Advancements of the SUSA Analysis Tool (SUSABlue)

Weiterentwicklung des Analysewerkzeugs SUSA (SUSABlue)

GRS - 778

Gesellschaft für Anlagen-
und Reaktorsicherheit
(GRS) gGmbH

Advancements of the SUSA Analysis Tool (SUSABlue)

Weiterentwicklung des Analysewerkzeugs SUSA (SUSABlue)

Final Report

Simone Palazzo
Tanja Eraerds
Martina Kloos
Jörg Peschke
Josef Scheuer
Jan Soedingrekso

September 2024

Remark:

This report refers to the in-house research project RS1599 which has been funded by German Federal Ministry for the Environment, Climate Action, Nature Conservation and Nuclear Safety (BMUKN).

The work was conducted by Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH.

The authors are responsible for the content of the report.

GRS - 778
ISBN 978-3-910548-71-8

Keywords

Best Estimate Plus Uncertainty (BEPU) Analysis, Markov Chain Monte Carlo, Monte Carlo Simulation, Sensitivity Analysis, SUSA, Uncertainty Analysis

Contents

Zusammenfassung

Das Analysewerkzeug SUSA ist eine etablierte Software zur Unsicherheits- und Sensitivitätsanalyse, mit welcher Ungewissheiten in nuklearen Sicherheitsanalysen im Rahmen von Best Estimate Plus Uncertainty (BEPU)-Ansätzen mittels Monte-Carlo-Simulationen berücksichtigt werden können. In SUSA können Toleranzintervalle oder Sensitivitätsmaße der sicherheitsrelevanten Simulationsergebnisse berechnet werden, indem die unsicheren Eingangsparameter variiert und die entsprechenden Simulationen eines deterministischen Codes ausgeführt werden. Im Rahmen des Forschungs- und Entwicklungsvorhabens RS1599 wurden weitere Methoden entwickelt und in SUSA implementiert, um einen Schätzer für die Wahrscheinlichkeit eines seltenen Ereignisses zu bestimmen und damit die Analyse zur Quantifizierung des Einflusses von Parametern zu unterstützen, die im Parameterraum der Einflussfaktoren in einem sehr geringen Wahrscheinlichkeitsbereich geschätzt werden. Hierbei handelt es sich um die Importance Sampling (IS)-Methoden.

In SUSA wurden zwei Ansätze implementiert, um eine optimale Importance Sampling-Dichte zu schätzen: die Approximation einer multivariaten parametrischen Verteilung und die Kernel-Dichte-Schätzung. Voraussetzung für beide Methoden ist eine Stichprobe aus dem (in der Regel wenig wahrscheinlichen) Bereich im Parameterraum der Einflussfaktoren. Eine solche Stichprobe kann z. B. aus den Ergebnissen eines adaptiven Monte-Carlo-Simulationsverfahrens gewonnen werden. Ein weiterer Entwicklungsbereich bezieht sich auf Methoden für Zuverlässigkeitsanalysen. Das von der GRS entwickelte Programm RAMESU (*R*eliability *A*nalysis with *M*arkov Models *E*xtended by an Option for *S*ensitivity and *U*ncertainty Analysis) wurde in SUSA integriert und über den bestehenden Funktionsumfang zur Zuverlässigkeitsanalyse hinaus um weitere Funktionalitäten ergänzt.

Die Modularität von SUSA wurde nach der Richtlinie eines mehrschichtigen Ansatzes verbessert, wobei die folgenden Bereiche berücksichtigt wurden: (1) grundlegende Python- oder FORTRAN-Routinen, (2) Funktionen oder Klassen auf höherer Ebene, die dem Benutzer eine komfortable Schnittstelle bieten, (3) Jupyter Notebooks, die die Implementierung einer ganzen Analysekette zeigen und dem Benutzer die Möglichkeit geben, den Analysevorgang interaktiv nachzuvollziehen und (4) eine grafische Benutzeroberfläche (GUI), die den Benutzern die Möglichkeit gibt, die Fähigkeiten von SUSA zu nutzen, ohne dass diese die Programmiersprache Python beherrschen muss.

Abstract

The SUSA analysis tool is an established software for uncertainty and sensitivity analysis which can be used to consider uncertainties in nuclear safety analyses in the frame of Best Estimate Plus Uncertainty (BEPU) approaches using Monte Carlo simulations. Within SUSA, tolerance intervals or sensitivity measures of the safety relevant simulation results can be calculated by varying the uncertain input parameters and running the respective simulations of a deterministic code. Within the frame of the research and development project RS1599, further methods have been developed and implemented in SUSA to determine an estimator for the probability of a rare event and thus to support the analysis for quantifying the influence of parameters which are estimated in a very low probability range in the parameter space of the influencing factors. These refer to the Importance Sampling (IS) methods.

Two approaches have been implemented in SUSA to estimate an optimal importance sampling density: the approximation of a multivariate parametric distribution and the kernel density estimation. The requirement for both methods is a sample from the (usually low probability) range in the parameter space of the influencing factors. Such a sample can be obtained, for example, from the results of an adaptive Monte Carlo simulation procedure. Another area of development refers to methods for reliability analyses. The RAMESU program (*R*eliability *A*nalysis with *M*arkov Models *E*xtended by an Option for *S*ensitivity and *U*ncertainty Analysis) developed by GRS has been integrated into SUSA and supplemented by further functionalities in addition to the existing range of functions for reliability analysis.

The modularity of SUSA has been improved according to the guideline of a layered approach where the following aspects have been considered: (1) basic Python or FORTRAN routines, (2) higher-level functions or classes which provide a convenient interface to the user, (3) Jupyter notebooks which show the implementation of a whole analysis chain and give the user the ability to interactively understand the analysis procedure, and (4) a graphical user interface (GUI) which provides the opportunity to the users of employing the capabilities of SUSA without the need to be proficient in the Python programming language.

1 Introduction and Objectives

GRS has been developing and using the analysis tool SUSA (*S*oftware for *U*ncertainty and *S*ensitivity *A*nalyses) for over 30 years to quantify the uncertainty associated with a simulation result and to determine the main causes of this uncertainty. The methods provided are based on probability calculations, Monte Carlo (MC) simulations and statistical methods. Based on probability distributions for the input parameters of a simulation program that cannot be clearly defined (uncertain), SUSA can be used to play out possible value constellations for these parameters and thus start corresponding simulation runs. The simulation results obtained can then be statistically analysed with regard to safety significant issues. For example, SUSA can be used to calculate a tolerance interval that covers a high proportion (generally $\geq$ 95 %) of the possible values of a simulation result with high statistical certainty (generally $\geq$ 95 %). SUSA thus provides important support in proving the acceptance criteria in the context of so-called BEPU analyses.

In addition, the sensitivity analysis methods implemented in SUSA can be used to determine the (uncertain) input parameters having the most influence on the uncertainty associated with a simulation result. Due to the implementation of advanced selection procedures and machine learning methods, SUSA can now also be used to determine the probability of a critical result (e.g., exceeding numerical safety criteria, failure of safety systems, core damage, release of radionuclides) and the unfavourable constellations of input parameters associated with the critical result with a practicable computational effort. In addition, the limit range between favourable and unfavourable parameter constellations, which is susceptible to cliff-edge effects (see /IAE 16/, p. 37, footnote 20), i.e. the range in which the system behaviour can change abruptly due to small fluctuations in the input parameters, can also be determined. The newly implemented methods can be used, e.g. to determine indicators for the safe operation of nuclear power plants or other criteria for risk management in accident sequences. SUSA consists of various platform-independent modules (in FORTRAN or Python) that are called by a higher-level application program for the corresponding calculations. As the application program and therefore the functionality of SUSA was previously only available on MS Windows-based operating systems, a Python-based core connection and prototype applications based on Jupyter notebooks /KLU 16/ were developed for a platform-independent application of SUSA. This sustainable implementation strategy is ideal for validation tests of existing as well as newly developed methods and can already be used for the first SUSA applications.

SUSA is a comprehensive, flexible and user-friendly analysis tool for dealing with uncertainties in the application of computational models. In order to maintain this status, it is to be continuously developed in line with the current state of knowledge and provide classic and advanced methods for uncertainty and sensitivity analyses. New methods are implemented in cases were the available implemented methods cannot be used to analyse the influence of uncertain input parameters on the simulation result for applications, either for reasons of practicability or because the corresponding methodological requirements are not met. New methods are implemented in a platform-independent manner to allow for a future use on Unix/Linux based computing clusters. SUSA is also being developed with regard to user-friendly access to the methods provided and the associated visualisation strategies as well as quality assurance standards in software development.

The following objectives should be pursued in detail:

1. The available options for reliability analysis should be extended.

   The RAMESU program developed by GRS /PES 91/ has been integrated into SUSA and supplemented by further functionalities in addition to the existing range of functions for reliability analysis. The range of functions in RAMESU is already significantly more extensive than in freely available programs (e.g. in Python PyDTMC /BEL 20/), which generally only allow standard modelling such as Markov chains and Markov processes. In particular, the program offers the modelling of semi-Markov properties, the consideration of allowable outage times (permissible failure times) of certain component states as well as the possibility of limiting the exponential growth of the state space by means of a so-called bound state.

   In addition, methods already developed at GRS for estimating distributions for reliability parameters have been integrated into SUSA. This enables the user to carry out a reliability analysis with SUSA, taking epistemic uncertainties into account. The methods for estimating distributions can be used to quantify the uncertainties in the input data of a probabilistic safety analysis (PSA) in a more transparent and comprehensible way. By integrating these methods into the SUSA application environment, strategies for quantifying distributions for reliability parameters from the operating experience and expert knowledge are made available to a wider range of users and made applicable in a user-friendly manner through practical and methodological documentation.

2. The available methods for advanced Monte Carlo simulation with machine learning algorithms should be maintained, benchmarked and extended.

   In classic MC simulations, parameter constellations are (randomly) selected from a joint probability distribution and corresponding simulation runs are started. The required number of parameter constellations depends on the underlying question and the estimator that is to be calculated to answer the question. For example, to determine an estimator for the probability of a rare event, such a high number of parameter constellations must be selected that it is no longer practicable to carry out corresponding simulation runs with a complex simulation code. Compared to classic MC simulation, advanced MC simulation requires only a relatively small number of parameter constellations and corresponding simulation runs.

   Advanced selection methods are used, which are often combined with machine learning (ML) methods. Such an advanced selection method is the Importance Sampling (IS) method in which the original joint probability distribution of the parameters is replaced by a distribution that weights parameter constellations in a certain range higher and thus makes their selection more probable. The IS method has been implemented in this project. The implementation has been carried out in such a way that the method can be used both independently and based on the results of an adaptive MC simulation.

   Adaptive MC simulation uses advanced selection procedures in combination with ML methods. An iterative process is used to control the selection of parameters to a specific range of the input parameter space, which is characterised, for example, by the fact that the parameter constellations contained therein result in a critical simulation result. The ML methods are used to determine a simple, fast-running replacement model (metamodel) for the actual simulation code. These metamodels are used to quickly calculate the results for a large number of selected parameter constellations.

   In addition to the methods already implemented, it is planned to integrate further ML methods for determining metamodels in SUSA. These include the support vector method and shallow neural networks (NNs). The properties of these methods as well as their advantages and disadvantages have been documented in order to provide support in the selection of a suitable metamodelling method for a specific application. The calculation results from the application of the metamodel are used to evaluate the associated parameter constellations according to their usefulness (i.e. their possible affiliation to the parameter range of interest). A scoring function (learning function) is often used for this purpose. The parameters with the best scores are then selected as input for the actual simulation runs. The convergence criterion of the iterative process of an adaptive MC simulation depends on the metamodel used (ML method).

3. The underlying source code of SUSA should be maintained, structured and developed towards platform independence and maintainability.

   The sustainable development of SUSA's software architecture aims to make the comprehensive analysis tool available and executable in as many common environments as possible. In addition, the software is to be made even more modular in order to allow a simpler, continuous and flexible expansion of the range of methods, e.g. through the use of scientific Python libraries such as scikit-learn /PED 11/ or SciPy /VIR 20/. In addition, through targeted API (*A*pplication *P*rogramming *I*nterface) development and the necessary harmonisation, sufficient compatibility with external developments have been achieved to allow efficient comparative calculations, data exchange and collaboration between software packages.

   Other important goals are universal and easy handling as well as appropriate support for the user. In the project RS1559, the foundations were laid to ensure flexible and modern applicability of the functionalities in SUSA, regardless of the platform. The new developments in this project are built upon this basis in order to meet the aforementioned objectives in compliance with the GRS software guidelines /GRS 20/.

4. The feedback of experience resulting from the use of SUSA should be continuously analysed and implemented accordingly in order to further improve the specific workflow and the general quality of SUSA.

5. The existing user documentation and method guide should be maintained. In accordance with the GRS specifications for software development /GRS 20/, every SUSA user is provided with method (or program) and user documentation /KLO 23/ as well as installation documentation. The method documentation is an updated version of /KLO 21a/. Method and user documentation are continuously expanded in line with the newly implemented methods and functionalities. Furthermore, the method documentation is supplemented by exemplary and documented Jupyter notebooks so that the user can work through the scope of SUSA's methods interactively and inde-

pendently, using selected, simple examples. The installation documentation for the classic SUSA GUI should also be adapted.

2 Reliability Parameters

2.1 Integration of the Functionality of the Markov Program RAMESU

Within PSA, the reliability of technical systems is determined using fault tree analyses. These reach the limits of their modelling capabilities, e.g.,

- if there are dependencies of the stochastic component behaviour on the state of other components or on changing environmental conditions (e.g. increasing temperature),
- if the dynamic behaviour of physical variables (e.g., pressure or temperature) is to be modelled,
- if 'phased mission' processes or switching of components take place at specific times,
- if tolerable downtimes (allowable outage times, AOTs) exist in the system,
- if repairs are carried out at certain times and the repairs are successful or not with certain probabilities,
- if maintenance is carried out on components at certain times and the components are not available during this maintenance, etc.

Using the RAMESU program developed and implemented in SUSA, dynamic processes of technical systems can be modelled and analysed in the form of Markov and semi-Markov processes. This allows a more realistic modelling of the dependencies that often occur in system behaviour. By implementing the Markov program in SUSA, it is possible to carry out an uncertainty and sensitivity analysis with regard to the reliability (e.g. probability of failure) of the modelled system in a user-friendly and efficient way. Markov and semi-Markov processes can be applied in various areas of system modelling and can be used to calculate the reliability and availability of a system. Influences of common-cause failures (CCFs) can be included just as easily as the influence of different test, maintenance and repair strategies, dependencies of the system on physical variables (such as pressure, temperature, etc.) and system states as well as influences of human actions and time-dependent phenomena.

A particular advantage of the developed program is that semi-Markov properties of a system can also be modelled that are not defined by exponentially distributed transition rates, such as system switchovers that are carried out at certain times and with certain probabilities of success. This alone distinguishes it from many commercial programs for Markov analysis. Using the semi-Markov properties of the program, it is also possible to take physical variables (temperature, pressure, flow rate, etc.) into account in discretised form. This allows dependencies of the failure behaviour of process variables to be modelled and their dynamic behaviour to be analysed.

Section 2.1.1 describes the methodology for calculating the state probabilities. Section 2.1.2 contains a description of the user input. Section 2.1.3 lists various application examples. For reasons of better comprehensibility, the application examples are kept relatively simple.

2.1.1 RAMESU Methodology

The aim of analysing technical systems using Markov processes is to calculate probabilities for the occurrence of system states over time. In many applications, dependencies occur between the system states of subsequent points in time or subsequent action steps. In this situation, the temporal dependencies in the behaviour of the system components must be taken into account. This cannot be modelled in sufficient detail using the classic method for determining the reliability of technical systems (e.g. fault tree analysis) as temporal dependencies can only be taken into account to a very limited and simplified extent. In order to consider these dependencies over time more precisely, mathematical methods of stochastic processes, for example, can be used.

2.1.1.1 Modelling Markov Processes

A stochastic process is defined as a set X_t, $t \in T$ of random variables, where T describes a discrete or continuous parameter space, e.g. discrete time steps or a continuous time interval. The simplest dependency structure between time-dependent random variables is obtained if the Markov property applies. This states that the future of the process only depends on the state of the present and not on the states that the system has assumed in the past. Formally, this can be expressed by equation (2.1):

$$\begin{aligned} &P(X(t_n) = i_n \mid X(t_{n-1}) = i_{n-1}, X(t_{n-2}) = i_{n-2}, \dots, X(t_0) = i_0) = \\ &P(X(t_n) = i_n | X(t_{n-1}) = i_{n-1}) \end{aligned} \tag{2.1}$$

with $t_n > t_{n-1} > \dots t_1 > t_0$ and $n \geq 1$.

I.e., the probability of the state $X(t_n) = i_n$ at the time t_n is only dependent on the last assumed state $X(t_{n-1}) = i_{n-1}$ at the time t_{n-1} and not on the states assumed at previous times $t_{n-2}, \dots, t_0$ assumed states $i_{n-2}, \dots, i_0$.

In order to model a system using a Markov process, the possible states of the system must be defined. If the system to be analysed can assume the N different states $Z_1, \dots, Z_N$, hen the probability $P_j(t)$ for the system state Z_j, $j = 1, \dots, N$ at any point in time t is obtained by applying a Markov model.

In addition to the definition of the state space of the system, the calculation of the state probabilities $P_j(t)$ also requires that:

- the matrix of transition rates $R = (r_{ij})$ from state Z_i to state Z_j and
- the initial state $Z_0(t = 0)$ of the system at the time $t = 0$

must be specified.

The elements r_{ij} in the rate matrix R (see equation 2.2) indicate the rate at which the process transitions from state Z_i to state Z_j. Since a Markov process is defined by the fact that a transition from one state Z_i to another state Z_j only occurs after exponentially distributed dwell times, each transition rate describes the parameter of an exponential distribution.

The rate matrix has the form:

$$\begin{pmatrix} -\sum_{k=2}^{N} r_{1,k} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & -\sum_{k=1,k\neq 2}^{N} r_{2,k} & \cdots & r_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ r_{N,1} & r_{N,2} & \cdots & -\sum_{k=1,k\neq N}^{N} r_{N,k} \end{pmatrix} \tag{2.2}$$

First, the initial state $Z_0(t = 0)$ of a system at the time $t = 0$ and the probability of occurrence for the initial state are specified. With the RAMESU program, it is also possible to specify uncertainties regarding the probability of occurrence. This means that different values from an epistemic distribution are possible for the probability of occurrence. More details on the consideration of uncertainties are described in Section 2.1.3.8.

With the definitions of

- the states *Z_1, …, Z_n* of the system to be calculated,
- the matrix of transition rates *$R = (r_{ij})$* from state *Z_1* to state *Z_j* and
- the initial state *$Z_0(t = 0)$* of the system with the associated probability

the state probabilities *$P_j(t)$, $j = 1, …, N$* for any time *t* can be in principle be solved by the system of differential equations shown in equation 2.3:

$$\begin{pmatrix} \frac{dP_1(t)}{dt} \\ \frac{dP_2(t)}{dt} \\ \vdots \\ \frac{dP_N(t)}{dt} \end{pmatrix} = \begin{pmatrix} -\sum_{k=2}^{N} r_{1,k} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & -\sum_{k=1,k\neq 2}^{N} r_{2,k} & \cdots & r_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ r_{N,1} & r_{N,2} & \cdots & -\sum_{k=1,k\neq N}^{N} r_{N,k} \end{pmatrix} \cdot \begin{pmatrix} P_1(t) \\ P_2(t) \\ \vdots \\ P_N(t) \end{pmatrix} \qquad (2.3)$$

The state space of the model can become very large when modelling systems using Markov models. In this case, finding a solution using a differential equation system becomes very difficult, therefore a different solution method was implemented to determine the time-dependent state probabilities.

The matrix *P* of the transition probabilities must first be calculated from the rate matrix. This can be done using the following calculation:

$$P = \frac{R}{r_{max}} + I_N \qquad (2.4)$$

where *I_N* denotes the unit matrix and *r_{max}* is the maximum of the amount of the diagonal elements $-r_{i,i}$ of the rate matrix *R*. I.e.,

$$r_{max} = \max\left(\sum_{k=2}^{N} r_{1,k}\,,\ \sum_{k=1,k\neq 2}^{N} r_{2,k}\,,\ \dots,\ \sum_{k=1,k\neq N}^{N} r_{N,k}\right) \qquad (2.5)$$

In the following, the state probability at a point in time *t* is briefly denoted as *$\mu(t) = P_1(t)$, …, $P_N(t)$ with $T \geq 0$*. It is assumed that the state probability at a certain point in time $t_a < t$ has already been calculated, i.e. *$\mu(t_a)$* is known.

Equation (2.6) is used to calculate the vector of state probabilities at any time $t > t_a$:

$$\mu(t) = \exp(-(t-t_a)\cdot r_{max})\cdot \mu(t_a)\cdot \exp\big((t-t_a)\cdot r_{max}\cdot P\big) \qquad (2.6)$$

where $\tau = (t - t_a) \cdot r_{max}$, $\mu(t_a) \cdot exp\big((t - t_a) \cdot r_{max}\big)$ can be represented by the following Taylor series expansion around the development point τ:

$$\mu(t_a) \cdot \exp(\tau \cdot P) = \sum_{i=0}^{\infty} \mu(t_a) \cdot \frac{\tau^i}{i!} \cdot P^i \tag{2.7}$$

The series in equation 2.7 can be calculated by the following recursion:

$$\nu^{(0)} = \mu(t_a) \tag{2.8}$$

$$\nu^{(n)} = \frac{\tau}{n} \cdot \nu^{(n-1)} \cdot P, \quad n > 0.$$

The vector of state probabilities *μ(t)* at the time *t* is calculated from equation 2.9:

$$\mu(t) = exp(-(t - t_a) \cdot r_{max}) \cdot \sum_{i=0}^{n} \nu^{(i)} \tag{2.9}$$

2.1.1.2 Modelling Semi-Markov Processes

In addition to calculating the state probabilities of a system using a Markov process, the RAMESU program is characterised by the fact that certain semi-Markov properties of a system can also be modelled. Semi-Markov behaviour of a system exists when state transitions of the system do not occur after exponentially distributed dwell times as with Markov processes but instead occur at certain given times with the corresponding probabilities. In order to take the semi-Markov behaviour of a system into account, so-called singular matrices are used.

Let $S_1, \ldots, S_m$ be the singular matrices that have been generated to describe the semi-Markov behaviour of the system to be analysed. For each singular matrix S_i, $i = 1, \ldots, m,$ times are specified at which S_i is applied. A detailed description of how singular matrices are defined is given in Section 2.1.2.5.

Let $T_R = t_1, \ldots, t_n$ be the set of times at which the state probabilities are determined via the rate matrix *R* or via the matrix *P* of transition probabilities derived from it. The set of application times of the singular matrix S_i is given by $T_{Si} = t_{i,1}, \ldots, t_{i,ni}$, $i = 1, \ldots, m$. In order to cover all times at which calculations must be performed, the union is formed from these time sets, i.e.:

$$T = T_R \cup T_{Si} \cup \ldots \cup T_{Sm} \tag{2.10}$$

For each time $t \in T$ it is checked whether $t \in T_{Si}$ for $i = 1, \ldots, m$. If $t \in T_{Si}$, the state transitions that are carried out at the time t with probability p are determined from the information of the singular matrix S_i.

Let $\mu(t) = (q_0, \ldots, q_n)$ be the vector of state probabilities at the time t and let the transition from state Z_i to state Z_j with probability p be defined by the singular matrix S_i. Then, at the time t, the probabilities of the states Z_i and Z_j of the state vector are modified by the calculation:

$$q_{ij} = q_j + q_i \cdot p,$$
$$q_i = q_i \cdot (1 - p) \tag{2.11}$$

2.1.1.3 Modelling Fixed Transitions

In addition to modelling Markov and semi-Markov processes, RAMESU offers the ability to also include transitions which always happen instantaneously, given a certain condition.

2.1.2 RAMESU User Input

In the input data set for the reliability analysis of a technical system, the system to be analysed and the system behaviour must be specified by the user. Currently, the input has to be provided in the form of a Python script, in future this will be simplified to improve usability also non-coding users. SUSA includes tests for its different components, also for RAMESU calculations. These tests provide good starting points for writing new RAMESU input.

The input is created by calling the input constructor *System* for constructing a full system description:

```
input_1 = System(),
```

The following objects are then registered step by step in the input by passing the input object as an attribute to their constructors:

- Input of the system components and states that the respective components can assume (Section 2.1.2.1).

- Input of the initial state (Section 2.1.2.2).
- Input of the times at which the state probabilities of the system are calculated (Section 2.1.2.3).
- Input of the transitions of states with associated transition rates (Markov), taking dependencies into account. (Section 2.1.2.4).
- Specification of the semi-Markov behaviour of the system to be analysed. The transitions of the semi-Markov behaviour are recorded in certain singular matrices S_i, $i = 1, \ldots, m$ (Section 2.1.2.5).
- Specification of the fixed transitions, which happen whenever a certain condition is fulfilled with the probability *1* and instantaneously (Section 2.1.2.6).

2.1.2.1 Component Definition

Each component can be added to the system by calling the system function:

```
add_component(name, states=(), description=""),
```

where *name* is the name of the component and *states* should be a tuple (set in curved brackets) listing the different potential states of the component. The description is optional; here, a short description of the component can be included as a string.

A potential input line for a pump component with two states (on and failed) could look like this:

```
system.add_component('pump_1', states=(1 , 2), description ='1-on 2- failed
').
```

2.1.2.2 Definition of the Initial State

The initial state should be defined by calling the system function:

```
set_initial(name, states, prob),
```

where *name* should be the name attached to the initial state, *states* should be a tuple specifying the state of each component and *prob* should be the probability of the initial state. The j^{th} element of the vector $[i_1, i_2, \ldots, i_{nK}]$ indicates the state of the j^{th} component (*j = 1, …, n_K*). The given state of each component should be included in the possible states for this component as specified in the component definition.

Example: A system consists of two components *K1* and *K2*. Both components should be intact at the beginning with probability 1, whereby an intact state is defined by the co-dimension 0. The input for the initial state is then:

```
set_initial(name, states=(0,0), prob=1.0).
```

The probability of the initial state is not necessarily 1.0. Probability values < 1 can also be entered for the initial state. In this case, the existence of the initial state is regarded as aleatory.

2.1.2.3 Definition of Calculation Times

The calculation times should be defined by calling the following system function:

```
add_calc_times( calculation_times ),
```

where *calculation_times* can be a list of time points at which the state probability should be determined:

```
calculation_times = [1, 2, 3.50 ],
```

or a range of time points

```
calculation_times = range( start_time, end_time, time_step ),
```

where *start_time*, *end_time* and *time_step* can be float or integer numbers and the provided time points will range from *start_time* to *end_time - time_step* with intervals of size *time_step*. Several comma-separated time ranges can also be provided as *calculation_times*.

2.1.2.4 Definition of Markov Transitions

Markov transitions can be specified by calling the system function *MarkovTransition*:

```
add_markov( conditions, transitions, rate ),
```

and providing the following attributes:

- The conditions under which a Markov transition should be considered are to be defined as value comparison in a Python expression. This could for example be the following string "*pump_1 == 1*", which compares the value of the component with the name "*pump_1*" to 1. Different value comparisons can be combined, for example by using the logical “and or”-operators.
- The transitions which should be performed are to be entered as Python statement, initialising an existing component to another state. An example would be "*pump_1 = 2*" in which the component with the name "*pump_1*" is set to the state 2.
- The rate should be provided in a Python-readable number format.

The following example

```
system.add_markov( pump_1 == 1, pump_1 = 2, rate =1.e -3),
```

describes the Markov transition which can only take place if pump_1 is in state 1. In this case, there will be a transition of pump_1 to the state 2 at a rate of 1.0 E-03.

2.1.2.5 Semi-Markov Transitions

Semi-Markov transitions can be specified by calling the following system function:

```
Semi-MarkovTransition:
```

```
add_semi_markov(conditions, transitions, probability, transition_times ),
```

and providing the following attributes:

- Conditions and transitions can be provided as described above for Markov transitions.
- The transition probability should be provided in a Python-readable number format.
- The times at which a semi-Markov transition should be regarded. As in Section 2.1.2.3, the times can either be given in a Python list format or as the range between a start time and an end time.

2.1.2.6 Definition of Fixed Transitions

Fixed transitions can be specified by calling the following system function:

```
add_fixed ( conditions , transitions ),
```

and providing the conditions and transition attributes as specified above in Section 2.1.2.4.

2.1.2.7 Running RAMESU

The following example shows how RAMESU can be applied and the vector of state probabilities for all calculation times printed:

```
ramesu = RAMESU( input )
ramesu.print_system_states()
ramesu.print_state_prob()
```

the input has to be defined following the instructions in Section 2.1.2. A list of *TimeState* objects (objects which have time and state as attributes) can be accessed using the *p_result* attribute:

```
p_result = ramesu.p_result
```

For each *TimeState*, object time and state can be accessed as attributes.

2.1.2.8 Including Uncertainties

To study the effect of uncertain input parameters, these input parameters need to be defined in the input definition as variables. For example, if the probability of a semi-Markov transition should be defined as uncertain, the probability needs to be set as parameter:

```
 system.add_semi_markov("pump_1 == 2 and pump_2 == 0 and switch == 0",
                        "switch = 1",
                        probablity = prob_sm_1 ,
                        calc_times = sm_markov_times ).
```

In the example above, the probability is defined as variable *prob_sm_1*. The new SUSA sampling module described in Section 2.3 can be used to sample the uncertain input parameter. The sampled input parameter can in turn be employed to generate a set of RAMESU inputs that serve to generate RAMESU outputs.

