COMPUTER METHODS IN MATERIALS SCIENCE

2021, vol. 21, no. 4, 219–232
https://doi.org/10.7494/cmms.2021.4.0775

A dedicated sensitivity analysis and optimization application for industrial processes

Kamila Myczkowska*, Danuta Szeliga

AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków, Poland.

Abstract

The paper describes the architecture and the use case of the developed Modelbox system for sensitivity analysis (SA), uncertainty analysis (UA) and the subsequent optimization of industrial processes. The proposed solution addresses the most common practical and technical problems encountered by researchers and engineers when performing sensitivity analysis. It combines the functions from the numerical toolbox with a simulation management system. Maintaining usability and a good user experience while managing complex investigations of time-consuming industrial process simulations is a very important feature of the system. Several improvements were introduced to optimize the computation time of analysis/modelling tasks, including the automatization of distributed calculations, persistent, transparent caching of simulation data and duration estimations from collected statistics. The system has the ability to perform remote, parallel, asynchronous computations of both analytic algorithms and numerical simulations. The system is dynamically scalable horizontally by using serverless computing endpoints and thus it can be easily adapted to the user's current needs in a flexible way. Modelbox provides web-based access to analysis/modelling tasks from sampling, SA/UA, optimization to metamodelling. It is extended with numerous interactive visualization components for effective results control. In addition, to access data from the completed analysis, the system supports convergence tracking for SA estimates and intermediate optimization results.

The process of controlled cooling of rails was considered as a case study. The formulated optimization task was to find a combination of process parameters that ensures a minimum volume fraction of bainite along with required interlamellar spacing and optimal homogeneity of hardness. Different sensitivity analysis methods were used to evaluate the significance of all variables with respect to their influence on the model output.

Keywords: sensitivity analysis, modelling of industrial process, optimization, model validation, cooling of rails

1. Introduction

Sensitivity analysis (SA) is the study of how variations in the output of a numerical model can be attributed to variations in its input parameters. It provides information on how well the model resembles the process under study by making it possible to identify which parameters and regions contribute most to output variability. By a parameter, we mean any independent variable that varies in a model, while by region, we mean the boundaries of the input space that are set on that parameter. Sensitivity analysis can be used in different ways during the modelling of a process. The main areas of application include the design of experiments (DOE), model definition, calibration and validation. One of the most common applications is to simplify the process model by eliminating unimportant parameters in order to reduce the computational cost of the execution of

* Corresponding author: kamila.myczkowska@gmail.com
ORCID ID's: 0000-0002-1626-7960 (K. Myczkowska), 0000-0002-2915-8317 (D. Szeliga)

the model. On the other hand, it can indicate important parameters and regions which require more precise handling. Some SA methods give information about interactions between input parameters, which can also be valuable.

SA is often used in conjunction with uncertainty analysis (US). The main objectives of UA are to quantify the influence of the uncertainties of model parameters on its prediction. It helps to learn how stable the process's result is and recognize whether the process is difficult to control due to an excessive influence of parameter uncertainties on the model output. SA/UA is a very useful tool for complex and black-box models and its application is increasing (Douglas-Smith et al., 2020).

The methods of SA are currently less frequently used in the field of industrial processes modelling compared to other areas, especially environmental models. The main reason for this is the very high computational cost of average industrial simulation. Models of industrial processes are often based on the finite element method or other numerical methods, all of which are time-consuming. Thus, SA and UA, which require many simulation runs, become even more challenging. Furthermore, the non-linearity and multimodality of the models force the use of non-deterministic optimization methods, and they require many model simulations. Therefore, the identification of model parameters or the design of an optimal industrial process are computationally costly and complex.

Another obstacle to SA/UA application in the industry is a relatively small number of control variables of processes, which are usually not more than several. A small parameter space decreases the importance of reducing non-influenceable parameters. However, technological progress and a better understanding of industrial processes cause the designs to be more complex, and the number of process parameters is also increased. On the other hand, this trend can make the applications of SA/UA even harder due to technical limitations related to the curse of dimensionality. As the number of parameters increases, the volume of the problem space grows exponentially, which means that a small number of samples cannot properly characterize the model. Thus, the convergence of the algorithms is slow and their results are less reliable. This obstacle to the practical SA/UA applications is confirmed by a survey conducted by Sheikholeslami et al. (2018). About 70% of the SA applications in the environmental modelling literature were performed on models with less than 20 parameters.

Major ambiguities also challenge the researchers in the field of sensitivity analysis, caused by dynamic development followed by a lack of practice guides on SA/US performance. There are many algorithms for SA and UA, but many provide different results. Therefore, the choice of the right method for SA is far from trivial and should depend on numerous aspects, including the number and distribution of input parameters, number of model outputs, time of a single simulation run, technical limitations and main purpose of the analysis. To outline the scope of the problem, Saltelli et al. (2019) found that up to about 40% of the highly cited papers did not meet the elementary requirement of correctly exploring the space of input parameters.

As mentioned above, process simulations can be computationally expensive, but in many cases they can be performed relatively easily, both in parallel mode and in a distributed environment. The majority of sensitivity analysis methods and some optimization methods (or at least parts of them) can also be parallelized. Another aspect of dealing with the industrial process is that it often requires collaboration with other professionals and also consists of many subtasks that need to be managed, such as testing different scenarios, SA/UA, optimizations, and convergence analysis. In addition, these subtasks can take a lot of time to be completed and can also be interdependent upon one another.

2. Sensitivity analysis support in remote simulation environments

In recent years, a variety of software tools have been developed to provide users with UA/SA methods. Douglas-Smith et al. (2020) collected an extensive list of the most popular packages: SAFE, SimLab, MCAT, Gui-HDMR, UQ Lab, GLUE, R Sensitivity are dedicated solutions for Matlab and R. There are also Python libraries (SALib, again SAFE), Fortran and C/C++ (Dakota, PSUADE, PEST/PEST++). One of the newest in this area is VARS-TOOL, compatible with Matlab, C++ and Python.

VARS-TOOL and SAFE, which are among recently developed packages, focus on good SA/UA practices. Pianosi et al. (2016) outlined some of them, including the possibility of reusing one sample for several SA methods, tools for accessing and revising user choices and visualization tools. The impact is put on advanced sampling techniques and SA methods for nonlinear models. Both SAFE and VARS-TOOL offer the user a robustness assessment and convergence analysis without further model runs. Compared to older libraries, new ones offer better protection against improper use. The awareness of the importance of good practices is increasing.

The capabilities of the packages make them very powerful tools, but often at the expense of simplicity. Very few packages have graphical user interfaces, instead usually relying heavily on scripts. As mentioned above, it can be difficult to select the right SA method. Well-designed GUI can play a useful role there by guiding the user, prohibiting improper use by early validation or warning about potential limitations. Although a script-based approach provides more flexibility, a certain level of programming skills and experience is required. That makes almost impossible to use the tool easily without familiarity with the library API or documentation. Having to develop your own scripts can put potential users off, especially while SA/UA is treated as an optional task for model verification or knowledge acquisition.

