1.204 Lecture 22

Unconstrained nonlinear optimization:
   Amoeba
   BFGS
Linear programming: Glpk

Multiple optimum values

Heuristics to deal with multiple optima:
• Start at many initial points. Choose best of optima found.
• Find local optimum. Take a step away from it and search again.
• Simulated annealing takes ‘random’ steps repeatedly

From Press
Nonlinear optimization

• Unconstrained nonlinear optimization algorithms generally use the same strategy as unconstrained
  – Select a descent direction
  – Use a one dimensional line search to set step size
  – Step, and iterate until convergence

• Constrained optimization used the constraints to limit the maximum step size
  – Unconstrained optimization must select maximum step size
  – Step size is problem-specific and must be tuned

• Memory requirements are rarely a problem
  – Convergence, accuracy and speed are the issues

Family of nonlinear algorithms

• Amoeba (Nelder-Mead) method
  – Solves nonlinear optimization problem directly
  – Requires no derivatives or line search
  – Adapts its step size based on change in function value

• Conjugate gradient and quasi-Newton methods
  – Require function, first derivatives* and line search
  – Line search step size adapts as algorithm proceeds

• Newton-Raphson method (last lecture)
  – Used to solve nonlinear optimization problems by solving set of first order conditions
  – Uses step size $dx$ that makes $f(x+dx)= 0$. Little control.
    • ‘Globally convergent’ Newton variant has smaller step size
  – Needs first and second derivatives (and function)
Choosing among the algorithms

- Amoeba is simplest, most robust, slowest
  - “Crawls downhill with no assumptions about function”
  - No derivatives required
- Conjugate gradient (Polak-Ribiere) (not covered)
  - Need first derivatives
  - Less storage than quasi-Newton but less accuracy
- Quasi-Newton (Davidon-Fletcher-Powell or Broyden-Fletcher-Goldfarb-Shanno)
  - Standard version uses first derivatives
  - Variation computes first derivatives numerically
  - Better than conjugate gradient for most problems
- Newton-Raphson
  - Needs function, first and second derivatives
  - Simplest code but not robust or flexible
  - Use amoeba if you want a simple approach

Amoeba algorithm

- The easiest algorithm for unconstrained nonlinear optimization is known as Nelder-Mead or “the amoeba”
- It is very different
  - It requires only function evaluations
    - No derivatives are required
  - It is less efficient than the line search algorithms
    - But it tends to be robust (line methods are temperamental)
  - It is short (~150 lines) and relatively easy to implement
  - Works in problems where derivatives are difficult:
    - Fingerprint matching
    - Models of brain function
  - We’ll use logit demand model estimation as test case for all the algorithms today
Amoeba steps

- Simplex is volume defined by n+1 points in n dimensions
  - In 3-D, it is a tetrahedron or pyramid
- Select a starting guess at point $P_0$
  - Set other points of simplex as $P_i = P_0 + \Delta e_i$
- Take a series of steps
  - Most steps move point of simplex where function is highest ("highest point"): "reflection"
    - Conserve volume of simplex -> avoid degeneracy
  - Where function is flat, method expands simplex to take larger steps: "expansion and reflection"
  - When it reaches a "valley floor", simplex "contracts" itself in transverse direction and tries to ooze down the valley
  - If trying to pass through "eye of needle" it "contracts itself in all directions" around its lowest (best) point

From Press et al.

Possible outcomes for a step in the downhill simplex method
**Amoeba pseudocode: minimization**

- **Start at initial guess**
  - Determine which point is highest by looping over simplex points and evaluating function at each
  - If difference between highest and lowest is small, return
- **Otherwise ooze (iterate):**
  - Reflect by factor= -1 through face of simplex from high point
    - If this is better than low point, reflect/expand by factor of 2
    - If this is worse than second highest, contract by 2 in this direction
    - If this is worst point, contract in all directions around lowest point
  - Select new face based on new high point and reflect again
  - Terminate if difference between highest and lowest points is small
    - Or terminate at maximum iterations allowed

