Mata routines for solution of nonlinear systems using interval methods

Matthew J. Baker - Department of Economics
Hunter College and the Graduate Center, CUNY

July 31, 2014

Multiple equilibria models

Current research interest: models with multiple equilibria.

- Examples:
  - Social interactions (teen smoking). (Bisin, Moro, and Topa, 2011)
  - Market outcomes that have game-theoretic or strategic underpinnings. (Bajari, Hong, and Ryan 2010)
  - Macro-models with multiple equilibria (e.g., Diamond, 1982), examples in Cooper (1999)
- For given values of parameters, explanatory variables, and error terms, *observed outcome y may not be the only outcome consistent with parameters and data.*

Estimation of Multiple Equilibria models

State-of-the-art:

1. Solve the model, given parameters and data. *Find all solutions!*
2. Probabilistically weight the observed outcome solution in forming moment conditions or likelihood.
3. Repeat 1-2 as objective is maximized.

Estimation of Multiple Equilibria models

State-of-the-art:

1. Solve the model, given parameters and data. *Find all solutions!*
2. Probabilistically weight the observed outcome solution in forming moment conditions or likelihood.
3. Repeat 1-2 as objective is maximized.

Requirement: a reliable way to find *all* solutions of the model.

Nonlinear solvers

There are a lot of nonlinear system solvers out there, but:

- Most find *a* solution to a system - finding all system solutions is a bit more tricky!
- Sometimes clumsy to use if one wants to work in Stata.

Goal: create a means of reliably finding all solutions to a nonlinear system within Stata (actually, Mata).

Interval methods?

First, why interval methods?

Interval methods?

First, why interval methods? Alternative ways of solving systems:

- Homotopy methods:
  - Polynomial systems.
  - Very elegant - use Bezout's theorem to guarantee all solutions.
  - Drawback: hard to use constraints, numerical stability, hard to adapt to non-polynomial cases.
- Polynomial ideals/Groebner bases:
  - Extremely elegant.
  - Drawback: numerical stability.

Interval methods?

First, why interval methods? Alternative ways of solving systems:

- Homotopy methods:
  - Polynomial systems.
  - Very elegant - use Bezout's theorem to guarantee all solutions.
  - Drawback: hard to use constraints, numerical stability, hard to adapt to non-polynomial cases.
- Polynomial ideals/Groebner bases:
  - Extremely elegant.
  - Drawback: numerical stability.
- **Interval methods**:
  - Slow. Brute-force, but...
  - Details are easy to grasp.
  - Numerically reliable, constraints easy to include, flexible.

Intervals and zeros: basics

Take an interval $x = [x] \in [\underline{x}, \overline{x}]$. Consider the problem of finding the zeros of some continuous, differentiable $f(x)$ in this interval.

1. Consider a list of intervals $[x]$.
2. Use *interval arithmetic* to compute largest and possible values that $f(x)$ could take in the intervals.
3. For each interval in the list, if $\underline{x} \leq 0$ and $\overline{x} \geq 0$, subdivide x and retest. Otherwise, throw out $[x]$.
4. Repeat until all remaining boxes are smaller than some specified size, or there are no more intervals. Solutions are bracketed by small intervals.

If $f(x)$ is multivariate, $x = [x]$ will be a vector of intervals - a "box".

Interval computations and functions

Interval arithmetic largely consists of operating on upper and lower bounds:

- Addition: $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. Example: $[0, 1] + [3, 4] = [3, 5]$.
- Multiplication: $[x][y] = [\min S, \max S]$, $S = [\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}]$. Example: $[1, 3][3, 4] = [3, 12]$.
- More complex operations are simple to program, especially if they are monotone: $e^{[x]} = [e^{\underline{x}}, e^{\overline{x}}]$.

Package `int_utils` and accompanying help files explain different operations.

Parallel computation with intervals

Backbone of `intsolver`: `int_utils` and another package `rowmat_utils`.

- `int_utils` interval operations on vectors, matrices.
- `rowmat_utils` is a collection of matrix operations where a matrix is written as a single row of a larger matrix. Idea: compute the inverse, say, of many (say 1000) smaller matrices all at once in parallel.

Will be clear how this works (hopefully) from examples. `int_utils` and `rowmat_utils` have expansive help files.

Inclusion functions

One can use interval computations to develop an "inclusion function $[f]([x])$," where $f([x]) \subseteq [f]([x])$. Bounds values of the domain of f given range $[x]$. Most inclusion functions are "pessimistic" and not unique. From MKC (2009):
$x(1-x) = x - x^2$.

