Structural Estimation of Dynamic Discrete Choice Models

Constrained and Unconstrained Optimization Approaches to Structural Estimation

Bertel Schjerning, University of Copenhagen

Dynamic Programming and Structural Econometrics #7

MPEC is used in multiple contexts

Single-Agent Dynamic Discrete Choice Models

- Rust (1987): Bus-Engine Replacement Problem
- Nested-Fixed Point Problem (NFXP)
- Su and Judd (2012): Constrained Optimization Approach

Random-Coefficients Logit Demand Models

- BLP (1995): Random-Coefficients Demand Estimation
- Nested-Fixed Point Problem (NFXP)
- Dube, Fox and Su (2012): Constrained Optimization Approach

Estimating Discrete-Choice Games of Incomplete Information

- Aguirregabiria and Mira (2007): NPL (Recursive 2-Step)
- Bajari, Benkard and Levin (2007): 2-Step
- Pakes, Ostrovsky and Berry (2007): 2-Step
- Pesendorfer and Schmidt-Dengler (2008): 2-Step
- Pesendorfer and Schmidt-Dengler (2010): comments on AM (2007)
- Kasahara and Shimotsu (2012): Modified NPL
- Su (2013), Egesdal, Lai and Su (2014): Constrained Optimization

Recall the Nested Fixed Point Algorithm

NFXP solves the unconstrained optimization problem

$$\max_{\theta} L(\theta, EV_\theta)$$

Outer loop (Hill-climbing algorithm):

- Likelihood function $L(\theta, EV_\theta)$ is maximized w.r.t. θ
- Quasi-Newton algorithm: Usually BHHH, BFGS or a combination.
- Each evaluation of $L(\theta, EV_\theta)$ requires solution of EV_θ

Inner loop (fixed point algorithm):
The implicit function EV_θ defined by $EV_\theta = \Gamma(EV_\theta)$ is solved by:

- Successive Approximations (SA)
- Newton-Kantorovich (NK) Iterations

Mathematical Programming with Equilibrium Constraints

MPEC solves the constrained optimization problem

$$\max_{\theta, EV} L(\theta, EV) \text{ subject to } EV = \Gamma_\theta(EV)$$

using general-purpose constrained optimization solvers such as KNITRO

Su and Judd (Ecta 2012) considers two such implementations:

MPEC/AMPL:

- AMPL formulates problems and pass it to KNITRO.
- Automatic differentiation (Jacobian and Hessian)
- Sparsity patterns for Jacobian and Hessian

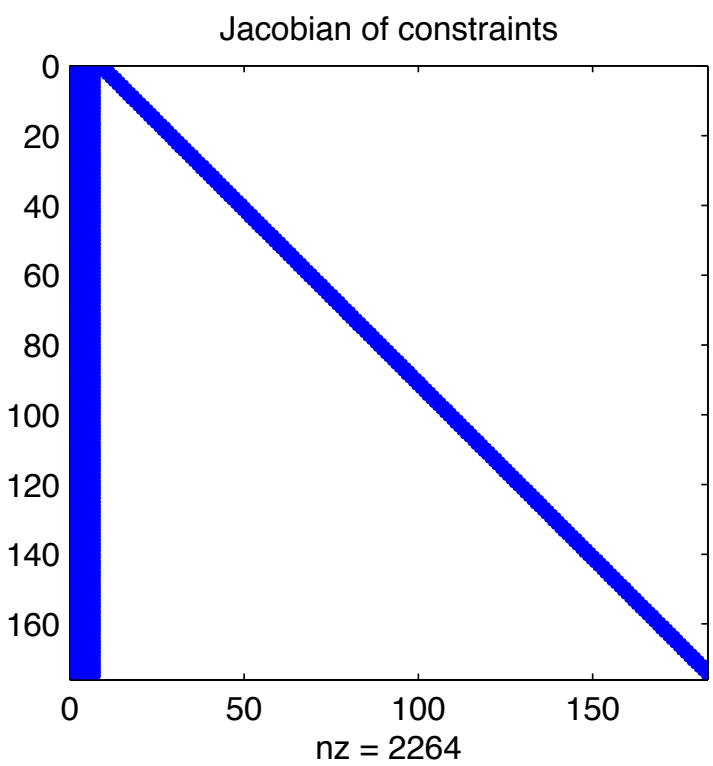
MPEC/MATLAB:

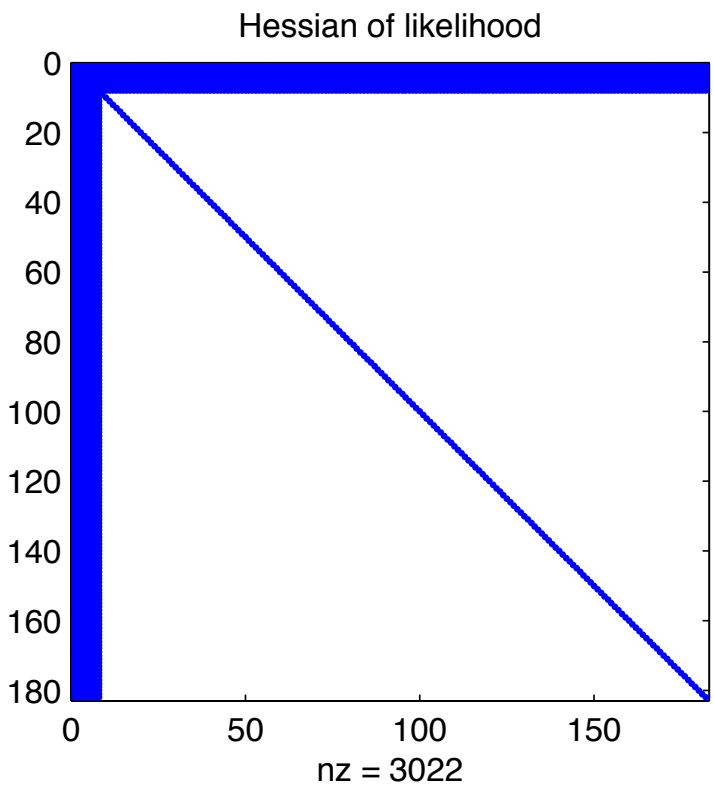
- User need to supply Jacobians, Hessian, and Sparsity Patterns
- Su and Judd do not supply analytical derivatives.
- ktrlink provides link between MATLAB and KNITRO solvers.

