InfsocSol3
An Updated MATLAB® Package for Approximating the Solution to a Continuous-Time Infinite Horizon Stochastic Optimal Control Problem

JACEK B. KRAWCZYK & ALASTAIR S. PHARO

Abstract. This paper describes a suite of MATLAB® routines devised to provide an approximately optimal solution to an infinite-horizon stochastic optimal control problem. The suite is an updated version of that described in [1] and [2]. Its routines implement a policy improvement algorithm to optimise a Markov decision chain approximating the original control problem, as described in [3].

2014 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 2014 research into
Computational Economics Methods
directed by Jacek B. Krawczyk

Correspondence should be addressed to either:

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

Alastair S. Pharo. Email: alastair.pharo@gmail.com

*Supported by the Victoria University Research Fund, Award Number 200757.

Contents

Introduction 1

1. Basic problem specification 2

2. Options and configuration 3

2.1. Purpose 3

2.2. Options 3

2.3. Configuring InfSOCSol 7

3. Solution of an optimal control problem 9

3.1. Purpose 9

3.2. Basic syntax 10

3.3. High-level API 10

3.4. Output 11

4. Determination of optimal control rules 11

4.1. Purpose 11

4.2. High-level interface 11

4.3. Low-level interface 12

5. Simulation of trajectories 13

5.1. Purpose 13

5.2. Implementation 13

5.3. High-level API 13

5.4. Low-level API 14

6. Determination of expected value 14

6.1. Purpose 14

6.2. Syntax 15

7. Loading and saving 15

7.1. Purpose 15

8. Using multiple CPUs 16

9. Large problems and sparse matrices 17

10. Tweaking optimizer settings 17

Appendices 19

A. Simple example of usage 19

A.1. Optimisation Problem 19

A.2. Solution Syntax 20
A.3. Retrieving results using the high-level API 20
A.4. Adding Constraints 24
B. An example of usage with a fisheries problem 28
B.1. Basic model 28
B.2. The fisheries problem in InfSOCSol 30
B.3. Solving the fisheries problem 32
B.4. Checking the results 32
B.5. Adding stochastic noise 33
C. Changes since version 2 37
C.1. Legacy API 37
D. Octave support 37
References 38
List of Figures 39
Index 40

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 [3] and [4]. This paper describes a suite of MATLAB®[1] routines implementing this method of approximating a solution to a continuous infinite horizon SOC problem.

The suite of routines developed updates and extends that described in [2]. Changes since version 2 of InfSOCSol are described in Appendix C. The presentation given here is based on that given in [6], which describes a similar suite of MATLAB® routines that may be used to solve finite 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 **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® functions similar to those introduced in [6]. The package is called InfSOCSol and is composed of four main modules:

1. *solution* of an optimal control problem (iss_solve)
2. determination of *optimal control rules* (iss_plot_contrule and iss_odm_control functions);
3. *simulation* of trajectories (iss_plot_sim and iss_sim functions); and
4. determination of expected value (iss_plot_valgraph and iss_sim functions).

InfSOCSol offers both a "high-level" and a "low-level" API, depending on the needs and preferences of the user. The former is oriented towards interactive or analytic usage. It is intended to be quick to get going with, and requires fewer lines of code. The latter are provided to enable embedding of InfSOCSol into other software.[2]

In the solution module, the iss_solve function provides a single interface for computing approximate numerical solutions to SOC problems. Its uses a policy improvement algorithm to optimise a Markov decision chain approximating the original control problem, as described in [3]. The resulting solution can then be stored to disk for later analysis (using the ProblemFile configuration setting; see Section 2), or used directly.

[1]See [5] for an introduction to MATLAB®.

[2]An example of such embedding is VIKAASA; see [7].

In the optimal control module, the solution is used to determine the continuous-time, continuous-state control to apply for a given situation. The low-level interface provides a function (iss_odm_control) that takes a state-space point and returns the optimal control for that point. The high-level interface (iss_plot_contrule) plots a graph showing how the control rule is affected by system state.

In the simulation module the control rule is used to simulate trajectories from a given start state, optionally perturbed by random noise. The high-level API provides a function (iss_plot_sim) that plots time profiles showing how the values of the system's state-space points and controls vary over as the system evolves. The low-level API provides direct access to simulation data via iss_sim.

In the value determination module, a high-level function (iss_plot_valgraph) is provided for plotting changes to the system's expected value, as the initial conditions change. It uses the simulation module to determine expected value.

1. Basic problem specification

In order to run InfSOCSol, it is necessary to define your problem. There are four basic details that InfSOCSol requires:

DeltaFunction is a function that represents the problem's equations of motion.[3]

- If the problem is deterministic, the function should return 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 should return a vector of length $2d$, 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 signature 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 should be ignored. The Time argument is included in order to allow reuse of function files between InfSOCSol and the finite-horizon SOCSol4L package (see [6]).

CostFunction is a function giving the instantaneous cost function $f\big(\mathbf{x}(t), \mathbf{u}(t)\big)$.[4]

The function should have a signature of the form

[3]Functions can be passed to InfSOCSol in any format that feval accepts. That is, strings representing function names, function handles or "inline" (anonymous) functions, as created with the inline command. Note however that because the the high-level API works by saving and loading data to disk, some function formats may not work as well as others. For this reason, it is suggested throughout this manual to write your functions in stand-alone files that are in the MATLAB® path, and to pass functions to InfSOCSol as plain strings. See Section 7 for more information, and Appendix A for an example using string function names.

[4]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 CostFunction.

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

where Control is a vector of length c and StateVariables is a vector of length d. As with the DeltaFunction, Time should be ignored.
The third argument, Conf, is a configuration data-structure which can be accessed if the cost function needs to make use of other information – see Section 2 for more details. As with the DeltaFunction, it is possible to reuse cost functions written for InfSOCSol in SOCSol4L (see [6]), but because SOCSol4L does not pass in a Conf parameter, you will need to use a varargin term.

StateLB is a row vector of length d giving the lower bounds of the rectangular area over which solutions to the SOC problem will be sought.

StateUB is a row vector of length d giving the upper bounds corresponding to those lower bounds given by StateLB. It must be the same length as StateLB.

By default, the search space given by StateLB and StateUB is discretised (as described in Section 3) into 11^d points. The number of points can however be changed by way of the StateStepSize configuration setting. See Section 2 for more details.

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.

2. Options and configuration

2.1. **Purpose.** All the modules in InfSOCSol accept a common set of options. In the low-level API, these options are managed using the iss_conf function. In the high-level API, these options are specified once using the iss_solve command, and are then saved to disk. Any option that is not specified will get a default value, so it is not necessary to specify any of these options unless the default is not satisfactory.

2.2. **Options.** A full list of options follows, with defaults given in parentheses. Some options are only useful in very particular applications.

2.2.1. *Options passed to* fmincon. InfSOCSol uses fmincon to perform numerical optimisation. These options are either passed directly as parameters to fmincon, or are passed as options via optimset.

See *Optimization Toolbox: fmincon* in MATLAB® help for further information about many of these options.

A, b, Aeq, beq (all [ ]). A and b 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 left empty (i.e., [ ]). then the problem is treated as having no linear inequality constraints on the control variable(s).
Aeq and beq 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 left empty (i.e., [ ]). then the problem is treated as having no linear equality constraints on the control variable(s).

UserConstraintFunctionFile ([ ]). A reference to a MATLAB® function representing the problem constraints (in particular, non-linear problem constraints). This parameter corresponds to the nonlcon parameter in fmincon. Although any function handle can be used, it it recommended to pass a string containing the function name (no `.m` extension), as for DeltaFunction.

