## Contents

# 1 Core Datatypes

In this section, the global architecture of the library developed in this thesis shall be elaborated. The full Python implementation of the library can be found in the digital appendix, only selected excerpts will be presented. The explanations are accompanied by graphical representations of the developed datatypes and procedures. Each element of the library has been assigned a unique geometric shape which will be used continuously in further graphic representations to support the comprehension of the interconnections of the library elements.

## 1.1 The *variable* datatype

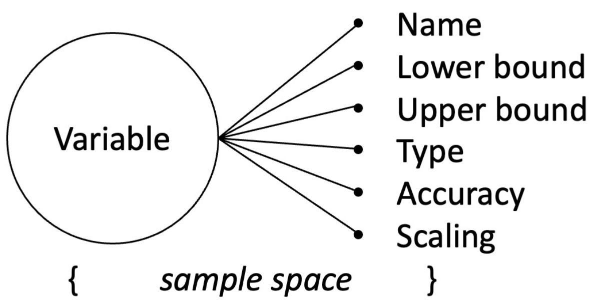


Figure 1.1: Graphic representation of the *variable* datatype, depicting its properties

A *variable* is assigned its *Name* from the Modelica model parameter it represents. Typically, the *Name* would be hierarchically derived through the model structure and stored as a string:

$$\textit{Name} \quad = \quad \text{'package.model.submodel.parameter'}$$

The remaining parameters are subject to the simulation scenario. *Lower* and *upper bound* define the validity interval for a *variable*. The *Type* property of a *variable* distinguishes between continuous and discrete variables. Although not necessary for the scope of this thesis and therefore not implemented, many more distinctions are possible e.g., such as types for switches. The discretization interval $N$ in equation **??** is stored in the *Accuracy* property, while the *Scaling* property enabled differentiation between linear and exponential intervals for discretization. The *sample space* is defined by equation **??**, storing discretized values $k$ from equation **??**. Although continuous variables feature an infinite sample space, certain optimization methods require a finite sample space. Therefore, the *sample space* of continuous *variables* is created by sampling a hundred values in the open interval from *lower bound* to *upper bound*. The header for the implementation of the *variable* datatype in Python for this library is shown in figure 1.2.

The *variable* datatype implements a method *transform*, which will return the discretized value of an input $x$ according to equation **??**, incorporating the propiertes stored in the *variable*.

```python
class variable:
    def __init__(self,
                 name: str,
                 lb: float | int,
                 ub: float | int,
                 val_type: str = 'real',  # possible values: 'real' or 'discrete'
                 acc: float = None,
                 scaling: str = None  # possible values: 'linear', 'exp'
                 ):
```

Figure 1.2: Header of the Python implementation of the *variable* datatype

$$\textit{transform}(x) = \begin{cases} x & \text{if } x \text{ is continuous} \\ \begin{cases} \text{round}\left(\frac{x}{\textit{Accuracy}}\right) & \text{if Scaling is linear} \\ \text{round}\left(\log_{\textit{Accuracy}}(x)\right) & \text{if Scaling is exponential} \end{cases} & \text{if } x \text{ is discrete} \end{cases} \tag{1.1}$$

This method is required to created the "index counterpart" for the *variable* later on when the *framework* is instantiated. This "index counterpart" is of identical *variable* dataype. In simple words, the *sample space* of the "index counterpart" is a kind of address register of where to find the actual requested value. It stores indices pointing to the actual values in the user-instantiated *variable*, as shown in 1.3.

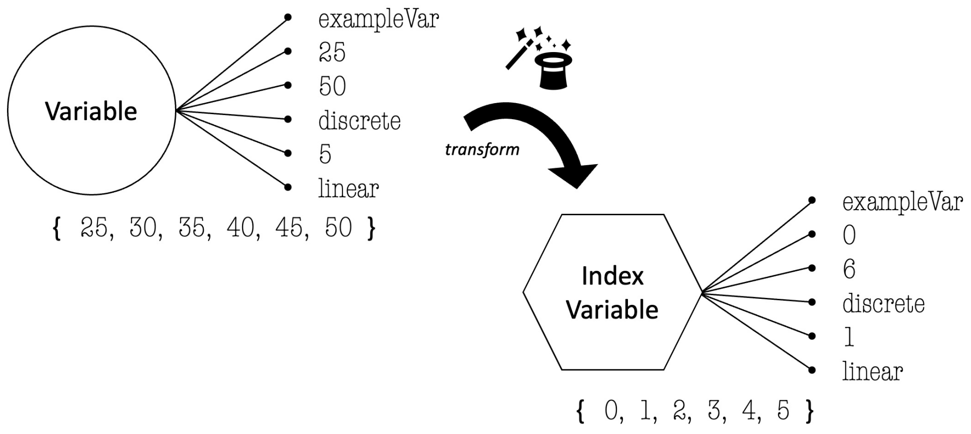


Figure 1.3: The transformation of an example optimization variable into its counterpart storing indices

