SOCSol4L: An improved MATLAB package for approximating the solution to a continuous-time stochastic optimal control problem

Azzato, Jeffrey D. and Krawczyk, Jacek B.

Victoria University of Wellington

December 2006

SOCSol4L

AN IMPROVED MATLAB® PACKAGE FOR APPROXIMATING THE SOLUTION TO A CONTINUOUS-TIME STOCHASTIC OPTIMAL CONTROL PROBLEM

JEFFREY D. AZZATO & JACEK B. KRAWCZYK

Abstract. Computing the solution to a stochastic optimal control problem is difficult. A method of approximating a solution to a given continuous-time stochastic optimal control problem using Markov chains was developed in [Kra01]. This paper describes a suite of MATLAB® routines implementing this method.

2006 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 2005–2007 research into
Computational Economics Methods
directed by Jacek B. Krawczyk and
supported by VUW FCA FRG-05 (24644)

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

Computing the solution to a stochastic optimal control problem is difficult. A method of approximating a solution to a given continuous-time stochastic optimal control (SOC) problem using Markov chains was developed in [KW97] and subsequently improved in [Kra01]. This paper describes a suite of MATLAB®[1] routines implementing this method.

The suite of routines developed updates and extends that described in [AK06]. Further details on the underlying method are available in [Kra05]. In Appendix A we provide an example showing how to generate a test result from [KW97].

The method 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 $\mathbf{W}$ is a standard Wiener process. In the optimisation method, we also allow for constraints on the control and state variables (local and mixed).

Note: Throughout the paper, the dimension of the state space shall be denoted by d, the dimension of the control by c, the length of the horizon by T, and the number of variables which are affected by noise by N ($N \leqslant d$).

To solve (1) subject to (2) and local constraints, we developed a package of MATLAB® programmes under the name `SOCSol4L`. The package is composed of five main modules:

1. `SOCSol`
2. `ContRule`
3. `GenSim`
4. `ValGraph`
5. `LagranMap`

`SOCSol` discretises a given SOC problem and then solves this "discretisation." If the problem is constrained, it can also produce maps from states and times to Lagrange multipliers' values. `ContRule` derives graphs of continuous-time, continuous-state control rules from the `SOCSol` solution. `GenSim` simulates the continuous system using such a control rule (also derived from the `SOCSol` solution). `ValGraph` provides an automated means of computing expected values for the continuous system as initial conditions change. `LagranMap` produces graphs of Lagrange multipliers' values from Lagrange maps generated by `SOCSol`.

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

1. SOCSol

1.1. **Purpose.** SOCSol takes the given SOC problem and approximates it with a Markov chain, which it then solves. This results in a discrete-time, discrete-space control rule. SOCSol does not perform the interpolation necessary to convert this discrete-time, discrete-space control rule into a continuous-time, continuous-state control rule (this is done by GenSim).

1.2. **Syntax.** SOCSol is called as follows.

```
SOCSol('DeltaFunctionFile', 'InstantaneousCostFunctionFile',
    'TerminalStateFunctionFile', StateLB, StateUB, StateStep,
    TimeStep, 'ProblemFile', Options, InitialControlValue, A,
    b, Aeq, beq, ControlLB, ControlUB,
    'UserConstraintFunctionFile')
```

Note: It is easiest to define these arguments in a script, and then call that script in MATLAB®. See Appendix A for an example of this.

DeltaFunctionFile**:**

A string giving the name (no .m extension) of a file containing a MATLAB® function representing the equations of motion.

If the problem is deterministic, the function returns a vector of length d corresponding to the value of $g(\mathbf{x}(t), \mathbf{u}(t), t)$.

If the problem is stochastic then the function returns a vector of length $2d$, the first d elements of which are $g(\mathbf{x}(t), \mathbf{u}(t), t)$ and the second d elements of which are $b(\mathbf{x}(t), \mathbf{u}(t), t)$. If some of the variables are undisturbed by noise (i.e., $N < d$), then the variables for which the diffusion term is constantly 0 must follow those that are disturbed by noise.

In either case the function should have a header of the form

```
function Value = Delta(Control, StateVariables, Time)
```

where Control is a vector of length c, StateVariables is a vector of length d, and Time is a scalar.

InstantaneousCostFunctionFile**:**

A string giving the name (no .m extension) of a file containing a MATLAB® function representing the instantaneous cost function $f(\mathbf{x}(t), \mathbf{u}(t), t)$.[2] The function should have a header of the form

```
function Value = Cost(Control, StateVariables, Time)
```

[2] A maximisation problem can be converted into a minimisation problem by multiplying the performance criterion by -1. Consequently, if the SOC problem to be solved involves maximisation, the negative of its instantaneous cost should be specified in InstantaneousCostFunctionFile.

where `Control` is a vector of length c, `StateVariables` is a vector of length d, and `Time` is a scalar.

`TerminalStateFunctionFile`**:**

A string containing the name (no `.m` extension) of a file containing a MATLAB® function representing the terminal state function $h(\mathbf{x}(T))$. This function should return only the single real value given by $h(\mathbf{x}(T))$ (even if the function is identically zero).[3]

The function should have a header of the form

```
function Value = Term(StateVariables)
```

where `StateVariables` is a vector of length d.

`StateLB`**,** `StateUB`**, and** `StateStep`**:**

These determine the finite state grid for the Markov chain that we hope will approximate the SOC problem. Each may be given as a vector or as an array, independent of the type of the other two.

The value of `StateLB` is the least possible state, while the value of `StateUB` is the maximum possible state.[4]

The value of `StateStep` determines the distances between points of the state grid. It has to be chosen so that its entry/entries corresponding to the i-th state variable exactly divides/divide the difference/differences between the corresponding entries of `StateLB` and `StateUB`. Of course, step size need not be the same for all state variables.

Each of these values can be given in two different ways:

1. As a vector of length d (the dimension of the state space, where each value in the vector corresponds to one dimension) In this case, the same set of numbers will apply to every decision stage of the Markov chain.
2. As a matrix with $S+1$ rows and d columns, where S is the number of decision stages in the Markov chain as determined by `TimeStep` below. The i-th row is then used to determine the state grid in the i-th stage.

`TimeStep`**:**

This determines the number of decision stages and their associated times.