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(u, x, Conf)
```

where u is a vector of length c, x is a vector of length d, and Conf is a configuration data-structure as described in Section 2.3. This means that all options (including default values) can be accessed via Conf.Options. For instance it is often helpful to make use of the time step for the relevant Markov chain time in constraints. This can be accessed as Conf.Options.TimeStep.

The default value ([ ]) indicates that there are no constraints requiring the use of a UserConstraintFunctionFile.

DerivativeCheck, Diagnostics, DiffMaxChange, DiffMinChange, Display, LargeScale, MaxIter , MaxSQPIter, OutputFcn, TolCon, TolFun, TolX, Algorithm, UseParallel. These options are all passed through to fmincon as options, using the opimset command. See [5] for more information on what these options do. A description of some of the more important options, with their default values in InfsocSol are as follows:

1. Display (`'off'`). 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).
2. Diagnostics (`'off'`). This controls whether fmincon prints diagnostic information about the cost-to-go functions that it minimises.
3. MaxFunEvals ($100c$). This sets fmincon's maximum allowable number of function evaluations. The default value is dependent on the number of controls, c.
4. MaxIter (400). This sets fmincon's maximum allowable number of iterations. It may be set to any natural number.
5. MaxSQPIter (Inf). 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.
6. TolCon (1e−6). 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.
7. TolFun (1e−6). 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.
8. TolX (1e−6). 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.

9. Algorithm ('active-set'). This determines the underlying optimisation algorithm used by fmincon. In general, the active−set and sqp algorithms appear to work well with InfSOCSol. The set of available algorithms differs depending on the version of MATLAB® being used.
10. UseParallel ('never'). This option can be switched on in order to tell fmincon to use more than one CPU. It is not recommended to turn on this option, as it is not compatible with InfSOCSol's PoolSize option.

If necessary, it is also possible to set:

11. DerivativeCheck ('off'). 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 (1e−1). 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.
13. DiffMinChange (1e−8). 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.
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.

2.2.2. *Discretisation.* These options control the discretisation of time and space in the discrete numerical approximation of the real continuous problem.

StateStepSize or States ((StateUB − StateLB)/ 10 and 11, respectively). These mutually exclusive options can be used to determine the number of points in the state grid. Only one of these options should be given; if both are provided States will take precedence. Both options are represented as a row vector of length d. If States is provided, then the number of states in each dimension should be given as a whole number. If StateStepSize is given, then the distance between points in the state grid should be given such that the 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 (1). The scalar δ to be used when formulating the Markov decision chain approximating the SOC problem (see [3]). 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.

2.2.3. *Simulation options.* These options are used by iss_sim, iss_plot_sim and iss_plot_valgraph to control InfSOCSol's simulation behaviour. For stochastic control problems, see also Section 2.2.4.

NumberOfSimulations (1). This is the number of simulations that should be performed. 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).
By default only a single simulation is performed.

LineSpec (`'r-'`). This specifies the line style, marker symbol and colour of time-paths. It is a string of the format discussed in the *MATLAB® Functions: LineSpec* section of MATLAB® help.
The default value gives a solid red line without markers.

TimepathOfInterest (0). This is an integer between 0 and $d + c$ (inclusive) that specifies which time-path(s) iss_plot_sim is to plot. If TimepathOfInterest is passed as zero iss_plot_sim plots time-paths for all state and control variables. Otherwise, if TimepathOfInterest is passed the value $i > 0$, InfSim plots the time-path of the i^{th} variable, where state variables precede control variables.

UserSuppliedNoise (-1). 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. A value of -1 (the default) disables this feature, causing random noise to be generated instead. See also Section 2.2.4.
Note that NumberOfSimulations should be 1 if UserSuppliedNoise is specified. If UserSuppliedNoise is left unspecified, iss_sim randomly selects a standard Gaussian noise realisation for each simulation. Naturally, UserSuppliedNoise has no effect on deterministic problems.

SimulationEnd (250). The final time value T at the end of simulation. The default value sets $T = 250$.

SimulationTimeStep (ones(1, 250)). 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.
The default value agrees with the default value of SimulationEnd, and partitions the time interval evenly into 250 blocks of unit-length.

VariableOfInterest (1). A scalar telling iss_plot_contrule and iss_plot_valgraph the index of the state variable to vary, i.e., numbers like “1” or “2” etc. have to be entered in accordance with the state variables’ order in the function DeltaFunction. The control rule profile appears with the nominated state variable along the horizontal axis.

ScaleFactor (1). A scalar used for plotting expected value graphs using iss_plot_valgraph. Maximisation problems must have all payoffs replaced by their negatives before entry into iss_solve, as it assumes that problems require minimisation. Setting the ScaleFactor to -1 “corrects” the sign on payoffs for maximisation problems.

2.2.4. *Stochastic control options.* These options control how InfSOCSol behaves when optimising and simulating a problem involving a stochastic process.

StochasticProblem (0). This should be set to 1 if your problem is stochastic. The default value is 0, i.e., the problem is assumed to be deterministic.

NoisyVars (d). This should be set to the number N, 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 this option.

NoiseSteps, Noise and NoiseProb (2, [−1, 1] and [1/2, 1/2], respectively). These options control how InfsocSol determines transition probabilities in stochastic settings using an Euler-Maruyama approximation (see [3] and [4]). By default InfsocSol constructs a "two-point" random variable, but could use a different scheme by altering these values.

2.2.5. *Other options.*

ProblemFile ([ ]). This is the prefix used to determine the file names to use to save results from running iss_solve. If specified, two files called `<<ProblemFile>>_options.mat` and `<<ProblemFile>>_solution.mat` will be created upon successful completion of iss_solve. The former contains all options used to run iss_solve, and the latter contains the resulting solution. Both files are in MATLAB®'s `.MAT` format, which can be opened and inspected using the load function.

DiscountRate (0.9). The scalar ρ.

ControlDimension. (1). This specifies the value c for your problem. It must be given as a natural number (in a string). The default value is 1, indicating that there is only one control.

ControlLB and ControlUB (−Inf and Inf, respectively). In general, vectors of lower and upper bounds (respectively) on the control variables. Setting ControlLB to −Inf (the default) indicates a a single control variable with no lower bound. Similarly, the default value of Inf for ControlUB indicates no upper bound as well. These vectors must have length of c (i.e., the value of ControlDimension).

Debug (0). If set to 1, iss_solve will output additional information to the MATLAB® console while running. This can be helpful in trying to understand how a particular solution is arrived at.

PolicyIterations (25). This specifies the maximum number of *value iterations* that may be performed in seeking a solution (see [3]). The default value is 25.

PoolSize (1). This specifies the number of CPUs to employ in running InfsocSol. Any value higher than 1 requires that MATLAB® have access to the Parallel Computing Toolbox. See Section 8 for more information.

StoppingTolerance (5*10^(Dimension−5)). This specifies the minimum norm of a "significant" policy change, as described in Section 3.4.

TransProbMin (0.01). When using sparse matrices, this is the minimum transition probability that is considered significant. See Section 9 for more details.

2.3. **Configuring InfsocSol.**

Each function in InfsocSol presents the same uniform interface for configuration. Any of the options described in Section 2.2 can be set at any time, regardless of whether they are useful at that stage. For instance, simulation options can be passed to iss_solve, even though they will not be useful until a simulation function (e.g., iss_sim) is called.

A *configuration data-structure* can be created by passing options to iss_conf. The syntax for this is as follows:

```
Conf = iss_conf(StateLB, StateUB, ...
                'Option1', value1, ...
                'Option2', value2, [···]);
```

where '[···]' is used to indicate that any number of options can be supplied. Here, StateLB and StateUB are vectors giving the minimum and maximum state-space points respectively, as detailed in Section 1.[5]

It is also possible to obtain a configuration data-structure using only the default options:

```
Conf = iss_conf(StateLB, StateUB);
```

An existing configuration data-structure can be altered or augmented by passing it to iss_conf as follows:

```
Conf = iss_conf(StateLB, StateUB, OldConf, ...
                'Option1', value1, ...
                'Option2', value2, [···]);
```

or:

```
Options = struct('Option1', value1, ...
                 'Option2', value2, [···]);
Conf = iss_conf(StateLB, StateUB, OldConf, Options);
```

where OldConf is a configuration data-structure returned from a previous call to iss_conf.

The only difference between the two formats described above is that in the second instance, a struct has been created and passed through to iss_conf, whereas in the first instance, the struct is created implicity. In either case, if no additional options are passed, the original OldConf data-structure will be returned unaltered.

Configuration options can be passed directly to InfSOCSol commands. For instance,

```
iss_sim(InitialCondition, ODM, ...
        DeltaFunction, CostFunction, ...
        StateLB, StateUB, ...
        'Option1', value1, ...
        'Option2', value2, [···]);
```

For low-level commands, this is generally a short-hand alternative to explicitly building a data-structure with the given options using iss_conf and then passing that data-structure to the command in question, as in the following example.

[5]Note that the resulting Conf object will depend upon the values given for StateLB and StateUB. That is, it is not advisable to change these values without also re-running iss_conf.

```
Conf = iss_conf(StateLB, StateUB, ...
                'Option1', value1, ...
                'Option2', value2, [···]);
iss_sim(InitialCondition, ODM, ...
        DeltaFunction, CostFunction, ...
        StateLB, StateUB, Conf);
```

When using the high-level API however (e.g., iss_plot_sim) building a configuration using iss_conf is not recommended, as the resulting configuration data-structure may override settings in the file. If this is required for some reason, the in-file configuration should be loaded first using iss_load_conf, as in the following example.

```
% Load settings from ProblemFile
[DeltaFunction, CostFunction, StateLB, StateUB, Conf] = ...
    iss_load_conf(ProblemFile);

% Augment settings using iss_conf, using StateLB and StateUB from the
% ProblemFile.
Conf = iss_conf(StateLB, StateUB, Conf, ...
                'Option1', value1, ...
                'Option2', value2, [···]);

% Pass the augmented configuration to iss_plot_sim.
iss_plot_sim('MyFile', InitialCondition, Conf);
```

Once a configuration data-structure has been created, all options passed to it (or the default values where no option was passed) are accessible via the Options field of the data-structure. For instance, the following retrieves the time-step, h.

```
% Assuming Conf has already been created/loaded ...
h = Conf.Options.TimeStep;
```

It is also possible to edit the configuration options in-place, but this is not recommended. Instead, one should pass the configuration data-structure to iss_conf in order to alter options.

More details and examples are given in the subsequent sections. Note that because setting up options will typically span multiple lines, 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.

3. Solution of an optimal control problem

3.1. **Purpose.** This section details using the iss_solve command from InfSOCSol to approximate a solution to a SOC problem with a Markov decision chain. When a ProblemFile option is given, both the configuration (as described in Section 2.3 and the solution are saved to disk, and can hence be subsequently accessed using iss_plot_contrule, iss_plot_sim and iss_plot_valgraph.

3.2. **Basic syntax.** iss_solve is called as follows.

```
[ODM, UOptimal, Value, Errors, Iterations] = ...
    iss_solve(DeltaFunction, CostFunction, ...
              StateLB, StateUB, Conf);
```

The parameters and return values are as follows. Unless using the low-level API, the return values can be ignored.

DeltaFunction, CostFunction, StateLB and StateUB are the core problem details, as described in Section 1.

Conf can be a configuration data-structure, a sequence of name-value pairs, or a combination of these two, as described in Section 2.3. It can also be omitted to use the default options.

ODM is a cell array containing the optimal control rules for each state in the discretised state space considered by InfSOCSol. It is not usually necessary to view the contents of this struct, but it is needed by low-level functions such as iss_odm_control and iss_sim. There are c cells in the array, and each cell contains a vector of length equal to the total number of discretised state space points.

UOptimal is the same data as in ODM, but arranged orthogonally. That is, the number of cells in the array is equal to the total number of discretised state space points, and each cell in the array is a vector of length c.

Value is a vector of length equal to the total number of discretised state space points, giving the value determination given in the last iteration of iss_solve. For more information on how to interpret this, refer to [3].

Errors is a vector of length equal to the total number of discretised state space points, whose value is zero if a feasible solution to the optimisation problem was found at each point, and one otherwise.

Iterations gives the number of policy iterations used by iss_solve.

3.3. **High-level API.** If intending to use the high-level API, it is important to specify a ProblemFile option at this stage. In this case the return values of the call to iss_solve can be ignored. For instance,

```
iss_solve(DeltaFunction, CostFunction, ...
          StateLB, StateUB, ...
          'ProblemFile', 'my_problem_file', ...
          [···]);.
```

where '[· · ·]' is used to indicate any other options used, as detailed in Section 2. This will create two `.MAT` files, `my_problem_file_options.mat` and `my_problem_file_solution.mat`, containing the problem description and the problem solution, respectively. The string `'my_problem_file'` can then be used to refer to these files when doing analysis using the high-level API.

3.4. **Output.** While running, iss_solve will output information the MATLAB® console. An example of what would be printed is as follows.[6]

```
* Iteration #1 ... completed.
* Iteration #2 ... completed; norm: 0.146290; # of differences: 142.
* Iteration #3 ... completed; norm: 0.020000; # of differences: 19.
* Iteration #4 ... completed; norm: 0.000001; # of differences: 8.
  – Norm is less than stopping tolerance of 0.005000; Stopping.

* Final value determination: −1574.665142
* Number of policy iterations: 4
* Errors: 137
* Error Percentage: 12.580349
```

Each "iteration" represents an iteration of the policy improvement algorithm (see [3]). From the second iteration onwards a check is performed to see whether that iteration resulted in a "significant" change in policy. This is determined by comparing the Euclidean norm of the the change in policy against a stopping tolerance level, which can be set via the StoppingTolerance option, and which by default is set to $5 \times 10^{d-5}$.[7] If the change is less than this stopping tolerance, the algorithm terminates and returns the control policy determined by the last completed policy improvement round.

The "final value determination" gives the value of $J(\cdots)$ in the last point in the discretised state space.

The number of errors is also displayed. This is the number of states in the state space for which fmincon returned a Flag value that was not equal to 1. Values that are not 1 are not necessarily a cause for concern, but if the percentage of such states is high, then it may be worth checking the Flags value returned by iss_solve to see what is going wrong. See [5] for more information on how to interpret the individual flag values.

4. Determination of optimal control rules

4.1. **Purpose.** Having solved a SOC problem using iss_solve, InfSOCSol can be used to produce controls for given state-space points.

4.2. **High-level interface.** iss_plot_contrule 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.

iss_plot_contrule is called as follows.

[6]If the Debug option is turned on, then this output will contain additional information.

[7]The norm is calculated as

$$\sqrt{((u_{1,1,i} - u_{1,1,i-1})^2 + \cdots + (u_{1,n,i} - u_{1,n,i-1})^2) + \cdots + ((u_{c,1,i} - u_{c,1,i-1})^2 + \cdots + (u_{c,n,i} - u_{c,n,i-1})^2)},$$

where n is the total number of discretised states under consideration (see 2.2.2), and $u_{a,b,c}$ is the control policy produced for dimension a, in state b from policy improvement iteration c.