2.1.3 RAMESU Examples

Within PSA, the reliability of technical systems is determined using fault tree analyses. These analyses reach the limits of their modelling capabilities, e.g.,

- if there are dependencies of the stochastic component behaviour on the state of other components or on changing environmental conditions (e.g. increasing temperature),
- if the dynamic behaviour of physical variables (e.g., pressure or temperature) is to be modelled,
- if ‘phased mission’ processes or switching of components take place at specific times,
- if tolerable downtimes (AOTs) exist in the system,
- if repairs are carried out at certain times and the repairs are successful or not with certain probabilities,
- if maintenance is carried out on components at certain times and the components are not available during this maintenance activities, etc.

The RAMESU program developed and implemented in SUSA allows dynamic processes of technical systems to be modelled and analysed by means of Markov and semi-Markov processes. This allows a more realistic modelling of the dependencies that often occur in system behaviour. By implementing the Markov program in SUSA, it is possible to carry out an uncertainty and sensitivity analysis with regard to the reliability (e.g. probability of failure) of the modelled system in a user-friendly and efficient manner.

Markov and semi-Markov processes can be applied in various areas of system modelling and can be used to calculate the reliability and availability of a system. Influences of CCFs can be included just as easily as the influence of different test, maintenance and repair strategies, dependencies of the system on physical variables (such as pressure, temperature, etc.) and system states as well as influences of human actions and time-dependent phenomena.

2.1.3.1 System 1: Two Redundant Components

System 1 consists of two redundant pumps that operate in ‘hot redundancy’ and have a failure rate of 0.001 each. The feed-in system is considered to be failed if both pumps have failed. Initially, both pumps are fully functional. The calculation times are calculated from 0 to 1000 h operational time in time steps of 50 h. The input file associated with this description is shown below:

```
def test_input_system1 ():

      r_input = RAMESUInput ()
      # -------------------------------------------------
      # COMPONENT - SECTION
      # -------------------------------------------------
      r_input.add_component (’pump_1 ’, (0, 1) , ’0 - ok 1 -failed ’)
      r_input.add_component (’pump_2 ’, (0, 1) , ’0 - ok 1 -failed ’)
      # ------------------------------------------------
      # Initial State section
      # ------------------------------------------------
      r_input.set_initial ((0 , 0) , 1.0)
      # ------------------------------------------------------
      # Calculation Times t :
      # ------------------------------------------------------
      r_input.add_calc_times ( range (0, 1001 , 1))
      # -------------------------------------------------------
      # TRANSITION SECTION
      # -------------------------------------------------------
      r_input.add_markov (’pump_1 == 0’, ’pump_1 = 1’, 1.e -3)
      r_input.add_markov (’pump_2 == 0’, ’pump_2 = 1’, 1.e -3)

     return r_input
```

The program can be executed using the following lines of code:

```
ramesu = RAMESU ( test_input_system1 ())
```

As the matrix operations in this program are carried out using sparse matrices, the reduction of matrix elements achieved by the sparse matrix method used here is specified as information:

- Number of elements in the original transition matrix: 16;
- Number of elements in the sparse matrix: 8;
- Sparse matrix reduction: 50 %.

As the number of system states for this small system is very small, system 1 is suitable for describing the output in more detail. The system states can be explicitly listed in the output, using the following command:

```
ramesu.print_system_states()
```

The resulting states are:

```
Number of System States = 4
Nbr          State of Components
0            [0 0]
1            [0 1]
2            [1 0]
3            [1 1]
```

The output of the system states of the generated state space is used to check whether the system states generated from the user input are correct. If system states occur that should not occur or expected system states were not created, the user input may need to be modified.

The system state number 0 is defined by the vector [0, 0]. The first element of the vector describes the state of the first component (i.e. pump 1) and the second element describes the state of the second component (i.e. pump 2). The system state [0, 0] thus expresses that both components are in the state 0 and are both in operation according to the definition in the input. In system state 1, component 1 is in operation (state 0), while component 2 has failed (state 1). This situation is expressed by the vector [0, 1].

In system state 2, component 1 has failed (state 1) and component 2 is in operation (state 0). In system state 3, both components have failed, which is expressed by the vector [1, 1].

As the number of states of system 1 is very small, the probabilities of all states can be output using the following command:

```
 ramesu.print_state_prob()
```

The resulting system probabilities are:

```
Time dependent Probabilities of System
               [0 0]         [0 1]           [1 0]         [1 1]
time
0            1.000000  0.000000  0.000000  0.000000
1            0.998002  0.000998  0.000998  0.000000
2            0.996008  0.001993  0.001993  0.000002
3            0.994018  0.002985  0.002985  0.000006
```

```
4             0.992032  0.003974  0.003974  0.000012
...             ...            ...             ...              ...
996         0.136422  0.232803  0.232803  0.396582
997         0.136150  0.232707  0.232707  0.397047
998         0.135878  0.232610  0.232610  0.397513
999         0.135606  0.232513  0.232513  0.397978
1000      0.135335  0.232416  0.232416  0.398443

[1001 rows x 4 columns ]
```

For larger systems where the state space (i.e. the number of system states) is very large, such a list would be too confusing. In this case, the user can specify which of the system states are to be printed. As a rule, these will be the critical system states, e.g. states in which the system has failed. Information about a special state can be gathered by calling the function *get_sys_state_info* and passing a list of interesting system states as input. An example is given in the following line of code which prints the time dependent information about a system in which pump 1 is failed, and pump 2 is working. Since only one state is gathered, no summation of state probabilities is required:

After 1000 h of operation, the system is in state [0, 0] with a probability of 0.1353, in state [0, 1] and in state [1, 0] with a probability of 0.2325 each and in state [1, 1] with a probability of 0.3995 that both components are failed.

If only one component (pump 1) was operated, the failure probability of the component after 1000 h of operation would be $0.632 = 1 - \exp(-0.001 \cdot 1000)$. Redundancy with two components therefore reduces the probability of failure of the system after 1000 h of operation by approx. 37 %.

2.1.3.2 System 2: Four Redundant Components

System 2 is used to analyse the effects of extending the two times redundant system 1 to a four times redundant system by adding two additional pumps. Pumps 3 and 4 have the same failure rates as pumps 1 and 2. The following changes must be made to the input data set for system 2 compared to system 1:

```
def test_input_system2 ():

   r_input = RAMESUInput ()
   # ----------------------------------------------
   # COMPONENT - SECTION
   # ----------------------------------------------
   r_input.add_component ('pump_1 ', (0, 1) , '0 - ok 1 -failed ')
   r_input.add_component ('pump_2 ', (0, 1) , '0 - ok 1 -failed ')
   r_input.add_component ('pump_3 ', (0, 1) , '0 - ok 1 -failed ')
   r_input.add_component ('pump_4 ', (0, 1) , '0 - ok 1 -failed ')
```

```
    # ----------------------------------------------
    # Calculation Times t :
    # -----------------------------------------------
    r_input.add_calc_times ( range (0, 1050 , 50))
    # ----------------------------------------------
    # Initial State section
    # ----------------------------------------------
    r_input.set_initial ((0 , 0, 0, 0) , 1.0)
    # --------------------------------------------
    # TRANSITION SECTION
    # --------------------------------------------
    r_input.add_markov ('pump_1 ==0 ', 'pump_1 =1 ', 1.e -3)
    r_input.add_markov ('pump_2 ==0 ', 'pump_2 =1 ', 1.e -3)
    r_input.add_markov ('pump_3 ==0 ', 'pump_3 =1 ', 1.e -3)
    r_input.add_markov ('pump_4 ==0 ', 'pump_4 =1 ', 1.e -3)

    return r_input
```

In a Markov analysis, the time-dependent probabilities are calculated for each state that the system can assume. This also determines, for example, the probabilities that only one, two or three components of system 2 have failed. The probability that in system 2 after 1.000 h operational time all four components have failed at the time t = 1000 h (state no. 15) is 0.159. Compared to system 1, this means a reduction in the system failure probability of approx. 60 %. Compared to a feed-in with only one pump, system 2 with four redundant pumps achieves an increase in system reliability of approx. 75 % after 1000 h of operation.

In a Markov analysis, the time-dependent probabilities are calculated for each state that the system can assume. The probability that all four components have failed at the time t = 1000 h is 0.159 (cf. Fig. 2.1). Compared to system 1, which features two redundant pumps, a reduction in system failure probability of approx. 60 % can be observed. Compared to a feed-in with only one pump, system 2 with four redundant pumps achieves an increase in system reliability of approx. 75 % after 1000 h of operation. The final probability that exactly three components have failed is p = 0.3712, as visible in Fig. 2.2. The final probability that exactly two components have failed in system 2 after 1000 h of operation is p = 0.324. The final probability that exactly one component is failed is p = 0.1259.

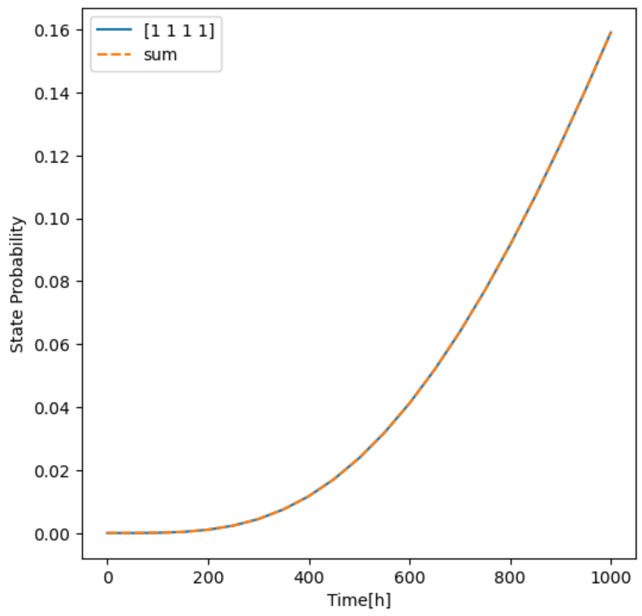


Fig. 2.1 Probability that all four components of system 2 are failed as a function of time

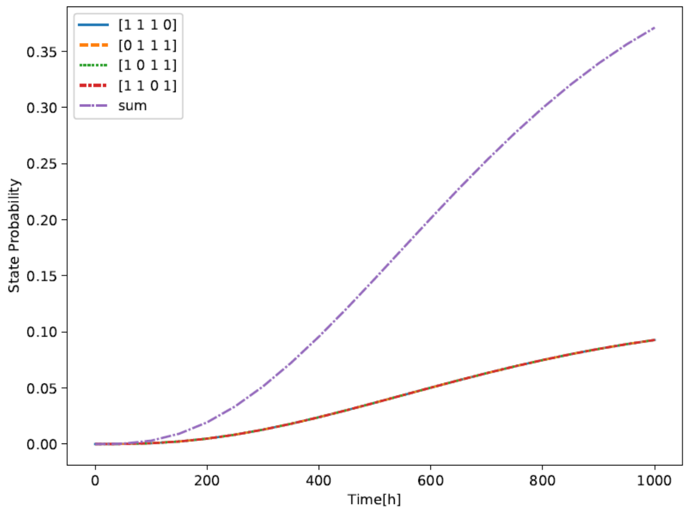


Fig. 2.2 Probability that exactly three components of system 2 are failed as a function of time; the purple line marks the probability development curve of the sum of the four contributing states

2.1.3.3 System 3: Common Failures

For the for redundant pumps of system 2, only the failure rates for independent pump failures have been considered so far. In system 3, the CCF rates of a 2o4, 3o4 and 4o4 failure are also considered. A *kor* failure means that *k* of *r* redundant components fail simultaneously or within a short time interval due to a joint CCF phenomenon that affects all components equally. The following rates are assumed:

$$\lambda_{2o4} = 4.\ 0 \cdot 10^{-4}, \lambda_{3o4} = 1.0 \cdot 10^{-4}, \lambda_{4o4} = 2.5 \cdot 10^{-5} \tag{2.12}$$

If a *ko4* failure occurs k = 2, 3, 4, k components fail simultaneously due to the underlying CCF phenomenon. Which components these are is normally random. For simplicity, it is however assumed for the model that pump 1 to pump *k* are affected by the kor CCF (German: GVA for gemeinsam verursachter Ausfall). In other words, in the event of a 2o4 CCF, it is assumed that the pumps 1 and 2 are affected by the CCF and have failed. The input file for system 3 is shown in the code section below:

```
def test_input_system3 ():

    r_input = RAMESUInput ()
    # ------------------------------------------------
    # COMPONENT - SECTION
    # ------------------------------------------------
    r_input.add_component(’pump_1 ’, (0, 1) , ’0 - ok 1 -failed ’)
    r_input.add_component(’pump_2 ’, (0, 1) , ’0 - ok 1 -failed ’)
    r_input.add_component(’pump_3 ’, (0, 1) , ’0 - ok 1 -failed ’)
    r_input.add_component(’pump_4 ’, (0, 1) , ’0 - ok 1 -failed ’)
    r_input.add_component(’gva ’,
                          states =(0 , 1, 2, 3) ,
                          description =’0 no GVA 1-2 v4 2-3 v4 3-4 v4 GVA
’)
    # -----------------------------------------------
    # Initial State section
    # -----------------------------------------------
    r_input.set_initial ((0 , 0, 0, 0, 0) , 1.0)
    # ----------------------------------------------------
    # Calculation Times t :
    # ----------------------------------------------------
    r_input.add_calc_times ( range (0, 1050 , 50))
    # -----------------------------------------------------
    # TRANSITION SECTION
    # -----------------------------------------------------
    # No GVA --> pumps fail independently
    r_input.add_markov(’pump_1 == 0 and gva == 0’, ’pump_1 = 1’, 1.e -3)
    r_input.add_markov(’pump_2 == 0 and gva == 0’, ’pump_2 = 1’, 1.e -3)
    r_input.add_markov(’pump_3 == 0 and gva == 0’, ’pump_3 = 1’, 1.e -3)
    r_input.add_markov(’pump_4 == 0 and gva == 0’, ’pump_4 = 1’, 1.e -3)
    # with a rate of 4.E -4 a 2v4 GVA happens
    r_input.add_markov(’gva == 0’, ’gva = 1’, 4.E -4)
    # with a rate of 1.E -4 a 3v4 GVA happens
    r_input.add_markov(’gva == 0’, ’gva = 2’, 1.E -4)
    # with a rate of 2.5E -5 a 4v4 GVA happens
    r_input.add_markov(’gva == 0’, ’gva = 3’, 2.5E -5)
    # if 2v4 GVA happens , pump3 and 4 can fail independently with a rate
1.E -3
    r_input.add_markov (’gva == 1 and pump_3 == 0’, ’pump_3 = 1’, 1.E -3)
    r_input.add_markov (’gva == 1 and pump_4 == 0’, ’pump_4 = 1’, 1.E -3)
    # if a 2v4 GVA happens pump 1 and 2 will fail too
    r_input.add_fixed (’gva == 1’, ’pump_1 = 1; pump_2 = 1’)
    # if a 3v4 GVA happens pump 1, 2 and 3 will fail too
    r_input.add_fixed (’gva == 2’, ’pump_1 = 1; pump_2 = 1; pump_3 = 1’)
    # if a 4v4 GVA happens pump 1, 2, 3 and 4 will fail too
    r_input.add_fixed (’gva == 3’,
                          'pump_1 = 1; pump_2 = 1; pump_3 = 1; pump_4 = 1')

    return r_input
```

The following failure states have been selected and regarded separately:

```
State 15 = [1, 1, 1, 1, 0] - no GVA , 4 pumps fail independently
State 19 = [1, 1, 1, 1, 1] - 2v4 - GVA ; 2 pumps also fail independently
State 21 = [1, 1, 1, 1, 2] - 3v4 GVA and 1 pump also fails independently
State 22 = [1, 1, 1, 1, 3] - 4v4 -GVA , i.e. all 4 pumps fail due to GVA
```

Fig. 2.3 shows the time development of the different state probabilities and the sum over these individual probabilities. The sum corresponds to the probability that all four pumps are failed.

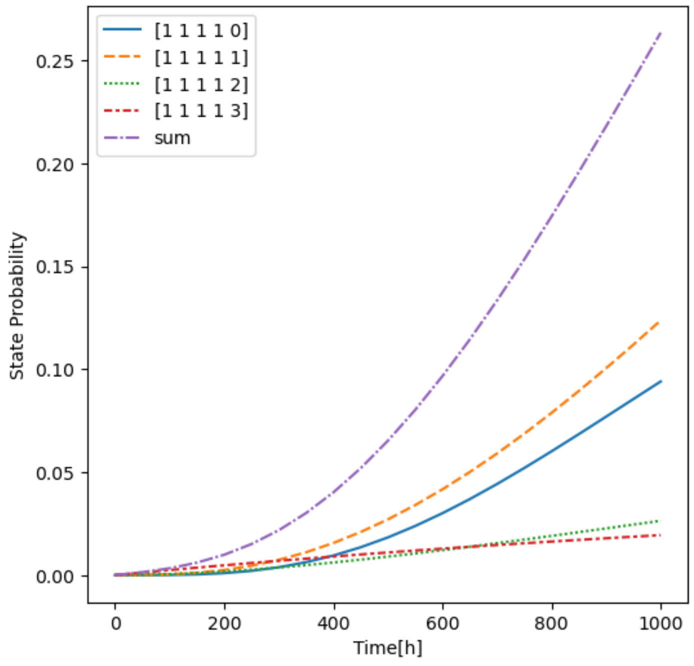


Fig. 2.3 State probabilities for RAMESU system 3 (CCF) as a function of time

The figure becomes more informative if the normalized state probability is shown as in Fig. 2.4. Whereas the relative probability of a complete 4o4 CCF decreases with time, the relative probabilities of a 2o4 CCF and a completely independent failure of all four pumps increase, the higher rates of independent pump failure start to dominate.

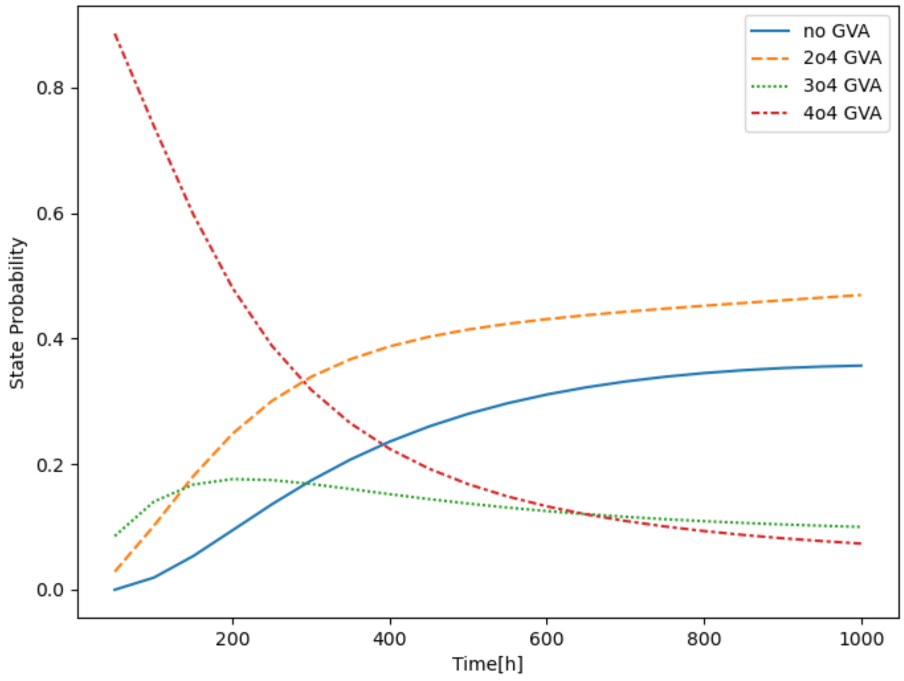


Fig. 2.4 Normalised state probabilities for RAMESU system 3 (CCF) as a function of time

2.1.3.4 System 4: Dependency of the Failure Behaviour on the System Status

System 4 is based on system 3. The difference is that the following dependencies exist in the failure behaviour in system 4: If a 2o4 CCF occurs and the pumps 1 and 2 fail the two remaining pumps 3 and 4 must ensure the required feed-in capacity and are used correspondingly more. The failure rate of the pumps 3 and 4 therefore increases from 1 E-03 to 2 E-03 compared to system 3.

If a 3o4 CCF occurs the failure rate of the remaining pump 4 increases from 1 E-03 to 5 E-03 compared to system 3.

```
def test_input_system4 ():

    r_input = RAMESUInput ()
    # ------------------------------------------------
    # COMPONENT - SECTION
    # ------------------------------------------------
    r_input.add_component('pump_1 ', (0, 1) , '0 - ok 1 -failed ')
    r_input.add_component('pump_2 ', (0, 1) , '0 - ok 1 -failed ')
    r_input.add_component('pump_3 ', (0, 1) , '0 - ok 1 -failed ')
    r_input.add_component('pump_4 ', (0, 1) , '0 - ok 1 -failed ')
    r_input.add_component('gva ', (0, 1, 2, 3) ,
                          '0 no GVA 1-2 v4 2-3 v4 3-4 v4 GVA ')
   # -----------------------------------------------
   # Initial State section
   # -----------------------------------------------
   r_input.set_initial((0 , 0, 0, 0, 0) , 1.0)
   # -----------------------------------------------
   # Calculation Times t :
   # -----------------------------------------------
   r_input.add_calc_times( range (0, 1050 , 50))
   # ----------------------------------------------
   # TRANSITION SECTION
   # -----------------------------------------------
   # no GVA --> pumps fail independently
   r_input.add_markov('(pump_1 , gva) == (0, 0)', 'pump_1 = 1', 1.e -3)
   r_input.add_markov('(pump_2 , gva) == (0, 0)', 'pump_2 = 1', 1.e -3)
   r_input.add_markov('(pump_3 , gva) == (0, 0)', 'pump_3 = 1', 1.e -3)
   r_input.add_markov('(pump_4 , gva) == (0, 0)', 'pump_4 = 1', 1.e -3)

   # with a rate of 4.E -4 a 2v4 GVA happens
   r_input.add_markov ('gva == 0', 'gva = 1', 4.E -4)
   # with a rate of 1.E -4 a 3v4 GVA happens
   r_input.add_markov ('gva == 0', 'gva = 2', 1.E -4)
   # with a rate of 2.5E -5 a 4v4 GVA happens
   r_input.add_markov ('gva == 0', 'gva = 3', 2.5E -5)

   # if 2v4 GVA happens , pump3 and 4 can fail independently
   # with an increases rate 2.E -3
   r_input.add_markov ('gva == 1 and pump_3 == 0', 'pump_3 = 1', 2.E -3)
   r_input.add_markov ('gva == 1 and pump_4 == 0', 'pump_4 = 1', 2.E -3)

   # if a 2v4 GVA happens pump 1 and 2 will fail too
   r_input.add_fixed ('gva == 1', 'pump_1 = pump_2 =1 ')
   # if a 3v4 GVA happens pump 1, 2 and 3 will fail too
   r_input.add_fixed ('gva == 2', 'pump_1 = pump_2 = pump_3 =1 ')


   # after a 3v4 GVA the failure rate of the remaining pump increases to
5.E -3
   r_input.add_markov ('gva == 2 and pump_4 == 0', 'pump_4 = 1', 5.E -3)

# if a 4v4 GVA happens pump 1, 2, 3 and 4 will fail too
   r_input.add_fixed ('gva == 3',
                     'pump_1 =1; pump_2 = pump_3 = pump_4 =1')

   return r_input
```

The generated space of system states corresponds to that of system 3. Fig. 2.5 shows the effects of increasing the independent failure rates when a CCF occurs on the probability of system failure.

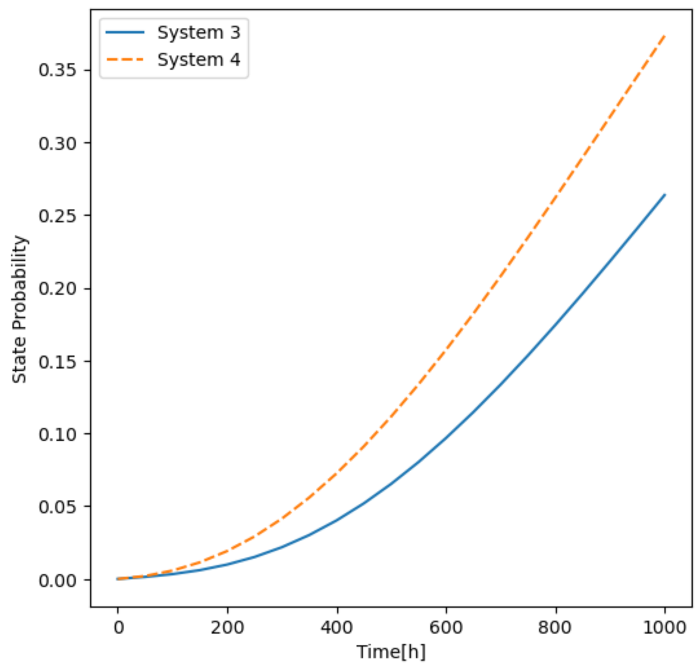


Fig. 2.5 System failure rate of system 3 in comparison to that of system 4

Due to the assumed increases in the failure rates still in operation when a CCF occurs, the failure probability of system 4 increases by approx. 30 % from 0.287 to 0.375 compared to system 3 after 1000 h of operation.

2.1.3.5 System 5: Cold Redundancy of Two Redundant Components

System 5 consists of two redundant components that are in operation at the same time (‘hot redundancy’). System 5 differs from system 1 in that the two redundant components operate in ‘cold redundancy’. This means that pump 1 runs first while pump 2 is in ‘stand-by’ idle mode. As soon as pump 1 fails, the system automatically switches over to pump 2, which then resumes operation. In this case, it is assumed that the switchover to pump 2 is successful with a probability of 95 % and fails with a probability of 5 %. If the switchover is not successful, pump 2 is not activated and is therefore not available for further operation. The failure of pump 2 can therefore be caused by the following two situations: (i) switchover to pump 2 works and pump 2 fails in the operational state at a rate of 1 E-03, and (ii) switchover to pump 2 does not work, and thus pump 2 is not available for further operation.

The system is considered to be failed if pump 1 and pump 2 are failed or are unavailable.

In this system modelling, the semi-Markov property is used the first time. The switchover to pump 2 does not take place via an exponentially distributed random time, but at the point in time at which pump 1 fails. The semi-Markov properties of a system are defined via singular matrices that are applied at specific points in time. As pump 1 can fail at any time in this example, a small value (here 1 h) is selected for the period of the application times of the singular matrix in order to achieve the best possible approximation to the continuous failure time of pump 1. An even smaller value could have been used for the period, but this would have no relevant influence on the results.

For system 5, three components were defined, two pumps and a switch ($Swtch$) for the changeover. For pump 1, two states are defined: 1 –running and 2 – failed. For pump 2, four states are defined: 0 – stand-by, 1 – running, 2 – failed and 3 – not available, as the switchover has not taken place. The switch has three states: 0 – inactive, 1 – switchover successful and 2 – switchover not successful. The initial state is [1, 0, 0], i.e. pump 1 is running, pump 2 is in stand-by mode, and the switch is inactive. When pump 1 is running, it can fail with a rate of 1 E-03 per hour. As long as pump 1 is running, pump 2 is in stand-by mode.

```
def test_input_system5 ( prob_sm_1 = 0.95 , prob_sm_2 = 0.05) :

    r_input = RAMESUInput ()
    # ------------------------------------------------
    # COMPONENT - SECTION
    # ------------------------------------------------
    r_input.add_component('pump_1 ', (1, 2) , '1-on 2- failed ')
    r_input.add_component('pump_2 ', (0, 1, 2, 3),
                          '1-on 2-failed 3-failure due to switch-over')
    r_input.add_component ('switch ', (0, 1, 2),
                           '0- idle 1-ok 2-failed ')
    # -----------------------------------------------
    # Initial State section
    # -----------------------------------------------
    r_input.set_initial((1 , 0, 0) , 1.0)
    # -----------------------------------------------------
    # Calculation Times t :
    # -----------------------------------------------------
    r_input.add_calc_times( range (0, 1050 , 50))
    # ------------------------------------------------------
    # TRANSITION SECTION
    # ------------------------------------------------------
    r_input.add_semi_markov('pump_1 == pump_2 == switch == 0',
                            'switch = 1', prob_sm_1 ,
                            calc_times = range (0, 1001 , 1))
    r_input.add_semi_markov('pump_1 == 2 and pump_2 == switch == 0',
                            'switch = 2', prob_sm_2,
                            calc_times = range (0, 1001 , 1))

    r_input.add_markov('pump_1 == 1', 'pump_1 = 2', 1.e -3)
    r_input.add_markov('pump_2 == switch == 1', 'pump_2 = 2', 1.e-3)

    r_input.add_fixed('pump_2 == 0 and switch == 1', 'pump_2 = 1')
    r_input.add_fixed('pump_2 == 0 and switch == 2', 'pump_2 = 3')
    r_input.add_fixed('pump_1 == 1', 'pump_2 = switch = 0')
    return r_input
```

The condition for the switchover is defined by the first singular matrix. The switch is in the inactive state 0 until pump 1 fails. As soon as pump 1 fails, the switch is activated and successfully switches to pump 2 with a probability of 0.95. If the switchover is successful, the switch assumes state 1. With a probability of 0.05 the switchover is not successful. In this case, the switch is set to state 2. This is defined by the conditions of the second singular matrices. If the switchover is successful and the switch is in state 1, pump 2 is set from stand-by to operational state (i.e. from state 0 to state 1), this is defined in the first fixed transition. If pump 2 is in the stand-by state and the switchover is not successful after pump 1 fails, the switch is set to state 2 (see second added semi-Markov transition). If the switch is in state 2, pump 2 is set to state 3, which describes the unavailability of pump 2 due to the faulty switchover (see second fixed transition). If pump 2 has been successfully switched over to pump 1 after pump 2 has failed, pump 2 can fail randomly during its operational state with a failure rate of 1 E-03 (see second Markov transition).