The details of simulation management are usually hidden in low-level libraries. Most such libraries focus on providing a comprehensive toolkit for performing sensitivity analysis while leaving the user with full responsibility for efficiently managing the tasks associated with the modelling. The libraries can be integrated with other external systems to which they transfer responsibility for resource and simulation management, but this requires additional work and knowledge. There are some exceptions, e.g. SAFE is already integrated into the optimization software tool OSTRICH, and additionally, it supports some parallelization.

Among many tasks that can be performed on a model are a generation of samples sets, SA/UA, metamodelling, and optimization. Some tasks may depend on others, and the dependence is not always obvious. For example, metamodels are known as a replacement of computationally costly original models in optimization tasks, but they can also be used as a model replacement in the case of sensitivity analysis. A reverse relation is also possible: a large number of samples generated during sensitivity analysis can be reused as training samples for metamodel. The dependency forces the sequence of task execution and due to limited resources, the need for a queuing mechanism. For long-running simulations, it is important to parallelize and distribute the computations. Hence, a complex model with many input and output parameters and numerous model tasks would be very difficult to manipulate with a low-level API. Therefore model management tasks are handled by separate systems dedicated to the remote execution of numerical simulations. Such systems should be able to handle problems closely related to distributed computing, such as load balancing, job queuing, on-demand resource provisioning, horizontal scaling of the architecture and self-healing of resources. To facilitate the development of experiments, users should also be separated as far as possible from the highlighted aspects of a system.

The first distributed simulation management platforms appeared over ten years ago. The problem of integration of various middleware components and services for grid platforms is the subject of Radecki et al. (2012). Among others, GridSpace2 and Scalarm are well known distributed implementations aimed at executing remote simulations using a distributed architecture. GridSpace2 environment constitutes a comprehensive platform that supports the whole e-science application lifecycle including development, execution, and result analysis as Ciepela et al. (2012) described. The system enables researchers to conduct virtual experiments on grid-based resources and other infrastructures. The GridSpace2 user-friendly web-based interface assists in the design of the experiment. The system facilitates the development of simulations by using high-level script languages like Ruby, Python, Perl and also bash. Developers can choose from many packages dedicated to handle various grid resources. Scalarm (Scalarm, n.d.) is a massively self-scalable platform, which enables conducting data experiments with heterogeneous computational infrastructure. It works with different architectures like PL-Grid, Clouds and private resources. The system allows users to dynamically monitor simulation executions and use built-in tools for results analysis. Simulations can be parallelized, and Scalarm is responsible for the coordination of distributed executions. The system is distributed as open-source under an MIT License.

Both GridSpace2 and Scalarm provide an environment for designing and performing the computer experiments, however, they are not integrated by default with sensitivity analysis methods. Although Scalarm has a number of input data generators, it does not have embedded sensitivity analysis methods in the current version, which places the responsibility to implement them on the user. A numerical library to address that problem has been prepared by Rauch et al. (2014). The library is dedicated to the generation of input samples for Morris and Sobol' methods and submitting them to Scalarm.

3. Modelbox system overview

The main idea of the developed Modelbox system is to prepare a framework for a modeller that combines data modelling tasks with a comprehensive set of sensitivity and uncertainty analysis methods while being independent of external systems. Modelbox combines low-level SA/UA features and optimization libraries

and enhances them with resource management features from remote simulation environments. The system offers the possibility to perform reliable SA/UA based on numerical simulations with a minimum amount of work needed. This goal is achieved in several ways. Modelbox has a number of tools dedicated to 1) dataset generation and 2) further analytical/modelling tasks (SA/UA, metamodelling, optimization):

1. The definition of sample set is usually the starting point of the modelling workflow. It is required for all operations on the model, including optimization. Samples can be submitted by the user or generated using various designs, including random, full factorial, fractional factorials and latin hypercube sampling. A new sample can also be obtained by merging existing samples or narrowing them to match custom filtering criteria. For more local analytics, the user can either limit sampling to arbitrary subspace or decide to generate random samples in the neighbourhood of a selected point, with a uniform or Gaussian probability distribution.
2. Modelbox contains implementations of the following global sensitivity analysis methods, which are described in detail by Saltelli et al. (2009) and Morris (1991): Morris method, full and fractional factorial design, Pearson correlation coefficient, Sobol's first order and total-effect indices, chi-squared test of independence, VARS measures proposed by Razavi et al. (2016) and graphical methods: parallel coordinates plot, matrix scatterplot. Morris and Sobol's algorithms have predefined variants based on jackknife and incremental resampling strategies, targeted at estimating the precision of SA results for a specified number of samples. For all SA methods, the user can choose for which inputs SA should be computed, leaving others fixed, and define boundaries for them. Unless the method enforces a specific sampling strategy, a set of initial samples can be defined, allowing for flexible reuse of samples for different methods. The system also supports building metamodels (Kriging, Radial Basis Functions – Xiong et al., 2006), uncertainty analysis (bootstrap confidence intervals, histograms and Cumulative Distribution Functions – CDF diagrams) and optimization algorithms (Nelder Mead, Particle Swarm Optimization (PSO), genetic algorithms (GA) and others – Kochenderfer et al., 2019). The implemented algorithms were verified on Sobol-g and Legendre polynomials' classical benchmark functions.

The system works with different types of models, including executable files and script files by using model adapters. If the user does not have an actual simulation model, but only simulation results (probably from experimentally collected data), the system can be still useful. However, its functions will be limited to methods that do not require new model calls, such as matrix scatterplot visualization. To overcome this limitation, a metamodel can be built on available data and then it can be used as an ordinary model.

Simulation configuration parameters, boundaries, and values of fixed parameters that have been selected for subsequent calculations are grouped together in entities called 'specifications'. A specification parameter can be both a technical parameter and inputs that are not the subject of analysis. In both sampling and analysis tasks, the specification is used as an argument, which allows for a flexible investigation of many model variants.

After the simulation execution is complete, the system calculates the simulation's average sequential and parallel duration. Thus, it is possible to give the end-user estimation of how much the potential execution of the task will take for a given configuration (samples size, etc.), both in time and in a number of simulation runs. It also considers that the results of the previously completed simulation are permanently cached, which leads to a more precise estimation. If the average cost of a single sequential simulation run is high and the simulation cannot be easily parallelized, the best sensitivity methods are those which do not require additional simulation runs or require a small number of simulation runs. Variance-based methods are not a good choice, while graphical methods and the Morris algorithm are a better selection. If the model is developed on the basis of experimental data and new samples cannot be generated, methods that impose a specific way of sampling are also not applicable.

