Subset Simulation Toolbox v3.3.0

Institute of Flight System Dynamics
Technical University of Munich
mcmc.fsd@ed.tum.de

January 3, 2022
# Contents

1 Prologue .......................... 1
   1.1 Traditional Monte Carlo Studies ........................................ 2
   1.2 Using Subset Simulation .................................................. 2

2 Getting Started .................. 3
   2.1 Installation .............................................................. 3
   2.2 Configuration ........................................................... 4
   2.3 Run a Demo ............................................................... 5

3 Core Concepts .................... 9
   3.1 Namespaces ............................................................... 9
   3.2 Object Oriented Design .................................................. 10
   3.3 Algorithm vs. Data Classes ............................................ 11
   3.4 Passive Instrumentation ................................................ 12
   3.5 Shorthand Accessors .................................................... 13

4 The Main Features ................. 16
   4.1 Define Model Parameters ............................................... 16
   4.2 Apply Parameters to the Model ....................................... 18
   4.3 Tune Parameter Sampling .............................................. 19
   4.4 Adjust Subset Progress and Precision ............................... 21
   4.5 Run the Algorithm ...................................................... 22
   4.6 Understand Subset Results ............................................. 23
   4.7 Post-process Subset Results to Study Sensitivities .............. 25

5 Advanced Topics .................. 28
   5.1 Achieve Good Acceptance Ratios ..................................... 28
   5.2 Achieve Precise Results ................................................. 29
   5.3 Accelerate Slow Models ................................................ 29
   5.4 Resolve Parameter Dependencies with Copulas .................... 30
   5.5 Debugging Model Handle Outputs .................................... 30
   5.6 Dealing with Errors in the Model Handle ........................... 31
1 Prologue

The safety and performance of aircraft systems must fulfill stringent requirements defined by certification authorities. Performance-based requirements are specified with a probability threshold. The probability of failure of the performance of the system must be below the threshold. Depending on the severity of the outcome, the probability thresholds are typically of the order $10^{-5}$ (less severe) to $10^{-9}$ (catastrophic result in loss of life). Estimation of such low probabilities using naive Direct Monte Carlo methods generates a significant computational load that increases development costs. [1]

This toolbox aims to provide one simple tool to efficiently estimate the probability of failure for high-dimensional uncertain parameter spaces. Read at the quote above – the target probabilities we want to estimate are very low. Traditional Monte Carlo studies need exponentially many samples to provide an accurate estimate.


@article{Schwaiger2022,
    author = {F. Schwaiger and D. Shi and C. Mishra and L. H{"o}hndorf and F. Holzapfel},
    title = {A Modular Subset Simulation Toolbox for Matlab},
    journal = {AIAA Scitech 2022 Forum},
    year = {2022},
    doi = {TODO},
}
1.1 Traditional Monte Carlo Studies

Monte Carlo sampling techniques evolve around the “Law of Great Numbers”, that says when you have sufficient enough observation samples, their discrete class distribution (i.e. success – fail) converges to the hidden true occurrence probability.

Let’s look at a one-dimensional example. What is the probability of a variable $X \sim \mathcal{N}(0, 1)$ exceeding the threshold of 5? With traditional Monte Carlo, you would test this numerically like so:

```matlab
N = 100000;
X = normrnd(0, 1, [N, 1]);
P_f = mean(X > 5);
```

You can go ahead and play around with the number of samples N to see how many you need to get an accurate, repeatable result.

How many samples do you need? What is the target failure probability?

Now this is a very simple example and the “cost” of evaluating samples is very low. $X > 5$ is a vectorized operation after all. But we can agree on one thing: that is too many function evaluations if you have to run complex simulation or optimization jobs for each one of them.

1.2 Using Subset Simulation

Subset simulation at its core splits the problem into a series of subset problems, where the final failure probability is the product of each of the conditional probabilities from the intermediate subsets. Now, this manual is not about how subset simulation works in detail. If you are interested in the mathematics, please have a look at [2]–[4].

With our toolbox you can formulate the problem quickly like so:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'SampleDistribution', {0, 1}
    'NumberOfSamples', 100
    'ModelHandle', @(x) 5 - x
});
P_f = algorithm.incidentprobability();
```

Have a look at the Getting Started section to learn in detail how to setup the toolbox and make it work for you.
2 Getting Started

If you are a newcomer and not really familiar with MATLAB, you should start here. Otherwise you may skip ahead and directly learn the interface from our included demos and unit tests.

2.1 Installation

The minimum requirements for this toolbox are:

- MATLAB R2019b or newer
- MATLAB “Statistics and Machine Learning Toolbox”

We use the statistics toolbox to provide you with a familiar interface for all the parameter distributions and being able to integrate our algorithm into your workflow. While older versions of MATLAB might also work, our continuous integration system has only verified the tests for R2019b and newer.

Optionally, you may also install the following to use additional features, which can help you speed up function evaluations and run all of our included demos:

- MATLAB “Parallel Computing Toolbox”
- Simulink

To install the release version of our toolbox, first retrieve the archive from our Downloads page. Then unzip to a folder on your machine you have access to. And done! Go ahead to configuration.

Should you instead have access to our development version, which is a git repository, you may instead add the toolbox as a submodule dependency to your project. If we assume you aggregate all your submodules in a folder called External, you may do:

```shell
git submodule add <insert/url/to/subset-simulation.git> External/subsetsim
```
You can also use console commands from within MATLAB. Prefix the command above with an exclamation mark ! on the integrated shell:

```
>> !git submodule add ...
```

2.2 Configuration

Unpacking the toolbox code does not immediately make it visible in MATLAB. The source code must be added to the search path. In the toolbox, you can find the following folders:

- code/
- demo/
- test/
- SubsetSimulation.prj

To use the toolbox in your own code, adding code/ to the path is sufficient. But be aware that future versions of this toolbox might add additional dependencies and more folders, so we suggest using our included project file instead. Opening the project will add all required folders to your search path automatically:

```
openProject path/to/subset-simulation/
```

If you are using MATLAB projects already, you can add this toolbox as a Project Reference. To do so, open your own project and click the + References button:
Chapter 2 Subset Simulation Toolbox

Figure 2.1: Adding a project reference to another project

If you do not use project, we encourage you to try them out. They help keeping software modules together.

MATLAB projects can be cumbersome to work with in version control, because they tend to create hundreds of files. You can switch to using “monolithic” projects in MATLAB’s main setting. Then there will be only a handful of files.

2.3 Run a Demo

Now we have everything set up and running. You can verify yourself by typing `tulrfsd.mcmc.SubsetSimulation` and verifying an object is created.

In the main toolbox folder, you can also find a subfolder `demo/`. Go ahead and open the first example `demo/01-one-dimensional/`:

```matlab
%% you should get this probability
P_F_true = normcdf(5, 0, 1, 'upper')

%% set up algorithm to solve the probability estimate
algorithm = tulrfsd.mcmc.SubsetSimulation{
```
This is the same example we mentioned in the prologue. Here we show a few more options how you can set up the algorithm. Let's go through line by line.

\[
\text{P}_F_{\text{true}} = \text{normcdf}(5, 0, 1, 'upper')
\]

The first line of code gives us a baseline for the estimate of the probability of failure, which is:

\[
P_f(X > 5|X \sim N(0, 1)) \approx 2.8665 \cdot 10^{-7}
\]

The next lines configure an instance of `tulrfsd.mcmc.SubsetSimulation`. If you wonder about instances and classes, have a look at section Code vs. Data Classes. These lines will not compute anything yet. They simply assign the algorithm properties such that it does the job.

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 5 - x
    'EvaluationMethod', 'vector'
    'NumberOfSamples', 10000
    'SamplingMethod', 'infinity'
    'SampleDistribution', [0, 1]
    'ProposalDistribution', [0, 0.5]
});
```
If you wonder about this syntax ({...}), we have written a constructor that will assign all key-value pairs from a cell array. We hope this will save you unnecessary typing. In short, the following syntax are equivalent:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 5 - x
    'EvaluationMethod', 'vector'
});
```

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation();
algorithm.ModelHandle = @(x) 5 - x;
algorithm.EvaluationMethod = 'vector';
```

-ModelHandle is a reformulation of our failure criterion $X > 5$. It must be written such that the return value is the margin towards failure. Using the threshold value, we can subtract the samples to get a positive output for small values, and negative output for failing samples. ModelHandle can do more cool things than that, and you can read up on the features in a later chapter.