System 5 is considered to have failed if pump has failed, and pump 2 is in state 2 or state 3. Pump 2 is in state 3 if it is not possible to switch to pump 2 due to the defective switch. The failure probabilities of system 1 (hot redundancy) and system 5 (cold redundancy) are shown graphically in Fig. 2.6 below.

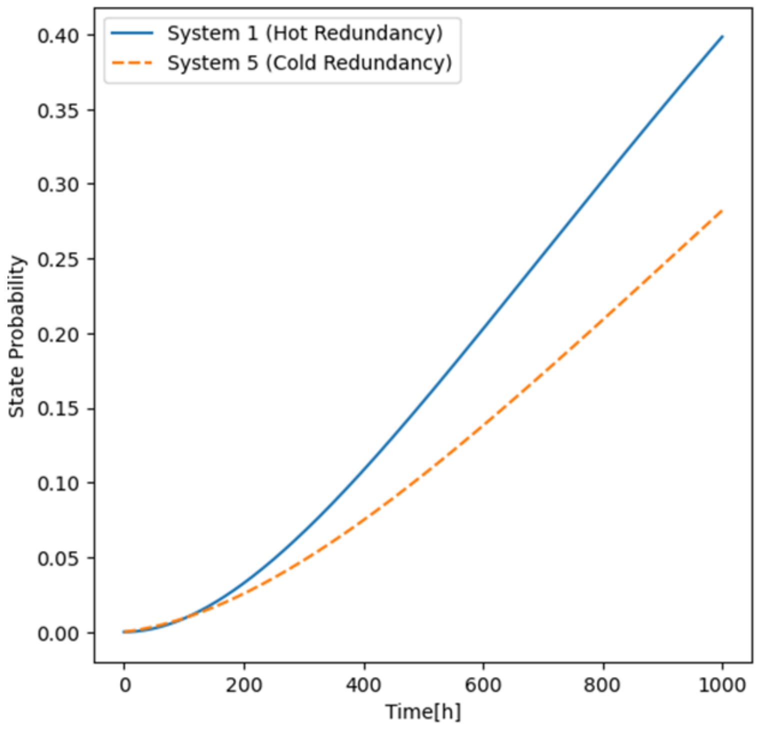


Fig. 2.6 Comparison of the system state probability, for a hot redundancy system (blue) and for a cold redundancy system (red)

Due to the cold redundancy, the failure probability of system 5 can be reduced by 30 % from approx. 0.4 (system 1) to 0.28 (system 5) after 1000 h of operation compared to system 1. This improvement in reliability was achieved even though a failure probability of 0.05 was assumed for the switchover to pump 2.

2.1.3.6 System 6: Cyclic Switching of the Pumps at Specific Times

Unlike in system 5, the switchover does not only take place when pump 1 fails, but at certain points in time when the system switches between pumps 1 and 2 cyclically. At the beginning, pump 1 is in operation, while pump 2 is in stand-by mode. After 250 h of operation, pump 1 switches over to pump 2 if pump 1 has not failed in the meantime. If the switchover is successful (p = 0.95) pump 2 is activated, and pump 1 is set to stand-by. Pump 2 also remains in operation for 250 h. Pump 1 is then switched back to pump 2, provided pump 2 has not failed in the meantime, and pump 2 is set to the stand-by state. It is assumed that the individual switchovers are each 95 % successful and that there is a 5 % probability of failure. If the switchover is unsuccessful, the pump in operation at that time will continue to operate up to the end as the switch for the switchover is assumed to have failed.

The pumps are not maintained during the stand-by phases. However, it is assumed that the failure rates of the pumps are reduced in their stand-by phases. It is assumed that the failure rate of the stand-by phase is only 1/5 of the failure rate of the operational phase, i.e. $\lambda_{standby} = 0.2 \cdot \lambda_{operation}$. In other words, the failure rate in the operational phase is 1 E-03, the failure rate in the stand-by phase is 2 E-04. If the switchover to pump 2 (or pump 1) is not successful, pump 2 (or pump 1) is considered unavailable for further operation. The system is considered to be failed if both pumps fail or are unavailable during their respective operating phases. Due to the more complex behaviour of system 6, the modelling of the singular matrices and the transition section are explained in detail.

The switchovers take place with a period of 250 h until the end of the calculation time. The switch can fail with a probability of 0.05 for each changeover in which the switch becomes active. This is modelled in the first two singular matrices. As it is assumed that the switch can only fail at the changeover times, the application times of these matrices start at *T_start* = 250 h and have a period of 250 h until the end of the time. The second semi-Markov transition is only used when pump 1 is in operation, pump 2 is in stand-by and the switch is intact, i.e. the system state [1, 0, 1] is present. This state can be present at the times t = 250 h and t = 750 h. The third semi-Markov transition is used when the system state [0, 1, 1] is present, i.e. pump 1 is in stand-by, pump 2 is in operation and the switch is intact. This state can be present at the times t = 500 h and t = 1000 h due to the cyclical switchovers. It should be noted that the same values for *T_start, T_end* and *Period* have been set in these semi-Markov transitions. This means that the corresponding changeovers are carried out at exactly these points in time.

If the switch has failed and it is therefore not possible to switch over to the pump that is in stand-by, this pump is considered unavailable in the further course and is set from state 0 to state 3. In this case, the running pump remains in operation. This situation is modelled by two fixed transitions.

```
def test_input_system6 ():

    r_input = RAMESUInput ()
    # ----------------------------------------------
    # COMPONENT - SECTION
    # ----------------------------------------------
    r_input.add_component ('pump_1 ', (0, 1, 2, 3) , '0- stand by 1-on 2-
failed 3-not available')
    r_input.add_component ('pump_2 ', (0, 1, 2, 3) ,'0- stand by 1-on 2-
failed 3-not available')
    r_input.add_component ('switch ', (1, 2) ,'1-ok 2- failed ')
    # ---------------------------------------------
    # Initial State section
    # -----------------------------------------------
    r_input.set_initial ((1 , 0, 1) , 1.0)
    # -----------------------------------------------------
    # Calculation Times t :
    # -----------------------------------------------------
    r_input.add_calc_times ( range (0, 1050 , 50))
    # ------------------------------------------------------
    # TRANSITION SECTION
    # ------------------------------------------------------
    # The switch can fail with a probability of 0.05 for each
    # changeover in which the switch becomes active
    r_input.add_semi_markov('(pump_1, pump_2, switch) == (1 ,0 ,1) or '
                            '(pump_1 , pump_2 , switch) == (0 ,1 ,1)',
                            'switch = 2',
                            prob =0.05 ,
                            calc_times = range (250 , 1250 , 250))
    # The second semi - markov transition is only used when pump 1 is in
    # operation , pump 2 is in standby and the switch is intact
    r_input.add_semi_markov('(pump_1 , pump_2 , switch) == (1 ,0 ,1)',
                            'pump_1 = 0; pump_2 = 1',
                            prob =1. ,
                            calc_times =[250 , 750] ,)
    # The third semi - markov transition is only used pump 1 is in standby
,
    # pump 2 is in operation and the switch is intact
    r_input.add_semi_markov('(pump_1 , pump_2 , switch) == (0 ,1 ,1) ',
                            'pump_1 = 1; pump_2 = 0',
                            prob =1. ,
                            calc_times =[500 , 1000])
    # The failure rate in the operating phase is 1.E -3
    r_input.add_markov('(pump_1 , pump_2 , switch ) == (1 ,0 ,1) or '
                       '(pump_1 , pump_2 , switch ) == (1 ,3 ,2) ',
                       'pump_1 = 2', rate =1.e -3)
    r_input.add_markov ('(pump_1 , pump_2 , switch ) == (0 ,1 ,1) or '
                        '(pump_1 , pump_2 , switch ) == (3 ,1 ,2) or '
                        '(pump_1 , pump_2 , switch ) == (2 ,1 ,1) or '
                        '(pump_1 , pump_2 , switch ) == (1 ,2 ,1) ',
                        'pump_2 = 2', rate =1.e -3)

    # The failure rate in the standby phase is 4.E -4
    r_input.add_markov ('(pump_1 , pump_2 , switch ) == (0 ,1 ,1) ',
                        'pump_1 = 2', rate =2.e -4)
    r_input.add_markov ('(pump_1 , pump_2 , switch ) == (1 ,0 ,1) ',
                        'pump_2 = 2', rate =2.e -4)
```

```
    # If the switch has failed or one pump is in standby and one failed
    # and it is therefore not possible to switch , the standby - pump is
considered
    # unavailable in the further course, is set from state 0 to state 3
    r_input.add_fixed ('(pump_1 , pump_2 , switch) == (1 ,0 ,2) or '
                       '(pump_1 , pump_2 , switch) == (2 ,0 ,1) ',
                       'pump_2 = 3')
    r_input.add_fixed ('(pump_1 , pump_2 , switch ) == (0 ,1 ,2) or '
                       '( pump_1 , pump_2 , switch ) == (0 ,2 ,1) ',
                       'pump_1 = 3')

    return r_input
```

Eleven different system states can be reached:

1. [1, 0, 1] pump 1 is running, pump 2 is in stand-by, and the switch is intact.
2. [2, 3, 1] pump 1 fails before the time of switchover. Although the switch is intact, pump 2 cannot be activated and is set to not available (= 3). This state is considered a system failure.
3. [0, 1, 1] pump 1 is on stand-by while pump 2 is running. The switch is in a functional state.
4. [3, 2, 1] pump 2 fails before the switchover time. Since the switchover time is not reached due to the premature failure of pump 2, pump 1 cannot be activated and is set to not available (= 3). This state is regarded as a system failure.
5. [2, 1, 2] pump 1 fails in the stand-by phase while pump 2 is in operation. The switch is in ok state. When the system reaches the switchover time, pump 2 continues to run.
6. [1, 2, 1] pump 2 fails in the stand-by phase while pump 1 is in operation. The switch is intact. When the system reaches the switchover point, pump 1 continues to run.
7. [1, 3, 2] switchover to pump 2 does not work because the switch is defective. pump 1 therefore continues running.
8. [2, 3, 2] switchover to pump 2 does not work because the switch is defective. While pump 1 continues to operate, pump 1 fails. This state is considered a system failure.
9. [3, 1, 2] switchover to pump 1 does not work because the switch is defective. Pump 2 therefore continues running.
10. [3, 2, 2] switchover to pump 1 does not work because the switch is defective. While pump 2 continues to operate, pump 2 fails. This state is considered a system failure.
11. [2, 2, 1] one of the two pumps fails in the stand-by phase (see state 4 or 5), the other pump fails while it is in operation.

Another way of verifying the model is to check the plausibility of the state probabilities. For example, the system state [3, 2, 1] has the probability 0 up to the first switchover time t = 250 h, as pump 2 is not in operation during this time period and therefore cannot fail during operation. For the same reason, the state [3, 2, 2] also has a probability of 0. The progression of the state probabilities over time is shown in Fig. 2.7.

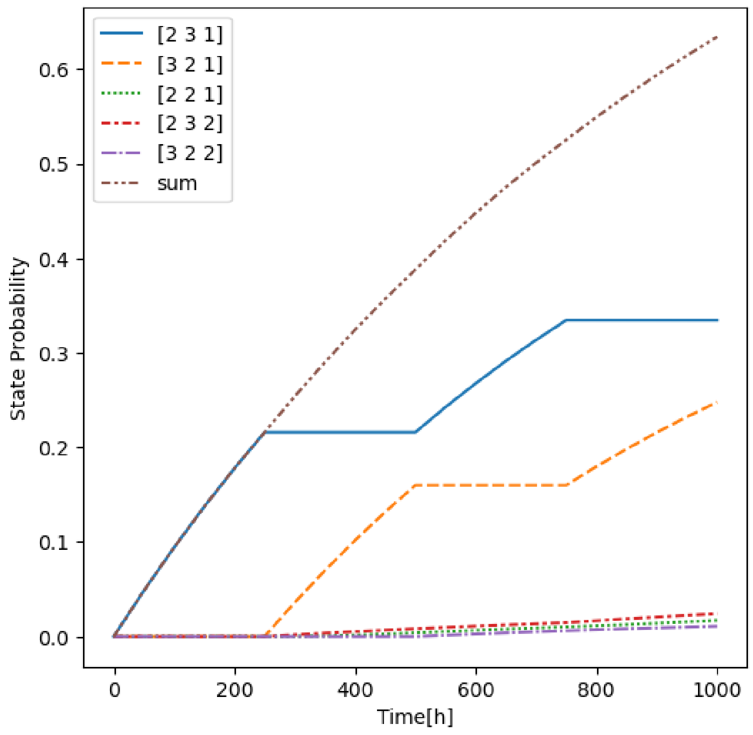


Fig. 2.7 Development of the state probability for state [3, 2, 1] (pump 2 fails before switchover time) and state [3, 2, 2] (switchover to pump 1 does not work because the switch fails)

The development over time of the probabilities of 'state [2 3 1]' and 'state [3 2 1]' (Fig. 2.7) also indicates the correctness of the model. For example, the influence of the switchovers on the probability curve can be recognised in these two states. In the time ranges in which the probabilities of 'state [2 3 1]' increase because pump 1 is in operation and can fail, the probabilities of 'state [3 2 1]' remain constant because pump 2 is in stand-by. The reverse is also true. In the time ranges in which the probabilities of 'state [3 2 1]' increase, the probabilities of 'state [2 3 1]' remain constant, as pump 1 is in stand-by during these time ranges. This figure clearly shows that the failures that occur in the operational state of the pumps provide the largest contribution to the system failure. This can be explained by the fact that the failure of one pump in the operational state results in the simultaneous unavailability of the other pump, as the switchover no longer occurs due to the failure. This is also associated with a system failure. The failures that occur during the stand-by phase of the pumps or due to a faulty changeover, on the other hand, provide a relatively small contribution to the system failure.

2.1.3.7 System 7: Influence of Repair

System 7 consists of two pumps with ‘hot redundancy’, i.e. both pumps run simultaneously. If both pumps run together, the failure rate is 0.001. If one of the two pumps fails, the failure rate of the other pump still running increases by 50 % to 0.0015.

The pumps are tested sequentially with a period of 200 h. In other words, pump 1 is tested at the time t = 200h, pump 2 after t =400 h, pump 1 after t = 600 h and pump 2 after t = 800 h. If the tested pump is in a failed state, it is repaired at a repair rate of 0.05, i.e. the average repair time is 20 h. As soon as the repair of the failed pump has been completed, it is switched on again and is assumed to be ‘as good as new’. When both pumps are running together again, the failure rate of both pumps is again 0.001.

It is assumed that at the later test times t = 600 h and t = 800 h the failure of the tested pump is recognised with a reduced probability of 90 %. This means that with a probability of 10 %, the failure remains undetected, and no repair takes place. System 7 is failed if both pumps are in a failed state. The variable ‘Repair’ was defined to indicate when the respective pumps are in the repair state. This variable indicates whether a failure has been detected, and a repair is required (1 – repair demanded) or not (0 – no repair). The ‘Repair’ variable therefore serves as an indicator variable here.

The initial state [1, 1, 0], which has a probability of 1, indicates that both pumps are in operation at the same time. In the singular matrices SM 1 – SM 4, the system checks at certain points in time whether the component being checked at that time is in a failed state. If so, the variable ‘Repair’ is set to state 1. For example, SM 1 is to be interpreted as follows: If pump 1 is in a failed state at the time t = 200 h and pump 2 is still in operation, the variable ‘Repair’ is set to the value 1 with probability 1. This means that the failure and the need for repair is recognised with certainty at this point in time. The situation is slightly different at the time t = 600 h, when the failure of pump 1 is only recognised with a probability of 90 %. The variable ‘*Repair*’ is used in the *TRANSITION-SECTION* to indicate that the corresponding component needs to be repaired.

```
def test_input_system7():

    r_input = RAMESUInput ()
    # ----------------------------------------------
    # COMPONENT - SECTION
    # ----------------------------------------------
    r_input.add_component('pump_1 ', (1, 2) , '1-on 2- failed ')
    r_input.add_component('pump_2 ', (1, 2) , '1-on 2- failed ')
    r_input.add_component('repair ', (0, 1) , '1-no repair, 2- repair
demanded ')
    # ----------------------------------------------
    # Initial State section
    # ----------------------------------------------
    r_input.set_initial((1 , 1, 0) , 1.0)
    # ----------------------------------------------------
    # Calculation Times t :
    # ----------------------------------------------------
    r_input.add_calc_times( range (0, 1050 , 50))
    # -----------------------------------------------------
    # TRANSITION SECTION
    # -----------------------------------------------------
    r_input.add_semi_markov('(pump_1 , pump_2 , repair ) = (2 ,1 ,0)',
                            'repair = 1', 1., calc_times =[200 , 600])
    r_input.add_semi_markov('(pump_1 , pump_2 , repair ) = (1 ,2 ,0) ',
                            'repair = 1', 1., calc_times =[400 , 800])
    # ----------------------------------------------------
    # Specification of Markov Transitions
    # ----------------------------------------------------
    # Failure rate if both pumps run at the same time
    r_input.add_markov('pump_1 == pump_2 == 1','pump_1 = 2', 1.e -3)
    r_input.add_markov('pump_1 == pump_2 == 1','pump_2 = 2', 1.e -3)
    # Failure rate if one pump is failed
    r_input.add_markov('(pump_1 , pump_2 ) == (1 ,2) ','pump_1 = 2', 1.5e -
3)
    r_input.add_markov('(pump_1 , pump_2 ) == (2 ,1) ','pump_2 = 2', 1.5e -
3)
    # Repair of pump1 happens with rate 0.05 if pump1 found failed at
testing time
    r_input.add_markov('(pump_1 , pump_2 , repair ) = (2 ,1 ,1) ', 'pump_1
= 1', 0.05)
    # Repair of pump2 happens with rate 0.05 if pump2 found failed at
testing time
    r_input.add_markov('(pump_1 , pump_2 , repair ) = (1 ,2 ,1) ', 'pump_2
= 1', 0.05)
    # -----------------------------------------------------
    # Specification of Fixed Transitions
    # -----------------------------------------------------
    # If one pump has been repaired and other pump is not yet failed set
repair to 0
    r_input.add_fixed ('pump_1 == pump_2 == repair == 1', 'repair = 0')

    return r_input
```

Fig. 2.8 shows the course of the probabilities for those states that lead to system failure. These are the states [2, 2, 0] (state 4) and [2, 2, 1] (state 5). The probability of failure of the system is given by the sum of the probabilities of states 4 and 5 at the respective calculation times. The influence of repairs is not clearly visible in this figure, as only failed states are shown.

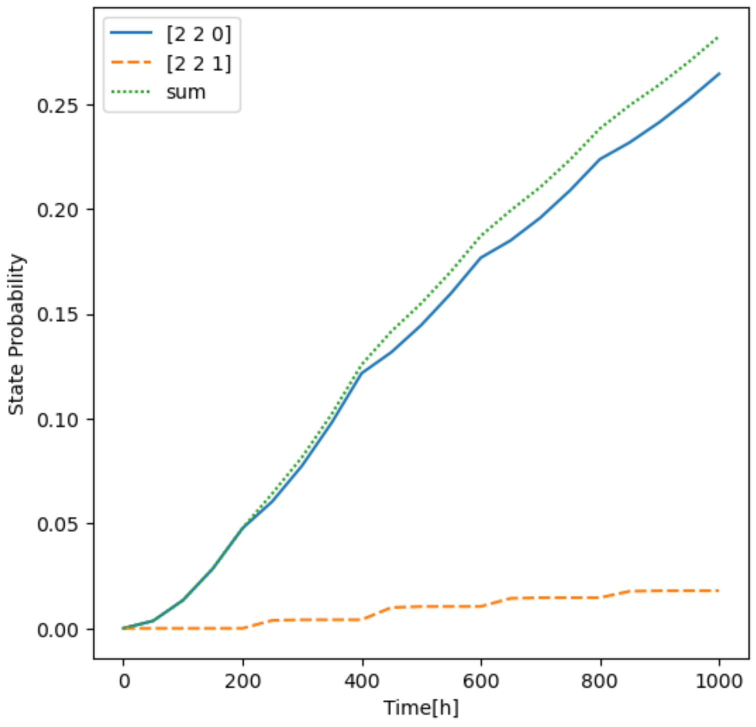


Fig. 2.8 Development of the state probabilities for system 7 for states in which the system is already failed

The influence of the repairs can be recognised more clearly for those states in which the system has not yet failed, for the states [1, 1, 0] (state 0), [2, 1, 0] (state 1) and [1, 2, 0] (state 2), as shown in Fig. 2.9.

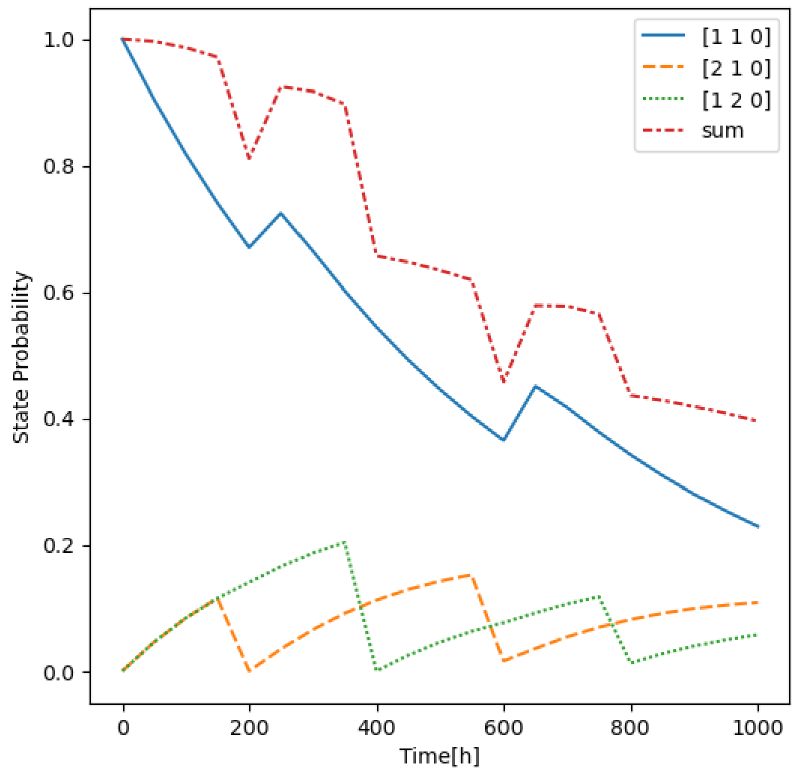


Fig. 2.9 Development of the state probabilities for system 7 for states in which the system has not yet failed

For all three states, for which probability curves are provided, the influence of the repairs at the respective maintenance times is clearly recognisable. For state 0, in which both pumps are intact, the probability increases slightly immediately after the maintenance times. This slight increase in probability can be explained by the repair of the failed pump. With regard to states 1 and 2, the staggered repair of the pumps at the respective maintenance times can be clearly recognised. Up to the first maintenance at the time t = 200 h, condition 1 and condition 2 have the same probability curve, as the same failure rates were assumed for both pumps. Up to this point in time, the probabilities of states 1 and 2 increase to 0.14. After 200 h, maintenance is carried out on pump 1, the failure of which is recognised and subsequently repaired with an exponentially distributed repair time of 20 h. At this point, the probability of state 1 decreases from 0.14 to 0, while the probability of state 2 continues to rise to 0.215 up to the maintenance time t = 400 h. The same applies to state 2 at the maintenance time t = 400 h.

For the maintenance times t = 600 h and t = 800 h, it can be seen that the probabilities of states 1 and 2 are slightly higher than 0 after the repairs. This is due to the fact that at these times the failure of the respective component is not recognised with a 10 % probability and in this case no repair takes place.

The following description refers to the representation of the probability curve shown below. During the maintenance of pump 2 at the time t = 400 h, the failure of this pump is recognised and changes to the repair state, which is defined by the system state [1, 2, 1] (state 3). At this time, state [1, 2, 0] (state 2) has a probability of 0 and state [1, 2, 1] a probability of 0.215. Due to the exponentially distributed repair time with a rate of 0.05, the probability of state [1, 2, 1] decreases to 0 until the next maintenance of pump 2 at the time t = 800 h, while the probability of state [1, 2, 0] rises to approx. 0.169 by this time. As the failure is recognised with a probability of 0.9 during the second maintenance of pump 2, the probability of the repair state [1, 2, 1] of pump 2 at the time t = 800 h is approx. 0.152, while the probability of the state [1, 2, 0] at this point in time does not decrease to 0 but to a value of approx. 0.017.

To determine the probability of pump 2 being in the failed state while pump 1 is still in operation, the probabilities of states 2 and 3 must be added up. The sum of the probabilities of these two states is also shown in Fig. 2.10. The probability behaviour for the failure of pump 1 can be explained in the same way as the probability behaviour for the failure of pump 2. The corresponding maintenance times for pump 1 are t = 200 h and t = 600 h.

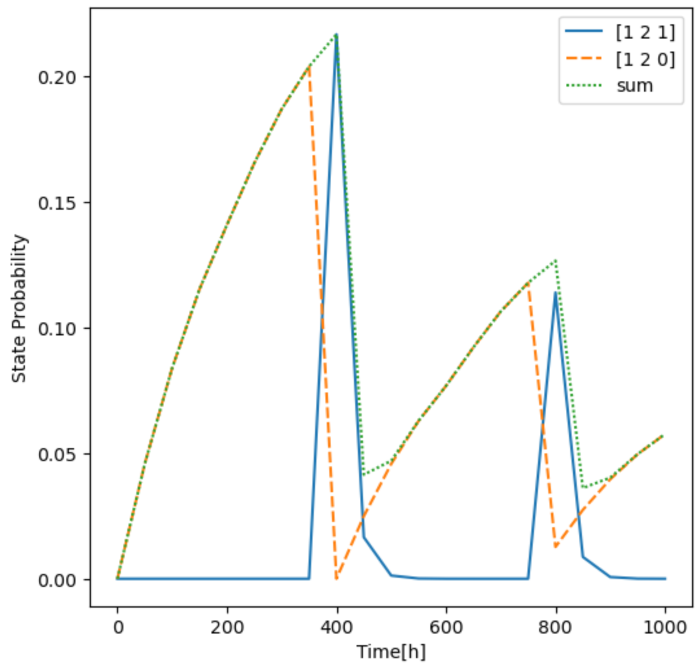


Fig. 2.10 Development of the state probabilities for system 7 for state 2 and state 3

2.1.3.8 Consideration of Uncertainties

In this section, uncertainties relating to system 7 are to be taken into account. The table below lists the distributions that are assumed for the uncertainties of system 7. There is no uncertainty with regard to the probability of the initial state.

