1.204 Lecture 21

Nonlinear unconstrained optimization:
First order conditions: Newton's method
Estimating a logit demand model

Nonlinear unconstrained optimization

- **Network equilibrium was a constrained nonlinear optimization problem**
  - **Nonnegativity constraints on flows**
  - **Equality constraints on O-D flows**
  - **Other variations (transit, variable demand) have inequality constraints**
- **In these two lectures we examine unconstrained nonlinear optimization problems**
  - **No constraints of any sort on the problem; we just find the global minimum or maximum of the function**
  - **Lagrangians can be used to write constrained problems as unconstrained problems, but it's usually best to handle the constraints explicitly for computation**

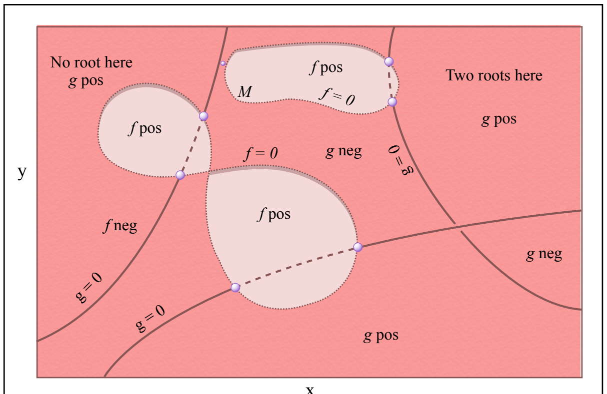
Solving systems of nonlinear equations

- **One way to solve for max z(x), where x is a vector, is to find the first derivatives, set them equal to zero, and solve the resulting system of nonlinear equations**
- **This is the simplest approach and, if the problem is convex (any line between two points on the boundary of the feasible space stays entirely in the feasible space), it is 'good enough'**
- **We will estimate a binary logit demand model with this approach in this lecture**
  - **We'll use a true nonlinear unconstrained minimization algorithm in the next lecture, which is a better way**

Solving nonlinear systems of equations is hard

- **Press, *Numerical Recipes*: "There are <u>no</u> good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there <u>never will</u> be any good, general methods."**
- **"Consider the case of two dimensions, where we want to solve simultaneously"**
  - **f(x, y)= 0**
  - **g(x, y)= 0**

Example of nonlinear system



Solution of two nonlinear equations in two unknowns

Figure by MIT OpenCourseWare.

From Press

Example, continued

- f and g are two functions
  - Zero contour lines divide plane in regions where functions are positive or negative
  - Solutions to f(x,y)=0 and g(x,y)=0 are points in common between these contours
    - f and g have no relation to each other, in general
  - To find all common points, which are the solutions to the nonlinear equations, we must map the full zero contours of both functions
    - Zero contours in general consist of a an unknown number of disjoint closed curves
  - For problems in more than two dimensions, we need to find points common to n unrelated zero contour hypersurfaces, each of dimension n-1
  - Root finding is impossible without insight
    - We must know approximate number and location of solutions a priori

From Press

Nonlinear minimization is easier

- There <u>are</u> efficient general techniques for finding a minimum of a function of many variables
  - Minimization is not the same as finding roots of n first order equations ($\partial z/\partial x = 0$ for all x_n variables)
  - Components of gradient vector ($\partial z/\partial x$) are not independent, arbitrary functions
    - Obey 'integrability conditions': You can always find a minimum by going downhill on a single surface
  - There is no analogous concept for finding the root of N nonlinear equations
- We will cover constrained minimization methods in next lecture
  - Nonlinear root finder has easier code but is less capable
  - Nonlinear minimization has harder code but works well

From Press

Newton-Raphson for nonlinear system

- We have n equations in n variables x_i:

$$f_i(x_0, x_1, x_2, \ldots, x_{n-1}) = 0$$

- Near each x value, we can expand f_i using a Taylor series:

$$f_i(x + \delta x) = f_i(x) + \sum_{j=0}^{n-1} \frac{\partial f_i}{\partial x_j} \delta x_j + O(\delta x^2)$$

- Matrix of partial derivatives is Jacobian J:

$$J_{ij} \equiv \frac{\partial f_i}{\partial x_j}$$

- In matrix notation our expansion is:

$$f(x + \partial x) = f(x) + J \cdot \partial x + O(\partial x^2)$$

Newton-Raphson, p. 2

- Ignore $O(\partial x^2)$ terms and set $f(x+\partial x)= 0$ to find a set of linear equations for the corrections ∂x to move each function in f closer to zero simultaneously:

