Class 8: Roots and minima, more C++, and using external libraries

Overview

- Root-finding algorithms in 1-d and multiple dimensions
- Minima-finding algoritms in 1-d and multiple dimensions
- A little more C++: derived classes
- Two examples of external libraries for minimization: GSL and ROOT

Root-finding:

Finding a root of a function (1-d): Solve $f(x) = 0$ for x.

- Without using the derivative of f, root can be found by bisection.
- With the derivative f', root can be found using Newton's method.

Finding root(s) of N functions of N variables: Solve $\underline{f}(\underline{x}) = 0$ for $\underline{x}$.

- Newton's method works here, generalized to N dimensions.
- A related but somewhat different problem: finding N-1 dimensional contours or surfaces satisfying $f(\underline{x}) = 0$, where f is a single function, or (N-M)-dimensional contours where M functions are zero.

***Numerical Recipes* describes other algorithms, and gives lots of** examples and explanations.

Helping the algorithms get the right answer

Give the algorithm a good starting point.

For 1-d root-finders, provide a bracket for the root.

- The bracket is two numbers defining the interval in which to find the root.
- A good bracket (a, b) is one in which the function has opposite sign at the two end: $f(a) < 0$ and $f(b) > 0$ or $f(x) > 0$ and $f(b) < 0$.

For the multi-dimensional case, bracketing isn't feasible. The algorithms generally need

- A good starting point $\underline{x}_0$ not too far from the root.
- An initial step size "hint" that is small enough not to overshoot the local minimum, but large enough to see some change in the functions.

Bisection algorithm

Shrink bracket until the root is pinned down, as follows:

```
Start with caller-provided bracket for the root $(a, b)$, tolerance $\epsilon$, and function $f$.
Check $sign(f(a)) \neq sign(f(b))$.
Loop:
    $x \leftarrow (a+b)/2$
    $y \leftarrow f(x)$
    If $y$ has same sign as $f(a)$, then
        $a \leftarrow x$
    else
        $b \leftarrow x$
Repeat until $|a - b| < \epsilon$
```

Newton's method

Repeatedly "shoot for the zero," using the approximation

$$f(x+d) \simeq f(x) + f'(x)d + O(f''d^2) = 0$$

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

Potential problems can occur when far from the root:

- Big overshoot if $|f'(x)|$ is small.
- Step in wrong direction if $f'(x)$ has wrong sign.
- Non-converging loop if $|f'(x)|$ gets shallower farther from the root.

(... Draw figures on board, insert in notes later ...)

Simple Newton's algorithm

```
Start with caller-supplied $f$ and $f'$ functions, initial $x$, and tolerance $\epsilon$.
Loop:
    $d \leftarrow -\frac{f}{f'}$
    $x \leftarrow x + d$
Repeat until $|d| < \epsilon$.
```

Important note: ϵ should be small enough for your application, but no smaller. Make sure it is large enough that $x + \epsilon$ is different from x in your machine's floating point representation.

Smarter Newton's algorithm

Use bracketing to protect against any craziness, bisect whenever a Newton's method step fails.

```
Start with caller-supplied $f$ and $f'$ functions, bracket $(a, b)$, and tolerance $\epsilon$.
Set initial $x \leftarrow (a + b)/2$.
Loop:
    $d \leftarrow -\frac{f}{f'}$
    $x_{\text{try}} \leftarrow x + d$
    Is $x_{\text{try}}$ in range $(a, b)$?
        If yes: $x \leftarrow x_{\text{try}}$
        If not: bisect range and try again
Repeat until $|d| < \epsilon$.
```

See full example in *Numerical Recipes* (sec 9.4 in 2nd or 3rd ed.).

Multidimensional case

We have N functions F_i of N variables x_j, and want to find the zeros.

There's a great illustration in *Numerical Recipes* showing why this is hard in general. (Fig. 9.6.1.) If you want to seek the nearest zero to some point, Newton's method should work.

$$F_i(\underline{x} + \underline{d}) = F_i(\underline{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j} d_j + O(d^2).$$

In matrix notation,

$$\underline{F}(\underline{x} + \underline{d}) = \underline{F}(\underline{x}) + \underline{\underline{J}} \cdot \underline{d} + O(d^2),$$

using the Jacobian matrix

$$\underline{\underline{J}} = [J_{ij}] \equiv \left[\frac{\partial F_i}{\partial x_j} \right].$$

To find the root, repeatedly solve

$$\underline{\underline{J}} \cdot \underline{d} = -\underline{F}.$$

Update $\underline{x}$ until convergence $|\underline{d}| < \epsilon$.

***Numerical Recipes* describes a method for "backtracking" if a** multi-dimensional Newton's method step makes things worse instead of better.

Minimization

Finding a local minimum: find value for which $f(x)$ or $f(\underline{x})$ is locally minimized.
(To find maxima, just minimize $-f(x)$.)

I will describe one particularly pretty 1-d minimization algorithm that requires no calculation of derivatives and that is *guaranteed* to find a local minimum.

There are faster algorithms that use derivatives, for 1-d and multi-d problems, including ones that only need the user-supplied function. (These algorithms estimate the derivatives themselves.)

I'll describe the general properties of these, and then we will talk about how to use pre-written implementations.