Tab. 2.1 Probability distributions for the uncertainties of system 7

No.	Parameter	Distribution
1	Failure rate pump 1 if both pumps are working	Gamma(1.5; 1000)
2	Failure rate pump 2 if both pumps are working	Gamma(1.5; 1000)
3	Failure rate pump 1 if pump 2 is failed	Gamma(2.5; 1000)
4	Failure rate pump 2 if pump 1 is failed	Gamma(2.5; 1000)
5	Repair time pump 1	tW1 ~ U(150; 250)
6	Repair time pump 2	tW2 ~ U(350; 450)
7	Time second repair pump 1	tW3 ~ U(500; 700)
8	Time second repair pump 2	tW3 + tW4 ~ U(50; 200)
9	Probability for passing to repair state tW1	U(0.8; 1.0)
10	Probability for passing to repair state tW2	U(0.8; 1.0)
11	Probability for passing to repair state tW3	U(0.6; 1.0)
12	Probability for passing to repair state tW4	U(0.6; 1.0)
13	Repair rate pump 1	Gamma(1.0; 20)
14	Repair rate pump 2	Gamma(1.0; 20)

```
def test_input_system7_uncertain ( rate_fail_p1_b =1.e -3,
                                   rate_fail_p2_b =1.e -3,
                                   rate_fail_p1_s =1.5e -3,
                                   rate_fail_p2_s =1.5e -3,
                                   t_repair_p1 =200,
                                   t_repair_p2 =400,
                                   t2_repair_p1 =600,
                                   t2_repair_p2=800,
                                   p_repair_tr1 =1.,
                                   p_repair_tr2 =1.,
                                   p_repair_tr3 =0.9,
                                   p_repair_tr4=0.9,
                                   rate_repair_p1 =0.05,
                                   rate_repair_p2 =0.05) :
    r_input = RAMESUInput ()
    # ------------------------------------------------
    # COMPONENT - SECTION
    # ------------------------------------------------
    r_input.add_component ('pump_1 ', (1, 2) , '1-on 2- failed ')
    r_input.add_component ('pump_2 ', (1, 2) , '1-on 2- failed ')
    r_input.add_component ('repair ', (0, 1) , '1-no repair 2- repair
demanded ')
```

```
    # ----------------------------------------------
    # Initial State section
    # ----------------------------------------------
    r_input.set_initial ((1 , 1, 0) , 1.0)
    # ----------------------------------------------------
    # Calculation Times t :
    # ----------------------------------------------------
    r_input.add_calc_times ( range (0, 1050 , 50))
    # -----------------------------------------------------
    # TRANSITION SECTION
    # -----------------------------------------------------
    r_input.add_semi_markov('pump_1 == 2 and pump_2 == 1 and repair==   0',
'repair = 1', p_repair_tr1 , calc_times =[ t_repair_p1 ])
    r_input.add_semi_markov('pump_1 == 1 and pump_2 == 2 and repair== 0',
'repair = 1', p_repair_tr2 , calc_times =[ t_repair_p2 ])
    r_input.add_semi_markov('pump_1 == 2 and pump_2 == 1 and repair== 0',
'repair = 1', p_repair_tr3 , calc_times =[ t2_repair_p1 ])
    r_input.add_semi_markov('pump_1 == 1 and pump_2 == 2 and repair== 0',
'repair = 1', p_repair_tr4 , calc_times =[ t2_repair_p2 ])
   --------------------------------------------------------------
   # Specification of Markov Transitions
   --------------------------------------------------------------
   # Failure rate if both pumps run at the same time
   r_input.add_markov('pump_1 == 1 and pump_2 == 1','pump_1 = 2',
rate_fail_p1_b )
   r_input.add_markov('pump_1 == 1 and pump_2 == 1','pump_2 = 2',
rate_fail_p2_b )
   # Failure rate if one pump is failed
   r_input.add_markov('pump_1 == 1 and pump_2 == 2','pump_1 = 2',
rate_fail_p1_s )
   r_input.add_markov('pump_1 == 2 and pump_2 == 1','pump_2 = 2',
rate_fail_p2_s )
   # Repair of pump1 happens with rate 0.05 if pump1 found failed at
testing time
   r_input.add_markov('pump_1 == 2 and pump_2 == 1 and repair == 1',
'pump_1 = 1',                       rate_repair_p1 )
   # Repair of pump2 happens with rate 0.05 if pump2 found failed at
testing time
   r_input.add_markov('pump_1 == 1 and pump_2 == 2 and repair == 1',
'pump_1 = 1', rate_repair_p2 )
--------------------------------------------------------------
# Specification of Fixed Transitions
-------------------------------------------------------------
# If one pump has been repaired and other pump is not yet failed set
# repair to 0
  r_input.add_fixed('pump_1 == 1 and pump_2 == 1 and repair == 1', 'repair
= 0')

 return r_input
```

The following uncertain parameters can be defined using the new SUSA sampling module:

```
rate_fail_p1_b = _dist.Parameter('rate_fail_p1_b ', _dist.Gamma (a=1.5 ,
scale =1./1000.))
rate_fail_p2_b = _dist.Parameter('rate_fail_p2_b ', _dist.Gamma (a=1.5 ,
scale =1./1000.))
rate_fail_p1_s = _dist.Parameter('rate_fail_p1_s ', _dist.Gamma (a=2.5 ,
scale =1/1000.))
rate_fail_p2_s = _dist.Parameter('rate_fail_p2_s ',  _dist.Gamma (a=2.5 ,
scale =1/1000.))
```

```
rate_repair_p1 = _dist.Parameter('rate_repair_p1 ', _dist.Gamma (a=1, scale
=1/20.))
rate_repair_p2 = _dist.Parameter('rate_repair_p2 ', _dist.Gamma (a=1, scale
=1/20.))
t_repair_p1 = _dist.Parameter('t_repair_p1 ', _dist.Uniform (150 , 100) )
t_repair_p2 = _dist.Parameter('t_repair_p2 ', _dist.Uniform (350 , 100) )
t2_repair_p1 = _dist.Parameter('t2_repair_p1 ', _dist.Uniform (500 , 200) )
t2_repair_p2 = _dist.Parameter('t2_repair_p2 ',_dist.Uniform (50 , 150) )
p_repair_tr1 = _dist.Parameter('p_repair_tr1 ',_dist.Uniform (0.8 ,0.2) )
p_repair_tr2 = _dist.Parameter('p_repair_tr2 ',_dist.Uniform (0.8 ,0.2) )
p_repair_tr3 = _dist.Parameter('p_repair_tr3 ',_dist.Uniform (0.6 ,0.4) )
p_repair_tr4 = _dist.Parameter('p_repair_tr4 ',_dist.Uniform (0.6 ,0.4) )
```

The full power of the SUSA sampling module can be used to sample the requested uncertain parameters.

```
params = [ rate_fail_p1_b, rate_fail_p2_b, rate_fail_p1_s, rate_fail_p2_s,
rate_repair_p1, rate_repair_p2 , t_repair_p1, t_repair_p2, t2_repair_p1,
t2_repair_p2, p_repair_tr1,  p_repair_tr2, p_repair_tr3, p_repair_tr4 ]

rng = _rngs.LatinHyperCube ( generator = 'Mersenne_Twister',
                             sample_correlation = False,
                             seed =1337,
                             median_interval_point_selection = True )
med_usa = Medusapy (params , [], rng)
arr = med_usa.get_sampled (20 , list_of_dicts = True )
```

The combination of the objects *VarFunction* and *VariationQueue* provided in the SUSA Simulation Run module allows to efficiently run the Markov analysis for all sampled parameter sets and to collect the results:

```
# The VarFunction takes as input one input function which takes the de-   #
fined uncertain parameter as named inputs
# and a helper function which returns the output of the RAMESU analysis  #
given the input
ramesu_func = VarFunction( test_input_system7_uncertain, ramesu_helper )
queue = VariationQueue( ramesu_func , 3 )
queue.stage(* arr). wait ()
Sample = [ res for res in queue [:].done.value ]
```

Finally, the influence of the uncertain variables on the probability of failure of system 7 is shown in Fig. 2.11. For better visualisation, the sample size of the uncertain variables was limited to 20. As already described in Section 2.1.3.7, the failure probability of the system is given by the sum of the probabilities of the states 4 = [2, 2, 0] and 5 = [2, 2, 1] at the respective calculation times. As a reminder: A system failure is defined by the fact that both pumps are in the failed state. State [2, 2, 1] refers to the occurrence of the system failure while one of the pumps is under repair. State [2, 2, 0] refers to the failure of both pumps without one of the pumps being under repair. The influence of the uncertainties on the state in which pump 1 is in operation and pump 2 is in failed state is shown in Fig. 2.12. By integrating the Markov program into SUSA, the varying results of the Markov model can be subjected to an uncertainty and sensitivity analysis as efficiently as possible.

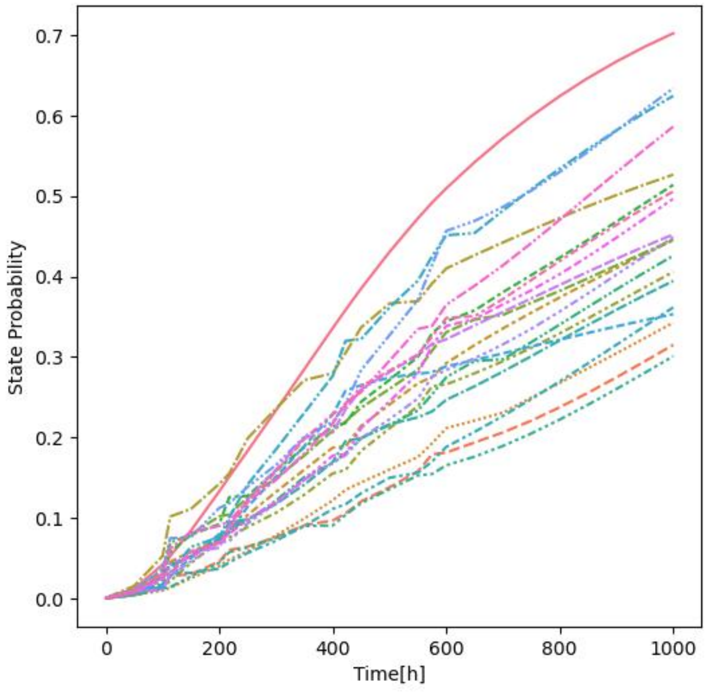


Fig. 2.11 Development of the probability of the state in which pump 1 is in operation and pump 2 is failed for various epistemic runs

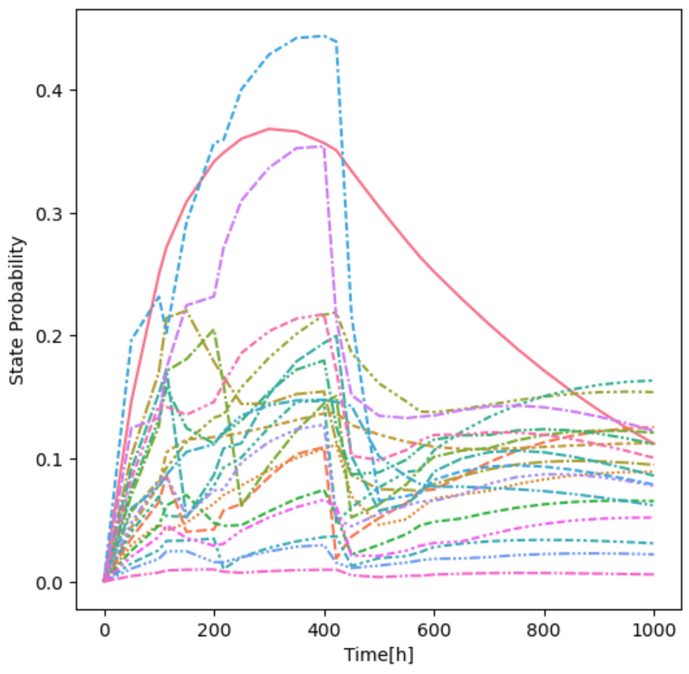


Fig. 2.12 Development of the probability of the state in which pump 1 is in operation and pump 2 is failed for various epistemic runs

2.2 Integration of the Functionality of the AURA Program for Estimating Distributions for Reliability Parameters

The quantification of model parameters plays an important role in the application of models for reliability analyses of technical systems. The model parameters largely consist of reliability parameters such as failure rates, repair rates and failure probabilities per demand.

Data from operating experience is generally used to estimate reliability parameters. As this data only has a limited scope, the estimates of reliability parameters are associated with more or less large uncertainties. In order to calculate how the uncertainties of the reliability parameters affect the results of the underlying model, it is necessary to quantify the uncertainties associated with the estimate. The quantification of the uncertainties is expressed in the form of a probability distribution.

With the methods implemented in SUSA, generic or system-specific distributions of the following three reliability parameters of components of technical systems can be estimated:

– failure rates,

- probabilities of failure on demand, and
- repair rates.

The observations from the installations are given respectively by the number of failures in a given observation time, the number of failures for a given number of demands, or the number of repairs with the repair time required for the repairs.

The mathematical methods implemented are essentially based on Bayesian approaches. The available ‘generic’ observations (observations from other comparable plants or information from expert judgement) can be used as prior information (a-priori information) and modified accordingly by current observations from the specific plant of interest. The user thus has the option of including prior information in the generation of the distribution and receives a probability distribution as a result that describes the updated state of knowledge with regard to the parameter of interest.

The information to be used for the calculation can be either data from a specific plant and/or observations from other, but comparable plants or expert judgement. Here, expert judgement refers to the knowledge of quantile data of the reliability distribution, where a maximum entropy approach is used to combine such prior information. If such expert knowledge is not available, Bayesian approaches are used to combine the prior information with a specific plant. Depending on the type of prior information, different approaches are available to arrive at a suitable prior distribution.

In the case of ‘diffuse’ knowledge, i.e. no available prior information, the non-informative prior distribution is used. For the determination of plant-specific distributions with prior information, a specific posterior distribution with either a mixed distribution as prior or with the superpopulation approach as prior, which corresponds to the posterior distribution with unconditional generic distribution, can be used. The derivation of the two-stage Bayesian approach is explained in /PES 97/, while the derivations of the other approaches are described in /PES 95/.

2.2.1 Data and Model Assumptions

This section describes the data on which the distribution estimates are based and the model assumptions for the respective reliability parameters.

2.2.1.1 Failure Rates

For the failure rates, the data consist of

- the number *k* of observed failures of a component and
- the observation time or operational time *T* of the component.

Although only one component is referred to in this section, the data can also relate to several identical components. The observation or operating times refer to the time unit defined by the user, e.g., hours, days, years. The starting point for the derivations is the underlying probability model for failure rates per time unit. The probability that *k* failures occur in an observation time of *T* hours follows a Poisson distribution with the density function

$$\mathrm{p}(k, T|\lambda) = \frac{T^k}{k!}\lambda^k e^{-\lambda T}, \quad \lambda > 0 \tag{2.13}$$

As shown in equation (2.13), the probabilities depend on the value of the unknown parameter λ. The parameter λ denotes the failure rate to be estimated for the component or system unit of interest.

2.2.1.2 Failure Probability per Demand

For failure probabilities per demand, the data consist of

- the number *k* of observed failures and
- the number *D* of requirements

of the component or component group. Assuming that the event ‘failure´ occurs with identical probability π for each request of a component (Bernoulli test), the probability for *k* failures for a given number of component requests D can be determined from a binomial distribution with the density function

$$\mathrm{p}(k, D|\pi) = \binom{D}{k}\pi^k \cdot (1-\pi)^{D-k}, \quad 0 \leq \pi \leq 1 \tag{2.14}$$

The probability depends on the value of the unknown parameter π to be estimated. π denotes the probability of failure of the component or system unit of interest per demand.

2.2.1.3 Repair Rate

For repair rates, the data consist of

- the number *k* of repairs observed and
- the sum of the repair times

of the component or component group.

The underlying probability model assumes that the repair times *T* are independent and identically exponentially distributed. The probability that *k* repairs are carried out for a total repair time of *T* hours can be calculated using a Poisson distribution:

$$p(k,T|\lambda) = \frac{T^k}{k!}\lambda^k e^{-\lambda T}, \quad \lambda > 0 \tag{2.15}$$

where λ is the repair rate to be estimated.

2.2.1.4 Application of the Bayesian Method for Estimating Distributions

The methods used are largely based on the Bayesian method, which is briefly described below.

The aim is to express the level of knowledge of the parameters λ (or π) quantitatively in the form of a suitable probability distribution. For this purpose, λ (or π) is considered as a random variable. The Bayesian approach provides a method that allows the calculation of a corresponding probability distribution by including prior information about a parameter of interest. With regard to the failure rate λ the Bayesian formula can be written as

$$p(\lambda|k_0,T_0) = \frac{p(k_0,T_0|\lambda)\cdot p_0(\lambda)}{\int_0^\infty p(k_0,T_0|\lambda)\cdot p_0(\lambda)\mathrm{d}\lambda} \tag{2.16}$$

where $p_0(\lambda)$ denotes the prior distribution of λ and expresses the prior information that exists about λ before concrete observations from the specific plant are available.

The distribution density $p(\lambda|k,T)$, which reflects the state of knowledge of λ after plant-specific observations have been included in the calculations, is called posterior density. The integral in the denominator of the expression (2.15) extends over the entire range of values of λ and is used for normalisation so that $p\ (\lambda|k,T)$ becomes a distribution density. The conditional probability $p\ (\lambda|k,T)$, for the observation *(k, T)* under the condition that a certain value λ of the failure rate is given is referred to as the likelihood of *(k, T)* for a given λ. The likelihood contains the additional information about the uncertain reliability parameter λ obtained from the observation. In this respect, it plays a decisive role in Bayes' theorem as it updates the prior state of knowledge.

The descriptions also apply analogously to repair rates and failure probabilities per requirement.

The use of the Bayesian approach has various advantages over the frequentist approach:

- The result of the Bayesian method is a distribution for the parameter of interest (posterior distribution), which expresses the current level of knowledge or uncertainty about the parameter. In contrast, the frequentist estimate provides a confidence interval that contains the 'true' value of the parameter with a probability of 90 %, for example.
- Observations from other, comparable plants can be integrated into the process as preliminary information in a mathematically consistent manner. In the first step, a generic distribution is determined with regard to the data from the preliminary information. In the second step, this distribution is integrated into the Bayesian formula as a prior distribution and linked to the current data of a specific plant.
- If no failures have been registered in *T* hours of observation (so-called 0-error statistics), frequentist estimates will only produce unsatisfactory results. In contrast, Bayesian methods can be used to determine mathematically consistent distributions.

2.2.2 System-specific Distribution Using a Non-informative Prior

A frequent criticism of the practical applicability of Bayesian methods is associated with the selection of the prior distribution. In 'classical´ statistics, prior information is only accepted insofar as it is based on frequentist data. In order to prevent the criticism of a lack of objectivity, the concept of the non-informative prior distribution offers the possibility of

- avoiding a subjective assessment of the prior information as far as possible, apart from the need to accept the model assumptions and
- finding an objective prior distribution without frequentist data.

The concept of a non-informative prior distribution is based on the idea of choosing the prior distribution in such a way that the information content of the posterior distribution is determined as far as possible by the likelihood function of the observed current data. The derivation of a non-informative prior distribution of a parameter or parameter vector of interest is carried out according to the Jeffreys method /JEF 46/.

2.2.2.1 Failure Rate

Given the observation (k_0, T) of the specific plant, according to the model assumptions of Section 4.2.1.1 the distribution of the number of failures k_0 in the given observation period T follows a Poisson distribution with unknown parameter λ. Furthermore, it is assumed that no further information or at best little knowledge about the failure rate λ exists so that the non-informative prior distribution for the parameter of the Poisson distribution is used. According to the rule of Jeffreys /BOX 11/, the following applies:

$$p_0(\lambda) \propto J(\lambda)^{0.5}, \qquad \text{with } \mathrm{J}(\lambda) = \mathrm{E}\left(\frac{\partial^2 \log p(x|\lambda)}{\partial\lambda^2}\right) \tag{2.17}$$

For a Poisson distributed random variable x, this results in

$$\mathrm{J}(\lambda) = \mathrm{E}\left(\frac{x}{\lambda^2}\right) \tag{2.18}$$

Since the expected value for a Poisson distributed random variable is λ, this results in

$$\mathrm{J}(\lambda) = \frac{1}{\lambda} \tag{2.19}$$

For the non-informative prior of the parameter λ of the Poisson distribution, this results in

$$p_0(\lambda) \propto \lambda^{-0.5} \tag{2.20}$$

Using the Bayesian formula, this leads to

$$\mathrm{p}(\lambda|k_0, T_0) = \frac{\frac{T_0^{k_0}}{k_0!}\lambda^{k_0} e^{-\lambda T_0}\lambda^{-0.5}}{\int_0^\infty \frac{T_0^{k_0}}{k_0!}\lambda^{k_0} e^{-\lambda T_0}\lambda^{-0.5} d\lambda} = \frac{T^{k_0+0.5}}{\Gamma(k_0+0.5)}\lambda^{k_0-0.5} e^{-\lambda T_0} \tag{2.21}$$

The system-specific posterior distribution of the failure rate with non-informative prior is thus a gamma distribution with the parameters $k_0 + 0.5$ and T.

2.2.2.2 Failure Probability

The observations (k_0, D_0) of the specific plant are given and the modelling assumptions of Section 2.2.1.2 apply. Analogous to the treatment of the failure rates, it is assumed that no further information or at most little prior knowledge about the failure probability π exists, so that the non-informative prior distribution is used for the parameter of the binomial distribution. According to Jeffrey /BOX 11/, the non-informative prior is

$$p_0(\pi) \propto \pi^{-0.5}(1-\pi)^{-0.5} \tag{2.22}$$

Using Bayes' theorem, a beta distribution with the parameters $k_0 + 0.5$ and $D_0 - k_0 + 0.5$ is obtained as the system-specific posterior of the probability of failure per demand, with non-informative prior of the parameter π of the binomial distribution k, i.e.:

$$\mathrm{p}(\pi|k_0, D_0) = \frac{\Gamma(D_0)}{\Gamma(k_0+0.5)\Gamma(D_0-k_0+0.5)} \pi^{k_0-0.5}(1-\pi)^{D_0-k_0-0.5} \tag{2.23}$$

2.2.2.3 Repair Rate

The number of repair times k_0 and the total repair time T_0 of the specific system are given and the modelling assumptions of Section 2.2.1.3 apply. In analogy to the derivation for failure rates, the following is obtained for the repair rate as a non-informative prior of the parameter λ of the exponential distribution

$$p_0(\lambda) \propto \lambda^{-1} \tag{2.24}$$

Using Bayes' theorem and the non-informative prior of λ, this results in a gamma distribution with the parameters k_0 and T_0 as the plant-specific distribution of the repair rate, i.e.:

$$\mathrm{p}(\lambda|k_0, T_0) = \frac{T_0^{k_0}}{\Gamma(k_0)} \lambda^{k_0} e^{-\lambda T_0} \tag{2.25}$$

2.2.3 Mixed Distribution Approach

The mixed distribution approach /FRO 85/ is based on an unconditional (i.e. not subject to the condition of certain estimated values) generic distribution, which is estimated by a weighted sum of gamma (or beta) distributions determined from the observations of comparable plants. It also takes into account the influence of the uncertainties of the individual λ (π) on the estimation of the generic distribution.

2.2.3.1 Failure Rate

It is assumed that each individual plant has its own individual failure rate and the number of failures in each plant follows a Poisson distribution with the parameter λ_i (see Section 2.2.1.1). Furthermore, it is assumed that no further information is available about the respective random variables Λ_i, which would justify the use of an informative prior distribution for Λ_i. For each individual plant i ($i = 1, \ldots, n$), the posterior of the respective Λ_i is therefore calculated on the basis of the non-informative prior distribution for Λ_i, together with the observation (k_i, T_i), whereby a $\Gamma_{k+0.5,T}$ distribution is obtained.

Since each of the available observations should receive the same weighting (and therefore also the determined $\Gamma_{k+0.5,T}$ distributions), the mixed distribution approach provides the arithmetic mean of the n calculated gamma distributions as a generic distribution, i.e.:

$$\mathrm{p}(\lambda|(k_i, T_i)i = 1, \ldots, n) = \frac{1}{n}\sum_{i=1} n \frac{T_i^{k_i+0.5}}{\Gamma(k_i+0.5)} \lambda^{k_i-0.5}\, e^{-\lambda T_i} \qquad (2.26)$$

A λ_i can be drawn from each of the individual gamma distributions and the empirical distribution of these rate values can be formed as an estimate of the conditional generic distribution. The mixture of many such empirical distributions (i.e. their arithmetic mean) would be the estimate of the (unconditional) generic distribution obtained according to the above procedure.

If the determined mixed distribution is used as the prior distribution for the specific plant with k_0 failures in the observation period T_0, the following plant-specific poster distribution density is obtained:

$$p(\lambda|k_0,T_0) = \frac{\frac{T_0^{k_0}}{k_0!}\lambda^{K_0}e^{-\lambda T_0}\cdot\frac{1}{n}\sum_{i=1}^{n}\frac{T_i^{k_i+0.5}}{\Gamma(k_i+0.5)}\lambda_i^{k_i-0.5}e^{-\lambda T_i}}{\int_0^\infty\frac{T_0^{k_0}}{k_0!}\lambda^{k_0}e^{-\lambda T_0}\cdot\frac{1}{n}\sum_{i=1}^{n}\frac{T_i^{k_i+0.5}}{\Gamma(k_i+0.5)}\lambda^{k_i-0.5}e^{-\lambda T_{i\mathrm{d}}}\lambda} =$$

$$\frac{1}{A}\sum 1i = 1^n\Gamma_{k_0+k_i+0.5,T_0+T_i} \tag{2.27}$$

with the weights

$$A = \sum i = 1^n A_i = \sum_{i=1}^{n}\frac{T_i^{k_i+0.5}}{\Gamma(k_i+0.5)}\frac{\Gamma(k_0+k_i+0.5)}{(T_0+T_i)^{k_0+k_i+0.5}} \tag{2.28}$$

and

$$\Gamma_{k_0+k_i+0.5,T_0+T_i} = \frac{(T_0+T_i)^{k_0+k_i+0.5}}{\Gamma(k_0+k_i+0.5)}\lambda^{k_0+k_i-0.5}e^{-\lambda(T_0+T_i)} \tag{2.29}$$

The plant-specific distribution with prior information determined by the mixed distribution approach is therefore a weighted average of gamma distributions, where the weights are given by $\frac{A_i}{A}$.

2.2.3.2 Failure Probability

The underlying ideas correspond to those for failure rates, whereby the modelling assumptions of Section 2.2.1.2 apply and the non-informative prior of the parameter *π* of the binomial distribution is used. The following unconditional generic density function for Π is obtained analogously to the procedure described above for failure rates:

$$p(\pi|k_i,D_i) = \frac{1}{n}\sum_{i=1}^{n}\frac{\Gamma(D_i+1)}{\Gamma(k_i+0.5)\Gamma(D_i-k_i+0.5)}\pi^{k_i-0.5}(1-\pi)^{D_i-k_i-0.5} \tag{2.30}$$

If this generic density function (it is the arithmetic mean of *n* beta distribution densities) is used as a prior, the following posterior distribution density is obtained with the observation (k_0, T_0) from the specific plant:

$$p(\pi|k_0,D_0) = \frac{1}{A}\sum A_i B\big(k_0+k_i+0.5,(D_0+D_i-(k_0+k_i)+0.5)\big) \tag{2.31}$$

with

$$A_i = \frac{B(D_0+D_i-(k_0+k_i)+0.5, k_0+k_i+0.5)}{B(D_i-k_i+0.5, k_i+0.5)} \tag{2.32}$$

The plant-specific distribution with prior information is thus a weighted average of beta distributions with the weights $\frac{A_i}{A}$.

2.2.3.3 Repair Rate

Analogously to the failure rate but with adjusted non-informative prior distribution, see Section 2.2.1.3, the following results for the repair rate

$$\mathrm{p}(\lambda|k_0, T_0) = \frac{1}{A}\sum_{i=1}^{n} A_i \Gamma_{k_0+k_i, T_0+T_i} \tag{2.33}$$

with the weights including the specific system data resulting in

$$A_i = \frac{T_i^{k_i}}{\Gamma(k_i)} \frac{\Gamma(k_0+k_i)}{(T_0+T_i)^{k_0+k_i}} \tag{2.34}$$

2.2.4 Superpopulation Approach

The situation is assumed that observations from various comparable plants are available, whereby a certain comparability in the operating and environmental conditions of the components in the various plants is given, but an identical underlying distribution of the reliability parameters of the components of the various plants cannot be assumed. In order to describe reality more appropriately, all components of the component group from each of the comparable plants are therefore assigned their common but plant-specific underlying distribution of the reliability parameters to be estimated.

The population of the component groups under consideration from an imaginary multitude of comparable systems, of which the actually existing systems can be understood as a random selection, is referred to as a superpopulation.

The aim is to derive a model that has the ability to utilise the observations from the available comparable plants in order to obtain an estimate of the unknown distribution of the reliability parameters in the superpopulation. The reliability parameter λ_i (π_i) to be estimated, which is common to all components of the component group in plant i, can then, as long as no plant-specific observations are available, be regarded as a realisation of a random sample from this distribution of the superpopulation.

2.2.4.1 Failure Rates

The probability model on which the observations (k_i, T_i) are based is a Poisson distribution (see Section 2.2.1.1) with fixed but unknown parameters λ_i. The λ_i are regarded as realisations of a random variable Λ, which follows the distribution in the superpopulation, i.e. the failure rate of the component group under consideration in the population of comparable plants.

The distribution model of the superpopulation is assumed to be a gamma distribution with the parameters α and β – referred to below as *Gamma*(α, β) – with the density function

$$p_\lambda(\lambda|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{\beta\lambda} \qquad (2.35)$$

A further specification of the assumed gamma distribution cannot be made as no direct information is available on the alpha and beta parameters.

The family of gamma distributions is sufficiently flexible and at the same time the gamma distribution is the conjugate distribution to the Poisson distribution of the observations, i.e. the Bayesian approach provides a posterior distribution based on a prior gamma distribution, which is also of the type of gamma distribution and whose parameters are not complicated to determine.

To estimate the parameters α, β, Bayes' theorem is used, starting from a non-informative prior for α, β. This has the property that the posterior distribution for α, β determined with Bayes theorem is essentially characterised by the observations from the *n* comparable plants. The expression

$$p_0(\alpha,\beta) \propto \alpha^{-0.5}\beta^{-1} \qquad (2.36)$$

is used as the non-informative prior $p_0(\alpha,\beta)$ for the parameters α and β of the gamma distribution, as described in /HOR 90/.

Under the assumptions listed above, the realisations λ_i of the *Gamma*(α, β) distributed random variable Λ, which are considered as a random sample from the sought-after distribution of the default rates of the superpopulation, have the likelihood function

$$\mathrm{L}(\alpha,\beta)|\lambda_1,\dots,\lambda_n) = \prod_{i=1}^{n} \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda_i^{\alpha-1}\, e^{-\beta\lambda_i} \qquad (2.37)$$

With the likelihood and the non-informative prior $p_0(\alpha, \beta)$, the two-dimensional density function p_1 of the gamma distribution parameters α and β can be determined, using Bayes' theorem

$$p_1(\alpha,\beta|\lambda_1,\dots\lambda_n) = \frac{L(\alpha,\beta|\lambda_1,\dots,\lambda_n)p_0(\alpha,\beta)}{\int_0^\infty\int_0^\infty L(\alpha,\beta|\lambda_1,\dots,\lambda_n)p_0(\alpha,\beta)\mathrm{d}\alpha\mathrm{d}\beta} \qquad (2.38)$$

The denominator of the expression (2.38) serves as the normalisation constant, which is henceforth referred to as *C*, whereby p_1 becomes a density.

Furthermore, the underlying assumption that the failure rate is gamma-distributed over the superpopulation with unknown parameters α and β as well as the knowledge of the two-dimensional density function of the parameters α and β can be utilised. If the two density functions (2.35) and (2.38) are multiplied together and the parameters α and β are integrated out, an unconditional distribution $p^{\sim}$ of the failure rate lambda is obtained with respect to α and β, which, however, still depends on the selected values $\lambda_1, \dots, \lambda_n$.

