

NOTES ON THE NUMERICAL COMPUTATIONS IN
“BID RIGGING — AN ANALYSIS OF
CORRUPTION IN AUCTIONS”

YVAN LENGWILER

University of Basel
Dept. of Economics (WWZ)
Petersgraben 51
CH-4003 Basel
Switzerland
yvan.lengwiler@unibas.ch

ELMAR WOLFSTETTER

Humboldt University at Berlin
Institute of Economic Theory I
Spandauer Str. 1
D-10178 Berlin
Germany
wolfstetter@wiwi.hu-berlin.de

April 2005

CONTENTS

1	The problem	2
2	Root finding methods	2
2.1	The Gauss-Newton method	3
2.2	Steepest descent method	3
2.3	A hybrid method	4
2.4	The Levenberg-Marquardt method	4
3	Optimizing the step size	5
4	Finding an initial guess	5
4.1	Starting from a linear bid function	5
4.2	Starting from a non-linear bid function	6
4.3	Grid search	7
4.4	A variation: progressively finer grid method	7
5	The results	7
6	The program	8

We want to find the solution to the delayed differential equation of the first-price auction. For the numerical exercise, we assume a uniform distribution of the valuations throughout. The differential equation then simplifies to

$$\begin{aligned}
D(v) := & \phi'(v)(v - (1 - \alpha)\beta(v) - \alpha\beta(\phi(v)))(n - 1)\phi(v)^{n-2} \\
& - (1 - \alpha)\beta'(v)\phi(v)^{n-1} \\
& + \alpha(v - \beta(v))(n - 1)v^{n-2} \left[\frac{\bar{\psi}'(v)}{2} - 1 \right] \\
& - (1 - \alpha)(n - 1)v^{n-2}(\bar{\psi}(v) - v) \\
& + \alpha(n - 1)(n - 2)v^{n-3} \left[(\bar{\psi}(v) - v)v - \int_v^{\bar{\psi}(v)} \beta(y)dy \right] = 0. \tag{1}
\end{aligned}$$

Note that the evaluation of (1) is not completely trivial because we first need to determine $\bar{\psi}(v)$ and $\phi(v)$. $\bar{\psi}$ involves the inverse of β , so we will have to make sure that our candidate β is always invertible. ϕ demands a little more. It is the solution of a fixed point problem,

$$\phi(v) + \beta(\phi(v)) - 2\beta(v) = 0. \tag{2}$$

Because the left-hand side of this equation is monotonic in $\phi(v)$, and is negative if $\phi(v) = 0$, and positive if $\phi(v) = v$ (as long as there is bid shading, $\beta(v) < v$), we can use the bi-section method and determine $\phi(v)$ to an arbitrary precision. The required precision is determined in the code by `phiEps` and set by default to machine precision. For the remaining, we draw heavily from the splendid book by Heath (2002).

2 ROOT FINDING METHODS

To make problem (1) suitable for numerical analysis we search for approximate solutions within the class of strictly increasing, continuous, piecewise linear functions that satisfy the boundary condition $\beta(0) = 0$. Let $G := \{0, 1/g, \dots, (g - 1)/g, 1\}$, for some $g \in \mathbb{N}$, be a uniform grid on the unit interval. A piecewise linear function is defined by the numbers $\{\beta(v) : v \in G\}$. This transforms the infinite-dimensional problem (1) into a g -dimensional problem. You can select the fineness of the grid on valuation space by changing the constant `g` in the source code.

All iterative procedures start from some initial “guess” (more on this later). Let β_0 be the initial bid function, and β_1, β_2, \dots denote the bid functions along an iteration. A root finding algorithm is a rule that prescribes how to get from β_i to β_{i+1} . The idea is to do this in such a way that the sequence of β_i approaches the true solution in the process. We have implemented several standard methods for such problems. You can select the method by setting the switch `rootmethod` to a value between 0 and 3.