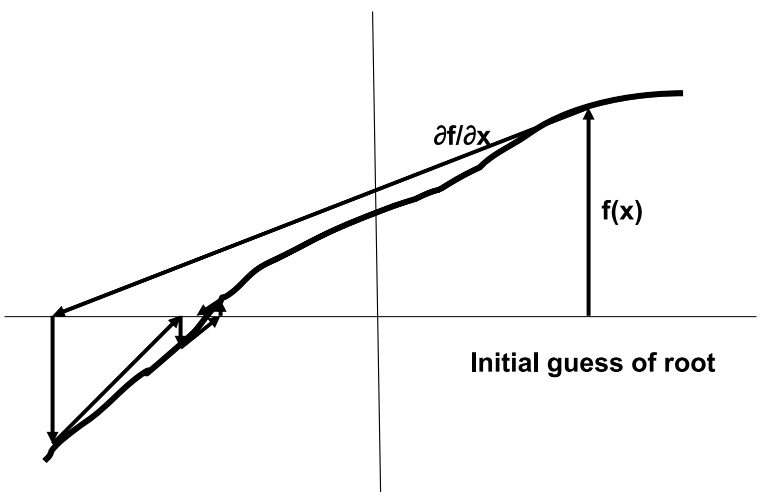
  $$J \cdot \delta x = -f$$

- We solve this system using Gaussian elimination or LU decomposition
- We add the corrections to the previous solution and iterate until we converge:

  $$x' = x + \delta x$$

- If high order derivatives are large or first derivative is small, Newton can fail miserably
  - Converges quickly if assumptions met

Newton-Raphson



In multiple dimensions, simultaneously

Newton class, method

```
public class Newton {
  public static double[] mnewt(int nTrial, double[] x,
      MathFunction2 func) {
    final double TOLERANCE= 1E-13;
    int n= x.length;
    double[] p= new double[n];           // δx
    double[] fvec= new double[n];        // Function value
    double[][] fjac= new double[n][n];  // Jacobian J
    for (int k= 0; k < nTrial; k++) {
      fvec= func.func(x);
      fjac= func.jacobian(x);
      double errf= 0.0;
      for (int i= 0; i < n; i++)         // Close enough to 0?
        errf += Math.abs(fvec[i]);
      if (errf < TOLERANCE)
        return x;
      // Continues on next slide
```

Newton class, method, p.2

```
      // Not close enough, solve for δx (p)
      for (int i= 0; i < n; i++)
        p[i]= -fvec[i];
      p= Gauss.gaussian(fjac, p);
      double errx= 0.0;
      for (int i= 0; i < n; i++) {
        errx += Math.abs(p[i]);
        x[i] += p[i];
      }
      if (errx <= TOLERANCE)
        return x;
    }
    return x;
  }
}
```

```
public interface MathFunction2 {
   double[] func(double[] x);
   double[][] jacobian(double[] x);
}
```

SimpleModel

```
// Solve x² + xy= 10   and   y + 3xy²= 57
public class SimpleModel implements MathFunction2 {
   public double[] func(double[] x) {
        double[] f= new double[x.length];
        f[0]= x[0]*x[0] + x[0]*x[1] - 10;
        f[1]= x[1] + 3*x[0]*x[1]*x[1] - 57;
        return f;
   }
   public double[][] jacobian(double[] x) {
        int n= x.length;
        double[][] j= new double[n][n];
        j[0][0]= 2*x[0] + x[1];
        j[0][1]= x[0];
        j[1][0]= 3*x[1]*x[1];
        j[1][1]= 1 + 6*x[0]*x[1];
        return j;
   }
}
```

SimpleModelTest

```
public class SimpleModelTest {
   public static void main(String[] args) {
        SimpleModel s= new SimpleModel();
        int nTrial= 20;
        double[] x= {1.5, 3.5};       // Initial guess
        x= Newton.mnewt(nTrial, x, s);
        for (double d : x)
               System.out.println(d);
   }
}

// Finds solution {2, 3}
```

Logit demand models

- **Mode choice example for work trip**
  - **Individual has choice between transit and auto**

	In vehicle time	Walk time	Wait time	Cost
Auto	20	3	0	1000
Transit	15	17	4	150

  - **We assume the utility of each choice is a linear function**
    - **U= β_0 + β_1*IVTT + β_2* Walk + β_3* Wait + β_4 * Cost**
  - **The probability p_i that a traveler chooses mode i is**

$$p(i) = \frac{e^{U_i}}{e^{U_i} + e^{U_j}} = \frac{1}{1 + e^{U_j - U_i}}$$

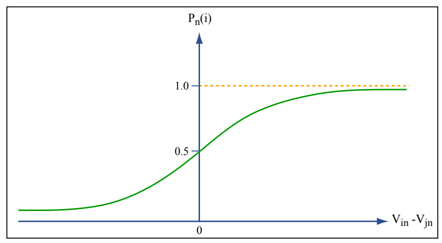


Figure by MIT OpenCourseWare.