The EvaluationMethod defines how our toolbox will execute the ModelHandle you specified. This simple function @(x) 5 - x can be vectorized to optimize performance, so we can choose vector as the option. The default would be serial, which gives at least fail-safe results. See this section about other options.

NumberOfSamples defines how many samples will be evaluated each subset. In this example the number is high, but you can go as low as 100 for simple problems.

SamplingMethod is the core of our algorithm and drives the Markov Chain Monte Carlo process. Different methods have different properties, and we have an overview for you in this section. For now, it is sufficient to say that 'infinity'-sampling will give fast and stable results even for high dimensional problems.

SampleDistribution defines how the input parameters to your model are distributed. Most of the time you will be using normal distributions, but we provide a wide array of options. Since normal distributions might be your main concern, we have written an initializer that takes this simplified syntax: {mu, sigma}.

ProposalDistribution is usually derived from SampleDistribution and specific to the kind of SamplingMethod you choose. More on that in later sections. For 'infinity'-sampling, we
specifically need a normal distribution with $\sigma \in (0, 1)$. From the experience of [5], 0.5 seems to cover most applications well so far.

And that's it! Now, calling `incidentprobability()` on the preconfigured algorithm will give you the failure probability $P_F$, one cause sample that is on the border to the failure domain, and the array of subsets data containers that you can use to post-process more metrics.

$$[P_F, \text{cause}, \text{subsets}] = \text{algorithm}.\text{incidentprobability}()$$

Congratulations! You have now solved a problem that was (almost) impossible to solve using Direct Monte Carlo alone.
3 Core Concepts

This chapter is for you if you are not quite familiar with some concepts from object-oriented programming in MATLAB. If you are a poweruser, then you may skip ahead to the next chapter without missing important details.

3.1 Namespaces

Suppose you write a function `draw_samples.m` and we write a function `draw_samples.m`. Which function will be used in the end? MATLAB has complex rules about that and in short, it is determined by the search path sort order. One version of the file always replaces the other, only one can be accessed.

To prevent naming conflicts between your code and our code, we can place `draw_samples.m` inside a package folder `+tulrfsd/`. Now you can run the code independently like so:

```matlab
draw_samples() % will run your version
tulrfsd.draw_samples() % will run our version
```

We have chosen `tulrfsd` as a unique identifier for our organization, since it reflects our institute (FSD) affiliation with our university (TUM) and department (LRG). It does not contain any other special meaning, it is simply meant to be unique and short at the same time.

In fact, we like to put our code inside another stacked package folder to make the code unique across different projects. The package folder for this toolbox is `+tulrfsd/+mcmc/`, creating the prefix `tulrfsd.mcmc.*`. This means typical class constructors are:

```matlab
tulrfsd.mcmc.SubsetSimulation()
tulrfsd.mcmc.MultivariateNormalDistribution()
tulrfsd.mcmc.SubsetResult()
```

To save you writing the package path over and over, you can import the namespace once:
import tulrfsd.mcmc.*

SubsetSimulation()
MultivariateNormalDistribution()
SubsetResult()

You can find the official documentation for about how package folders create namespaces here: https://www.mathworks.com/help/matlab/matlab_oop/scoping-classes-with-packages.html

3.2 Object Oriented Design

MATLAB supports objects oriented programming (OOP) techniques. Instead of providing you with a bunch of functions, we use that approach to present you with a clean OOP interface to this toolbox. In summary, you will instantiate objects from our class definitions and execute their algorithms. This documentation is not to repeat OOP principles, but in the following what we have chosen to implement.

You can find the official documentation for object oriented programming here: https://www.mathworks.com/help/matlab/object-oriented-programming.html

The main interface to this toolbox is the class definition tulrfsd.mcmc.SubsetSimulation. Here all configuration comes together to set up the algorithm step-by-step. The moment you instantiate our `classdef`, you have an instance, also called an `object`. This object has `properties` and `methods` according to our definitions in the `classdef`. You can modify any properties using dot-notation:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation();
algorithm.SamplingMethod = 'infinity';
subsets = algorithm.subsetsim(); % will use the setting above
```

These properties remain with the object as long as it lives. When you finally trigger the algorithm, then the settings will be used.
The main benefit of using objects vs. structs to store configuration is that we can validate your input the moment you try to set the value. You cannot put garbage let’s say into SamplingMethod. Try it out for yourself! 42 is not a valid sampling method.

At some points we even use **hierarchies of objects** as configuration. Have a look at the following:

```plaintext
class SampleDistribution =
    tulrfsd.mcmc.MultivariateNormalDistribution(0, 1);
```

This creates another object of type `tulrfsd.mcmc.MultivariateNormalDistribution` and assigns it to the `SampleDistribution` property. The reason we do this is that we use a feature from OOP called **polymorphism**, which in short means that you can substitute a class by any of its child classes. In fact there are other distributions you can choose, more on that in the dedicated chapter on distributions:

- `tulrfsd.mcmc.CopulaDistribution`
- `tulrfsd.mcmc.ElementwiseDistribution`
- `tulrfsd.mcmc.NormalDistribution`
- `tulrfsd.mcmc.MultivariateNormalDistribution`
- `tulrfsd.mcmc.ManualDistribution`

All these classes have one thing in common: they are **child classes** of `tulrfsd.mcmc.Distribution`. When you inspect that class definition, you can see the keyword `(Abstract)` in the first line and the methods-block. `tulrfsd.mcmc.Distribution` is called an **interface**, or **abstract class**. It does not have any behaviour on its own, but defines the methods that child classes must implement. As long as our subset simulation receives an object that is an instance of `tulrfsd.mcmc.Distribution`, it will work. That way we can swap out distributions easily without writing a lot of if/else decision logic in the core algorithm.

### 3.3 Algorithm vs. Data Classes

In this toolbox, we like to separate **behaviour** from **storage**. We define classes that have only properties for configuration of algorithms and the respective methods. One example is `tulrfsd.mcmc.SubsetSimulation`.

In contrast, we also define **storage containers**, such as `tulrfsd.mcmc.SubsetResult`. Storage containers wrap a lean interface around a block of data to make it easier to access. They are defined by their **value**.
As such, we use a concept called **handle classes** for our algorithms and **value classes** for data. Their conceptual difference is, in short, that handle objects are **passed by reference**, whereas value objects are **passed by value**. Modifying properties on a reference to a handle class will modify the original:

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation();
reference = algorithm;
reference.SamplingMethod = "infinity";
% algorithm.SamplingMethod == "infinity"
```

Instead, modifying content on a value class will create a copy and leave the original untouched:

```plaintext
result = tulrfsd.mcmc.SubsetResult();
copied = result;
copied.Acceptance = true;
% result.Acceptance == []
```

This allows us to pass SubsetResult objects from method to method without destroying any previous result. Also it gives you, the user, the safety that we do not accidentally tinker with your data.

You can find out about all the differences between handle and value classes in the official documentation: [https://www.mathworks.com/help/matlab/matlab_oop/comparing-handle-and-value-classes.html](https://www.mathworks.com/help/matlab/matlab_oop/comparing-handle-and-value-classes.html)

### 3.4 Passive Instrumentation

Handle classes in Matlab support a feature called **event listeners**. You can define a method that is invoked when certain events happen:

```plaintext
algorithm.addlistener("OnSubsetComplete", @runsAfterEachSubset);
```

Events are defined with the class interface of tulrfsd.mcmc.SubsetSimulation. We define a couple, amongst others:

- "OnNewSubsetStart"
- "OnSubsetComplete"
• "OnEvaluateSample"
• ...

Inside our algorithm we can then publish any event using `self.notify("OnSubsetComplete")` – if you defined a listener your function handle will be executed. If not, nothing happens. That makes for a simple system to **instrumentate the algorithm**, i.e. define tools to inspect how it progresses.


