InfSOCSol2 An updated MATLAB Package for Approximating the Solution to a Continuous-Time Infinite Horizon Stochastic Optimal Control Problem with Control and State Constraints

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

31. August 2009

InfSOCSol2

AN UPDATED MATLAB® PACKAGE FOR APPROXIMATING THE SOLUTION TO A CONTINUOUS-TIME INFINITE HORIZON STOCHASTIC OPTIMAL CONTROL PROBLEM WITH CONTROL AND STATE CONSTRAINTS

JEFFREY AZZATO* & JACEK B. KRAWCZYK

ABSTRACT. This paper is a successor of [AK08]. Both papers describe the same suite of MATLAB® routines devised to provide an approximately optimal solution to an infinite-horizon stochastic optimal control problem. The difference is that this paper explains how to allow for state and control constraints. The suite routines implement a policy improvement algorithm to optimise a Markov decision chain approximating the original control problem, as described in [Kra01c] and [Kra01b].

2009 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, Financial engineering, Approximating Markov decision chains.

This report documents 2008 – 2009 research into Computational Economics Methods directed by Jacek B. Krawczyk. Help by Research Assistant Feliks Krawczyk is thankfully acknowledged.

Correspondence should be addressed to:

Jacek B. Krawczyk. Faculty of Commerce and Administration, Victoria University of Wellington, PO Box 600, Wellington, New Zealand. Fax: +64-4-4635014 Email: Jacek.Krawczyk at vuw.ac.nz ; webpage: http://www.vuw.ac.nz/staff/jacek_krawczyk

*Supported by the Victoria University Research Fund, Grant Number 32665, 2008 – 2009.

Contents

Introduction

Computing the solution to a stochastic optimal control problem is difficult. A method of approximating a solution to a given infinite horizon stochastic optimal control (SOC) problem using Markov chains was outlined in [Kra01c]. Along with [AK08] this paper describes a suite of MATLAB®[1] routines implementing this method of approximating a solution to a continuous infinite horizon SOC problem. The difference between this paper and [AK08] is that here, the computation example in Appendix A allows for control and state constraints.

The suite of routines developed updates and extends that described in [Kra01b]. The presentation given here is based on that given in [AK06], which describes a similar suite of MATLAB® routines that may be used to solve finnite horizon SOC problems.

The method used here deals with discounted infinite horizon stochastic optimal control problems having the form

$$\min_{\mathbf{u}} J(\mathbf{u}, \mathbf{x}_0) = \mathbb{E}\left[\int_0^\infty e^{-\rho t} f\big(\mathbf{x}(t), \mathbf{u}(t)\big) dt \,\middle|\, \mathbf{x}(0) = \mathbf{x}_0 \right] \tag{1}$$

subject to

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

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, 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 similar to those introduced in [AK06]. The package is called `InfSOCSol2` and is composed of four main modules:

1. `InfSOCSol`
2. `InfContRule`
3. `InfSim`
4. `InfValGraph`

`InfSOCSol` discretises a given SOC problem and then solves this "discretisation." `InfContRule` derives graphs of continuous-time, continuous-state control rules from the `InfSOCSol` solution. `InfSim` simulates the continuous system using such a control rule (also derived from the `InfSOCSol` solution). `InfValGraph` provides an automated means of computing expected values for the continuous system as initial conditions change.

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

1. InfSOCSol

1.1. **Purpose.** `InfSOCSol` takes the given SOC problem and approximates it with a Markov decision chain, which it then solves. This results in a discrete-time, discrete-space control rule. `InfSOCSol` 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 `InfSim`).