The *lower* and *upper bounds* of the "index-counterpart" now represent the minimum index - which is always 0 - and the maximum index indicating the power of the *variable*'s *sample space*. When given an index $i$ of an *variable* in the transformed domain, its real world value $v$ is retrieved by:

$$v \quad = \quad \textit{sample space}\left[\,\textit{sample space}_{\,index}[i]\,\right] \tag{1.2}$$

## 1.2 The *variable array* datatype

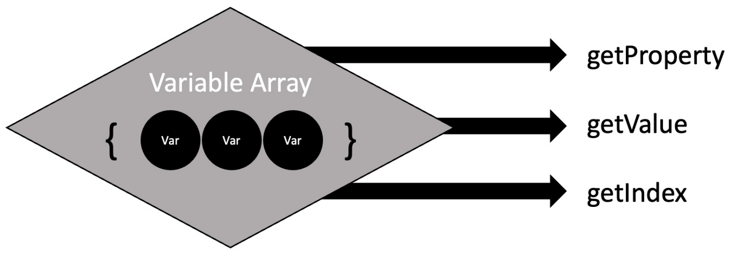


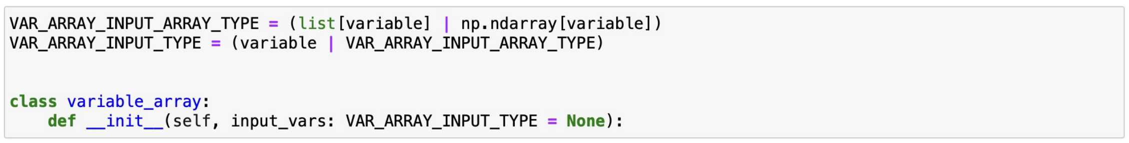
Figure 1.4: Graphic representation of the *variable array* datatype with its methods

The *variable array* consists primarily of an array in which the *variables* are stored. The core functionality of the *variable array* is to execute methods on grouped *variables*. These methods are summarized in table 1.1.

| Method | functionality |
|---|---|
| getProperty | returns the value of a specified property of a *variable* |
| getValue | given an index it returns the associated value of a *variable* according to equation 1.2 |
| getIndex | returns the associated index of a transformed *variable* by executing its *transform*-method described in 1.1 on a given value |

Table 1.1: Description of the functionality of the *variable array* methods

The rather trivial header of the Python implementation including the associated type definitions is shown in figure 1.5 [1]



```
VAR_ARRAY_INPUT_ARRAY_TYPE = (list[variable] | np.ndarray[variable])
VAR_ARRAY_INPUT_TYPE = (variable | VAR_ARRAY_INPUT_ARRAY_TYPE)


class variable_array:
    def __init__(self, input_vars: VAR_ARRAY_INPUT_TYPE = None):
```

Figure 1.5: Header of the Python implementation of the *variable array* datatype, including the associated type definitions

[1]In depicted code throughout this thesis, 'np' references to the NumPy library [**numpy**]