`TimeStep` is a vector of step lengths that partition the interval $[0, T]$; i.e., their sum should be T. The number of elements of `TimeStep` is the number of decision stages in the Markov chain, which we denote by S.

[3] A maximisation problem can be converted into a minimisation problem by multiplying the performance criterion by -1. Consequently, if the SOC problem to be solved involves maximisation, the negative of its terminal state should be specified in `TerminalStateFunctionFile`.

[4] The solution is routinely disturbed close to the state boundaries `StateLB` and `StateUB`. Consequently, these should be chosen "generously." Of course, larger state grids require greater computation times—see Section 7.1 for advice.

`ProblemFile`:

A string containing the name (with no extension) of the problem. This name is used to store the solution on disk. `SOCSol` produces at least two files with this name: one with the extension `.DPP`, which contains the parameters used to compute the Markov chain, and one with the extension `.DPS`, which contains the solution itself. If the `LagrangeMaps` option (described below) is set to 'yes', a third file with the extension `.DPL` is also produced. This contains the Lagrange multipliers.

The `.DPP` and `.DPS` files are used by `GenSim` to produce the continuous-time, continuous-state control rule. Note that the routines `ContRule`, `GenSim` and `ValGraph` (see Sections 2, 3 and 4 respectively) require that the `.DPP` and `.DPS` files exist and remain unchanged, while the routine `LagranMap` (see Section 5) requires that the `.DPP` and `.DPL` files exist and remain unchanged.

`Options`:

This vector (in fact, a cell array) of strings controls various internal settings that need only be adjusted infrequently. Two types of options can be set using the `Options` vector: options directly related to `SOCSol4L`, and options used by `fmincon`, a MATLAB® routine employed by `SOCSol4L`.

The user need only specify those options that are to be changed from their default values. If all options are to remain at their default values, then `Options` should be passed as empty, i.e., as { }.

In order to alter an option from its default value, the option should be named (in a string) followed directly by the value to which it is to be set (in another string). For example, if it was desired to set the `ControlDimension` option to 2 and turn on the `Display`, then `Options` could be set as

```
Options = {'ControlDimension' '2' 'Display' 'on'};
```

Note that the number 2 is entered as the string '2'. While it is important that an option be followed directly by the value to which it is to be set, option-value pairs can be given in any order. So it would be equally valid to set the above as

```
Options = {'Display' 'on' 'ControlDimension' '2'};
```

The options related directly to `SOCSol4L` are:

1. `ControlDimension`. This contains the value c for your problem. It must be given as a natural number (in a string). The default value is 1.
2. `StochasticProblem`. This should be set to 'yes' if your problem is stochastic. The default value is 'no', i.e., the problem is assumed to be deterministic.
3. `NoisyVars`. This should be set to the number N (in a string) if $N < d$. The default value is d, i.e., all variables are assumed to be noisy. If the problem is deterministic, `SOCSol` ignores the value of `NoisyVars`.
4. `LagrangeMaps`. This specifies whether `SOCSol` should produce an output file containing the Lagrange multipliers associated with the problem's constraints. If so,

LagrangeMaps should be set to 'yes'. The default value is 'no'. This option must be enabled if the user later wishes to utilise the LagranMap routine (see Section 5).

In general, fmincon can use either large-scale or medium-scale algorithms. While large-scale algorithms are more efficient for some problems, the use of such an algorithm requires differentiability of the function to be minimised. This is not generally true of the cost-to-go functions that SOCSol4L passes to fmincon. Consequently, SOCSol4L employs only fmincon's medium-scale algorithms.

As a result of this, those fmincon options specific to large-scale algorithms are not set through the Options vector, but instead passed their default values by SOCSol4L. However, the fmincon options specific to medium-scale algorithms may be set through the Options vector. These include:

1. Diagnostics. This controls whether fmincon prints diagnostic information about the cost-to-go functions that it minimises. The default value is 'off', but Diagnostics may also be set to 'on'.
2. Display. This controls fmincon's display level. The default value is 'off' (no display), but Display may also be set to 'iter' (display output for each of fmincon's iterations), 'final' (display final output for each call to fmincon) and 'notify' (display output only if non-convergence is encountered).
3. MaxFunEvals. This sets fmincon's maximum allowable number of function evaluations. The default value is $100c$, but MaxFunEvals may be set to any natural number (in a string).
4. MaxIter. This sets fmincon's maximum allowable number of iterations. The default value is 400, but MaxIter may be set to any natural number (in a string).
5. MaxSQPIter. This sets fmincon's maximum allowable number of sequential quadratic programming steps. The default value is ∞, but MaxSQPIter may be set to any natural number (in a string).
6. TolCon. This sets fmincon's termination tolerance on constraint violation. The default value is 10^{-6}, but TolCon may be set to any positive real number (in a string).
7. TolFun. This sets fmincon's termination tolerance on function evaluation. The default value is 10^{-6}, but TolFun may be set to any positive real number (in a string). See Section 7.3 for more on the use of TolFun.
8. TolX. This sets fmincon's termination tolerance on optimal control evaluation. The default value is 10^{-6}, but TolX may be set to any positive real number (in a string).

If necessary, it is also possible to set:

11. DerivativeCheck. This controls whether fmincon compares user-supplied analytic derivatives (e.g., gradients or Jacobians) to finite differencing derivatives. The default value is 'off', but DerivativeCheck may also be set to 'on'.
12. DiffMaxChange. This sets fmincon's maximum allowable change in variables for finite difference derivatives. The default value is 0.1, but DiffMaxChange may be set to any positive real number (in a string).
13. DiffMinChange. This sets fmincon's minimum allowable change in variables for finite difference derivatives. The default value is 10^{-8}, but DiffMaxChange may be set to any positive real number (in a string).

14. `OutputFcn`. A string containing the name (no `.m` extension) of a file containing a MATLAB® function that is to be called by `fmincon` at each of its iterations. Such a function is typically used to retrieve/display additional data from `fmincon`.
    See *Optimization Options :: Argument and Options Reference (Optimization Toolbox)* in MATLAB® help for more information on output functions.

For more information on `fmincon`, and `fmincon` options in particular, see *fmincon :: Functions (Optimization Toolbox)* and *Optimization Options :: Argument and Options Reference (Optimization Toolbox)* in MATLAB® help.