Golden section search

Similar to bisection, except it starts with a triplet of values (a, b, c) such that f at the middle point is less than at the two ends: $f(b) < f(a)$ and $f(b) < f(a)$.

(... figure ...)

General outline of algorithm:

```
Start with $(a, b, c)$, function $f$, and tolerance $\epsilon$
Loop:
    Pick a new point $x$ and try it (*).
    If it's lower than the current lowest point, then
        use it as the new center point, use the old $b$ as edge of bracket
    else
        adjust the end of the bracket
until $|c - a| < \epsilon$.
```

(*) The point x is chosen in the range (a, b) or (b, c) such that the updated triplet will have the same proportions as the original triplet. The ideal ratio turns out to be the *golden mean.*

More general minimization

Near minimum, to 2nd order,

$$f(\underline{x} + \underline{d}) = f(\underline{x}) + \sum_j \frac{\partial f}{\partial x_j} d_j + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_j \partial x_i} d_i d_j$$

$$= c + \underline{b} \cdot \underline{d} + \frac{1}{2} \underline{d}\ \underline{\underline{A}}\ \underline{d}$$

Terminology: The matrix $\underline{\underline{A}}$ is called the *Hessian.* The vector $\underline{b}$ is the gradient.

$$\underline{\nabla} f = \underline{\underline{A}} \cdot \underline{d} - \underline{b}$$

At an extremum, $\underline{\nabla} f = 0$. So we just solve

$$\underline{\underline{A}} \cdot \underline{d} = \underline{b}.$$

Essentially the same algorithm as Newton's method works if we know $\underline{\underline{A}}$ and $\underline{b}$ well enough.

Types of minimization algorithms

Various algorithms differ in how they guard against crazy steps and whether they need the caller to provide a function for the derivatives or Hessian.

- If we have functions for the 2nd derivatives ($\underline{\underline{A}}$) and 1st derivatives, we can use an algorithm that uses them.
- If we have just the first derivatives, we need an algorithm that estimates the Hessian using the derivatives.
- If we have only the function, we need an algorithm that estimates both the derivatives and the Hessian using finite differences.

What's common among the algorithms:

- All algorithms ultimately seek $\underline{\nabla} f = 0$.
- All need an initial starting point $\underline{x}$ not too far from the minimum being sought.
- A step size parameter tells the algorithm what steps are reasonable for doing finite differencing and making the first few trial steps.

Numerical Recipes has many different algorithms.
More importantly, there are existing libraries for minimization of functions.

Break

A little more C++: derived classes

C++ lets you define a new class based on an existing class, like this:

```
class NewClass : public BaseClass {
  // usual class definition
};
```

This defines `NewClass` as having all the same data and member functions as `BaseClass`, with the following enhancements:

- Add any new data members defined in `NewClass`.
- Add any new function members declared in `NewClass`.
- If a virtual function member declared in `BaseClass` is redeclared in `NewClass`, the hidden virtual function pointer point to the new function for `NewClass` objects.
- If you start the definition with `class NewClass : private BaseClass`, then the base class functions will be made `private` in the derived class, even if they were `public` in `BaseClass`.

This is not impossible in C, just clunky

C++	C
`class MyBase {`	`struct MyBase_s   {`
`  int i;`	`  int i;`
`public:`	
`  virtual void set_i(int ii);`	`  void (*set_i)(int ii);`
`};`	`};`
`class MyClass2 : public MyBase {`	`struct MyClass2_s {`
	`  strict MyBase_s base;`
`  int j;`	`  int j;`
`public:`	
`  virtual void set_i(int ii);`	
`  virtual void set_j(int jj);`	`  void (*set_j)(int jj);`
`};`	`};`
C++ constructor sets virtual function table for new MyClass2 object's set_i and set_j to the correct addresses for MyClass2::set_i() and MyClass2::set_j().	For any new MyClass2_s variable made, C programmer has to set the values of .set_j and .base.set_i pointers to point to the right functions, and i field must be accessed as base.i.

Example of minimizer in C using object-oriented approach: GSL

See very nice documentation in *GNU Scientific Library Reference Manual*, section titled "Multidimensional Minimization". http://www.gnu.org/software/gsl/manual/html_node/Multidim
Minimization.html

Things to note about how they write documentation:

- The especially nice write-up of the problem being solved and the algorithm properties.
- The especially nice write-up of how to use the functions.
- The really useful examples subsection.

Things to note about the API:

- It's C, but object oriented, using pointers to structs and functions.

Example of a minimizer in C++: ROOT

The best documentation from the authors is at http://root.cern.ch/root/html526/TMinuitMinimizer.html

- It's very detailed at the low level, and has some high level overview.
- However, there are some critical things missing at the middle level, such as a clear statement of how to make it actually start minimizing.

  - It turns out one of the 49 functions defined for the class is named `Minimize()`, and that's the one to make it go, after setting up using `SetFunction()` and `SetVariable()`.

- On their website, I found lots of examples of fitting, but none of pure minimization, so I wrote one for you. (See link in course web page.)
  - The example finds the minimum of the function $f(\alpha, x, y) = -\alpha^{-1/2} e^{-(1+x^2+y^2)/(2\alpha^2)}$.

Assignment (easy?)

Get the ROOT examples from the course web page to compile and run.