Matlab – Optimization and Integration

Paul Schrimpf

January 14, 2009

This lecture focuses on two ubiquitous numerical techiniques:

1. Optimization and equation solving
   - Agents maximize utility / profits
   - Estimation
2. Integration
   - Expectations of the future or over unobserved variables

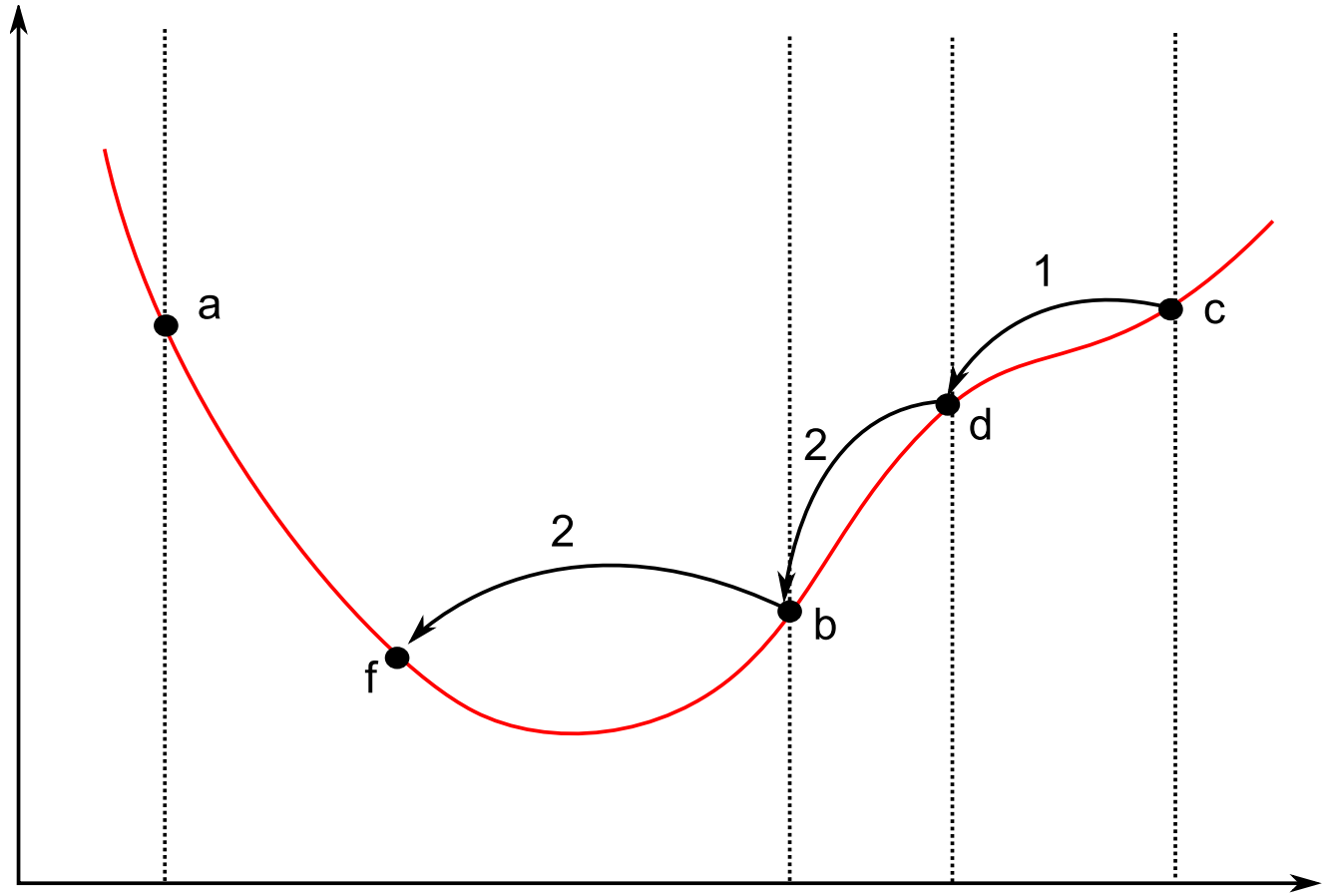
Optimization

- Want to solve a minimization problem:

$$\min_x f(x)$$

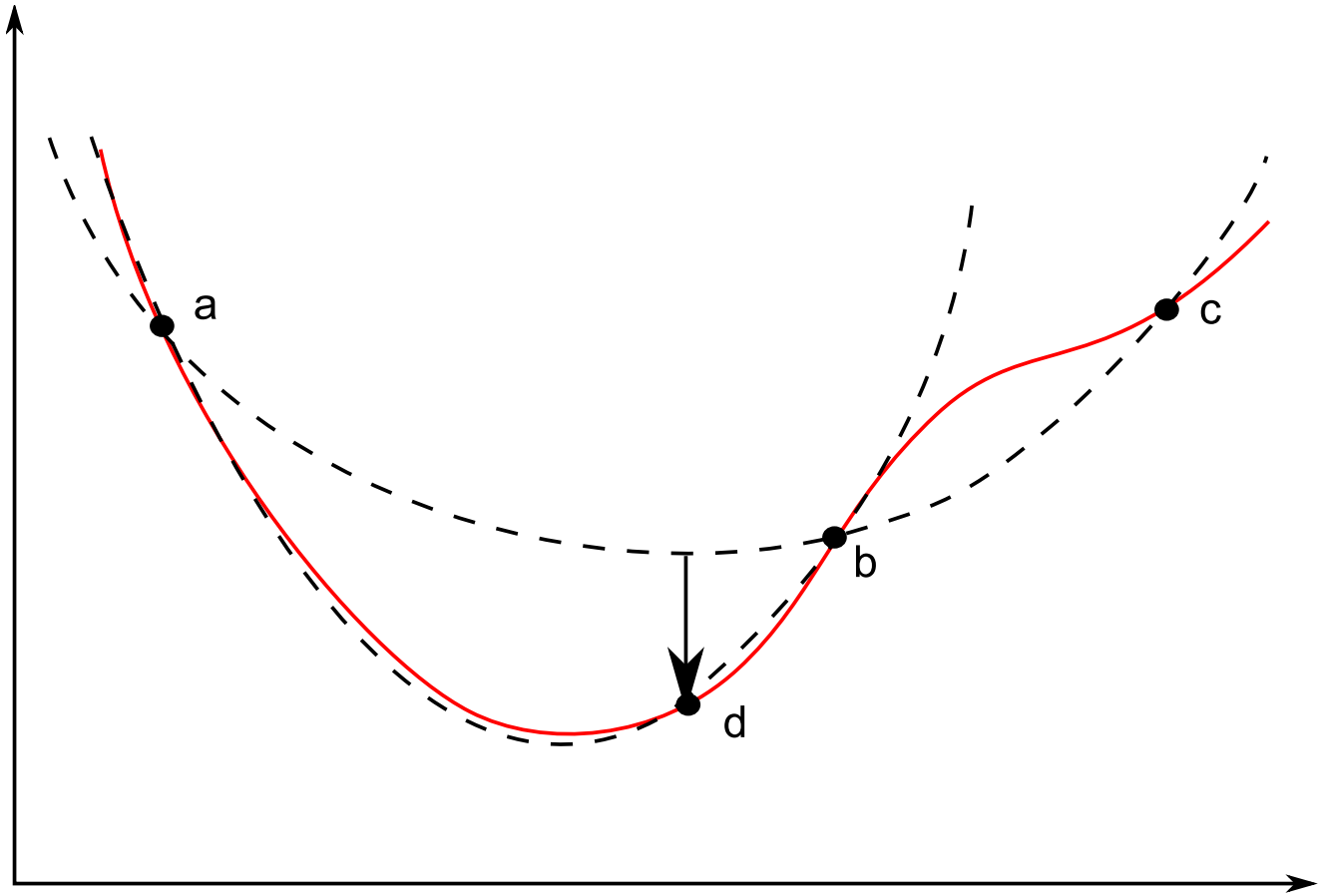
- Two basic approaches:
  1. Heuristic methods search over x in some systematic way
  2. Model based approaches use an easily minimized approximation to $f()$ to guide their search
- First, $x \in \Re$ for intuition
- Then, $x \in \Re^n$