The unconditional (i.e. not under the condition of a specially selected pair of parameter values (α, β)) generic distribution, depending on all λ_i, therefore has the density function

$$p^{\sim}(\lambda|\lambda_1,\dots,\lambda_n) = \int_0^\infty\int_0^\infty p_\lambda(\lambda|\alpha,\beta)\cdot p_1(\alpha,\beta|\lambda_1,\dots,\lambda_n)\mathrm{d}\alpha\beta$$

$$= \int_0^\infty\int_0^\infty \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\lambda\beta} \cdot \frac{\prod_{i=1}^{n}\frac{\beta^\alpha}{\Gamma(\alpha)}\lambda_i^{\alpha-1}e^{-\beta\lambda_i}\alpha^{-0.5}\beta^{-1}}{C}\mathrm{d}\alpha\mathrm{d}\beta \qquad (2.39)$$

Before the last step to generate the unconditional generic distribution of the failure rate λ is carried out, the meaning of the equation (2.39) should be discussed for a better understanding.

It is based on the idea that all theoretically possible values $\alpha*$, $\beta*$ with $\alpha* \in (0, \infty)$, $\beta* \in (0, \infty)$ are used for the parameters α and β. By combining all possible parameter values, all theoretically possible *Gamma*$(\alpha*, \beta*)$ distributions are taken into account to determine the failure rate distribution of interest.

Each individual gamma distribution is assigned the weight according to the density function p_1 over the parameter pair (α, β). The gamma distributions whose parameter combinations (α, β) have the highest density values in the two-dimensional density function are assigned the greatest weight. However, the largest density values in equation (2.38) are those parameter combinations (α, β) that receive the largest likelihood given the selected $\lambda_1, \dots, \lambda_n$ (maximum likelihood estimates). Since the density distribution over α and β depends on the realisations λ_i, the gamma distributions that best correspond to the selected λ_i are assigned the most importance.

The integration carried out in equation (2.39) over all theoretically possible parameter values results in a mixture of all conceivable gamma distributions, whereby the mixture weight of the individual gamma distributions depends on how well they fit the individual realisations λ_i fit.

So far, only the sample $\lambda_1, \dots, \lambda_n$, which are realisations of randomly selected failure rates from the distribution of the superpopulation, have been discussed. However, the available data are not the randomly selected default rates mentioned so far, but the observation pairs $(k_1, T_1), \dots, (k_n, T_n)$ from the various plants.

According to Section 2.2.1.1, the probability for the occurrence of k_i failures in the observation time T_i in plant i follows a Poisson distribution with the parameter λ_i. Since it is assumed that the observations are independent of each other, the joint density of the observations (k_i, T_i) is given by

$$\mathrm{p}(k_1|\lambda_1, T_1) \cdot \ldots \cdot p(k_n|\lambda_n, T_n) = \prod_{i=1}^{n} \frac{(\lambda_i T_i)^{k_i}}{k_i!} e^{-\lambda_i T_i} \tag{2.40}$$

The density value of a Poisson distribution for observation (k_i, T_i) and the parameter λ_i is proportional to the gamma distribution density at λ_i with the parameters $\alpha = k_i + 1$ and $\beta = T_i$:

$$\mathrm{p}(k_i|\lambda_i, T_i) = \frac{(\lambda_i T_i)^{k_i}}{k_i!} e^{-\lambda_i T_i} = \frac{T_i^{k_i}}{\Gamma(k_i+1)} \lambda_i^{k_i} e^{-\lambda_i T_i}$$
$$= \frac{1}{T_i} \frac{T_i^{k+1}}{\Gamma(k_i+1)} \lambda_i^{k_i} e^{-\lambda_i T_i} = \mathrm{Gamma}(k_i+1, T_i) = \frac{1}{T_i} \mathrm{p}(\lambda_i|k_i+1, T_i) \tag{2.41}$$

The joint density of the observations in Expression (2.40) can be expressed by equation (2.41) as a likelihood function of all λ_i and can be expressed according to:

$$L^{\sim}(\lambda_1, \ldots, \lambda_n|(k_i, T_i)i = 1, \ldots, n) =$$
$$\prod_{i=1}^{n} \frac{1}{T_i} \mathrm{p}(\lambda_i|k_i+1, T_i) \propto \prod_{i=1}^{n} Gamma(k_i+1, T_i) \tag{2.42}$$

Using equations (2.39) and (2.42), the following density function is obtained:

$$p^*\left(\lambda\middle|(k_i, T_i)_{i=1,\ldots,n}\right) =$$
$$\frac{\int_0^\infty \cdots \int_0^\infty p^{\sim}(\lambda|\lambda_1, \ldots, \lambda_n) \cdot L^{\sim}\left(\lambda_1, \ldots, \lambda_n\middle|(k_i, T_i)i = 1, \ldots, n\right) \mathrm{d}\lambda_1, \ldots, \mathrm{d}\lambda_n}{\int_0^\infty \int_0^\infty \cdots \int_0^\infty p^{\sim}(\lambda|\lambda_1, \ldots, \lambda_n) \cdot L^{\sim}\left(\lambda_1, \ldots, \lambda_n\middle|(k_i, T_i)i = 1, \ldots, n\right) \mathrm{d}\lambda \mathrm{d}\lambda_1, \ldots, \mathrm{d}\lambda_n} \tag{2.43}$$

Since a double integral already has to be solved in equation (2.39) and equation (2.43) additionally has to be integrated over all λ_i, an analytical solution of equation (2.43) is out of the question.

An approximate solution of the integration over all λ_i is available using Monte Carlo simulation. For this purpose, a sample of size *s* is drawn from each *Gamma(k_i +1, T_i)* distribution. This provides the sample *(λ_i(1), …, λ_i(s))* from the density function $\frac{T_i^{k_i+1}}{T(k_i+1)} \lambda_i^{k_i} e^{-\lambda_i T}$ given by the observation. If the sample values *(λ_i(1), …, λ_i(s))* are inserted into the Expression $p^{\sim}$ in (2.39) for each simulation run *j*, the result is a function of lambda

$$g_j(\lambda) = p^{\sim}\left(\lambda\middle|\lambda_{1,(j)}, \ldots, \lambda_{\{n,(j)\}}\right\}, \text{with } j = 1, \ldots, s \tag{2.44}$$

With the values determined in equation (2.44), the unconditional generic distribution can be approximated by the following expression:

$$p*(\lambda) \sim \frac{1}{s}\sum_{j=1}^{s} g_j(\lambda) \tag{2.45}$$

To determine the Q % quantile x_Q, the equation

$$\frac{1}{s}\sum_{j=1}^{s}\int_0^{x_Q} g_j(\lambda)\mathrm{d}\,\lambda = \frac{Q}{100} \tag{2.46}$$

has to be solved with respect to x_Q.

Sample calculations have shown that the double integration to be carried out using the parameters α and β can take a relatively long time to calculate. However, the integral over β can be solved analytically.

The formula

$$\begin{aligned} g_j(\lambda) &= \frac{1}{C}\int_0^{\infty}\int_0^{\infty}\frac{\beta^{\alpha}}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\lambda\beta}\prod_{i=1}^{n}\left(\frac{\beta}{\Gamma(\alpha)}\lambda_{i,(j)}^{\alpha-1}e^{-\lambda_{i,(j)}\beta}\right)\alpha^{-0.5}\beta^{-1}\mathrm{d}\,\beta\mathrm{d}\alpha = \\ &\frac{1}{C}\int_0^{\infty}\left(\frac{1}{\Gamma(\alpha)}\right)^{n+1}\left(\lambda\cdot\lambda_{1,(j)}\cdot\ldots\cdot\lambda_{n,(j)}\right)^{\alpha-1}\alpha^{-0.5}\int_0^{\infty}\frac{\beta^{(n+1)\alpha-1}}{e^{\left(\lambda+\lambda_{1,(j)}+\cdots+\lambda_{n,(j)}\right)\beta}}\mathrm{d}\,\beta\mathrm{d}\alpha \end{aligned} \tag{2.47}$$

with

$$\int_0^{\infty} x_n e^{-\mu x}\mathrm{d}\,x = \frac{\Gamma(n+1)}{\mu^{n+1}} \tag{2.48}$$

and

$$\int_0^{\infty}\frac{\beta^{(n+1)\alpha-1}}{e^{\left(\lambda+\lambda_{1,(j)}+\cdots+\lambda_{n,(j)}\right)\beta}}\mathrm{d}\beta = \frac{\Gamma((n+1)\alpha)}{\left(\lambda+\lambda_{1,(j)}+\cdots+\lambda_{n,(j)}\right)^{(n+1)\alpha}} \tag{2.49}$$

can be transformed into

$$g_j(\lambda) = \frac{1}{C}\int_0^{\infty}\left(\frac{1}{\Gamma(\phi(x))}\right)^{n+1}\left(\lambda\cdot\lambda_{1,(j)}\cdot\ldots\cdots\lambda_{n,(j)}\right)^{\phi(x)-1}\phi(x)^{-0.5} \tag{2.50}$$

2.2.5 Description of the User Input

The AURA method has been integrated into SUSA as a module. It provides different classes and functions to calculate the desired failure rates/probabilities and repair rates.

The methods can be integrated into a program as a Python library or used directly in SUSA to include the distributions of the reliability parameters. The different reliability parameters can be used both via command line interface or scripts and in Jupyter note-books. With the help of tuples of plant-specific data as well as the inclusion of compara-ble plant data as preliminary information, these can be applied.

To determine the distribution of Failure Rate, the class '*FailureRate*' can be called. To determine the probability of failure per request, the class '*FailureProbability*' can be called. The class '*RepairRate*' can be used to determine the repair rate.

2.2.5.1 Application of Non-informative Prior Information

The method *non-informative_prior()* can be used to determine the failure rate without including prior information, i.e. only with system-specific data and with a generic Jeffrey Prior as prior. An exemplary call could look like this:

```
NFails = 10
x = 10000
distribution = FailureRate.noninformative_prior(NFails , x),
```

where *x* can be either the observation time or the number of demands. In contrast, if the repair rate should be determined, a call like the following can be used:

```
NRepairs = 10
RepairTime = 10000
distribution = RepairRate.noninformative_prior( NRepairs , RepairTime ).
```

2.2.5.2 Application of Mixed Distribution Approach

The mixed distribution approach mixes the information provided by the prior and the new data to generate the posterior information. The user provides the prior data in the form of a two-dimensional array or as list, either with dimension (n,2) or (n,3) where n is the number of reference plant data used as prior. Each line contains the information of a reference plant structured as tuple/list with information (*n_fails*, *observation_time*, *weight*) or (*n_fails*, *n_demands*, *weight*) or (*n_repairs*, *total_repair_time*, *weight*) if an (n,2) array is given, equal weights are assumed. Optionally, data of a specific plant can be provided by the user in the form of a tuple of two entries structured as (*n_fails*, *obs_time*), where *n_fails* is the number of failures in the observation time and *obs_time* is the observation time.

As for the non-informative prior approach described above, it is possible to either determine the failure rate or the repair rate using the corresponding class objects. One exemplary call to determine the failure rate given prior information and specific data could look like this:

```
priorInf = [ [1, 10530] , [0, 9460] , [2, 11300] , [0, 8760] ]
specData = [0 ,1000]
distribution = FailureRate.mixed_distribution( priorInf , specData )
```

2.3 Calling FORTRAN-based SUSA Sampling Modules Using a Generic Interface

Four different FORTRAN-based SUSA components currently form the basis for the SUSA sampling functionality. MEDUSA is the baseline sampling module in SUSA, it allows sampling values from parameter distributions based on the distributions provided and potential dependencies between the parameters. DIVIS /KLO 91/ helps the user to determine which distribution function is suitable for his or her needs depending on information provided by the user. This information could be given in form of one or more quantiles, the median and a factor, or the expected value and the standard deviation. Based on this information, several potential distributions are proposed by DIVIS. BetaFit provides the user with the means to exchange log-normal distributions in the parameter definition with matching beta distributions. Unlike log-normal distribution, a beta distribution has upper and lower bounds, which is often more suitable to describe the epistemic uncertainties modelled with SUSA.

One of the project goals is to present one Python-based interface to the user but still provide compatibility with the original results gained by running the FORTRAN-based SUSA codes. In order to achieve this goal, a Python-based interface has to be designed which internally calls the different FORTRAN-based SUSA codes MEDUSA (Sampling), DIVIS (Distribution Finding) and BetaFit (Fitting Beta Distribution to Log Normal Distributions). This development is the requisite for the next step, translating the different

FORTRAN codes into Python and thus providing code in one programming language which significantly improves maintainability. The interface designed in this project will then be used run automatized tests comparing the results of the new Python codes to those of the FORTRAN-based codes, a necessary step towards providing the desired quality assurance of the SUSA results. In order to achieve the Python-FORTRAN interface, SUSA has been extended by several modules:

1. Modules for defining parameters and their distributions (*distributions.py*, *distribution_creator.py*).
2. Modules for defining parameter dependencies (*dependencies.py*, *linear_combinations.py*, *functional_combination.py*).
3. The *Controller (_controller.py)*, which inherits and extends the *FDEControllable* module of the generic FORTRAN interface FDE (*F*ORTRAN *D*evelopment *E*xtensions) /SCH 24/ for control and data exchange.
4. The *InputWriter (input_writer.py)*, a module to write the necessary input files for the FORTRAN-based SUSA functions.
5. The *InputWriterDivis* (*input_writer_divis.py*), a module specially adapted for the DIVIS functionality.

All three SUSA sampling components, MEDUSA, DIVIS and BetaFit, need as input a clear definition of the desired parameter distributions. The new SUSA distributions module can be used to create objects for the various potential distribution functions, which contain all necessary information to uniquely identify one distribution. Since all distribution classes inherit from the same base class, they share one common interface. The distributions provided are:

– Normal Distribution,
– Uniform Distribution,
– Beta Distribution,
– Chi2 Distribution,
– Discrete Distribution,
– Discrete Uniform Distribution,
– Exponential Distribution,

- Fisher Distribution,
- Frechet Distribution,
- Gamma Distribution,
- Geometric Distribution,
- Gumbel Distribution,
- Rayleigh Distribution,
- Trapez Distribution,
- Triangular Distribution,
- Log Normal Distribution,
- Log Uniform Distribution,
- Log Triangular Distribution,
- Negative Binominal Distribution,
- Polygonal Distribution,
- Weibull Distribution,
- Histogram,
- Log Histogram.

These distributions are based on the corresponding distributions defined in the *sklearn* Python library. The distribution classes implemented in SUSA extend the functionalities of the corresponding *sklearn* classes in order to suit the needs of the SUSA applications. For example, in accordance with the classic SUSA implementation, the possibility to truncate the different distributions has been implemented for most distributions.

In addition to the parameter definitions, which are based on the distribution classes, MEDUSA also needs the dependencies between the different parameters as input. In order to provide the parameter dependencies in a suitable form, a new dependencies module has been implemented in SUSA. In this module, the dependency base class as well as specific dependency classes inheriting from this base class are implemented. All objects of these classes implement a function called *calc_sample*, which returns an array of sampled values for which the specific dependency has been taken into account. The attributes of this function are just the number of required samples and the random number generator to be used. All dependencies available in the classic SUSA software have also been implemented in the Python-based SUSA version.

The new SUSA Controller module (*controller.py*) provides a function *medusa_calc* which encapsulates the FORTRAN-based MEDUSA code. The new SUSA Controller class extends the *FDEControllable* class of the generic interface FDE /SCH 24/ for control and data exchange. The *FDEControllable* reads a dynamic linked library (DLL) of the FORTRAN program which should be made callable from Python and provides a handle to the different functions of this FORTRAN program. In the case of SUSA, this FORTRAN program provides functions to access the different SUSA components. The SUSA Controller module extends the *FDEControllable* functionality by providing additional functions which act as convenient wrapper surrounding the FORTRAN-based SUSA functions and allow to perform all steps in the workflow of the SUSA components with just one function call. These additional functions write the necessary input files for the FORTRAN-based SUSA functions, pass these input files as attribute to the functions and use the generated output to write the input files for the next steps in the workflow. In order to sample parameters using the MEDUSA functionality, the user provides a Python random sampler and the number of desired samples as input parameter to the controller function *medusa_calc*; optionally, the user can also provide dependencies between the parameter-input and output directory , if existent, for the generated files, if the current directory should not be used, and the decision on the output files to be generated. The FORTRAN-based MEDUSA component is called by the Python code and produces as output an ASCII-based design file which includes the sampled parameter values. This file is read in by the controller and the sampled values are returned to the user as NumPy array.

In the case of DIVIS, the user provides an object of the class *DistributionCreator* to the Controller function *divis_calc*. The *DistributionCreator* prepares the necessary input for the DIVIS FORTRAN program based on the user input, this includes transforming the provided input parameters into those required by the DIVIS FORTRAN program. The *InputWriterDivis* fills the information provided by the *DistributionCreator* into a prepared DIVIS template. The *DistributionCreator* in turn reads the information generated by the DIVIS FORTRAN program and creates an object of one of the distribution classes listed above based on this information.

In the case of BetaFit, the SUSA controller module provides a function *beta_fit*, which encapsulates the corresponding FORTRAN. The user passes the original log-normal distribution as attribute into this function call, additionally the so-called fit criterion can be provided. The fit criterion specifies how the beta function should be fitted to the provided log-normal function, either by optimizing the agreement of mean and standard deviation of the target and the origin distribution or by choosing the distribution which best fits the quantiles of the log-normal distribution, either for two or three quantiles.

3 Methods for Advanced Monte Carlo Simulation with Machine Learning Algorithms

The consideration of uncertainties in safety analyses can be achieved within the framework of BEPU approaches using Monte Carlo (MC) simulations. For each simulation run, a set of the uncertain input parameters is sampled according to the uncertainty distributions including their dependencies, which get applied to a deterministic simulation code. SUSA is an established software for uncertainty and sensitivity analyses, covering such BEPU analyses from parameter sampling to simulation and statistical evaluation, e.g. of tolerance intervals. A classic MC sampling approach becomes resource intensive for analyses focused on the evaluation of rare events. These events can typically only be reached from small regions of the input parameter space. A large number of simulations would be required to identify the region of interest in the input parameter space and accurately quantify the probability of the rare event. To perform probabilistic evaluations of rare scenarios with reasonable computational effort, adaptive sampling techniques can be used. Thereby, machine learning algorithms are used to iteratively adapt the sampling range of input parameters to those that most effectively increase the robustness and accuracy of the probabilistic evaluation. In the frame of the last SUSA project RS1559 /KLO 21a/, two adaptive sampling approaches were implemented in SUSA. One approach uses a support vector regression metamodel in the context of a subset simulation and the other approach uses a combination of a genetic adaptive sampling algorithm with an ensemble of classification algorithms. Both algorithms have been compared, discussing the advantages of both algorithms while getting applied to benchmark examples as well as to an accident scenario in a nuclear power plant. This benchmarking is described in Section 3.1.

In addition, two other methods have been implemented to get the necessary metamodel to be provided to the subset simulation. First, a support vector classification approach, which can be used if the target is not defined in a continuous parameter space but rather as a discrete variable, for example in order to answer the question if there is an entry of cooling liquid in the sump or not. The second method for generating metamodels which has been added to SUSA are flat neural networks. Both approaches have been implemented and tested. The implementation and test of both approaches is described in Section 3.2

In addition to these new methods for generating metamodels, methods have been implemented in SUSA to explain and use the results generated, using machine learning approaches. One is the usage of Shapley values in order to better understand the results of a SUSA analysis and to identify the most important influencing factors. The implementation and application of Shapley values is described in Section 3.3.

Section 3.4 finally details how the results of the implemented adaptive sampling methods can be used to derive a kernel density estimation for a later importance sampling application. In this way, the derived knowledge can be used in later applications to optimize the sampling efficiency in the critical parameter space.

3.1 Benchmarking of the Developed Iterative Methods for an Adaptive Monte Carlo Simulation

Adaptive sampling algorithms have been developed to reduce the number of samples required for analyses of rare events by iteratively adapting the sample range to the analysis objectives. Given a dataset of sampled input parameter sets, e.g., from a classic MC sample, with the corresponding results of their simulation runs, the idea of adaptive sampling methods is to iteratively train and refine a metamodel, e.g., a machine learning algorithm, to predict the simulation outcomes of interest until a precise computation of the desired analysis target (e.g. probability of the rare event) is achieved. At each iteration, the metamodel is refined by identifying regions of the input parameter space that are either most promising to lead to the rare event of interest or that have the greatest predictive uncertainties. Simulation runs are performed using the most promising parameter sets from the identified regions, expanding the training data set for the metamodels. With this approach, the metamodels are trained with a minimal number of simulation runs while providing accurate estimates of the analysis targets. To identify promising candidate parameter samples, there is a trade-off between exploring the unknown parameter space and tending to predict results near the desired region of interest (rare event). In both cases, a metric or distance measure must be defined in the multidimensional parameter space- or metamodel-related predictions to sort the candidates and evaluate how promising they are. Since the desired region of interest may also depend on multiple simulation outcome variables, this approach is also applicable to identify possible input parameter sets that lead to a combined rare event.

For simplicity, the desired region of interest in the benchmark application depends on only a single simulation outcome variable. This also simplifies the metamodels, which only need to predict the final state of a single simulation variable. However, in general, metamodels can also be trained to predict multiple parameters of the simulation outcome for more complex applications. Choosing how to build the metamodel and identifying the most promising parameter samples are the key challenges of adaptive sampling approaches and the main differences. The following subsections describe the different approaches implemented in SUSA that solve these challenges, including the adjustable termination criteria of the algorithms. The two algorithms that were compared in the benchmark are Subset Simulation with Support Vector Regression (SuSSVR) and the Genetic Adaptive Sampling Algorithm combined with the Probability Estimation using an Ensemble of Classification Algorithms (GASA-PRECLAS). Both are described in detail in /KLO 21a/.

3.1.1 General Idea of the Adaptive Sampling Approach

The general idea of the iterative approach of adaptive sampling can be described in the following steps:

1. The initial step is to create a training dataset by randomly sampling the uncertain input parameters according to their probability distributions and to run the simulations with these samples. Due to the long duration of the simulation runs, only a small set of samples, e.g. 20 to 50, should be created for efficiency reasons. Since this initial step is not an integral part of an adaptive sampling algorithm, the initial training dataset can alternatively be taken from a previous uncertainty analysis if the uncertain parameters are the same.
2. The training dataset created is used to train a single or multiple metamodels, i.e. machine learning algorithms, to predict the simulation result for the considered output quantity.
3. A large set of input parameter values is randomly sampled according to the probability distributions, e.g. 10^4 to 10^5 samples, depending on the applied adaptive sampling algorithm, but instead of running the simulations, the trained metamodel(s) are applied to this sample to predict the simulation results.
4. The predictions of the metamodel(s) are used to identify candidates of parameter combinations that are best suited to be added to the training dataset to improve the predictions of the metamodel(s), especially in the vicinity of the targeted parameter region. For these candidates, the results are calculated by the actual simulation code; therefore, only a few samples should be selected, e.g. 5 to 8 candidates.

5. Depending on predefined termination criteria, the algorithm either terminates and provides the estimated probability of the targeted region, i.e. probability of the undesired scenario, or the algorithm is repeated and returns to step 2, now with the enhanced training dataset.

3.1.2 Subset Simulation with Support Vector Regression

Subset Simulation (SuS) is a combined sampling and simulation approach for small failure probabilities described as the product of much larger conditional probabilities of intervening events approaching the actual failure event, while the conditional samples for each intervening event are sampled using Markov Chain Monte Carlo simulation /AU 01/, /PAP 15/. In the adaptive sampling algorithm implemented in SUSA, Support Vector Regression (SVR) is used as a metamodel within the Subset Simulation to predict the simulation outcome, as described in /KLO 20/. This SuSSVR algorithm consists of three iteration cycles.

In a first cycle, the algorithm is repeated until at least a certain percentage of the training dataset, e.g. 10 %, is in the desired region of interest. In a second cycle, the goal of the algorithm is to converge to a robust metamodel, i.e., a robust prediction of the SVR. Adjustable threshold parameters are given for the so-called switching rate, i.e., the mean fraction of parameter sets in the last subset sample that were classified differently in the last – say 5 – iterations and the mean rate of change of the rare event probabilities calculated in the last iterations. In both cycles, the new parameter candidates for runs with the actual simulation code and, thus, for the training dataset are selected from the last subset sample obtained in an iteration step (random or cluster-based selection). The last cycle is iterated using a larger subset sample size to obtain a robust rare event probability estimate. The number of iterations in the last cycle is ten or more to get information on the variation of the probability estimate due to the random sampling. Since no refinement of the metamodel is performed and thus no further simulation run with the actual simulation code is required, this cycle is comparatively fast.

3.1.3 GASA-PRECLAS Algorithm

The GASA-PRECLAS algorithm presented in /SOE 22/ consists of two iteration cycles and divides the sampling problem into two parts, each of which is solved using an optimized algorithm. First, the GASA algorithm is used to effectively explore the parameter space to obtain a training dataset with a certain number of samples in the region of interest, e.g., n = 5 samples. The GASA algorithm provides the training data so that the classifiers in the second cycle can distinguish between interesting and uninteresting events. Its aim is comparable to the first cycle of the SuSSVR algorithm. If there are multiple regions of interest that are separated from each other, this should also be taken into account when choosing the termination criterion of the GASA algorithm, i.e., the number of samples in the desired region of interest should be increased.

The second cycle uses a combination of classification algorithms as a metamodel to predict whether a parameter sample leads to a rare event. A Bayesian approach is used to calculate the probability distribution for the likelihood of the rare event based on a large parameter sample generated at each iteration and the corresponding predictions of the fitted classification metamodels. The adjustable termination criteria refer to the variation of the calculated probability distribution of the rare event over the last – say 5 – iterations.

The use of an ensemble of classification algorithms reduces the impact of the uncertainties of a single classification algorithm and the impact of an incorrect prediction. Combined with the Bayesian calculation of the probability distribution, an additional refinement loop for the probability estimate and its uncertainty – as in the SuSSVR algorithm – is not necessary. At each iteration step, the new parameter candidates for the actual simulation runs are selected from the large parameter sample which is also the basis for the estimation of the rare event probability. The selection of the candidates is based on criteria associated with the fitted metamodels and calculated for each element of the parameter sample.

3.1.4 Comparison of Application Examples

Two benchmark functions are used to analyse the implemented adaptive sampling methods. Both consist of a simple function that can be quickly evaluated. The first example with a biological dose model tests how a very small probability, e.g. about 1 E-06, can be estimated in a six-dimensional parameter space. The second example with the Ishigami function tests how to identify four separate regions in a strongly non-linear function. Although this is only a three-dimensional problem with a probability of about 1 E-03, finding all four maxima of this function is a difficult task that requires advanced sampling algorithms for proper likelihood estimation. Finally, a thermal-hydraulic code simulating a LOCA scenario in a nuclear power plant is used as a more realistic and complex application example where a single simulation run requires several hours. This example considers a high-dimensional parameter space (35 uncertain parameters) and demonstrates the need for an adaptive sampling algorithm, as a classic MC sampling would require at least 1 E +04 or more simulation runs, which is not feasible.

3.1.4.1 Benchmark Example: Biological Dose Model

In this example, it is assumed that during the normal operation of a nuclear power plant, small concentrations of radionuclides are released and enter the food chain of a population group. To calculate the maximum annual dose-equivalent of an individual of the population group, the following simple deterministic model is applied:

$$y = c \cdot (x_{rate1} \cdot x_{conc1} + x_{rate2} \cdot x_{conc2}) \cdot \exp(-0.2 \cdot dt) \tag{3.1}$$

The description of the parameters and their distributions is listed in Tab. 3.1.

Tab. 3.1 Uncertain parameters and their distributions for the biological dose model

Name	Symbol	Distribution	Distribution Parameter

Dose conversion parameter	C	Normal	Mean = 3.29 E-08
			Std = 1.11 E-08
			Min =1 .00 E-08, Max = 5.00 E-08

Consumption rate of meal 1	x_{rate1}	Log-Uniform	Min =1 0, Max =
Radio conc. in meal 1	x_{conc1}	Uniform	Min = 10, Ma x= 35

Consumption rate of meal 2	x_{rate2}	Log-Normal	Mean = 4.7552, std = 0.1993
			Min = 0.5, Max = 400

Radio conc. in meal 2	x_{rate2}	Uniform	Min = 10, Max = 30

Delay time	Dt	Triangular	Mode = 0.8, Min = 0.5
			Max = 20

The region of interest is defined by the maximum annual dose equivalent exceeding 0.25 mSv, which means that in equation 3.1 the result y exceeds 2.5 E-04. A simple MC sample of the formula with 1.0 E-08 samples identified 386 samples above the threshold. The resulting 95 % confidence interval using Clopper and Pearson /CLO 34/ is [3.48 E-06, 4.26 E-06]. Both algorithms start with an initial training pool of 50 samples using simple MC sampling. In addition, both algorithms use a maximum variation of probability calculation over the last four iterations of 0.1 for the second learning cycle, defining the robustness of the metamodel. The additional threshold for the switching rate of the

SuSSVR algorithm was set to 2.5 %. In order to accumulate enough rare events of interest in the training sample in the first learning cycle, a termination criterion of at least 10 % was used for the SuSSVR approach, while an absolute value of at least five rare events was set for the GASA-PRECLAS algorithm. The SuSSVR algorithm terminated after 165 additional calculations of the simulation function and estimated a probability for the region of interest within the interval [3.16 E-06, 4.19 E-06]. The GASA-PRECLAS algorithm terminated after 180 additional simulation runs, estimating a probability inside the interval [3.52 E-06, 7.05 E-06]. Both algorithms require almost the same number of additional calculations and estimate compatible intervals for the probability, which could mean that both algorithms pass this benchmark test without problems or further adjustments. However, the low probability of about 1.0 E-06 reveals limitations of the PRECLAS algorithm. Since this algorithm actually estimates the probability based on a simple random sample of parameter values and corresponding predictions of the metamodels, 1.0 E+08 sample elements are required for a robust probability estimate. Furthermore, to select new parameter candidates, all 1.0 E+08 parameter sample elements must be graded according to specific selection criteria. While this amount of data can still be processed with high computing power, applications with even smaller probabilities, thus producing larger amounts of data, introduces runtime and memory problems. In future developments, an advanced sampling method, such as importance sampling, should be introduced to solve this problem. These problems do not arise with the SuSSVR approach because the Subset Simulation does not need a large sample to predict a low probability.

