A report on using parallel MATLAB for solutions to stochastic optimal control problems

Azzato, Jeffrey D. and Krawczyk, Jacek B.
Victoria University of Wellington

12. August 2008

A REPORT ON USING PARALLEL MATLAB® FOR SOLUTIONS TO STOCHASTIC OPTIMAL CONTROL PROBLEMS

JEFFREY D. AZZATO & JACEK B. KRAWCZYK

Abstract. Parallel MATLAB® is a recent MathWorks™ product enabling the use of parallel computing methods on multicore personal computers. `SOCSol` is the generic name of a suite of MATLAB® routines that can be used to obtain optimal solutions to continuous-time stochastic optimal control problems. In this report, we compare the performance of a new version of `SOCSol` utilising parallel MATLAB® with that of another version not using parallel computing methods.

2008 Working Paper
School of Economics and Finance

JEL Classification: C63 (Computational Techniques), C87 (Economic Software).
AMS Categories: 93E25 (Computational methods in stochastic optimal control).
Authors' Keywords: Computational economics, Approximating Markov decision chains.

This report documents 2008 research into
Computational Economics Methods
directed by Jacek B. Krawczyk and
supported by URF GMS Grant 32665

Correspondence should be addressed to:

Jacek B. Krawczyk. Faculty of Commerce and Administration, Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand. Fax: +64-4-4635014 Email: `J.Krawczykatvuw.ac.nz` Webpage: http://www.vuw.ac.nz/staff/jacek_krawczyk

Contents

Introduction

Performing independent calculations in parallel should make it possible to dramatically reduce overall computation times for certain types of problem, such as those solvable by dynamic programming. One class of problems solvable by dynamic programming are the stochastic optimal control (SOC) problems. These are frequently either analytically unsolvable, or sufficiently complicated for analytical solutions to be impractical. Consequently, good numerical methods for solving SOC problems are desirable.

A method for approximating a solution to a given continuous-time SOC problem was developed in [KW97] and subsequently improved in [Kra01]. The method works by discretising the problem in both state and time, forming an equivalent Markov chain and then solving this Markov chain by backward induction. While elementary, this approach has the advantages of generality and providing feedback (as opposed to open-loop) solutions. A disadvantage of this approach is that it is subject to the "curse of dimensionality" when the state space is effectively multidimensional (i.e., when $d > 1$ —see Section 1): halving the discretisation steps of all state variables simultaneously increases computation time by a factor of 2^d.[1] So implementing this approach for high-dimensional SOC problems rapidly becomes computationally costly.

However, parallel computing may provide a means of fighting this phenomenon. In backward induction, computing the optimal return for a given state and time only requires knowledge of optimal returns at the next time. The state-independence of this calculation means that it can be performed in parallel across several different states simultaneously. So increasing the number of parallel workers by a factor of 2^d when halving all state discretisation steps should leave overall computation time relatively unchanged. Of course, it is not always realistic to exponentially increase computing resources. Nonetheless, the growing prevalence of multicore processors and small computing grids makes this approach increasingly attractive.

A suite of MATLAB®[2] routines implementing the method described in [Kra01] was developed under the name `SOCSol4L`. The use of this suite is described in [AK06]. With the introduction of extensive parallel computing support in MATLAB®, a new suite of MATLAB® routines implementing the same method using parallel computing methods is being developed under the name `Parallel SOCSol`. The use of this new suite is described in [AK08].

This article compares the computation times of the two suites of routines for a pair of deterministic test problems. Each test problem was solved with each suite for various discretisations on each of two different computers. This should help to quantify the potential of applying parallel computing methods in this area of research.

[1] In particular, if $d = 1$, halving the state discretisation step doubles computation time—an outcome not shared by all approaches based on dynamic programming.

[2] See [Mat92] for an introduction to MATLAB®.

