Documentation for "Stochastic Reachability for Systems up to a Million Dimensions"

Adam J. Thorpe, Vignesh Sivaramakrishnan, Meeko M. K. Oishi

October 26, 2019

Documentation for the algorithms presented in "Stochastic Reachability for Systems up to a Million Dimensions" by Adam J. Thorpe, Vignesh Sivaramakrishnan, Meeko M. K. Oishi [6].

Contents

List of Symbols

$\mathcal{H}$	Markov Control Process 3, 5, 6
$\mathcal{X}$	State Space 3, 4, 5, 6
$\mathcal{U}$	Control Space 3, 4
Q	Stochastic Kernel 3, 4
$\mathscr{B}(\mathcal{X})$	Borel σ-algebra associated with $\mathcal{X}$ 4
$Q(\cdot \mid x, u)$	Conditional probability measure 4
$\mathcal{S}$	Sample Set 4, 5
M	Number of Samples 4, 5
$\bar{x}_i$	State Samples 4
$\bar{u}_i$	Input Samples 4
$\bar{y}_i$	Output Samples 4, 7
π	Control Policy 4
$\mathcal{T}$	Target Set 5, 6
$\mathcal{K}$	Safe Set 5, 6
N	Time Horizon 5, 6

1 Start Here

The algorithms presented in [6] are designed to use *kernel distribution embeddings* [3] to solve the terminal-hitting time and first-hitting time stochastic reachability problems defined in [1, 5]. Detailed instructions for running the code are outlined in 2. This section also includes instructions for modifying the code for your own system. A description of the algorithms is presented in 5. The problem definitions are documented in section 4.

1.1 Entry Point

The two main entry points for the code are located in the files:

- run_unit_tests.m
- run_perf_tests.m

These files run the unit tests for the code, and run the performance tests to determine computation times for the algorithms. In order to run them, open the code in Matlab and run the following commands:

```
% Run the unit tests for the code.
run_unit_tests

% Run the performance tests to generate the computation time table.
run_perf_tests

% WARNING: The performance tests can take a significant amount of time to run.
% The high-dimensional repeated quadrotor example takes approx. 1.5 hours to
% run, and the performance testing framework will run this about seven times.
```

1.2 Plots

The figures from [6] can also be generated using the scripts in the /plots folder. Each script is labeled according to the plot number it generates in the paper.

2 Instructions

2.1 Running The Code

2.2 Generating the Figures

2.3 Modifying the Code

Using a different system for the algorithms is relatively simple. Simply substitute samples generated from your system in the examples to perform stochastic reachability calculations.

3 Samples

We consider a a Markov control process $\mathcal{H}$, which is defined in [5] as a 3-tuple:

$$\mathcal{H} = (\mathcal{X}, \mathcal{U}, Q) \tag{1}$$

where $\mathcal{X} \subseteq \Re^n$ is the state space of the system, $\mathcal{U} \subseteq \Re^m$ is the control space, and Q is a stochastic kernel $Q : \mathscr{B}(\mathcal{X}) \times \mathcal{X} \times \mathcal{U} \rightarrow [0, 1]$, which is a Borel-measurable function that maps a probability measure $Q(\cdot \,|\, x, u)$ to each $x \in \mathcal{X}$ and $u \in \mathcal{U}$ in the Borel space $(\mathcal{X}, \mathscr{B}(\mathcal{X}))$. A Markov control process can describe a wide class of stochastic, time-invariant systems, that can have either linear or nonlinear dynamics, as well as non-Gaussian disturbances. We consider a set $\mathcal{S}$ of M samples of the form $\{(\bar{x}_i, \bar{u}_i, \bar{y}_i)\}_{i=1}^{M}$ taken from the stochastic kernel, such that $\bar{y}_i$ is drawn i.i.d. from the stochastic kernel Q, and $\bar{u}_i$ is drawn from a fixed Markov policy π.

$$\bar{y}_i \sim Q(\cdot \,|\, x, u) \tag{2}$$
$$\bar{u}_i = \pi(\bar{x}_i) \tag{3}$$

The samples can be generated experimentally or via simulation, meaning they can be taken from real observations of the system evolution, or they can be generated using a known model. For demonstration purposes, all examples use samples collected via simulation. Once the samples are generated, the algorithm assumes no knowledge of the stochastic kernel Q or the disturbance.