2.1 The Gauss-Newton method

A standard method for moving from β_i to β_{i+1} is the Gauss-Newton method. It involves the Jacobian of D_i with respect β_i , which we denote with J_i . To approximately compute the components of the Jacobian, for each $v_k := k/g \in G$, we compute D_i at two points, namely β_i^+ and β_i^- , given by $\beta_i^+(v_k) := \beta_i(v_k) + \delta$, $\beta_i^-(v_k) := \beta_i(v_k) - \delta$, and $\beta_i^+(v) = \beta_i^-(v) = \beta_i(v)$ for all $v \neq v_k$.¹ The Jacobian is then approximately given by

$$J \approx \begin{bmatrix} \frac{D_1(\beta_1^+) - D_1(\beta_1^-)}{2\delta} & \dots & \frac{D_1(\beta_g^+) - D_1(\beta_g^-)}{2\delta} \\ \vdots & \ddots & \vdots \\ \frac{D_g(\beta_1^+) - D_g(\beta_1^-)}{2\delta} & \dots & \frac{D_g(\beta_g^+) - D_g(\beta_g^-)}{2\delta} \end{bmatrix}.$$

The iteration proceeds according to the following rule

$$\beta_i \mapsto \beta_{i+1} := \beta_i - \tau_i J_i^{-1} D_i. \quad (3)$$

τ_i is a positive number called the step size. As explained later, we optimize over the step size τ_i .

2.2 Steepest descent method

Another standard method is the method of steepest descent. Compute the sum of squared errors,

$$\text{SSE}_i := \frac{1}{2} \sum_{v \in G} D_i(v)^2,$$

(the division by 2 is a normalization that will make sense later). We aim at minimizing SSE. To that avail we compute the gradient of SSE with respect to $\{\beta_i(v) : v \in G\}$. As before, we approximate the gradient by computing a discrete difference of the components of the bid function ($\pm\delta$): we compute $\text{SSE}_i^+(\beta_i, v)$ as the SSE that would result if $\beta_i(v) \mapsto \beta_i(v) + \delta$ for a given $v \in G$, and $\text{SSE}_i^-(\beta_i, v)$ as the SSE that would result if $\beta_i(v) \mapsto \beta_i(v) - \delta$. The gradient is then approximately equal to

$$\nabla \text{SSE}(\beta_i) := \left(\frac{\text{SSE}_i^+(\beta_i, v) - \text{SSE}_i^-(\beta_i, v)}{2\delta} \right)_{v \in G}.$$

With this information, we can iterate according to the following rule

$$\beta_i \mapsto \beta_{i+1} := \beta_i - \tau_i \nabla \text{SSE}(\beta_i). \quad (4)$$

We move here into the direction in which SSE decreases the fastest locally, hence the name of the method. τ_i is again the step size, which we optimize.

¹ δ is called `diffDelta` in the program, and is equal to `diffDelta = 1E-10` by default; you are free to change this parameter.

2.3 A hybrid method

The Gauss-Newton method finds the minimum of a quadratic function in one iteration. For non-quadratic problems, the initial point has to be sufficiently close to the solution. If not, the method might not converge at all. So this method is fast if we are close to the solution, but quite demanding in terms of choice of starting point.

Compared to the Gauss-Newton method, the steepest descent method is slow. Like the Gauss-Newton method, it exhibits global convergence for quadratic problems (though not in one iteration step). The advantage of steepest descent over the Gauss-Newton method is that steepest descent is much less demanding with respect to the starting point. Even if we are far from the solution, the method will point into more or less the right direction² and initially converge at a decent speed. Only when we get close to the solution does convergence speed deteriorate.

These observations suggest a hybrid method, that combines the advantages of the steepest descent method in the early phase of the process, and later switches to the Gauss-Newton method. We switch from steepest descent to Gauss-Newton when there has been no significant improvement (`switchEps` = 1E-12) over sufficiently many consecutive iterations (`switchKeep` = 5).

2.4 The Levenberg-Marquardt method

This method is similar in spirit to the hybrid method we have just discussed, but instead of completely switching from one method to the other, it moves more gradually between the two. The iteration rule is given by

$$\beta_i \mapsto \beta_{i+1} := \beta_i - \tau_i ((1 - \lambda_i) J_i^T J_i + \lambda_i I)^{-1} J_i^T D_i, \quad \lambda_i \in (0, 1). \quad (5)$$

Note that (5) is a convex combination between the Gauss-Newton rule (3), when $\lambda_i = 0$, and the steepest descent rule (4), when $\lambda_i = 1$.³

The strategy of the method is to dynamically change λ_i in a smart fashion. Define $\mu_i := \lambda_i(1 - \lambda_i)^{-1}$. We start from some initial μ_0 (`muInit` = 1E-3). If the SSE improves from one iteration to the next, we decrease μ by dividing it by some factor $m > 0$ (in the code, m is called `muFactor` = 10), thus moving closer to the Gauss-Newton method; if the SSE deteriorates we increase μ_i by multiplying it with m , thus moving into the direction of the steepest descent method.