Section Search



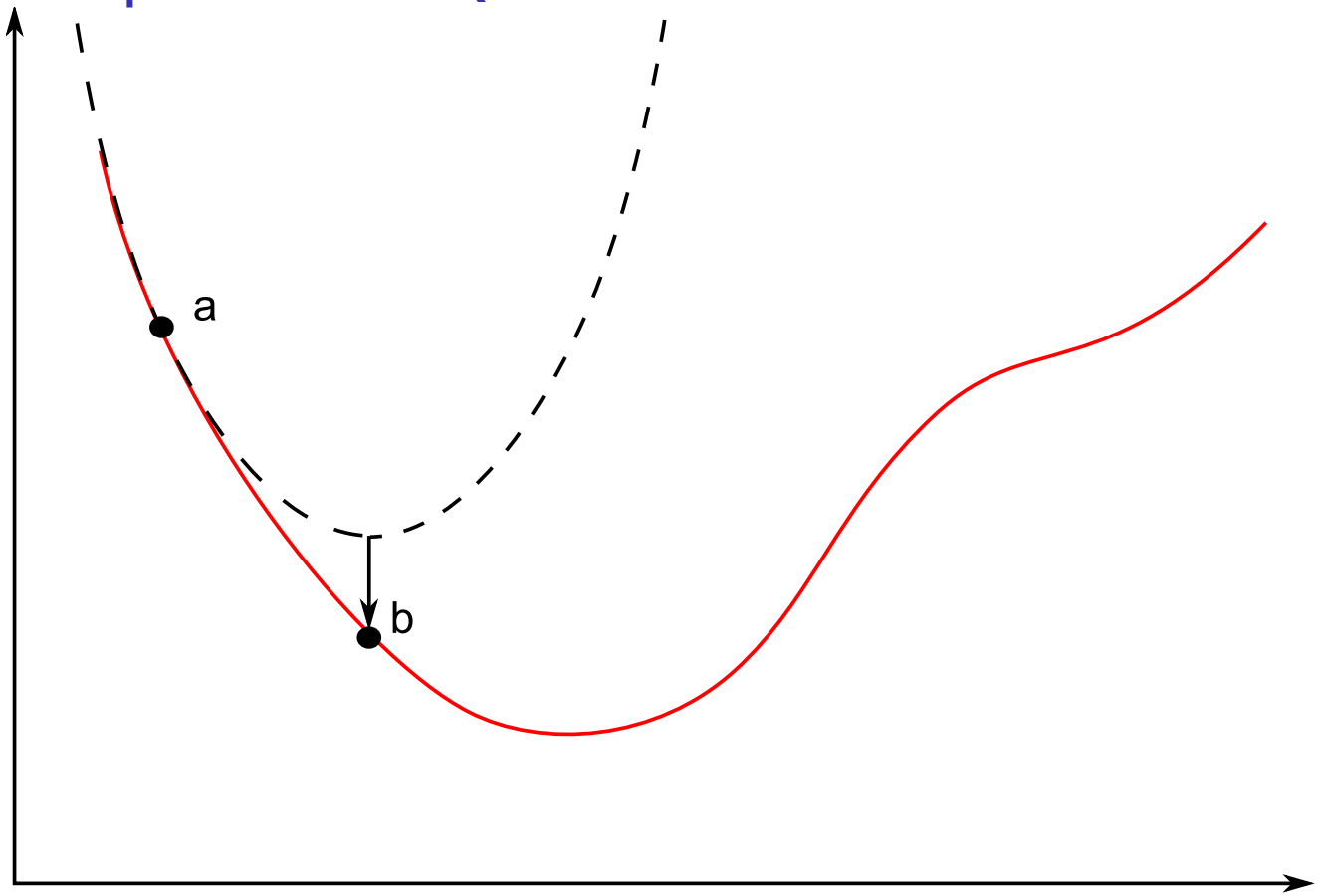
- Form bracket $x_1 < x_2 < x_3$ such that $f(x_2) < f(x_1)$ and $f(x_3)$
- Try new $x \in (x_1, x_3)$, update bracket

Quadratic Interpolation



- More general interpolation methods possible
- e.g. Matlab uses both quadratic and cubic interpolation for line search

Newton-Rhapson and Quasi-Newton



- Newton: use $f'(x)$ and $f''(x)$ to construct parabola

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- Quasi-Newton: approximate $f''(x_n)$ with $\frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}$
- BHHH: approximate $f''(x)$ with $\frac{1}{\sum} f'(x) f'(x)'$