Estimating a logit model

Simple Binary Example

	β_1	β_2
Auto utility, V_{An}	1	Transit travel time [min]
Transit utility, V_{Tn}	0	Transit travel time [min]

Data for Simple Binary Example

Observation number	Auto time	Transit time	Chosen alternative
1	52.9	4.4	Transit
2	4.1	28.5	Transit
3	4.1	86.9	Auto
4	56.2	31.6	Transit
5	51.8	20.2	Transit
6	0.2	91.2	Auto
7	27.6	79.7	Auto
8	89.9	2.2	Transit
9	41.5	24.5	Transit
10	95.0	43.5	Transit
11	99.1	8.4	Transit
12	18.5	84	Auto
13	82.0	38.0	Auto
14	8.6	1.6	Transit
15	22.5	74.1	Auto
16	51.4	83.8	Auto
17	18.0	19.2	Transit
18	51.0	85.0	Auto
19	62.2	90.1	Auto
20	95.1	22.2	Transit
21	41.6	91.5	Auto

Figure by MIT OpenCourseWare.

From Ben-Akiva, Lerman

Maximum likelihood estimation

$$L^*(\beta_0, \beta_1, ..., \beta_{k-1}) = \prod_{n=0}^{N-1} P_n(i)^{y_{in}} \cdot P_n(j)^{y_{jn}}$$

It is easier to work with the log of this product; the solution is the same:

$$L(\beta_0, \beta_1, ..., \beta_{k-1}) = \sum_{n=0}^{N-1} [y_{in} \log P_n(i) + y_{jn} \log P_n(j)]$$

Solve for the maximum of L by setting its first derivatives to zero.

For the logit model, see Ben-Akiva and Lerman for the algebra to obtain:

$$\frac{\partial \mathrm{L}(\beta)}{\partial \beta_\mathrm{k}} = \sum_{n=0}^{N-1} [y_{in} - P_n(i)] x_{nk} = 0$$

To use Newton-Raphson, we need the derivatives of the equation system above:

$$\frac{\partial^2 \mathrm{L}(\beta)}{\partial \beta_\mathrm{k} \partial \beta_\mathrm{l}} = -\sum_{n=0}^{N-1} P_n(i)(1 - P_n(i)) x_{nk} x_{nl}$$

DemandModel: constructor, func

```
public class DemandModel implements MathFunction2 {
  private double[][] x;  // Variables for each traveler
  private double[] y;    // Observed choice: 1 auto, 0 if transit
  private double[] p;    // Probability of individual, each iter
  public DemandModel(double[][] x, double[] y) {
    this.x = x; this.y = y; p= new double[y.length]; }

  public double[] func(double[] beta) {
    int n= y.length;      // Number of observations
    int k= beta.length;  // Number of parameters to estimate
    double[] f= new double[beta.length];
    for (int i= 0; i < n; i++) {  // Compute utility
      double util= 0;
      for (int j= 0; j < k; j++)
        util += beta[j]*x[i][j];
      p[i]= 1/(1 + Math.exp(-util));  // Compute estimated prob
    }
    for (int j= 0; j < k; j++)       // Loop over equations
      for (int i= 0; i < n; i++)     // Loop thru observations
        f[j] += (y[i]- p[i])*x[i][j]; // Compute likelihood
    return f;
  }
```

DemandModel: jacobian, logLikelihood

```
public double[][] jacobian(double[] beta) {
  int n= y.length;
  int k= beta.length;
  double[][] jac= new double[k][k];
  for (int j= 0; j < k; j++)
    for (int jj= 0; jj < k; jj++)
      for (int i= 0; i < n; i++)
        jac[j][jj] -= (p[i]*(1-p[i]))*x[i][j]*x[i][jj];
  return jac;
}
public double logLikelihood(double[] beta) {
  int n= y.length;
  int k= beta.length;
  double result= 0.0;
  for (int i= 0; i < n; i++) {        // Compute utility
    double util= 0;
    for (int j= 0; j < k; j++)
      util += beta[j]*x[i][j];
    p[i]= 1/(1 + Math.exp(-util));    // Compute estimated prob
    result += y[i]*Math.log(p[i]) + (1-y[i])*Math.log(1-p[i]);
  }
  return result;}
```

DemandModel: print

```
public void print(double log0, double logB, double[] beta,
    double[][] fjac) {
  int n= fjac.length; // 2nd derivatives give var-covar matrix
  double[][] variance= Gauss.invert(fjac);
  for (int i= 0; i < n; i++)
    for (int j= 0; j < n; j++)
      variance[i][j]= -variance[i][j];
  for (int i= 0; i < beta.length; i++)
    System.out.println("Coefficient "+ i + " : "+ beta[i]+
          " Std. dev. "+ Math.sqrt(variance[i][i]));
  System.out.println("\nLog likelihood(0) "+ log0);
  System.out.println("Log likelihood(B) " + logB);
  System.out.println("-2[L(0)-L(B)]     " + -2.0*(log0-logB));
  System.out.println("Rho^2             " + (1.0 - logB/log0));
  System.out.println("Rho-bar^2         " + (1.0 - (logB-
          beta.length)/log0));
  System.out.println("\nVariance-covariance matrix");
  for (int i= 0; i < n; i++) {
    for (int j= 0; j < n; j++)
      System.out.print(variance[i][j]+" ");
    System.out.println();
} }
```