1.2. **Syntax.** `InfSOCSol` is called as follows.

```
InfSOCSol('DeltaFunctionFile',
    'InstantaneousCostFunctionFile', StateLB, StateUB,
    StateStep, TimeStep, DiscountRate, 'ProblemFile',
    Options, 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 4.2 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 $g\big(\mathbf{x}(t), \mathbf{u}(t)\big)$.

If the problem is stochastic then the function returns a vector of $2d$, the first d elements of which are $g\big(\mathbf{x}(t), \mathbf{u}(t)\big)$ and the second d elements of which are $b\big(\mathbf{x}(t), \mathbf{u}(t)\big)$. 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. The argument `Time` is not used by `InfSOCSol`, but its inclusion allows use of the `DeltaFunctionFile` in a finite horizon context by the `SOCSol` routine of the `SOCSol4L` package (see [AK06]).

`InstantaneousCostFunctionFile`**:**

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

The function should have a header of the form

[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`.

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

where `Control` is a vector of length c, `StateVariables` is a vector of length d, and `Time` is a scalar. As for the `DeltaFunctionFile`, the argument `Time` is not used by `InfSOCSol`, but its inclusion allows use of the `InstantaneousCostFunctionFile` in a finite horizon context by the `SOCSol` routine of the `SOCSol4L` package (see [AK06]).

As `InfSOCSol` only requires the discount rate ρ for its *value determination* step, there is no need to specify the discount factor $e^{-\rho t}$ in the `InstantaneousCostFunctionFile`.

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

These determine the finite state grid for the Markov chain that we hope will approximate the SOC problem.

The value of `StateLB` is the least possible state, while the value of `StateUB` is the maximum possible state.[3]

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

`TimeStep`:

The **scalar** δ to be used when formulating the Markov decision chain approximating the SOC problem (see [Kra01c]). As small values of δ are frequently advantageous and computation time depends predominantly on the size of the `StateStep`, a relatively small choice `TimeStep` is advisable.

See [AK06] for further information on choosing the `StateStep` and `TimeStep`.

`DiscountRate`:

The scalar ρ.

`ProblemFile`:

A string giving the name (with no extension) of the problem. This name is used to store the solution on disk. `InfSOCSol` produces two files with this name: one with the extension `.DPP`, which contains the parameters used to compute the Markov decision chain, and one with the extension `.DPS`, which contains the solution itself.

These files are used by `InfSim` to produce the continuous-time, continuous-state control rule. Note that the routines `InfContRule`, `InfSim` and `InfValGraph` (all explained below) require that the `.DPP` and `.DPS` files exist and remain unchanged.

[3] The solution can be disturbed close to the state boundaries `StateLB` and `StateUB`. Consequently, these should be chosen "generously." Of course, larger state grids require greater computation times.

`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 `InfSOCSol`, and options used by `fmincon`, a MATLAB® routine employed by `InfSOCSol`.

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 `InfSOCSol` are:

1. `ControlDimension`. This specifies 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, `InfSOCSol` ignores the value of `NoisyVars`.
4. `PolicyIterations`. This specifies the maximum number of *value iterations* that may be performed in seeking a solution (see [Kra01c]). The default value is 25.

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 `InfSOCSol` passes to `fmincon`. Consequently, `InfSOCSol` 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 `InfSOCSol`. However, the `fmincon` options specific to medium-scale algorithms may be set using 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).
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 Toolbox: Function Reference: Output Function* in MATLAB® help for more information on output functions.

For more information on `fmincon`, and `fmincon` options in particular, see *Optimization Toolbox: fmincon* and *Optimization Toolbox: Function Reference: Optimization Parameters* in MATLAB® help.

`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 $k(\mathbf{u}, \mathbf{x}) \leqslant 0$ and equality constraints are written in the form $keq(\mathbf{u}, \mathbf{x}) = 0$.

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 are not used by `InfSOCSol`.

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

See *Optimization Toolbox: fmincon* in MATLAB® help for further information about `A`, `b`, `Aeq`, `beq`, bounds and the specification of non-linear problem constraints.

2. InfContRule

2.1. **Purpose.** `InfContRule` produces graphs of the continuous-time, continuous-state control rule derived from the solution computed by `InfSOCSol`. Each control rule graph holds all but one state variable constant.

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

```
InfContRule('ProblemFile', InitialCondition,
```

```
    VariableOfInterest, LineSpec);

ControlValues = InfContRule(···);
```

Calling `InfContRule` without any output arguments produces control rule profiles and displays some technical information in the MATLAB® command window. However, `InfContRule` may also be called with a single output argument. In this instance, `InfContRule` 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 $\texttt{VariableOfInterest}^{th}$ 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 `InfSOCSol` from the disk. `InfContRule` requires that the files produced by `InfSOCSol` 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, i.e., numbers like "1" or "2" etc. have to be entered in accordance with the state variables' order 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 *MATLAB® Functions: LineSpec* section of MATLAB® help.

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

3. InfSim

3.1. **Purpose.** `InfSim` derives a continuous-time, continuous-state control rule from the solution computed by `InfSOCSol` and then 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 `InfSOCSol` requires some form of interpolation in both state and time. In the effort to keep the script simple the interpolation in state

is linear. States which 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 which 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.** `InfSim` is called as follows.