`InitialControlValue`**:**

In general, a vector of initial values for the control variables. This is used by `SOCSol4L` as an approximate starting point in its search for an optimal discrete-state, discrete-time decision rule. Consequently, it may be chosen with some inaccuracy.

`A` and `b`:

These allow for the imposition of the linear inequality constraint(s) $A\mathbf{u} \leqslant \mathbf{b}$ on the control variable(s). In general, `A` is a matrix and `b` is a vector. If there are no linear inequality constraints on the control variable(s), both `A` and `b` should be passed as empty: [ ].

`Aeq` and `beq`:

These allow for the imposition of the linear equality constraint(s) $Aeq \cdot \mathbf{u} = \mathbf{beq}$ on the control variable(s). In general, `Aeq` is a matrix and `beq` is a vector. If there are no linear equality constraints on the control variable(s), both `Aeq` and `beq` should be passed as empty: [ ].

`ControlLB` and `ControlUB`:

In general, vectors of lower and upper bounds (respectively) on the control variables. If a control variable has no lower bound, the corresponding entry of `ControlLB` should be set to $-\mathbf{Inf}$. Similarly, if a control variable has no upper bound, the corresponding entry of `ControlUB` should be set to **Inf**.

`UserConstraintFunctionFile`**:**

A string containing the name (no `.m` extension) of a file containing a MATLAB® function representing problem constraints (in particular, non-linear problem constraints).

This function should return the value of inequality constraints as a vector `Value1` and the value of equality constraints as a vector `Value2`, where inequality constraints are written in the form $k(\mathbf{u}, \mathbf{x}, t) \leqslant 0$ and equality constraints are written in the form $keq(\mathbf{u}, \mathbf{x}, t) = 0$.

The function should have a header of the form

```
function [Value1, Value2] = Constraint(Control,
    StateVariables, TimeStep)
```

where `Control` is a vector of length c, `StateVariables` is a vector of length d, and `TimeStep` is a scalar.

Note that the `TimeStep` argument makes the time step for the relevant Markov chain time readily available for incorporation in constraints. It should not be confused with the `Time` arguments of the other user-specified functions, which correspond to the Markov chain times themselves.

In the absence of constraints requiring the use of `UserConstraintFunctionFile`, `UserConstraintFunctionFile` should be passed as empty: [ ].

See *fmincon :: Functions (Optimization Toolbox)* in MATLAB® help for further information about solution starting points, `A`, `b`, `Aeq`, `beq`, bounds and the specification of non-linear problem constraints. Some hints on enforcing problem constraints are given in Sections 7.5, 7.6 and 7.7.

2. ContRule

2.1. **Purpose.** `ContRule` produces graphs of the continuous-time, continuous-state control rule derived from the solution computed by `SOCSol`.[5] Each control rule graph is produced for a given time and holds all but one state variable constant.

2.2. **Syntax.** `ContRule` is called as follows.

```
ContRule('ProblemFile', Time, InitialCondition,
    VariableOfInterest, LineSpec);

ControlValues = ContRule(···);
```

Calling `ContRule` without any output arguments produces control rule profiles for the given time and displays some technical information in the MATLAB® command window. However, `ContRule` may also be called with a single output argument. In this instance, `ContRule` also assigns the output argument the values of the control rules in the form of an $M \times c$ array, where

$$M = \frac{\texttt{StateUB}_{\texttt{VariableOfInterest}} - \texttt{StateLB}_{\texttt{VariableOfInterest}}}{\texttt{StateStep}_{\texttt{VariableOfInterest}}} + 1.$$

So the rows of this array correspond to points of the `VariableOfInterest`-th dimension of the state grid, while its columns correspond to control dimensions.

`ProblemFile`**:**

A string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by `SOCSol` from the disk. `ContRule` requires that the `.DPP` and `.DPS` files produced by `SOCSol` still exist and remain unchanged.

[5] Analogous graphs of the Lagrange multipliers associated with problem constraints can be obtained via `LagranMap`—see Section 5.

`Time`**:**

A scalar in the interval $[0, T]$ telling `ContRule` the time for which a control profile is to be computed. If `Time` is not a Markov chain time, `ContRule` computes a control profile for the last Markov chain time before `Time`.

`InitialCondition`**:**

A vector determining the values of the fixed state variables. A value must be given for the `VariableOfInterest` as a placeholder, although this value is not used.

`VariableOfInterest`**:**

A scalar telling the routine which of the state variables to vary, i.e., numbers like "1" or "2" etc. have to be entered in accordance with the state variables' ordering in the function `DeltaFunctionFile`. The control rule profile appears with the nominated state variable along the horizontal axis.

`LineSpec`**:**

This specifies the line style, marker symbol and colour of timepaths. It is a string of the format discussed in the *LineSpec :: Functions (MATLAB Function Reference)* section of MATLAB® help.

If `LineSpec` is not specified, it defaults to 'r-' (a solid red line without markers).

3. GenSim

3.1. **Purpose.** `GenSim` takes the solution computed by `SOCSol`, derives a continuous-time, continuous-state control rule and simulates the continuous system using this rule. It returns graphs of the timepaths of the state and control variables and the associated performance criterion values for one or more simulations.

3.2. **Implementation.** The derivation of the continuous-time, continuous-state control rule from the solution computed by `SOCSol` requires some form of interpolation in both state and time. In an effort to keep the script simple the interpolation in state is linear. States that are outside the state grid simply move to the nearest state grid point. For times between Markov chain times, the control profile for the most recent Markov chain time is used.

The differential equation that governs the evolution of the system is simulated by interpolation of its Euler-Maruyama approximation. The performance criterion integral is approximated using the left-hand endpoint rectangle rule.

3.3. **Syntax.** `GenSim` is called as follows.

```
SimulatedValue = GenSim('ProblemFile', InitialCondition,
    SimulationTimeStep, NumberOfSimulations, LineSpec,
    TimepathOfInterest, UserSuppliedNoise)
```

`ProblemFile`:

A string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by `SOCSol` from the disk. `GenSim` requires that the `.DPP` and `.DPS` files produced by `SOCSol` still exist and remain unchanged.

`InitialCondition`:

This is a vector of length d that contains the initial condition: the point from which the simulation starts.

`SimulationTimeStep`:

This is a vector of step lengths that partition interval $[0, T]$; i.e., their sum should be T.