The system is capable of automatic caching simulation results. Before the simulation is executed, the system checks whether the result for the specified sample is already in a cache. If there is a match, the result is retrieved from the cache. Otherwise, control is passed to the original model, and the new result is saved in the memory after processing. This approach supports the minimization of the number of model runs and is especially useful when working on the same data while investigating complex, computationally costly process simulations. Cache storage can be reused by different analysis and optimization methods. It can be especially useful when testing different versions of a model, which is common practice in industrial process modelling. Examination of an industrial process chain with a modified single process, when most of the results remain the same, would be an example of this. The system can also

minimize the cost of performing SA/UA, e.g. by automatically reusing results from simulation runs needed by SA/UA as starting points for optimization. Currently, the cache only returns results for exactly the same data samples, but it potentially can be extended with metamodel with the desired level of accuracy.

Sensitivity analysis measures alone do not contain information about how stable the solution is, which is described below. With small sample sets, the results may be highly biased and may possibly lead to misleading conclusions. For this reason, convergence and robustness analysis was introduced in the form of Jackknife and incremental (one-more-sample-at-a-time) methods. They are intended to automatically calculate the variance of measures for both Morris and Sobol's methods. To minimize the number of additional simulations, the new versions of these methods were developed and introduced to the system. They work on subsets of samples, and they are independent of the number of omitted samples. In the Morris algorithm, the distortion of each parameter is calculated with respect to the total sample number.

The implemented analytic/modelling algorithms have built-in support for parallel model execution where it is applicable. A further reduction of their overall computing time is achieved by distributing the simulation executions among working nodes. Each model can be packed into a single Java archive package (JAR) together with the required resources (such as configuration and executable files) and uploaded to the system. It is then automatically distributed to local and remote working nodes. The system takes care of unpackaging and executing the simulation itself, followed by dividing the workload of analytic/modelling tasks among the nodes.

For an even more flexible compute service, models can be wrapped in serverless lambdas deployed in the cloud (for example, in Amazon cloud, AWS, n.d.), using and producing JSON resources. This scenario allows using models written in any language supported by the cloud. For AWS, natively supported runtimes are: Java, Go, PowerShell, Node.js, C#, Python, and Ruby. The platform charges only for a number of requests and time of code execution. It fits in the dynamic needs of the simulation environment very well, where the load is highly variable: by default, very low, interrupted with high peaks when analytical tasks trigger many concurrent simulations. Moving simulation codes to lambdas reduces the responsibility to overprovision servers to cope with peak load.

The Modelbox queuing mechanism makes it possible to create dependent tasks in advance. It allows the user to compose workflow from the user interface without having to wait for previous tasks to complete. This allows unfinished computations to be chained as inputs for subsequent tasks. The dependent task automatically waits until the blocking task is completed and/or resources are available. An example of such a workflow could be:

- performing optimization using PSO,
- performing Simplex optimization for the best optimum from PSO,
- performing UA for the best optimum from Simplex optimization.

Based on the available analytic/modelling tasks, the application supports the methodology shown in Figure 1 for developing a model and its subsequent optimization, which includes the following steps:

- model definition (including a selection of input/output parameters),
- selection of the experiment design, the choice of sample size and sensitivity analysis methods,
- sensitivity analysis – together with its validation,
- model validation in terms of verification that the data collected is of physical meaning,
- model calibration – review previously selected parameters, response variables and their ranges,
- calibration of the optimization method by choosing optimization method, adjusting optimization parameters (like PSO swarm size based on parameters non-linearity or GA crossover rate based on sensitivity indices),
- optimization of the process,
- uncertainty analysis.

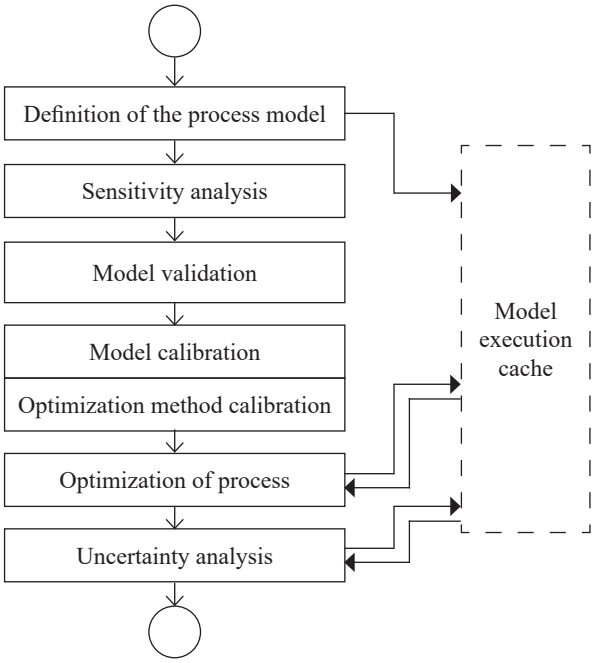

Fig. 1. Methodology of optimization procedure

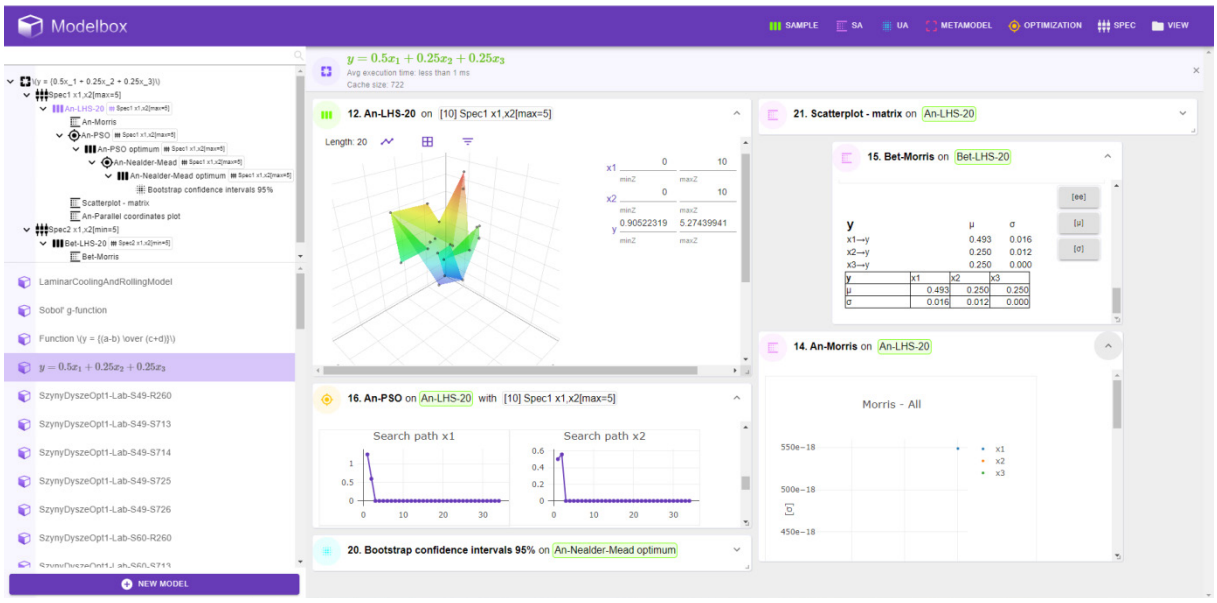

Fig. 2. The main view of the application with an example model

The sensitivity analysis process can be divided into two phases for long-running simulations. In the first, the screening methods should be performed to identify the most influential parameters, and for only indicated ones more detailed analysis should be performed.