1. The Test Problems

The method employed by the `SOCSol4L` and `Parallel SOCSol` suites of MATLAB® routines deals with finite-horizon, free terminal state SOC problems having the form

$$\min_{\mathbf{u}} J(\mathbf{u}, \mathbf{x}_0) = \mathbb{E}\left[\int_0^T f\big(\mathbf{x}(t), \mathbf{u}(t), t\big)\,\mathrm{d}t + h\big(\mathbf{x}(T)\big) \,\middle|\, \mathbf{x}(0) = \mathbf{x}_0 \right] \tag{1}$$

subject to

$$\mathrm{d}\mathbf{x} = g\big(\mathbf{x}(t), \mathbf{u}(t), t\big)\,\mathrm{d}t + b\big(\mathbf{x}(t), \mathbf{u}(t), t\big)\,\mathrm{d}\mathbf{W} \tag{2}$$

where $T > 0$ is the length of the optimisation horizon, $\mathbf{x}\colon [0,T] \to X \subseteq \mathbb{R}^d$ is state evolution, $\mathbf{u}\colon [0,T] \to U \subseteq \mathbb{R}^c$ is control and $\mathbf{W}$ is a standard Wiener process in which precisely $N \leqslant d$ components are not constantly zero (so the state space has dimension d, the control has dimension c and N state variables are affected by noise). The method also allows for constraints on the control and state variables (both local and mixed).

In order to provide a basis for comparison, we solved two deterministic (i.e., $\mathbf{W} \equiv \mathbf{0}$) test problems of this form using both `SOCSol4L` and `Parallel SOCSol`. These are formulated as described below.

1.1. The First Test Problem.

The first test problem is an elementary linear-quadratic optimal control problem having a one-dimensional state space. It involves the minimisation of

$$J_1\left(u, \frac{1}{2}\right) := \int_0^1 \frac{u(t)^2 + x(t)^2}{2}\,\mathrm{d}t + \frac{x(1)^2}{2} \tag{3}$$

with respect to u, subject to

$$\mathrm{d}x = u(t)\,\mathrm{d}t, \text{ and} \tag{4}$$

$$x(0) = \frac{1}{2}. \tag{5}$$

1.2. The Second Test Problem.

The second test problem is similar to the first, but involves a two-dimensional state space in which the control u now affects the first state variable x_1 only indirectly through the second state variable x_2. It involves the minimisation of

$$J_2\left(u, \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) := \int_0^1 \frac{u(t)^2 + x_2(t)^2}{2}\,\mathrm{d}t + \frac{x_1(1)^2}{2} \tag{6}$$

with respect to u, subject to

$$\mathrm{d}\mathbf{x} = \begin{bmatrix} x_2(t) \\ u(t) \end{bmatrix}\mathrm{d}t, \text{ and} \tag{7}$$

$$\mathbf{x}(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{8}$$

2. Measuring Computation Times

Computation times were obtained for the test problems of Section 1 by treating the relevant solution routine (of either `SOCSol4L` or `Parallel SOCSol`) as a "black box," simply timing how long it took to run. This approach determines alterations in computation times as they would be experienced by a user. It would also be possible to directly measure alterations in computation times for those subroutines that have been "parallelised." While this might quickly quantify how much improvement could be obtained under a parallel implementation, it is unrealistic, as most programs involve some fixed overhead (e.g., for initilisation and saving output).

The `SOCSol4L` solution routine is structured as follows.

```
{Initialise parameters}

for {each state}
  {Compute terminal optimal return}
end;

for {each stage}
  for {each state}
    {Compute optimal return, optimal control and possibly
    Lagrange multipliers}
  end;
end;

{Save results}
```

The `Parallel SOCSol` solution routine modifies this basic structure as follows.

```
{Initialise parameters}
{Open a MATLAB pool}

parfor {each state}
  {Compute terminal optimal return}
end;
{Store terminal optimal return}

for {each stage}
  parfor {each state}
    {Compute optimal return (OR), optimal controls (OCs) and
    possibly Lagrange multipliers (LMs)}
  end;
  {Store OR, OCs and possibly LMs}
end;

{Close the MATLAB pool}
{Save results}
```

Here, each **parfor** loop executes in parallel. So if there are two MATLAB® workers in the pool, the loop can (in principle) execute in half of the time taken by the corresponding **for** loop. However, this decrease in computation time is balanced by increases in fixed overhead due to

1. the opening and closing of the pool of MATLAB® workers, and
2. the need to use temporary variables within the **parfor** loops, the content of which is subsequently transferred to the arrays of interest.