```
SimulatedValue = InfSim('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 `InfSOCSol` from the disk. `InfSim` requires that the files produced by `InfSOCSol` 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 in approximating the equations of motion using the Euler-Maruyama scheme and in approximating the performance criterion using the left-hand endpoint rectangle method.

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, `InfSim` 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 *MATLAB® Functions: LineSpec* 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) `InfSim` is to plot. If `TimepathOfInterest` is passed the value 0, `InfSim` plots timepaths for all state and control variables. Otherwise, if `TimepathOfInterest` is passed the value $i > 0$, `InfSim` 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 zero 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, `InfSim` 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 `InfSOCSol`). This average is left for the user to compute.

4. InfValGraph

4.1. **Purpose.** `InfValGraph` 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 `InfSOCSol`) as the initial conditions change. In a similar spirit to `InfContRule` above, it deals with one state variable at a time (indentified by `VariableOfInterest`), while the other state variables remain fixed.

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

```
InfValGraph('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 `InfSOCSol` from the disk. `InfValGraph` requires that the files produced by `InfSOCSol` 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.

`IndependentVariable`:

This scalar tells the routine which of the state variables to vary. The value graphs appear with this state variable along the horizontal axis.

`VariableOfInterest`:

A scalar telling the routine which of the state variables to vary. The value graph appears with this state variable along its 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 `InfSim` above.

`NumberOfSimulations`:

This is the number of simulations that should be performed. `NumberOfSimulations` behaves like the similarly-named argument in `InfSim` above, 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 `InfSOCSol`, as it assumes that problems require minimisation. Setting the `ScaleFactor` to -1 "corrects" the sign on payoffs for maximisation problems.

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

Appendix A. Example of use of InfSOCSol2

This example corresponds to that given in the appendix of [AK06] (and to *Example 2.3.1* of [Kra01a]), but with discounting and an infinite horizon.

A.1. **Optimisation Problem — Without Constraints.** The optimisation problem is determined in $\mathbb{R}^1$ by

$$\min_u J(u, x_0) := \frac{1}{2}\int_0^\infty e^{-\rho t}\big(u(t)^2 + x(t)^2\big)dt \tag{3}$$

subject to

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

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

Later, in Appendix A.4, we will extend this example to allow for constraints.

As this problem is formulated with an infinite horizon, discounting is necessary. Here we take the discount rate to be $\rho := \frac{9}{10}$, giving a discount factor of $e^{-\rho} = 0.4066$ (4 s.f.).

A Markovian approximation to this problem is to be formed and then solved. The optimisation call for the routine that does this is InfSOCSol$(\cdots)$, 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;
```

Similarly, for the other functions in this routine:

InstantaneousCostFunctionFile:

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

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

The parameters used in `InfSOCSol` 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 smaller or larger values than these are unnecessary.

`StateStep`: This is given a value of 0.01, making the discrete state space the set $\{0, 0.01, 0.02, \ldots, 0.49, 0.5\}$.

`TimeStep`: Given by the **scalar** 0.02.

`DiscountRate`: This is given a value of 0.9.

`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.

`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 `InfSOCSol` could be called in MATLAB® as follows:

```
InfSOCSol('Delta', 'Cost', 0, 0.5, 0.01, 0.02, 0.9,
    'TestProblem', { }, [ ], [ ], [ ], [ ], -Inf, Inf, [ ]);
```

TestProblem is just the header part (without the `.DPS` and `.DPP` extensions) of the two results files saved by `InfSOCSol` and stored for later use (see Section 3, for example).

