AMSC/CMSC 460 Computational Methods, Fall 2007
UNIT 5: Nonlinear Equations
Dianne P. O'Leary

Solving Nonlinear Equations and Optimization Problems

Read Chapter 8. Skip Section 8.1.1. Skim Sections 8.2.1 and 8.2.2.

Motivation:

- We have already seen an example of a system of nonlinear equations when we studied Gaussian integration (p.8 of integration notes)

$$\begin{aligned} I(1) &= 2 = \omega_1 + \omega_2 + \omega_3 \\ I(t) &= 0 = \omega_1 t_1 + \omega_2 t_2 + \omega_3 t_3 \\ I(t^2) &= \frac{2}{3} = \omega_1 t_1^2 + \omega_2 t_2^2 + \omega_3 t_3^2 \\ I(t^3) &= 0 = \omega_1 t_1^3 + \omega_2 t_2^3 + \omega_3 t_3^3 \\ I(t^4) &= \frac{2}{5} = \omega_1 t_1^4 + \omega_2 t_2^4 + \omega_3 t_3^4 \\ I(t^5) &= 0 = \omega_1 t_1^5 + \omega_2 t_2^5 + \omega_3 t_3^5 \end{aligned}$$

- Suppose we want to find the zeros (or roots) of a polynomial; that is, we want to find values x for which $p(x) = 0$, where p might be the polynomial

$$p(x) = x^5 + 3x^3 + 2x + 4\,.$$

  Galois in 1830 proved that there is no finite sequence of rational operations plus square and cube roots that can solve this problem for every polynomial of degree 5 or higher. So we'll need a numeric algorithm.

- Suppose we have a nonlinear least squares problem: for example, we believe our data should fit the model

$$f(t) = a_1 e^{b_1 t} + a_2 e^{b_2 t}$$

  where a_1, a_2, b_1, b_2 are unknown. If we measure 20 data points (t_i, f_i), $i = 1, \ldots, 20$, we can try to minimize the sum of squares of distances between the data points and the model's predictions:

$$\min_{a_1,a_2,b_1,b_2} \sum_{i=1}^{20} (f(t_i) - f_i)^2\,.$$

The plan

- Methods for solving one nonlinear equation with one variable (8.1):
  - bisection method
  - Newton method
  - Newton variants.
  - Practical algorithm: `fzero`
- Methods for solving systems of nonlinear equations (8.4)
- Methods for minimizing a function of a single variable (8.2)
- Method for minimizing multivariate functions (8.3)

Note 1: If we try to minimize a differentiable function $F(x)$, then calculus tells that a necessary condition for x^* to be a minimizer is that

$$F'(x^*) = 0\,.$$

Thus, if we can solve nonlinear equations, then we can solve minimization problems, although each candidate solution must be checked to see that it is a minimizer instead of a maximizer or an inflection point.

Note 2: The solution $\mathbf{x}$ to a system of equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is called a zero of the system.

Note 3: Van Loan's section 2.4.2 discusses inverse interpolation, one way to get an approximate zero of a function.

One Nonlinear Equation, One Variable

One Nonlinear Equation, One Variable

Problem: Given a continuous function $f(x)$, with as many continuous derivatives as we need for a particular method, find a point x^* so that $f(x^*) = 0$.

Bisection Method

Suppose we are given two points a and b so that $f(a)f(b) < 0$. Then, since f has opposite signs at the endpoints of the interval $[a, b]$, and f is continuous, there must be a point $x^* \in [a, b]$ so that $f(x^*) = 0$.

Let's guess that that point is the midpoint $m = (a + b)/2$. Then one of three things must be true:

1. $f(m) = 0$, so $x^* = m$ and we are finished.
2. $f(m)$ has the same sign as $f(a)$. Then there must be a root of $f(x)$ in the interval $[m, b]$, which has half the length of our previous interval.
3. $f(m)$ has the same sign as $f(b)$, so there is a root in $[a, m]$.

After 1 iteration, our interval has been reduced by a factor of $1/2$. After k iterations, it has been reduced by a factor of $1/2^k$, so we terminate when the interval is small enough.

See the code on p. 280 of Van Loan.

Advantages: This is

- a foolproof algorithm,
- simple to implement,
- with a guanteed convergence rate.
- doesn't require derivative evaluations.
- cost per iteration = 1 function value: very cheap.

Disadvantages:

- Slow convergence. We would like to do better.

Newton and Its Variants

Let's use our polynomial approximation machinery. Suppose we have the same information as in bisection: $(a, f(a))$,$(b, f(b))$. Bisection predicts the root to be the midpoint $(a + b)/2$. Is there a better prediction?