We slightly vary this strategy because we have found that it works better. We decide about increasing or decreasing μ_i not by comparing the SSE from one iteration to the next. Instead, we decrease μ_i only if the current SSE is better than the best SSE that has been encountered so far during the whole process; we increase it otherwise.

²Of course, we might get stuck on a local minimum if SSE is not unimodal.

³Note that $\nabla \text{SSE} = J^T D$. Here is where the division by 2 in the definition of SSE comes in.

Optimization of the step size is quite important, particularly as long as we are far from the solution. Moreover, our root finding methods might suggest a step that would make the next bid function β_{i+1} locally decreasing or that might violate weak bid shading. Non-monotonicity in particular would cause trouble in further calculations by making the bid function not invertible. We proceed as follows: in each iteration step, we first determine the maximum size of τ that still guarantees weak monotonicity and bid shading; call this step size τ^* . Then we optimize by searching for a step τ in the interval $(0, \tau^*)$ which minimizes SSE.⁴

A good algorithm for finding the minimum along a line if the function is unimodal is golden section search. You can select this option by setting `taumethod = 1` along with the required precision (`tauEps`).

Because we cannot be sure that SSE is unimodal in our search direction, we have also implemented a more expensive procedure (`taumethod = 0`), which we call “brute force”: we compute SSE for step sizes $\tau \in \{\tau^* s S^{-1} : s = 1, \dots, S-1\}$, and we select the best one. S is called `tausteps` in the program. Note that we restrict $0 < \tau^* S^{-1} \leq \tau \leq \tau^* (S-1) S^{-1} < \tau^*$. $\tau = 0$ would immediately lead to an infinite loop; the algorithm would be stuck. $\tau = \tau^*$ is very likely to lead to a locally flat bid function. This makes inverting the bid function impossible, and also raises the possibility that τ^* would become zero in the next iteration step. The algorithm would again be stuck. Yet, the strategy of disallowing extreme step sizes is only partially successful. It happens on rare occasions that τ^* becomes smaller than machine precision. The algorithm stops in such a case.

If we are close to the solution, it can be that the smallest step is still too large so that the algorithm would overshoot. If the SSE is larger even with the smallest allowed step, we re-optimize the step size by evaluating the SSE at steps $\tau \in \{\tau^* s S^{-k} : s = 1, \dots, S-1\}$, with $k = 2$. We keep increasing k until the iteration leads to an improvement of the SSE. However, in no case do we allow τ to become smaller than $\tau^* \text{SSE} / (g + 1)$. This limit allows smaller steps once the error is small, but avoids getting stuck on very small steps as long as the error is still large.

4 FINDING AN INITIAL GUESS

4.1 *Starting from a linear bid function*

Because we cannot hope to have global convergence, the choice of the initial bid function β_0 has to be done with care, and we should try to start from a point which is as close as possible to the solution. A simple idea is to start from the solution of a simpler problem

⁴If this requirement does not put a constraint on the step size, we allow τ^* to be at most equal to `GRANDMAXTAU = 2.0`, so the bid function is changed by at most twice as much as the step size suggested by the chosen method.

which we can solve explicitly. We start from the solution of the restricted first-price auction with type I corruption only. The rationale for this choice is the hope that adding the possibility of type II corruption does not alter the solution in a too extreme fashion. With the uniform distribution of valuations, this bid function is linear and given by

$$\beta_0(v) := \frac{n-1}{n-\alpha} v. \quad (6)$$

More generally, you can choose to start from any linear bid function by setting `initmethod = 0`. The slope of this initial bid function is set in the variable `slope`, which is by default set to the slope defined in (6).

4.2 Starting from a non-linear bid function

One could also start from a non-linear bid function. To do that, you need to write an alternative to the procedure `InitLinear`, maybe along these lines:

```
private static void InitPower()
{
    Console.WriteLine(">>> Filling in power bid function " +
        "as an initial guess...");
    for (int j=1; j<=g; j++)
        betavec[j, 0] = slope * Math.Pow(val(j), exponent);
    compute();
}
```

You will have to declare the constant `exponent` in the beginning of the code somewhere (preferably in the neighborhood of the declaration of `slope`), and also change a part of the `Main()` method as follows,

```
switch (initmethod)
{
    case 0:
        InitLinear();
        break;
    case 1:
        RunGridSearch();
        OutputResult();
        break;
    case 2:
        InitPower();
        break;
}
```

Setting `initmethod = 2` would then select the power bid function as the starting point.