With more simulation steps there is less error from approximating the equations of motion using the Euler-Maruyama scheme and from approximating the performance criterion using the left-hand endpoint rectangle method. In general, the simulation step is much smaller than the time step used to compute the solution (i.e., smaller than the `TimeStep` argument given to `SOCSol`).

If no `SimulationTimeStep` vector is given, the `SimulationTimeStep` vector defaults to the `TimeStep` vector used for computing the `ProblemFile`.

`NumberOfSimulations`:

This is the number of simulations that should be performed. If `NumberOfSimulations` is passed as negative, `GenSim` performs $|$`NumberOfSimulations`$|$ simulations, but does not plot any timepaths.

Multiple simulations are normally performed when dealing with a stochastic system. Each simulation uses a randomly determined noise realisation (unless this is suppressed by the `UserSuppliedNoise` argument).

If `NumberOfSimulations` is not specified, it defaults to 1.

`LineSpec`:

This specifies the line style, marker symbol and colour of timepaths. It is a string of the format discussed in the *LineSpec :: Functions (MATLAB Function Reference)* section of MATLAB® help.

If `LineSpec` is not specified, it defaults to 'r-' (a solid red line without markers).

`TimepathOfInterest`:

This is an integer between 0 and $d + c$ (inclusive) that specifies which timepath(s) `GenSim` is to plot. If `TimepathOfInterest` is passed the value 0, `GenSim` plots timepaths for all state and control variables. Otherwise, if `TimepathOfInterest` is passed the value $i > 0$, `GenSim` plots the timepath of the i-th variable, where state variables precede control variables.

If `TimepathOfInterest` is not specified, it defaults to 0.

`UserSuppliedNoise`**:**

This entirely optional argument enables the user to override the random generation of noise realisations. If `UserSuppliedNoise` is passed the value 0, a constantly 0 noise realisation is used. Otherwise, `UserSuppliedNoise` should be passed a matrix with N columns and a row for each entry of `SimulationTimeStep`.

Note that `NumberOfSimulations` should be 1 if `UserSuppliedNoise` is specified. If `UserSuppliedNoise` is left unspecified, `GenSim` randomly selects a standard Gaussian noise realisation for each simulation. Naturally, `UserSuppliedNoise` has no effect on deterministic problems.

`SimulatedValue`**:**

The MATLAB® output consists of a vector of the values of the performance criterion for each of the simulations performed.

If the problem is stochastic and noise realisations are random, then the average of the values from a large number of simulations can be used as an approximation to the expected value of the continuous stochastic system (under the continuous-time, continuous-state control rule derived from the solution computed by `SOCSol`). This average is left for the user to compute.

4. ValGraph

4.1. **Purpose.** `ValGraph` automates the process of computing expected values for the continuous system (under the continuous-time, continuous-state control rule derived from the solution computed by `SOCSol`) as the initial conditions change. In a similar spirit to `ContRule` (see Section 2), it deals with one state variable at a time (identified by `VariableOfInterest`), while the other state variables remain fixed.

4.2. **Syntax.** `ValGraph` is called as follows.

```
ValGraph('ProblemFile', InitialCondition, VariableOfInterest,
    VariableOfInterestValues, SimulationTimeStep,
    NumberOfSimulations, ScaleFactor, LineSpec)
```

`ProblemFile`**:**

A string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by `SOCSol` from the disk. `ValGraph` requires that the `.DPP` and `.DPS` files produced by `SOCSol` still exist and remain unchanged.

`InitialCondition`**:**

A vector determining the values of the fixed state variables. A value must be given for the `VariableOfInterest` as a placeholder, although this value is not used.

`VariableOfInterest`:

A scalar telling the routine which of the state variables to vary. The value graph appears with this state variable along the horizontal axis.

`VariableOfInterestValues`:

A vector containing the values of the `VariableOfInterest` at which the system's performance is to be evaluated.

`SimulationTimeStep`:

This is as for `GenSim` in Section 3.

`NumberOfSimulations`:

This is the number of simulations that should be performed. `NumberOfSimulations` behaves like the identically-named argument for `GenSim` in Section 3, except if passed the value 1 for a stochastic problem, it yields a constantly zero noise realisation.

If `NumberOfSimulations` is not specified, it defaults to 1.

`ScaleFactor`:

This simply scales all the resulting values by the given factor.

Maximisation problems must have all payoffs replaced by their negatives before entry into `SOCSol`, as it assumes that problems require minimisation. Setting `ScaleFactor` to -1 "corrects" the sign on payoffs for maximisation problems.

Naturally, if `ScaleFactor` is not specified, it defaults to 1.

`LineSpec`:

This specifies the line style, marker symbol and colour of timepaths. It is a string of the format discussed in the *LineSpec :: Functions (MATLAB Function Reference)* section of MATLAB® help.

If `LineSpec` is not specified, it defaults to 'r-' (a solid red line without markers).

5. LagranMap

5.1. **Purpose.** `LagranMap` utilises `SOCSol`'s output to produce graphs of the Lagrange multipliers associated with problem constraints.[6] Each Lagrange map graph is produced for a given time and holds all but one state variable constant.

5.2. **Syntax.** `LagranMap` is called as follows.

```
LagranMap('ProblemFile', Time, InitialCondition,
    VariableOfInterest, LineSpec, MultiplierOfInterest);
```

[6] Analogous graphs of the problem's controls can be obtained via `ContRule`—see Section 2.

```
LagrangeValues = LagranMap(···);
```

Calling LagranMap without any output arguments produces Lagrange multiplier profiles for the given time and displays some technical information in the MATLAB® command window. However, LagranMap may also be called with a single output argument. In this instance, LagranMap also assigns the output argument the values of the Lagrange multipliers in the form of an $M \times c$ array, where

$$M = \frac{\texttt{StateUB}_{\texttt{VariableOfInterest}} - \texttt{StateLB}_{\texttt{VariableOfInterest}}}{\texttt{StateStep}_{\texttt{VariableOfInterest}}} + 1.$$

So the rows of this array correspond to points of the VariableOfInterest-th dimension of the state grid, while its columns correspond to control dimensions.

ProblemFile:

A string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by SOCSol from the disk. LagranMap requires that this solution include a .DPL file,[7] and that the .DPP and .DPL files produced by SOCSol still exist and remain unchanged.