```
iss_plot_contrule(ProblemFile, InitialCondition, Conf);
```

The parameters in question are as follows.

ProblemFile is a string designating the pair of files saved by iss_solve containing the SOC problem's details and its solution. See Section 2.2.5 for more information.

InitialCondition is a row vector of length d giving a point in the state space to use for constructing the plot. The plot will change the variable whose index is given by the VariableOfInterest option (defaults to 1) from its minimum to maximum value, as determined by the StateLB and StateUB parameters provided to iss_solve, thereby drawing a line through the state-space that intersects with InitialCondition.

Conf can be either a configuration data-structure, a sequence of name-value pairs, or it may be omitted entirely. Any options that are passed will be merged with those originally passed to iss_solve. See Section 2.3 for more information.
Options that are of particular relevance here are VariableOfInterest, which controls which variable will be changed, and LineSpec, which controls how the line in the plot appears. These options can be set at plot time, as shown in the following example.

```
iss_plot_contrule(ProblemFile, InitialCondition, ...
                  'VariableOfInterest', 2, ...
                  'LineSpec', '+');
```

If they are not set, then the values passed to iss_solve will be used, or otherwise, the defaults.

iss_plot_contrule may also be called with a single output argument, as in the following example.

```
ContValues = iss_plot_contrule('ProblemFile', InitialCondition, Conf);
```

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

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

That is, the rows of this array correspond to points of the VariableOfInterestth dimension of the state grid, while its columns correspond to control dimensions.

4.3. Low-level interface.

The function iss_odm_control can be used to return a single control vector for a given state-space point. To do this, it requires the ODM object returned by iss_solve.

```
ODM = iss_solve(DeltaFunction, CostFunction, StateLB, StateUB, Conf);
U = iss_odm_control(ODM, MyStateVector, StateLB, StateUB, Conf);
```

The first function call here uses iss_solve to generate a solution to the SOC problem. This solution is then fed into the call to iss_odm_control, along with a row vector MyStateVector , giving a state-space point.

As explained in Section 2.3, Conf can either be a sequence of name-value pairs, or a configuration data-structure (or a mixture). Note however that it is necessary to pass the same values for StateLB, StateUB and Conf to both function calls in order to get sensible results. For this reason, it is recommended to build an configuration data-structure using iss_conf.

The return value, U will be a row vector of length c, giving the optimal control policy at for the given state, according to InfSOCSol.

5. Simulation of trajectories

5.1. Purpose. This section describes how to run simulations using the continuous-time, continuous-state control rule derived from the solution computed by iss_solve. iss_plot_sim provides a high-level API for plotting time-paths of the state and control variables and the associated performance criterion values for one or more simulations. iss_sim provides a corresponding low-level interface for use when embedding simulation functionality in other MATLAB® code.

5.2. Implementation. The derivation of the continuous-time, continuous-state control rule from the solution computed by iss_solve 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.

5.3. High-level API. iss_plot_sim is called as follows.

```
SimulatedValue = iss_plot_sim(ProblemFile, InitialCondition, Conf);
```

The parameters and return value in question are as follows.

ProblemFile is a string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by iss_solve from the disk. See Section 2.2.5 for more information.

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

SimulatedValue, the return value of the function call, is a vector of the values of the performance criterion for each of the simulations performed. It can safely be ignored if not required.
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 iss_solve). This average is left for the user to compute.

Conf can be either a configuration data-structure, a sequence of name-value pairs, or it may be omitted entirely. Any options that are passed will be merged with those originally passed to iss_solve. See Section 2.3 for more information, and Section 2.2.3 in particular.

5.4. **Low-level API.** inf_sim is called as follows.

```
[SimulatedValue, StateEvolution, DeltaEvolution, Control] = ...
        iss_sim(InitialCondition, ODM, ...
                DeltaFunction, CostFunction, ...
                StateLB, StateUB, Conf);
```

The parameters and return values in question are as follows.

InitialCondition is the the same as described in Section 5.3.
ODM is the solution data-structure returned by iss_solve.
DeltaFunction, CostFunction, StateLB and StateUB give the basic problem specification. They are described in Section 1.
Conf can either be a sequence of name-value pairs, a configuration data-structure (or a mixture), or it may be omitted, as described in Section 2.3. Configuration options of particular relevance to iss_sim are described in Section 2.2.3.
SimulatedValue is a cell-array, each element of which is a scalar giving the value of a simulation run.
StateEvolution is a cell-array, each element of which is a matrix with d rows giving the simulation's trajectory in the state space. Each row of the matrix corresponds to a state-space point, and the number of columns matches the length of the SimulationTimeStep option, plus one.
DeltaEvolution is a cell-array, each element of which is a matrix with d rows, each row giving the value of the problem's equations of motion a discretised time step, as specified in SimulationTimeStep. The number columns in each matrix is equal to the length of SimulationTimeStep.
Control is a cell-array, each element of which is a matrix with c rows, each row giving the control choice made by iss_sim at a discretised time step, as specified by SimulationTimeStep. The number columns in each matrix is equal to the length of SimulationTimeStep.

6. Determination of expected value

6.1. **Purpose.** This section details the process of computing expected values for the continuous system (under the continuous-time, continuous-state control rule derived from the solution computed by iss_solve) as the initial conditions change. Only a high-level API (iss_plot_valgraph) is provided for this purpose, as expected values are already returned by iss_sim for low-level use.

In a similar spirit to iss_plot_contrule above, it deals with one state variable at a time (identified by VariableOfInterest), while the other state variables remain fixed.

6.2. **Syntax.** iss_plot_valgraph is called as follows.

```
iss_plot_valgraph(ProblemFile, InitialCondition, ...
                  VariableOfInterestValues, Conf);
```

The parameters are as follows.

ProblemFile is a string containing the name (with no extension) of the problem. This name is used to retrieve the solution produced by iss_solve from the disk. See Section 2.2.5 for more information.

InitialCondition is a vector determining the values of the fixed state variables, as described in Section 5.3.

VariableOfInterestValues is a vector containing the values of the VariableOfInterest at which the system's performance is to be evaluated. VariableOfInterest defaults to 1, but it can be changed by specifying its value as an option.

Conf can either be a sequence of name-value pairs, a configuration data-structure (or a mixture), or it may be omitted, as described in Section 2.3. Configuration options of particular relevance to iss_plot_valgraph are described in Section 2.2.3. In particular, ScaleFactor can be used to scale the the value graph by a constant. A negative number can be used to reverse the sign; useful when the "true" problem involves maximisation.

7. Loading and saving

7.1. **Purpose.** This section describes low-level functions included in InfSOCSol for saving and loading configuration and solution files. It is usually not necessary to use these functions directly, unless interacting with the low-level API.

iss_save_conf can be used to save a complete InfSOCSol problem description. Usage is as follows.

```
iss_save_conf(DeltaFunction, CostFunction, ...
              StateLB, StateUB, Conf);
```

DeltaFunction, CostFunction, StateLB and StateUB are the basic SOC problem details as described in Section 1. Conf is a configuration data-structure, as described in Section 2.3. Note that unlike elsewhere in InfSOCSol, options cannot be provided directly to iss_save_conf.

This function will find the ProblemFile option in the configuration file, and save to a file called `<<ProblemFile>>_options.mat` in `.MAT` format.

iss_load_conf can be used to restore a SOC problem previously saved using iss_save_conf. Usage is as follows.

```
[DeltaFunction, CostFunction, StateLB, StateUB, Conf] = ...
    iss_load_conf(ProblemFile);
```

As elsewhere, ProblemFile should be a string giving only the prefix part of the file name (i.e., omitting the `_options.mat` part). The values returned should be identical to the ones passed to iss_save_conf.

Note that MATLAB®'s behaviour when loading data involving anonymous or inline functions is not well understood by the authors. It appears to be necessary at least to have the file where the functions were defined present in the MATLAB® path, but there may be additional requirements. This can be a source of problems if attempting to use the iss_save_conf and iss_load_conf functions with a DeltaFunction or a CostFunction that is not defined in its own file. To avoid these issues it is recommended always define these functions as separate files, and to refer to them using strings, (e.g., `'MyCostFunc`) rather than function handles (e.g., @MyCostFunc).

iss_save_solution is used by iss_solve to save the solution from a successful run into a file. Using this function, one could potentially override a solution provided by iss_solve and then use that solution with the other functions provided in InfSocSol, such as iss_sim or iss_plot_valgraph. Usage is as follows.

```
iss_save_solution(ODM, Conf);
```

ODM should be a cell array as described in Section 3.2, and Conf should be a configuration data-structure created by iss_conf, as defined in Section 2.3. The solution will be saved to a file called `<<ProblemFile>>_solution.mat` in `.MAT` format, where ProblemFile should be given via an option in the configuration data-structure.

iss_load_solution can be used to retrieve the solution to an SOC problem previously solved using iss_solve. Usage is as follows.

```
ODM = iss_load_solution(Conf);
```

The file to be loaded will be given by the ProblemFile option in the Conf configuration data-structure.

8. Using multiple CPUs

InfSocSol provides the ability to utilise multiple CPUs in parallel using the PoolSize option (see Section 2.2). In MATLAB®, this uses the parfor loop construct, and requires that the MATLAB® Parallel Toolbox be available.

Using parallel processors can significantly reduce the amount of time needed to run iss_solve and iss_sim (and other high-level functions that depend on iss_sim). However, it is important to note that it comes at the cost of increased memory usage. For problems with very large numbers of state space points to consider, memory usage can become as much of an issue as computation time, so it is important to be aware of both when deciding how many processors to use.

Under MATLAB®, InfSocSol will start and stop a worker pool automatically if one is not already running. It is worth noting that starting and stopping the worker pool involves a time penalty, and as such for smaller problems, using the worker pool may actually increase the amount of time needed to run InfSOCSol. In our experience, this trade-off is worthwhile once a problem involves more than approximately 200 states.

If a worker pool is already running, InfSOCSol will attempt to detect whether it is running as the master process or as a worker process. In the former case, the parallel functionality will be enabled using the existing worker set-up. This means that the number of workers may not correspond to the PoolSize option. If running in a worker process, InfSOCSol will behave as if it is running on a single CPU, as it is not possible for workers to spawn more workers in MATLAB®. Whenever a worker pool is already running when InfSOCSol starts, it will not attempt to shut it down upon completion. This means that InfSOCSol can be embedded within other software that is already managing the worker pool without conflict.

Under Octave, worker processes are not used to enable parallel functionality, and parallel processing can be used to advantage regardless of how small the state grid is.

9. Large problems and sparse matrices

InfSOCSol will automatically start using sparse matrices for certain operations in iss_solve when necessary.[8] If this happens, a message is displayed on the MATLAB® console. This is done in order to avoid exceeding MATLAB®'s size limitations with normal matrices.

When sparse matrices are in use, the TransProbMin option becomes significant. Reducing this number will increase the accuracy of the SOC solution, but at the cost of using more memory.

The overall largest possible problem that could be analysed using InfSOCSol is platform-dependent. The largest data structure that InfSOCSol needs to create is matrix that is s-by-s, where s is the total number of discretised state-space points. Hence, the maximum problem size is probably close to or equal to the largest square sparse matrix that can be created.

10. Tweaking optimizer settings

When running iss_solve a policy improvement algorithm is used to approximate the optimal control solution to the problem, as described in [3]. Due to the numerical nature of routines such as fmincon however, iss_solve may in some cases require some tweaking in order to converge on the optimal solution, or in order to reach a sufficiently smooth approximation. The following example shows a transcript from an iss_solve run in which the solution failed to converge.