4.3 Grid search

An alternative to a fixed initial guess is to run a grid search (select `initmethod = 1`). In a grid search, we determine the RMSE of a large number of bid functions and select the one with the smallest RMSE as the starting point. More precisely, for the first dimension $i = 1$ (that is, $v_1 = 1/g$), we select Q equally spaced bids b_1 strictly between 0 and v_1 (Q is called `GridSearchSteps` in the code). For each of these bids, we then select Q bids for the second dimension, again equally spaced, and strictly between b_1 and v_2 , in order to observe strictly monotonicity and bid shading. We do this through all g dimensions. This gives rise to Q^g bid functions. It is obvious that with a reasonably fine grid on the valuation space, say $g = 50$, this kind of grid search is not feasible because already with $Q = 2$ we would have to evaluate more than 10^{15} bid functions. So grid search is feasible only if we work with a coarse grid on the valuations, at least initially.

4.4 A variation: progressively finer grid method

Since it is important to start from an initial bid functions which is close to the final solution, and it is easier to find the root for a problem with less dimensions rather than more, one can follow a strategy of starting with a relatively coarse grid on the valuation space, and making the grid progressively finer. To consider an extreme example, suppose we start with $g = 1$. This amounts to searching for a solution within the class of linear functions. Apply a root finding method as described before. Then subdivide the grid on the valuation space, say by 4. So now $g = 4$ and we linearly interpolate the bid function for the new points in the valuation grid. We then again apply one of the root finding methods, and again subdivide the valuation space. We keep on doing this until we reach a reasonable fineness of the grid on the valuation space. This method has potentially two advantages. Firstly, it allows us to start with a very coarse grid, making the grid search a feasible choice for the initial guess. Secondly, after each subdivision of the valuation space, we start from a bid function which should be rather close to the solution for the new valuation grid.

In the code, the progressively finer grid method is controlled by two parameters: `NbSubdivisions` is the number of rounds in which the valuation grid is made progressively finer, and `SubdivisionFactor` is the factor by which it is made finer in each round. Thus, if we start with a grid of $g = 3$, which we subdivide twice with a factor of four, we end up with $3 \times 4^2 = 48$ points in the valuation space. Setting `NbSubdivisions = 0` turns off the progressively finer grid method.

5 THE RESULTS

We have not made very good experiences with the progressively finer grid method, so we do not use it (`NbSubdivisions = 0`). Instead we run a rather fine grid on the valua-

tion space to begin with ($g = 200$). This rules out the grid search for the starting point (`initmethod = 0`), and we use a linear function following (6).

We find that the Levenberg-Marquardt method is the best choice for our problem. An iteration step is quite slow, but it makes up for this drawback with fast convergence. If there is an equilibrium, it finds it within a small number of iterations. The steepest descent method works also, although convergence is very slow and it does not seem to generate the same amount of precision.

For the optimization of the step size, we find that golden section search works well, so we use it throughout. The brute force method often works too, but is somewhat inferior.⁵

Finally, we stop the iteration if no improvement has occurred for sufficiently many (`keepiter = 20`) iterations, or after a maximum of `maxiter = 100` iterations. We then report the iteration step with the smallest SSE that we have detected so far.

Figure 1 depicts the approximate equilibrium bid functions; Table 1 reports the precision we achieve with these computations. For each n we have investigated, there is a borderline α at which the SSE increases dramatically when we increase α by just 0.0001 beyond this limit. These borderline α s are identified in the table.