Time:

A scalar in the interval $[0, T]$ telling LagranMap the time for which Lagrange map profile is to be computed. If Time is not a Markov chain time, LagranMap computes a Lagrange map profile for the last Markov chain time before Time.

InitialCondition:

A vector determining the values of the fixed state variables. A value must be given for the VariableOfInterest as a placeholder, although this value is not used.

VariableOfInterest:

A scalar telling the routine which of the state variables to vary, i.e., numbers like "1" or "2" etc. have to be entered in accordance with the state variables' ordering in the function DeltaFunctionFile. The Lagrange map profile appears with the nominated state variable along the horizontal axis.

LineSpec:

This specifies the line style, marker symbol and colour of timepaths. It is a string of the format discussed in the *LineSpec :: Functions (MATLAB Function Reference)* section of MATLAB® help.

If LineSpec is not specified, it defaults to 'r-' (a solid red line without markers).

MultplierOfInterest:

This is an integer between 0 and the number of bounds & constraints (inclusive) that specifies which Lagrange map(s) LagranMap is to plot. If MultplierOfInterest is

[7] See Options in Section 1.2.

passed the value 0, LagranMap plots Lagrange maps for all bounds and constraints. Otherwise, if MultplierOfInterest is passed the value $i > 0$, LagranMap plots the Lagrange map of the i-th bound/constraint.

If MultplierOfInterest is not specified, it defaults to 0.

6. Technical Information

6.1. **Encoding the State Space.** Each point of the state grid is assigned a unique number in the SOCSol routine. This state number is then used as an index in all the resultant matrices.

A CodingVector is computed for each stage. As a precursor to this, the number of states along each of the d state variable axes is first computed by

```
States = round((Max - Min)./StateStep + 1);
```

where Max, Min, and StateStep are vectors appropriate for the current stage. Taking the product of the entries of States gives TotalStates for the current stage. The CodingVector is then defined by

```
c = cumprod(States);
CodingVector = [1, c(1:Dimension - 1)];
```

where Dimension is the internal name for d, the number of state variables. The CodingVector is used to compute the state number for each point of the state grid for the current stage.

Suppose that x is a point of the state grid. Then x is converted into a vector of numbers called StateVector (which has $1 \leq$ StateVector(i) $\leq$ States(i) for each $i = 1, \ldots, d$) by

```
StateVector = round((x - Min)./StateStep + 1);
```

Finally, the state number for x, called State, is computed by taking the dot product of StateVector - 1 with the CodingVector and then adding 1, i.e.,

```
State = (StateVector - 1)*CodingVector + 1;
```

This process assigns a unique State to each point of the state grid. Every number between 1 and TotalStates (inclusive) corresponds to a point of the state grid. By construction, 1 corresponds to Min and TotalStates to Max.

Converting a state number back into a point of the state grid is equally simple. The first step is to convert the State into a StateVector. This is carried out by a function SnToSVec:

```
function StateVector = SnToSVec(StateNumber, CodingVector,
    Dimension)

StateNumber = StateNumber - 1;
StateVector = zeros(1, Dimension);
```

```
for i = Dimension:-1:1
    StateVector(i) = floor(StateNumber/CodingVector(i));
    StateNumber = StateNumber
                  - StateVector(i)*CodingVector(i);
end;

StateVector = StateVector + 1;
```

Converting a `StateVector` to its associated `x` is as simple as executing:

```
x = (StateVector - 1).*StateStep + Min;
```

6.2. **The Solution Files.** `SOCSol` writes at least two files to disk. The file with the `.DPP` extension contains the parameters used to compute the approximating Markov chain. The file with the `.DPS` extension contains the solution matrices. These are indexed using the state number encoding of the state space (see Section 6.1). Additionally, if `SOCSol` is called with the `LagrangeMaps` option set to 'yes', a file with the extension `.DPL` is produced. This contains the Lagrange multiplier matrices.

The `.DPP` File:

The first three lines of this file hold (as strings) the variables `DeltaFunctionFile`, `InstantaneousCostFunctionFile`, and `TerminalStateFunctionFile`. These variables are retrieved using `fscanf` with a call to `fgets` after the third `fscanf` to remove the remaining new line.

The remaining portion of the file holds the matrices `StateLB`, `StateUB`, `StateStepSize` and `TimeStep`, together with the values specified for `Options` by the user. Additional information may be appended at the end of the file (currently `SOCSol` appends the computation time taken to compute the solution (in seconds) and the number of times that `fmincon` was called). All these variables (with the exception of the additional information) are stored and retrieved using the `MatWrite` and `MatRead` functions. These simple functions store the matrices in a binary form that is more time and space efficient than a text representation.

The `.DPS` File:

This file contains only matrices, written and retrieved by the `MatWrite` and `MatRead` functions. The file begins with the matrices representing the optimal control. There is one such matrix for each dimension of the control space. These can be retrieved using a call of the form

```
for i = 1:ControlDimension
    eval(['ODM', int2str(i), '=MatRead(fid);'])
fclose(fid);
```

These matrices are followed by a single matrix holding the cost-to-go for each state of each stage of the Markov chain.

The `.DPL` File:

This file also contains only matrices, written and retrieved by the `MatWrite` and `MatRead` functions. The file begins with six matrices (in fact, scalars) that respectively give the numbers of lower bounds, upper bounds, linear inequalities, linear equalities, non-linear inequalities and non-linear equalities that constrain the controls. These are then followed by matrices representing each of these constraints, grouped in the order above. The matrices can be retrieved using a call of the form:

```
NumLower = MatRead(fid);
NumUpper = MatRead(fid);
NumIneqLin = MatRead(fid);
NumEqLin = MatRead(fid);
NumIneqNonlin = MatRead(fid);
NumEqNonlin = MatRead(fid);
for i = 1:NumLower
    eval(['LagrangeLower', int2str(i), '=MatRead(fid);']);
end;
for␣j␣=␣1:NumUpper
␣␣␣␣eval(['LagrangeUpper', int2str(j), '=MatRead(fid);']);
end;
⋮
```

7. Hints

7.1. **Choosing State Variable Bounds.** The formulation of a SOC problem may give rise to a "natural" choice of bounds on its state variable(s), e.g., $x_1 \in [0, 0.5]$. However, `SOCSol4L`'s solutions are routinely disturbed close to the bounds specified. Consequently, it may be sensible to specify a larger state grid than might initially appear necessary (e.g., $x_1 \in [-0.2, 0.7]$). Figure 1 demonstrates how this can move distortion outside the area of interest.