## 1.3 The *constraint* datatype

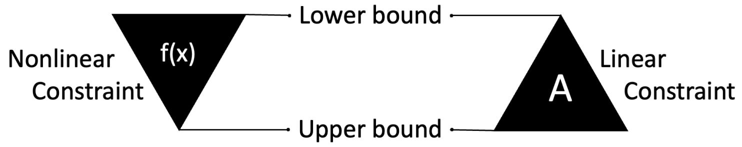


Figure 1.6: Graphic representation of the *constraint* datatype

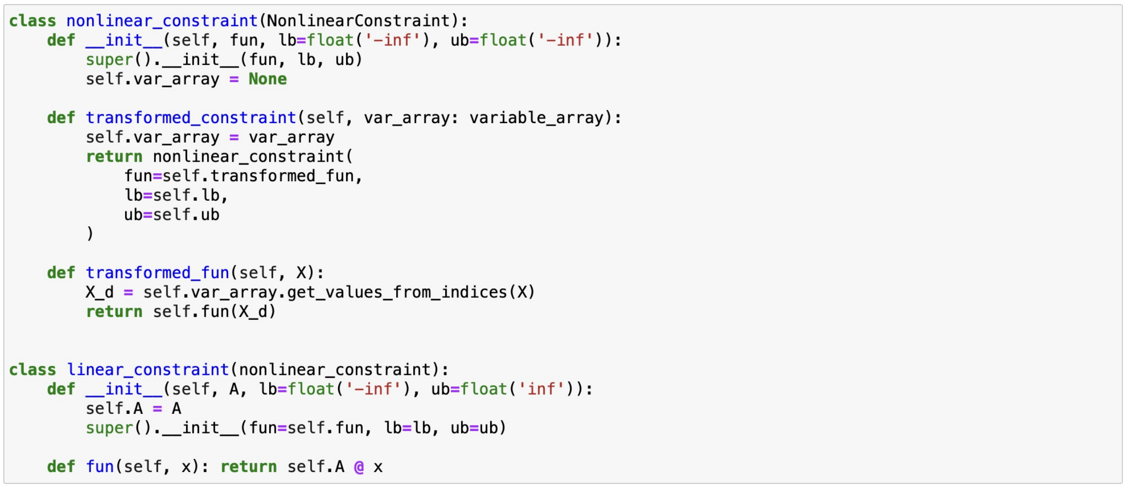
The *constraint* datatype is designed as a wrapper around SciPy's NonlinearConstraint [**scipy-nonlinearconst** object. The *constraint* datatype has been seperated into *LinearConstraint* and *NonlinearConstraint*. This enables accelerated matrix execution of the former. They both feature a *lower* and *upper bound* property. The *NonlinearConstraint* is defined with a *fun* property, a function $f(x)$ representing the constraint. It is important that this function only takes single input argument. If more arguments are needed, the construction an external wrapping function is recommended. The input argument needs to be of identical shape as *lower* and *upper* bounds.

$$\textit{lower bound} \;\leq\; f(x) \;\leq \textit{upper bound} \tag{1.3}$$

Similarly, the *LinearConstraint* stores a matrix *A* instead of the *fun* property. The dot product of a variable vector $x$ and the matrix *A* forms a linear equation system which defines the valid value range of the variable vextor. This dot product has to be of equal shape as the *lower* and *upper bounds*, therefore the matrix *A* has to be designed accordingly.

$$\textit{lower bound} \;\leq\; A \cdot x \;\leq \textit{upper bound} \tag{1.4}$$

The complete implementation of the *constraint* datatype is shown in figure 1.7.



```
class nonlinear_constraint(NonlinearConstraint):
    def __init__(self, fun, lb=float('-inf'), ub=float('-inf')):
        super().__init__(fun, lb, ub)
        self.var_array = None

    def transformed_constraint(self, var_array: variable_array):
        self.var_array = var_array
        return nonlinear_constraint(
            fun=self.transformed_fun,
            lb=self.lb,
            ub=self.ub
        )

    def transformed_fun(self, X):
        X_d = self.var_array.get_values_from_indices(X)
        return self.fun(X_d)


class linear_constraint(nonlinear_constraint):
    def __init__(self, A, lb=float('-inf'), ub=float('inf')):
        self.A = A
        super().__init__(fun=self.fun, lb=lb, ub=ub)

    def fun(self, x): return self.A @ x
```

Figure 1.7: The Python implementation of the *constraint* datatype

## 1.4 The *framework* datatype

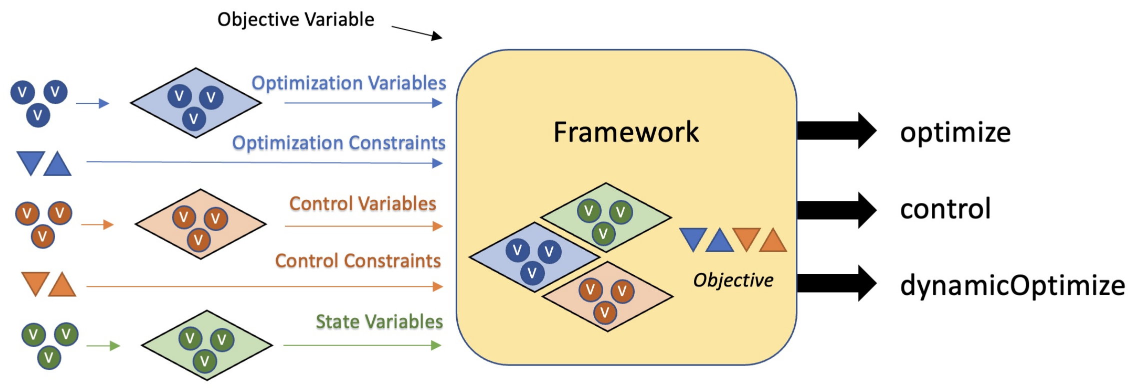


Figure 1.8: Graphic representation of the *framework* datatype and its initialization requirements