6 THE PROGRAM

An explanation about our choice of platform is in order. The program is essentially a simple procedure, a list of commands that is executed in a given sequence. Given this fact, it may seem surprising that we chose $C\sharp$ for implementation. Object orientation — one of the more important aspects of $C\sharp$ — is not important for our task, and is in fact not used. So it appears that $C\sharp$ is an odd choice for such a project. It would be more natural to implement it with Gauss or Matlab. The decisive advantage of the .NET Framework (and the $C\sharp$ compiler that comes with it) is that it is free. As a consequence, everyone (with a Windows computer and internet access) can run our program without spending a nickel or wasting time trying to get spending approved by the University bureaucracy. To use the $C\sharp$ code, you need Microsoft's .NET Framework SDK Version 1.1, which is available from <http://msdn.microsoft.com/netframework/>. You may find it comfortable to use an IDE for changing the program (we did, at least). The standard IDE for the .NET Framework is Microsoft's Visual Studio .NET, which is very good but also very expensive. A free alternative can be downloaded from <http://www.icsharpcode.net/OpenSource/SD/>.

A ZIP archive containing the program and computed results is available for download from our websites (at the time of this writing, <http://www.wvz.unibas.ch/witheo/yvan/research.html>). The archive contains this description as a PDF file. In addition, it contains a file called `Main.cs`, which is the $C\sharp$ source code, and an Excel file called

⁵The parameters are `tauEps = 1E-2` for the golden section search and `tausteps = 10` for the brute force method. Brute force step size optimization often seems to work more harmoniously with the steepest descent root finding method, but since we do not use steepest descent, we have also no need for brute force step size optimization.