```
* Iteration #1 ... completed.
* Iteration #2 ... completed; norm: 0.335982; # of differences: 354.
* Iteration #3 ... completed; norm: 0.779021; # of differences: 143.
* Iteration #4 ... completed; norm: 0.203272; # of differences: 44.
* Iteration #5 ... completed; norm: 0.434971; # of differences: 27.
* Iteration #6 ... completed; norm: 0.476361; # of differences: 21.
```

[8] This depends on the platform that InfSOCSol is running on. InfSOCSol automatically falls back to using sparse matrices when an ordinary matrix of the required size cannot be created.

```
* Iteration #7 ... completed; norm: 0.437835; # of differences: 20.
* Iteration #8 ... completed; norm: 0.586157; # of differences: 23.
* Iteration #9 ... completed; norm: 0.396332; # of differences: 18.
* Iteration #10 ... completed; norm: 0.440907; # of differences: 19.
* Iteration #11 ... completed; norm: 0.479186; # of differences: 18.
* Iteration #12 ... completed; norm: 0.482928; # of differences: 25.
* Iteration #13 ... completed; norm: 0.556201; # of differences: 27.
* Iteration #14 ... completed; norm: 0.479499; # of differences: 23.
* Iteration #15 ... completed; norm: 0.514626; # of differences: 21.
* Iteration #16 ... completed; norm: 0.476361; # of differences: 21.
* Iteration #17 ... completed; norm: 0.481787; # of differences: 15.
* Iteration #18 ... completed; norm: 0.392402; # of differences: 21.
* Iteration #19 ... completed; norm: 0.342140; # of differences: 22.
* Iteration #20 ... completed; norm: 0.553498; # of differences: 32.
* Iteration #21 ... completed; norm: 0.540119; # of differences: 23.
* Iteration #22 ... completed; norm: 0.320200; # of differences: 23.
* Iteration #23 ... completed; norm: 0.553407; # of differences: 24.
* Iteration #24 ... completed; norm: 0.481787; # of differences: 19.
* Iteration #25 ... completed; norm: 0.569868; # of differences: 24.

* Final value determination: 6278.387249
* Number of policy iterations: 25
* Errors: 56
* Error Percentage: 14.000000

* All iterations were used.
```

Here the norm of control changes does not significantly reduce from one iteration to the next. Another problem that can arise is that iss_solve appears to converge, but examination of the solution shows that it is "jaggy" – that is, it contains large noticable imperfections.

Both of these issues can usually be resolved by changing some of the options that are fed into the optimizer. In our experience the two most important options are TolFun and MaxFunEvals. The former may need to be made smaller (i.e. the tolerance may need to be *tightened*) than its default value of 10^{-6}, whereas the latter may need to be increased from its default value of 100. Changing these values may make iss_solve slower, however, so they should only be altered when the default values are insufficient. Appendix A below shows an example of such tweaking.

A. Simple example of usage

This example corresponds to that given in the appendix of [6] (and to *Example 2.3.1* of [8]), but with discounting and an infinite horizon.

A.1. **Optimisation Problem.** 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

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

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

This is a linear-quadratic problem whose optimal solution exists without discounting. However, for consistency, here we solve a discounted problem with the discount rate $\rho := \frac{9}{10}$, 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 iss_solve$(\cdots)$, where the arguments inside the brackets are the same as those of Section 3.2. 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:

DeltaFunction. This is called, for example, Delta.m and is written as follows.[9]

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

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

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

The parameters used in iss_solve are described in Section 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

[9]It is also possible to specify functions in InfSOCSol as inline or anonymous functions, for instance, Delta = @(u,x,t)u; or Delta = inline('u','u','x','t');. However, when using the high-level interface, inline functions affect the robustness of the saved .MAT files, so they are not recommended. They are however perfectly fine to use with the low-level interface.

monotonically from 0.5 to a small positive value. Consequently smaller or larger values than these are unnecessary.

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

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

DiscountRate: This is given a value of 0.9. This is the default.

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

ControlLB and ControlUB: As the control variable is unbounded (which is the default), these options are not specified.

A.2. **Solution Syntax.** Consequently iss_solve could be called in MATLAB® as follows:

```
iss_solve('Delta', 'Cost', 0, 0.5, ...
          'StateStepSize', 0.01, ...
          'TimeStep', 0.02, ...
          'DiscountRate', 0.9, ...
          'ProblemFile', 'TestProblem');
```

'TestProblem' is just the prefix part of the names of the two result files saved by iss_solve and stored for later use (see Section 5, 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;
Conf = iss_conf(StateLB, StateUB, ...
                'StateStepSize', 0.01, ...
                'DiscountRate', 0.9, ... % not strictly necessary
                'TimeStep', 0.02);

% The 'ProblemFile' option could also have been set using iss_conf.
iss_solve('Delta', 'Cost', StateLB, StateUB, Conf, ...
          'ProblemFile', 'TestProblem');
```

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 file names.

A.3. **Retrieving results using the high-level API.** The high-level API provides three types of figures: control versus state policy rules, state and control time-paths, and value graphs.

A.3.1. *Control vs. state.* The MATLAB® routine iss_plot_contrule is used to obtain a graph of the control rule from iss_solve's solution. The following values are specified for the parameters described in Section 4.2.

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 VariableOfInterest) to vary. Consequently, this is set arbitrarily to 0.5.
VariableOfInterest: This is set to 1 (the default), as there is only one state variable to vary. If this example had more than one state dimension, VariableOfInterest could be any natural number between 1 and d (inclusive), depending on which dimension/variable was to be varied.
LineSpec: This is left unspecified, assuming its default of `'r−'`.
LineWidth: This is set to 2 in order to make the line stand out.

Consequently iss_plot_contrule is called as follows.[10]

```
iss_plot_contrule('TestProblem', 0.5, 'LineWidth', 2);
```

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

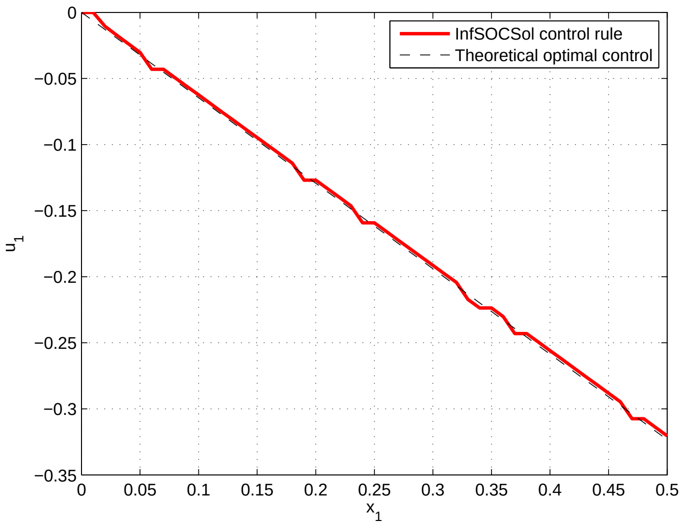


FIGURE 1. Optimal and approximated control rules for a simple unconstrained problem.

The control produced clearly approximates the theoretical optimal control, but suffers from some "jagginess". In order to rectify this, we can run iss_solve again with a tighter TolFun setting and a larger MaxFunEvals setting, as described in Section 10. This will allow the optimizer to take slightly longer in order to find a smoother solution.

[10]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.

```
iss_solve('Delta', 'Cost', 0, 0.5, ...
        'StateStepSize', 0.01, ...
        'TimeStep', 0.02, ...
        'DiscountRate', 0.9, ...
        'ProblemFile', 'TestProblem', ...
        'TolFun', 1e−8, ...
        'MaxFunEvals', 400);
```

Repeating our plotting command with the new solution file yields a smoother line, as shown in Figure 2. Note that the blue line is obtained on setting StateStepSize and TimeStep to 0.1 and 0.2 respectively and rerunning iss_solve.

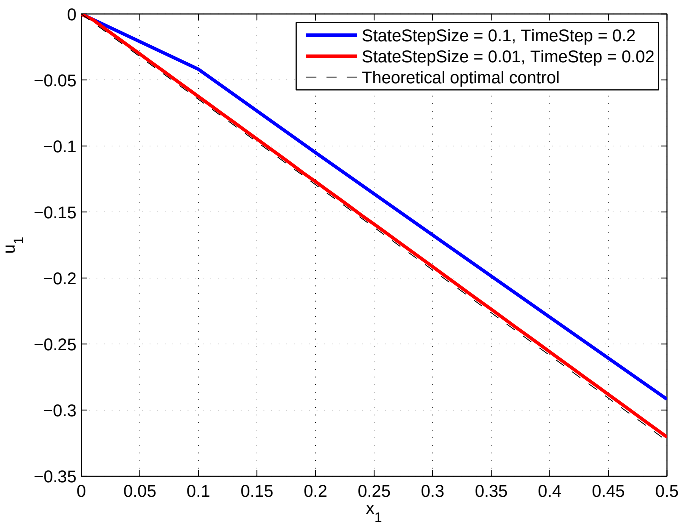


FIGURE 2. Optimal and approximated control rules for a simple unconstrained problem with reduced jagginess.

A.3.2. *State and control vs. time.* The files TestProblem_options.mat and TestProblem_solution .mat are used to derive a continuous-time, continuous-state control rule. The system is then simulated using this rule. We use the MATLAB® routine iss_plot_sim with the following values for the parameters and functions described in Section 5.3.

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 time-paths in Figure 3 indicates that 10 is sufficiently large in this case.
NumberOfSimulations, LineSpec and UserSuppliedNoise: These are not passed, as the default values suffice.