Sparsity patterns for MPEC

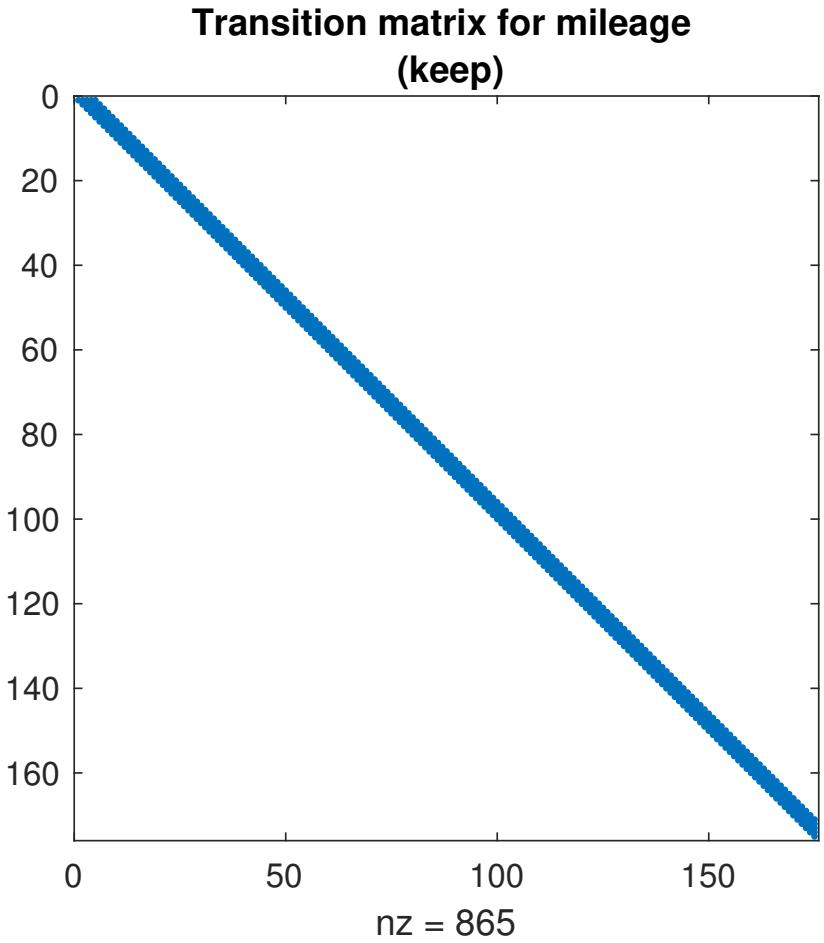
Two key factors in efficient implementations:

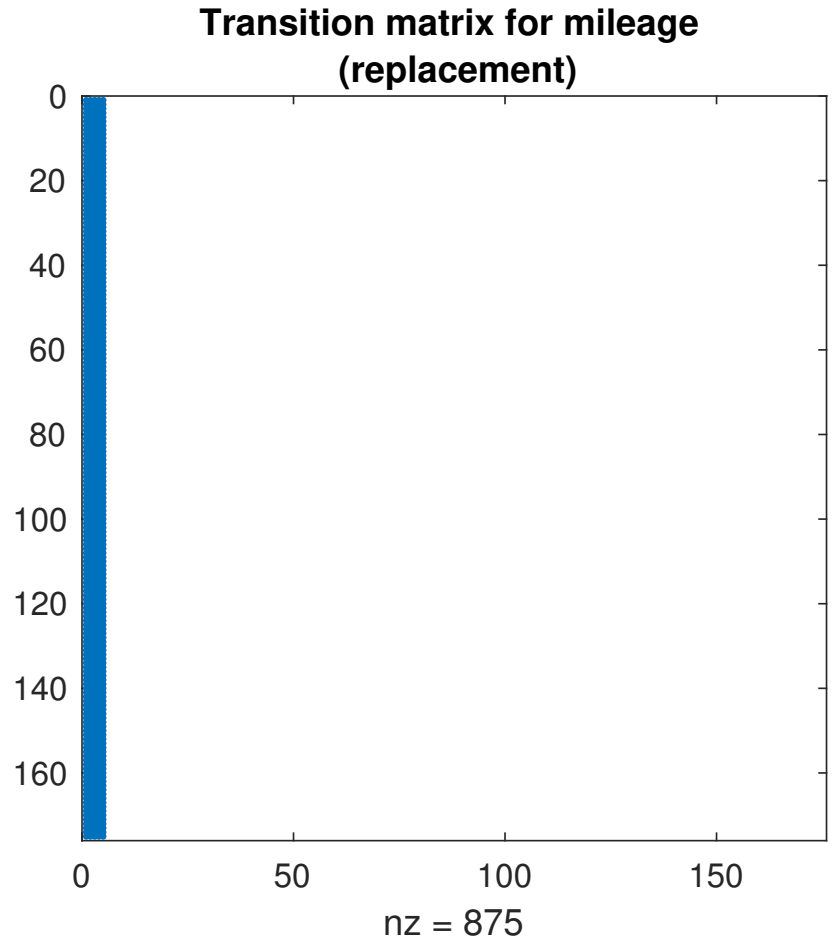
- Provide analytic-derivatives (huge improvement in speed)
- Exploit sparsity pattern in constraint Jacobian (huge saving in memory requirement)





Transition matrix is sparse





Monte Carlo: Rust's Table X - Group 1,2, 3

- Fixed point dimension: $n = 175$
- Maintenance cost function: $c(x, \theta_1) = 0:001 * \theta_1 * x$
- Mileage transition: stay or move up at most $L = 4$ grid points
- True parameter values:
  - $\theta_1 = 2:457$
  - $RC = 11.726$
  - $\theta_2 = (\pi_1, \pi_2, \pi_3, \pi_4) = (0.0937, 0.4475, 0.4459, 0.0127)$
- Solve for EV at the true parameter values
- Simulate 250 datasets of monthly data for 10 years and 50 buses

Is NFXP a dinosaur method?

TABLE II

NUMERICAL PERFORMANCE OF NFXP AND MPEC IN THE MONTE CARLO EXPERIMENTS[a]

β	Implementation	Runs Converged (out of 1250 runs)	CPU Time (in sec.)	# of Major Iter.	# of Func. Eval.	# of Contraction Mapping Iter.
0.975	MPEC/AMPL	1240	0.13	12.8	17.6	–
	MPEC/MATLAB	1247	7.90	53.0	62.0	–
	NFXP	998	24.60	55.9	189.4	134,748
0.980	MPEC/AMPL	1236	0.15	14.5	21.8	–
	MPEC/MATLAB	1241	8.10	57.4	70.6	–
	NFXP	1000	27.90	55.0	183.8	162,505
0.985	MPEC/AMPL	1235	0.13	13.2	19.7	–
	MPEC/MATLAB	1250	7.50	55.0	62.3	–
	NFXP	952	43.20	61.7	227.3	265,827
0.990	MPEC/AMPL	1161	0.19	18.3	42.2	–
	MPEC/MATLAB	1248	7.50	56.5	65.8	–
	NFXP	935	70.10	66.9	253.8	452,347
0.995	MPEC/AMPL	965	0.14	13.4	21.3	–
	MPEC/MATLAB	1246	7.90	59.6	70.7	–
	NFXP	950	111.60	58.8	214.7	748,487

[a]For each β, we use five starting points for each of the 250 replications. CPU time, number of major iterations, number of function evaluations and number of contraction mapping iterations are the averages for each run.

NFXP survival kit

Step 1: Read NFXP manual and print out NFXP pocket guide
Step 2: Recenter logit and logsum formulas
Step 3: Use Fixed Point Poly-Algorithm (SA+NK)
Step 4: Provide analytical gradients of Bellman operator
Step 5: Provide analytical gradients of likelihood
Step 6: Use BHHH (outer product of gradients as hessian approx.)

STEP 1: NFXP documentation

References


Rust (1987): "Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher" Econometrica 55-5, pp 999-1033.