The Modelbox user interface is fully web-based. The visual representation of the tasks in the web application is shown in Figure 2. Each analytical task is represented as a navigable, searchable element in a tree on the left side. It is placed as a child node under the task it depends on. If it depends directly on more than one task, additional dependencies are displayed as tags next to its name. Tasks are grouped by specifications, which are the first items under the root. Tasks can be searched by a unique id, associated tags and name. If not explicitly specified, the latter is defined as a concatenation of inherited prefix, algorithm name and sometimes also sample size. All of this facilitates the ease of tracking the origin of tasks.

The Modelbox frontend helps the user to configure, execute, supervise and analyze tasks remotely. Tasks can be dynamically removed and added without the need to restart the workflow, which might be required in the case of a script-based approach. If a task fails, only the dependent tasks are not completed, but the others are not blocked. Tasks results are displayed as separate, draggable and resizable blocks on the grid. By clicking on the header of the block, information about the task can be viewed along with its configuration start time and duration. Some task types, such as optimizations or sampling methods, enable previewing intermediate results before they are completed.

The state of the grid is stored in the backend. The grid allows for flexible visual grouping of tasks according to the user's needs. For example, it facilitates the comparison of results between different datasets by placing them next to each other. The results of the analysis are visualized in real-time with interactive components, without which the graphical methods of SA/UA would be less useful. Parallel coordinates plot may indicate relevant correlations and subregions of parameter space or filter out dense data to be more meaningful. The results can also be downloaded as excel spreadsheets for further processing.

4. System architecture

The main architecture of the developed system is presented in Figure 3.

Modelbox Core Library contains the main computational model and analysis/modelling toolkit, including optimization, metamodelling and SA/UA. It can be used separately from other modules, directly from the command line or as a Java library. *Modelbox Simulation Manager* (MSM), built around the core library, adds features related to model execution management such as scheduling model executions, supervising analytical/modelling tasks executions, caching, saving and combining results. *Modelbox Web Client* is a frontend module for MSM, accessible through a web browser. *Model Repository* is a local repository containing executable codes with simulation definitions in the form of JAR packages. Depending on model configuration, MSM can either run one or more sim-

ulations locally (in the same Java Virtual Machine), dispatch execution to one or more execution nodes or model lambdas. *Modelbox Node* is an optional remote model execution environment designed for efficient horizontal scaling. It accepts packages sent from MSM (and initially fetched from Model Repository) that contain model implementation and configuration. The received package is then deployed, and Modelbox Node coordinates model's execution. It is capable of executing many model runs of different simulations at the same time. It exposes RESTful API for communication with MSM. Except for JAR package, remote models can also take the form of lambda serverless functions, but automatic deployment in the cloud is not yet implemented for that case.

Modelbox has been designed to facilitate the integration of new models and the algorithms working with them. The universal model interface called *Model* is designed to hide the real model behind itself, decouple it from other parts of the system and translate it into the form understood by other modules. The schema of the core Modelbox classes used by the interface is shown in Figure 4.

Specification is a map that contains all configuration parameters of the model. For example, fixed model's parameters or, where applicable, desired number of parallel instances. It is mostly up to the user and specific cases what configuration parameters should be kept inside specification. The specifications field *parameters* hold a description of input and output variables, including their ranges and distribution (for inputs). Most of the parameter descriptions are optional, with only the parameter name required because it is used for accessing sample values.

Input is a container for samples and specifications that is passed for each simulation execution. *Samples* are containers of input/output parameter values. By default, it is a wrapper for arrays of doubles, where each array corresponds to a different parameter. For more advanced uses, a sparse version can be implemented.