Of course, *ceteris paribus*, specifying a larger state grid increases computation time. If the SOC problem has many dimensions, this increase may be considerable.

7.2. **Choosing Discretisation Step Sizes.** The choice of state step(s) and the choice of time step are not entirely independent. In particular, the time step should not be "too small" in comparison to the state step(s). This is because some control $\hat{\mathbf{u}}$ is optimal for the Markov chain at a given grid point. However, the control chosen at this point is a multiple of the time step. So if the time step is "too small," $\hat{\mathbf{u}}$ cannot be realised, making the optimal solution unobtainable.[8]

Empirical evidence suggests that the time step can also be "too large" in comparison to the state step(s). It is thus desirable to try a variety of discretisations, which may then indicate some choices of step sizes that are effective for a particular problem. This is easily done by extending the script suggested in Section 1.2 to call `SOCSol` several times, specifying `ProblemFile` differently each time.

[8] See p. 16 of [Kra01] for further discussion.

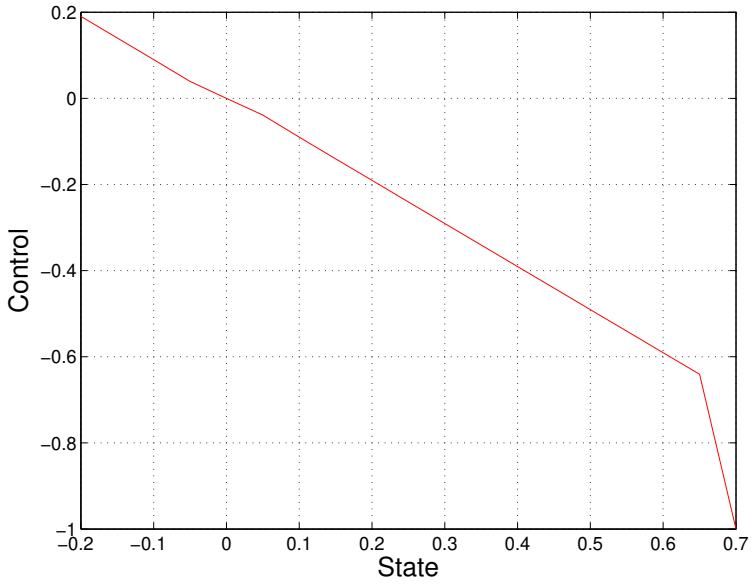

FIGURE 1. Control rules with distortion at the right.

7.3. **Use of `TolFun`.** When fine discretisation is used (in particular, a small time step), the cost-to-go functions that `fmincon` must minimise are relatively flat. Consequently `fmincon` may make significant errors when computing controls if `TolFun` is left at its default level. These errors often manifest as large "steps" in control rules and/or timepaths.

As a result of this, `TolFun` should be set to an especially small value (e.g., 10^{-12}) when fine discretisation is used.[9]

7.4. **Relative Magnitudes of Controls.** Most numerical optimisation methods work most effectively when the solution components (here, the entries of $\mathbf{u}$) are of comparable size. However, not all problems naturally accommodate this. For example, it may be that $u_1 \in [0, 1]$ while $u_2 \geqslant 0$ realises large values (e.g., 10^4). In certain cases, it is possible to remove such non-comparability by rescaling one or more controls. For example, if $u_2 \propto x$ and x realises values comparable to u_1, then replacement of u_2 by a suitably chosen u_2' results in both u_1 and u_2' realising comparable values.[10]

7.5. **Enforcing Bounds on Directly Controlled State Variables.** Call a state variable whose derivative explicitly depends on at least one control variable *directly controlled*. A method of enforcing bounds on directly controlled variables is sketched below.

Suppose that a state variable x_i has

$$\mathrm{d}x_i = g_i(\mathbf{x}, \mathbf{u})\mathrm{d}t$$

[9] See p. 15 of [Kra01] for further discussion.

[10] See p. 13–14 of [Kra01] for further discussion.

where at least one entry of $\mathbf{u}$ appears in g_i (i.e., x_i is directly controlled). If the `TimeStep` is constant and denoted by δ and t is a Markov chain time, then

$$x_i(t+\delta) = x_i(t) + \delta g_i(\mathbf{x}, \mathbf{u}).$$

So specifying the `UserConstraintFunctionFile` as

```
function [v1 v2] = ConFun(u, x, ts)

v1 = [a - x(i) - ts*g_i(x, u)];
v2 = [];
```

enforces the lower bound $a - x_i \leqslant 0$.

7.6. **Enforcing Bounds on Monotonic State Variables.** Suppose that an SOC problem is formulated in such a way that it can be shown analytically that in its optimal solution, some state variable x_i must be monotonic with respect to time. Then if x_i is decreasing, enforcing the upper bound $x_i \leqslant a$ amounts to changing `GenSim`'s initial condition to $x_i(0) = a$ if it was previously $x_i(0) = a + p$ for some positive p.

Naturally, an analogous method applies if x_i is increasing. See Section 7.7 below for another way of enforcing bounds on monotonic state variables.

7.7. **Enforcing Terminal State Constraints.** Suppose that a state variable x_i is subject to the terminal constraint $x_i(T) \leqslant a(\mathbf{x}_{-i}(T), \mathbf{u}(T))$, where

$$\mathbf{x}_{-i} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d).$$

Such a terminal constraint can be enforced by adding a summand of the form

$$k\big(\max\{0, x_i - a(\mathbf{x}_{-i}, \mathbf{u})\}\big)^2$$

to `TerminalStateFunction`, where $k > 0$ is a constant. It may be necessary to run `SOCSol` several times to find a value of k sufficiently large to enforce the constraint.

This method can be used to enforce lower bounds on decreasing state variables and upper bounds on increasing state variables.

7.8. **Retrieving Additional Information.** If additional information is required from one of the `SOCSol4L` routines, it is often possible to retrieve this by declaring a global variable. For example, suppose that it were necessary to obtain the state values used by `GenSim` in plotting a particular state timepath. Identifying the appropriate internal variable as `GenSim`'s `StateEvolution`, one could procede as follows:

1. Execute the statement `global temp` in the MATLAB® command window.
2. Append `GenSim` with the lines

   ```
   global temp
   temp=StateEvolution;
   ```

   and save.
3. Call `GenSim` in the MATLAB® command window.