Rates of Convergence

- Newton's method converges quadraticly, *i.e.* in a neighborhood of the solution,

$$\lim_{n \to \infty} \frac{||x_{n+1} - x^*||}{||x_n - x^*||^2} = C$$

- Parabolic interpolation and quasi-Newton methods also achieve better than linear rates of convergence, but (usually) less than quadratic, *i.e.*

$$\lim_{n \to \infty} \frac{||x_{n+1} - x^*||}{||x_n - x^*||^r} = C$$

  for some $r \in (1, 2]$
- Can achieve faster than quadratic convergence by using more information
- Usually, happy with any rate better than 1

Trust Regions

- Problem: For a function that is not globally concave, quadratic interpolation and Newton methods might prescribe an upward step and can fail to converge
- Solution: Combine them with a sectional search, or more generally, a **trust region**
  - Region, R where we "trust" our quadratic approximation, $f(x) \approx \tilde{f}(x)$

$$x_{n+1} = \arg\min_{x \in R} \tilde{f}_n(x)$$

    Shrink or expand R based on how much better $f(x_{n+1})$ is than $f(x_n)$
- Brent's method combines quadratic interpolation with sectional search

Matlab Implementation

- `fminbnd()` uses Brent's method
- No uni-dimensional implementation of any type of Newton method, but could use multi-dimensional versions
- We used `fminbnd()` in lecture 1:

```
1 optimset('fminbnd') % this returns the default options for
2
3 % change some options
4 opt = optimset('Display','iter', ... % 'off','final','noti
5                 'MaxFunEvals',1000,'MaxIter',1000, ...
6                 'TolX',1e-8);
7 % maximize the expected value
8 [c ev] = fminbnd(@(c) obj(c,x,parm), cLo, cHi,opt);
```

- For all optimization functions, can also set options through a graphical interface by using `optimtool()`

Multi-Dimensional Optimization with Quadratic Approximation

- Basic idea is the same:
  - Construct a quadratic approximation to the objective
  - Minimize approximation over some region
- Complicated by difficulty of constructing region
  - Cannot "bracket" a minimum in R^n, would need an $n-1$ dimensional manifold
  - Two approaches:
    - ★ Use n dimensional trust region
    - ★ Break problem into sequence of smaller-dimensional minimizations

Directional Search Methods

1. Choose a direction
   - Naïve approach: use basis or steepest descent directions
     - ★ Very inefficient in worse case
   - Try new directions, keep good ones: Powell's method or conjugate gradients
   - Use Newton or quasi-Newton direction
     - ★ Generally fastest method
2. Do univariate minimization along that direction, this step is called a "line search"
   - Exact: find the minimum along the direction
   - Approximate: just find a point that is enough of an improvement
3. Choose a different direction and repeat

Trust Region Methods

- Same as one dimension:
  - Region, R where we "trust" our quadratic approximation, $f(x) \approx \tilde{f}(x)$

$$x_{n+1} = \arg\min_{x \in R} \tilde{f}_n(x)$$

    Shrink or expand R based on how much better $f(x_{n+1})$ is than $f(x_n)$
- Hybrid method: combine directional seach and a trust region
  1. Use one of approaches from previous slide to choose a $m < n$ dimension subspace
  2. Use trust-region method to minimize f within the subspace
  3. Choose new subspace and repeat

Matlab Implementation

- `fminunc()` offers two algorithms
  1. `optimset('LargeScale','off')` $\longrightarrow$ quasi-Newton with approximate line-search
     - ★ Needs gradient, will compute finite difference approximation if gradient not supplied
     - ★ Hessian approximation underdetermined by $f'(x_n)$ and $f'(x_{n-1})$ in dimension > 1
     - ★ Build Hessian approximation with recurrence relation: `optimset('HessUpdate','bfgs')` (usually better) or `optimset('HessUpdate','dfp')`
  2. `optimset('LargeScale','on')` $\longrightarrow$ hybrid method: 2-d trust region with conjugate gradients
     - ★ Needs the hessian, will compute finite difference approximation if hessian not supplied
     - ★ Can exploit sparisty pattern of Hessian `optimset('HessPattern',sparse(kron(eye(K),ones(J))))`
- Generally, these algorithms perform much better with user-supplied derivatives thatn with finite-difference approximations to derivatives
  - `optimset('GradObj','on','DerivativeCheck','on')`
  - `optimset('Hessian','on')`