As a consequence, we expect that `Parallel SOCSol` will take longer than `SOCSol4L` to solve "small" SOC problems, but may be nearly twice as fast for "large" SOC problems when using two MATLAB® workers. Here, a "large" SOC problem is one in which the relevant solution routine (of either `SOCSol4L` or `Parallel SOCSol`) is run over a discrete state grid containing a sufficiently large number of points. Such grids can arise from either

1. high-dimensional state spaces (i.e., large numbers of state variables), or
2. fine discretisation of the state variable(s).

3. Comparison of Solution Routines

Here we compare the times taken by the solution routines of `SOCSol4L` and `Parallel SOCSol` to solve the test problems defined in Section 1. Each test problem was solved with each solution routine for various discretisations on each of two different computer systems. The full results are tabulated in Appendix E.

3.1. **The First Test Problem.** The times achieved for the first test problem are displayed in Figure 1. Several unsurprising trends are immediately clear from this figure, namely:

- Finer state discretisation (i.e., more state steps) increases computation time.
- Finer time discretisation (i.e., more time steps) increases computation time.
- The 2.1GHz Core™ 2 Duo computer system is faster than the 3.4GHz Pentium® 4 computer system.

Two other trends are also clear from Figure 1. The first of these is that `SOCSol4L` outperforms `Parallel SOCSol` when state-time discretisation is sufficiently coarse, and vice-versa when state-time discretisation is sufficiently fine. This is consistent with the heuristic analysis given in Section 2. Examination of the values tabulated in Appendix E suggests that the time improvements introduced by the parallel code in `Parallel SOCSol` outweigh the additional fixed overheads imposed when the discrete state-time grid used has of the order of 10000 points or more.

The second interesting trend apparent in Figure 1 is that for "large" choices of the state-time grid (i.e., 10000 points or more), using two workers in `Parallel SOCSol` yields much greater relative time reductions with the Core™ 2 Duo computer system than with the Pentium® 4 system. For example, if both state and time steps are set to 0.005 (yielding a state-time grid having $36000 > 10000$ points), using two workers instead of one yields only an additional 5% time reduction when operating on

the Pentium® 4 system (79% vs. 74%). However, using two workers instead of one yields an additional 32% time reduction when operating on the Core™ 2 Duo system (85% vs. 53%) — a marked decrease. This supports the hypothesis that computation time improvements obtained through parallel MATLAB® are likely to be greater with newer generation CPUs.