3.1.4.2 Benchmark Example: Ishigami

The Ishigami function is often used for benchmarking advanced sampling methods and sensitivity indices. It is defined by the following formula:

$$y = \sin(x_1) + c_1 \cdot \sin^2(x_2) + c_2 \cdot x_3^4 \cdot \sin(x_1) \tag{3.2}$$

In this benchmark example, the GASA algorithm performs slightly better in exploring the uncertain parameter space than the first cycle of the SuSSVR algorithm. How the GASA algorithm targets the region of interest while exploring the rest of the parameter space is shown in Fig. 3.1, which illustrates the evolution of the training data set for the GASA-PRECLAS algorithm.

After the initial 50 samples using classic MC sampling (shown in blue in Fig. 3.1), the GASA algorithm (orange) required 240 additional function evaluations, and the PRECLAS algorithm (green) required 72 ones.

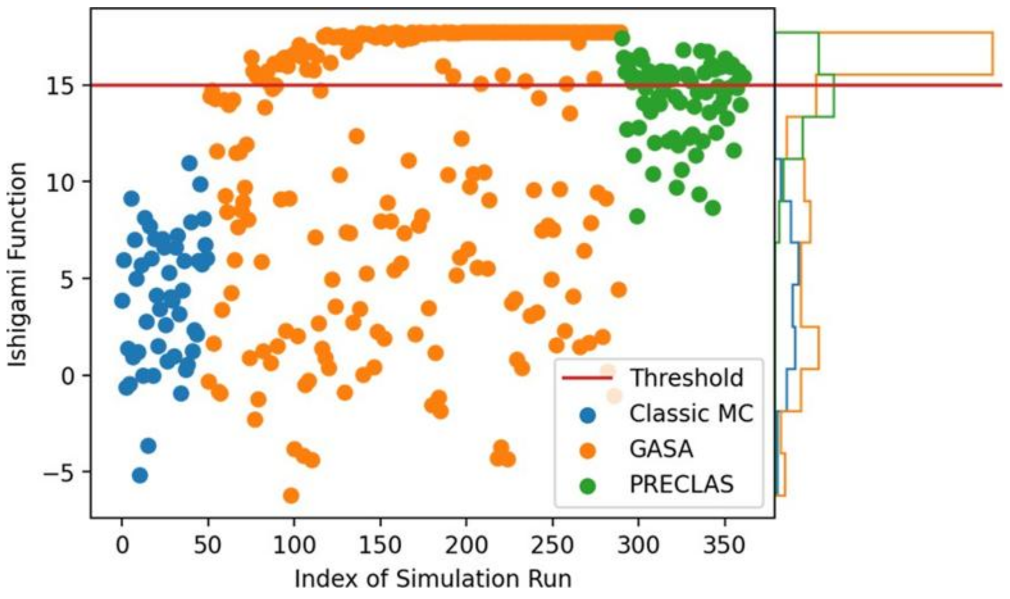


Fig. 3.1 Development of the Ishigami function evaluations during the adaptive sampling of the GASA-PRECLAS algorithm

However, the necessary adjustments in the SuSSVR and in the GASA algorithm show the limitations of both exploration algorithms. Without the increased number of training data in the desired region of interest, the GASA algorithm or the first cycle of the SuSSVR algorithm would not have found all four separated regions and would have underestimated the rare event probability. This can be explained by the fact that neither approach is designed to find multiple separated regions of interest. Only the higher statistics and thus higher probability of finding all maxima prevented an underestimation of the probability. This can be improved, e.g. by introducing prior knowledge about the four separate maxima or by adding an optional parameter that controls whether the new samples should be more biased towards one (already found) region of interest or towards the exploration of the unknown parameter space.

3.1.4.3 Application to a LOCA Scenario

In this example, the objective is to estimate the probability that the peak cladding temperature (PCT) exceeds 1200 °C in a LOCA scenario inside a nuclear power plant. A conservative reference model of a pressurized water reactor with four cooling circuits and an electrical power of 1425 MW is used, in which a double-ended guillotine rupture is initiated in the cold section of the main coolant line. The simulation model using ATHLET /WIE 19/ was developed based on previous analyses /KLO 16/, /POI 18/. 35 uncertain input parameters are considered in this work. Both algorithms start with an initial training data set of 50 samples and with the same termination criteria as the biological dose model. The SuSSVR approach terminated after 364 additional simulation runs and estimated a probability between [8.16 E-03, 9.36 E-03]. The GASA-PRECLAS approach terminated after 103 additional simulation runs with an estimated probability between [1.06 E+02, 2.07 E-02]. The GASA-PRECLAS algorithm converges earlier. Although the estimated probability intervals of the SuSSVR and GASA-PRECLAS algorithms do not overlap, the intervals are close, and their results are consistent. This shows that both algorithms work well also for high-dimensional parameter spaces with 35 dimensions. However, such high-dimensional parameter spaces make the sampling algorithms less and less efficient. Analogous to the biological dose model results, where an iterative refinement of the parameter space was suggested, an intermediate algorithm during the iteration of the adaptive sampling could reduce the dimensionality by analysing the influence of the uncertain parameters on the simulation results of interest and remove those that have low influence.

3.1.4.4 Conclusion

In this work, two adaptive sampling algorithms implemented in SUSA, the SuSSVR and GASA-PRECLAS algorithms, were compared to two benchmark applications and a LOCA scenario. In general, both approaches require the same order of magnitude of simulation runs to train a robust metamodel and compute a robust prediction for the rare scenario of interest. However, for runtime intensive calculations, already 100 additional simulation runs are expensive. But even when using the same algorithm but with a different seed, larger differences, e.g. of 100 additional simulation runs, can occur.

The probability intervals provided by the two approaches slightly differ. This can be explained by the fact that the parameters defining the termination criteria are not completely identical for the two algorithms. While the threshold for the variation of the probability is set to the same value, the switching rate is defined only in the SuSSVR algorithm. There is no such equivalent termination criterion in the PRECLAS algorithm, since this algorithm includes several classification algorithms that compensate for the uncertainty or switching points of the predictions of a single metamodel. However, it has been shown that the switching point is the most important parameter that determines the number of learning cycles required in the SuSSVR algorithm. For testing purposes, a switch point parameter was also implemented in the PRECLAS algorithm, but this did not lead to a subsequent termination of the iteration cycle. As for the first cycle in exploring the parameter space, both algorithms performed well when there is only a single contiguous target region. In such applications, an advanced parameter space exploration method is not required. However, if there are multiple target regions that are not connected to each other, the GASA algorithm performs slightly better and can play its advantage of effectively exploiting the parameter space over the first iteration cycle of the SuSSVR algorithm. As for the second cycle in building a robust metamodel for prediction, both algorithms perform well when the probability is not too small. However, for probabilities below 1 E-06, the simple MC sampling in the PRECLAS algorithm requires too many parameter samples and leads to runtime and memory problems. The Subset Simulation, which is used in the SuSSVR algorithm, already prevents such behaviour.

In summary, both implemented adaptive sampling algorithms are promising approaches for estimating the probability of a rare scenario. However, there is also room for improvements regarding the modularity of the algorithm. For some applications it might be useful combine the GASA algorithm with the second cycle of the SuSSVR algorithm. Furthermore, the sampling for the pool of parameter samples can be decoupled from the metamodel and decision process. In addition, an analysis of the relevant parameter space in terms of truncating an uncertain parameter distribution (importance sampling) or identifying irrelevant parameters would be beneficial to simplify the problem iteratively.

3.2 Subset Sampling with Flat Neural Networks

The SUSA adaptive learning suite has been extended so that it is now possible run the subset sampling algorithm and to use flat neural networks for metamodel generation. The neural network implementation in SUSA is based on the Pytorch library /PAZ 19/ for generating neural networks in Python.

In order to run the subset sampling algorithm with neural networks as metamodel generators, the layout of the neuronal network needs to be defined first. This can be done following the Pytorch API. One example is given in the code below for a network with six input nodes, two hidden layers, one output and a configurable number of nodes in the second hidden layer.

```
# pytorch includes
import torch .nn. functional as F
import torch .nn as nn

class PyTorchNN (nn. Module ):
   def __init__ (self , num_units = None ):
       super (). __init__ ()
       n_input = 3
       num_units = num_units or 13
       self.hid1 = nn.Linear( n_input, num_units )
       self.hid2 = nn.Linear( num_units, num_units )
       self.oupt = nn.Linear( num_units, 1)
       # initialize weights
       nn.init.xavier_uniform_( self.hid1.weight )
       nn.init.zeros_( self.hid1.bias )
       nn.init.xavier_uniform_( self.hid2.weight )
       nn.init.zeros_( self.hid2.bias )
       nn.init.xavier_uniform_( self.oupt.weight )
       nn.init.zeros_( self.oupt.bias )

   def forward (self , x):
       x = F.relu( self . hid1 (x))
       x = F.relu( self . hid2 (x))
       x = self.oupt (x)
       return x
```

The class *PyTorchNN* inherits from *nn.Module*, which is a base class for all neural network modules in PyTorch.

The following attributes of *PyTorchNN* are defined in the constructor:

- **Parameters:** The constructor takes an optional parameter *num_units*, which specifies the number of hidden units in the hidden layers. If not provided, it defaults to 13.
- **Input Size:** The network is designed to take an input of size 3, representing the three input parameter of the Ishigami function.
- **Layers:**
  - hid1: A linear transformation from six inputs to *num_units* hidden units.
  - hid2: Another linear transformation from *num_unit*s to *num_units*.
  - oupt: A linear transformation from *num_units* to a single output.

The weights of each layer are initialized using the Xavier uniform distribution, which is a common technique to help with convergence during training. The biases for each layer are initialized to zero. The method forward(self, x) defines how data flows through the network:

- Input *x* is passed through the first hidden layer, followed by a ReLU activation function (which introduces non-linearity).
- The output of the first layer passes the second hidden layer, again followed by a ReLU activation.
- Finally, the output from the second hidden layer goes to the output layer, which produces the final output.

Once the network layout has been defined, the neural network-based subset-sampling method can be called in the following way:

```
from python.AdaptMCS import WorkFlow, Transformer, Pool
from python.Simulation import Simulator
import torch as T

#%% Working directory
output_dir = 'C:\\Desktop\\test_Ishigami'

#%% Simulator
sim = Simulator.Ishigami()


#%% Initial producer of input sample
initProd = Pool.SimpleRandomSampler( distribution = sim._specific )
#%% Workflow
NN_broker = WorkFlow.SuSNNR( model = PyTorchNN (10),
                             optimizer = T.optim .Adam,
                             learning_rate = 0.0001,
                             loss = nn.MSELoss(),
                             epochs = 500,
                             max_learn_cycles = 100,
                             initproducer = initProd,
                             simulator = sim)

NN_broker.setFeatures( sim. _specific . keys ())

NN_broker.setScaler( input = Transformer.DistributionTransformer(
                                  distribution = sim. _specific ))
NN_broker.setTarget( dict(_y = [15 , 'upper ']) )
NN_broker.setBreakCriteria( nModels = 4,
                            switchFraction = 0.05,
                            ProbRate = 0.1 )
itr = iter ( NN_broker.learnCycle( nCandidate = 5,
                                   nPool = 10000,
                                   nInit = 50))
for c in itr:
   c.reportProgress( verbose = True )
```

This call to the subset sampling method makes use of the modules implemented and adapted in the research and development project RS1559: *WorkFlow, Transformer, Pool* and *Simulator*. In the *WorkFlow* module, the classes performing the different adaptive learning procedures are implemented. For performing the subset-sampling method with neural networks as metamodel generators, a new class has been added to the *WorkFlow* module, the *SuSNNR* class. The *SuSNNR* workflow class calls internally the *NNRMetaModelPyTorch* class which implements the neuronal network learning cycle. When training a neuronal network, each training step consists of one forward pass of the given parameter input through the neuronal net, a comparison with the expected output and, depending on the difference between the expected and realized output (the loss), the weights and biases of the network, defined above, are optimized in a so-called backward propagation pass.

The user can set several attributes of the *SuSNNR WorkFlow* object and the underlying *NNRMetaModelPyTorch* object. In the example above, the *WorkFlow* object is initialized using the following attributes:

- *model*: The neuronal network model to be used in the fitting process. In this example, this is the *PyTorchNN* model defined above with eight nodes in the second hidden layer.
- *optimizer*: A neuronal net optimizer is used to adapt the weights inside a neuronal net based on the provided training data in order to minimize the losses. The chosen Adam (adaptive momentum estimation) optimizer is one of the most popular gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients, similar to momentum, and also the decaying average of the past squared gradients.
- *learning_rate*: The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs, given the smaller changes made to the weights each update, whereas larger learning rates result in more rapid changed and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck. In combination with the Adam optimizer, the passed-in learning rate is the initial learning rate which gets adapted based on the decaying average of the past gradients and squared gradients.
- *loss*: The loss function is used to compare the predicted results obtained by one pass of input parameters through the neuronal network with those provided in the training

data. The larger the loss, the larger the needed correction to the network weights and biases. In the example above the mean squared error (MSE) is used as loss function.

- *epochs*: How often a neuronal net should be trained on the training data. It is possible to specify the so-called batch size (*batch_size*). In this case the neuronal network is trained in each epoch by iterating through separate parts of the training data set. The *batch_size* defines the size of the batches into which the training data is separated.
- *max_learn_cycles*: The maximum number of subset-sampling learning cycles to be performed.
- *initproducer*: The random sampler to be used. The random sampler is responsible for sampling the uncertain input parameters.
- *simulator*: The *simulator* generates target values from passed-in parameter values.

In the example, the distributions of all feature values are specified in *'sim._specific'* and passed to the *SimpleRandomSampler*. An object of the *SimpleRandomSampler* class is used to produce the initial feature values using a simple MC sampling. In the example, an object of the *BiometricDose* class is used as Simulator, meaning that the produced samples will follow the biometric dose distribution. Additionally, the user can set various attributes of the workflow, such as

- the names of the regarded uncertain input parameter (the features);
- the scaling method to be used: In the example, an object of the *DistributionTransformer* class of the *Transforme*r module is used as the provided input data is generated following the biometric dose distribution. Objects of the *DistributionTransformer* class are responsible for scaling and re-scaling the provided data according to the underlying distribution. Scaling is necessary as a lot of metamodel generators such as SVM depend on scaled input data.
- The target region: In the example above, the targets are result values larger than 15.
- The termination criteria for the subset sampling method: Two criteria define whether the iteration should be stopped. The first break criterion checks for each iteration step for a certain number of previously fitted metamodels *nModels* if the predicted number of events in the target region by those metamodels differs from the currently predicted number. The fraction of models for which such a discrepancy is found is compared to the value *switchFraction* defined in the break criteria. In addition, it is checked if

the rate in which the probability of events being in the target region changes from iteration to iteration is below the value of $ProbRate$ defined in the break criteria. If the actual switching fraction is lower or equal to the $switchFraction$ value in the break criteria and the change in the probability rate is lower or equal than $ProbRate$, the iteration process is terminated.

Finally, in the code above, an iterator for the learning cycle is generated, as described in Section 3.1.1 '$nInit$' defines how many sets of feature values (uncertain input parameters) are generated for the initial training step. The value of $nPool$ determines how many input parameter values are randomly sampled and then predicted using the trained metamodel(s) (step 3 as described in Section 3.1.1). The value $nCandidate$ defines how many best suited (according to the metamodel prediction) parameter value sets should be added to the training dataset of the metamodels (step 4 as described in Section 3.1.1). In order to understand how the prediction accuracy of the neuronal network changes over multiple cycles, both the training and test losses as well as the training and test accuracy are stored.

The new algorithm has been successfully applied to the Ishigami distribution. Fig. 3.2 shows the learning progress of the neuronal network over the learning cycles, the mean squared error decreases over the first 150 cycles and then remains almost constant.

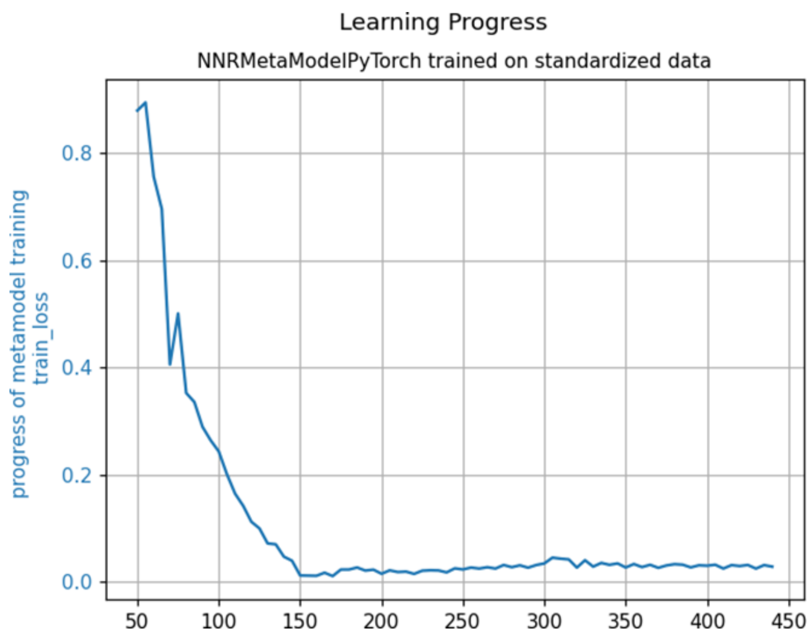


Fig. 3.2 Development of the model accuracy over the subset sampling cycles

Fig. 3.3 shows the parameter points sampled in the final subset sampling cycle. It is visible that in each minimum of the Ishigami Function, the density of sampled points is increased, as should be the case. The difference between predicted and observed output values is in some cases as large as 15, which is the limit value used for defining the target region.

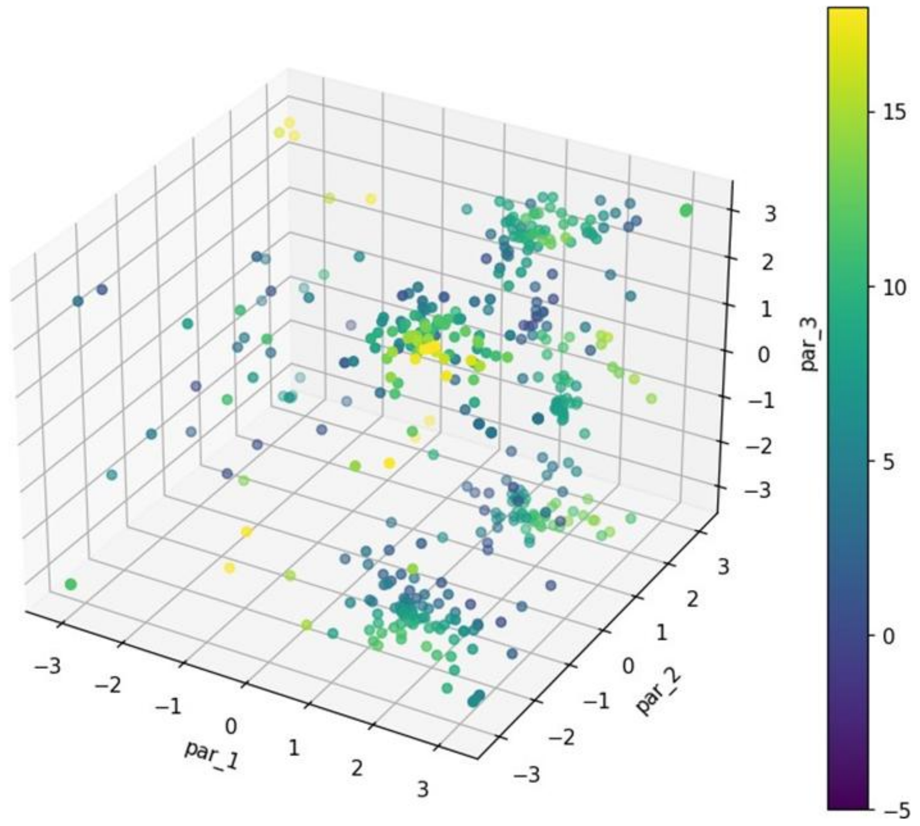


Fig. 3.3 Sampled parameter points in the final subset sampling cycle; the colour scale indicates the difference between predicted and observed output values

Fig. 3.4 shows how the observed result values of the Ishigami function develop as a function of the subset sampling cycles. In the presented example, 400 cycles are necessary for reaching the termination criteria defined above. This figure can be compared to Fig. 3.5, which shows the same plot created by using support vector regression for creating metamodels, with the following support vector regression specific parameter (kernel =' rbf', C = 100, epsilon = 0.1, gamma = 'scale'). It is clearly visible that for the examples given above, the usage of SVR instead of a flat neuronal network leads to a faster convergence towards the desired region. In the future, it could be studied in which circumstances (e.g., type of problem, hyperparameter optimization) a neuronal network as metamodel generator would bring advantages compared to the simpler SVR algorithm.

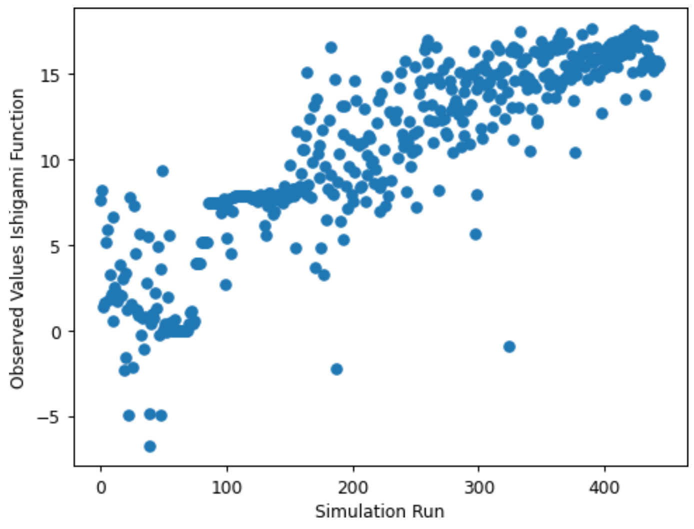


Fig. 3.4 Development of the observed values of the Ishigami function as a function of the simulation run (subset sampling cycle)

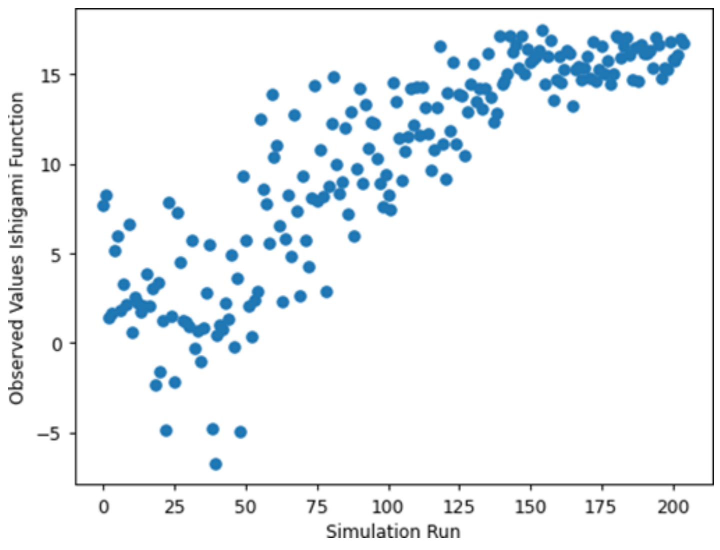


Fig. 3.5 Development of the observed values of the Ishigami function as a function of the simulation run (subset sampling cycle plus support vector regression)

3.3 Interpretability and Transparency of Machine Learning Algorithms

Shapley values are a concept from game theory that is used to determine the contribution of individual players to the total profit in a co-operative game. The Shapley value of a player *i* is calculated as the weighted average of the marginal contributions across all possible coalitions (subsets) of players. If *N* denotes the set of all players, then the marginal contribution of a player $i \in N$ to a given coalition $S \subseteq N$ is given by $v\,(S \cup \{i\}) - v(S)$, where *v* denotes the win function for a coalition. The empty coalition assigns *v* the value

0. The formula for calculating the Shapley value Shi for a player *i* ∈ *N* is given in equation 3.3. Here, *n* denotes the number of all players in the set *N* and ||*S*|| the number of players in the coalition *S*. The formula is based on the assumption that the players can enter the game in any conceivable order

$$Sh_i = \sum_{S \subseteq N \setminus \{i\}} \left(\frac{(n-1-|S|)!|S|!}{n!} \left(v(S \cup \{i\}) - v(S) \right) \right) \tag{3.3}$$

The application of Shapley values to machine learning models has recently gained in importance as they offer a theoretically sound approach to interpreting model results. In addition, Shapley values can quantify the contribution of individual variables (input parameters, features) to the model result. Bias and undesirable influences of individual variables can be identified and addressed. The interpretability of machine learning models is of central importance, especially in applications where trust and explainability are crucial. Shapley values offer a solution by providing a clear and consistent framework for explaining model results. They can help to overcome the ‘black box’ nature of complex models. The interpretability analysis of a machine learning model is performed for a given combination of values $x' = (x'_1, \ldots, x'_n)$ of the model variables, where $x'_1, \ldots, x'_n$ are considered as players that jointly contribute to the model outcome (profit). The expected marginal contribution of a subset $S \subseteq N = x'_1, \ldots, x'_n$, e.g. $S = x'_1, x'_2$, is defined by the function $v_{x'}(S)$ in the following equation 3.4:

$$v_{x'}(S) = v_{x'}(\{x'_1, x'_2\}) = \int g(x'_1, x'_2, X_3, \ldots, X_n) f!\,(x'_1, x'_2, X_3, \ldots, X_n)\ d\mathrm{X}_3 \ldots d\mathrm{X}_n - E\big(g(X)\big) \tag{3.4}$$

In equation 3.4, *g* denotes the function of the machine learning model, $X = (X_1 \ldots, X_n)$ the variable vector of the learning model *g*, *f* the multivariate density of *X*, and *E*(*g*(*X*)) the expected value of the model result *g*(*X*).

The exact calculation of the Shapley values for a large number n of variables can be very computationally intensive because the marginal contribution $v\ (S \cup \{x_i\}) - v(S)$ of x'_i, $I = 1, \ldots, n$, must be calculated for all possible subsets $S \subseteq N\ \{x'_i\}$ (equation (2). For this reason, an approximation using Monte Carlo simulation was proposed in /STR 14/. The Shapley value for a given variable value x'_i is estimated using the formula in equation 3.5:

$$Sh_i = \frac{1}{M} \sum_{m=1}^{M} \left(g(x_{+i}^m) - g(x_{-i}^m) \right) \tag{3.5}$$

In equation 3.4, M is the number of iterations for estimating ^{Sh_i} of the Shapley value (e.g. M = 1000) and g is the function of the machine learning model. x_{-i}^{m} is a randomly selected vector in which some of the variable values are replaced. Exactly which variable values are replaced by which values is specified by a randomly selected vector in which some of the variable values are replaced. Exactly which variable values are replaced by which values is specified by a randomly selected subset $S \subseteq N\{x'_i\}$. The vector x_{+i}^{m} is almost identical to x_{-i}^{m}. Only the value of the ith variable of x_{-i}^{m} is different and identical to the variable value x'_i. The Monte Carlo simulation for approximating the Shapley value for the variable value x'_i is outlined below:

Given are a vector $x' = (x'_1, \ldots, x'_n)$ and a data matrix X, with which the machine learning model g was trained. The following steps are carried out for each iteration run $m = 1, \ldots, M$:

1. Random selection of a vector z from the data matrix X.
2. Random selection of a subset S from the set N $x'i$.
3. Construction of two new vectors:
   - x_{-i}^{m}: Set $x_{-i}^{m} = \mathrm{z}$ and then replace the values of the corresponding variables with the values in S.
   - x_{+i}^{m}: Set $x_{+i}^{m} = x_{-i}^{m}$, and then replace the value of the ith variable with the value x'_i.
4. Calculate the marginal contribution: $\mathrm{g}(x_{+i}^{m}) - \mathrm{g}(x_{-i}^{m})$.

When all iteration steps have been carried out, the Shapley value Sh^i is calculated according to equation 3.5. To obtain the Shapley values for all variable values of the given vector $x' = (x'_1, \ldots, x'_n)$, the procedure must be repeated the corresponding number of times.

The algorithm described above was implemented in SUSA. In addition, SUSA offers the option of using the freely available Python library *shap*, which specialises in Shapley values. The functions provided by *shap* can be applied to various types of models, including neuronal networks, decision trees, and regression models.