Set Based Methods

- Idea:
  1. Evaluate function on a set of points
  2. Use current points and function values to generate candidate new points
  3. Replace points in the set with new points that have lower function values
  4. Repeat until set collapses to a single poitn
- Examples: grid search, Nelder-Mead simplex, pattern search, genetic algorithms, simulated annealing
- Pros: simple, robust
- Cons: inefficient – an interpolation based method will usually do better

Nelder-Mead Simplex Algorithm

- `fminsearch()` uses it
- $N + 1$ points in N dimensions form a polyhedron, move the polyhedron by
  1. Reflect worst point across the center, expand if there's an improvement
  2. Shrink, e.g. toward best point, other variations possible

▸ Animation ▸ Another Animation

Pattern Search

- Uses set of $N + 1$ or more directions, $\{d_k\}$
- Each iteration:
  - Evaluate $f(x_i + d_k \Delta)$
  - If $f(x_i + d_k \Delta) < f(x_i)$, set $x_{i+1} = x_i + d_k \Delta$, increase Δ
  - If $\min_k f(x_i + d_k \Delta) > f(x_i)$, set $x_{i+1} = x_i$, decrease Δ
- In Matlab, `[x fval] = patternsearch(@f,x)`
  - Requires Genetic Algorithm and Direct Search Toolbox

Genetic Algorithm

- Can find global optimum (but I do not know whether this has been formally proven)
- Begin with random "population" of points, $\{x_n^0\}$, then
  1. Compute "fitness" of each point $\propto -f(x_n^i)$
  2. Select more fit points as "parents"
  3. Produce children by mutation $x_n^{i+1} = x_n^i + \epsilon$, crossover $x_n^{i+1} = \lambda x_n^i + (1-\lambda)x_m^i$, and elites, $x_n^{i+1} = x_n^i$
  4. Repeat until have not improved function for many iterations
- In Matlab, `[x fval] = ga(@f,nvars,options)`
  - Requires Genetic Algorithm and Direct Search Toolbox
  - Many variations and options
    - ★ Options can affect whether converge to local or global optimum
    - ★ Read the documentation and/or use `optimtool`

Simulated Annealing and Threshold Acceptance

- Can find global optimum, and under certain conditions, which are difficult to check, finds the global optimum with probability 1
- Algorithm:
  1. Random candidate point $x = x_i + \tau\epsilon$
  2. Accept $x^{i+1} = x$ if $f(x) < f(x_i)$ or
     - ★ simulated annealing: with probability $\frac{1}{1+e^{(f(x)-f(x_i))/\tau}}$
     - ★ threshold: if $f(x) < f(x_i) + T$
  3. Lower the temperature, τ, and threshold, T
- In Matlab, `[x,fval] = simulannealbnd(@objfun,x0)` and `[x,fval] = threshacceptbnd(@objfun,x0)`
  - Requires Genetic Algorithm and Direct Search Toolbox
  - Many variations and options
    - ★ Options can affect whether converge to local or global optimum
    - ★ Read the documentation and/or use `optimtool`

Constrained Optimization

$$\min_x f(x) \tag{1}$$

$$\text{s.t.} \tag{2}$$

$$g(x) = 0 \tag{3}$$

- Quasi-Newton with directional search $\rightarrow$ sequential quadratic programming
  - Choose direction of search both to minimize function and relax constraints
- `fmincon()`
  - `'LargeScale','off'` does sequential quadratic programming
  - `'LargeScale','on'` only works when constraints are simple bounds on x, it is the same as large-scale fminunc
- Matlab's pattern search and genetic algorithm work for constrained problems

Solving Systems of Nonlinear Equations

- Very similar to optimization

$$F(x) = 0 \iff \min_x F(x)'F(x)$$

- Largescale `fsolve` = Largescale `fmincon` applied to least-squares problem
- Mediumscale `fsolve` = Mediumscale `lsqnonlin`
  - Gauss-Newton: replace F by first order expansion

$$x_{n+1} - x_n = (J'(x_n)J(x_n))^{-1}J'(x_n)F(x_n)$$

  - Levenberg-Marquardt: add dampening to Gauss-Newton to improve performance when first order approximation is bad

$$x_{n+1} - x_n = (J'(x_n)J(x_n) + \lambda_n I)^{-1}J'(x_n)F(x_n)$$

Derivatives

- The best minimization algorithms require derivatives
- Can use finite difference approximation
  - In theory: still get better than linear rate of convergence
  - In practice: can be inaccurate
  - Takes n function evaluations, user written gradient typically takes 2-5 times the work of a function evaluation
- Easier analytic derivatives:
  - Use symbolic math program to get formulas – e.g. Matlab Symbolic Math Toolbox / Maple, Mathematica, Maxima
  - Use automatic differentiation
    - ★ In Matlab – INTLAB, ADMAT, MAD, ADiMat, or a version that we will create in the next lecture
    - ★ Switch to a language with native automatic differentiation – AMPL, GAMS