- $x(1-x)$: $[0,1](1-[0,1]) = [0,1][0,1] = [0,1]$
- $x - x^2$: $[0,1] - [0,1][0,1] = [0,1] - [0,1] = [-1,1]$
- True range is $[0, \frac{1}{4}]$!

But generall *convergence* of $[f]([x])$ to $f([x])$ as interval shrinks occurs at a factor > 1.

`intsolver` inclusion function

`intsolver` represents multivariate problems using a Taylor-expansion based inclusion function. If $[x]$ is a box, $[x] = [(\underline{x}_1, \overline{x}_1), (\underline{x}_2, \overline{x}_2), \ldots, (\underline{x}_n, \overline{x}_n)]$ and x_c is the center of the box ($x_c = \frac{\overline{x} - \underline{x}}{2}$), we have:

$$f([x]) \subseteq f(x_c) + [J]([x])([x] - x_c)$$

If one can program versions of the function and Jacobian, `intsolver` uses an interval Gauss-Seidel method on the above function to iteratively narrow potential boxes.

More precisely...

In an N-dimensional nonlinear system, pick a variable $i \in N$ and an equation $j \in N$ in an N-dimensional system. Then:

$$f_j([x]) = f_j(x_c) + \sum_{k=1}^{N} [J]_{jk}([x])([x]_k - x_{ck})$$

Or, viewed as a recursion:

$$[\tilde{x}]_i = \frac{f_j([x]) - \sum_{k=1,k\neq i}^{N} [J]_{jk}([x])([x]_k - x_{ck})}{[J]_{ji}([x])}$$

Update by computing $[\tilde{x}]_i$ and then updating: $[x]_i \leftarrow [\tilde{x}]_i \cap [x]_i$. If this doesn't work, subdivide $[x]_i$, and keep going with a larger list of intervals.

Using `intsolver`

Usage is via a `mata` structure, similar to how one uses `moptimize`. An example: suppose one wishes to find all zeros of the function:

$$f(x) = \begin{pmatrix} 1 - 2x_1x_2 - 3x_1^2 \\ 1 - x_1^2 - 3x_2^2 \end{pmatrix}$$

Which has Jacobian:

$$J(x) = \begin{pmatrix} -2x_2 - 6x_1 & -2x_1 \\ -2x_1 & -6x_2 \end{pmatrix}$$

One now needs to program ordinary versions of the above, and their interval counterparts, in a particular way.

A note on arguments...

`intsolver` works with sets of points in parallel. Functions should be written with a single matrix as an argument, with rows representing points:

$$x = \begin{pmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ \vdots & \vdots \\ x_{1D} & x_{2D} \end{pmatrix}$$

This allows `intsolver` to do stuff like Newton iteration in parallel on a large number of potential points.

Programming the functions

For the example, from the `mata` prompt:

```
: real matrix fun(real matrix x,real scalar i)
{
if (i==1) return(1:-2*x[,1]:*x[,2]:-3*x[,1]:^2)
if (i==2) return(1:-x[,1]:^2:-3*x[,2]:^2)
}
```

Scalar argument following the set of points x is an equation number. Note how the function computes in parallel across many points.

Programming part 2

The jacobian is then:

```
: real matrix jac(real matrix x,real scalar i,
 real scalar j)
{
if (i==1 & j==1) return(-2*x[,2]:-6:*x[,1])
if (i==1 & j==2) return(-2*x[,1])
if (i==2 & j==1) return(-2*x[,1]])
if (i==2 & j==2) return(-6*x[,2])
}
```

Here, i, j indicate the ith row and jth column of the jacobian, and x again carries x_1, x_2 points as rows.
The next step is to program interval versions of these functions.

Arguments again...

Interval functions should be constructed the same way - so that each row of a matrix x can be regarded as an interval. That is:

$$x = \begin{pmatrix} \underline{x}_{11} & \overline{x}_{11} & \underline{x}_{21} & \overline{x}_{21} \\ \underline{x}_{12} & \overline{x}_{12} & \underline{x}_{22} & \overline{x}_{22} \\ \vdots & \vdots & & \\ \underline{x}_{1D} & \overline{x}_{1D} & \underline{x}_{2D} & \overline{x}_{2D} \end{pmatrix}$$

x is a $D \times 2N$ matrix, where D is a number of points, and N is the dimension of the problem. Odd columns are lower bounds, even columns are upper bounds on an interval for each variable. Function needs x, row (and column) labels, and a final argument, d, which controls "outward rounding."

Programming part 3 - interval function

```
real matrix fun_I(real matrix x,real scalar i, real scalar d)
{
X1=x[,1::2];X2=x[,3::4]
if (i==1) {
A=2*int_mult(X1,X2,d)
I=J(rows(A),2,1)
B=int_sub(I,A,d)
C=3*int_pow(X1,2,d)
D=int_sub(B,C,d)
return(D)  }
if (i==2) {
A=int_mult(X1,X1,d)
B=3*int_pow(X2,X2)
I=J(rows(A),2,1)
C=int_sub(I,A,d)
D=int_sub(C,B,d)
return(D)  }
}
```

Operations defined in `int_utils` help.