**Amoeba termination criteria**

- **All nonlinear optimization algorithm termination criteria are difficult**
  - Terminate when step size is small (~10^-8 with doubles)
  - Change in function value is small (~10^-14 with doubles)
- **Termination can occur at a local minimum**
  - Restart amoeba at a minimum it found
    - Reset initial simplex using \( P_i = P_0 + \Delta e_i \),
    - Don’t use simplex that existed at termination
- **Code is in download:**
  ```java
  public void Amoeba(double ftol)
  public double[] minimize(double[] point, double del,
  MathFunction3 mf3)
  // And other optional methods
  ```
  - This is very easy; code ran first time on demand model
    - Experiment with starting point and del, but it’s usually easy
AmoebaTest

```java
public class AmoebaDemandTest {
    public static void main(String[] args) {
        double[][] x = { {1, 52.9 - 4.4},
                        {1, 4.1 - 28.5}, // Etc. }
        double[] y = {0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0};
        // Almost identical to DemandModel from last lecture
        // Implements MathFunction3, which has method func2()
        // func2() returns -logLikelihood()
        DemandModelAmoeba d = new DemandModelAmoeba(x, y);
        double[] beta = {0, 0}; // initial guess
        double log0 = d.func2(beta);
        double[] initialPoint = {0.0, 0.0};
        double initialDelta = 0.1;
        double ftol = 1E-14;
        Amoeba a = new Amoeba(ftol);
        beta = a.minimize(initialPoint, initialDelta, d);
        double logB = d.func2(beta);
        double[][] jacobian = d.jacobian(beta);
        d.print(log0, logB, beta, jacobian);
    }
} // Output identical to last lecture example. 116 evals
```

Quasi-Newton methods (DFP, BFGS)

- We used a similar method to BFGS in constrained optimization:
  - Find derivatives
  - Find direction that yields maximum estimated objective function change
  - Use line search to find optimal step size
  - Move, and repeat
- DFP and BFGS are essentially the same
  - One additional 'correction term' in BFGS
  - Treatment of roundoff errors, tolerances is different
  - Empirically, it seems BFGS is better than DFP
Quasi-Newton methods

Split problem into two minimization problems, each with a “quadratic envelope”
You must understand your problem well enough to do this

Newton method

If the minimum region is narrow or twisty, it is hard to get down into it.
Full Newton steps give little control. (Amoeba just munches along)
Quasi-Newton methods

BFGS takes a fraction of the Newton step, so that $f(x)$ decreases at some minimum rate proportional to the average decrease.

High order derivatives

We may be tempted to use higher derivatives to fit our nonlinear functions. More terms in the Taylor series expansion mean a higher degree polynomial fit, but using higher order derivatives is difficult because they are “stiff.”

This is the challenge of nonlinear methods: use only low order derivatives to explore complex surfaces. There aren’t many general ways to do this. Solutions are problem-specific but use BFGS or other general algorithm:

- Understand problem regions;
- Have good starting point;
- Understand surface.
We may be tempted to use higher derivatives to fit our nonlinear functions. More terms in the Taylor series expansion mean a higher degree polynomial fit, but using higher order derivatives is difficult because they are “stiff”.

This is the challenge of nonlinear methods: use only low degree derivatives to explore complex surfaces. There aren’t many general ways to do this. Solutions are problem-specific but use BFGS or other general algorithm:
Understand problem regions; have good starting point; understand surface
High order derivatives

We may be tempted to use higher derivatives to fit our nonlinear functions. More terms in the Taylor series expansion mean a higher degree polynomial fit, but using higher order derivatives is difficult because they are “stiff.”

This is the challenge of nonlinear methods: use only low degree derivatives to explore complex surfaces. There aren’t many general ways to do this. Solutions are problem-specific but use BFGS or other general algorithm:
Understand problem regions; have good starting point; understand surface

BFGS

Take a point P as origin. Approximate f using Taylor series:

\[ f(x) = f(P) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i x_j} x_i x_j \]

\[ \approx c - b \cdot x + \frac{1}{2} x \cdot A \cdot x \]

where
\[ c = f(P) \]
\[ b = -\nabla f \bigg|_P \]
\[ [A]_{ij} = \frac{\partial^2 f}{\partial x_i x_j} \bigg|_P \]

A is the Hessian matrix. Gradient of f is:
\[ \nabla f = A \cdot x - b \]
BFGS, p.2

BFGS constructs a sequence of matrices $H_i$ such that:

$$\lim_{i \to \infty} H_i = A^{-1}$$

At point $x_i$, using a Taylor series again

$$f(x) = f(x_i) + (x - x_i) \cdot \nabla f(x_i) + \frac{1}{2} (x - x_i) \cdot A \cdot (x - x_i)$$