---

### 3.5 Shorthand Accessors

Matlab classes also support **dependent properties**. These properties do not refer to actual memory locations, but define a virtual interface to others. When you define a dependent property, you must define setters and getters yourself:

```matlab
classdef SubsetSimulation < handle
    properties (Dependent)
        N0
    end

    methods
        function value = get.N0(self)
            value = self.NumberOfSamples;
        end

        function set.N0(self, value)
            self.NumberOfSamples = value;
        end
    end
end
```

With this setup, you can use `N0` as an alias for `NumberOfSamples`. The original property is easier more clear to use, the other might be fewer characters to type. We defined the following shorthand accessors for you:

- `p0` is an alias for `ConditionalProbability`
• $N_0$ is an alias for $\text{NumberOfSamples}$
• $N_s$ is an alias for $\text{NumberOfSubsets}$

See more on the topic here: https://www.mathworks.com/help/matlab/matlab_oop/access-methods-for-dependent-properties.html

Finally, we have defined useful constructors for the configuration containers, that work great together with property validation:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'Model', @(x) 5 - x,
    'Sample', makedist('norm', 0, 2),
    'Sampling', 'infinity',
    'Proposal', {0, 0.5}
});
```

In the example above, the cell array $\{0, 0.5\}$ will be converted into a `NormalDistribution` object on the fly. Property validation in the child object then applies. Also in the example you can see we abbreviated all of the property names.

Our constructors take either:

• key-value parameter pairs
• a $(N, 2)$ cell array of key-value pairs
• a struct with key-value pairs
• partial parameter names
• values that not of the target type, but at least convertible

Both the following are equal to the example above:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation( ... 'ModelHandle', @(x) 5 - x, ...
    'SampleDistribution', makedist('norm', 0, 2), ...
    'SamplingMethod', 'infinity', ...
    'ProposalDistribution', {0, 0.5} ...)
```
import tulrfsd.mcmc.*
config = struct('ModelHandle', @(x) 5 - x, ...
    'SampleDistribution', makedist('norm', 0, 2), ...
    'SamplingMethod', 'infinity', ...
    'ProposalDistribution', NormalDistribution(0, 0.5))
algorithm = SubsetSimulation(config)
4 The Main Features

These are the first recipes you should read to get a basic understanding how to use our algorithm.

4.1 Define Model Parameters

The algorithm estimates the probability that the response from a ModelHandle becomes negative, when evaluated with sample vectors. Sample vectors are drawn from parameter distributions. The following statements always hold true:

- sample vectors are row vectors
- sample vectors can be concatenated vertically

Inside a sample row vector, each element is drawn from another parameter. These parameters can be either correlated or uncorrelated. Either way, you must define a tulrfsd.mcmc.Distribution container to model a set of parameters and their optional dependencies. We provide the following basic distributions:

- tulrfsd.mcmc.ElementwiseDistribution for independent arbitrary parameter distributions
- tulrfsd.mcmc.NormalDistribution for independent normal distributions
- tulrfsd.mcmc.MultivariateNormalDistribution for dependent multivariate normal distributions

The distributions are explained in all detail in chapter Parameter Distributions. For now, we want to focus on the following:

- ElementwiseDistribution uses a vector of probability distributions
- NormalDistribution uses mean and std. dev. row vectors
- MultivariateNormalDistribution uses a mean vector and covariance matrix

You can put two distinct parameter distribution models into the algorithm in the distribution properties:
import tulrfsd.mcmc.*
algorithm = SubsetSimulation({
    'SampleDistribution', NormalDistribution([0, 0, 0], [1, 2, 3])
    'ProposalDistribution', NormalDistribution([0, 0, 0], [1, 2, 3] / 2)
})

In this example we have a three-dimensional problem with parameters distributed around zero with varying standard deviation. Also we have defined the proposal distribution with half the width of the sample distribution, which is most often a good idea.

We can simplify the example above using the shorthand notation we defined for you:

algorithm = tulrfsd.mcmc.SubsetSimulation({
    'SampleDistribution', {[0, 0, 0], [1, 2, 3]}
    'ProposalDistribution', {[0, 0, 0], [1, 2, 3] / 2}
})

If you want an array of element-wise distributions, you can write above example as such:

import tulrfsd.mcmc.*
algorithm = SubsetSimulation({
    'SampleDistribution', ElementwiseDistribution([makedist('norm', 0, 1), ...
        makedist('norm', 0, 2), makedist('norm', 0, 3)])
})

Or again, in less code:

algorithm = tulrfsd.mcmc.SubsetSimulation({
    'SampleDistribution', [arrayfun(@(s) makedist('norm', 0, s), 1:3)]
})

If you know you will only use normal distributions, you should use the {mu, sigma} syntax, since it maps to the NormalDistribution container which will be slightly faster than the ElementwiseDistribution container. The element-wise distribution container should be used preferably with non-normal parameter distributions, e.g. weibull, uniform,...
4.2 Apply Parameters to the Model

Given the sample row vectors mentioned in the previous section, you define a function handle ModelHandle that receives one or more row vector from samples and should return a scalar or row vector for the model outputs:

% e.g. sample == [0.57, -0.83, 1.33]
margin = myModelHandle(sample)
% e.g. margin == 0.42

You can inject your model like so:

algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @myModelHandle
});

Sometimes, anonymous function handles are all you need:

algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 5 - x
});

By default, your model will be called once for each sample row vector, which in situations can induce a lot of performance overhead. There is another accompanying property that defines how the model should be evaluated:

algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 5 - x
    'EvaluationMethod', 'vector'
});

Now the function handle will be invoked with a sample matrix, where the first (vertical) dimension are individual samples and the second (horizontal) dimension are the different parameters.

We provide the following options for EvaluationMethod:
Chapter 4 Subset Simulation Toolbox

<table>
<thead>
<tr>
<th>Option</th>
<th>Behaviour</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;auto&quot; (default)</td>
<td>&quot;parallel&quot; if a parpool is open, else &quot;serial&quot;</td>
<td>(1, M)</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>&quot;serial&quot;</td>
<td>call the function for each sample</td>
<td>(1, M)</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>&quot;parallel&quot;</td>
<td>same as above, but on the parpool</td>
<td>(1, M)</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>&quot;vector&quot;</td>
<td>call the function for a batch of samples</td>
<td>(N, M)</td>
<td>(N, 1)</td>
</tr>
</tbody>
</table>

If you are using our pre-defined instrumentation options to print output to the command line, the "vector" option will print progress dots in batches instead of individual samples, making it look like your algorithm might be stuck.

4.3 Tune Parameter Sampling

Markov Chain Monte Carlo sampling is the backbone of the subset simulation algorithm. You can configure the algorithm via the SamplingMethod property. We have implemented several different sampling schemes from literature:

- 'metropolis'
- 'metropolishastings'
- 'metropolishastingsewise' (the default)
- 'infinity'

'metropolis'-sampling proposes chain samples from ProposalDistribution, re-centered at the previous chain elements, then accepts or rejects sample vectors based on their distance from the previous elements and their failure margin. Any samples that do not make progress towards the failure domain are rejected. Samples that are close to their seeds are more likely to be accepted than samples far off from the seeds. This is done by comparing the ratio of the sample pdf \( p \) evaluated for the \( j + 1 \)-th proposed sample \( x^{(i+1)} \) divided by the pdf of the seed \( x^{(i)} \) at subset level \( (i) \) with a uniform random sample between 0 and 1 [6]:

\[
\text{accepted} = \left( \frac{p(x^{(i+1)})}{p(x^{(i)})} \geq \rho \in \mathcal{U}(0, 1) \right) \land \left( \text{ModelHandle}(x^{(i+1)}) < \text{threshold} \right)
\]
'metropolis'-sampling assumes symmetric proposal distributions $q$, which is true for most cases, such as normal and uniform distributions. For others, you should choose 'metropolishastings', a generalized form [6]:

$$\frac{p(x^{*(i+1)}|x^{(i)}) q(x^{(i)}|x^{*(i+1)})}{p(x^{(i)}|x^{*(i+1)}|x^{(i)})} \geq \rho \in U(0,1) \land \text{ModelHandle}(x^{*(i+1)}) < \text{threshold}$$

For symmetric proposal distributions the quotient $q(...)/q(...)$ cancels out.