The *framework* datatype acts as the "brain" of the library. Its main purposes are to control the optimization procedure and exchange data between the transformed domain and its origin. Additionally, it provides an interface that enables straightforward user interaction by executing the methods of the *framework* datatype.

To begin the initialization process of the *framework*, one or several lists or arrays of *variable* instances are passed when invoking the constructor of the *framework*. The *framework* separately stores optimization *variables*, control *variables* as well as state *variables*. The latter is only required for optimal control, storing information about the state of an optimal control scenario that is not captures by the decision variables. It is also possible to pass the *variables* as a *variable array*, although this is unnecessary user effort. The *framework* will convert a list or array of *variables* automatically into a *variable array* and store it. It is not necessary to pass each a collection of optimization, control and state *variables*, the optimization scenario dictates the required information. However, the minimum is either an optimization or a control *variable*. In addition to the *variables* and *constraints*, the name of the objective variable as specified in the Modelica simulation needs to be passed as well. The header of the constructor of the *framework* datatype for the Python implementation is shown in figure 1.9.

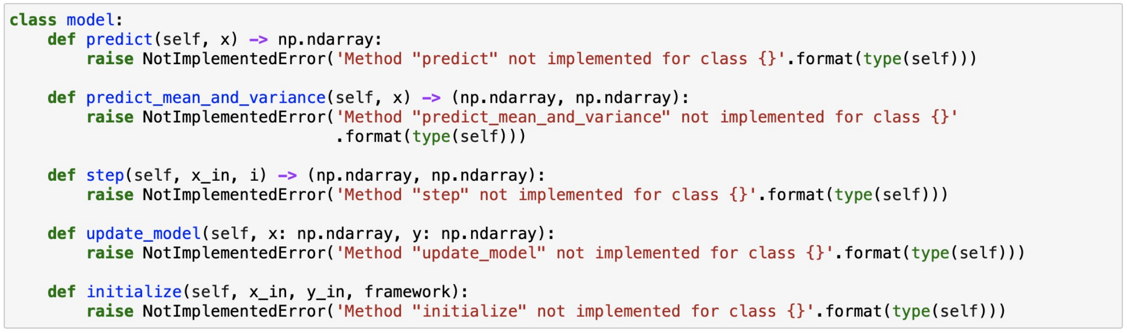
```python
CONSTRAINT_VARIABLE_TYPE = (linear_constraint | nonlinear_constraint)
CONSTRAINT_ARRAY_TYPE = (list[CONSTRAINT_VARIABLE_TYPE] | np.ndarray[CONSTRAINT_VARIABLE_TYPE])
CONSTRAINT_TYPE = CONSTRAINT_VARIABLE_TYPE | CONSTRAINT_ARRAY_TYPE


class framework(sub_framework):
    def __init__(self,
                 objective_variables: VAR_ARRAY_INPUT_TYPE,
                 optimisation_variables: VAR_ARRAY_INPUT_TYPE = None,
                 control_variables: VAR_ARRAY_INPUT_TYPE = None,
                 state_variables: VAR_ARRAY_INPUT_TYPE = None,
                 optimisation_constraints: CONSTRAINT_TYPE | CONSTRAINT_ARRAY_TYPE = None,
                 control_constraints: CONSTRAINT_TYPE | CONSTRAINT_ARRAY_TYPE = None
                 ):
```

Figure 1.9: The Python implementation of the *framework* datatype, including associated type definitions [2]

The *variable arrays* that have been created in the first step of the initialisation will next be transformed as described in equation 1.1. The transformed *variables* are then also stored in their own respective *variable arrays*. After instantiating a *framework* object, the *constraints* are added to the framework with the respective methods provided for them. Again, it is to be distinguished between optimization and control *constraints*. The importance of the differentiation between optimization and control properties will become apparent later in this section. The *optimize*, *control* and *dynamicOptimize* methods of the *framework* initiate the optimization respective process. Their functionality will be discussed at length in sections 3 of this chapter.