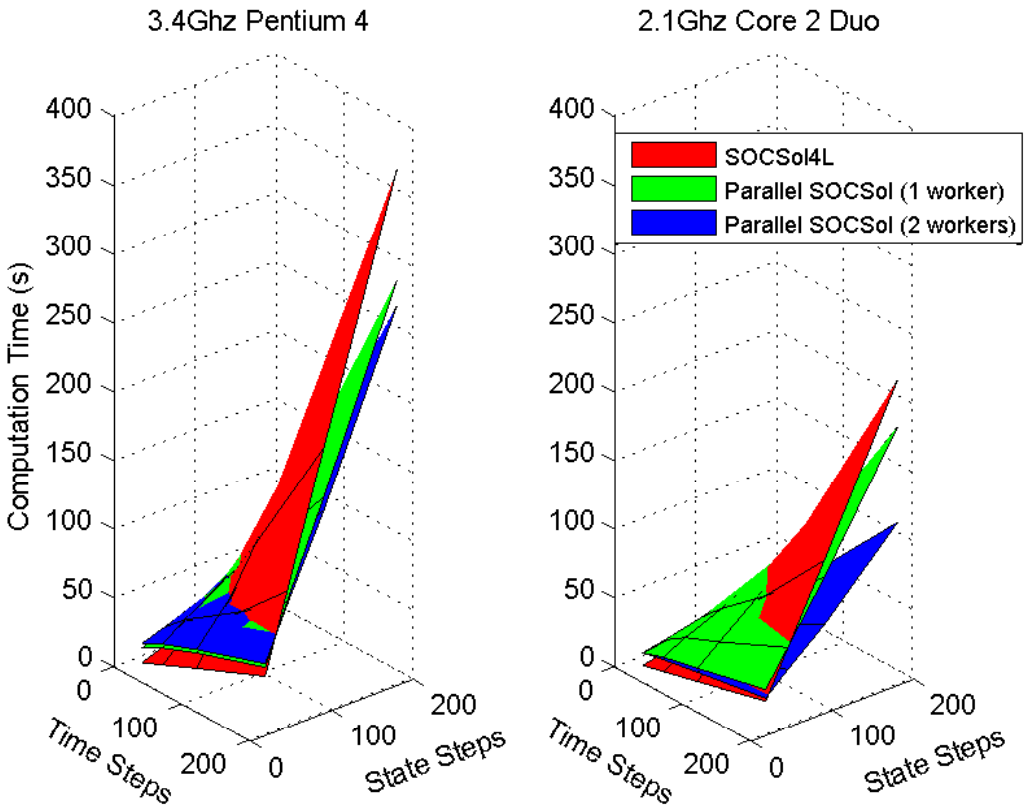

Figure 1: Computation times for the first test problem.

3.2. **The Second Test Problem.** The computation times achieved for the second test problem are displayed in Figure 2. While most trends present match those observed in Section 3.1, there is one key difference: `SOCSol4L` now outperforms `Parallel SOCSol` for "large" state-time discretisations tested when *only one* worker is employed by the latter. This not surprising, as running `Parallel SOCSol` with only one worker should not yield time reductions due to parallel computation (at least two workers should be needed for this), but is nonetheless subject to the additional fixed overhead associated with parallel code (see Section 2).