The algorithms accept sample data drawn from a stochastic kernel Q. The data should be formatted such that the realizations of the stochastic kernel are formatted into the columns of a sample vector:

$$\bar{x} = [\bar{x}_1, \ldots, \bar{x}_M] \tag{4}$$
$$\bar{u} = [\bar{u}_1, \ldots, \bar{u}_M] \tag{5}$$
$$\bar{y} = [\bar{y}_1, \ldots, \bar{y}_M] \tag{6}$$

where the number of columns is M, and the number of rows is the dimensionality of the samples. For example, if $\bar{x}_i, \bar{y}_i \in \Re^n$, $\bar{x}$ and $\bar{y}$ should be $[n \times M]$, and if $\bar{u}_i \in \Re^m$, $\bar{u}$ should be $[m \times M]$.

3.1 System Samples

The system definition for the algorithms is a set of samples organized in a class called `SystemSamples`. For example, to generate samples for the discrete-time double integrator system with sampling time T,

$$\boldsymbol{x}_{k+1} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \boldsymbol{x}_k + \begin{bmatrix} \frac{T^2}{2!} \\ T \end{bmatrix} u_k + \boldsymbol{w}_k \tag{7}$$

we begin by defining the constant values for the system dimensionality, the sampling time of the system, and the system matrices. In the case of the stochastic double integrator, the system dimensionality is $n = 2$, and the input space dimensionality is $m = 1$. We choose the sampling time to be $T = 0.25$.

```
% Dimensionality of the state space samples.
n = 2;
% Dimensionality of the input space samples.
m = 2;
% Sampling time.
T = 0.25;

% Construct the state and input matrices.
A = [1 T; 0 1];
B = [(T^2)/2!; T];
```

Then, we can generate a set of samples, $\mathcal{S}$, using the known dynamics of the system. For this example, we choose $M = 1000$ samples to be generated randomly from a Gaussian distribution $x_0 \sim \mathcal{N}(0,1)$.

```matlab
% Number of samples.
M = 1000;

% Compute random initial states sampled from a zero-mean Gaussian.
X = randn(n, M);
% Compute the input samples. For this example, the input is chosen to be 0.
U = zeros(m, M);
% Compute the disturbance.
W = randn(n, M);

% compute the output samples.
Y = A*X + B*U + W;
```

Once we have generated our samples, they can be stored using the `SystemSamples` class. The class accepts the system dimensionality, specified as a vector `[n, m]`, and name/value pairs corresponding to $\bar{x}$, $\bar{u}$, and $\bar{y}$, and can be used like so:

```matlab
% Create a SystemSamples object.
samples = SystemSamples([n, m], 'X', X, 'U', U, 'Y', Y);
```

4 Problems

We consider two problems defined in [5]:

1. The **terminal-hitting time problem** (4.1), which is the probability that a system will remain safe over a time horizon and reach a target set at the final time.

2. The **first-hitting time problem** (4.2), which is the probability that a system will reach a target set at some time in the time horizon, and remain safe until it reaches the target set.

Let $\mathcal{T}, \mathcal{K} \in \mathcal{X}$ denote the *target set* and *safe set*, respectively. We denote the time horizon of the problem as $k \in [0, N]$, where $N \in \mathbb{N}$ is the final time of the system, and $k \in \mathbb{N}$ is the time index. We define sets in the code using functions that test whether a point is within the set or not. These functions can be Matlab function handles or defined as anonymous functions. The algorithms scale well with increased time horizons, provided that the functions used to check for set containment are not computationally intensive. These functions should be analogous to simple indicator functions.

4.1 Terminal-Hitting Time Problem

The terminal-hitting time problem is defined in [5] as the probability that a system $\mathcal{H}$ will reach a target set $\mathcal{T}$ at some time N while avoiding the *unsafe set* $\mathcal{X} \backslash \mathcal{K}$ for $k \in [0, N-1]$.

As an example, we consider a stochastic double integrator. We can construct the safe set and target set by specifying the sets as functions which check whether a sample is contained within the set.

```matlab
% Define the time horizon.
N = 5;

% Define the target set.
T = @(x) all(-1 <= x & x <= 1);
```

```

% Define the safe set.
K = @(x) all(-1 <= x & x <= 1);
```

Then, we can construct the terminal-hitting time problem using `TerminalHittingTimeProblem`.

```
% Define the terminal-hitting time problem.
problem = TerminalHittingTimeProblem(N, T, K);
```

4.2 First-Hitting Time Problem

The first-hitting time problem is defined in [5] as the probability that a system $\mathcal{H}$ will reach a target set $\mathcal{T}$ at some time $j \in [0, N]$ while avoiding the *unsafe set* $\mathcal{X} \backslash \mathcal{K}$ for $k \in [0, j-1]$.