Rust (2000): “Nested Fixed Point Algorithm Documentation Manual: Version 6”
`https://editorialexpress.com/jrust/nfxp.html`


Iskhakov, F. , J. Rust, B. Schjerning, L. Jinhyuk, and K. Seo (2015): "Constrained Optimization Approaches to Estimation of Structural Models : Comment." Econometrica 84-1, pp. 365-370.

Nested Fixed Point Algorithm

NFXP Documentation Manual version 6, (Rust 2000, page 18):

Formally, one can view the nested fixed point algorithm as solving the following constrained optimization problem:

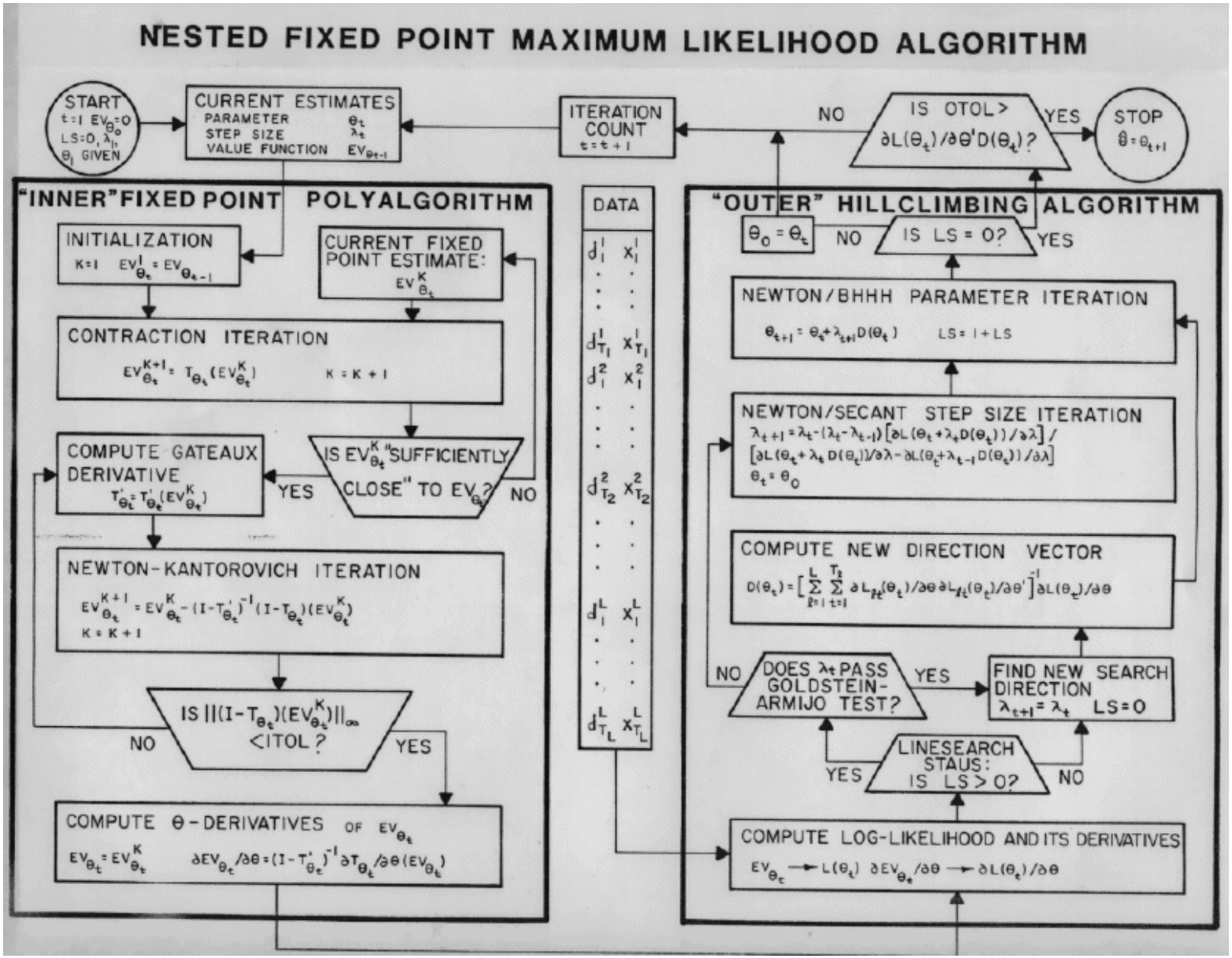
$$\max_{\theta, EV} L(\theta, EV) \text{ subject to } EV = \Gamma_\theta(EV) \tag{1}$$

Since the contraction mapping Γ *always has a unique fixed point, the constraint* $EV = \Gamma_\theta(EV)$ *implies that the fixed point* EV_θ *is an implicit function of* θ*. Thus, the constrained optimization problem (1) reduces to the unconstrained optimization problem*

$$\max_{\theta} L(\theta, EV_\theta) \tag{2}$$

where EV_θ *is the implicit function defined by* $EV_\theta = \Gamma(EV_\theta)$*.*

NFXP pocket guide



STEP 2: Recenter to ensure numerical stability

Logit formulas must be reentered.

$$P_i = \frac{\exp(v_i)}{\sum_j \exp(v_j)} = \frac{\exp(v_i - v_0)}{\sum_j \exp(v_j - v_0)}$$

and "log-sum" must be recentered too

$$\ln \sum_j \exp(v_j) = v_0 + \ln \sum_j \exp(v_j - v_0)$$

If v_0 is chosen to be $v_0 = \max_j v_j$ we can avoid numerical instability due to overflow/underflow

STEP 3: Use Fixed Point Poly-Algorithm (SA+NK)

Problem: Find fixed point of the contraction mapping, Γ_θ

$$EV_\theta = \Gamma(EV_\theta)$$

Fixed Point Poly-Algorithm:

1. Successive Approximations (SA) by contraction iteration:

$$EV_{k+1} = \Gamma_\theta(EV_k)$$

   - Error bound: $||EV_{k+1} - EV|| \leq \beta||EV_k - EV||$
     $\rightarrow$ Linear convergence $\rightarrow$ slow when β close to 1

2. Newton-Kantorovich (NK) iteration:
   - Solve $F = [I - \Gamma](EV_\theta) = 0$ using Newtons method

$$EV_{k+1} = EV_k - (I - \Gamma')^{-1}(I - \Gamma)(EV_k)$$

     Γ'_θ is the Fréchet derivative of Γ_θ
     I is the identity operator on B
     0 is the zero element of B
   - Error bound: $||EV_{k+1} - EV|| \leq A||EV_k - EV||^2$
     $\rightarrow$ Quadratic convergence around fixed point, EV