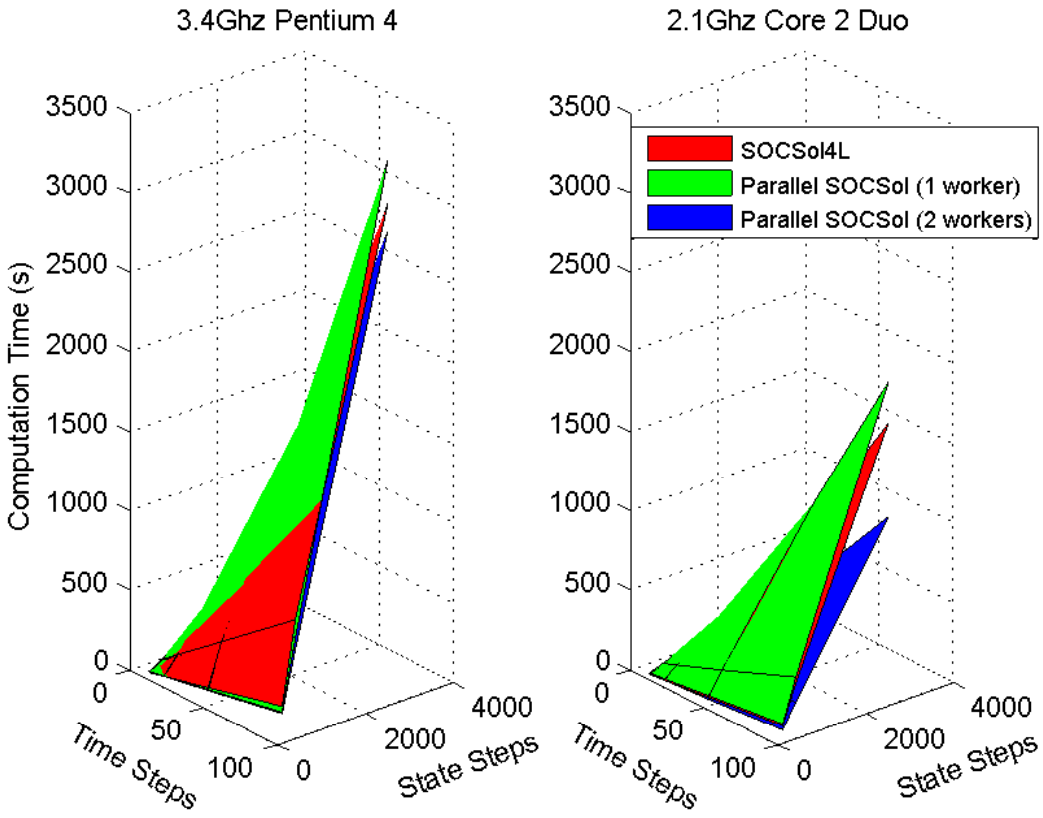

Figure 2: Computation times for the second test problem.

4. Summary

The use of **parfor** loops in MATLAB® implementations of the SOC problem solution method described in [Kra01] both introduces additional fixed overhead and the potential for reductions in loop running times. The experimental results presented here suggest that reductions in loop running times are likely to outweigh the additional fixed overhead for sufficiently large problems when at least two MATLAB® workers are used in parallel. The results are also consistent with the hypothesis that reductions in overall computation time are more likely to be achieved when using newer generation CPUs.

Appendix A. SOCSol4L Commands for the First Test Problem

The `SOCSol4L` commands used to obtain results for the first test problem are detailed below. This section should be read in conjunction with [AK06].

A.1. **Delta Function File.** This `.m` file defines the problem's dynamics $g(\mathbf{x}(t), \mathbf{u}(t), t)$. It is called `delta.m` and is written as follows.

```
function v = delta(u, x, t)
v = u;
```

A.2. **Instantaneous Cost Function File.** This `.m` file gives the instantaneous cost function $f(\mathbf{x}(t), \mathbf{u}(t), t)$. It is called `cost.m` and is written as follows.

```
function v = cost(u, x, t)
v = 0.5*u^2 + 0.5*x^2;
```

A.3. **Terminal State Function File.** This `.m` file gives the terminal state function $h(\mathbf{x}(T))$. It is called `term.m` and is written as follows.

```
function v = term(x)
v = 0.5*x^2;
```

A.4. **Solution Syntax.** It is easiest to write a MATLAB® script to set the `SOCSol4L` parameters and call the `SOCSol` solution routine. An example of such a script is given below for state and time discretisation steps each equal to 0.02. The **tic**; and **toc**; commands time how long the `SOCSol` solution routine takes to execute.

```
StateLB = -0.2;
StateUB = 0.7;
Options = {'TolFun' '1e-12'};
InitialControlValue = -0.5;
A = [];
b = [];
Aeq = [];
beq = [];
ControlLB = -Inf;
ControlUB = Inf;
UserConstraintFunctionFile = [];

StateStep = 0.02;           % This varies.
TimeStep = ones(1, 50)/50; % This varies.

tic;
SOCSol('delta', 'cost', 'term', StateLB, StateUB, StateStep,
  TimeStep, 'SOCSol4L_TestProb1_02_02', Options,
  InitialControlValue, A, b, Aeq, beq, ControlLB, ControlUB,
  UserConstraintFunctionFile);
toc;
```

Appendix B. Parallel SOCSol Commands for the First Test Problem

The `Parallel SOCSol` commands used to obtain results for the first test problem are detailed below. This section should be read in conjunction with [AK08].