To find a minimum, find a zero of the gradient of $f(x)$:

$$\nabla f(x) = \nabla f(x_i) + A \cdot (x - x_i)$$

Newton's method sets $\nabla f(x) = 0$ to find the next point:

$$(x - x_i) = -A^{-1} \cdot \nabla f(x_i)$$

BFGS, p.3

- Newton’s method, far from the minimum, can project us to points $x$ where $f(x)$ is greater than the current value
  - Large steps based on a quadratic approximation will not necessarily lead to improvements in ill-behaved function
  - BFGS does a line search along the Newton direction
  - It finds a point at which the objective has decreased
  - This point is used to update the Hessian (which is a matrix of second derivatives)
  - The Hessian is based on the previous Hessian and a set of correction terms based on $\nabla f, x$ and $x_i$
  - See Numerical Recipes for BFGS updating formula
- Proofs are difficult
BFGS, p.4

• The series of $H_i$ converge to $A^{-1}$ while guaranteeing movement downhill
  - Convergence takes $N$ steps if $f$ is quadratic
• The algorithm is sensitive to variable scaling:
  - Step length must be tuned
  - Scale variables to stay in “low” range of double values
• The algorithm is not too sensitive to the accuracy of the line search

BFGS dfpmin() pseudocode

• Compute function, gradient at start point $P$
• Initialize inverse Hessian to identity matrix
• Find initial descent direction along gradient
• Loop until convergence
  - Line search along direction to find decrease (not min)
  - Update gradient and line direction
  - Check if change in $x$ is small enough to converge
  - Check if gradient is close enough to 0 to converge
  - Compute difference in gradients, $x$ for BFGS update
  - Calculate next direction
• If iteration limit reached, terminate
BFGS Insrch() pseudocode

- Takes point, function and gradient value at point, and direction (line) to search along as input
- Finds new point with lower function value
  - Finding minimum along line requires too much computation
  - Finding any point whose f is less than current isn’t good enough either; we may take too many steps
  - We find a point where the average decrease from the current point is a fraction of the gradient projection
  - We require a minimum step size so we don’t stall
  - We use a quadratic interpolation of f along the line to choose the first new point
  - We then use a cubic interpolation, now that we have 3 points (f(0), f(1) and f(m)) for remaining points

BFGS code

- First code set
  - BFGS: constructor, dfpmin(), Insrch()
  - MathFunction4: interface with func(), df() [gradient]
  - DemandModelBFGS: Almost same as in Newton
  - DemandModelBFGSTest: Almost same as in Newton
- Second code set
  - Same BFGS, MathFunction4, DemandModelBFGSTest
  - DemandModelBFGS df() method computes gradient numerically, without analytic expressions for derivatives
- Both BFGS implementations converge quickly on same logit model coefficients as Newton
  - Code is subtle, especially interaction between dfpmin() and Insrch().
BFGSTest

```java
public class DemandModelBFGSTest {
    public static void main(String[] args) {
        double[][] x = {{1, 52.9 - 4.4},
                        {1, 4.1 - 28.5}, // Etc. }
        double[] y = {0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1};
        // Almost identical to DemandModel from last lecture
        // Implements MathFunction4, with func() and df()
        DemandModelBFGS d = new DemandModelBFGS(x, y);
        double[] beta = {0, 0}; // Initial guess
        double log0 = d.func(beta);
        double gtol = 1E-14;
        BFGS b = new BFGS(d, gtol);
        double logB = b.dfpmn(beta);
        double[][] jacobian = d.jacobian(beta);
        d.print(log0, logB, beta, jacobian);
    }
} // Output identical to last lecture example. 10 iterations
```

Nonlinear method performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Iterations</th>
<th>Function evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>6</td>
<td>$6n^2 = 24$</td>
</tr>
<tr>
<td>BFGS</td>
<td>10 or 11</td>
<td>$10n^2 = 20$</td>
</tr>
<tr>
<td>Amoeba</td>
<td>116</td>
<td>116</td>
</tr>
</tbody>
</table>

This example is illustrative only, based on our logit test case. n is the number of variables (coefficients in our test case). As n increases, amoeba performance is relatively more competitive. Amoeba and BFGS have better convergence and precision than Newton.
Summary: nonlinear continuous methods

- Note the similarities in solving discrete and continuous problems
  - Starting solution is null or \{0, 0, 0, \ldots\}
    - Or a good guess or greedy if we need a good first guess
  - Choose initial decision or initial direction/gradient
  - Find initial solution at stage/branch or Newton/other step
  - Look for better solution locally through greedy/branching/stages or new gradient
  - Or globally through branch-and-bound/dynamic programming or with advanced continuous methods
    - Simulated annealing, multiple restarts
  - Terminate when:
    - Local or global optimum found, or
    - Upper and lower bound close enough, or
    - Maximum iterations reached