Consequently iss_plot_sim is called as follows.

```
SimulatedValue = ...
    iss_plot_sim('TestProblem', 0.5, ...
                 'SimulationTimeStep', 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

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

iss_plot_sim also produces the red lines in the graphs of time-paths shown in Figure 3 below.

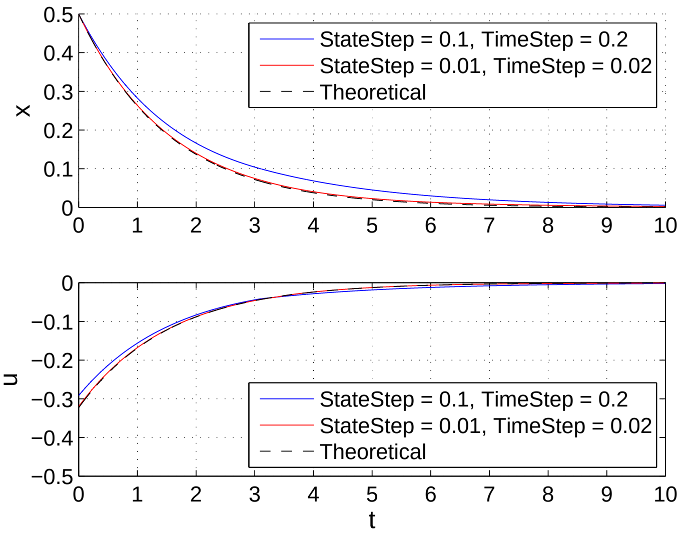


FIGURE 3. Optimal and approximated trajectories for a simple unconstrained problem.

A.3.3. *Value vs. state.* Finally, the routine iss_plot_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 6.2.

ProblemFile and InitialCondition: 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, VariableOfInterest and ScaleFactor: These are not passed, as the default values suffice.

Hence iss_plot_valgraph is called as follows.

```
iss_plot_valgraph('ProblemFile', 0.5, 0:0.05:1, ...
                  'SimulationTimeStep', ones(1,10000)/1000);
```

This produces the graph shown in Figure 4 below.

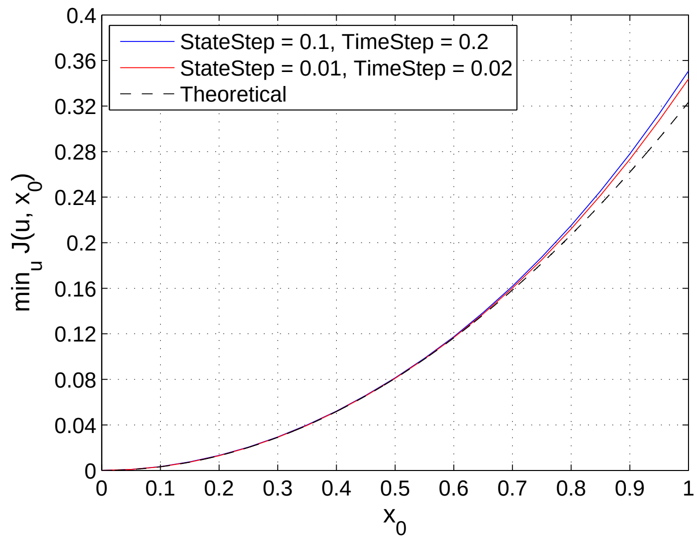


FIGURE 4. Optimal and approximated value functions for a simple unconstrained problem.

A.4. **Adding 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 iss_solve we append an extra option, ContolLB. For this new example the lower bound for the control is -0.2.[11] So, the iss_solve call is as follows:

```
iss_solve('Delta', 'Cost', 0, 0.5, ...
          'StateStepSize', 0.01, ...
          'TimeStep', 0.02, ...
          'DiscountRate', 0.9, ...
          'ProblemFile', 'TestProblem', ...
          'ControlLB', −0.2);
```

This will produce then produce the following graphs starting from the control rules shown in Figure 5.

[11]If for example it was -0.4 there would be no change to the graph; see the values of u in Figure 3 that are contained in approximately $[-0.3, 0]$.

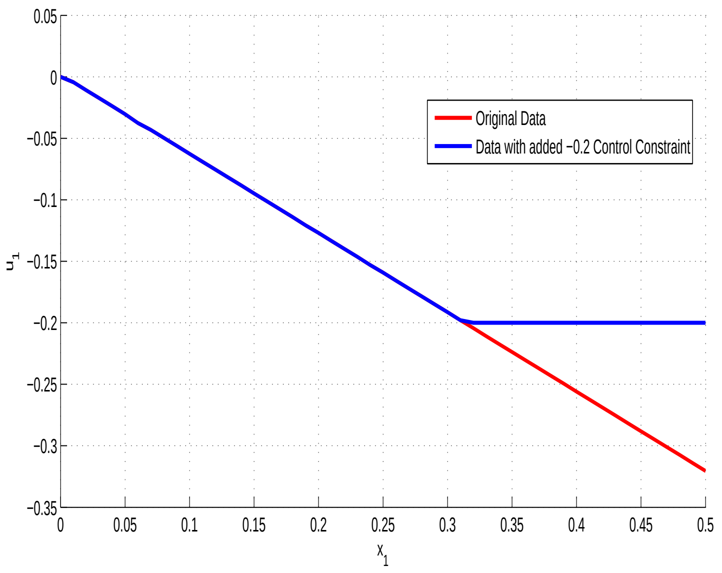


FIGURE 5. Approximately optimal control rules with a control constraint.

The following time-paths graph (see Figure 6) clearly shows what effect the control constraint of -0.2 has on the original profile.

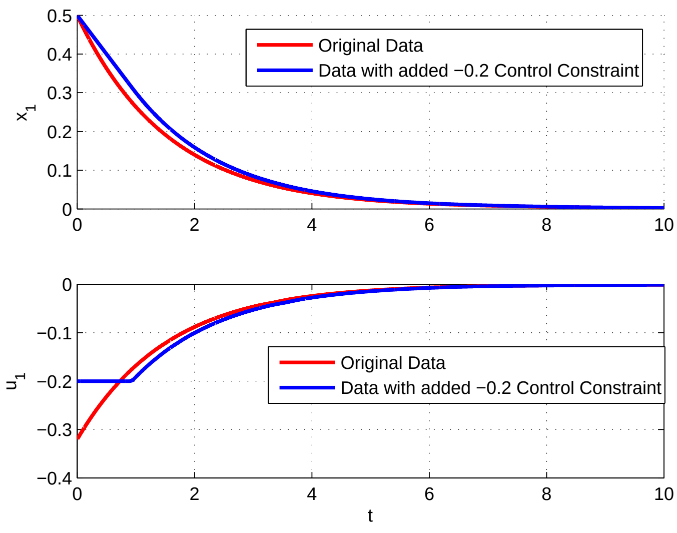


FIGURE 6. Approximately optimal state and control time profiles with a control constraint.

Lastly the value function's graph is shown in Figure 7.

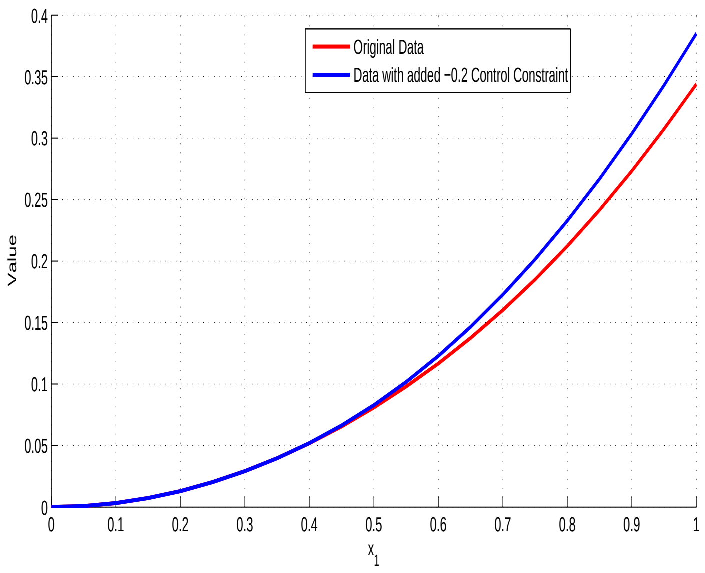


FIGURE 7. Approximately optimal value function with a control constraint.

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 create a constraint function for the UserConstraintFunctionFile option.[12]

Suppose we want to enforce $x \geqslant 0.1$. Then we would write a function like the following.

```
function [c, ceq] = Constraint(u, x, h, Conf)
    c = −(x + h*u) + 0.1;
    ceq = [ ];
```

The easiest way for InfsocSol to find this function is for it to reside in a file with the same name (plus a `.m` extension) in the MATLAB® path). Accordingly, we will place the above function into a file called `Constraint.m`.

Now we are ready to solve our original problem with the added state constraint. We call iss_solve as follows, with the additional UserConstraintFunctionFile option. [13]