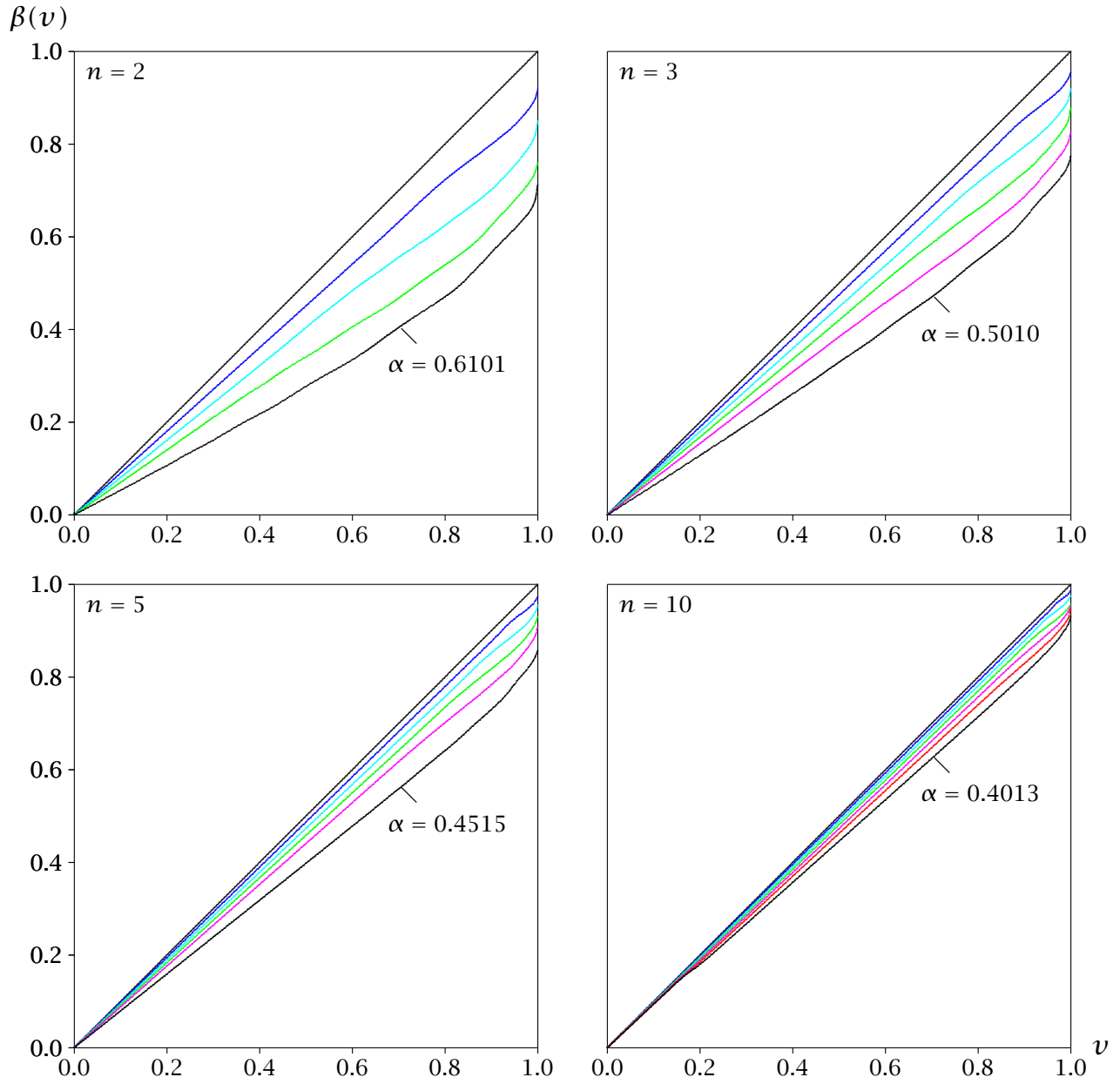


Figure 1: Approximate equilibrium bid functions for the first-price auction with type I and type II corruption, for $n \in \{2, 3, 5, 10\}$ and $\alpha \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, and, for each n , for the smallest α for which we found a solution.

Table 1: Precision of the computations for the first-price auction with both types of corruption.

$n = 2$	α	# iterations	SSE	RMSE	max absolute error	mean error
	0.9	32	4.40×10^{-22}	2.09×10^{-12}	2.04×10^{-11}	-1.43×10^{-16}
	0.8	34	2.23×10^{-17}	4.71×10^{-10}	3.94×10^{-9}	$+8.86 \times 10^{-14}$
	0.7	100	9.51×10^{-15}	9.73×10^{-9}	1.06×10^{-7}	$+6.23 \times 10^{-11}$
	0.6101	43	6.17×10^{-12}	2.48×10^{-7}	3.50×10^{-6}	-1.78×10^{-8}
.....						
	0.6100	25	9.76×10^{-8}	3.12×10^{-5}	2.96×10^{-4}	-7.41×10^{-6}
	0.6	29	3.04×10^{-8}	1.74×10^{-5}	1.84×10^{-4}	-2.06×10^{-6}
	0.5	61	5.51×10^{-2}	2.34×10^{-2}	6.33×10^{-2}	-1.95×10^{-2}
$n = 3$	α	# iterations	SSE	RMSE	max absolute error	mean error
	0.9	46	3.82×10^{-28}	1.95×10^{-15}	1.80×10^{-14}	$+6.49 \times 10^{-17}$
	0.8	31	4.39×10^{-25}	6.61×10^{-14}	3.72×10^{-13}	-1.97×10^{-16}
	0.7	50	9.73×10^{-24}	3.11×10^{-13}	2.77×10^{-12}	-2.28×10^{-16}
	0.6	37	2.15×10^{-24}	1.46×10^{-13}	1.38×10^{-12}	$+2.20 \times 10^{-15}$
	0.5010	50	3.98×10^{-16}	1.99×10^{-9}	2.82×10^{-8}	-1.37×10^{-10}
.....						
	0.5009	58	5.73×10^{-9}	7.55×10^{-6}	6.58×10^{-5}	$+3.13 \times 10^{-7}$
	0.5	25	1.00×10^{-1}	3.15×10^{-2}	2.22×10^{-1}	-6.96×10^{-3}
$n = 5$	α	# iterations	SSE	RMSE	max absolute error	mean error
	0.9	19	7.18×10^{-7}	8.45×10^{-5}	8.85×10^{-4}	-8.19×10^{-7}
	0.8	99	6.32×10^{-21}	7.93×10^{-12}	1.11×10^{-10}	$+4.07 \times 10^{-13}$
	0.7	98	5.85×10^{-20}	2.41×10^{-11}	2.59×10^{-10}	-2.41×10^{-12}
	0.6	16	8.84×10^{-17}	9.38×10^{-10}	7.90×10^{-9}	-1.15×10^{-10}
	0.5	99	7.64×10^{-19}	8.72×10^{-11}	8.17×10^{-10}	-5.11×10^{-12}
	0.4515	98	2.55×10^{-18}	1.59×10^{-10}	1.91×10^{-9}	-1.93×10^{-11}
.....						
	0.4514	18	1.24×10^{-5}	3.51×10^{-4}	4.47×10^{-3}	$+9.85 \times 10^{-6}$
	0.4	13	2.11×10^{-1}	4.59×10^{-2}	2.30×10^{-1}	-1.86×10^{-2}
$n = 10$	α	# iterations	SSE	RMSE	max absolute error	mean error
	0.9	19	2.46×10^{-7}	4.95×10^{-5}	3.33×10^{-4}	-1.16×10^{-5}
	0.8	44	9.75×10^{-18}	3.11×10^{-10}	1.74×10^{-9}	-7.96×10^{-11}
	0.7	98	6.07×10^{-18}	2.46×10^{-10}	1.47×10^{-9}	-5.77×10^{-11}
	0.6	100	4.15×10^{-17}	6.43×10^{-10}	3.78×10^{-9}	-1.55×10^{-10}
	0.5	98	1.25×10^{-16}	1.12×10^{-9}	6.38×10^{-9}	-2.97×10^{-10}
	0.4013	37	2.95×10^{-15}	5.42×10^{-9}	3.08×10^{-8}	-1.61×10^{-9}
.....						
	0.4012	43	8.74×10^{-10}	2.95×10^{-6}	2.93×10^{-5}	-4.56×10^{-8}
	0.4	100	2.17×10^{-5}	4.65×10^{-4}	2.09×10^{-3}	-2.44×10^{-4}

