - f(x)$ for h close to 0 means many digits of accuracy are being lost.

However, taking h large means you are probably not near the limit.

Numerical Differentiation

The forward-difference approximation is

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

which is what we've been discussing.

However, we can also use central-differences (a two-sided formula)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}.$$

Two-sided differences tend to be more accurate, but aren't always. It depends on the relative magnitudes of the second and third derivatives. The extra function evaluations are often not worth it.

Numerical Differentiation

Note: sometimes, we may not know the derivative of the underlying object, but have an approximation whose derivatives are known.

Example: piecewise-polynomial approximations have

$$f(x) = a + bx + cx^2 + dx^3$$

and so

$$f'(x) = b + 2cx + 3dx^2$$

Very easy to get derivatives "analytically" in this case.

Numerical Differentiation

Computing second derivatives:
Sometimes, you may need second derivatives.
For instance, if you are doing Bayesian estimation, at some point (I am told) you'll want the inverse of the Hessian.
Bad way to get second derivatives:

- Compute $f'(x-h)$ and $f'(x+h)$ (using the values of f at $x-h, x, x+h$).
- Compute $f''(x) = (f'(x+h) - f'(x-h))/2h$.

The error in computing f' will get passed on to the f''.

Numerical Differentiation

It is better to proceed directly:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.$$

Question: Where does this formula come from?
Answer: Taylor expansion manipulation.

Descent methods

Note that Newton's method applied to $f(x) = 0$

- may not converge
- converges rapidly (if it converges)

On the other hand, a minimization of $SSRS(x) = \sum_i f_i(x)^2$

- always converges to something
- may converge to a local min
- may converge slowly

Descent methods

Descent methods try to take advantage of the strengths of both approaches:

- Given a point x_n, a search direction s_n is chosen. Methods differ in how s_n is chosen.
- Given s_n, the one-dimensional minimization problem

$$\min_{\lambda} SSRS(x) \text{ s.t. } x = x_n + \lambda s_n$$

  is solved (or a λ is found s.t. $SSRS(x_n + \lambda s_n) < SSRS(x_n)$).
- The new point is $x_{n+1} = x_n + \lambda_n s_n$ where λ_n is the argmin of the above problem.

Newton's method just sets $\lambda = 1$ and throws caution to the wind.

Descent methods

Note how this procedure always “works” in that it will converge to some x that has weakly smaller SSRS than your original point. However, there is no guarantee of finding a root.

Descent methods

Examples of descent methods:

- Newton with line-search: $s_n = -J(x_n)^{-1} f(x_n)$.
- Steepest/gradient descent: $s_n = -\nabla SSRS(x_n)$.
- Powell's hybrid method: s_n convex combination of Newton and gradient.
- Levenberg-Marquardt algorithm: s_n solves a "dog-leg" problem.

Powell's hybrid method

Powell's hybrid method

1. Initially computes the Jacobian by finite-differences.
2. Searches along a convex combination of the direction from Newton's method and the steepest descent method.
3. Updates the Jacobian using information from computed function values (using Broyden's method).
4. Recomputes the Jacobian whenever satisfactory progress is not being made.

Powell's hybrid method

Implementations

Fortran

Powell's hybrid method is implemented in MINPACK (available at `http://www.netlib.org/minpack/`).
In particular, `hybrd1.f` is the primary routine for solving $f(x) = 0$ without an analytic Jacobian.

Note: MINPACK also implements the Levenberg-Marquardt algorithm in `lmdif1`.

Matlab

Matlab's `fsolve` can be configured to use Powell or the Levenberg-Marquardt algorithm, but the default is Powell's hybrid method (they call it the "Trust-Region Dogleg Method").