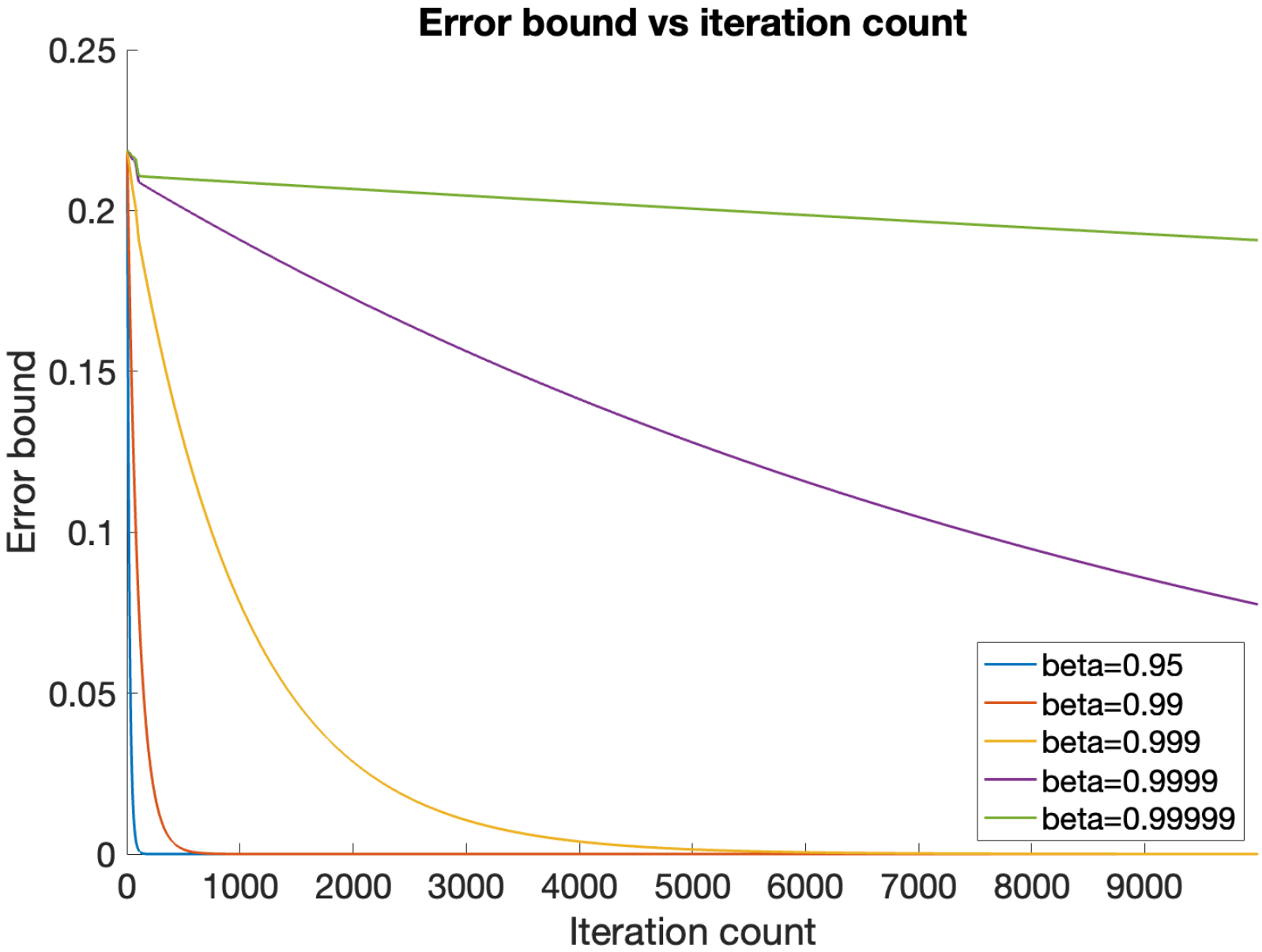
Successive Approximations, $\beta = 0.9999$

```
>> run_fxp
iter            tol         tol(j)/tol(j-1)
   1         0.21854635        1.00000000
   2         0.21852208        0.99988895
   3         0.21849729        0.99988654
   :              :                :
 49998       0.00142119        0.99990000
 49999       0.00142105        0.99990000
 50000       0.00142090        0.99990000
Maximum number of iterations exceeded without convergence!
Elapsed time: 1.44489 (seconds)
```

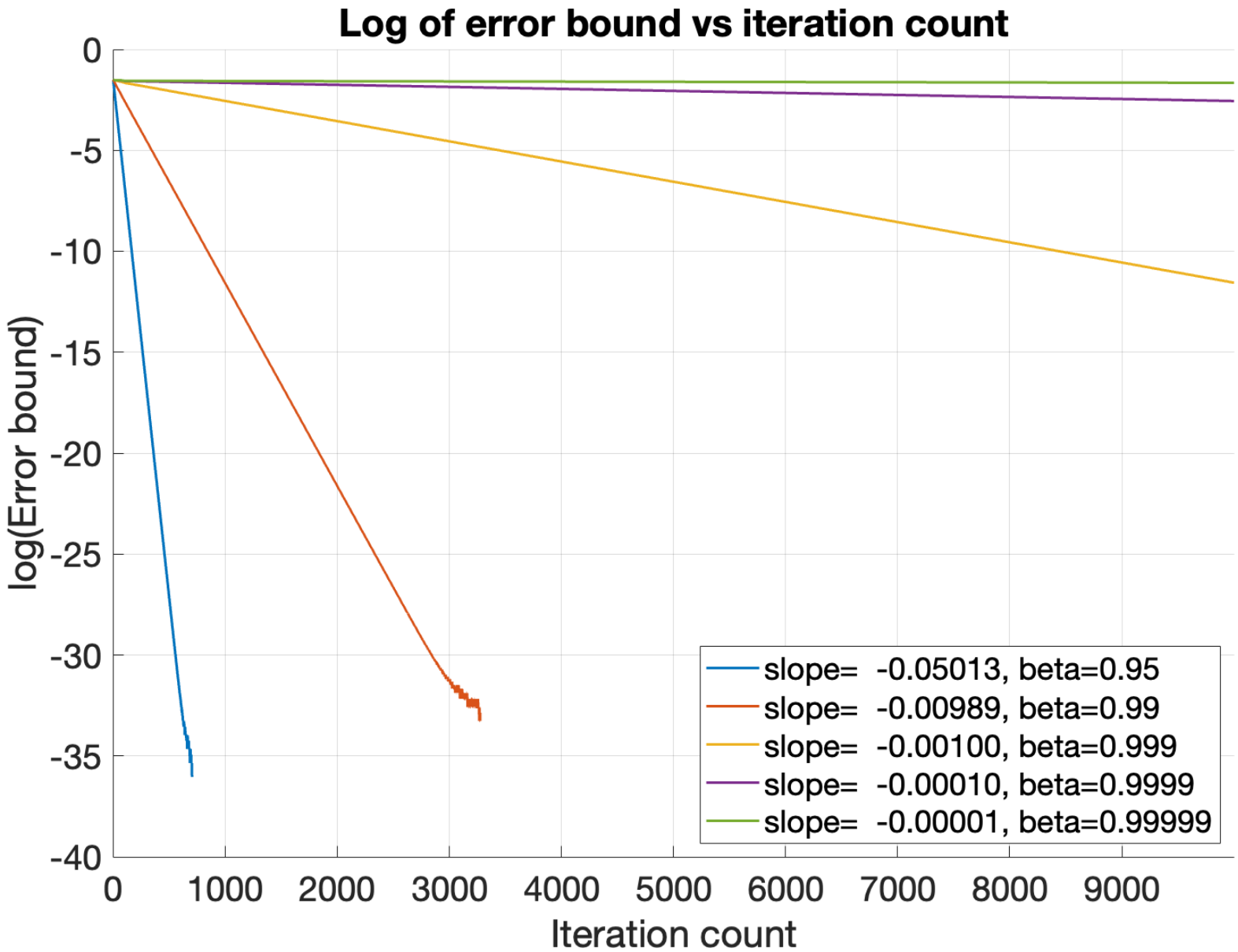
Observations:

- $tol_k = \|EV_{k+1} - EV_k\| < \beta\|EV_k - EV\|$
- Tolerance always improves due to contraction property
- tol_k quickly slow down and declines very slowly for β close to 1
- Relative tolerance tol_{k+1}/tol_k approach β

Successive Approximations - VERY slow when β close to 1



Successive Approximations - linear convergence



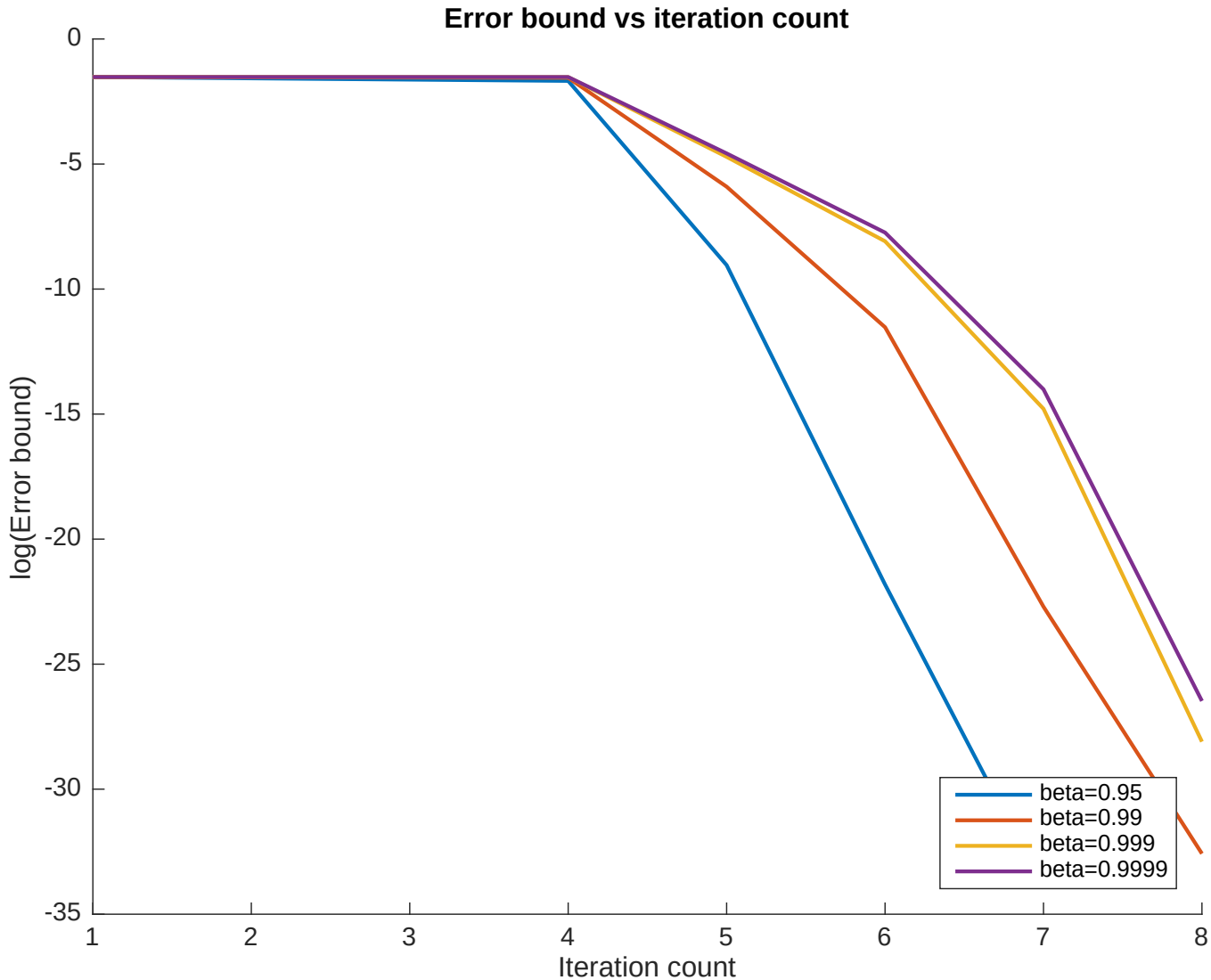
Newton-Kantorovich Iterations, $\beta = 0.9999$

```
>> run_fxp
Begin contraction iterations (for the 1. time)
iter            tol         tol(j)/tol(j-1)
   1         0.21854635        1.00000000
   2         0.21852208        0.99988895
SA stopped prematurely due to rel. tolerance. Begin NK iterations
Elapsed time: 0.00147 (seconds)

Begin Newton-Kantorovich iterations (for the 1. time)
iter            tol         tol(j)/tol(j-1)
   1         0.01037444                NaN
   2         0.00041127                NaN
   3         0.00000069                NaN
   4         0.00000000                NaN
N-K converged after 4 iterations, tolerance: 2.27374e-12
Elapsed time: 0.00331 (seconds)

Convergence achieved!
Total elapsed time: 0.00300 (seconds)
```

Observations:

- Quadratic convergence!
- Very fast, once in domain of attraction

Newton-Kantorovich Iterations - quadratic convergence!



When to switch to Newton-Kantorovich?

When to switch to Newton-Kantorovich?

- Suppose that $EV_0 = EV + k$.
  (Initial EV_0 equals fixed point EV plus an arbitrary constant)
- Another successive approximation does not solve this:

$$\begin{aligned} tol_0 &= \|EV_0 - \Gamma(EV_0)\| = \|EV + k - \Gamma(EV + k)\| \\ &= \|EV + k - (EV + \beta k)\| = (1 - \beta)k \\ tol_1 &= \|EV_1 - \Gamma(EV_1)\| = \|EV + \beta k - \Gamma(EV + \beta k)\| \\ &= \|EV + \beta k - (EV + \beta^2 k)\| = \beta(1 - \beta)k \\ tol_1/tol_0 &= \beta \end{aligned}$$

- Newton will immediately "strip away" the irrelevant constant k
- Switch to Newton whenever tol_1/tol_0 is sufficiently close to β

The Fixed Point (poly) Algorithm

Fixed Point poly Algorithm

1. Successive contraction iterations

$$EV_{k+1} = \Gamma_\theta(EV_k)$$

   until EV_k is in the domain of attraction
   (i.e. when tol_{k+1}/tol_k is close to β)

2. Newton-Kantorovich (quadratic convergence)

$$EV_{k+1} = EV_k - (I - \Gamma')^{-1}(I - \Gamma)(EV_k)$$

   until convergence
   (i.e. when $\|EV_{k+1} - EV_k\|$ is close to machine precision)