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 = 0.01;
TimeStep = 0.02;
DiscountRate = 0.9;
Options = { };
A = [ ];
b = [ ];
Aeq = [ ];
beq = [ ];
ControlLB = -Inf;
ControlUB = Inf;
Constraint = [ ];
InfSOCSol('Delta', 'Cost', StateLB, StateUB, StateStep,
    TimeStep, DiscountRate, 'TestProblem', Options, A, b,
    Aeq, beq, ControlLB, ControlUB, Constraint);
```

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 MATLAB® routine `InfContRule` is used to obtain a graph of the control rule from `InfSOCSol`'s solution. The following values are specified for the parameters described in Section 2.2.

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

`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 (namely `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 natural 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 `InfContRule` is called as follows.

```
InfContRule('TestProblem', 0.5, 1)
```

This produces the red line in the graph of control rules shown in Figure 1 below.

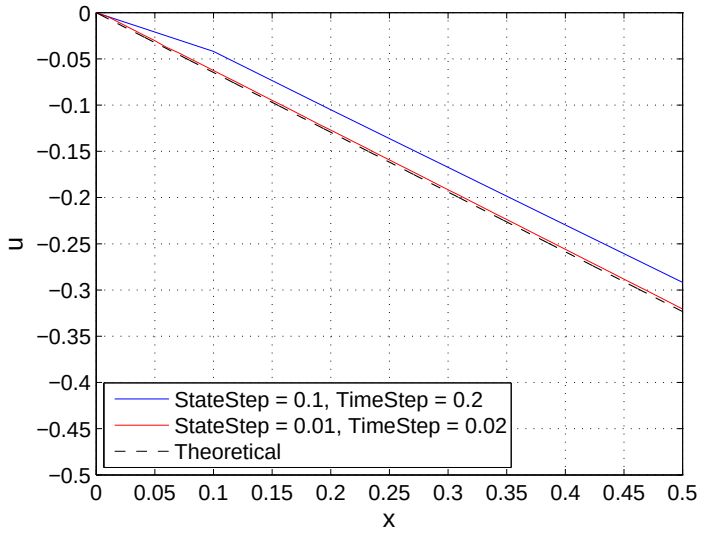

FIGURE 1. Optimal and approximated control rules.

Note that in this graph the optimal solution is presented as a dashed line, while our computed control rules are presented as solid lines. This convention is followed for subsequent graphs. The blue line is obtained on setting `StateStep` and `TimeStep` to 0.1 and 0.2 respectively.

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 MATLAB® routine `InfSim` with the following values for the parameters and functions described in Section 3.3.

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

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

`SimulationTimeStep`: For 10000 equidistant time steps each of size 0.01 this can be specified as ones(1,10000)/1000. This determines a horizon of length 10. In order to obtain a good approximation for the problem, it is necessary to choose a horizon for which the system has effectively become stationary and costless. The flattening of the timepaths in Figure 2 indicates that 10 is sufficiently large in this case.

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

Consequently `InfSim` is called as follows.

```
SimulatedValue = InfSim('TestProblem', 0.5,
    ones(1,10000)/1000)
```

`SimulatedValue` gives the simulated value of the performance criterion. This is 0.08090 (4 s.f.) with the choice of parameters given here, which compares favourably with the actual performance criterion

$$\min_u J\left(u, \frac{1}{2}\right) = \frac{5}{18 + 2\sqrt{481}} = 0.08082 \quad \text{(4 s.f.)}.$$

`InfSim` also produces the red lines in the graphs of timepaths shown in Figure 2 below.

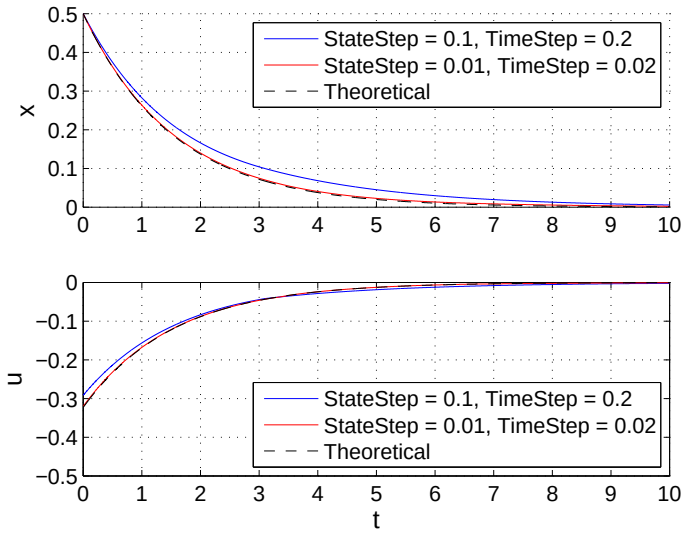

FIGURE 2. Optimal and approximated trajectories.

A.3.3. *Value vs. state.* Finally, the MATLAB® routine `InfValGraph` 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, 0:0.05:1 is used.

`SimulationTimeStep`: For 10000 equidistant time steps each of size 0.1 this can be specified as ones(1,10000)/1000.

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

Hence `InfValGraph` is called as follows.