Programming part 4 - interval jacobian

```
: real matrix jac_I(real matrix x,real scalar i,real scalar j, real scalar d)
{
X1=x[,1::2];X2=x[,3::4];r=rows(X1)
if (i==1 & j==1) {
A=int_mult(J(r,2,-2),X2,d)
B=int_mult(J(r,2,-6),X1,d)
C=int_add(A,B,d)
        return(C) }
if (i==1 & j==2) {
A=int_mult(J(r,2,-2),X1,d)
        return(A) }
if (i==2 & j==1) {
A=int_mult(J(r,2,-2),X1,d)
return(A) }
if (i==2 & j==2) {
A=J(rows(X1),2,-6)
B=int_mult(A,X2,d)
return(B)  }
}
```

Problem definition

Having programmed everything in, we now can define a problem. From within Mata:

```
: Prob=int_prob_init()
: int_prob_args(Prob,2)
: int_prob_f_Iform(Prob,&fun_I())
: int_prob_jac_Iform(Prob,&jac_I())
: int_prob_f(Prob,&fun())
: int_prob_jac(Prob,&jac())
```

Then, the problem can be solved once some boxes are passed along:

```
: Ival=(-100,100) \ (-100,100)
: int_prob_ival(Prob,Ival)
: int_solve(Prob)
```

The problem has been solved! What do the solutions look like?

Problem solutions

The solutions are returned as intervals:

```
:  int_prob_ints_vals(Prob)
```

1	2	3	4
-.7249408	-.7249155	.3976849	.3976926
-.4291268	-.4291181	-.5214909	-.5214856
.4291181	.4291268	.5214856	.5214909
.7249155	.7249408	-.3977003	-.3976926
.7249155	.7249408	-.3976926	-.3976849

Typically, there is some redundancy/closeness/overlap in boxes. One can control how small boxes are in options (described in help file). Here, box size is $1e-4$ (default setting).

Refining solutions to a point

It has been shown (Pandian, 1984) that the midpoints of solution boxes constitute great and reliable points for Newton iteration. One can do this and see resulting points using the commands:

```
: int_newton_iter(Prob)
: int_prob_pts_vals(Prob)
```

1	2
-.7249360956	.3976881766
-.4291212844	-.5214898284
.4291212844	.5214898284
.7249360956	-.3976881766

Less rigor, more speed...

If one wishes, one can simply start with a set of random points covering an interval and do Newton iteration - and avoid using interval methods all together. In my experience, this actually works quite well in finding a, or all, solutions to a system:

```
: Prob2=int_prob_init()
: int_prob_args(Prob2,2)
: int_prob_f(Prob2,&func())
: int_prob_jac(Prob2,&jac())
: randpoints=-100:+200*runiform(10,2)
: int_prob_init_pts(Prob2,randpoints)
: int_prob_method(Prob2,"newton")
: int_newton_iter(Prob2)
: int_prob_pts_vals(Prob2)
```

Mata packages to be uploaded on SSC soon!

- `intsolver`, with a help file and a .do file of examples, including a number of common nonlinear test functions.
- Some more complex examples include additional arguments (Diamond 1982 model)
- `int_utils`, a collection of interval operations.
- `rowmat_utils`, write and operate on a sequence of matrices in a parallel, where each row of a meta matrix is a matrix.

Other materials

Do file to accompany packages contains examples, including examples of how to pass along additional arguments, and treatment of some famous test problems.
Many details of package, such as control of rounding errors, how to do floating point arithmetic, etc. can be controlled by user. Detailed in the help files.

Also, a Stata function - `dagsolve`

Reason for development of `int_solver` - usage in estimation or other packages. Example: `dagsolve`.
Uses `intsolver` extensively - finds all solutions to "discrete action" games with an arbitrary number of players, each of whom has an arbitrary number of actions.
A very difficult computational problem - mixed strategies for games with more than two players require repeated solution of polynomial systems on the $0, 1$ interval. Also to be posted!

Conclusion

Other plans:

- Form additional Stata routines to estimate some standard nonlinear, multiple-solution models.
- Develop a parser or otherwise automate the process of writing interval functions.
- Expand the algorithm to include some other interval techniques, preconditioning, etc.
- Tools for checking internal consistency of user-programmed functions. (debuggers)

If you have a nonlinear system that needs to be solved, please let me know! I am happy to help and am looking for functions!