## 2 Interfacing a simulation

Firstly, a base interface called *model* has been created. Interfaces are useful in programming because they provide a way to define a contract or set of rules that classes and datatypes must adhere to. This promotes code modularity, as classes and dataypes can be developed independently as long as they implement the required interface. Interfaces also enable polymorphism[3], allowing objects of different classes that implement the same interface to be treated interchangeably, enhancing code flexibility and reusability. The *model* interface is used not only for the execution of Modelica simulation models but also for approximation tools of the former, a topic that will be further elaborated on in the upcoming section. The major advantage of providing the *model* interface architecture is that the optimization library can be extended to exchange data with any other simulation environment such as for example MATLAB, AspenX, Dymola - or even any general input-output based program, allowing optimization of e.g. Microsoft Excel sheets, online sources or custom tools - by simply implementing a data exchange module using the *model* interface. This thesis however focuses solely on the data exchange with Modelica simulation models. Figure 2.1 shows the implementation of the *model* interface for the Python library. The Python programming



```python
class model:
    def predict(self, x) -> np.ndarray:
        raise NotImplementedError('Method "predict" not implemented for class {}'.format(type(self)))

    def predict_mean_and_variance(self, x) -> (np.ndarray, np.ndarray):
        raise NotImplementedError('Method "predict_mean_and_variance" not implemented for class {}'
                                  .format(type(self)))

    def step(self, x_in, i) -> (np.ndarray, np.ndarray):
        raise NotImplementedError('Method "step" not implemented for class {}'.format(type(self)))

    def update_model(self, x: np.ndarray, y: np.ndarray):
        raise NotImplementedError('Method "update_model" not implemented for class {}'.format(type(self)))

    def initialize(self, x_in, y_in, framework):
        raise NotImplementedError('Method "initialize" not implemented for class {}'.format(type(self)))
```

Figure 2.1: Python implementation of the *model* interface

language does not explicitly provide support interfaces, as for example Java or C# do. Instead it

[2]The VAR_ARRAY_INPUT_TYPE definition corresponds to the one in figure 1.5

[3]Polymorphism in programming refers to the ability of objects of different types to be treated as objects of a common base type. This allows for code to be written in a more generic and flexible manner, as operations can be performed on objects without needing to know their specific types.

offers a library for so-called abstract classes, as for example C++ and Rust[4] do as well. For this Python implementation though, the *model* has been designed as a non-abstract superclass with a set of methods, each of of them yielding in a runtime error by default. A class or datatype implementing the *model* interface does not need to implement each method but only the ones it has to in order to fulfil its expected functionality. If a *model* subclass is used out-of-context, the default implementation of its *model* superclass will raise a runtime error.

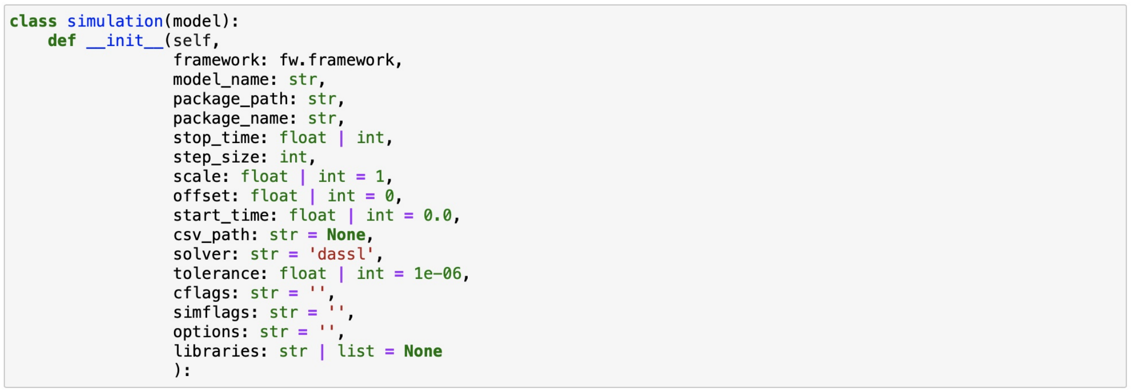
Based on the above runtime analysis, a subclass of the *model* interface called *simulation* has been implemented. It is designed to handle all data exchange such as the input and output communication of the Modelica simulation model and manipulation of parameter values as well as setting simulation options and extracting information from the model. The handling of input and output values is achieved via CSV files that can be read into and loaded after the simulation. Parameter manipulation and settings of simulation options are communicated using appropriate flags when executing the binary from the CLI. Information from the model can be extracted by parsing the XML[5] file. It is to be noted that the implementation is tested on UNIX-based systems[6]. It is therefore recommended to use the WSL[7] feature of Windows for the execution of scripts that use this library, since the compilation process of the MSRS and the subsequent execution call for the simulation binary possibly differ. The header of the *simulation* subclass constructor is shown in figure 2.2 and its arguments are explained in table 2.1.