```
InfValGraph('TestProblem', 0.5, 1, 0:0.05:1,
    ones(1,10000)/1000)
```

This produces the graph shown in Figure 3 below.

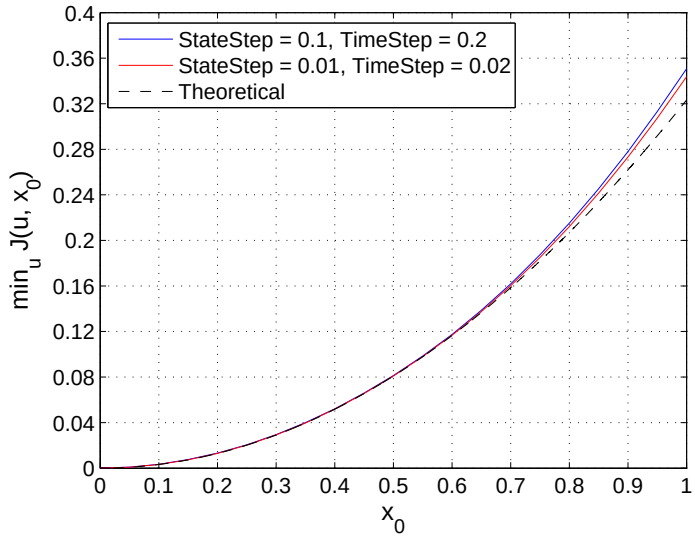

FIGURE 3. Optimal and approximated value functions.

A.4. **Optimisation Problem — With Constraints.** We will explain the syntax when constraint enforcement is necessary.

A.4.1. *Allowing for control constraints.* In the previous example we did not allow for control constraints, luckily it is not difficult to do so. For example, if we need to add a constraint for the lower bound of the control we follow the same method as before. However, now when calling `InfSOCSol` we input the lower control constraint where

previously there was $-\mathbf{Inf}$. For this new example the lower bound for the control is -0.2.[4] So, the `InfSOCSol` call is as follows:

```
InfSOCSol('Delta', 'Cost', 0, 0.5, 0.01, 0.02, 0.9,
    'TestProblem', { }, [ ], [ ], [ ], [ ], -0.2, Inf, [ ]);
```

This will produce then produce the following graphs starting from the control rules shown in Figure 4.

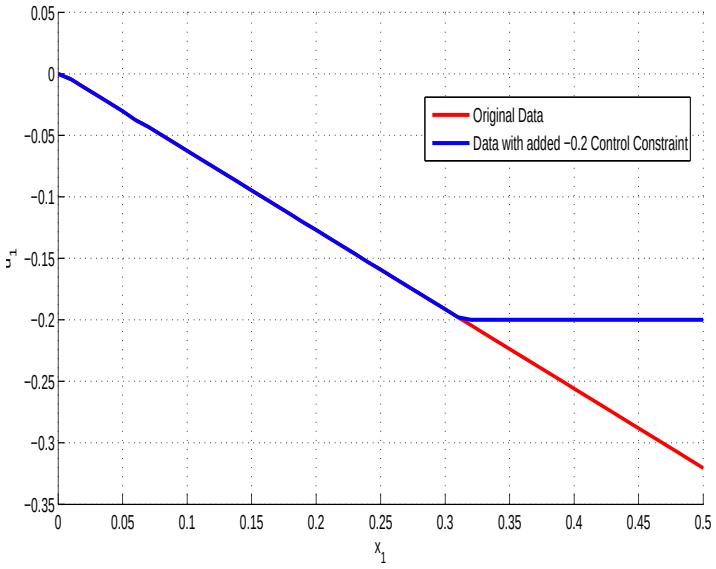

FIGURE 4. Approximately optimal control rules with a control constraint.

The following timepaths graph (see Figure 5) clearly shows what effect the control constraint of -0.2 has on the original profile.

Lastly the value function's graph is shown in Figure 6.

A.4.2. *Allowing for state constraints.* Problems with state constraints may also be encountered but are also simple to implement. Firstly we need to call a constraint file **StateConstraintFile**.

Suppose we want to enforce $x \geqslant 0.1$. This requires `Constraint.m` to be written as follows.