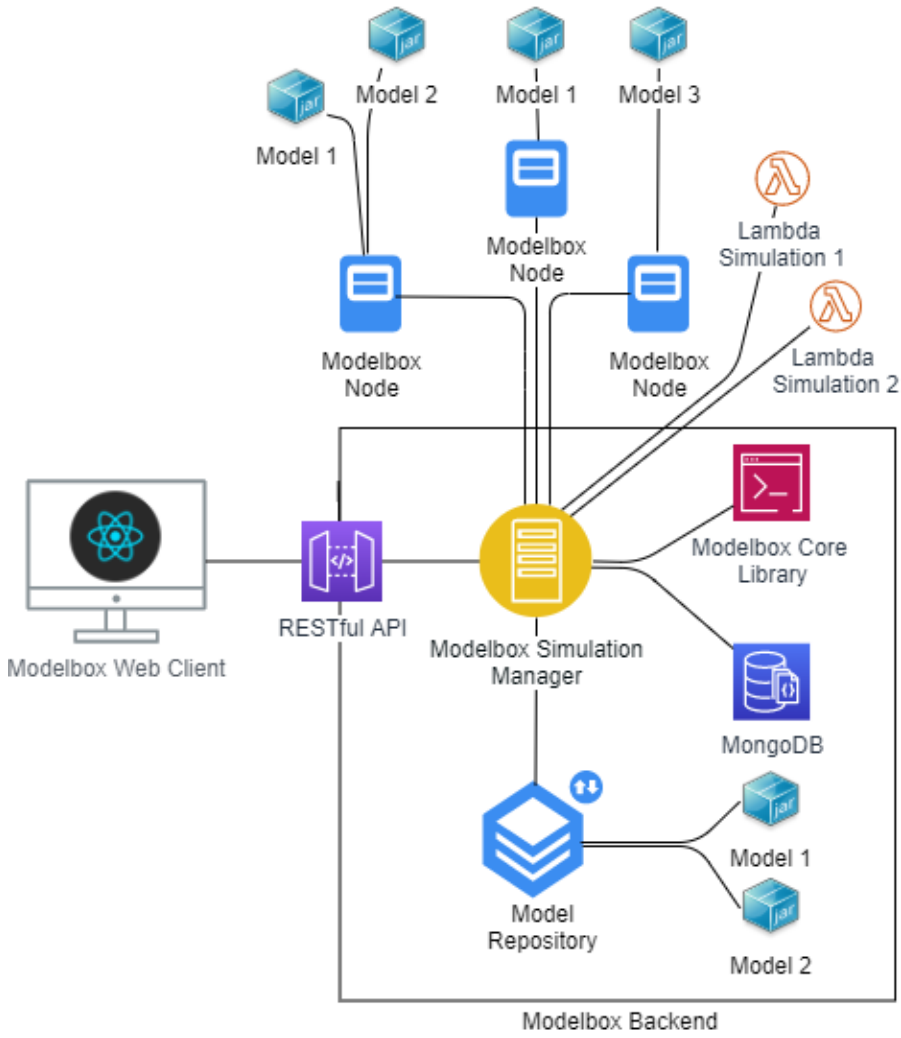

Fig. 3. The architecture of Modelbox

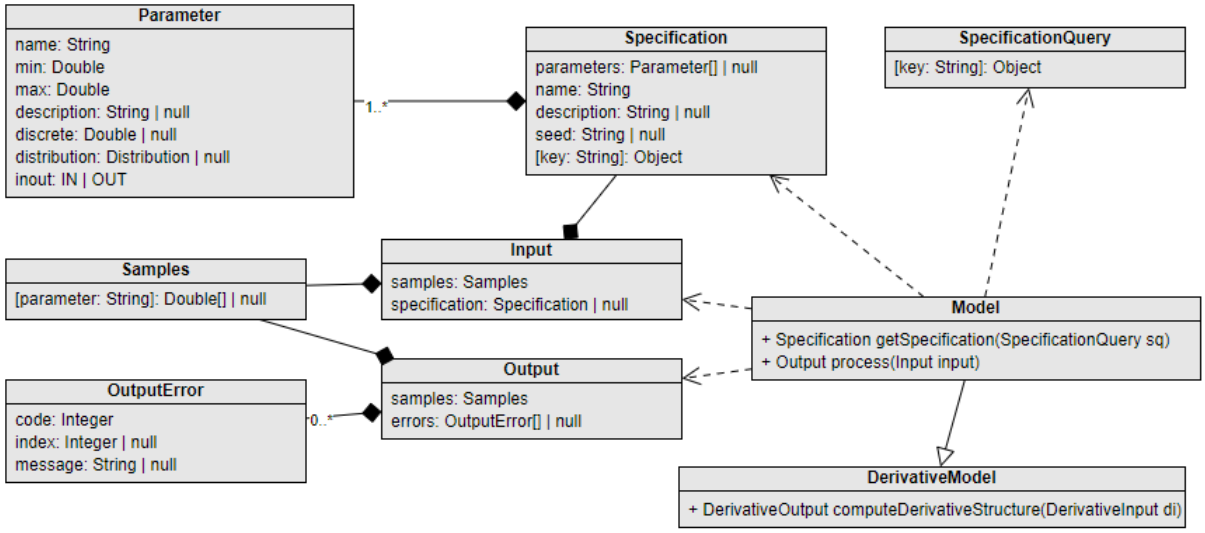

Fig. 4. Schema of core Modelbox objects

To plug one's own model to work with the system, one only needs to implement the two methods listed below:

```
Specification getSpecification
(@Nullable SpecificationQuery query)

Output process(Input input)
```

The *getSpecification* method returns the specification provided by a model. It should contain at least information about parameters and model name as it is aimed mainly at configuring and managing the model for a graphical environment. The *query* argument allows returning dynamic specifications based on the caller's criteria. If it is not set, then the default specification should be returned.

The *process* method reads the values of the input parameters from the supplied samples and updates the values of the output parameters. If no specification is provided, the default one is used. This basic set of functionality is sufficient for most sample generators, SA/UA methods and many optimization methods. The presented design allows for easy embedding and chaining models with each other. In case there were errors during processing, the corresponding information in the *Output* is updated. This way, if some samples cannot be processed, the ones that are completed successfully still can be returned to the caller. The *index* in the *OutputError* refers to the index of the sample that caused the error, or the index is not set if the problem is global. The *code* refers to a type of error, so the caller can react to the problem respectively. The HTTP response status codes have been chosen, so the caller can distinguish automatically between the client (4xx) and server errors (5xx). In the case of client errors, the algorithm may decide to replace the sample with the valid one, and in the case of server error it may throw an exception.

Design is prepared for RESTful integration. Schema classes can be easily converted to/from JSON format for interoperability reasons. The *process* method takes intentionally only one parameter so it can be sent as a body of HTTP post request. Objects are kept as simple as possible to minimize unnecessary network bandwidth. It can be noticed that the *Specification* object is passed as an *Input* field every time *process* function is called. This may seem redundant - but it isn't obligatory – the default version will be used when it's not provided. Nevertheless, some caching mechanisms could still be implemented for multiple large specifications.

Thanks to *Model* interface, model executions can be easily proxied and chained. MSM makes use of some proxy implementations: (a) samples caching, (b) adapters to different programming languages and (c) parallel and/or remote executions. As a result of the latter, an algorithm (SA, UA, optimization) can transparently execute the target model remotely and concurrently, without any changes in its code. Of course, the *process* method will block further calculations until all results are returned, but many algorithms still will gain significant speedup. Another proxy use case would be the bulk optimization on the model level.

The models are intended to be stateless. The example use case consists of two simulations on the same operating system using the same directory for outputting some intermediate data. To ensure that concurrent

simulations will not collide with each other, the *Specification* object has *seed* – a unique identifier assigned to all concurrent simulation instances, set and managed exclusively by MSM. This way, the model can specify the unique working directory based on the *seed*, and copy inside the files that shouldn't be shared by other instances.

For models with automatically calculated differentials of outputs with respect to the model parameters, an additional interface has been designed:

```
DerivativeOutput computeDerivative-
  Structre(DerivativeInput input)
```

where *DerivativeOutput* is a wrapper for *DerivativeStructure* class from commons-math3 library (Apache Commons, 2021) and *DerivativeInput* is an extension of *Input* class with additional derivative order information (integer type).

The Modelbox core library is implemented in Java, while the backend is in Kotlin. Java has been chosen because of its popularity, especially for existing business solutions and a large number of available libraries, and Kotlin because of its full compatibility with Java and the ability to write cleaner code than in pure Java. Another reason was the lack of systems in Java similar to a developed one, in contrast to the variety of solutions offered by Python libraries. However, due to the increasing popularity of Python/JavaScript in recent years, the core library contains Python and JavaScript model adapters (thanks to Jython, n.d. and Java runtime library). As a consequence, dynamic script-based models can be easily created, modified and executed without the need for prior manual compilation.

A very important feature of the system is its use of an asynchronous, non-blocking processing paradigm as much as possible. The benefits of introducing reactivity are especially important for aggregator-type of services that strongly depend on many external resources. Modelbox belongs to this group because of delegating simulations' work to local and remote nodes. The reactive approach eliminates the need to maintain a large pool of threads that are most of the time in a sleeping or waiting states. It results in the better utilization of modern processors, although a significant difference is visible only for high throughput workloads. To maximize benefits, it requires maintaining a non-blocking approach throughout the whole processing flow, including scenarios like a) serving responses to web page and b) performing simulations in the background with asynchronous updates on the database. The used technologies have been selected in order to fulfil this assumption. Both Java and Kotlin offer asynchronous solutions that allow the developer to avoid dependencies among nested callbacks in the code. In the case of Java, there is a family of reactive libraries like Reactor or RxJava. In Kotlin there is an alternative approach based on coroutines and suspension functions (Kotlin, 2021). The latter enabled to write asynchronous code chunks in a sequential manner without much of boilerplate code. Parallelization of model execution has also been achieved with the help of coroutines. For each model definition, the specified number of model instances (or model proxies delegating implementations to remote nodes) has been created. Each model instance is associated with a coroutine block, which executes until there are no more samples to process. Ideally, the input/output operations should also be as much as possible non-blocking. Both libraries chosen for the web layer – Ktor (Ktor, n.d.) and persistence layer – Kmongo (for accessing document-based database MongoDB – MongoDB, n.d.) have support for asynchronous processing.