Suppose we form the linear polynomial that interpolates this data:

$$p(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a)\,.$$

Then a prediction of the point where $f(x) = 0$ is the point were $p(x) = 0$, and this is

$$x = a - \frac{b-a}{f(b)-f(a)} f(a).$$

This gives us the finite difference Newton algorithm:

Choose an initial guess a. Find a nearby point b and use the formula to predict a new point x. Replace a by x and repeat.

This algorithm requires 2 function values per iteration.

In the secant algorithm, the nearby point is chosen to be the next-newest guess. This reduces the cost to 1 function value per iteration.

In the Newton method, we replace the formula

$$x = a - \frac{b-a}{f(b)-f(a)} f(a).$$

by

$$\begin{aligned} x &= a - \lim_{b \to a} \frac{b-a}{f(b)-f(a)} \; f(a) \\ &= a - f(a)/f'(a). \end{aligned}$$

The cost per iteration is 1 function value and one derivative value per iteration.

How fast is Newton?

Your book on p.285 presents a theorem showing that if

- f' doesn't change sign in a neighborhood of x^*,
- f is not "too nonlinear"
- and we start the iteration close enough to x^*

then the convergence rate is quadratic:

$$|x - x^*| \leq c|a - x^*|^2$$

where c is a constant. This is very fast convergence:
If our initial error is $1/2$,
then after 1 iteration it is at most $1/4$,
after 2 iterations, at most $1/16$,
after 3 iterations, at most $1/256$, etc.

Advantages:

- When Newton works, it converges very quickly.

Disadvantages:

- It requires evaluation of the derivative, and this is sometimes expensive.
- It is notoriously unreliable.

How fast is the Secant Method?

If the method converges, then

$$|x - x^*| \leq c|a - x^*|^\alpha$$

where

$$\alpha = \frac{1 + \sqrt{5}}{2} \approx 1.6\,.$$

This intriguing number is related to the Fibonacci series, and the proof of the result is interesting, but we'll skip it.

Fzero

We want to combine the reliability of bisection with the speed of Newton variants.

Idea:

- We first determine an interval $[a, b]$ with $f(a)f(b) < 0$.
- At each iteration, we reduce the length of this interval in one of two ways:
  - Compute the secant guess s. If this gives us a new interval $[a, s]$ or $[s, b]$ that has length not much longer than half the old length, then we accept it.
  - Otherwise we use bisection at this iteration.

This algorithm is a variant of one by Dutch authors including Dekker, and improved by Richard Brent and others.

Systems of Nonlinear Equations

First let's use an example to illustrate some of the basics.

Example: Let the function $\mathbf{f}(\mathbf{x})$ be defined by

$$\begin{aligned} f_1(x_1, x_2) &= x_1^3 + \cos(x_2) \\ f_2(x_1, x_2) &= x_1 x_2^2 - x_2^3 \,. \end{aligned}$$

Then in solving $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, we look for a point x_1, x_2 where both f_1 and f_2 are zero.

The derivative of the function is called the Jacobian matrix. It is a matrix $\mathbf{J}(\mathbf{x})$ defined by

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 3x_1^2 & -\sin(x_2) \\ x_2^2 & 2x_1 x_2 - 3x_2^2 \end{bmatrix}$$

[]

Example: See `nonlindemo.m`

Newton's Method

For a single variable, Newton's method looked like

$$x = a - f(a)/f'(a), .$$

It's geometric interpretation is that we follow the tangent line at a until it hits zero.

For the multivariate case, we follow the tangent plane. The computation involves the inverse of the Jacobian matrix instead of the inverse of the derivative:

$$\mathbf{x} = \mathbf{a} - \mathbf{J}(\mathbf{a})^{-1}\mathbf{f}(\mathbf{a}) \,.$$

To implement this, we need Matlab functions to evaluate both the function and the Jacobian matrix.

Since we know that we (almost never) invert a matrix, we actually implement the algorithm by solving the equation

$$\mathbf{J}(\mathbf{a})\mathbf{p} = -\mathbf{f}(\mathbf{a})$$

for $\mathbf{p}$ and then forming

$$\mathbf{x} = \mathbf{a} + \mathbf{p} \,.$$

If the Jacobian is too hard or too expensive...

... Then we can replace it by a finite difference approximation:

$$\frac{\partial f_i}{\partial x_j}(\mathbf{x}) \approx \frac{f_i(\mathbf{x} + h\mathbf{e}_j) - f_i(\mathbf{x})}{h}$$

where h is a small scalar and $\mathbf{e}_j$ is the jth column of the identity matrix, $j = 1, \ldots, n$.

Note that this requires an extra n function evaluations.

Minimizing a Function of a Single Variable

Minimizing a Function of a Single Variable

Caution: We look for local minimizers, the "best in their neighborhood" rather than global minimizers, the "best in the world".