DemandModelTest

```
public class DemandModelTest {
   public static void main(String[] args) {
        double[][] x= { {1, 52.9 - 4.4},
                        {1, 4.1 - 28.5},  // And all other obs
        };              // Note we use the difference in times
        // 0: transit chosen, 1: auto chosen
        double[] y= {0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
                1, 1, 0, 1, 1, 0, 1};

        DemandModel d= new DemandModel(x, y);
        int nTrial= 20;               // Max Newton iterations
        double[] beta= {0, 0};               // Initial guess
        double log0= d.logLikelihood(beta);
        // Minor tweak to Newton: add getFjac() method
        beta= NewtonForDemand.mnewt(nTrial, beta, d);
        double logB= d.logLikelihood(beta);
        d.print(log0, logB, beta, NewtonForDemand.getFjac());
   }
}
```

DemandModelTest output

```
Iteration 0 coeff 0 : -0.06081971708   coeff 1 : -0.028123966581
Iteration 1 coeff 0 : -0.14520466978   coeff 1 : -0.042988257069
Iteration 2 coeff 0 : -0.21506935954   coeff 1 : -0.051110192177
Iteration 3 coeff 0 : -0.23641429578   coeff 1 : -0.053023776033
Iteration 4 coeff 0 : -0.23757284839   coeff 1 : -0.053109661993
Iteration 5 coeff 0 : -0.23757544483   coeff 1 : -0.053109827465
Iteration 6 coeff 0 : -0.23757544484   coeff 1 : -0.053109827465

Coefficient 0 : -0.237575444848     Std. dev. 0.75047663238
Coefficient 1 : -0.053109827465     Std. dev. 0.02064227879

Log likelihood(0) -14.556090791
Log likelihood(B) -6.1660422124
-2[L(0)-L(B)]     16.7800971586
Rho^2             0.57639435610
Rho-bar^2         0.43899482840

Variance-covariance matrix
0.56321517575      0.00254981359
0.00254981359      4.2610367391E-4
```

Demand model output

Estimation results for simple binary logit example

Variable number	Variable name	Coefficient estimate	Asymptotic standard error	*t* statistic
1	Auto constant	-0.2375	0.7505	-0.32
2	Travel time (min)	-0.0531	0.0206	-2.57

Summary Statistics

Number of observations = 21

Number of cases = 21

$L(0) = -14.556$

$L(c) = -14.532$

$L(\beta) = -6.116$

$-2[L(0) - L(\beta)] = 16.780$

$-2[L(c) - L(\beta)] = 16.732$

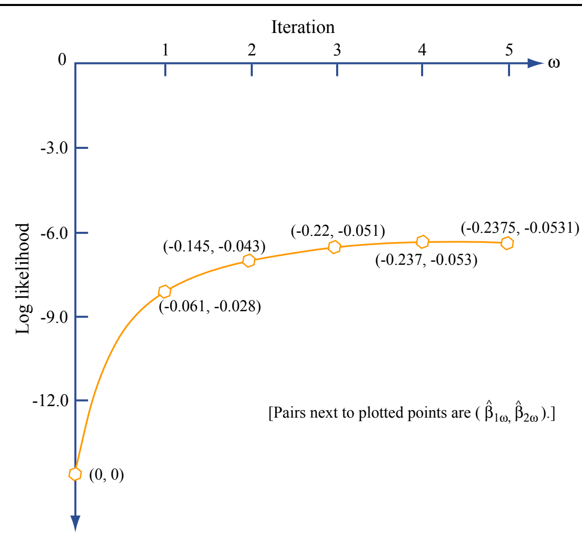
$\rho^2 = 0.576$

$\rho^2 = 0.439$

Figure by MIT OpenCourseWare.

From Ben-Akiva, Lerman

Demand model output



Log likelihood values in estimation of simple binary logit example

Figure by MIT OpenCourseWare.

From Ben-Akiva, Lerman

Summary

- This model is convex, so convergence is easier than many nonlinear models
  - Demand model variations can be more difficult to solve
  - We cover direct minimization methods next lecture, some of which give more control in solving harder problems
- We didn't need a good first guess here, but we almost always do
  - Generate good first guesses through analytical approximations (as in lecture 23 and homework 8)

1.204 Computer Algorithms in Systems Engineering
Spring 2010