The idea of reactivity can be extended even further. In the ongoing code development, the reactive version of the core model interface is being prepared, adding to a model the time dimension. Because the core library is written in Java, the Reactor (Project Reactor, 2021) library is used as follows:

```
          Flux<Samples>
process(Flux<Samples>samples,
  Specification specification)
```

Flux stands for a reactive streams publisher with basic rx operators (ReactiveX, n.d.) that completes successfully by emitting a single element or with an error. Available operators include mapping, flat mapping, filtering, repeating, retrying, caching, delaying, timeout handling, merging and combining publishers. All of them can be used by algorithms that operate on model, including optimizations and SA/UA calculations. Although most of the implemented analytical/modelling tasks already run a group of models executions in parallel, there is always at least one place inside the algorithm that blocks the thread waiting for results. The biggest profits can be gain for tasks that have many such blocking operations. It has to be noted that the reactive approach on the model level imposes rewriting algorithms to work in a reactive manner.

The backend of the system is created using the Kodein (Kodein, n.d.), a library dedicated to dependency injection and maintaining a clear separation between object creation/use phases. Such an approach, characterized by loosely coupled elements, strictly based on user interfaces and their implementations, allows the

system to maintain a good level of flexibility, simplicity and modularity, as described by Prasanna (2009). Interoperability is achieved by using JSON (JSON JavaScript Object Notation, n.d.) for configuration and results specification. The system uses a lot of functions from Commons-math3, a library dedicated to math operation, numerical methods, statistics and optimization. For processing large matrices, EJML (EJML, 2015) was selected. For evaluation of math equations directly from string form, the system uses an Exp4j (Exp4j, 2017) library. Communication between frontend and backend modules is based on stateless RESTful services created with the already mentioned Ktor. Independence on user session results in better scalability of the solution. A server push technology SSE (Server-Sent Events) over HTTP has been used for real-time client updates.

The frontend module has been implemented in TypeScript. React (React, n.d.) library has been employed for building a reactive user interface. For interactive 2D and 3D charts, Plotly (Plotly, 2020) library has been selected. Rendering math equations in a web browser is possible with MathJax (MathJax, n.d.) library that works with various formats, for example, mathML, TeXand LaTeX. State management is handled by the Redux (Redux, n.d.) architecture.

5. A case study

To demonstrate the use of the system, a model of the controlled cooling of rails was selected as a case study. The model predicted the temperature field during cooling, the kinetics of phase transformations, microstructural parameters and mechanical properties of the final product. It was described more precisely in Kuziak et al. (2012). The process consisted of multi-stage immersing of the railhead in a cooling solution intermittently with cooling in the air. Such an approach should have resulted in a decrease in the average temperature of the pearlitic transformation while the occurrence of the bainite was avoided. In consequence, fine pearlite microstructure was obtained, which gave increased abrasive wear strength, fatigue strength and resistance to contact-fatigue defects. In the specified work, identification of the phase transformation model was performed on the basis of dilatometric tests and inverse analysis. During optimization the objective function was formulated to give minimum possible interlamellar spacing in pearlite, microstructure free of bainite and uniform hardness distribution in the railhead. The analysis of the specified model with the use of the Modelbox system is presented below.

The external model was an executable file with several configuration files with plain text format and output redirected either to the standard output stream or output file. The model has been wrapped in Modelbox adapter and packaged into JAR file. Therefore, it could be run in parallel or remotely. The average execution time was decreased from 9 sec to 2 sec by employing 4 workers.

Three model parameters were selected as input: austenite grain size ($D\gamma$, μm), number of cooling intervals (Cs, 1–9), and offset of heat transfer coefficient (ΔHTC, W/m^{2}K). The latter shifts the $HTC(T)$ up and down. Other parameters were fixed at their nominal values. The time of cooling interval was set to 10 s, and the time between intervals was set to 20 s. The initial step was to create $n = 100$ samples using latin hypercube design and analyze the results visually. Result variables were: bainite fraction (Fb, %), pearlite fraction (Fp, %), average hardness (HVa, the average of Vickers hardness, calculated by Milenin et al., 2020), hardness homogeneity (HVh) and interlamellar spacing ($S0$, μm, Milenin et al., 2020). A pearlite fraction in steel after cooling with respect to ΔHTC and $D\gamma$ is presented in Figure 5.

The results, presented in Figures 5–10, showed large regions with negligible impact of optimization variables on a fraction of bainite, pearlite, average hardness and hardness homogeneity. It was also observed that in the case of small grain size, the model was almost not sensitive for the ΔHTC and Cs, and $D\gamma$ was the most important parameter. It turned out that the cooling rate was too fast to start pearlite phase transformation, and for a given cooling times, the region boundaries could be shrunk. Therefore, sensitivity analysis was conducted for the grain size 30% ±10% of the value obtained in the previous process. Initially, methods that do not require any additional model executions were performed. The matrix scatterplot revealed a similar, highly linear character of relations between parameters HVa, Fp and $S0$. In chi-squared test, most of the parameters were classified as significant (for a significant level of 0.05). However, the heatmaps presenting areas of high influence obtained using this test showed a large difference in the sensitivity of the model. Parallel coordinates plot allowed the investigation of the indicated areas of parameters space and revealed a strong correlation between HVh and $D\gamma$. Next, the Morris method was selected for execution on the initial sample-set. The computational cost of new model invocations was automatically calculated, and results for already executed samples have been reused. The results revealed the significant impact of $D\gamma$ and, what's new compared to previous methods, the impact of Cs. The deviation of Morris's elementary effects for both parameters indicated possible non-linearity and interactions.

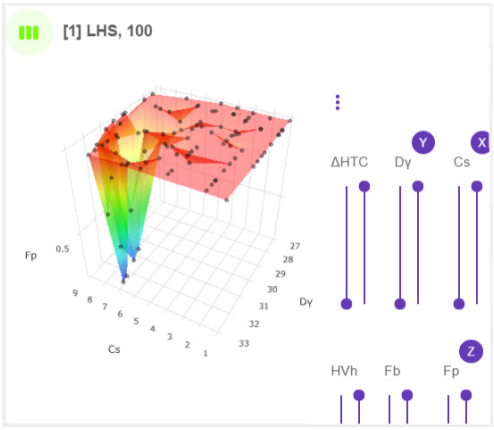

Fig. 5. Interactive 3D chart rendering *Fp* with respect to *Dγ* and *Cs*, number of samples *n* = 100, sampling type = LHS, cooling time = 10 s