```
iss_solve('Delta', 'Cost', 0, 0.5, ...
          'StateStepSize', 0.01, ...
          'TimeStep', 0.02, ...
          'DiscountRate', 0.9, ...
          'ProblemFile', 'TestProblem', ...
          'ControlLB', −0.4, ...
          'UserConstraintFunctionFile', 'Constraint');
```

[12]Unlike other function options in InfsocSol, this function has two return values, and so cannot be written as an inline or anonymous function.

[13]Please note that when calling iss_solve in this example the lower control bound is set to -0.4 and not -0.2

It should also be noted that in the following graph the StateStepSize and TimeStep options have been changed to 0.005 and 0.025, respectively, to produce smoother line of the control rules as shown in Figures 8–10 below.

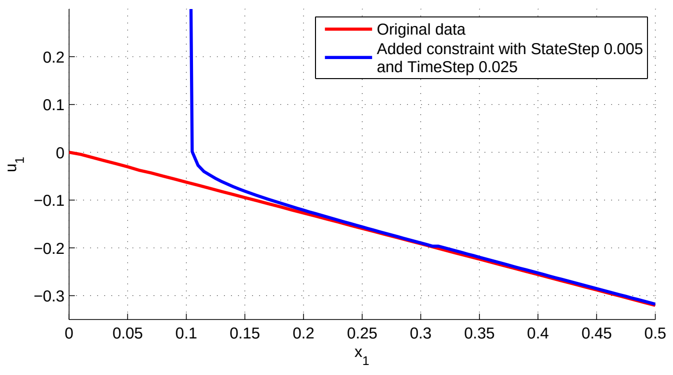


FIGURE 8. Optimal and approximated value functions with a state constraint.

iss_plot_sim is called as per the previous example. With our current constraints it produces time-paths shown in Figure 9 below.

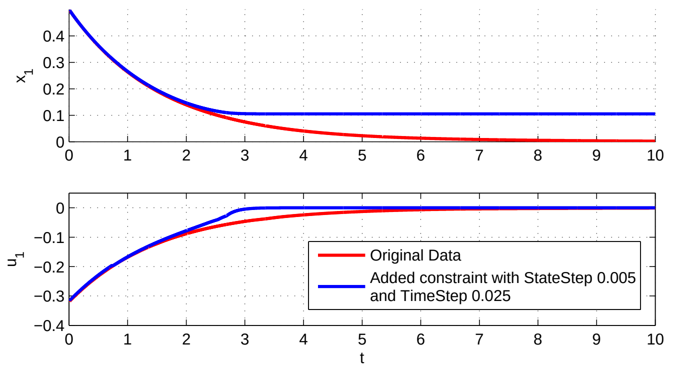


FIGURE 9. Approximately optimal state and control time profiles with a state constraint.

iss_plot_valgraph is also called as per the previous example, producing the optimal and approximated value functions in Figure 10 below.

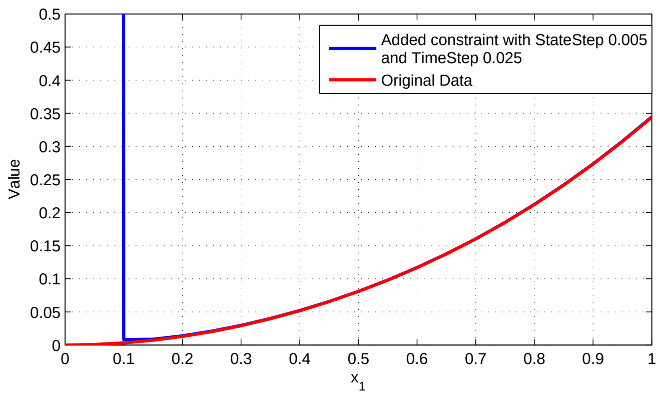


FIGURE 10. Optimal and approximated value functions with a state constraint.

B. An example of usage with a fisheries problem

This problem is adapted from [9], which is in turn adapted from [10]. Some of the results in [9] are themselves obtained using InfsocSol with VIKAASA (see [7]) to solve a *viability theory* problem. Here, this problem has been modified to represent an infinite-horizon profit-maximisation problem for a fishing fleet.

B.1. Basic model.

The elements of the model are as follows:

1. The system is described by of two dynamic variables: fish biomass, $x(t)$ and effort, $e(t)$. Effort is exerted by the fishing fleet to extract the resource (i.e. fish) at time t. This is a fixed fleet-size model, so there is no variation in capital to consider. A "catchability coefficient" q_x is defined to determine the quantity of biomass that each unit of effort extracts, relative to the total size of the biomass at the time. Thus, the *harvest rate* at time t is

$$h_x(t) = q_x e(t) x(t). \tag{6}$$

The product of q_x and $e(t)$ is generally termed the level of fishing mortality and it is expressed as a proportion of biomass.

2. Two constraints are given. *Firstly*, we suppose that a "safe minimum biomass level" (SMBL), $x_{\text{min}} > 0$ has been determined, which the industry's regulator will not allow fish stocks to fall below.
The *second* constraint concerns minimum and maximum levels of effort. The minimum level e_{min} is determined by social requirements for the preservation of employment and maintenance of traditional fishing communities. The maximum level e_{max} is determined not by any normative considerations but rather by the physical capabilities of the fishing fleet. Given that there is no variation in capital and labour, it is supposed that maximum effort is constrained by a fixed production capacity.
3. The fleet's objective is to maximise their infinite-horizon profit. Profit is given by

$$\pi_x(t) = p_x h_x(t) - ce(t) - C, \tag{7}$$

where p_x is the price of a unit of biomass (fixed in this model), and as explained above, $h_x(t)$ is the harvest rate, making $p_x h_x(t)$ the fleet's revenue flow. C is some fixed cost, and c is a variable cost for each unit of effort.

4. The fleet is able to select the effort level to exert, but effort variation is bounded by $U = [\delta^-, \delta^+]$, where $\delta^- < 0$ and $\delta^+ > 0$.
5. Fish population levels $x(t)$ are governed by a *logistic differential equation*,

$$\dot{x}(t) = r_x x(t) \left(1 - \frac{x(t)}{L_x}\right) - q_x e(t) x(t). \tag{8}$$

   This equation is based on [11], and is common in models of population growth. The resource grows at a rate proportional to r_x, up to the limit carrying capacity, L_x of the resource's ecosystem, less the harvest rate, $h_x(t) = q_x e(t) x(t)$.
   The differential equation for e is:

$$\dot{e}(t) = u(t) \in U = [\delta^-, \delta^+]. \tag{9}$$

   This equation represents bounds on the speed at which the fleet can change fishing intensity.

The parameter values for the considered model are:

$$r_x = 0.4 \tag{10}$$
$$L_x = 600 \tag{11}$$
$$x_{\text{min}} = 60 \tag{12}$$
$$p_x = 4 \tag{13}$$
$$q_x = 0.5 \tag{14}$$
$$c = 10 \tag{15}$$
$$C = 150 \tag{16}$$
$$e_{\text{max}} = 1 \tag{17}$$
$$e_{\text{min}} = 0.1 \tag{18}$$
$$\delta = 0.01 \tag{19}$$

With these parameters the differential equations become:

$$\dot{x}(t) = \frac{2}{5} x(t) \left(1 - \frac{x(t)}{600}\right) - \frac{1}{2} e(t) x(t) \tag{20}$$

and:

$$\dot{e}(t) \in U = \left[-\frac{1}{100}, \frac{1}{100}\right]. \tag{21}$$

The profit equation is:

$$\pi_x(t) = 2e_x(t)x(t) - 10e(t) - 150. \tag{22}$$

Finally, the discount rate ρ (see 1) has been reduced for this example from the default value of 0.9 to just 0.1. This means that the profit-maximising fishing fleet is more "patient", in that future profits have a higher present value. This has been added to this problem in order to simplify it by diminishing the trade-off between maintaining fish stock levels above the SMBL level in the long term, and exploiting the stock for short-term gain.

B.2. **The fisheries problem in InfsocSol.**

B.2.1. *State space.* The system state will be represented by a vector of length 2, $[x(t), e(t)]$. InfsocSol requires a rectangular area over which to seek optimal solutions, given by StateLB and StateUB. Constraints that we are interested in imposing on the system are a good place to start in determining these. In this case, those constraints are described in item 2. in Section B.1. Substituting the values from Equation 11, Equation 12, Equation 17 and Equation 18, we would have:

```
StateLB = [60, 0.1];
StateUB = [600, 1];
```

Note that the upper bound on biomass is because Equation 20 precludes $x(t)$ rising above $L_x = 600$.

B.2.2. *Equations of motion.* We will write the equations of motion for this problem, as described in Equation 20 and Equation 21 into a file called `fisheries_delta.m`.

```
function v = fisheries_delta(u,s,t)
  % Extract the named values from the state vector
  x = s(1);
  e = s(2);

  % Equation of motion for biomass
  xdot = 0.4*x*(1 - x/600) - 0.5*e*x;

  % Equation of motion for effort
  edot = u;

  % Combine into a vector
  v = [xdot, edot];
end
```

B.2.3. *Instantaneous cost function.* InfsocSol needs an instantaneous cost function to minimise. In fact, we have a profit function to maximise. As described in Section 1, we can multiply our profit function by -1 in order to convert our maximisation problem into a minimisation one suitable for InfsocSol. We will write this cost function into a file called `fisheries_cost.m`.

```
function cost = fisheries_cost(u,s,t,Conf)
  % Extract the named values from the state vector
  x = s(1);
  e = s(2);

  % Profit
  pi = 2*e*x - 10*e - 150;
  % Convert to cost
  cost = −pi;
end
```

B.2.4. *Constraint function.* In this section we define a function that specifies the *non-linear* constraints to be imposed on the control set, $U = \left[-\frac{1}{100}, \frac{1}{100}\right]$. For our purposes, any constraints that depend on either the system state or the system's dynamics need to be specified here. As stated in item 2. of Section B.1, effort levels have been proscribed by $e(t) \in [e_{\min}, e_{\max}] = [0.1, 1]$. This means that effort variation, $\dot{e}(t)$ should be constrained when "close" to the boundaries of this set. because InfSOCSol uses a Euler-Maruyama approximation, this becomes a matter of determining whether the "next" state lies outside of $[0.1, 1]$ or not. Thus, the constraint function can be expressed as follows.

```
function [c, ceq] = fisheries_constraint(u, s, Conf)
  % Extract the time−step.
  h = Conf.Options.TimeStep;

  % Determine the "next" state. The zero can be ignored.
  next_s = s + h*fisheries_delta(u, s, 0);

  % Extract the named values from the state vector
  x = next_s(1);
  e = next_s(2);

  % The inequality constraints should be a vector of numbers that are
  % negative iff the constraints are satisfied.
  c = [0.1 − e, e − 1, 60 − x];

  % There are no equality constraints
  ceq = [];
end
```

B.2.5. *Other options.* We use iss_conf to create a configuration data-structure here, as described in Section 2.3, in order to hold the other options. We need to set constraints on the controls, according to Equation 9, and we will decrease the StateStepSize in order to consider more points. Additionally, we will set a ProblemFile option, so that the results of running iss_solve are saved to disk.

```
% Note that StateLB and StateUB were defined previously.
Conf = iss_conf(StateLB, StateUB, ...
                'ControlLB', −0.01, ...
                'ControlUB', 0.01, ...
                'DiscountRate', 0.1, ...
```

```
                  'StateStepSize', (StateUB - StateLB) / 20, ...
                  'UserConstraintFunctionFile', 'fisheries_constraint', ...
                  'ProblemFile', 'fisheries');
```

B.3. **Solving the fisheries problem.** Now we use iss_solve to compute the solution to the fisheries problem. `'fisheries_delta'` `'fisheries_cost'` are references to the functions we created earlier. It is assumed that StateLB, StateUB and Conf were defined in the current workspace, as described in the previous section.

```
iss_solve('fisheries_delta', 'fisheries_cost', ...
          StateLB, StateUB, Conf);
```

Running this will save the problem and then solution into files `fisheries_options.mat` and `fisheries_solution.mat`, respectively.

B.4. **Checking the results.** Here we plot some control rules and simulations in order to see what the result returned by iss_solve looks like.

To plot control rules against state, we use iss_plot_contrule, as follows.