Simple MLE Example: Binary Choice

```
% Script for estimating a binary choice model
% Paul Schrimpf, May 27, 2007
clear;
% set the parameters
data.N = 1000; % number of observations
data.nX = 2;   % number of x's
parm.beta = ones(data.nX,1);
% note use of function handles for distribution
% estimation assumes that the distribution is known
parm.dist.rand = @(m,n) random('norm',1,0,m,n);
parm.dist.pdf = @(x) pdf('norm',x,0,1);
parm.dist.dpdf = @(x) pdf('norm',x,0,1).*x; % derivative of pd
parm.dist.cdf = @(x) cdf('norm',x,0,1);
```

```
% create some data
data = simulateBinChoice(data,parm);
% set optimization options
opt = optimset('LargeScale','off', ...
               'HessUpdate','bfgs', ...
               'GradObj','on', ...
               'DerivativeCheck','on', ...
               'Display','iter', ...
               'OutputFcn',@binChoicePlot);
b0 = zeros(data.nX,1);
[parm.beta like] = fminunc(@(b) binChoiceLike(b,parm,data), ...
                           b0,opt);
% display results
fprintf('likelihood = %g\n',like);
for i=1:length(parm.beta)
  fprintf('beta(%d) = %g\n',i,parm.beta(i));
end
```

simulateBinChoice()

```
1 function data=simulateBinChoice(dataIn,parm)
2 % ... comments and error checking omitted ...
3   data.x = randn(data.N,data.nX);
4   data.y =  (data.x*parm.beta + epsilon > 0);
5 end
```

```
1  function [like grad hess gi] = binChoiceLike(b,parm,data)
2  % returns the -loglikelihood of 'data' for a binary choice mod
3  % the model is  y = (x*b + eps > 0)
4  % ... more comments omitted
5    xb = data.x*b;
6    % l_i will be N by 1, likelihood for each person
7    l_i = parm.dist.cdf(-xb);
8    l_i(data.y) = 1-l_i(data.y);
9    if any(l_i==0)
10     warning('likelihood = 0 for %d observations\n',sum(l_i==0)
11     l_i(l_i==0) = REALMIN; % don't take log(0)!
12   end
13   like = -sum(log(l_i));
```

Gradient for binChoiceLike()

```
% gradient of l_i
g_i = -(parm.dist.pdf(-xb)*ones(1,length(b))).*data.x;
g_i(data.y,:) = -g_i(data.y,:);
% change to gradient of log-like
grad = -sum(g_i./(l_i*ones(1,length(b))),1)';
```

Hessian for binChoiceLike()

```
1    % calculate hessian
2    h_i = zeros(length(xb),length(b),length(b));
3    for i=1:length(xb)
4      h_i(i,:,:) = (parm.dist.dpdf(-xb(i))* ...
5                    data.x(i,:)'*data.x(i,:)) ...
6          ... % make hessian of log-like
7            / l_i(i);
8    end
9    h_i(data.y,:,:) = -h_i(data.y,:,:);
10   % make hessiane of log-like
11   hess = -(sum(h_i,1) - g_i'*(g_i./(l_i.^2*ones(1,length(b)))))
```

binChoicePlot.m

```
1  function stop = binChoicePlot(x,optimvalues,state)
2    if(length(x)==2)
3      if (optimvalues.iteration==0)
4         hold off;
5      end
6      grad = optimvalues.gradient;
7      f = optimvalues.fval;
8      plot3(f,x(1),x(2),'k*');
9      if (optimvalues.iteration==0)
10        hold on;
11     end
12     quiver3(f,x(1),x(2),0,grad(1),grad(2));
13     drawnow
14     pause(0.5);
15   end
16   stop = false;
17 end
```