B.1. **Function Files.** The delta function file, instantaneous cost function file and terminal state function file are defined as per Appendices A.1, A.2 and A.3 respectively.

B.2. **Solution Syntax.** A MATLAB® script is written exactly as shown in Appendix A.4, except that the Options vector is altered to either

```
Options = {'PoolSize' '1' 'TolFun' '1e-12'};
```

or

```
Options = {'PoolSize' '2' 'TolFun' '1e-12'};
```

as appropriate, and the name for the output files is altered sensibly, perhaps to 'ParallelSOCSol_PoolSize2_TestProb1_02_02'.

Appendix C. SOCSol4L Commands for the Second Test Problem

The SOCSol4L commands used to obtain results for the first test problem are detailed below. This section should be read in conjunction with [AK06].

C.1. **Delta Function File.** This `.m` file is called `delta.m` and is written as follows.

```
function v = delta(u, x, t)
v = [x(2), u];
```

C.2. **Instantaneous Cost Function File.** This `.m` file is called `cost.m` and is written as follows.

```
function v = cost(u, x, t)
v = 0.5*u^2 + 0.5*x(2)^2;
```

C.3. **Terminal State Function File.** This `.m` file is called `term.m` and is written as follows.

```
function v = term(x)
v = 0.5*x(1)^2;
```

C.4. **Solution Syntax.** An example of a MATLAB® script setting the SOCSol4L parameters and calling the solution routine SOCSol for state and time discretisation steps each equal to 0.01.

```
StateLB = [0.5, -0.5];
StateUB = [1.5, 0.5];
Options = {'TolFun' '1e-12'};
InitialControlValue = -0.5;
A = [];
b = [];
Aeq = [];
beq = [];
ControlLB = -Inf;
ControlUB = Inf;
UserConstraintFunctionFile = [];

StateStep = [0.01, 0.01];     % This varies.
TimeStep = ones(1, 100)/100; % This varies.

tic;
SOCSol('delta', 'cost', 'term', StateLB, StateUB, StateStep,
  TimeStep, 'SOCSol4L_TestProb2_01_01_01', Options,
  InitialControlValue, A, b, Aeq, beq, ControlLB, ControlUB,
  UserConstraintFunctionFile);
toc;
```

Appendix D. Parallel SOCSol Commands for the Second Test Problem

The `Parallel SOCSol` commands used to obtain results for the first test problem are detailed below. This section should be read in conjunction with [AK08].

D.1. **Function Files.** The delta function file, instantaneous cost function file and terminal state function file are defined as per Appendices C.1, C.2 and C.3 respectively.

D.2. **Solution Syntax.** A MATLAB® script is written exactly as shown in Appendix C.4, except that the Options vector is altered to either

```
Options = {'PoolSize' '1' 'TolFun' '1e-12'};
```

or

```
Options = {'PoolSize' '2' 'TolFun' '1e-12'};
```

as appropriate, and the name for the output files is altered sensibly, perhaps to 'ParallelSOCSol_PoolSize2_TestProb2_01_01_01'.

Appendix E. Tables of Computation Times

The computations times achieved for the test problems defined in Section 1 are given in Tables 1 and 2. Computation times were measured on two different computers for various combinations of discretisation parameters. The first computer had a 3.4Ghz

Pentium® 4 CPU, 2Gb of RAM and a Windows® XP Home Edition operating system. The second had a 2.1GHz Core™ 2 Duo CPU, 2Gb of RAM and a Windows® XP Professional operating system. In each case, the computation times were measured with only those background processes activating on system startup operating in the background.

It should be noted that each computation time quoted is *not* an average over several executions of the problem whose computation time was to be measured. This should be taken into account when making inferences using these values.