As an example, we consider a stochastic double integrator. We can construct the safe set and target set by specifying the sets as functions which check whether a sample is contained within the set.

```
% Define the time horizon.
N = 5;

% Define the target set.
T = @(x) all(-0.5 <= x & x <= 0.5);

% Define the safe set.
K = @(x) all(-1 <= x & x <= 1);
```

Then, we can construct the first-hitting time problem using `FirstHittingTimeProblem`.

```
% Define the first-hitting time problem.
problem = FirstHittingTimeProblem(N, T, K);
```

5 Algorithms

The algorithms presented

See section 3 for more information about the required sample format and an example of generating samples via simulation.

5.1 Kernel Distribution Embeddings Backward Recursion Algorithm

```
% Define the algorithm Kernel Distribution Embeddings
algorithm = KernelDistributionEmbeddings('Sigma', 0.1, 'Lambda', 1);

% Compute the safety probabilities at the points in '(Xtest, Utest)'.
algorithm.ComputeSafetyProbabilities(problem, samples, Xtest, Utest);
```

5.2 Kernel Distribution Embeddings Backward Recursion (RFF) Algorithm

```
% Define the algorithm Kernel Distribution Embeddings (RFF)
algorithm = KernelDistributionEmbeddingsRFF('Sigma', 0.1, 'Lambda', 1, 'D', 1000);

% Compute the safety probabilities at the points in '(Xtest, Utest)'.
algorithm.ComputeSafetyProbabilities(problem, samples, Xtest, Utest);
```

6 Troubleshooting

In this section, we address some common pitfalls when using the algorithm.

1. **Choosing incorrect parameter values.** The algorithm parameter values can have a dramatic impact on the quality of the approximation computed via the algorithm. However, the parameter values are less important than in other kernel methods, such as Support Vector Machines [2]. This is handled somewhat by the normalization coefficient η, that is applied when the algorithm is computed. The default algorithm parameters are: $\sigma = 0.1$ and $\lambda = 1$. In practice, these values do not need to be changed, though it is possible to choose slightly better parameters via cross validation. A cross-validation scheme for choosing these parameters is discussed in [4].

2. **Choosing a low value of D.** For algorithm 2 using random Fourier features, the algorithm requires you to specify a parameter value D, which is the dimensionality of the reduced Gram matrix. For low-dimensional systems, this value may result in a higher-dimensional Gram matrix than algorithm 1. Increasing the value of D may help improve the approximation using algorithm 2.

3. **Choosing samples strictly within the safe set.** If all samples are strictly within the safe set, the probabilities computed near the boundaries of the safe set might be overly-optimistic. This is because the function approximation generated by the algorithm will not drop sharply to zero outside the safe set, pulling the probabilities computed near the safe set boundary down. Choosing samples over a wider area of the state space, including outside of the safe set, will help alleviate this issue.

4. **No output samples $\bar{y}_i$ are within the target set.** If no output samples are within the target set, the backward recursion will initialize itself with a vector of all zeros. Because the safety probabilities are multiplied together at each time step, this can introduce `NaN` values in some of the calculations and will produce undefined results. In practice, a small percentage of the samples, such as 5-10% must be within the target set for the algorithms to work properly.

5. **Confusing samples and test points.** There is a difference between what are called "samples" or "observations", which are used to create the approximation, and "test" or "evaluation" points, which are the points we want to evaluate the safety probabilities for via the algorithms. The samples can be thought of as the data, which should characterize the space as much as possible. The test points are the initial conditions we want to compute the safety probabilities for.

Several safeguards are present in the code which prevent some of these issues from arising, but care should still be taken to ensure that they are addressed before running the algorithm.

References

[1] Alessandro Abate, Maria Prandini, John Lygeros, and Shankar Sastry. Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems. *Automatica*, 44(11):2724–2734, 2008.

[2] Harris Drucker, Christopher JC Burges, Linda Kaufman, Alex J Smola, and Vladimir Vapnik. Support vector regression machines. In *Advances in neural information processing systems*, pages 155–161, 1997.

[3] Steffen Grünewälder, Guy Lever, Luca Baldassarre, Massimilano Pontil, and Arthur Gretton. Modelling transition dynamics in mdps with rkhs embeddings. In *Proceedings of the 29th*

International Coference on International Conference on Machine Learning, pages 1603–1610. Omnipress, 2012.

[4] Charles A Micchelli and Massimiliano Pontil. On learning vector-valued functions. *Neural computation*, 17(1):177–204, 2005.

[5] Sean Summers and John Lygeros. Verification of discrete time stochastic hybrid systems: A stochastic reach-avoid decision problem. *Automatica*, 46(12):1951–1961, 2010.

[6] Adam J. Thorpe, Vignesh Sivaramakrishnan, and Meeko M. K. Oishi. Stochastic reachability for systems up to a million dimensions, 2019.