Linear programming

- Linear programming has a linear objective function and linear constraints
- Used as a subproblem several times already:
  - Convex combinations (lecture 18)
  - Homework 8 transit network optimization
  - Network equilibrium used shortest path algorithm as a special case of linear programming
  - Often used as subproblem in branch and bound algorithms
    - We used shortest paths in our warehouse branch and bound as a special case of linear programming
    - See any integer programming text for a list of problems that use linear programming as a subproblem
Linear programming

\[
\begin{align*}
\text{maximize} & \quad Z = 10x_1 + 6x_2 + 4x_3 \\
\text{subject to} & \quad x_1 + x_2 + x_3 \leq 100 \\
& \quad 10x_1 + 4x_2 + 5x_3 \leq 600 \\
& \quad 2x_1 + 2x_2 + 6x_3 \leq 300 \\
\text{where all variables are non-negative} & \quad x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0
\end{align*}
\]

From GLPK manual

Linear programming standard form

\[
\begin{align*}
\text{maximize} & \quad Z = 10x_1 + 6x_2 + 4x_3 \\
\text{subject to} & \quad p = x_1 + x_2 + x_3 \\
& \quad q = 10x_1 + 4x_2 + 5x_3 \\
& \quad r = 2x_1 + 2x_2 + 6x_3 \\
\text{and bounds of variables} & \quad -\infty < p \leq 100 \quad 0 \leq x_1 < +\infty \\
& \quad -\infty < q \leq 600 \quad 0 \leq x_2 < +\infty \\
& \quad -\infty < r \leq 300 \quad 0 \leq x_3 < +\infty
\end{align*}
\]

Formulate constraints as a set of equations and a set of bounded variables. This is easy to translate to the simplex tableau used for the computations.
import org.gnu.glpk.*;  // Glpk starts numbering at 1, not 0
public class LP {
    public static void main(String[] args) {
        int[] ia = new int[1+1000];  // Row index of coefficient
        int[] ja = new int[1+1000];  // Col index of coefficient
        double[] ar = new double[1+1000];  // Coefficient
        double z, x1, x2, x3;  // Obj value, unknowns
        GlpkSolver solver = new GlpkSolver();
        solver.setProbName("sample");
        solver.setObjDir(GlpkSolver.LPX_MAX);  // Maximization
        solver.addRows(3);
        solver.setRowName(1, "p");
        solver.setRowBnds(1, GlpkSolver.LPX_UP, 0.0, 100.0);
        solver.setRowName(2, "q");
        solver.setRowBnds(2, GlpkSolver.LPX_UP, 0.0, 600.0);
        solver.setRowName(3, "r");
        solver.setRowBnds(3, GlpkSolver.LPX_UP, 0.0, 300.0);
        ia[1] = 1; ja[1] = 1; ar[1] = 1.0; /* a[1,1] = 1 */
        ia[8] = 3; ja[8] = 2; ar[8] = 2.0; /* a[3,2] = 2 */
LP program, p.3

```java
Solver.loadMatrix(9, ia, ja, ar);
Solver.simplex();
z = Solver.getObjVal();
x1 = Solver.getColPrim(1);
x2 = Solver.getColPrim(2);
x3 = Solver.getColPrim(3);
System.out.printf("z = %.3f; x1 = %.3f; x2 = %.3f; x3 = %.3fn",
                 z, x1, x2, x3);
}
```

// Output:
// z = 733.333; x1 = 33.3333; x2 = 66.6667; x3 = 0.00000

Using GLPK under WindowsXP

- Download files, documentation from http://bjoern.dapnet.de/glpk/
- Copy glpk.jar to C:\Program Files\Java\jdk1.6.0_05\jre\lib
- Copy glpk_jni.dll to the directory of project in Eclipse
- Eclipse: Project->Properties->Java Build Path. Add external JAR: glpk.jar
- Create Eclipse project, write class as usual
- See GLPK documentation (pdf). Java method names are based on C names: lpx_set_obj_dir() becomes setObjDir()
- glpk.jar uses the Java Native Interface (JNI) to create Java stubs for the C native functions in the glpk library
- You now have a callable library of linear programming routines that you can use from Java
- GLPK also has:
  - Mixed integer programming method, using branch and bound
    - Use this instead of your own B&B unless you have problem-specific pruning
  - Interior point linear programming method