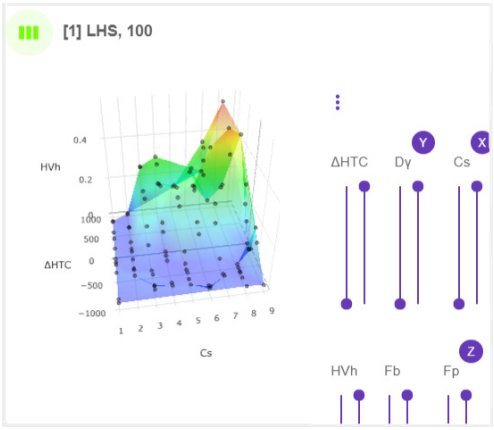

Fig. 6. Interactive 3D chart rendering *HVh* with respect to *ΔHTC* and *Cs*, number of samples *n* = 100, sampling type = LHS

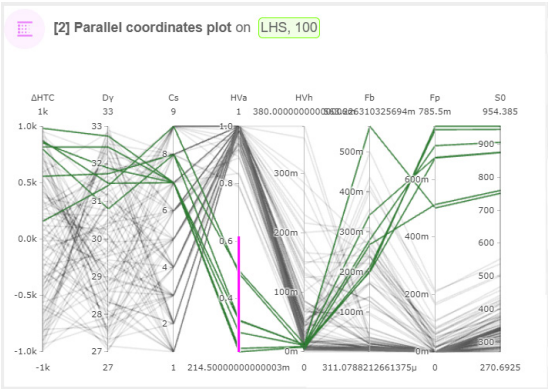

Fig. 7. Results of sensitivity analysis: parallel coordinates plot

Fig. 8. Results of sensitivity analysis: chi-square test

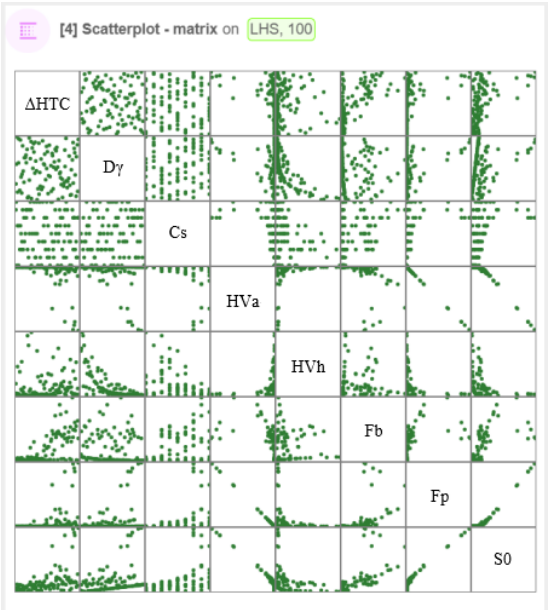

Fig. 9. Results of sensitivity analysis: scatterplot matrix

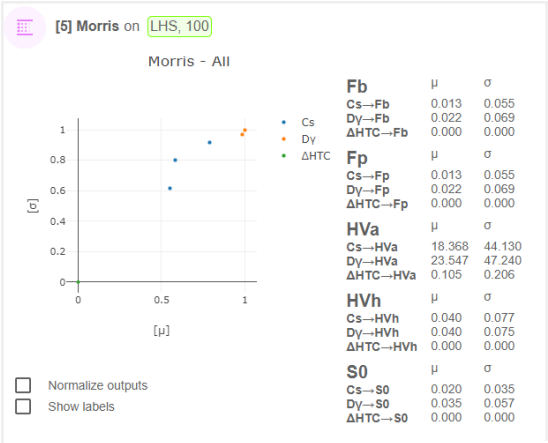

Fig. 10. Results of sensitivity analysis: Morris method, sensitivity measures presented on the chart have been normalized to allow for a comparison between different outputs

Summing up, all methods confirmed the greatest impact of $D\gamma$, therefore, it is important to control that variable precisely. The significance of ΔHTC under-researched conditions was negligible, so it could be marked as a candidate for exclusion from the subsequent optimization.

The goal of optimization was to find a combination of parameters that minimizes the formulated objective function:

$$\text{optGoal}(x) = \sqrt{\Delta HVh^2 + Fb^2 + S0^2}$$

An optimum was sought using a genetic algorithm. The initial population consisted of the samples used for computing sensitivity analysis. Moreover, Morris's sensitivity measures were used to modify crossover probabilities in the following empirical way: for significant parameters the probabilities were increased. In that way, SA allowed reducing the number of solver runs. After 200 model simulations, the following result was obtained: optGoal(x) = 0.08 for $\Delta HTC = -132.16$, $D\gamma = 28.20$, $Cs = 7$, $HVa = 344.42$, $HVh = 0.07$, $Fb = 0$, $Fp = 1$, and $S0 = 0.04$. The results are presented in Figure 10. In the last step, the uncertainty analysis for the optimum point was performed (Morris method on the region ±5% of optimum sample, Fig. 11), which again confirmed the high propagation uncertainty of input to output for $D\gamma$ and Cs, and no significant ΔHTC impact for a specified area.

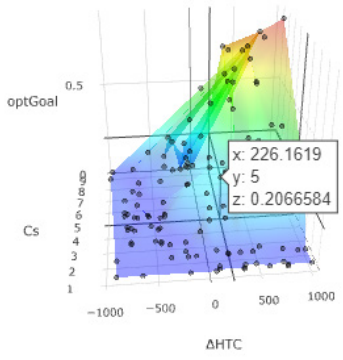

Fig. 11. Interactive 3D chart rendering goal function with respect to ΔHTC and Cs. All samples were taken from the cache, which means they were previously passed to the model, in the process of sensitivity analysis and optimization

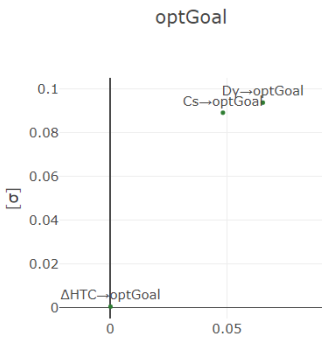

Fig. 12. Results of Morris method performed for goal function on the region ±5% of optimum

6. Summary

The presented system easily applies existing sensitivity and uncertainty analysis methods to internal or external simulations and combines them with other analytical/modelling tasks. Analytical tasks can be created directly from the website, with a minimised need for programmer intervention. The system offers various tools which are useful for preparing, designing, and executing computer experiments, like caching, concurrency support, input/output files processors, and visualization toolbox. The presented architecture can be adjusted to one's needs, starting from a low-level library through a standalone system and finishing with a fully distributed system exploiting serverless simulations.

The system decreases the effort needed to perform sensitivity analysis for the investigated industrial process by automatizing such operations as sample preparation, sensitivity analysis method selection, concurrent execution, and results visualization. The application programming interface described in the paper has been used to plug the existing model into the system. Sensitivity analysis performed on an example model of the industrial process allowed the most important parameter which should be carefully controlled to be detected, and non-influential regions that had been excluded from further calculations.