Fig. 3.6 shows the Shapley values for the different input values of the biometric dose function (equation 3.1). The larger the absolute Shapley value, the larger the contribution. The sign of the Shapley value is negative if an increase in this value decreases the outcome and positive if it increases the outcome. In the shown example, the largest influence factor is clearly 'dt' which is negatively correlated to the result value.

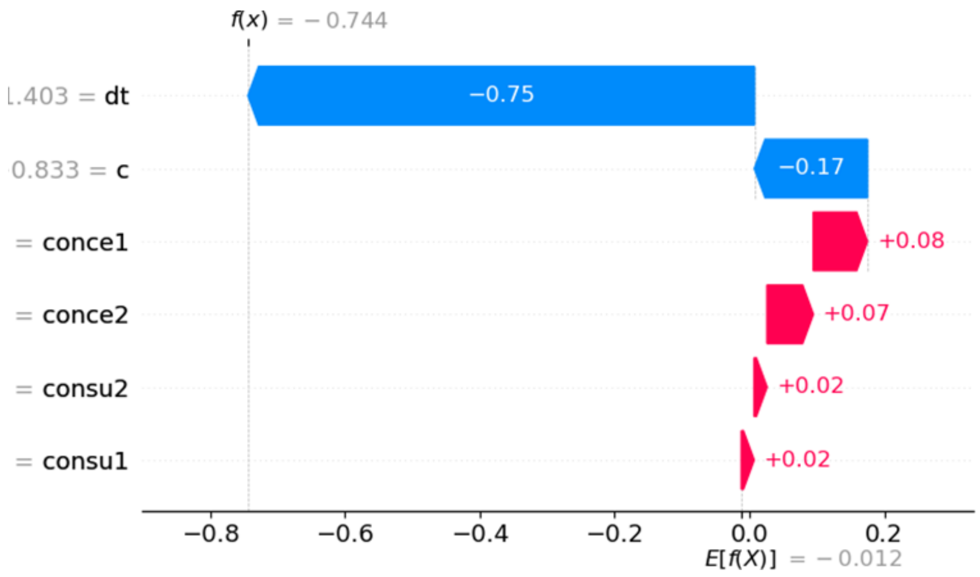


Fig. 3.6 Shapley values for the input parameters of the biometric dose function (equation 3.1)

One of the fundamental properties of Shapley values is that they always sum up to the difference between the game outcome when all players are present and the game outcome when no players are present. For machine learning models, this means that *shap* values of all the input features will always sum up to the difference between baseline (expected) model output and the current model output for the prediction being explained.

3.4 Importance Sampling

3.4.1 Introduction to Importance Sampling

Monte Carlo simulations are often carried out to estimate the expected value $E(y)$ of a variable y, where y is the result of a calculation model f with the influencing factors $x = (x_1 \ldots , x_d)$, i.e. $y = f(x)$. The expected value $E(y)$ is defined as follows:

$$E(y) = E\big(f(x)\big) = \int f(x)p(x)dx \tag{3.6}$$

where *p* is the multivariate density function of the influencing factors *x*. If *y* is an indicator variable for the occurrence of a certain event *E* (e.g. event = system failure), i.e. *y* = 1 if *E* occurs and y=0 otherwise, then the expected value *E (y)* is identical to the probability *Prob(E)* for the event *E* (e.g. probability of a system failure). If *y* only differs significantly from zero in a very low probability range in the parameter space of *x* (or is equal to 1 as an indicator variable), then the estimation of the expected value *E (y)* or the probability *Prob(E)* using simple MC simulation is very inefficient. In addition, simple MC simulation is no longer practicable if complex calculation models are used because too many calculation runs have to be carried out.

To increase the efficiency (variance reduction) of a MC simulation and for more practicability when using complex calculation models, the method of Importance Sampling is suitable. The sample elements of *x* are not selected from the actual distribution *p*(*x*) but from the distribution *q*(*x*), which concentrates on the low-probability range of interest in the parameter space. The distribution *q*(*x*) denotes the Importance Sampling density. The following applies: $q(x) > 0$ if $f(x)p(x) \neq 0$. If the Importance Sampling method is carried out with the Importance Sampling density *q(x),* then the expected value *E* (*y*) is determined as follows:

$$E(y) = E\big(f(x)\big) = \int \frac{f(x)\cdot p(x)}{q(x)} q(x)dx = E_q\left(\frac{f(x)\cdot p(x)}{q(x)}\right) \tag{3.7}$$

E_q stands for the expected value under the condition that *x* is distributed according to *q*(*x*). This results in the following estimator for the expected value:

$$\hat{E}(y) = \hat{E}_q\left(\frac{f(x)\cdot p(x)}{q(x)}\right) = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)\cdot p(x_i)}{q(x_i)} \tag{3.8}$$

where ${x_i}_{i=1}^{n}$ is a sample of the size *n* from the distribution *q*(*x*). The following applies to the variance of the estimator:

$$Var\left(\hat{E}(y)\right) = \frac{1}{n} Var_q\left(\frac{f(x)\cdot p(x)}{q(x)}\right) \tag{3.9}$$

$$\boldsymbol{Var\left(\hat{E}(y)\right) = Var\left(\hat{E}(y)\right) = 0}, \text{ if } \boldsymbol{q(x) = q_{opt}(x) = \frac{f(x)\cdot p(x)}{E(y)}} \tag{3.10}$$

where is the optimal Importance Sampling density. However, its calculation is not practically feasible because it requires knowledge of *E(y).* In order to estimate *$q_{opt}(x)$*. the approximation of a multivariate parametric distribution and kernel density estimation can be used. Both methods are described in the next section.

3.4.2 Approximation of an Importance Sampling Density

The approximation of a multivariate parametric distribution or kernel density estimation can be used to estimate an optimal Importance Sampling density. The prerequisite for both methods is a sample from the (usually low probability) range in the parameter space of *x*, for which *y* = *f(x)* is clearly different from zero (or equal to 1 if y is an indicator variable), and for which *y* = *f(x)* is otherwise almost zero (or equal to zero). This means that a sample from the unknown Importance Sampling distribution must be available. Such a sample can be obtained, for example, from the results of an adaptive MC simulation method developed at GRS /KLO 21a/.

The quality of the approximated multivariate distribution or the kernel density as an estimator for the optimal Importance Sampling density depends in particular on the probabilistic properties of the sample elements. Ideally, these should be distributed independently and identically to the Importance Sampling density. If the available sample elements were obtained by using the subset sampling method, such as in the adaptive MC simulation method SuSSVR (combination of subset sampling and support vector regression, /KLO 21a/ developed by GRS, the sample elements are not independent due to the use of Markov chains. This leads to a slowdown in the convergence of the estimators compared to the ideal case with independent sample elements. However, the use of the sample elements from the subset sampling is justified because the estimators converge as the sample size increases.

3.4.2.1 Approximation of a Multivariate Parametric Distribution

If $x = (x_1, \ldots, x_d)$ is a random vector with an unknown distribution and ${x_i}_{i=1}^{n}$ is a sample from this distribution, i.e. the $x_1, \ldots, x_n$ are independent and identical to *x* distributed random vectors, then for each variable x_j of the vector *x* their mean $E(x_j)$ and variance $Var(x_j)$ can first be estimated from the sample

$$\hat{E}(x_j) = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \; \widehat{\mathrm{Var}}(x_j) = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_{ij} - \hat{E}(x_j)\right)^2 \tag{3.11}$$

Instead of mean $E(xj)$and variance *Var(*x_j*)* quantiles can also be estimated. The following applies to the estimator of the p-quantile (e.g. $p = 0.05$) of the variable x_j:

$$\hat{q}_{pj} = x[n \cdot p]j \qquad (3.12)$$

where $\hat{q}_{pj}$is the $[n \cdot p]$-th value of the ordered sample $x[1]j \leq x[2]j \leq \cdots \leq x[n]$. The distribution parameters of the selected univariate and continuous distribution types can be determined using SUSA /KLO 21/ on the basis of $\hat{E}(x)_j$ and $\widehat{Var}(x_j)$ or a selection of (e.g. three) quantile estimators (e.g. for p = 0.05, 0.50 and 0.95). In the next step, a Kolmogorov-Smirnov or Lillifors (assuming a normal, lognormal or exponential distribution) adjustment test /KLO 21/ is carried out for each completely defined distribution. Both tests are non-parametric and compare the empirical distribution function from an existing sample with a selected continuous distribution. Based on the test results for the present sample ${x_{ij}}_{i=1}^{n}$, a suitable univariate distribution can be selected for the variable x_j. Finally, the correlation coefficients between the variables $x_1, \ldots, x_d$ must be estimated from the available sample. The normal Pearson correlation coefficient between the variables *xj* and x_k is estimated as follows:

$$\hat{\rho}(x_j, x_k) = \frac{\sum_{i=1}^{n}\left(x_{ij}-\hat{E}(x_j)\right)!\left(x_{ik}-\hat{E}(x_k)\right)}{\sqrt{\sum_{i=1}^{n}\left(x_{ij}-\hat{E}(x_j)\right)^2 \cdot \sum_{i=1}^{n}\left(x_{ik}-\hat{E}(x_k)\right)^2}} \qquad (3.13)$$

If the continuous univariate distributions for the variables $x_1, \ldots, x_d$ and the correlation coefficients between these variables are determined on the basis of a sample from the unknown Importance Sampling distribution, a multivariate distribution density can ultimately be defined as an approximation to the Importance Sampling density. In SUSA, a multivariate distribution is defined by entering the respective univariate distributions of the uncertain parameters (variables) and the correlation coefficients between the parameters. If several correlation coefficients are strongly positive or negative, then some eigenvalues of the correlation matrix can be zero or almost zero. This can lead to eigenvalues less than or equal to zero as part of the sampling procedure in SUSA due to numerical errors. As a result, the Cholesky decomposition of the estimated correlation matrix would not be possible and sampling from the multivariate distribution would not be feasible within SUSA. One solution would be to decompose the correlation matrix and replace the non-positive eigenvalues with a small positive epsilon.

3.4.2.2 Kernel Density Estimation to Derive an Importance Sampling Density

Kernel density estimation (KDE) is a non-parametric statistical method for estimating the density of an unknown probability distribution on the basis of a sample. Unlike a histogram, a kernel density estimator is a continuous estimator of density. If ${x_i}_{i=1}^{n}$ is a sample from an unknown univariate distribution – i.e. the $x_1, \dots, x_n$ are independent and identically distributed random variables with an unknown distribution – and K is a kernel, the kernel density estimator with bandwidth $h > 0$ is defined as

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h}K\left(\frac{x-x_i}{h}\right) \tag{3.14}$$

The kernel density estimator is therefore the weighted sum of correspondingly scaled kernels, which are positioned depending on the sample realisation. The choice of bandwidth *h* is decisive for the quality of the kernel density estimator. With a bandwidth chosen as a function of the sample size, the sequence of kernel density estimators almost certainly converges uniformly towards the unknown probability density as the sample size increases. Intuitively, one would like to choose h as small as possible. However, a balance must always be struck between the bias of the estimator and its variance.

A number of distribution densities are available as (univariate) kernels *K*, such as the densities of the uniform distribution, triangular distribution, standard Cauchy distribution, or standard normal distribution (Gaussian distribution). Due to its mathematical properties, the density of the standard normal distribution is often used, i.e.:

$$K(z) = \frac{\mathbf{1}}{\sqrt{\mathbf{2\pi}}}\boldsymbol{e}^{-\frac{1}{2}tz^2} \tag{3.15}$$

In the multivariate case, where there is a sample of *n* independent and identically distributed random vector $(x_i, \dots, x_{id})_{i=1}^{n}$, $\mathrm{d} > 1$, a multivariate kernel must be used for kernel density estimation. If the variables of the vector $x = (x_1, \dots, x_d)$ are independent, the multivariate kernel can be represented as a product of univariate kernels K:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{d}\frac{1}{h_j}K\left(\frac{x_j-x_{ij}}{h_j}\right) \tag{3.16}$$

The kernel density estimator for the multivariate case with correlated variables (x_1, …, x_d) can be formulated as follows:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\det(H)}K\left(H^{-1}(x - x_i)\right) \tag{3.17}$$

where *K* is the multivariate kernel and *H* is the matrix of bandwidths. The multivariate normal distribution with covariance matrix *H* is often used as the multivariate kernel.

3.4.2.3 Implementation of kernel density estimation

In Python, there are several options for kernel density estimation in the multivariate case. The three best-known freely available options are:

- SciPy: *gaussian_kde*,
- Statsmodels: *KDEMultivariate*,
- Scikit-learn: *KernelDensity*.

In a comparison, *KDEMultivariate* from Statsmodels and *KernelDensity* from *Scikit- learn* proved to be the most suitable. The kernel density estimation with *gaussian_kde* from SciPy is based on the covariance matrix, which was estimated from the underlying sample. A Cholesky decomposition is performed for this covariance matrix, but this often does not work because the matrix is not positive definite. Therefore, the error 'singular matrix' is often obtained. Kernel density estimation with *KDEMultivariate* or *KernelDensity* is based on the equation (*), i.e. it assumes uncorrelated variables. As the equation shows, this kernel density estimation depends significantly on the selection of the bandwidth. *KernelDensity* from Scikit-learn uses only a single value for the bandwidth for all variables involved. This is fine if the individual variables are not very heterogeneous. However, if the variables differ by orders of magnitude, then *KernelDensity* is not suitable for kernel density estimation. In comparison, *KDEMultivariate* from Statsmodels uses a separate bandwidth for each variable and is therefore well suited for heterogeneous variables. A major disadvantage is that it does not take correlations between the variables into account.

3.4.2.4 Sample from a Kernel Density

The option to sample from a kernel density has been implemented in SUSA. In order to draw a sample from the kernel density, it is not necessary to estimate a kernel density. The only information needed for the sampling procedure is the sample values for the kernel density estimation and the bandwidth values. The sampling procedure can be outlined as follows:

- Draw a vector $(x_1, \ldots, x_d)$ from the sample $(x_i, \ldots, x_{id})_{i=1}^{n}$, $d \geq 1$ on which the kernel density estimation is based.
- For each value x_{ij}, *j = 1. …, d,* of the drawn vector, draw a value from the univariate kernel that relates to x_{ij} I.e. if the kernel is a standard normal distribution, then draw a value from the normal distribution with the mean x_{ij} and the standard deviation h_j (bandwidth).
- Go back to step 1.

A new vector is sampled each time the procedure is run. The number of passes determines the size of the sample from the kernel density.

4 Enhancement of Platform Independency and Expansion of the Range of Applications

The modularity of SUSA has been improved following the guideline of a layered approach. The following layers have been identified:

1. Basic Python or FORTRAN routines which could also be called in a command line interface.
2. Higher-level functions or classes which provide a convenient interface to the user. An example of this is the provision of a generic interface for calling FORTRAN-based functions, as described in Section 4.3. For most of these higher-level functions, example scripts have been made available.
3. Jupyter notebooks which show the implementation of a whole analysis chain and give the user the ability to interactively understand the analysis procedure. These notebooks have been provided for the RAMESU examples presented in Section 4.1.3, but also for the neural-network-based application of the subset sampling algorithm for the Ishigami function discussed in Section 5.2 as well as for the sensitivity and tolerance limit methods available in SUSA.
4. A GUI which gives the user the opportunity of employing the capabilities of SUSA without the need to be proficient in the Python program- ming language. The first development steps as well as the planned future layout of this GUI are described in the following sections.

4.1 Developments Towards a SUSA GUI

4.1.1 Basic Concept

The following requirements for the future SUSA GUI have been identified:

- The new SUSA GUI should be platform independent.
- It should be easy to couple it to the new Python SUSA.
- The classic SUSA GUI user should be able to receive a similar level of support if this is required.

- On the other hand, the advanced user should have the option to quickly enter the envisaged distributions and parameters without having to pass through the same steps of guidance as a user needing support and have a similar or wider range of capabilities.

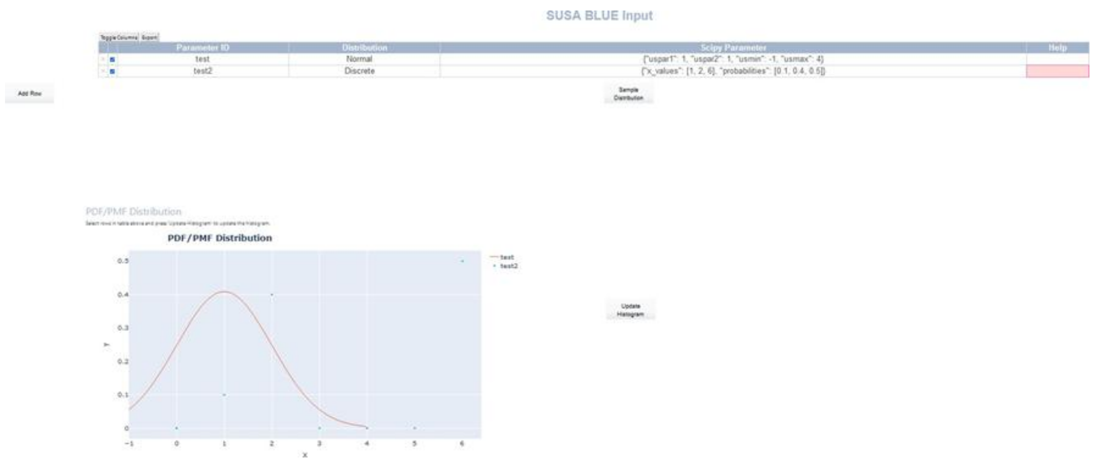
The open-source library Dash Open-Source has been chosen as basis to develop the new SUSA GUI in a web-based dashboard style. Dash Open-Source is distributed under the permissive open-source MIT (*M*assachusetts *I*nstitute of *T*echnology) licence. Currently, the SUSA GUI can be used by running the underlying dash app on localhost; this way, it can only be accessed from the machine which started the app. In addition, methods exist to turn the resulting Dash app into a standalone desktop application.

Similar to the classic SUSA, it is planned to structure the new SUSA GUI into four main parts:

1. Definition of the input uncertainties;
2. Sampling of the modelled uncertain parameters;
3. Generation of the input files for different simulation software and potentially start of simulations;
4. Provision of data analysis capabilities.

4.1.2 Distribution Input Example

A first working example for parts of the distribution input section is shown in Fig. 4.1. The central part of the distribution input section is a table collecting all passed-in information. The table is editable and the lines (called rows on the dashboard) are selectable.


SUSA BLUE Input

Parameter ID	Distribution	Scipy Parameter	Help
test	Normal	{"uspar1": 1, "uspar2": 1, "usmin": -1, "usmax": 4}	
test2	Discrete	{"x_values": [1, 2, 6], "probabilities": [0.1, 0.4, 0.5]}	

Fig. 4.1 First working draft of the new SUSA GUI dashboard showing the tab for entering parameter distributions

The main columns of the tables are:

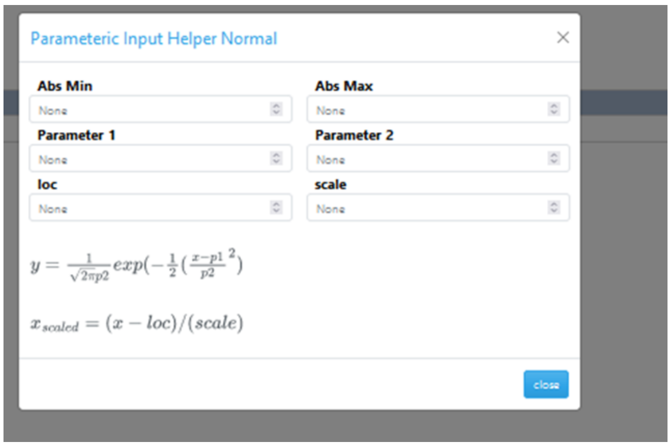
- **Parameter ID:** ID of the parameter. The ID needs to contain, in accordance with the requirements on a valid Python identifier, only alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number or contain any spaces. In addition, each ID needs to be unique. If these requirements are not met, an adequate error message will be presented to the user. The parameter ID can be accompanied by a more descriptive parameter name in the column "Descriptive Parameter Name".
- **Distribution:** A dropdown menu providing the user with the names of available distribution types.
- **SUSA parameter:** Here, the experienced user can provide the expected parameters, as detailed for the corresponding SUSA parameters in the updated SUSA User Guide. Lines for which parameter ID, distribution and SUSA parameter are provided will be marked as complete, as no further input is required.
- **Help:** In case the user is not sure about the parameter format and values needed, the Help field in the corresponding line can be clicked, and a pop-up will appear, guiding the user through the available options. The kind of pop-up depends on the selected distribution. Once all information is entered into the input helper and the helper is closed, the SUSA parameter column is updated with the appropriate information. In this way, the user can easily copy, paste and adapt for further similar uncertain parameter distributions.

The following columns can be made visible with use of the *Toggle* columns button on the upper left of the dashboard:

- **Descriptive parameter name:** As mentioned before, this column allows the user to provide a more descriptive parameter name, which can be stored and used for documentation purposes.
- **Reference value:** This is the value that will be used if no distribution is provided. Lines for which a parameter ID and a best estimate or reference value are given will be marked as complete, since in this case no distribution should be added.
- **Best estimate value:** This is an alternative value that can be used if no distribution is provided. Lines for which a parameter ID and a best estimate or a reference value are given will be marked as complete, as in this case no distribution should be added.
- **Unit:** This column allows the user to store the unit of the parameter for documentation purposes in the table.
- **Notes:** This column allows the user to add all kinds of additional notes for documentation purposes.

The button *Add Row* below the table on the left-hand side of the dashboard allows to add additional lines and therefore also additional uncertain parameter to the table. In this way, the table extends as needed by the user. The *Export* button above the table downloads the contents of the *DataTable* as .csv file. Currently, the *Export* button automatically downloads the table to the "*Download*" directory. In the future, an input field will be provided so that the user can set the output directory. The *Update Histogram* button allows the visualization of the selected distributions detailed in the table.

When clicking on one of the cells in the rightmost column of the table, a popup window appears in which the user can enter information about the desired distribution. Two types of these input helper windows can be distinguished, an input helper for parametric distributions and an input helper for non-parametric distributions, which encompasses discrete distributions, polygonal distributions, and histograms. Fig. 4.2 shows the input helper for a normal distribution and Fig. 4.3 that of a discrete distribution.

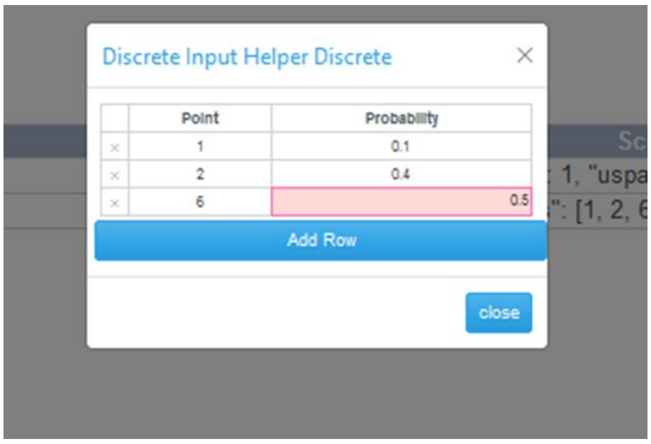

Parameteric Input Helper Normal

Abs Min	Abs Max
None	None
Parameter 1	**Parameter 2**
None	None
loc	**scale**
None	None

$$y = \frac{1}{\sqrt{2\pi p2}} exp(-\frac{1}{2}(\frac{x-p1}{p2}^2)$$

$$x_{scaled} = (x - loc)/(scale)$$

Fig. 4.2 First working draft of the new SUSA GUI dashboard showing the tab for entering normal distributions


Discrete Input Helper Discrete

	Point	Probability
×	1	0.1
×	2	0.4
×	6	0.5

Fig. 4.3 First working draft of the new SUSA GUI dashboard showing the tab for entering normal distributions

The input helper for parametric distributions provides input fields for the minimum and maximum truncation values, for the parameters needed by the distribution, for location (LOC) and scale parameters which allow to modify the distribution, and formulas both for the distribution and for the way LOC and $scale$ modify the distribution.

The input helper for non-parametric distributions provides a table for point/probability in case of discrete distributions, for lower bin limit/height in case of histograms, and x-coordinate/y-coordinate in case of polygonal functions.

4.1.3 Planned Extensions

Fig. 4.4 provides an outlook on how the final SUSA dashboard is envisioned. The envisioned dashboard is divided into four tabs, with each tab belonging to one important part of the SUSA software package. The tabs ‘*Input Uncertainties*’, ‘*Sample Generation*’ and ‘*Computer Code Preparation*’ correspond to dropdown selections ‘*Input Uncertainties*’, ‘*Sample Generation*’ and ‘*Computer Code Runs*’ in the classic SUSA GUI. The name ‘*Computer Code Runs*’ will be changed to ‘*Computer Code Preparation*’, as it is more descriptive with respect to the nature of the provided services. The classic SUSA dropdown selections ‘*Uncertainty Analysis*’, ‘*Sensitivity Analysis*’, ‘*Scatter Plot*’ and ‘*Cobweb*’ will be grouped into one dashboard tab ‘*Data Analysis*’. Back and forth tabs in the lower part will guide the user through the dashboard.

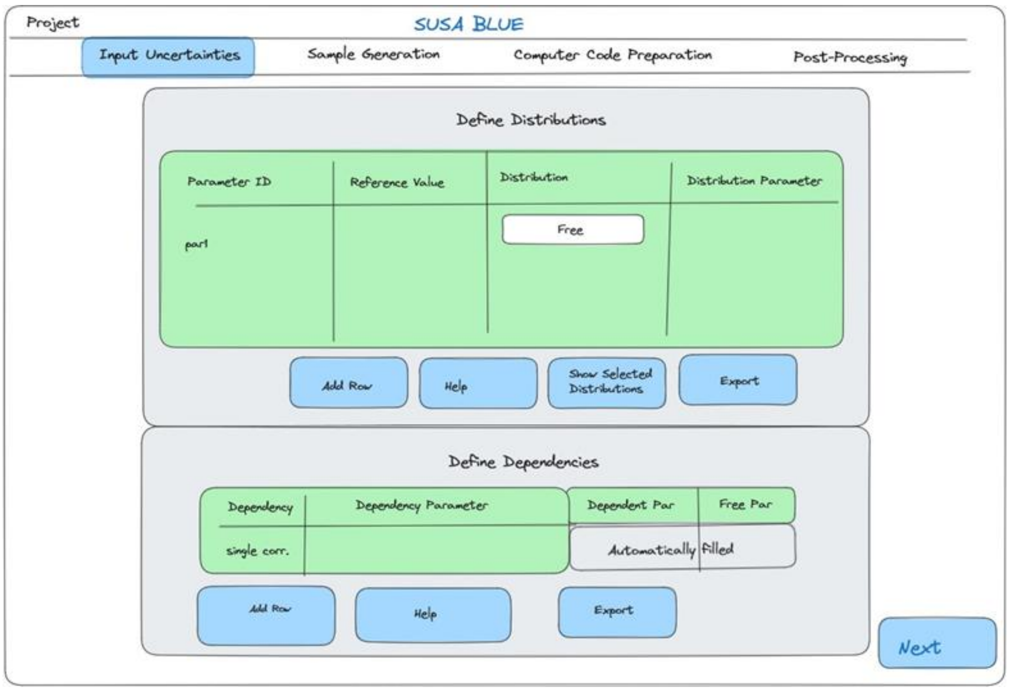


Fig. 4.4 Visualization of the concept for the new SUSA dashboard GUI

One important part of the input uncertainty handling still missing from the first implementation of the input tab are dependencies between uncertain parameters. In the new SUSA GUI, uncertainties will be collected in the same dashboard tab as the input parameter distributions, enabling the user to see the full modelled input in one page as shown in Fig. 4.4. The tab will be divided into two parts, one part for the distribution input and one part for the dependency input.

Like the distribution input, the dependency input will be in table format. The first column of the table will contain all available dependencies. The second column corresponds to the SUSA parameter column of the distribution table; here the expert user can enter the required distribution parameter in form of a Python dictionary. Like for the distribution input, a help button should be available for non-expert users to guide them through the process of defining the desired dependencies.

The classic SUSA GUI distinguishes between population- and sample-related dependencies. The following dependencies are provided in both cases:

- Full dependency;
- Conditional distribution;
- Function of parameters;
- Inequality.

In addition, the classic SUSA allows specifying association measure correlation dependencies, such as Pearson correlation, Blomqvist medial correlation, Kendall rank correlation and Spearman rank correlation. Only the Spearman rank correlation is available for sample-related dependencies. A detailed description of the two types of dependencies can be found in the SUSA method guide.

Since the distinction between population- and sample-related dependencies does only affect correlation dependencies directly, the user should only be asked to make a decision between sample- and population-related dependencies if he/she is interested in correlation dependencies.

The following dependencies will be available in the new SUSA:

- Single correlation;
- Matrix correlation;
- Inequality;
- Conditional distribution;
- Function of parameters;
- Functional combinations.

The classic SUSA single correlation will be complemented by a matrix correlation. Matrix correlation is very useful if the correlation dependency can be described by a large linear equation system. In this way, a single matrix can be used to describe the correlation between a multitude of parameters. A further addition to the provided dependencies will be the functional combinations. The principal idea behind functional combination is that the user can select a functional dependency between the dependent parameters and several free parameters. Functional combinations encompasses three special cases: first, linear combinations which are linear combinations of several ’free’ parameters defining the dependent parameter; second, functional combinations which are all non-linear combinations of several ’free’ parameters defining the dependent parameter; and third, proportions. Proportions are the association of multiple uncertain parameters which represent the proportions (percentages) of a whole and, therefore, must sum up to 1.0 (100 %). For each of these three cases, a popup window will be provided, guiding the user through the process of defining the dependency.