```
function [c, ceq] = Constraint(u, x, tS)
    c = -(x + tS*u) + 0.1;
    ceq = [ ];
```

Now we are ready to solve our original problem with the added state constraint. We call `InfSOCSol` as follows, with the final [ ] being replaced by 'Constraint' (please

[4] If for example it was -0.4 there would be no change to the graph; see the values of u in Figure 2 that are contained in approximately $[-0.3, 0]$.

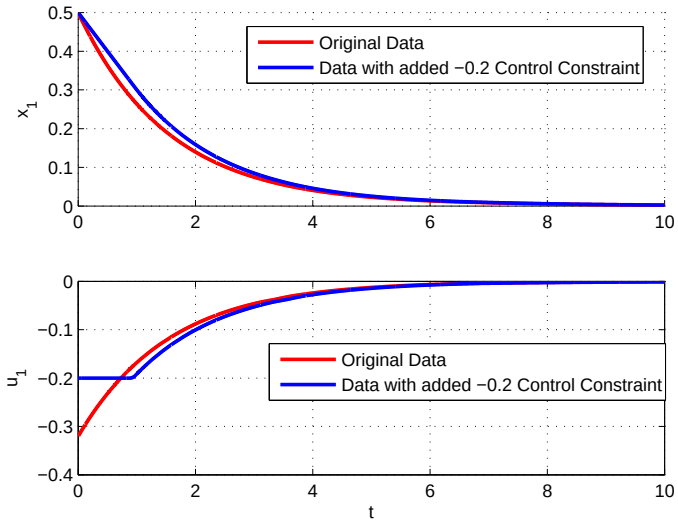

FIGURE 5. Approximately optimal state and control time profiles with a control constraint.

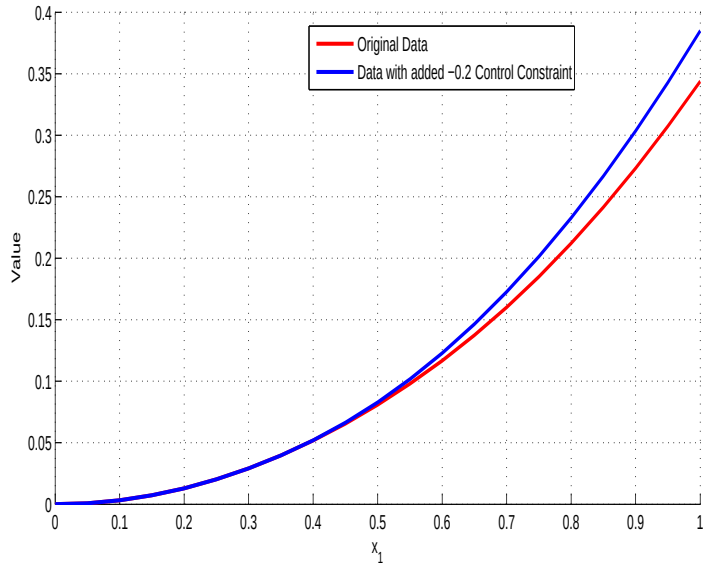

FIGURE 6. Approximately optimal value function with a control constraint.

note that when calling `InfSOCSol` in this example the lower control bound is set to -0.4 and not -0.2).