For many parameters, 'metropolis' and 'metropolishastings' lead to poor acceptance ratios, because acceptance is decided for the whole parameter vector. If a single parameter moves out too far, the whole vector is discarded. With many parameters you should use 'metropolishastingsewise' or 'infinity' sampling instead.

Element-wise acceptance in 'metropolishastingsewise' can help improve the acceptance ratio in deeper subsets. In that case, the independent $p_k$ and $q_k$ are evaluated according to the scheme above, and parameter elements are accepted or rejected individually.

'infinity'-sampling is implemented according to [5]. The theory behind is that with an infinite number of Markov transitions, the proposed chain elements become distributed normally around the seeds.

$$x^{*(i+1)} \sim N(\sqrt{1 - \sigma^2} x^{(i)}, \sigma), \quad \sigma \in [0,1]$$

Here, $\sigma$ is relative to the width of the sample distribution and should be a value between 0 and 1:

- $\sigma = 0$ would make the chains immobile
- $\sigma = 1$ would make subset simulation equal to Direct Monte Carlo, since all samples are drawn from the sample distribution
- $0.4 < \sigma < 0.6$ is considered the optimal range [5][p. 53]

Since infinity sampling draws samples from the chain's limit state distribution, we can accept all sample vectors by default. Acceptance is then only determined by progress below the level threshold:

$$\text{accepted} = \text{ModelHandle}(x^{*(i+1)}) < \text{threshold}$$
To use infinity sampling, there is a special requirement for the `ProposalDistribution` property. It must be a `tulrfsd.mcmc.NormalDistribution` container with appropriate \( \sigma \):

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 5 - x
    'SamplingMethod', 'infinity'
    'ProposalDistribution', [0, 0.5]
})
```

The algorithm will return a validation error if the proposal distribution is not configured accordingly.

We only used a scalar \( \sigma \) for now, but it can be a row vector as well. Then each element affects another dimension of the parameter space. You could use that e.g. if you have a random uniform seed for a random number generator:

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'Model', @(x) abs(x(1)) - max(WhiteNoise(x(2)))
    'Sample', [someDist, makedist('uni', 0, 1e9)]
    'Sampling', 'infinity'
    'Proposal', {[0, 0], [0.5, 1]} % 1 == stationary
})
```

### 4.4 Adjust Subset Progress and Precision

To tune the precision of the MCMC sampling and how subset simulation progresses, you can use the following properties:

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'NumberOfSubsets', 1e1
})
```
Chapter 4 Subset Simulation Toolbox

 NumberOfSamples, or using the shorthand \( N_0 \), is the number of samples generated each subset. This number should be sufficiently large to accurately estimate the conditional probability, but as low as possible to get the most out of subset simulation.

 ConditionalProbability, or using the shorthand \( p_0 \), determines how fast subset simulation progresses towards lower probabilities. Each subset, the estimated failure probability will be multiplied with this value. The \( p_0 \times N_0 \) worst samples are declared as seeds for the next subset and \( N_0 \) new samples are generated. This value must be a real fraction of the number of samples. E.g. for 128 samples, you can use \( p_0 = 1/8 \), but not \( p_0 = 0.1 \).

 Read more on optimal values for \( N_0 \) and \( p_0 \) in the section about tuning the settings.

### 4.5 Run the Algorithm

Until now we only covered the configuration of the algorithm. Now let’s assume we have a well-defined problem formulation. You can then trigger the algorithm like so:

```matlab
subsets = algorithm.subsetsim();
% subsets is a (1,M) tulrfsd.mcmc.SubsetResult array
```

This will give you the raw output from the subset simulation. To get meaningful results, you can use the following post-processor methods:

```matlab
[P_f, cause] = algorithm.incidentprobability(subsets)
[beta_dist] = algorithm.incidentdistribution(subsets)
[P_f_bounds, cov_bounds] = algorithm.incidentbounds(subsets)
[structWithEverythingAndSubsets] = algorithm.study()
```
incidentprobability(subsets) will give you the estimated failure probability and a sample row vector cause that is on the boundary to the failure domain. \( P_f \) will be a value between 0 and 1, and \( \text{NaN} \) in particular if the algorithm failed to find failing samples. Then you can find an upper bound on the failure probability manually via:

\[
\% \text{ assuming } P_f \text{ was } \text{NaN} \\
P_f_{\text{max}} = (\text{algorithm.p0} ^ \text{algorithm.Ns})
\]

incidentdistribution(subsets) will give you a beta distribution fit for the true failure probability. The mean of the distribution is equal to the estimated failure probability.

incidentbounds(subsets) will give you lower and upper c.o.v. bounds as a \((1,2)\) double array. The first output argument applies the c.o.v. bounds to the estimate's failure probability to give you a value between 0 and 1. \( P_f \) will lie inbetween.

study() and incident***() trigger subsetsim() internally, if the subsets argument is not provided. It is common to skip the first method call and only use study() as the main interface:

\[
\text{result} = \text{algorithm.study()}
\]

### 4.6 Understand Subset Results

As noted in the previous section, subsetsim() or incidentprobability() will give you an array of tulrfsd.mcmc.SubsetResult objects. Let's see what they contain:

\[
\% \text{ algorithm} = \text{tulrfsd.mcmc.SubsetSimulation}({}
\% \quad \text{.. 'ModelHandle', @(x) } 2 - x
\% \quad \text{.. 'NumberOfSamples', 8}
\% \quad \text{.. 'ConditionalProbability', 0.25}
\% \quad \text{.. })};
\% \text{ subsets} = \text{algorithm.subsetsim()};
\% \text{ subsets}(5)
\text{ans} =
\text{SubsetResult with properties:}
\% \quad \text{Samples: [8x1 double]}
\% \quad \text{Outputs: [8x1 double]}
\% \quad \text{Accepted: [8x1 logical]}
\% \quad \text{Threshold: 1.0306}
\]

Institute of Flight System Dynamics, Technical University of Munich 23
In this example, the acceptance ratio in the 2nd subset was 3/8, which is very low, but understandable for this low number of samples.

Let us now have a closer look at the first subset:

```plaintext
>> [subsets(1).Samples, subsets(1).Outputs, subsets(1).Accepted]
ans =
  -0.0479    3.0479    1.0000  % 1st seed
   1.7013    1.2987    1.0000
  -0.5097    3.5097    1.0000
  -0.0029    3.0029    1.0000
   0.9199    2.0801    1.0000
   0.1498    2.8502    1.0000
  1.4849    1.5951    1.0000  % 2nd seed
  1.0341    1.9659    1.0000
```

As expected, in the Direct Monte Carlo stage, all samples are accepted and unique. The two samples with the worst outputs are marked above. These will be the seeds for the next subset. Going to the second subset we see:

```plaintext
>> [subsets(2).Samples, subsets(2).Outputs, subsets(2).Accepted]
ans =
   1.8471    1.1529    1.0000  % 1st seed, 1st chain item
   1.4049    1.5951    0     % 2nd seed, 1st chain item, became seed
   1.9694    1.0306    1.0000  % 1st seed, 2nd chain item
   1.4049    1.5951    0     % 2nd seed, 2nd chain item, became 1st
   1.9694    1.0306    0     % 1st seed, 3rd chain item, became 2nd
   1.4049    1.5951    0     % 2nd seed, 3rd chain item, became 2nd
   2.1821    0.8179    1.0000  % 1st seed, 4th chain item
   1.4049    1.5951    0     % 2nd seed, 4th chain item, became 3rd
```
4.7 Post-process Subset Results to Study Sensitivities

Once you finished a run of subset simulation, you can use our report generator tool (that uses the official mlreportgen toolbox internally), to infer, which distributions are most sensitive to failure, i.e. which distributions dominate the probability estimator output. There are various ways to initialize the report generator, but we prefer the following:

```matlab
subsets = algorithm.subsetsim();
reportgen = tulrfsd.mcmc.ReportGenerator(subsets, algorithm);