State Step Size	Time Step Size	3.4Ghz Pentium® 4			2.1GHz Core™ 2 Duo		
			Parallel SOCSol			Parallel SOCSol	
		SOCSol4L	One Worker	Two Workers	SOCSol4L	One Worker	Two Workers
0.005	0.005	374.110	293.772	275.259	221.465	187.655	117.880
	0.010	186.085	156.784	144.025	109.501	98.429	61.410
	0.020	96.188	95.684	88.971	55.430	61.333	40.087
	0.050	39.707	51.989	50.964	23.428	34.417	24.428
0.010	0.005	192.573	156.561	146.724	110.718	98.502	64.129
	0.010	95.631	88.511	84.612	54.021	56.051	38.572
	0.020	49.084	54.753	53.731	27.577	35.555	25.679
	0.050	20.607	31.715	33.756	11.612	20.979	17.134
0.020	0.005	98.678	91.469	88.606	57.323	58.068	41.951
	0.010	49.586	50.546	54.457	28.032	35.031	26.820
	0.020	24.924	36.536	37.627	14.073	24.653	19.663
	0.050	10.611	21.428	23.923	5.982	14.832	13.652
0.050	0.005	43.291	49.800	52.478	25.245	33.746	27.466
	0.010	22.032	33.117	35.674	12.532	22.121	19.297
	0.020	11.029	23.139	26.030	6.262	16.615	14.901
	0.050	4.590	15.227	18.448	2.599	11.638	11.620

Table 1: Computation times (in seconds) for the first test problem.

State Step Size	Time Step Size	3.4Ghz Pentium® 4			2.1GHz Core™ 2 Duo		
		SOCSol4L	Parallel SOCSol		SOCSol4L	Parallel SOCSol	
			One Worker	Two Workers		One Worker	Two Workers
0.020	0.010	3159.540	3431.787	2983.602	1778.700	2036.238	1186.886
	0.020	1594.308	1723.660	1510.596	930.570	1045.442	586.961
	0.050	670.036	766.477	674.027	391.984	461.538	268.596
	0.100	360.858	461.630	401.896	211.750	271.754	158.458
0.050	0.005	1266.210	1283.203	1112.251	723.980	762.454	430.487
	0.010	749.906	637.184	562.868	352.962	390.310	233.750
	0.020	374.455	325.713	289.249	175.794	197.446	115.560
	0.050	151.185	140.254	128.146	71.562	87.418	53.925
	0.100	74.929	79.804	75.165	35.247	50.794	33.638
0.100	0.010	236.144	199.106	191.231	110.920	123.245	85.075
	0.020	116.586	104.770	98.746	54.918	66.288	42.800
	0.050	47.624	48.506	47.829	22.261	31.352	23.156
	0.100	22.690	30.281	31.311	10.672	20.949	16.732

Table 2: Computation times (in seconds) for the second test problem.

References

[AK06] Jeffrey D. Azzato and Jacek B. Krawczyk. SOCSol4L: An improved MATLAB® package for approximating the solution to a continuous-time stochastic optimal control problem. Working paper, School of Ecnomics and Finance, Victoria University of Wellington, Dec 2006.

[AK08] Jeffrey D. Azzato and Jacek B. Krawczyk. Parallel SOCSol: A parallel MATLAB® package for approximating the solution to a continuous-time stochastic optimal control problem. Working paper, School of Ecnomics and Finance, Victoria University of Wellington, Jul 2008.

[Kra01] Jacek B. Krawczyk. A Markovian approximated solution to a portfolio management problem. *ITEM.*, 1(1), 2001. Available at http://www.item.woiz.polsl.pl/issue/journal1.htm on 22/04/2008.

[KW97] Jacek B. Krawczyk and Alistor Windsor. An approximated solution to continuous-time stochastic optimal control problems through Markov decision chains. Technical Report 9d-bis, School of Economics and Finance, Victoria University of Wellington, 1997. Available at http://ideas.repec.org/p/wpa/wuwpco/9710001.html on 31/07/2008.

[Mat92] The MathWorks Inc. *MATLAB®. High-Performance Numeric Computation and Visualization Software*, 1992.