The desired state values would then be available in the variable `temp`. It is recommended that the name chosen for this variable (here, "temp") not coincide with the name of any `SOCSol4L` internal variable.

Appendix A. Example of use of SOCSol4L

A.1. **Optimisation Problem.** We will show how to obtain the results reported in Example 2.3.1 of [Kra01].

The optimisation problem is determined in $\mathbb{R}^1$ by

$$\min_{u(\cdot)} J = \frac{1}{2}\int_0^1 \Big(u(t)^2 + x(t)^2\Big)\,\mathrm{d}t + \frac{1}{2}x^2(1), \tag{3}$$

subject to

$$\dot{x} = u \text{ and} \tag{4}$$

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

A Markovian approximation to this problem is to be formed and then solved. The optimisation call for the routine that does this is SOCSol$(\cdots)$, where the arguments inside the brackets are the same as those of Section 1.2. Details of the specification of these arguments for this example are given below.

The following functions are defined by the user and saved in MATLAB® as `.m` files in locations on the path. Each file has a name (for example `Delta.m`), and consists of a header, followed by one (or more) commands. For this example we have:

DeltaFunctionFile:

This is called, for example, `Delta.m` and is written as follows.

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

InstantaneousCostFunctionFile:

This is called `Cost.m` and is written as follows.

```
function v = Cost(u, x, t)
v = (u^2 + x^2)/2;
```

TerminalStateFunctionFile:

This is called `Term.m` and is written as follows.

```
function v = Term(x)
v = x^2/2;
```

The parameters used in `SOCSol` are described in Section 1.2. In this example they are specified as follows.

`StateLB` and `StateUB`: For this one-dimensional Linear-Quadratic problem we give 0 and 0.5 respectively. This is because it is anticipated that the state value will diminish monotonically from 0.5 to a small positive value. Consequently values smaller or larger than these are unnecessary.

`StateStep`: This is given a value of 0.05, making the discrete state space the set $\{0, 0.05, 0.1, \ldots, 0.45, 0.5\}$.

`TimeStep`: Given by the vector ones(1, 10)/10, this divides the interval $[0,1]$ into 10 equal subintervals.

`ProblemFile`: This is the name for the files that the results are to be stored in, say TestProblem.

`Options`: In this example it is not necessary to include this vector: the default of { } suffices.

`InitialControlValue`: This is given a value of 0.5. As this value is only an approximate starting point for the routine, it may be specified with some inaccuracy.

`A`, `b`, `Aeq` and `beq`: As there are no linear constraints, these are all passed as empty: [ ].

`ControlLB` and `ControlUB`: As the control variable is unbounded, these are passed as $-$**Inf** and **Inf** respectively.

`UserConstraintFunctionFile`: As there are no constraints requiring the use of this argument, it too is passed as empty: [ ].

A.2. **Solution Syntax.** Consequently `SOCSol` could be called in MATLAB® as follows.

```
SOCSol('Delta', 'Cost', 'Term', 0, 0.5, 0.05, ones(1, 10)/10,
    'TestProblem', { }, 0.5, [ ], [ ], [ ], [ ], -Inf, Inf,
    [ ]);
```

TestProblem is just the header part (without the `.DPS` and `.DPP` extensions) of the two results files saved from `SOCSol` and stored for later use (see Section 6.2 for details).

While the call above is clear for such a simple problem, it is preferable to write MATLAB® scripts for more involved problems. For this example, a script could be written as follows.

```
StateLB = 0;
StateUB = 0.5;
StateStep = 0.05;
TimeStep = ones(1, 10)/10;
Options = { };
InitialControlValue = 0.5;
A = [];
b = [];
Aeq = [];
beq = [];
ControlLB = -Inf;
ControlUB = Inf;
SOCSol('Delta', 'Cost', 'Term', StateLB, StateUB, StateStep,
    TimeStep, 'TestProblem', Options, InitialControlValue, A,
    b, Aeq, beq, ControlLB, ControlUB, [ ]);
```

If this script were called `work_space.m` and placed in a directory visible to the MATLAB® path, it would then only be necessary to call work_space in MATLAB®.

In each case, the `.m` extensions are excluded from the filenames.

A.3. **Retrieving Results Syntax.** The results are communicated by means of three types of figures: control-versus-state policy rules, state-and-control timepaths, and value graphs.

A.3.1. *Control vs. State.* The routine `ContRule` is used to obtain a graph of the control rule at time 0 from `SOCSol`'s solution. The following values are specified for the parameters described in Section 2.2.

`ProblemFile`: As above, this is 'TestProblem'.

`Time`: This is given a value of 0, as this is the time for which a control rule is to be computed.

`InitialCondition`: As there is no need to hold a varying variable fixed, this condition does not matter in a one-dimensional example, where we only have one state variable (i.e., `IndependentVariable`) to vary. Consequently, this is set arbitrarily to 0.5.

`IndependentVariable`: This is set to 1, as there is only one state variable to vary. If this example had more than one state dimension, `IndependentVariable` could be any number beween 1 and d (inclusive), depending on which dimension/variable was to be varied.

`LineSpec`: This is left unspecified, assuming its default of 'r-'.

Consequently `ContRule` is called as follows.

```
ContRule('TestProblem', 0, 0.5, 1)
```

This produces the graph shown in Figure 2.

Note that in this graph the optimal solution is presented as a dashed line, while our computed trajectories are presented as solid lines. This convention is followed for subsequent graphs.