% this produces a sorted table of the most sensitive distribution names
reportgen.ranking()
% this produces a figure with the likelihood and the posterior
reportgen.plot("x1")
% this produces a struct with sensitivity info
reportgen.info("x1")
% this produces a document that the user can edit
reportgen.document("report.docx")
```

Consider the shipped demo demo/02-two-dimensional/, where the failure probability depends only on the first distribution. In that case the report generator will produce the following Figures 4.1–4.2, one for the distribution called “sensitive” and one for the distribution “unused”:
In the first Figure 4.1 for distribution “sensitive”, you can see that the green and blue histograms do not align with the sample distribution (line plots), where the black dashed line is the model for the sample distribution PDF $f(\cdot)$ and the red highlight line is the approximation from the actual sampled distribution. The green histogram in the left subfigure represents the “likelihood” of the distribution for failure $P(\cdot \in j | F)$, i.e. the likelihood that the distribution was sampled in the given intervals within the failure domain $F$. The right subplot shows the “posterior” of the distribution $P(F | \cdot \in j)$, i.e. how likely the failure $F$ was when the distribution hit the given intervals. A shift in both histograms like shown is a good indicator that the distribution influences the failure probability significantly. The color grading from solid to white is an indicator whether there were sufficient samples in the according intervals, so that the meaningfulness of the histograms is given. (Almost) white bars should be trusted less.

**Figure 4.1**: Sensitivity plot from the report generator for the distribution “sensitive” from demo “02-two-dimensional”.

Institute of Flight System Dynamics, Technical University of Munich
In the second Figure 4.2 for distribution “unused”, you can see how an un-influential distribution looks like. The blue horizontal bar in the right figure is the overall failure Probability $P_F$. The histogram of the posterior follows this like a uniform distribution. The chatter at the outsides of the histogram is simply due to statistical variation. The likelihood in the left histogram follows the sample distribution almost perfectly, showing no shift between distributions. These two histograms together are a good indicator that this distribution does not influence the failure probability, which by design we know it does not.
5 Advanced Topics

The previous chapter should have given you a firm understanding how to configure and run the subset simulation algorithm. This chapter will dive deeper into solutions for issues you might encounter on the way.

5.1 Achieve Good Acceptance Ratios

According to [7], the optimal acceptance ratio should be between 40% and 60%. This will achieve the lowest coefficient of variation on the failure estimator. From our personal experience, such acceptance ratios can be achieved if the proposal distribution is half as wide as the sample distribution.

The acceptance ratio is influenced primarily by the choice of the sampling method. See section Tune Parameter Sampling to see how they define the acceptance criterion. While there are some differences, one thing they have in common: samples that do not progress towards the failure domain are rejected. In other words, only samples that produce a failure margin less than the current subset threshold are accepted. This implies for 'infinity' sampling, where the failure margin is the only criterion, that in the case of a monotonous problem 'ModelHandle' and a symmetric proposal distribution, there should be about 50% acceptance.

Metropolis sampling has a lower acceptance ratio, because the acceptance term also rejects samples that are too far away from the seeds. This is especially detrimental for the vector-wise acceptance in 'metropolis' and 'metropolishastings' in a high-dimensional parameter space. Typically you will see the acceptance ratio go towards zero for more than 10 parameters already. Element-wise acceptance in 'metropolishastingsewise' helps restore larger acceptance ratios.

The rejection of samples too far away from their seeds can be counteracted by reducing the width (standard deviation) of the proposal distribution. For example:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation(%
    % [...]
    'SampleDistribution', {{[0,0,0], [2,4,8]} % mean and std
    'ProposalDistribution', {{[0,0,0], [1,2,4]} % mean and std
    })
```
However, reducing the proposal distribution width too much will again reduce the acceptance ratio.
You can investigate the overall acceptance ratios after simulating like so:

```matlab
subsets = algorithm.subsetsim();
ratioEachSubset = [subsets.AcceptanceRatio];
```

### 5.2 Achieve Precise Results

You can get an estimate for the accuracy of the failure probability estimator with the `incident-bounds()` function after running subset simulation:

```matlab
[P_f, cause, subsets] = algorithm.incidentprobability();
[P_f_bounds, c_o_v_bounds] = algorithm.incidentbounds(subsets);
```

This will produce the lower and upper c.o.v. bounds as described in a section in the previous chapter. According to [7], the bounds will become the smallest for acceptance ratios between 40% and 60%. The 'infinity' sampling method is a solid choice.

### 5.3 Accelerate Slow Models

The 'ModelHandle' property that you specify may point to a function that evaluates a costly simulation model. This function will be evaluated for all the parameter samples – at most \( m_{subsets} \times n_{samples} \), but a few less times because rejected samples are never evaluated. You have the following options to speed up the evaluation of your function:

1) Sometimes you can vectorize your function, see Apply Parameters to the Model
2) You can also parallelize computations, see above link.
3) You can replace function evaluations by training faster surrogate models.

In case of option 3), you can specify a property for 'SurrogateModel' on the algorithm:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    % [...] 
    'SurrogateModel', 'MLS'
});
```
In the code example above, the algorithm will use only a fraction of the samples each subset to train an accurate “Moving Least Squares” (MLS) surrogate model, then use that model to evaluate the majority of samples each subset. The algorithm will use an iterative refinement strategy that is detailed in the chapter on Surrogate Models.

### 5.4 Resolve Parameter Dependencies with Copulas

If your model parameters are not independent, our sampling methods will produce sample vectors that are not distributed according to their true occurrence. In that case, you can model their dependencies with our tulrfsd.mcmc.CopulaDistribution object. Read more in the dedicated section. In summary, you need to extract their dependency structure with the vinecopulib toolbox to produce the copula.json file, then wrap the container around independent marginal distributions:

```python
import tulrfsd.mcmc.*
marginals = ElementwiseDistribution([marginal1, marginal2, ..])
dependent = CopulaDistribution(marginals, "copula.json")
algorith = SubsetSimulation({"SampleDistribution", dependent})
```

The concept of using copulas to model parameter dependencies is described in detail by Höhndorf et. al [8].

### 5.5 Debugging Model Handle Outputs

Previously, we always mentioned that your function handle for 'ModelHandle' must receive a row vector for the parameter sample and returns a scalar metric that indicates the failure margin. In fact, that is the core concept, but there is also an additional feature. The following is also a valid model handle:

```python
function output = myModelHandle(sample)
    % [ margin = ... ]
    output = [margin, more, debug, values, as, row, vector];
end
```
The first column of your output must still return the failure margin. Any other appended values will be ignored by the algorithm and still captured in the SubsetResult objects:

```matlab
>> subsets = algorithm.subsetsim();
>> subsets(1)
an
ans =
    SubsetResult with properties:
        Samples: [100x1 double]
        Outputs: [100x7 double]  % <-- 7 columns here
        % [...]
```

### 5.6 Dealing with Errors in the Model Handle

Sometimes there will be errors when evaluating your function handle. Subset simulation will always aim for extreme parameter samples, so this case might happen sooner or later. However, if your function handle errors, so does subset simulation:

```matlab
>> algorithm = tulrfsd.mcmc.SubsetSimulation({'Model', @thisWillChrash})
>> algorithm.subsetsim()
Error using thisWillChrash
Something went horribly bad.