For nonlinear equations we had Bisection; for function minimization we have Golden Section Search. We won't discuss the details; just know that it involves evaluating the function at a pattern of points and choosing the best. The interval containing the root is reduced by a factor of approximately .618 each iteration, a linear rate of convergence

For nonlinear equations we had Newton's method; for function minimization we have Parabolic Fit. This involves fitting a quadratic function (a parabola) to three points and approximating the minimizer of the function by the minimizer of the parabola. The convergence rate is superlinear, better than linear, but the method is not as reliable as Golden Section.

---

fminbnd

This is the Matlab function that minimizes functions of single variables. Just like Fzero, it combines a slow, reliable algorithm (Golden Section) with a fast but unreliable one (Parabolic Fit) to obtain a fast, reliable algorithm.

(The book may talk about an older version of the function, called fmin.)

Minimizing Multivariate Functions

Minimizing Multivariate Functions

First let's use an example to illustrate some of the basics. Suppose we want to solve

$$\min_{\mathbf{x}\in\mathcal{R}^2} \mathbf{F}(\mathbf{x})$$

where

$$\mathbf{F}(\mathbf{x}) = x_1 x_2 + e^{x_1 x_2}$$

Then the gradient (derivative) of the function $\mathbf{F}(\mathbf{x})$ is

$$\mathbf{g}(\mathbf{x}) = \nabla\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \frac{\partial\mathbf{F}(\mathbf{x})}{\partial x_1} \\ \frac{\partial\mathbf{F}(\mathbf{x})}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 + x_2 e^{x_1 x_2} \\ x_1 + x_1 e^{x_1 x_2} \end{bmatrix}$$

and the second-derivative matrix, the Hessian of the function, is also the Jacobian of the gradient:

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2\mathbf{F}(x)}{\partial x_1^2} & \frac{\partial^2\mathbf{F}(x)}{\partial x_1 \partial x_2} \\ \frac{\partial^2\mathbf{F}(x)}{\partial x_2 \partial x_1} & \frac{\partial^2\mathbf{F}(x)}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} x_2^2 e^{x_1 x_2} & 1 + x_1 x_2 e^{x_1 x_2} + e^{x_1 x_2} \\ 1 + x_1 x_2 e^{x_1 x_2} + e^{x_1 x_2} & x_1^2 e^{x_1 x_2} \end{bmatrix}.$$

Important note: Programming derivatives, gradients, Jacobians, and Hessians is tedious and error-prone. There are automatic systems to derive these functions from your code for function evaluation. One of the best is called `Adifor`, and there are also packages for use with Matlab.

The Steepest Descent Algorithm

The gradient is the direction of steepest ascent; i.e., if you are standing on a mountain in the fog, then the steepest uphill direction is in the direction of the gradient $\mathbf{g}(\mathbf{x})$, and the steepest downhill direction is in the opposite direction $-\mathbf{g}(\mathbf{x})$.

This insight leads to the steepest descent algorithm: Take a step in the direction of the negative gradient:

- Either step that way until the function stops decreasing (you start going uphill on the mountain)
- or take a fixed length step, as long as it does not bring you too far uphill.

Advantages:

- The algorithm is relatively easy to program.
- It requires no 2nd derivatives.

Disadvantages:

- The algorithm is slow!

Alternatives to Steepest Descent

Your book really does not discuss any good algorithms for minimizing general functions of several variables. Here is some advice:

1. If gradients are available, the best algorithms are conjugate gradients or, if n^2 storage is also available, Quasi-Newton algorithms.
2. If Hessians are available, a quality implementation of Newton's method is a good alternative.
3. If no derivatives are available, a pattern search method is usually used. The Nelder-Meade simplex algorithm (different from the simplex algorithm for linear programming) is popular and is available in Matlab as `fminsearch`, but more recent pattern search methods are better.

If the Minimization Problem is Nonlinear Least Squares...

... then a very good alternative is the Gauss-Newton algorithm, which applies a Newton-like method to finding a zero of the gradient. Instead of computing the Hessian matrix exactly, we approximate it by $\mathbf{J}^T\mathbf{J}$, where $\mathbf{J}$ is the Jacobian matrix for the residual vector

$$\begin{bmatrix} f(x_1) - f_1 \\ \cdots \\ f(x_m) - f_m \end{bmatrix}$$

This method works well as long as the model is a good fit to the data.

Final Words

- $f(a)f(b) < 0$ is not a foolproof test. Why? Hint: underflow/overflow
- To find complex roots, can use Newton or secant with complex initial guesses.

- If you are finding the roots of a polynomial, make sure that it is not an eigenvalue problem

$$det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

  in disguise. If it is, you will almost surely get better results by working with the original matrix and using `eig` to find the eigenvalues.

- The multivariate methods discussed in the book are linesearch methods. Trust region methods are newer and equally useful.