STEP 4: Analytical derivative of Bellman operator

Derivative of Bellman operator, $\bar{\Gamma}'$

- Needed for the NK iteration
- In the discretized approximation, $\bar{\Gamma}'$ is a $n \times n$ matrix with partial derivatives of the $n \times 1$ vector function $\bar{\Gamma}(V_\theta)$ with respect to the $n \times 1$ vector $\bar{V}_\theta$
- $\bar{\Gamma}'_\theta$ is simply β times the choice probability weighted state transition probability matrix

$$\bar{\Gamma}'_\theta = \beta \sum_j \Pi(j) .* P(j)$$

- One line of code in MATLAB
- A similar matrix can be derived for Γ'

STEP 1-4: MATLAB implementation of $\bar{\Gamma}_\theta$ and $\bar{\Gamma}'_\theta$

```
function [V1, pk, dBellman_dV]=bellman_iv(V0, mp, u, P)
  vK= u(:,1) + mp.beta*P{1}*V0;     % Value of keeping
  vR= u(:,2) + mp.beta*P{2}*V0;     % Value of replacing

  % Recenter logsum
  maxV=max(vK, vR);
  V1=(maxV + log(exp(vK-maxV)   +   exp(vR-maxV)));

  % If requested, compute keep probability
  if nargout>1
    pk=1./(1+exp((vR-vK)));
  end

  % If requested, compute derivative of Bellman operator
  if nargout>2
    dBellman_dV=mp.beta*(P{1}.*pk + P{2}.*(1-pk));
  end
end
```

STEP 1-4: MATLAB implementation of Γ_θ and Γ'_θ

```
function [ev, pk, dbellman_dev]=bellman_ev(ev0, mp, u, P)
  vK= u(:,1) + mp.beta*ev0;       % Value off keep
  vR= u(:,2) + mp.beta*ev0(1);    % Value of replacing

  % Need to recenter logsum by subtracting max(vK, vR)
  maxV=max(vK, vR);
  V=(maxV + log(exp(vK-maxV)  +  exp(vR-maxV)));
  ev=P{1}*V; % compute expected value of keeping
             % ev(1) is the expected value of replacing

  % If requested, also compute choice probability
  if nargout>1
    pk=1./(1+exp((vR-vK)));
  end

  % If requested, compute derivative of Bellman operator
  if nargout>2
    dbellman_dev=mp.beta*(P{1}.*pk');
    % Add additional term for derivative wrt ev(1),
    % since ev(1) enter logsum for all states
    dbellman_dev(:,1)=dbellman_dev(:,1)+mp.beta*P{1}*(1-pk);
  end
end
```

STEP 5: Provide analytical gradients of likelihood

Simple use of chain rule:

3. Gradients (wrt utility parameters) - similar to standard logit

$$\partial \ell_i^1(\theta)/\partial \theta_1 = \sum_t \sum_j [y_{j(it)} - P(j|x_{it}, \theta)] \partial v(x_{it}, j)/\partial \theta_1$$

2. Derivative of the choice specific value function

$$\partial v(j)/\partial \theta_1 = \partial u(j)/\partial \theta_1 + \beta \Pi(j) \partial \bar{V}/\partial \theta_1$$

- $\partial u(j)/\partial \theta_1$, is trivial to compute
- $\partial \bar{V}_\theta/\partial \theta$ can be obtained by the implicit function theorem

$$\partial \bar{V}_\theta/\partial \theta = [I - \bar{\Gamma}'_\theta]^{-1} \partial \bar{\Gamma}/\partial \theta$$

where $[I - \bar{\Gamma}'_\theta]^{-1}$ is a by-product of the N-K algorithm!!!.

1. Derivative of Bellman operator wrt. θ_1

$$\partial \bar{\Gamma}/\partial \theta_1 = \beta \sum_j P(j) \cdot \partial u(j)/\partial \theta_1$$

where $\cdot$ is the element by element product

STEP 5: MATLAB implementation of scores

```
function score = score(data, mp, P, pk, px_j, V0, du, dBellman_dV);
  y_j=[(1-data.d) data.d];  % choice dummies [keep replace]

  % Compute scores (use chain rule - three steps)

  % STEP 1: derivative of bellman operator wrt. utility parameters
  dbellman=pk.*du(:,:,1) + (1-pk).*du(:,:,2);
  if strcmp(mp.bellman_type, 'ev');
    dbellman=P{1}*dbellman;
  end

  % STEP 2: derivative of fixed point, V, wrt. utility parameters
  dV=(speye(size(dBellman_dV)) - dBellman_dV)\dbellman;

  % STEP 3: derivative of log-likelihood wrt. utility parameters
  score=0;
  for j=1:size(y_j, 2);
    dv= du(:,:,j) + mp.beta*P{j}*dV;
    score = score+ (y_j(:,j)-px_j(:,j)).*dv(data.x,:);
  end
end
```

STEP 6: BHHH

- Recall Newton-Raphson

$$\theta^{g+1} = \theta^{g} - \lambda \left(\Sigma_i H_i \left(\theta^{g} \right) \right)^{-1} \Sigma_i s_i \left(\theta^{g} \right)$$

- Berndt, Hall, Hall, and Hausman, (1974):
  Use outer product of scores as approx. to Hessian

$$\theta^{g+1} = \theta^{g} + \lambda \left(\Sigma_i s_i s_i' \right)^{-1} \Sigma_i s_i$$