```
InfSOCSol('Delta', 'Cost', 0, 0.5, 0.01, 0.02, 0.9,
    'TestProblem', {}, [], [], [], [], -0.4, Inf,
    'Constraint');
```

It should also be noted that in the following graph the `State Steps` and `Time Steps` have been changed to 0.005 and 0.025, respectively, to produce smoother line of the control rules as shown in Figures 7–9 below.

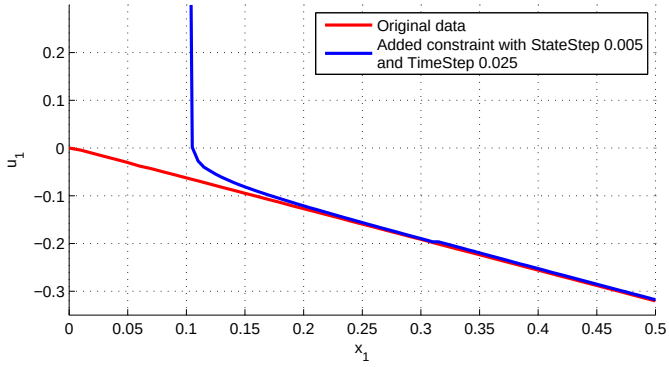

FIGURE 7. Approximately optimal control rules with a state constraint.

`InfSim` is called as per the previous example. With our current constraints it produces timepaths shown in Figure 8 below.

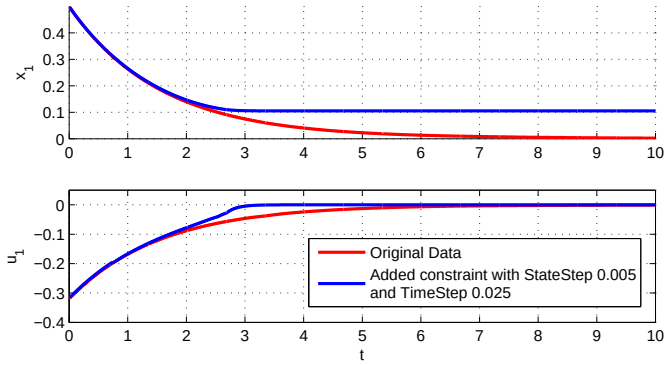

FIGURE 8. Approximately optimal state and control time profiles with a state constraint.

`InfValGraph` is also called as per the previous example, producing the optimal and approximated value functions in Figure 9 below.

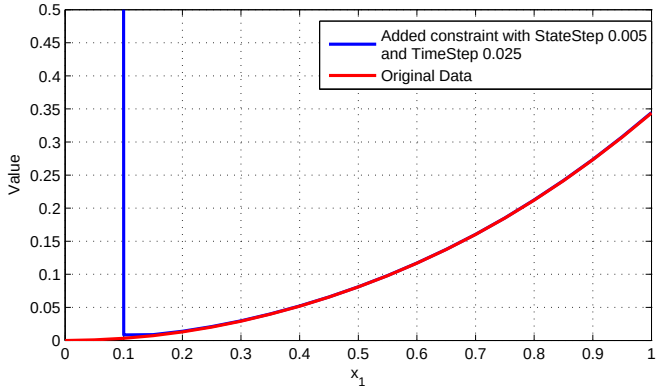

FIGURE 9. Approximated value function with a state constraint.

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 Economics and Finance, Victoria University of Wellington, 2006.

[AK08] Jeffrey D. Azzato and Jacek B. Krawczyk. `InfSOCSol2` an updated MATLAB® package for approximating the solution to a continuous-time infinite horizon stochastic optimal control problem. *Munich Personal RePEc Archive.*, 2008. Available at `http://mpra.ub.uni-muenchen.de/8374/` on 31/08/2009.

[Kra01a] 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 19/04/2008.

[Kra01b] Jacek B. Krawczyk. SOCSOL-II: A MATLAB package for approximating the solution to a continuous-time infinite horizon stochastic optimal control problem. Working paper, School of Economics and Finance, Victoria University of Wellington, 2001.

[Kra01c] Jacek B. Krawczyk. Using a simple Markovian approximation for the solution to continuous-time infinite-horizon stochastic optimal control problems. Working paper, School of Economics and Finance, Victoria University of Wellington, 2001.

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