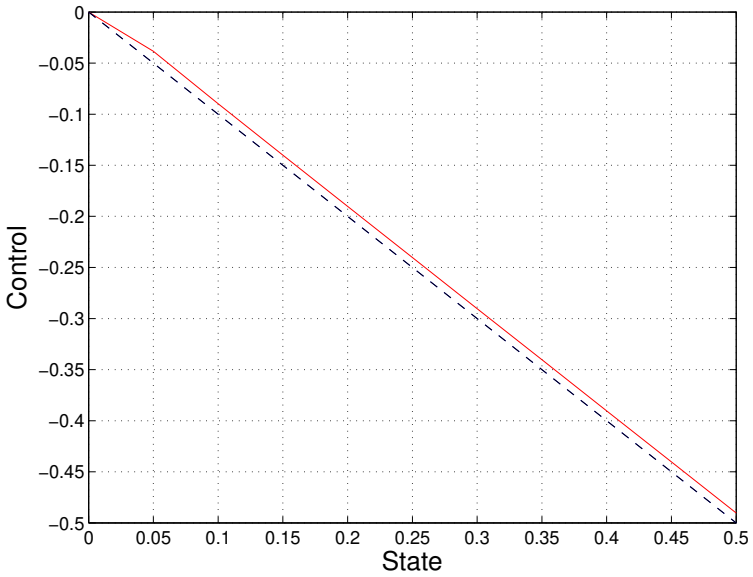

FIGURE 2. Control rules at time $t = 0$.

A.3.2. *State and Control vs. Time.* The files `TestProblem.DPP` and `TestProblem.DPS` are used to derive a continuous-time, continuous-state control rule. The system is then simulated using this rule. We use the routine `GenSim` with the following values for the parameters and functions described in Section 3.

`ProblemFile`: This is 'TestProblem' as before.

`InitialCondition`: As the simulation starts at $x_0 = 0.5$, this is specified as 0.5.

`SimulationTimeStep`: For 100 equidistant time steps this is given as ones(1, 100)/100.

`NumberOfSimulations`, `LineSpec` and `UserSuppliedNoise`: These are not passed, as the default values suffice.

Consequently `GenSim` is called as follows.

```
SimulatedValue = GenSim('TestProblem', 0.5, ones(1, 100)/100)
```

`SimulatedValue` gives the simulated value of the performance criterion. This is 0.1252 (4 s.f.) with the choice of parameters given here.

`GenSim` also produces the graph of timepaths shown in Figure 3.

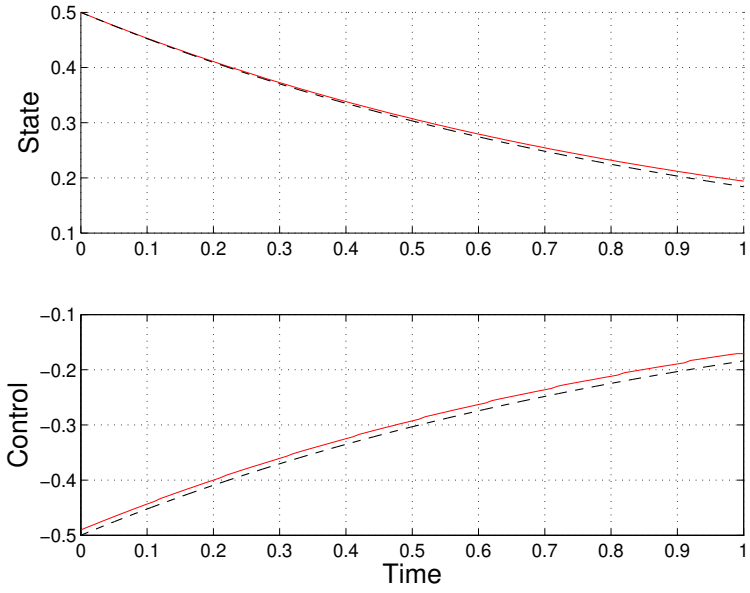

FIGURE 3. Optimal and approximated trajectories.

A.3.3. *Value vs. State.* Finally, the routine `ValGraph` computes the expected value of the performance criterion for the continuous system as the initial conditions vary. This routine has the following values for the parameters described in Section 4.2.

`ProblemFile`, `InitialCondition` and `VariableOfInterest`: These are given the same values as the corresponding arguments in Section A.3.2 above.

`VariableOfInterestValues`: This vector determines for what values of the variable of interest the performance criterion should be calculated. In this example, [0:0.05:0.5] is used.

`SimulationTimeStep`: For 100 equidistant time steps this is given as ones(1, 100)/100.

`NumberOfSimulations` and `ScaleFactor`: These are not passed, as the default values suffice.

Hence `ValGraph` is called as follows.

```
ValGraph('TestProblem', 0.5, 1, [0:0.05:0.5],
    ones(1, 100)/100)
```

This produces the graph shown in Figure 4.

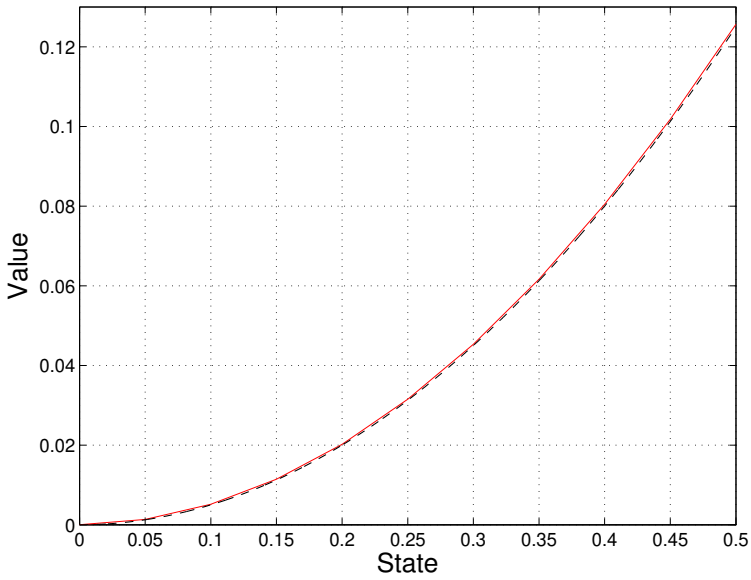

FIGURE 4. Optimal value function at time $t = 0$.

REFERENCES

[AK06] Jeffrey D. Azzato and Jacek B. Krawczyk. SOCSol4: A 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, Aug 2006.

[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.

[Kra05] J. B. Krawczyk. Numerical solutions to lump-sum pension problems that can yield left-skewed fund return distributions. In Christophe Deissenburg and Richard F. Hartl, editors, *Optimal Control and Dynamic Games*, number 7 in Advances in Computational Management Science, chapter 10, pages 155–176. Springer, New York, 2005.

[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 19/04/2008.

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