```
% 'fisheries' is the problem file name.
iss_plot_contrule('fisheries', [60, NaN], ...
                  'VariableOfInterest', 2);
```

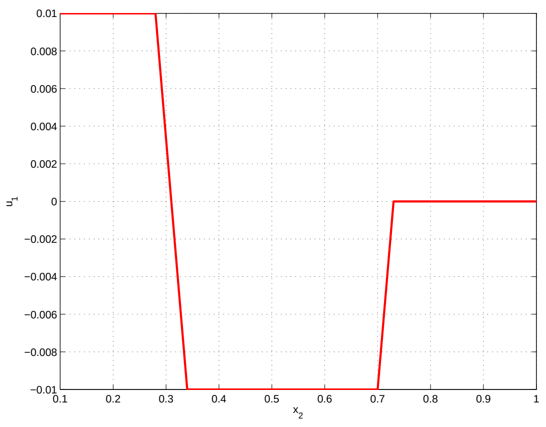
Here we have specified NaN instead of giving a second parameter, to make it clear that the second parameter will be varied. In Figure 11 we show the result of running this command for various values of fish biomass (x).[14] We can see that in all cases, the control policy involves reducing effort when effort is "high", and increasing it when it is "low". The threshold between "low" or "high" increases, the more biomass there is. This fits with intuition about how we would expect the fleet to manage the stock – fishing more when there are more fish, and less when there are fewer.

We can now plot some simulations using this control rule, in order to see how the system behaves when controlled in this way. To do so we use iss_plot_sim as follows.

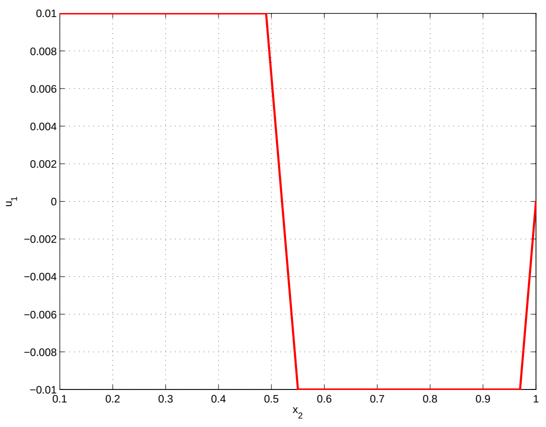
```
% Add this in order to be able to plot all graphs together.
hold on

% Use the 'LineSpec' option to change the colour of each simulation.
iss_plot_sim('fisheries', [78, 0.1], 'LineSpec', '-b');
iss_plot_sim('fisheries', [78, 0.9], 'LineSpec', '-c');
iss_plot_sim('fisheries', [582, 0.1], 'LineSpec', '-g');
iss_plot_sim('fisheries', [582, 0.9], 'LineSpec', '-m');
```

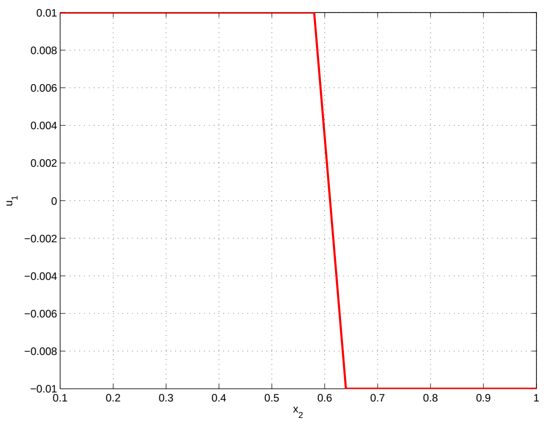
[14] Note that "x_2", as displayed in Figure 11 represents effort, e. The label is generated automatically by InfSOCSol.



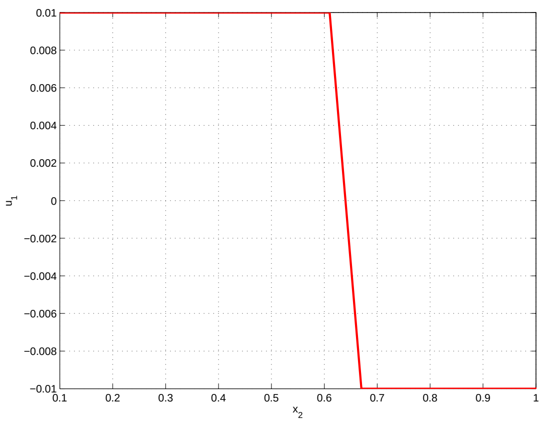
(A) $x = 60$



(B) $x = 240$



(C) $x = 420$



(D) $x = 600$

FIGURE 11. Controle rules for the fisheries problem using deterministic settings.

```
% Add a red dashed line to show where the SMBL is
subplot(3, 1, 1);
plot([0, 250], [60, 60], 'r--');
```

The results of this are displayed in Figure 12. We can see that the SMBL is avoided in all cases except where the initial state involves high effort and low stocks. In this case it appears that it is not possible, given the constraints of effort variation, to reduce effort quickly enough to avoid crossing the SMBL.

B.5. **Adding stochastic noise.** In this section we alter the fisheries problem presented in Section B.1 so that biomass, $x(t)$ becomes a stochastic, rather than a deterministic process. For simplicity, the stochastic component does not vary in state or control size. That is,

$$dx = \left(r_x x\left(1 - \frac{x}{L_x}\right) - q_x e x\right) dt + n d\mathbf{W}, \tag{23}$$

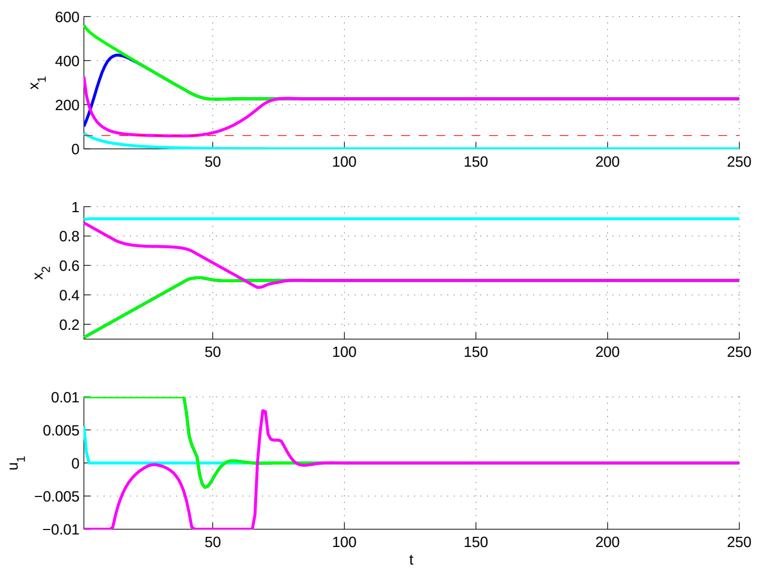


FIGURE 12. Simulations of optimally controlled trajectories for the fisheries problem using deterministic settings.

where n is a constant, and $\mathbf{W}$ is a Wiener process.[15] We will set the value of n to 10.

In order to analyse this new problem in InfSOCSol, we first need to define a modified delta function, as follows.

```
function v = fisheries_delta_stoch(u,s,t)
  % Extract the named values from the state vector
  x = s(1);
  e = s(2);

  % Equation of motion for biomass
  xdot = 0.4*x*(1 − x/600) − 0.5*e*x;

  % Equation of motion for effort
  edot = u;

  % Combine.
  v = [xdot, edot, 10, 0];
end
```

The only change is to include extra values in the return function, giving the size of the stochastic noise in each dimension.

We now create a new configuration data-structure, using the modified delta function, and passing the StochasticProblem option.

[15]It should be noted that this is not in fact a very good way of modelling a stochastic fish biomass process, as it does not preclude a shock from causing biomass to become negative. It is presented here purely as an illustration of InfSOCSol's capacities and is not supposed to present a realistic model.

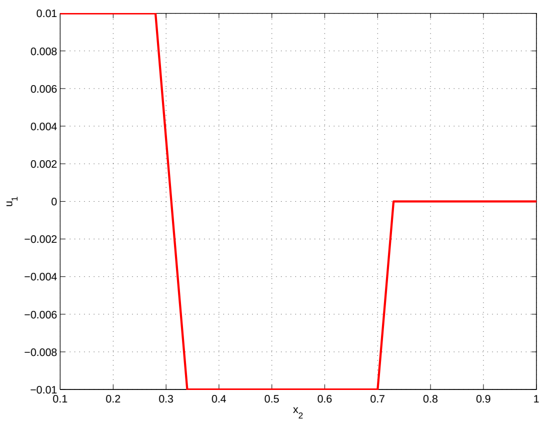
```
Conf = iss_conf(StateLB, StateUB, ...
                'ControlLB', -0.01, ...
                'ControlUB', 0.01, ...
                'DiscountFactor', 0.1, ...
                'StateStepSize', (StateUB - StateLB) / 20, ...
                'StochasticProblem', 1, ...
                'UserConstraintFunctionFile', 'fisheries_constraint', ...
                'ProblemFile', 'fisheries_stoch');
```

This problem will also use a different ProblemFile setting than the previous one. Note that we can continue to use the existing UserConstraintFunctionFile option under an "expected value" interpretation, where we are constraining the expected next state.

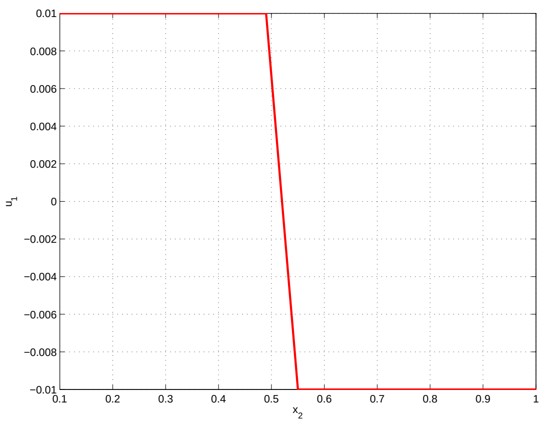
As before, we can solve this problem as follows.