Numerical Integration

- Want:

$$F(x) = \int_a^b f(x)\omega(x)dx$$

- Approximate:

$$F(x) \approx \sum_i^n f(x_i)w_i$$

- Different methods are different ways to choose n, x_i, and w_i
- Quadrature: choose w_i so the approximation is exact for a set of n basis elements
- Monte Carlo: set $w_i = \frac{1}{n}$, choose x_i randomly
- Adaptive: refine n, x_i, and w_i until approximation error is small

Quadrature

- Suppose x_i, n are given, need to choose w_i
- Let $\{e_j(x)\}$ be a basis for the space of functions such that $\int_a^b f(x)\omega(x)dx < \infty$
  - $\int_a^b e_j(x)\omega(x)dx$ should be known $\forall j$
- $\{w_i\}_{i=1}^n$ solve

$$\sum_{i=1}^{n} w_i e_j(x_i) = \int_a^b e_j(x)\omega(x)dx \text{ for} j = 1..n \tag{4}$$

- Example: Newton-Cotes
  - basis = polynomials
  - $\omega(x) = 1$
  - Resulting rule is exact for all polynomials of degree less than or equal to n

Gaussian Quadrature

- Now suppose n is given, but we can choose both w_i and x_i
- Same idea, $\{w_i, x_i\}_{i=1}^{n}$ solve

$$\sum_{i=1}^{n} w_i e_j(x_i) = \int_a^b e_j(x)\omega(x)dx \text{ for} j = 1..2n-1 \tag{5}$$

- Exact for functions in the space spanned by $\{e_j\}_{j=1}^{2n-1}$
- If $\{e_j\}$ is an orthogonal polynomial basis, then $\{x_i\}_{i=1}^{n}$ will be the roots of e_n
- Different gaussian quadrature rules correspond to different values of a, b, and $\omega(x)$

Common Forms of Gaussian Quadrature

Interval	$\omega(\mathbf{x})$	**Name**
$[-1, 1]$	1	Legendre
$(-1, 1)$	$(1-x)^{\alpha}(1+x)^{\beta}$	Jacobi
$(-1, 1)$	$\frac{1}{\sqrt{1-x^2}}$	Chebyshev (first kind)
$[-1, 1]$	$\sqrt{1-x^2}$	Chebyshev (second kind)
$[0, \infty)$	e^{-x}	Laguerre
$(-\infty, \infty)$	e^{-x^2}	Hermite

Quadrature in Matlab

- Not built in
- Many implementations available at the Mathworks file exchange
- Great for single dimension integration
- Multiple dimension integration is harder
  - $\sum_{i_1}^{n} \ldots \sum_{i_m=1}^{n} f(x_{i1}, ..x_{im}) w_{im} .. w_{im}$ works, but needs m^n function evaluations
  - More sophisticated methods exist – called cubature, sparse grid, or complete polynomials – see e.g. Encyclopedia of Cubature Formulas

Adaptive Integration

$$F(x) = \int_a^b f(x)\omega(x)dx$$

- Idea: subdivide (a, b) into smaller intervals, use simple quadrature rule on each interval, and repeat until convergence
- e.g. trapezoid rule, Simpson's rule
- `trapz()`, `quad()`, `quadl()`
- Pro: computes integral to known accuracy
- Con: care must be used when part of an objective function
  - Makes the objective function discontinuous at points where solution goes from k to $k + 1$ subdivisions $\rightarrow$ integration accuracy must be set much higher than convergence criteria of optimization

Monte Carlo Integration

- Randomly draw x_i from distribution $p(x) \propto \omega(x)$, set $w_i = \frac{1}{n} \int \omega(x) dx$
- Many methods for sampling from $p(x)$: inverse cdf, acceptance sampling, importance sampling, Gibbs sampling, Metropolis-Hastings
- Pros: simple to understand, easy to implement, scales well, requires little a priori knowledge of $f(x)$
- Cons: inefficient – for a fixed n, the right quadrature rule will do much better
  - But when computing something like $\sum_i g\left(\sum_s f(y_i, x_{i,s}) w_s\right)$, errors in $g\left(\sum_s f(y_i, x_{i,s}) w_s\right)$ for different i can offset one another