In the following, the help for the inexperienced user will be described for each dependency.

4.1.3.1 Concept of the Dependency Input for the New SUSA GUI

Fig. 4.5 shows the popup mask which will appear if the user selects *Single Correlation* in the table shown in Fig. 4.4 and presses the *Help* button. The user will get dropdown selections for

- population or sample related,
- free parameter 1,
- free parameter 2,
- correlation type, and
- correlation value.

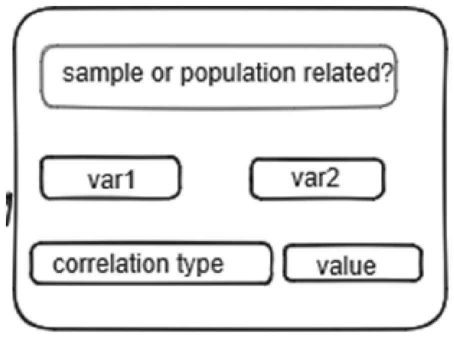


Fig. 4.5 Sketch of the potential popup input helper for correlation dependencies

Fig. 4.6 shows the popup mask which will appear if the user selects *Matrix Correlation* in the table visible in Fig. 4.4 and presses the *Help* button. In this case, the user will be able to upload two files, one file containing the correlation parameter and one file containing the correlation. Once both files have been uploaded, two tables will be populated and presented in the popup, one table containing the parameter names and another table containing the correlation values.

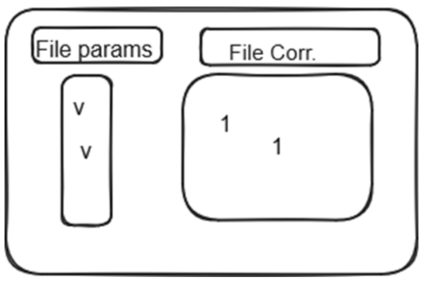


Fig. 4.6 Sketch of the potential popup input helper for matrix dependencies

The distribution table will only show parameters for which no distribution has been entered as potential dependent parameters; Parameters for which a distribution is entered in the distribution table will be shown as potential free parameters. In addition, an input section for numbers will be provided in which the user can enter the correlation value. This input section will only allow numbers between 0 and 1 to be entered. In case a sample-related correlation is chosen, only Spearman rank is available as correlation type.

Fig. 4.7 shows the pop-up helper to be provided for inequality dependencies. The user will get dropdown selections for:

– value modification or resampling,

– dependent parameter,

– free parameter, and

– the inequality factor.

The available inequality factors are *">", "<", "≥"* and *"≤"* as stated before. In addition, a two-dimensional graph will be shown illustrating the selected parameter area.

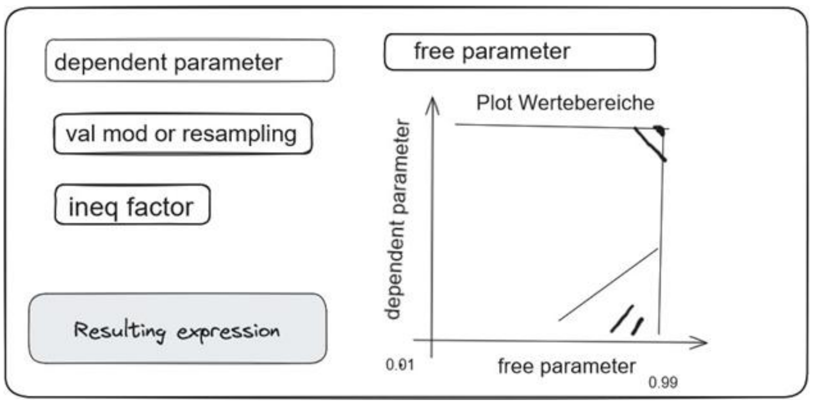


Fig. 4.7 Sketch of the potential popup input helper for inequality dependencies

The option *Inequality* should be selected if the relationship between two parameters *X* and *Y* is given by the inequality equation *Y* = *inequality_factor a * X*, where the factor '*a*' is a real number and the inequality factor is one of the four logic comparisons ">", "<", ">=" and "<=".

There are two alternatives to implement the inequality:

1. '*Independent repeated sampling*' until a sample is obtained that satisfies the inequality;
2. the modification of the values sampled for *Y*.

Both alternatives may affect the marginal distribution specified for parameter Y. 'Independent repeated sampling' may require long computing time. The modification of Y (2nd alternative) is performed according to the following formula:

$$Y' = a \cdot X + \frac{\max Y - a \cdot X}{\max Y - \min Y}(Y - \min Y) \qquad (4.1)$$

where *X* and *Y* are the parameters before the modification, and *Y'* is the modification of *Y*.

In case of a conditional dependency, the distribution of a parameter (considered as the dependent parameter) is a conditional distribution dependent on another parameter (free parameter). The conditional distribution of the dependent parameter has to be specified for each sub-interval of an exhaustive set of mutually exclusive intervals over the range of the free parameter.

The intended layout of the conditional dependency popup is presented in Fig. 4.8. This popup provides two dropdown selections, for the dependent and the free parameter, an input table, which should be used to define the mutually exclusive intervals, two buttons *Add Row* and *Help*, and a graph showing the distribution of the free parameter and the provided intervals. The table will be editable, except for the *Minimum Limit* column, lines will be deletable and a single line can be selected. The *Add Row* button allows the addition of lines (new intervals) to the table, either at the end of the table or, if one line is selected, below the selected row. The *Help* button allows opening a distribution helper popup to define the distribution in a selected line, similar to the *Help* button in the distribution table.

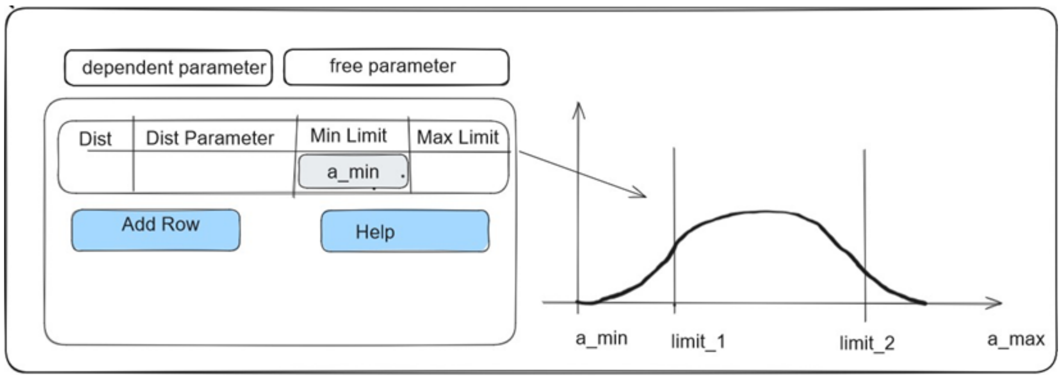


Fig. 4.8 Sketch of the potential popup input helper for conditional dependencies

The table consists of four columns:

- a distribution ID column similar to the distribution ID column in the distribution table. This column contains a dropdown selection of all available distributions,
- the distribution parameter column, in which the advanced user can enter the necessary distribution parameter in form of a Python dictionary,
- the minimum limit column, in which the user can enter the lower limit of the distribution. This value is predefined for each line and cannot be set by the user. When the popup is opened, the *Minimum Limit* column in the first line shows the minimum value

of the free parameter. Once a line is added using the *Add Row* button, the new line *Minimum Limit* column contains the *Maximum Limit* value of the line before and so on. If a line is deleted, the *Minimum Limit* column of the following line will be updated.

- the *Maximum Limit* column, which should be set to the maximum value of the free parameter in the intended interval. The predefined value of this column is the maximum value of the free parameter.

In this case, the dependent parameter Y is associated with other free parameters by an explicit functional relationship. The new SUSA dashboard will offer the option to formulate such a relationship as a Python formula, as shown in an exemplary manner in equation (4.2):

$$Y = \frac{a \cdot X_i + b \cdot X_j}{\sqrt{(X_k)}} \tag{4.2}$$

where Y is the parameter name of the dependent parameter and X_i, X_j and X_k are the parameter names of the free parameters. The values of uncertain parameter Y are derived from the explicit functional dependency on the parameters X_i, X_j, …, X_n. They are affected by the a priori specified marginal distributions F_{X_i}, F_{X_j}. The corresponding pop-up input helper will offer two input fields, one drop-down selection field containing all parameters for which no distribution has been specified and one string input field to enter the Python expression. The new SUSA dashboard will accept all valid Python expressions as formula. The layout of the corresponding popup will contain one dropdown selection for the dependent parameter and an input area in which the formula for the functional dependency can be entered. An automatic check will be performed if the parsed formula is legit. In case this check fails, a warning will be given to the user.

Linear combinations are special cases of functional combinations and Proportions special cases of linear combinations as described above. Due to this relation, only one popup is created for the three cases. The planned popup design is shown in Fig. 4.9. Once the popup has been opened, only the uppermost dropdown selection can be modified by the user, everything else is locked. In this selection, users can decide if they are specially interested in proportions, more general in linear combinations or most generic in functional combinations. Depending on the decision, the elements below the dropdown selection are unlocked and modified.

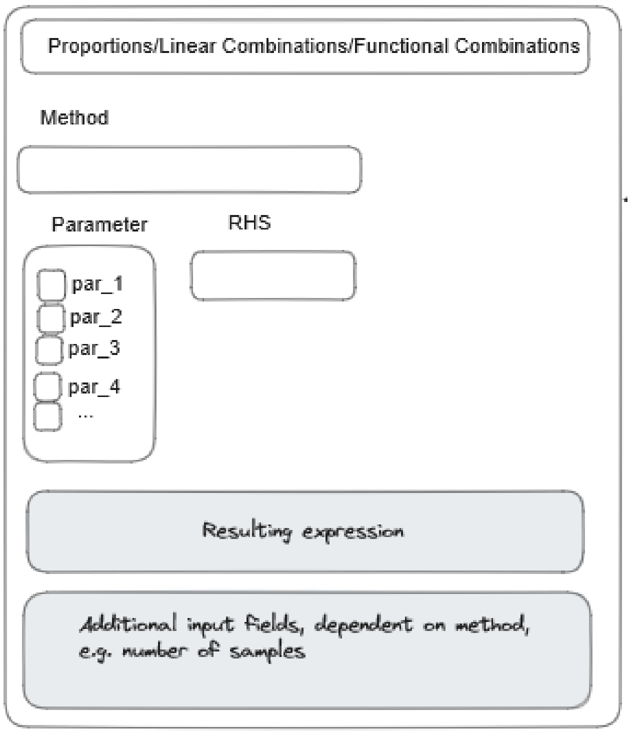

Proportions/Linear Combinations/Functional Combinations
Method
Parameter
RHS
par_1
par_2
par_3
par_4
...
Resulting expression
Additional input fields, dependent on method, e.g. number of samples

Fig. 4.9 Sketch of the potential popup input helper for conditional dependencies

In case '*Proportions*' is selected, the field in which different parameters can be selected is unlocked, method, RHS (*r*ight-*h*and side of the equation) and resulting expression remain locked. The number 1 is entered into the field '*RHS*'. The field *'resulting expression*' is automatically completed with the corresponding expression

$$\sum_i par_i = 1 \qquad (4.3)$$

In addition, an entry field for '*number of samples*' appears in the section '*Additional input fields'*.

In case *linear combinations* is chosen, the fields *method* and *RHS* get unlocked additionally. Different linear combination methods should be selectable, e.g. the classic SUSA method, which leads to a modification of the different distributions, or a brute-force rejection sampling method which rejects all samples not adhering to the required combination. Depending on the chosen linear combination method, additional input fields could appear in the section *Additional input fields*, such as *number of samples*.

In case the most generic option *Functional Combination* is selected, the input fields *Method* and *Parameter* are supposed to be locked, the input field Expression and *RHS* will be unlocked. Potential additional input fields in the *Additional input fields* section could be again the number of samples or maybe the maximum number of iterations, depending on the implementation of the functional combination procedure.

5 Summary and Outlook

For the continuous development of the software analysis tool SUSA, work has been carried out in the research and development project RS1599:

- to extend and enhance the available methods for reliability analysis,
- to extend, harmonize and benchmark the SUSA adaptive sampling methods,
- to improve the platform independency and expand the range of applications, and
- to maintain and adapt the SUSA Users and Methods Guide.

Three reliability software codes have been integrated into SUSA to extend the SUSA capabilities towards reliability analysis.

1. RAMESU: A program to model and analyse dynamic processes of technical systems in the form of Markov and semi-Markov processes and associated uncertainties.
2. AURA: A program to generate generic or system-specific distributions of failure rates or probabilities of failure on demand or repair rates based on the observed number of failures in a given observation time, the number of failures for a given number of demands, or the number of repairs with the repair time required for the repairs.
3. BetaFit: A program which fits a beta distribution to a log-normal distribution.

The implementation of RAMESU in SUSA allows combining the advantage of Markov models, namely gaining a more realistic modelling of system dependencies, with the sampling and analysis functions implemented in SUSA. In this way, the time-dependent development of system failure probabilities can be modelled and analysed for various scenarios, e.g. CCFs or failure behaviours which depend on the system status. RAMESU example Jupyter notebooks have been included in SUSA, showing the application of the new SUSA RAMESU module for seven different example scenarios.

The AURA program, now included in SUSA, is essentially based on Bayesian approaches. The derivation of the two-stage Bayesian approach is explained in /PES 97/, while the derivations of the other approaches are described in /PES 95/. The available 'generic' observations (observations from other comparable plants or information from expert judgement) can be used as prior information (a-priori information) and modified accordingly by current observations from the specific plant of interest. The user thus has the option of including prior information in the generation of the distribution and receives a probability distribution as a result that describes the updated state of knowledge with regard to the parameter of interest. The information to be used for the calculation can be either data from a specific plant and/or observations from other but comparable plants or expert judgement. Hereby, expert judgement refers to the knowledge of quantile data of the reliability distribution, where a maximum entropy approach is used to combine this prior information. If such expert knowledge is not available, Bayesian approaches are used to combine the prior information with a specific plant. Depending on the type of prior information, different approaches are available to arrive at a suitable prior distribution. In the case of 'diffuse' knowledge, i.e. no available prior information, the non-informative prior distribution is used. For the determination of plant-specific distributions with prior information, a specific posterior distribution with either a mixed distribution as prior or with the superpopulation approach as prior, which corresponds to the posterior distribution with unconditional generic distribution, can be used.

The BetaFit program has been included completely in the range of SUSA sampling capabilities. It is now an additional option for the user to obtain a sensible modelling of the regarded input uncertainties. It allows replacing a log-normal distribution with potential unphysical values larger than 1 with a beta distribution which is bounded between 0 and 1.

The advanced MC simulation methods available in SUSA have been harmonized and extended. Compared to classic MC simulation, advanced MC simulation requires only a relatively small number of parameter constellations and corresponding simulation runs. Advanced selection methods are used which are often combined with ML methods. In this project, two of the available advanced MC simulation methods have been benchmarked, once using a biological dose function to generate the target values, once using the Ishigami function, and once using a thermal hydraulic simulator for a LOCA.

The biological dose function tests how a very small probability (1 E-06) can be estimated in a six-dimensional parameter space. The second example with the Ishigami function tests how to identify four separate regions in a strongly non-linear function. Although this is only a three-dimensional problem with a probability of about 1 E-03, finding all four maxima of this function is a difficult task that requires advanced sampling algorithms for proper likelihood estimation. The thermal hydraulic simulator provides a more realistic and complex application example, considering a high-dimensional parameter space (35 uncertain parameters).

The GASA-PRECLAS algorithm has been compared to the SuSSVR for each of these cases. These benchmark tests have also been used to test the power of hyperparameter optimization for each algorithm. In each case, the best convergence criteria were used for the two tested algorithms. Both algorithms function in two cycles, a first cycle in which the parameter space is explored and a second one for providing a good estimation of the probability of the failure region. As for the first cycle in exploring the parameter space, both algorithms performed well when there is only a single contiguous target region. In such applications, an advanced parameter space exploration method is not required. However, if there are multiple target regions that are not connected to each other, the GASA algorithm performs slightly better and can play its advantage of effectively exploiting the parameter space over the first iteration cycle of the SuSSVR algorithm.

As for the second cycle in building a robust metamodel for prediction, both algorithms perform well if the probability is not too small. However, for probabilities below 1 E-06, the simple MC sampling in the PRECLAS algorithm requires too many parameter samples and leads to runtime and memory problems. The Subset Simulation, which is used in the SuSSVR algorithm, already prevents such behaviour. These benchmark tests also provided new ideas for improving the SUSA adaptive sampling capabilities, e.g. by combining the algorithms with an importance ranking of the uncertain parameters, thus reducing the time spent on optimizing parameters of minimal importance.

In addition to this benchmark test, the SUSA advanced MC simulation methods have also been extended by a combination of the subset sampling algorithm with flat neural networks. A Jupyter notebook has been developed to show the application of this new adaptive sampling method using the Ishigami function to provide the target values. The user is enabled to set up a neural network, which can be defined using the Pytorch library, best suited to the given scenario. Further research is needed to understand which kind of neural networks are suited best for different needs.

The application of advanced sampling algorithms, based on machine learning algorithms, often lead to the question which parameters drive the resulting metamodels. In order to increase the interpretability and transparency of the resulting models, methods for calculating Shapley value have been added to the SUSA framework. Shapley values are a concept from game theory that is used to determine the contribution of individual players to the total profit in a co-operative game. Shapley values offer a theoretical approach to interpreting model results. In addition, Shapley values can quantify the contribution of individual variables (input parameters, features) to the model result. Bias and undesirable influences of individual variables can be identified and addressed.

In addition to the adaptive sampling algorithms described above, more traditional methods such as Importance Sampling can also be used to increase the efficiency of MC sampling, even for low-probability target regions. If Importance Sampling is used, sample elements of *x* are not selected from the actual distribution *p(x)* but from the distribution *q(x),* which concentrates on the low-probability range of interest in the parameter space. In the course of the research project RS1599, methods have been developed to extract an Importance Sampling kernel from the results of an adaptive sampling method. In this way, the results of an adaptive sampling run can be reused for further exploration of the target region.

In addition to the method developments described above, the SUSA source code has also been maintained and its modularity improved. In the course of the implementation of the various reliability methods described above, the guideline of a layered setup of the SUSA source code has been followed to provide a consistent interface between underlying FORTRAN applications, such as BetaFit and the SUSA distribution finder, and a Python frontend. This Python frontend can be called either from a command line interface or by running a Python script or a Jupyter notebook. In order to provide the new SUSA capabilities also to users who are not proficient in Python or for a simple and convenient way to use the available method, work on a new graphical user interface has started. A concept for this user interface has been derived and a first prototype implemented. This prototype already provides basic capabilities for modelling the desired input parameter distributions.

A released SUSA package will be accompanied by a Models and Methods Handbook, based on /KLO 21/, and a User Guide /KLO 23/ for the classic SUSA GUI. The Methods Handbook has been updated with regard to the new developments considering the advanced sampling algorithms and the new reliability methods. It is planned to distribute a new SUSA release.

References

/AU 01/ Au, S. K., and J. Beck: Estimation of small failure probabilities in high dimensions by subset simulation, in: Probabilistic Engineering Mechanics 16.4, pp. 263 - 277, Elsevier, October 2001, https://doi.org/10.1016/S0266-8920(01)00019-4.

/BEL 20/ Belluzzo, T.: PyDTMC – A framework for discrete-time Markov chains analysis, 2020, https://github.com/TommasoBelluzzo/PyDTMC.

/BOX 11/ Box, G. E. P., and G. C. Tiao: Bayesian Inference in Statistical Analysis, ISBN 978-1-118-03144-5, John Wiley & Sons, January 2011, https://onlinelibrary.wiley.com/doi/book/10.1002/9781118033197.

/CLO 34/ Clopper, C. J., and Pearson, E. S.: The Use of Confidence or Fiducial Limits illustrated in the Case of Binomial, Biometrika, Volume 26, Issue 4m pp. 404 - 413, Oxford University Press, December 1934, https://doi.org/10.1093/biomet/26.4.404.

/FRO 85/ Fröhner, F. H.: Analytic Bayesian Solution of the Two-Stage Poisson-Type Problem in Probabilistic Risk Analysis, in: Risk Analysis 5.3, pp. 217 - 225, Wiley & Sons online, May 2006, https://onlinelibrary.wiley.com/doi/10.1111/j.1539-6924.1985.tb00172.x.

/GRS 20/ Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH: Managementhandbuch, Kap. 2.2.3.5 Softwareentwicklung (TKP 03-05), Stand 26.05.2020, Köln, Germany, May 2020.

/HOR 90/ Hora, S. C., and R. L. Iman: Bayesian Modelling of Initiating Event Frequencies at Nuclear Power Plants, in: Risk Analysis 10.1, pp. 103 - 109, Wiley online library, March 1990, https://doi.org/10.1111/j.1539-6924.1990.tb01025.x.

/IAE 16/ International Atomic Energy Agency (IAEA): Safety of Nuclear Power Plants: Design, Specific Safety Requirements, IAEA Safety Standards Series No. SSR-2/1 (Rev. 1), STI/PUB/1715, ISBN 978-92-0-109315-8, Vienna, Austria, February 2016, https://www-pub.iaea.org/MTCD/Publications/PDF/Pub1715web-46541668.pdf.

/JEF 46/ Jeffreys, H.: An invariant form for the prior probability in estimation problems, in: Proceedings of the Royal Society A 186.1007, pp. 453 - 461, doi: 10.1098/rspo.1946.0056, Royal Society Publishing, 1946, https://doi.org/10.1098/rspa.1946.0056.

/KLO 91/ Kloos, M., E. Nowak, and E. Hofer: DIVIS: An Interactive Software Package to Support the Probabilistic Modelling of Parameter Uncertainties, GRS-A-1760, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS), GmbH, Garching, Germany, 1991.

/KLO 16/ Kloos, M., and W. Pointner: A More Realistic Uncertainty Analysis Approach for the LOCA Safety Criterion. In: Proceedings of the 13th Probabilistic Safety Assessment and Management Conference (PSAM13), Seoul, Republic of Korea, 2016, October 2016, https://publons.com/journal/325745/proceedings-of-the-international-conference-on-pro.

/KLO 20/ Kloos, M., et al.: Adaptive Monte Carlo simulation for detecting critical regions in accident analyses, in: Baraldi, P., F. Di Maio, and E. Zio (Eds.), Proceedings of 30th European Safety and Reliability Conference (ESREL 2020) and the 15th Probabilistic Safety Assessment and Management Conference (PSAM 15), ISBN: 981-973-4949-00-0, Research Publishing, Singapore, November 2020, https://doi.org/10.3850/981-973-4949-00-0 esrel2020psam15-paper, https://www.rpsonline.com.sg/proceedings/esrel2020/html/4949.xml.

/KLO 21/ Kloos, M., et al.: Adaptive Monte-Carlo-Simulation basierend auf maschinellen Lernalgorithmen und Entwicklungen zur Plattformunabhängigkeit, GRS-634, ISBN 978-3-949088-23-0, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Köln, Germany, January 2021, https://www.grs.de/sites/default/files/2021-12/GRS-634.pdf.

/KLO 21a/ Kloos, M., and N. Berner: SUSA – Software for Uncertainty and Sensitivity Analyses Classical Methods, GRS-631, ISBN 978-3-949088-20-9, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Köln, Germany, March 2021, https://www.grs.de/sites/default/files/publications/grs-631.pdf.

/KLO 23/ Kloos, M., and J. Soedingrekso: SUSA: Version 4.3, Users Guide and Tutorial, Software for Uncertainty and Sensitivity Analysis, GRS-P-5, Vol. 1, Rev. 6, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Köln, Germany, 2023.

/KLU 16/ Kluyver, T., et al.: Jupyter Notebooks – a publishing format for reproducible computational workflow, in: Loizides, F., and B. Schmidt (Eds.): Positioning and Power in Academic Publishing: Players, Agents and Agendas, ELPUB, pp. 87 - 90, doi:10.3233/978-1-61499-649-1-87, 2016, https://eprints.soton.ac.uk/403913/1/STAL9781614996491-0087.pdf.

/PAP 15/ Papaioannou, I., et al.: MCMC algorithms for Subset Simulation, in: Probabilistic Engineering Mechanics 41 (2015) , pp. 89 - 103, July 2015, https://doi.org/10.1016/j.probengmech.2015.06.006.

/PAZ 19/ Paszke, A., et al.: An Imperative Style, High-Performance Deep Learning Library, in: Advances in Neural Information Processing Systems 32 (NeurIPS2019), Vancouver, BC, Canada, 2019, https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

/PED 11/ Pedregosa, F., et al.: Scikit-learn: Machine Learning in Python, in: The Journal of Machine Learning Research 12 (2011), pp. 2825 - 2830, https://doi.org/10.48550/arXiv.1201.0490.

/PES 91/ Peschke, J.: Erweiterung des Programmpakets für Zuverlässigkeits-berechnungen mit Markov-Modellen um eine Option zur Durchführung von Unsicherheits- und Sensitivitätsanalysen, GRS-A-1743. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH, Garching, Germany, 1991.

/PES 95/ Peschke, J.: Methoden zur Gewinnung von Verteilungen für Zuverlässigkeitskenngrößen aus Vorinformation und anlagenspezifischer Betriebserfahrung, GRS-A-2220. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH, Garching, Germany, 1995.

/PES 97/ Peschke, J.: Der Superpopulationsansatz zur Ermittlung von Verteilungen für Ausfallraten und Eintrittshäufigkeiten auslösender Ereignisse, GRS-A-2444. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH, Garching, Germany, 1997.

/POI 18/ Pointner, W., et al.: Statistische LOCA-Analysen, GRS-519, ISBN 978-3-947685-04-2, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Köln, Germany, July 2018, https://www.grs.de/sites/default/files/publications/grs-519.pdf.

/SCH 24/ Scheuer, J., et al.: FORTRAN Development Extensions (libfde), Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH; Köln, Germany, 2024.

/SOE 22/ Soedingrekso, J., et al.: Probabilistic Evaluation of Critical Scenarios with Adaptive Monte Carlo Simulations Using the Software Tool SUSA, Paper No. JA104, in: Proceedings of the 16th Probabilistic Safety Assessment and Management Conference (PSAM16), Honolulu, HI, USA, 2022, https://www.iapsam.org/PSAM16/paper.php?ID=JA.104.

/STR 14/ Strumbelj, E., and I. Kononenko: Explaining prediction models and individual predictions with feature contributions. Knowledge and Information Systems 41(3), pp. 647 - 665, DOI:10.1007/s10115-013-0679-x, December 2014, https://link.springer.com/article/10.1007/s10115-013-0679-x.

/VIR 20/ Virtanen, P., et al.: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, in: nature methods 17.3, pp. 400 - 409, 2020, https://www.nature.com/articles/s41592-019-0686-2.

/WIE 19/ Wielenberg, A., et al.: Recent Improvements in the system code package AC^2 2019 for the safety analysis of nuclear reactors, in: Nuclear Engineering and Design 354), p. 110211, December 2019, https://www.sciencedirect.com/science/article/pii/S0029549319302286.

List of Figures

List of Tables

Abbreviations and Acronyms

ADAM	Adaptive Momentum Estimation
AOT	Allowable Outage Time
API	Application Programming Interface
BEPU	Best Estimate Plus Uncertainty
BETAFIT	Fitting Beta Distribution to Log Normal Distributions
BMUKN	Bundesministerium für Umwelt, Klimaschutz, Naturschutz und nukleare Sicherheit (German for: Federal Ministry for the Environment, Climate Action, Nature Conservation and Nuclear Safety)
BMUV	Bundesministerium für Umwelt, Naturschutz, nukleare Sicherheit und Verbraucherschutz (German for: Federal Ministry for the Environment, Nature Conservation, Nuclear Safety and Consumer Protection)
CCF	Common-cause Failure
DLL	Dynamic Linked Library
FDE	FORTRAN Development Extensions
GASA	Genetic Adaptive Sampling
GRS	Gesellschaft für Anlagen- und Reaktorsicherheit (Germany)
GUI	Graphical User Interface
GVA	German for: gemeinsam verursachter Ausfall
IAEA	International Atomic Energy Agency
IRSN	Institut de Radioprotection et de Sûreté Nucléaire (France)
IS	Importance Sampling
KDE	Kernel Density Estimation
LOC	Location
MC	Monte Carlo
ML	Machine Learning
MSE	Mean Squared Error
MIT	Massachusetts Institute of Technology
NEA	Nuclear Energy Agency
NN	Neural Network
OECD	Organisation for Economic Co-operation and Development
PCT	Peak Cladding Temperature
PRECLAS	Probability Estimation using an Ensemble of Classification Algorithms
PSA	Probabilistic Safety Analysis
RAMESU	Reliability Analysis with Markov Models Extended by an Option for Sensitivity and Uncertainty Analysis
RHS	Right-hand Side
SUSA	Software for Uncertainty and Sensitivity Analyses

SuS	Subset Simulation
SuSSVR	Subset Simulation with Support Vector Regression
SVR	Support Vector Regression

Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH

Schwertnergasse 1
50667 Köln
Telefon +49 221 2068-0
Telefax +49 221 2068-888

Boltzmannstraße 14
85748 Garching b. München
Telefon +49 89 32004-0
Telefax +49 89 32004-300

Kurfürstendamm 200
10719 Berlin
Telefon +49 30 88589-0
Telefax +49 30 88589-111

Theodor-Heuss-Straße 4
38122 Braunschweig
Telefon +49 531 8012-0
Telefax +49 531 8012-200

www.grs.de

ISBN 978-3-910548-71-8