Error in tulrfsd.mcmc.SubsetSimulation/evaluate
Error in tulrfsd.mcmc.SubsetSimulation/sample4toplevel
Error in tulrfsd.mcmc.SubsetSimulation/subsetsim
```

If you do not want subset simulation to stop there, you must ...

- prevent your error from leaving your model handle, and ...
- produce a reasonable failure margin.

If your errors are more sporadic and are not likely caused systematically, you should consider returning a very large value as failure margin, even `inf`. Subset simulation chooses the worst samples as seeds for the next subset and determines acceptance based on the previous subset threshold. With a large margin, guaranteed to be above the previous threshold, sporadic error samples can be automatically rejected.
Rejected samples are not collected in the SubsetResult output containers. If you want to observe errors as they happen, register an event handler to 'OnEvaluateSample':

```plaintext
algorithm.addlistener('OnEvaluateSample', @logError);
```

---

### 5.7 Saving Results in an Accessible Way

Saving result data in MATLAB is easy:

```plaintext
subsets = algorithm.subsetsim()
save('solution.mat', 'subsets')
```

However, this gives you one problem: when you do not have the source code for the toolbox available, you cannot unpack the MAT file! “Why would I not have the code?” you might ask. Well, imagine you are exporting results and give them to a colleague to review, and they don’t have the toolbox installed.

To make it easy for you to store and load data in a way that makes data transparent to everyone, we implemented the `struct()` method with all of our classes:

```plaintext
subsets = algorithm.subsetsim()
subsets = struct(subsets)
save('solution.mat', 'subsets')
```

Now, everyone with MATLAB alone can read your subset simulation results. If you want to load the struct data back into an object array, you do the inverse and pass the struct to the class constructor:

```plaintext
subsets = load('solution.mat', 'subsets').subsets
subsets = tulrfsd.mcmc.SubsetResult(subsets)
```

You can do the same with the subset simulation algorithm itself too:

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation({})
algorithm = struct(algorithm)
save('problem.mat', 'algorithm')
```
And to load:

```matlab
algorithm = load('solution.mat', 'algorithm').algorithm
algorithm = tulrfsd.mcmc.SubsetSimulation(algorithm)
```

### 5.8 Stop and Restart the Algorithm

You can request to stop subset simulation from any of your event listeners. See Event Handlers on how to define custom listeners. To request to stop subset simulation (e.g. to define a dynamic cut-off threshold criterion):

```matlab
algo.addlistener("OnSubsetComplete", @onSubsetComplete)
function onSubsetComplete(algo, data)
    if % my condition triggers
        algo.stop()
    end
end
```

When you stop subset simulation via `stop()`, then the algorithm will error right after your event handlers finish. You will see the error identifier `tulrfsd:mcmc:StopRequested` in the console window. While you can write a `try/catch` to deal with the case that subset simulation is incomplete (e.g. skip any post processing), this will not give you access to the intermediate algorithm results.

To obtain access to the intermediate subset results and algorithm state, you need to register an event handler on `OnAlgorithmError`:

```matlab
algo.addlistener("OnAlgorithmError", @onAlgorithmError)
function onAlgorithmError(algo, data)
    subsets = data.Subsets;
    save("intermediate.mat", "subsets");
end
```

### 5.9 Save Algorithm State If Error Occurs

To obtain access to the intermediate subset results and algorithm state when the algorithm errors (e.g. due to model simulation error), register an event handler to `OnAlgorithmError`. You may
read Event Handlers to find out more details how the listeners work.

```matlab
algo.addlistener("OnAlgorithmError", @onAlgorithmError)

function onAlgorithmError(algo, data)
    subsets = data.Subsets;
    save("intermediate.mat", "subsets");
end
```

Errors that make it into `OnAlgorithmError` are non-recoverable for subset simulation. However, if you store the intermediate subsets result containers, you will be able to warm-start subset simulation from the last good subset:

```matlab
finishedSubsets = algo.subsetsim(previousSubsets)
```

### 5.10 Learning from Tests and Demos

If you would like to see more examples how to use the algorithm and what results to expect, we include with this toolbox:

- our unit tests in the `test/` folder
- our examples in the `demo/` folder

Go ahead and have a look at the assumptions and documentation there. They might give you just the extra understanding you need.
6 Parameter Distributions

The parameter distribution containers we provide all follow the same interface:

```matlab
classdef (Abstract) Distribution
    methods (Abstract)
        % draws random samples: \( x(N,M), m(N,M) \)
        x = rnd(self, m)
        % multivariate pdf: \( d(N,1), x(N,M), m(1,M) \) optional
        d = mvpdf(self, x, m)
        % marginal pdf: \( d(N,M), x(N,M), m(1,M) \) optional
        d = pdf(self, x, m)
        % marginal cdf: \( d(N,M), x(N,M) \)
        d = cdf(self, x)
        % marginal inverse cdf: \( x(N,M), d(N,M) \)
        x = icdf(self, d)
        % first statistical moment: \( m(1,M) \)
        m = mean(self)
        % second statistical moment: \( s(1,M) \)
        s = std(self)
        % serialize for saving to file
        d = struct(self)
    end
end
```

There are the following implementations for this interface that are explained in the following sections:

- `tulrfsd.mcmc.NormalDistribution`
- `tulrfsd.mcmc.MultivariateNormalDistribution`
- `tulrfsd.mcmc.ElementwiseDistribution`
- `tulrfsd.mcmc.CopulaDistribution`
- `tulrfsd.mcmc.ManualDistribution`
- `YourOwnSubclass`
6.1 Independent Normal Distribution

The `tulrfsd.mcmc.NormalDistribution` container combines a set of independent normal distributions. They are specified with a row vector for their mean, and a row vector for their standard deviation, or sigma:

```plaintext
dist = tulrfsd.mcmc.NormalDistribution([0, 0, 0], [1, 2, 3])
```

Both vectors are validated to have the same length. Also the standard deviation must be an array of finite positive values.

You will use this distribution container probably most often, and also for setting up infinity sampling, where it is even required. To read more about the example below, have a look at the previous section on infinity sampling.

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation(
    'ModelHandle', @(x) 5 - x,
    'SamplingMethod', 'infinity',
    'ProposalDistribution', [0, 0.5]
)
```

Internally, the container is using the built-in methods `normrnd`, `normcdf`, `normpdf`, and `norminv` that provide good numerical performance.

6.2 Multivariate Normal Distribution

The multivariate counterpart `tulrfsd.mcmc.MultivariateNormalDistribution` adds a feature for optional correlation of samples on top. The correlation is defined as the off-diagonal entries to the square covariance matrix $\Sigma$:

```plaintext
dist = tulrfsd.mcmc.MultivariateNormalDistribution([0, 0, 0], [
    1.0, 0.0, 0.5  % one off-diagonal value here ..
    0.0, 4.0, 0.0
    0.5, 0.0, 9.0  % .. and here
])
```
Since the covariance matrix contains variances instead of standard deviations, you must square the diagonal entries yourself!

6.3 Element-wise Distribution

If you are working with independent marginal distributions that are anything else than normal, e.g. uniform, then you should use tulrfsd.mcmc.ElementwiseDistribution instead:

```matlab
dist = tulrfsd.mcmc.ElementwiseDistribution([marginal1, marginal2, ..., marginalN])
```

Here `marginal*` are objects of the type `prob.ProbabilityDistribution` that is implemented in the MATLAB “Statistics and Machine Learning Toolbox”. You can instantiate them using the `makedist()` function:

```matlab
marginal1 = makedist('normal', 0, 1)
marginal2 = makedist('uniform', 0, 360)
% [...]
```

This makes `tulrfsd.mcmc.ElementwiseDistribution` merely a lean wrapper around the array of the marginals. This distribution container can treat the marginals only as independent. You must model the dependencies separately, preferrably using a copula distribution wrapper.

6.4 Copula Distribution

To model parameter dependencies with copulas, we are interfacing with the open source module vinecopulib (available here: https://github.com/vinecopulib/vinecopulib). You can find more information about how Copulas work on the following page of the Mathematics Department at our university: https://www.groups.ma.tum.de/en/statistics/research/vine-copula-models/.
With the vinecopulib library and its toolbox implementation for e.g. R, you can estimate the copula model for given sample data. The library will also allow you to export the copula model in JSON format, for example:

```json
{
    "pair copulas": {
        "tree0": {
            "pc0": {
                "family": "Gaussian",
                "rotation": "0",
                "parameters": [ [ "0.35251536775169223" ] ],
                "var_types": [ "c", "c" ],
                "nobs_": "1000",
                "loglik": "65.024534509551131"
            }
        }
    },
    "structure": {
        "array": [ [ "2" ] ],
        "order": [ "1", "2" ]
    },
    "var_types": [ "c", "c" ],
    "nobs_": "1000",
    "threshold": "0",
    "loglik": "65.024534509551131"
}
```

With another `turlfsd.mcmc.Distribution` object as the marginals and the JSON configuration file, you can instantiate our copula distribution object:

```python
marginals = turlfsd.mcmc.ElementwiseDistribution([marginal1, marginal2, ..])
dependent = turlfsd.mcmc.CopulaDistribution(marginals, "copula.json")
```

This will respect the parameter dependencies within subset simulation and give you a more accurate failure probability.

### 6.5 Writing Your Own

There will be cases where none of the solutions mentioned before will work for you. In that case, you are free to implement your custom distribution container according to the interface defined in `turlfsd.mcmc.Distribution`. It could look like this:
classdef MyDist < tulrfsd.mcmc.Distribution
    methods
      function x = rnd(self, m)
        % [...]
      end

      function d = mvpdf(self, x, m)
        % [...]
      end

      function d = pdf(self, x, m)
        % [...]
      end

      function d = cdf(self, x)
        % [...]
      end

      function x = icdf(self, d)
        % [...]
      end

      function m = mean(self)
        % [...]
      end
    end
end

Of course you can derive your subclass from any other of our distribution containers. If you simply like to create a subclass for organizational purposes, you can override the constructor only:

classdef AtmosphericDisturbances < tulrfsd.mcmc.ElementwiseDistribution
    methods
      function self = AtmosphericDisturbances()
        uncertainties = [
          makedist('normal', 0, 10) % the wind speed in m/s
          makedist('uniform', 0, 360) % the wind direction
        ];
        self@tulrfsd.mcmc.ElementwiseDistribution(uncertainties);
      end
    end
end
end
7 Surrogate Models

As mentioned in Accelerate Slow Models, you may specify surrogate models that replace the majority of 'ModelHandle' function evaluations by specifying the model while constructing the algorithm:

```matlab
algorithm = tulrfsd.mcmc.SubsetSimulation({
    % [...] 
    'SurrogateModel', 'MLS' % option 1) 
    'SurrogateModel', MyCustomModel() % option 2)
})
```

You can specify models either via shorthand string (see options below), the full class name of the model, or a preconfigured handle of a `tulrfsd.mcmc.SurrogateModel` object. If you want to use our predefined models, use the shorthands. For custom models see the section about the model interface.

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Surrogate Model Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;none&quot;</td>
<td>acceleration feature is disabled</td>
</tr>
<tr>
<td>&quot;LS&quot;</td>
<td><code>tulrfsd.mcmc.model.LeastSquares</code></td>
</tr>
<tr>
<td>&quot;MLS&quot;</td>
<td><code>tulrfsd.mcmc.model.MovingLeastSquares</code></td>
</tr>
<tr>
<td>&quot;MLSK&quot;</td>
<td><code>tulrfsd.mcmc.model.MovingLeastSquaresKernel</code></td>
</tr>
<tr>
<td>&quot;GLM&quot;</td>
<td><code>tulrfsd.mcmc.model.GeneralizedLinearModel</code></td>
</tr>
<tr>
<td>&quot;interp&quot;</td>
<td><code>tulrfsd.mcmc.model.ScatteredInterpolant</code></td>
</tr>
</tbody>
</table>

After enabling the surrogate model on the algorithm, you may tune any parameters of the model through reference:

```matlab
algorithm.SurrogateModel.MovingWindowCutoffThreshold = 1e-2;
```
7.1 The Acceleration Strategy

Despite the high efficiency of subset simulation at estimating rare failure probabilities, thousands of model evaluations are still required to achieve sufficient accuracy. This becomes impractical when the model evaluation is computationally expensive. To further enhance the efficiency of subset simulation, the surrogate model is combined with subset simulation. Specifically, a surrogate model is constructed using a comparatively small set of training samples, and then many calls to the true model are replaced by those to the surrogate model. As the samples approach the failure domain progressively, more training samples are chosen to refine the surrogate model [9].

7.2 Model Details

The following subsections describe the different models that we ship with the toolbox.

7.2.1 LeastSquares (LS)

The `tulrfsd.mcmc.model.LeastSquares` model, also known as polynomial response surface, exploits second-order polynomial expansions to fit the relationship between uncertain inputs and model outputs. The expansion coefficients are estimated by ordinary least-squares regression. This surrogate model may be imprecise for highly nonlinear problems. You can set the following parameters on the model, the values shown here are the defaults:

```matlab
model.NumTrainingPointsPerParam = 2;
model.MatrixRegularizationFactor = 1e-6;
```

With `NumTrainingPointsPerParam` you specify how many training points will be used to solve the fitting problem. The number of expansion coefficients to be estimated is \((N + 1)(N + 2)/2\) for \(N\)-dimensional case. This property increases the number of initial training points to `NumTrainingPointsPerParam` times of the number of expansion coefficients. If `NumberOfSamples` is less than the expected number of training points, then all the available samples will be taken as initial training points.

Ordinary least-squares regression requires the inversion of moment matrix. With `MatrixRegularizationFactor` you specify equivalent elements that are added to the diagonals of moment matrix, thus alleviate the problem of a near-singular moment matrix. The `MatrixRegularizationFactor` value should be a very small positive number.
Chapter 7 Subset Simulation Toolbox

7.2.2 MovingLeastSquares (MLS)

The `tulrfsd.mcmc.model.MovingLeastSquares` model builds a local response surface which estimates the sample response by only considering the samples in its support domain. The expansion coefficients are determined by a weighted least-squares method. The weights decrease within the support domain and vanish outside. You can set the following properties on the model object:

```matlab
model.NumTrainingPointsPerParam = 2;
model.MatrixRegularizationFactor = 1e-6;
model.MinMovingWindowPointsIncreaseFactor = 1.5;
model.MovingWindowCutoffThreshold = 1e-3;
```

The properties `NumTrainingPointsPerParam` and `MatrixRegularizationFactor` are the same with those in the `tulrfsd.mcmc.model.LeastSquares` model.

With `MinMovingWindowPointsIncreaseFactor` you specify the size of support domain. The number of points in the support domain is `MinMovingWindowPointsIncreaseFactor` times of the number of expansion coefficients. The `MinMovingWindowPointsIncreaseFactor` value should be smaller than `NumTrainingPointsPerParam`, otherwise the initial training points cannot satisfy the size requirement of support domain. On the other hand, its value should be greater than 1 to achieve sufficient regression accuracy.

The property `MovingWindowCutoffThreshold` specifies the threshold of weights. If a sample is assigned a weight below this threshold, it barely has an impact on the weighted regression result and it is thus dropped from the support domain.

7.2.3 MovingLeastSquaresKernel (MLSK)

The `tulrfsd.mcmc.model.MovingLeastSquaresKernel` model is almost identical with the `tulrfsd.mcmc.model.MovingLeastSquares` model. The only difference is the way to solve the regression problem. The `tulrfsd.mcmc.model.MovingLeastSquaresKernel` model applies kernel ridge regression, which does not need to calculate the expansion terms. The drawback is that the dimension of the inversion problem is increased by the factor `MinMovingWindowPointsIncreaseFactor`. The adjustable parameters and their defaults are the same with those in the `tulrfsd.mcmc.model.MovingLeastSquares` model.
### 7.2.4 ScatteredInterpolant (interp)

This feature requires the “MATLAB Statistics and Machine Learning Toolbox” [10].

The `tulrfsd.mcmc.model.ScatteredInterpolant` model is a linear interpolation model for 2D and 3D problems (max 3 parameters in each sample). It works by meshing the sample space over the training points and interpolating query points on the triangles / tetrahedrons. Larger-dimensional problems will yield an error on training. You can set the following parameters on the model, the values shown here are also the defaults:

```matlab
model.NumTrainingPointsPerParam = 10;
model.InterpolationMethod = "linear"; % "nearest", "cubic", ...
model.ExtrapolationMethod = "nearest";
```

With `NumTrainingPointsPerParam` you specify how many points the model will require for accurate training. Subset simulation will try to honor that as an upper bound. If your algorithm `NumberOfSamples` is less than this property values times the number of parameter dimensions, then it will train the model will less points. Specify a lower value if this model is too slow, and a larger value if you get inaccurate results.

`InterpolationMethod` and `ExtrapolationMethod` are properties from the official `scatteredInterpolant` function from the “MATLAB Statistics and Machine Learning Toolbox” [10].

### 7.2.5 GeneralizedLinearModel (GLM)

This feature requires the “MATLAB Statistics and Machine Learning Toolbox” [10].

The `tulrfsd.mcmc.model.GeneralizedLinearModel` model is a wrapper around the method `fitglm()` from the “MATLAB Statistics and Machine Learning Toolbox” [10]. You can set the following properties on the model object:
model.NumTrainingPointsPerParam = 2;
model.Modelspec = "quadratic";
model.Args = {​};
model.UseDropout = true;

The NumTrainingPointsPerParam adjusts the requested number of training points, as it scales with the dimensionality of the problem. Specify a value larger than 1 for a more accurate result at increased execution time during training.

The Modelspec value is passed through to fitglm(). You can specify "quadratic", "interactions", "linear", as well a formula for the model terms, e.g. "sin(x)". Quadratic terms include all element-wise squares ($x_i^2$, $\forall i$), as well as interaction terms ($x_i x_j$, $i \neq j$). Interaction terms include linear terms ($x_i$). Linear terms also always include the constant offset.

You can pass more arguments to fitglm() during training with the cell array of Args, which is simply passed through for your convenience.

The model also comes with a dropout feature (property UseDropout), which removes old training points to keep the fitting problem small. After all, the model must only be accurate for the current subset.

7.3 Writing a Custom Model

If you want to define your own model, you must create a subclass for the interface tutorial.mcmc.SurrogateModel, which forces you to implement the following member functions:

```python
n = scale(self, samples);
train(self, samples, outputs);
[outputs, errors] = evaluate(self, samples);
reset(self);
```

scale() receives the sample matrix for the current subset and must estimate, **how many training samples** the model needs to train it accurately. In other words, you must determine the **scale** of the parameterization problem. Typically this is determined by the number of sample dimensions size(samples, 2) and limited to a fraction of total samples size(samples, 1) * 0.1.

train() will receive a matrix of samples (each row is a sample) and a column vector of model responses. You must parametrize your model here and store the training state within your model instance in your custom class properties. Also, you should already prepare a model for the maximum expected evaluation residuals.
evaluate() must run the trained model function against the given samples and return the model outputs as the first return argument. The second expected return argument is the \textbf{expected residual} for the model. You should implement a separate model for the propagation of the error through the model for that.

reset() should \textbf{un-train} the model and erase any custom training state on the model instance.
8 Instrumentation

“In the context of computer programming, instrumentation refers to the measure of a product’s performance, to diagnose errors, and to write trace information.” [11]

We give you a list of pre-defined and customizable options to trace our algorithm’s execution.

8.1 Built-in Options

Our pre-defined instrumentation options all revolve around printing debug information to the command line. The responsible options are tied to the algorithm instance and can be set as follows:

```plaintext
algorithm = tulrfsd.mcmc.SubsetSimulation({
    'ModelHandle', @(x) 3 - x
    'NumberOfSamples', 50
    'Instrumentation', 'progress'
});
algorithm.subsetsim();
```

The default option is 'progress'.

Now the algorithm won't be silent but print a single dot for each evaluation of the model handle and a summary after each subset:

```
09-Mar 16:52 level 0, N = 50, threshold 1.92, acc. 100%, P_f < 1
09-Mar 16:52 level 1, N = 50, threshold 0.886, acc. 39%, P_f < 0.1
```

Institute of Flight System Dynamics, Technical University of Munich
You can be more 'minimal':

<table>
<thead>
<tr>
<th>Date</th>
<th>Level</th>
<th>N</th>
<th>Threshold</th>
<th>Acceptance</th>
<th>P_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-Mar 16:52</td>
<td>level 0</td>
<td>50</td>
<td>1.92</td>
<td>100%</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>09-Mar 16:52</td>
<td>level 1</td>
<td>50</td>
<td>0.886</td>
<td>39%</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>09-Mar 16:52</td>
<td>level 2</td>
<td>50</td>
<td>0.263</td>
<td>27%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>09-Mar 16:52</td>
<td>level 3</td>
<td>50</td>
<td>-0.373</td>
<td>17%</td>
<td>&lt; 0.001</td>
</tr>
</tbody>
</table>

In these summary lines you can see for each subset how the subset threshold moves down towards zero. The problem is solved when the threshold becomes zero or negative. Also you can see the acceptance ratio for the MCMC chains. The first subset is Direct Monte Carlo and will always have 100% acceptance. Lower subsets should have a ratio between 40% and 60% to produce results with minimal c.o.v. [7]. Finally, the line is terminated with the current maximum failure probability, should the next subset contain all failure samples.

We can see from this trace that the acceptance ratio has dropped below 20% in the last subset. The reason is that in the default setup, the sample distribution is as wide as the proposal distribution. Making the proposal distribution more narrow will increase the acceptance ratio. Read more in this section.

Or more 'verbose':

```
{n = 57, accept = [0], output = [0.5001], sample = [0.5001]}
{n = 58, accept = [1], output = [0.051723], sample = [0.051723]}
{n = 59, accept = [0], output = [-2.4895], sample = [-2.4895]}
{n = 60, accept = [0], output = [0.20526], sample = [0.20526]}
{n = 61, accept = [1], output = [-0.11451], sample = [-0.11451]}
{n = 62, accept = [1], output = [-0.77877], sample = [-0.77877]}
```

In total, we have the following pre-defined options for you:
Besides using the algorithm property to set the instrumentation level, you can also use the environment variable MCMC_LOG_LEVEL:

```plaintext
MCMC_LOG_LEVEL=none matlab.exe -batch "runtests test/unit"
```

If you are running GitLab CI, you can define the variable in a variables dictionary, either globally or in the job itself:

```plaintext
variables:
  MCMC_LOG_LEVEL: summary

test-unit:
  extends: .test
  script: matlab.exe -batch "runtests test/unit"

test-perf:
  extends: .test
  variables:
    MCMC_LOG_LEVEL: none
    script: matlab.exe -batch "runtests test/perf"
```

### 8.2 Event Handlers

You can define your own event listeners and attach them to our algorithm. The order of events invoked can be seen in the following flowchart:
Figure 8.1: Events notified during subset simulation

To listen to any of these events, you must provide a function handle with two input parameters. The first argument will always be a reference to the subset simulation algorithm, and the second will be a container with event-specific information:

```plaintext
listener = @(algorithm, event) do_something_with_it(event);
algorithm.addlistener('OnSubsetComplete', listener);
```

Some of the events publish specific data. These are:

- 'OnProposedSample' will give you a tulrfsd.mcmc.event.SampleEvent
- 'OnEvaluateSample' will give you a tulrfsd.mcmc.event.SampleEvent
- 'OnSubsetComplete' will give you a tulrfsd.mcmc.event.SubsetEvent
- 'OnAlgorithmError' will give you a tulrfsd.mcmc.event.ErrorEvent

From the SampleEvent, you can get the following information:
event.Sample % is a \((N,M)\) double with all samples proposed or evaluated
event.Output % is a \((N,M)\) double, empty for 'OnProposedSample'
event.Accept % is a \((N,1)\) logical, which samples are accepted
event.Timing % is a \((1,1)\) double, how long the ModelHandle took

From the SubsetEvent, you can read:

- event.Subset % is a \((1,1)\) tulrfsd.mcmc.SubsetResult
- event.Number % is a \((1,1)\) double, 0 for DMC stage, 1 for first MCMC stage

From the ErrorEvent, you can read:

- event.Subsets % is a \((1,:)\) tulrfsd.mcmc.SubsetResult
- event.Exception % is a \((1,1)\) MException

The tulrfsd.mcmc.SubsetSimulation object is a handle, which allows you to adapt the algorithm properties from within the listener function, but you cannot modify any data from the event object.
9 API Reference

For the full documentation of the classes contained, consider using the help and doc commands on our class definitions.
10 Contributors

Contributors to this project are:

- Florian Schwaiger (Maintainer)
- Chinmaya Mishra
- Dalong Shi
- David Braun
- Lukas Höhndorf
- Xiaolong Wang
- Phillip Koppitz
- Chong Wang
References