```
iss_solve('fisheries_delta_stoch', 'fisheries_cost', ...
          StateLB, StateUB, Conf);
```

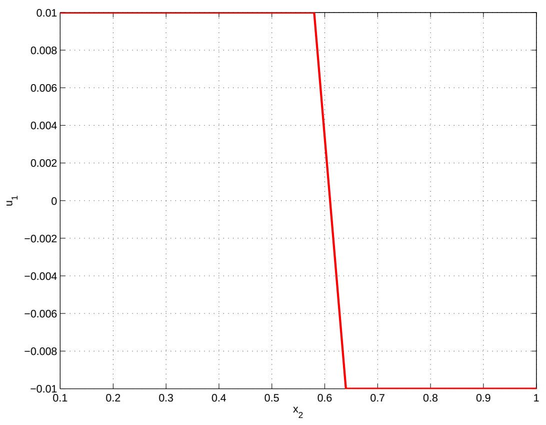
Plots of control rules and simulations are given in Figure 13 and Figure 14, respectively. For the simulations, the default options were used (as described in Section 2.3) meaning random noise has been generated, and only one simulation run was performed for each starting point.



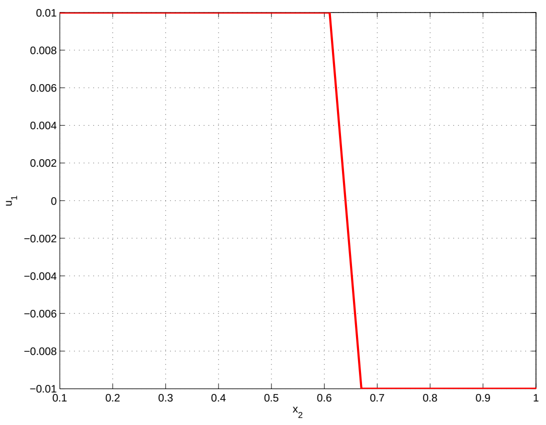
(A) $x = 60$



(B) $x = 240$



(C) $x = 420$



(D) $x = 600$

FIGURE 13. Controle rules for the fisheries problem with stochastic noise.

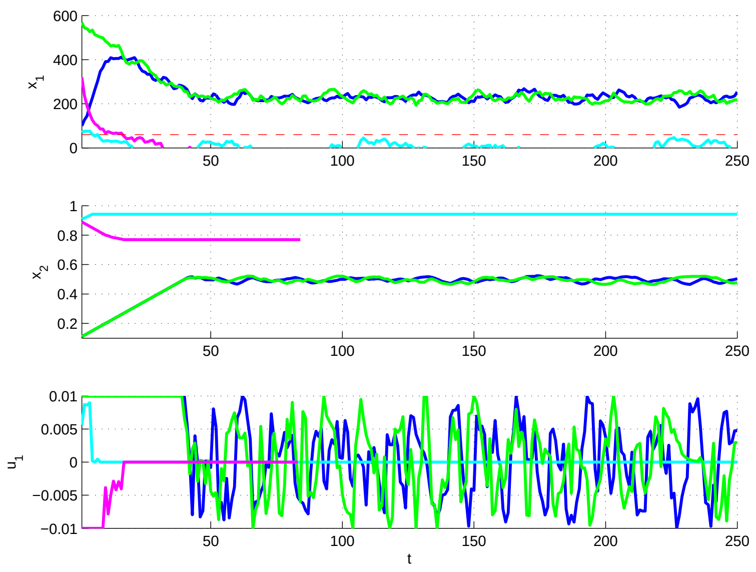


FIGURE 14. Simulations of optimally controlled trajectories for the fisheries problem with stochastic noise.

C. Changes since version 2

This section lists changes to InfSOCSol since the previous version, described in [2]. It can be used as a guide for migrating code depending on the previous version.

1. The old "legacy" API has been deprecated in favour of the one described in this document. See Section C.1 for details about these changes.
2. Introduction of a distinction between "high-level" and "low-level" APIs, enabling InfSOCSol to be integrated into other software more easily, and without plotting or saving to disk where this is not required.
3. Support for arbitrary length vector controls was introduced. This did not work in InfSOCSol2 due to a bug.
4. Support for parallel processing was introduced. See Section 8 for information.
5. Support for larger problems using sparse matrices to overcome matrix size limits in MATLAB®. See Section 9 for details.
6. Initial support for GNU Octave. Functionality is largely the same, but with some caveats. See Appendix D for details.

C.1. Legacy API.

Prior to version 3, InfSOCSol provided an API that more closely resembled the one provided by SOCSol4L (see [6]). In version 3, this interface has been deprecated in favour of the one described in this manual. However, in order to retain compatibility with code written for older versions of InfSOCSol, the old API is still available in InfSOCSol as a set of wrappers around the new functions. This is called the "legacy API". The functions provided are:

- InfSOCSol (replaced by iss_solve),
- InfContRule (replaced by iss_plot_contrule),
- InfSim (replaced by iss_plot_sim) and
- InfValGraph (replaced by iss_plot_valgraph).

Calls to these functions should work as before (see [2] for details), with some caveats:

- The old `.DPS` and `.DPP` file formats are no longer supported, so InfSOCSol3 will not be able to open SOC problem and solution files created with older versions. Instead, the wrapper functions now use the new `.MAT` format.
- The Options parameter to InfSOCSol required numeric options to be encoded as strings in version 2. String conversion is no longer performed in version 3, so these options will not be understood if they are still encoded.
- The text output provided when running the functions is similar, but not the same.

D. Octave support

InfSOCSol3 provides basic support for GNU Octave. This is done by utilising Octave's basic compatibility with MATLAB® and substituting the functionality that is only present in MATLAB® with equivalent functionality in Octave. From the perspective of the end-user, InfSOCSol should usually function the same on either platform. However, there are a number of issues to be aware of, which are listed here.

- Where possible, InfSOCSol will use the nonlin_min function, which is part of the "optim" package, available from Octave-Forge. See http://octave.sourceforge.net/ for information about how to install this package.
  The nonlin_min function provides a similar interface to MATLAB®'s fmincon function, including accepting most of the same optimization options as described in Section 2.2.1. A notable difference however is that the set of algorithms available to nonlin_min is not the same as for fmincon. InfSOCSol will therefore use the lm_feasible algorithm by default instead of active−set when running under Octave. Other algorithms can be specified using the Algorithm option, just as they can under MATLAB®.
- When the "optim" package is not available, InfSOCSol will fall-back on the (usually) slower built-in sqp function. This function has a different set of options from both fmincon and nonlin_min and as such most options in InfSOCSol relating to fmincon (e.g., Algorithm; see Section 2.2.1) will have no effect. The exceptions to this rule are the linear constraint options, UserConstraintFunctionFile, TolX and MaxIter, which are mapped to the appropriate options in sqp.
- In order to use multiple CPUs under Octave, the "parallel" package from Octave-Forge must be installed.

References

[1] Krawczyk, J. B. (2001) SOCSOL-II: A MATLAB package for approximating the solution to a continuous-time infinite horizon stochastic optimal control problem. Working Paper SOCSol-II/S, School of Economics and Finance, Victoria University of Wellington.

[2] Azzato, J. D. and Krawczyk, J. B. (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. Available at http://mpra.ub.uni-muenchen.de/17027/ on 03/09/2009.

[3] Krawczyk, J. B. (2001) Using a simple markovian approximation for the solution to continuous-time infinite-horizon stochastic optimal control problems. Working Paper inf_1.5, School of Economics and Finance, Victoria University of Wellington.

[4] Krawczyk, J. B. (2005) Numerical solutions to lump-sum pension fund problems that can yield left-skewed fund return distributions. Deissenberg, C. and Hartl, R. F. (eds.), *Optimal Control and Dynamic Games*, chap. 11, pp. 155–176, Springer. URL: http://www.vuw.ac.nz/staff/jacek_krawczyk/somepapers/pens_fund_SS_ta.pdf.

[5] The MathWorks Inc. (1992) *MATLAB®. High-Performance Numeric Computation and Visualization Software*.

[6] Azzato, J. D. and Krawczyk, J. B. (2006) 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.

[7] Krawczyk, J. B. and Pharo, A. (2011) Viability Kernel Approximation, Analysis and Simulation Application – VIKAASA Manual. SEF Working Paper 13/2011, Victoria University of Wellington. URL: http://hdl.handle.net/10063/1878.

[8] Krawczyk, J. B. (2001) A markovian approximated solution to a portfolio management problem. *Inf. Technol. Econ. Manag.*, **1**. URL: http://www.item.woiz.polsl.pl/issue/pdf/portitem.pdf.

[9] Krawczyk, J. B., Pharo, A., Serea, O. S., and Sinclair, S. (2013) Computation of viability kernels: a case study of by-catch fisheries. *Computational Management Science*, **10**, 365–396. URL: http://dx.doi.org/10.1007/s10287-013-0189-z, doi:10.1007/s10287-013-0189-z.

[10] Béné, C., Doyen, L., and Gabay, D. (2001) A viability analysis for a bio-economic model. *Ecological Economics*, **36**, 385–396.

[11] Schaefer, M. B. (1954) Some aspects of the dynamics of populations. *Bull. Int. Am. Trop. Tuna Comm.*, **1**, 26–56. URL: http://www.iattc.org/PDFFiles2/Bulletins/IATTC-Bulletin-Vol-1-No-2.pdf [cited 2013-08-05].

List of Figures

1 Optimal and approximated control rules for a simple unconstrained problem. 21

2 Optimal and approximated control rules for a simple unconstrained problem with reduced jagginess. 22

3 Optimal and approximated trajectories for a simple unconstrained problem. 23

4 Optimal and approximated value functions for a simple unconstrained problem. 24

5 Approximately optimal control rules with a control constraint. 25

6 Approximately optimal state and control time profiles with a control constraint. 25

7 Approximately optimal value function with a control constraint. 26

8 Optimal and approximated value functions with a state constraint. 27

9 Approximately optimal state and control time profiles with a state constraint. 27

10 Optimal and approximated value functions with a state constraint. 28

11 Controle rules for the fisheries problem using deterministic settings. 33

12 Simulations of optimally controlled trajectories for the fisheries problem using deterministic settings. 34

13 Controle rules for the fisheries problem with stochastic noise. 36

14 Simulations of optimally controlled trajectories for the fisheries problem with stochastic noise. 36

Index

Functions
InfContRule, 12, 37
InfSim, 6, 37
InfSOCSol, 37
InfValGraph, 37
iss_conf, 3, 7–9, 13, 16, 20, 31, 35
iss_load_conf, 9, 15, 16
iss_load_solution, 16
iss_odm_control, 1, 2, 10, 12, 13
iss_plot_contrule, 1, 2, 6, 10–12, 15, 21, 32, 37
iss_plot_sim, 1, 2, 5, 6, 9, 10, 13, 22, 23, 27, 32, 37
iss_plot_valgraph, 1, 2, 5, 6, 10, 14–16, 23, 24, 27, 37
iss_save_conf, 15, 16
iss_save_solution, 16
iss_sim, 1, 2, 5–10, 13, 14, 16
iss_solve, 1, 3, 6, 7, 9–22, 24, 26, 31, 32, 35, 37

Options and Parameters
A, 3
Aeq, 3
Algorithm, 4, 5, 38
b, 3
beq, 3
Conf, 3, 4, 8–10, 12–16, 19, 20, 26, 30–32, 35
Control, 2, 3, 14
ControlDimension, 7
ControlLB, 7, 20
ControlUB, 7, 20
CostFunction, 2, 10, 14–16, 19
Debug, 7, 11
DeltaEvolution, 14
DeltaFunction, 2–4, 6, 10, 14–16, 19
DerivativeCheck, 4, 5
Diagnostics, 4
DiffMaxChange, 4, 5
DiffMinChange, 4, 5
DiscountRate, 7, 20
Display, 4
Errors, 10, 11, 18
InitialCondition, 12–15, 21–23
Iterations, 10
LargeScale, 4
LineSpec, 6, 12, 21, 22
MaxFunEvals, 4, 18, 21
MaxIter, 4, 38
MaxSQPIter, 4
Noise, 7
NoiseProb, 7
NoiseSteps, 7
NoisyVars, 7
NumberOfSimulations, 6, 22, 23
ODM, 10, 12, 14, 16
OutputFcn, 4, 5
PolicyIterations, 7
PoolSize, 5, 7, 16, 17
ProblemFile, 1, 7, 9, 10, 12, 13, 15, 16, 20–23, 31, 35
ScaleFactor, 6, 15, 23
SimulatedValue, 13, 14, 23
SimulationEnd, 6
SimulationTimeStep, 6, 14, 22, 23
StateEvolution, 14
StateLB, 3, 5, 8, 10, 12–15, 19, 20, 30, 32, 35
States, 5
StateStepSize, 3, 5, 20, 22, 27, 31
StateUB, 3, 5, 8, 10, 12–15, 19, 20, 30, 32, 35
StochasticProblem, 6, 34
StoppingTolerance, 7, 11
TimepathOfInterest, 6
TimeStep, 4, 5, 9, 20, 22, 27, 31
TolCon, 4
TolFun, 4, 18, 21
TolX, 4, 38
TransProbMin, 7, 17
UOptimal, 10
UseParallel, 4, 5
UserConstraintFunctionFile, 4, 26, 35, 38
UserSuppliedNoise, 6, 22
Value, 2, 3, 10
VariableOfInterest, 6, 12, 15, 21, 23
VariableOfInterestValues, 15, 23