Useful Matlab functions for Monte Carlo Integration

```
x = rand(2,3); % 2 by 3 matrix of x ¬ U[0,1]
y = randn(10); % 10 by 10 matrix of y ¬ N(0,1)
% more generally
t = random('t',3,2,1); % 2 by 1, t-dist with 3 df
% many other distributions possible
```

Integration Example

```
clear;
% some integration experiments

% Example 1: E[p(x)], x¬N(0,1), p(x) polynomial
degree = 10;
p = rand(1,degree+1); % make some polynomial
fprintf('adaptive quad, tol %.1g = %.10g\n', ...
        1e-10, quad(@(x) polyval(p,x).*normpdf(x), ...
                    -30,30,1e-10));
```

Integration Example

```
fprintf(' gauss-hermite quadrature\n');
for n=1:((degree+1)/2+4)
  % use hermite -- will be exact for n≥(degree+1)/2
  int = gaussHermite(n);
  % notice the change of variables
  ep = polyval(p,int.x*sqrt(2))'*int.w/sqrt(pi);
  if (n==round((degree+1)/2))
    fprintf('--- the rest should be exact ----\n');
  end
  fprintf('n=%2d  E[p(x)] = %.10g\n',n,ep);
end
```

Integration Example

```
fprintf('\n monte carlo integration\n')
for n=1:6
  fprintf('n=10^%d  E[p(x)] = %.10g\n',n, ...
          mean(polyval(p,randn(10^n,1))));
end
```

MCMC

- For nonstandard distributions, it is often impossible to sample from $p(x)$ directly
- MCMC constructs a Markov Chain, $x_t \sim p(x_t|x_{t-1})$, with stationary distribution $p(x)$ and transition kernel, $p(x_t|x_{t-1})$ that can be sampled from
- Very common in Bayesian statistics

Metropolis-Hastings

- General method for constructing a Markov chain
- Algorithm to draw from $p(x)$: beginning with some x_0
  1. Draw $y \sim q(x_t, \cdot)$
  2. Compute $\alpha(x_t, y) = \frac{p(y)q(y,x_t)}{p(x_t)q(x_t,y)}$
  3. Draw $u \sim U[0, 1]$
  4. If $u < \alpha(x_t, y)$ set $x_{t+1} = y$, otherwise set $x_{t+1} = x_t$
- Choice of candidate density, q, affects behavior of chain
  - If q is too disperse, will not accept many draws
  - If q is too concentrated, will accept lots of draws, but they'll be close together
- Example: metropolisHastings.m

Exercises

1. If you have written any code that involves optimization or integration, try modifying it to use a different method.
2. Modify the dynamic programming code from lecture 1 to allow for a continuous distribution for income. If you are clever, will be able to evaluate $E\tilde{v}(x', y')$ exactly, even if $Ev(x', y')$ does not have an analytic solution.
3. We know that value function iteration converges linearly. Gauss-Newton and other quadratic approximation based method converge at faster rates. Change the dynamic programming code from lecture 1 to use one of these methods instead of value function iteration.
4. Change the binary choice model into a multinomial choice model. Allow for correlation between the shocks. Try to preserve the generality of the binary model, but feel free to limit the choice of distribution if it helps.

More Exercises

1. (hard) I don't know much about cubature, but I'd like to learn more. Read about a few methods. Find or write some Matlab code for one of them. Explore the accuracy of the method. To maintain a desired level of accuracy, how does the number of points grow with the number of dimensions? Compare it monte carlo integration.
2. (hard) Matlab lacks an implementation of an optimization algorithm that uses interpolation in multiple dimensions. Remedy this situation. Find or develop an algorithm and implement it.
3. (hard, but not as hard) Write a function for computing arbitrary Gaussian quadrature rules with polynomials as a basis. Given integration limits, a weight function, $\omega(x)$, and the number of points, n, your function should return the integration points and weights. You might want to use the following facts taken from Numerical Recipes. Let $\langle f|g\rangle = \int_a^b f(x)g(x)\omega(x)dx$ denote the inner product. Then the following recurrence relation will construct a set of orthogonal polynomials:

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= \left(x - \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle}\right)x - \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle}p_{j-1}(x) \end{aligned}$$

Recall that the roots of the n degree polynomial will be the integration points. If you have the roots, $\{x_j\}_{j=1}^n$. Then the weights are given by

$$w_j = \frac{\langle p_{n-1}|p_{n-1}\rangle}{p_{n-1}(x_j)p'_n(x_j)}$$