results.xls, containing some statistics and graphs of the computations. Also included are plain text files which are copies of the output produced by the program. The naming conventions of these files are as follows: the file with name x-0y.txt contains the computation for $n = x$ and $\alpha = 0.y$.

All parameters and choices of methods are specified at the beginning of the source code (Table 2), so you do not need to modify the program itself when making your choices.

Table 2: Section of the source code in which all parameters are declared.

```
// =====
// In this section, the parameters of the problem are given.
// You are free to change these parameters.

const byte n = 2;           // # bidders (greater than or equal to 2)
const double alpha = 0.9;   // the bidder's share (between 0.0 and 1.0)

// =====
// Next we define the parameters defining the discrete approximation, the methods
// for find the root and the initial guess, and the stopping conditions.
// These settings should only be changed with caution.

// --- discrete valuation space -----

static int g = 200;         // initial # of points in the space of valuations
const int SubdivisionFactor = 4; // factor by which valuation space is subdivided
const int NbSubdivisions = 0; // number of times the space is subdivided

// "delta" for finite difference derivatives
const double diffDelta = 1E-10; // this is the "delta" we use to compute the
// derivatives

// --- initial bid function -----

const int initmethod = 0;   // choice of method for initial guess
// initmethod = 0 : linear guess
// initmethod = 1 : grid search

const double slope = (n-1)/(n-alpha); // slope of linear initial guess

const int GridSearchSteps = 50; // parameter for the grid search
// GridSearchSteps = # values that are tried for
// each point in the valuation grid
```

→ *continued on next page*

(Table 2 continued)

```
// --- step size optimization -----
const int taumethod = 1;          // taumethod = 0: brute force
                                  // taumethod = 1: golden section search

const int tausteps = 10;         // number of stepsizes we try out in each iteration step
                                  // when using brute force
const double tauEps = 1E-2;      // required precision when using golden section search

const double GRANDMAXTAU = 2.0; // maximum step size under all circumstances

// --- root finding -----
const int rootmethod = 3;        // choice of method for root finding
                                  // rootmethod = 0 : steepest descent method
                                  // rootmethod = 1 : Gauss-Newton method (inverse Jacobian)
                                  // rootmethod = 2 : hybrid method
                                  // rootmethod = 3 : Levenberg-Marquardt method

// switching rule for hybrid method
const double switchEps = 1E-12;
const double switchKeep = 5;

// coefficients for Levenberg-Marquardt method
const double initMu = 1E-3;      // initial mu
const double muFactor = 10;      // factor by which mu is multiplied or divided
const double maxMu = 1E+100;    // upper bound for mu
const double minMu = Double.Epsilon; // lower bound for mu

// --- stopping rules -----
const double eps = Double.Epsilon; // stop iterating if SSE < eps
                                  // (Double.Epsilon is machine precision, so this
                                  // effectively turns off this stopping rule; you
                                  // can choose a larger value for eps)

const int keepiter = 20;         // stop iterating if no improvement in so many steps
const int maxiter = 100;        // stop after so many steps in any case

// =====
```

REFERENCES

HEATH, M. T. (2002): *Scientific Computing. An Introductory Survey*. McGraw-Hill, 2nd edn.