Future plans include completing reactive versions of analytical/modelling tasks as well as integration with new algorithms that have emerged recently in the field (progressive latin hypercube sampling distributed evaluation of local sensitivity analysis).

References

Apache Commons (2021). *Commons Math: The Apache Commons Mathematics Library*. http://commons.apache.org/proper/commons-math/.

AWS (n.d.). *AWS Lambda. Run code without thinking about servers or clusters*. https://aws.amazon.com/lambda/.

Ciepela, E., Nowakowski, P., Kocot, J., Harężlak, D., Gubała, D., Meizner, J., Kasztelnik, M., Bartyński, T., Malawski, M., Bubak, M. (2012). Managing entire lifecycles of e-Science applications in the GridSpace2 Virtual Laboratory – from motivation through idea to operable web-accessible environment built on top of PL-Grid e-Infrastructure. In M. Bubak, T. Szepieniec, K. Wiatr (Eds.), *Building a National Distributed e-Infrastructure – PL-Grid* (pp. 228–239). Springer Berlin, Heidelberg.

Douglas-Smith, D., Iwanaga, T., Croke, B.F.W., Jakeman, A.J. (2020). Certain trends in uncertainty and sensitivity analysis: An overview of software tools and techniques. *Environmental Modelling & Software*, *124*, https://doi.org/10.1016/j.envsoft.2019.104588.

EJML (2015). *Efficient Java Matrix Library*. https://code.google.com/p/efficient-java-matrix-library/.

Exp4j (2017). http://www.objecthunter.net/exp4j/.

JSON JavaScript Object Notation (n.d.). *Introducing JSON*. Retrieved 2021 from http://www.json.org/.

Jython (n.d.). *What is Jython?*. Retrieved 2020 from http://www.jython.org/.

Kochenderfer, M.J., Wheeler, T.A. (2019). *Algorithms for Optimization*, The MIT Press.

Kodein (n.d.). Retrieved 2021 from https://github.com/Kodein-Framework/Kodein-DI/.

Kotlin (2021). *Coroutines*. https://kotlinlang.org/docs/reference/coroutines-overview.html.

Ktor (n.d.). Retrieved 2021 from https://github.com/Kotlin/ktor.

Kuziak, R., Pietrzyk, M. (2012). Numerical simulation of controlled cooling of rails as a tool for optimal design of this process. *Computer Methods in Materials Science*, *12*(4), 233–243.

MathJax (n.d.). Retrieved 2020 from https://www.mathjax.org.

Milenin, A., Zalecki, W., Pernach, M., Rauch, Ł., Kuziak, R., Zygmunt, T., Pietrzyk, M. (2020). Numerical simulation of manufacturing process chain for pearlitic and bainitic steel rails. *Archives of Civil and Mechanical Engineering*, *20*(4), 107, doi.org/10.1007/s43452-020-00107-0.

MongoDB (n.d.). Retrieved 2021 from https://www.mongodb.com/.

Morris, M.D. (1991). Factorial sampling plans for preliminary computational experiments. *Technometrics*, *33*(2), 161–174.

Pianosi, F., Beven, K., Freer, J., Hall, J., Rougier, J., Stephenson, D.B., Wagener, T. (2016). Sensitivity analysis of environmental models: A systematic review with practical workflow. *Environmental Modelling and Software*, *79*, 214–232.

Plotly (n.d.). Retrieved 2020 from https://plot.ly/.

Prasanna, D.R. (2009). *Dependency Injection. Design Patterns Using Spring and Guide*. Manning.

Project Reactor (n.d.). Retrieved 2021 from https://projectreactor.io.

Radecki, M., Szymocha, T., Harężlak, D., Pawlik, M., Andrzejewski, J., Ziajka, W., Szelc, M. (2012). Integrating various grid middleware components and user services into a single platform. In M. Bubak, T. Szepieniec, K. Wiatr (Eds.), *Building a National Distributed e-Infrastructure – PL-Grid* (pp. 15–26). Springer Berlin, Heidelberg.

Rauch, Ł., Szeliga, D., Bachniak, D., Bzowski, K., Pietrzyk, M. (2014). Application of sensitivity analysis to grid-based procedure dedicated to creation of SSRVE. In M. Bubak, J. Kitowski, K. Wiatr (Eds.), *eScience on Distributed Computing Infrastructure. Achievements of PLGrid Plus Domain-Specific Services and Tools* (pp. 364–377). Springer Cham.

Razavi, S., Gupta, H.V. (2016). A new framework for comprehensive, robust, and efficient global sensitivity analysis: 1. Theory. *Water Resources Research*, *52*(1), 423–439.

React (n.d.). *React. A JavaScript library for building user interfaces*. Retrieved 2021 from https://facebook.github.io/react/.

ReactiveX (n.d.). Retrieved 2021 from http://reactivex.io/documentation/operators.html.

Redux (n.d.). Retrieved 2021 from https://github.com/reactjs/redux.

SALib (n.d.). *SALib. Sensitivity Analysis Library in Python (Numpy). Contains Sobol, Morris, and FAST methods*. Retrieved 2020 from http://salib.github.io/SALib/.

Saltelli, A., Chan, K., Scott, E.M. (2009). *Sensitivity Analysis*, Wiley.

Saltelli, A., Aleksankina, K., Becker, W., Fennell, P., Ferretti, F., Holst, N., Li, S., Wu, Q. (2019). Why so many published sensitivity analyses are false: A systematic review of sensitivity analysis practices. *Environmental Modelling and Software*, *114*, 29–39.

Scalarm (n.d.). Retrieved 2017 from https://github.com/Scalarm/scalarm.

Sheikholeslami, R., Razavi, S., Gupta, H.V., Becker, W., Haghnegahdar, A. (2019). Global sensitivity analysis for high-dimensional problems: How to objectively group factorsand measure robustness and convergence while reducing computational cost. *Environmental Modelling and Software*, *111*, 282–299.

SimLab (n.d.). *SIMLAB and other software*. Retrieved 2020 from https://ec.europa.eu/jrc/en/samo/simlab.

Szeliga, D., Kusiak, J., Rauch, Ł. (2012). Sensitivity analysis as support for design of hot rolling technology of dual phase steel strips. In J. Kusiak, J. Majta, D. Szeliga (Eds.), *Metal Forming 2012. Proceedings of the 14th International Conference on Metal Forming* (pp. 1275–1278). Wiley-VCH.

Szeliga, D., Sztangret, Ł., Kusiak, J., Pietrzyk, M. (2013). Optimization as a support for design of hot rolling technology of dual phase steel strips. In S.-H. Zhang, X.-H. Liu, M. Cheng, J. Li (Eds.), *AIP Proceedings of the 11th International Conference on Numerical Methods in Industrial Forming Processes: NUMIFORM* (pp. 183–191).

Xiong, Y., Chen, W.D., Apley, D., Ding, X. (2006). A non-stationary covariance-based Kriging method for metamodelling in engineering design. *International Journal for Numerical Methods in Engineering*, *71*(6), 733–756, https://doi.org/10.1002/nme.1969.