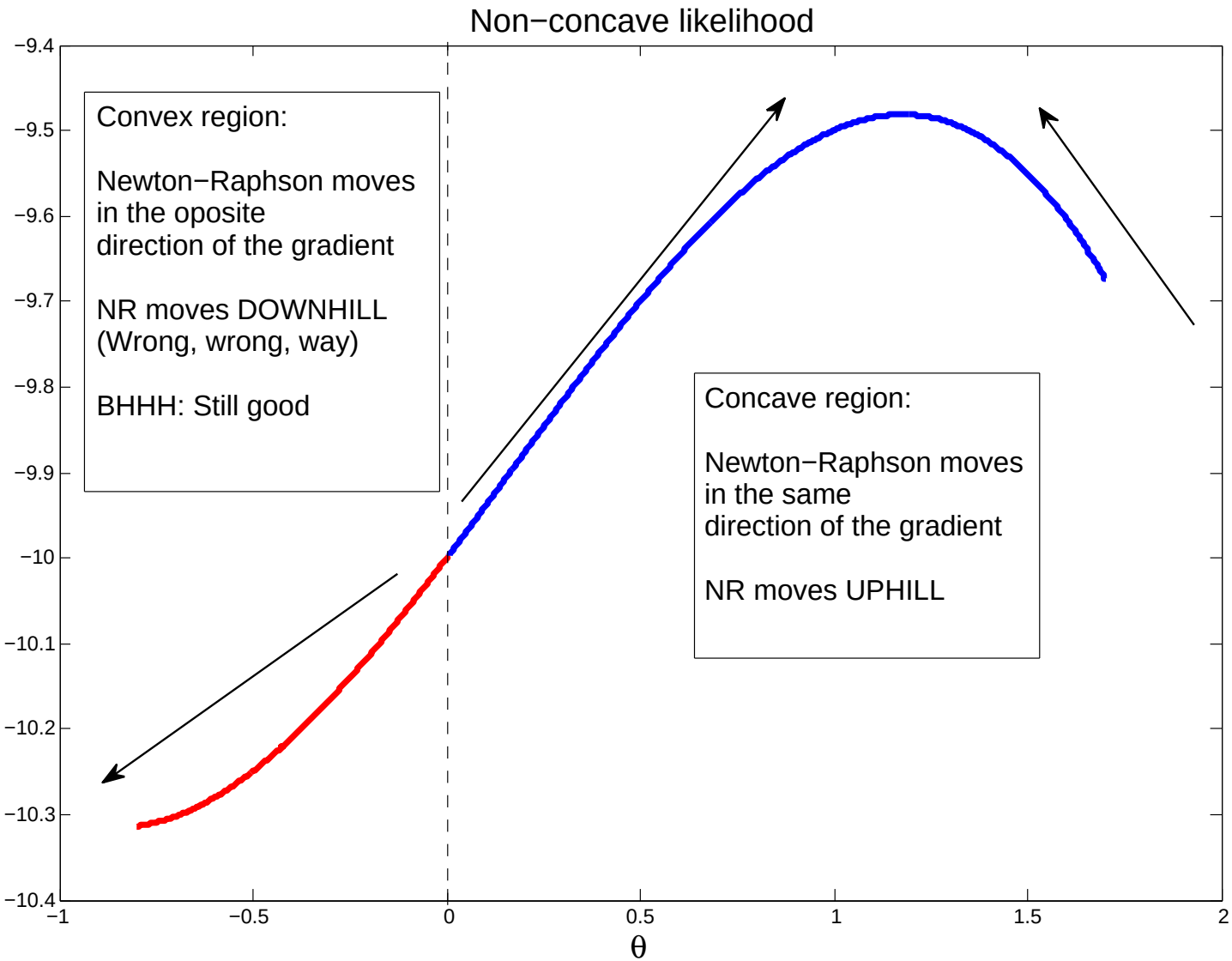
- Why is this valid? Information identity:

$$-E \left[H_i \left(\theta \right) \right] \quad = \quad E \left[s_i \left(\theta \right) s_i \left(\theta \right)' \right]$$

  (valid for MLE if model is well specified)

STEP 6: BHHH

Some times linesearch may not help Newtons Method



STEP 6: BHHH

Advantages

- $\Sigma_i s_i s_i'$ is always positive definite
  I.e. it always moves uphill for λ small enough
- Does not rely on second derivatives

Disadvantages

- Only a good approximation
  - At the true parameters
  - for large N
  - for well specified models (in principle only valid for MLE)
- Only superlinear convergent - not quadratic

We can always use BHHH for first iterations and the switch to BFGS to update to get an even more accurate approximation to the hessian matrix as the iterations start to converge.

STEP 6: BHHH

Advantages

- $\Sigma_i s_i s_i'$ is always positive definite
  I.e. it always moves uphill for λ small enough
- Does not rely on second derivatives

Disadvantages

- Only a good approximation
  - At the true parameters
  - for large N
  - for well specified models (in principle only valid for MLE)
- Only superlinear convergent - not quadratic

We can always use BHHH for first iterations and the switch to BFGS to update to get an even more accurate approximation to the hessian matrix as the iterations start to converge.

STEP 6: BHHH

Use BHHH!

Convergence!

```
>> run_busdata
Structural Estimation using busdata from Rust(1987)
Bustypes       = [ 1  2  3  4 ]
Beta           =    0.99990
n              =  175.00000
Sample size    = 8156.00000

Method nfxp (mle)
    Param.                      Estimates          s.e.         t-stat
-------------------------------------------------------------------------
    RC                             9.7915        1.2689         7.7168
    c                              1.3488        0.3460         3.8982
    p                   (1)        0.1070        0.0034        31.2111
    p                   (2)        0.5152        0.0055        93.0533
    p                   (3)        0.3622        0.0053        68.0413
    p                   (4)        0.0143        0.0013        10.8947
    p                   (5)        0.0009        0.0003         2.6469
-------------------------------------------------------------------------
log-likelihood    = -8607.88844
runtime (seconds) =    0.07882
g'*inv(h)*g       = 7.26689e-09
```

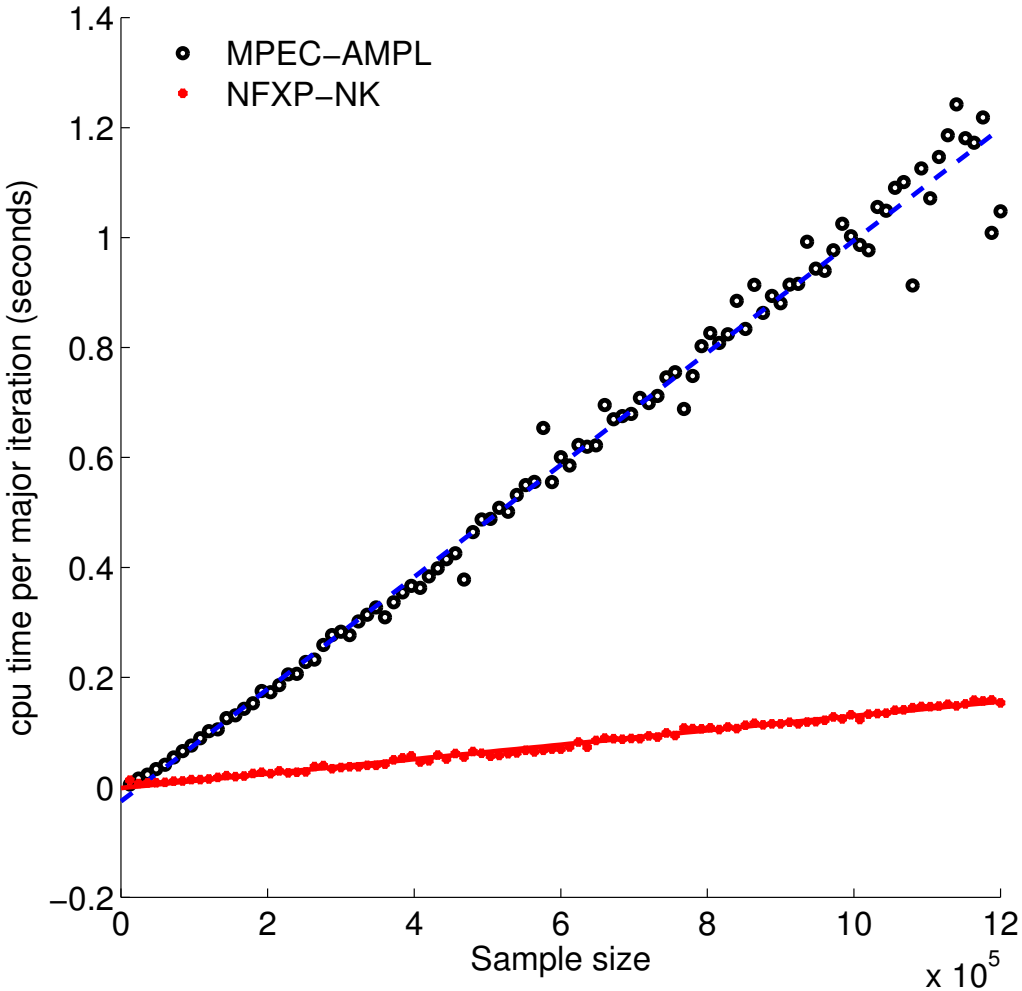
MPEC versus NFXP-NK: sample size 6,000

β	Converged (out of 1250)	CPU Time (in sec.)	# of Major Iter.	# of Func. Eval.	# of Bellm. Iter.	# of N-K Iter.
			MPEC-Matlab			
0.975	1247	1.677	60.9	69.9		
0.985	1249	1.648	62.9	70.1		
0.995	1249	1.783	67.4	74.0		
0.999	1249	1.849	72.2	78.4		
0.9995	1250	1.967	74.8	81.5		
0.9999	1248	2.117	79.7	87.5		
			MPEC-AMPL			
0.975	1246	0.054	9.3	12.1		
0.985	1217	0.078	16.1	44.1		
0.995	1206	0.080	17.4	49.3		
0.999	1248	0.055	9.9	12.6		
0.9995	1250	0.056	9.9	11.2		
0.9999	1249	0.060	11.1	13.1		
			NFXP-NK			
0.975	1250	0.068	11.4	13.9	155.7	51.3
0.985	1250	0.066	10.5	12.9	146.7	50.9
0.995	1250	0.069	9.9	12.6	145.5	55.1
0.999	1250	0.069	9.4	12.5	141.9	57.1
0.9995	1250	0.078	9.4	12.5	142.6	57.5
0.9999	1250	0.070	9.4	12.6	142.4	57.7

MPEC versus NFXP-NK: sample size 60,000

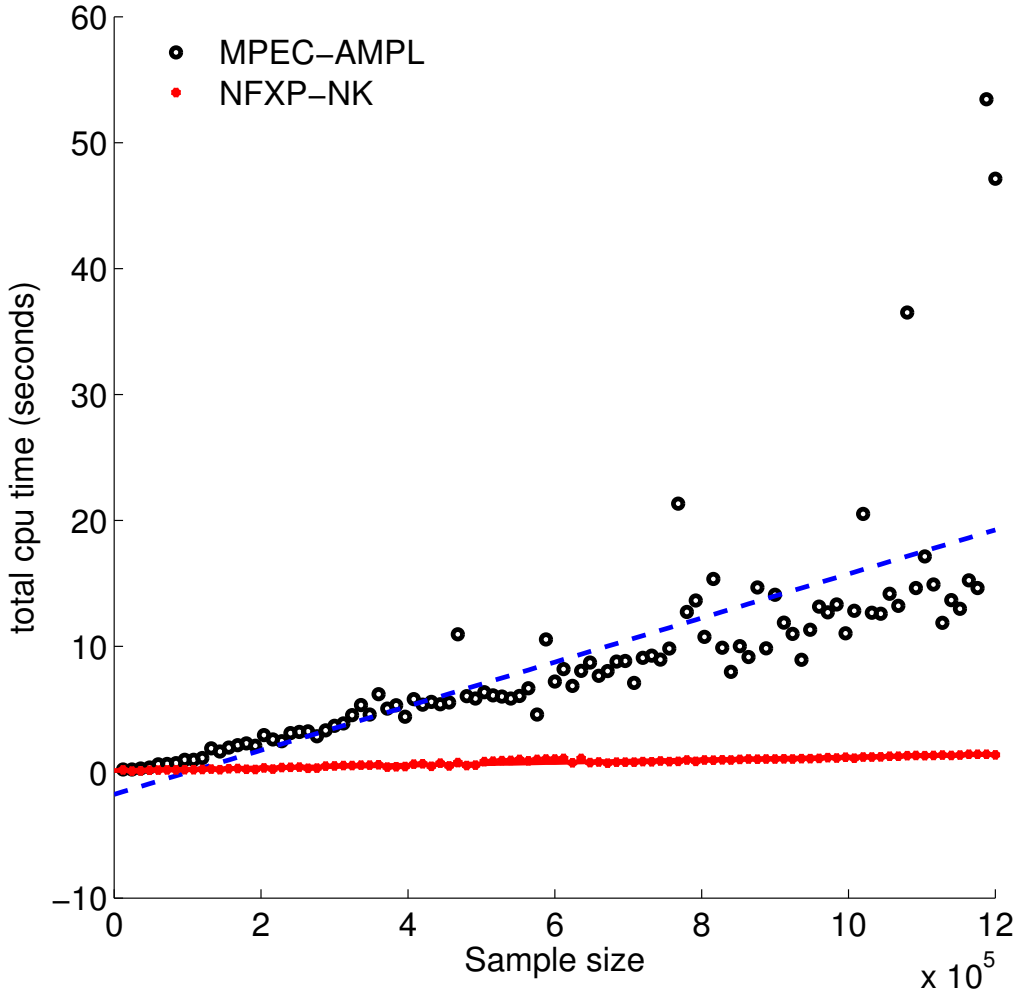
β	Converged (out of 1250)	CPU Time (in sec.)	# of Major Iter.	# of Func. Eval.	# of Bellm. Iter.	# of N-K Iter.
			MPEC-AMPL			
0.975	1247	0.53	9.2	11.7		
0.985	1226	0.76	13.9	32.6		
0.995	1219	0.74	14.2	30.7		
0.999	1249	0.56	9.5	11.1		
0.9995	1250	0.59	9.9	11.2		
0.9999	1250	0.63	11.0	12.7		
			NFXP-NK			
0.975	1250	0.15	8.2	11.3	113.7	43.7
0.985	1250	0.16	8.4	11.4	124.1	46.2
0.995	1250	0.16	9.4	12.1	133.6	52.7
0.999	1250	0.17	9.5	12.2	133.6	55.2
0.9995	1250	0.17	9.5	12.2	132.3	55.2
0.9999	1250	0.17	9.5	12.2	131.7	55.4

CPU time is linear sample size



$T_{NFXP} = 0.001 + 0.13x$ $(R^2 = 0.991)$, $T_{MPEC} = -0.025 + 1.02x$ $(R^2 = 0.988)$.

CPU time is linear sample size



$T_{NFXP} = 0.129 + 1.07x$ $(R^2 = 0.926)$, $T_{MPEC} = -1.760 + 17.51x$ $(R^2 = 0.554)$.

Summary remarks

Su and Judd (Econometrica, 2012) used an inefficient version of NFXP

- that solely relies on the method of successive approximations to solve the fixed point problem.

Using the efficient version of NFXP proposed by Rust (1987) we find:

- MPEC and NFXP-NK are similar in performance when the sample size is relatively small.
- NFXP does not slow down as $\beta \to 1$

Desirable features of MPEC

- Ease of use by people who are not interested in devoting time to the special-purpose programming necessary to implement NFXP-NK.
- Can easily be implemented in the intuitive AMPL language.

Inference

- NFXP: Trivial to compute standard errors by inverting the Hessian from the unstrained likelihood (which is a by-product of NFXP).
- MPEC: Standard errors can be computed inverting the bordered Hessian Reich and Judd (2019): Develop simple and efficient approach to compute confidence intervals.

MPEC does not seem appropriate when estimating life cycle models