[4]In the Rust programming language, there the trait type is used to implement interfaces. Interfaces and trait types are not completely interchangeable though.

[5]An XML (eXtensible Markup Language) file is a text-based file format used for storing and transporting structured data. It uses tags to define elements and their hierarchical relationships, allowing for the representation of complex data structures in a human-readable and machine-readable format.

[6]UNIX-based systems, such as Linux and macOS, are operating systems that are built on the principles of the original UNIX operating system. Microsoft Windows has a different architecture and system design compared to UNIX-based systems, and it has its own set of tools and utilities.

[7]WSL stands for Windows Subsystem for Linux. It is a compatibility layer in Windows that allows you to run a Linux environment directly on your Windows machine.



```python
class simulation(model):
    def __init__(self,
                 framework: fw.framework,
                 model_name: str,
                 package_path: str,
                 package_name: str,
                 stop_time: float | int,
                 step_size: int,
                 scale: float | int = 1,
                 offset: float | int = 0,
                 start_time: float | int = 0.0,
                 csv_path: str = None,
                 solver: str = 'dassl',
                 tolerance: float | int = 1e-06,
                 cflags: str = '',
                 simflags: str = '',
                 options: str = '',
                 libraries: str | list = None
                 ):
```

Figure 2.2: Python implementation of the *simulation* submodel that handles the interaction with Modelica simulation models

Table 2.1: Explanation of the arguments to initialize a *simulation* instance

| Argument | optional | type | default | description |
|---|---|---|---|---|
| framework | no | *framework* | | Passes the optimization *framework* |
| model_name | no | String | | Name of the Modelica model |
| package_path | no | String | | Path to the package of the model (if the model is not part of a package, specify model path instead) |
| package_name | yes | String | ” | Name of the package of the model |
| libraries | yes | str, list | ” | Additional libraries required |
| csv_path | yes | str | None | Path to CSV input files, if required |
| start_time | yes | float, int | 0.0 | Start time of the simulation |
| stop_time | no | float, int | | Stop time of the simulation |
| step_size | no | float, int | | Interval between two simulation steps |
| scale | yes | float, int | 1 | Scaling of the objective variable. |
| offset | yes | float, int | 0 | Offset of the objective variable. |
| solver | yes | String | 'dassl' | Solver used for Modelica DAEs |
| tolerance | yes | float, int | 1e-06 | Specifies tolerance for the Modelica solver |
| cflags | yes | String | ” | Additional compilation flags |
| simflags | yes | String | ” | Additional simulation flags |
| options | yes | String | ” | Other flags for the exectution of the model See Modelica Documentation [**openmodelica**] |

## 2.1 Approximating the simulation behaviour

### 2.1.1 Gaussian Process Regression

Based on the *model* interface from the previous section, the functionality of the Gaussian process regression has been implemented in two varieties. To examine possible runtime improvements, the library supports GPR execution on both the CPU and the GPU[8]. Modern GPUs are optimized for parallel processing and have a highly parallel structure that makes them more effective than CPUs for algorithms where processing of large blocks of data is done in parallel. The GPU implementation of the GPR object in the library is built around the *gpflow*[**gpflow**] library for Python, which itself is based on *TensorFlow*[9]. For CPU-based computation, the implementation has been built around the widely-used *scikit-learn*[**skikit-learn**] library. Furthermore, the possibility for dimensionality reduction has been added. As a pre-requisite for adequate principal component analysis, a scaling functionality has been added as well. Both of these additions are based on the *scikit-learn* library. As seen in figure 2.3, a custom kernel for the GPR can be passed when initializing the submodule.

```
class gaussian_process(model):
    def __init__(self, x_in: np.ndarray = None, y_in: np.ndarray = None, kernel=None, use_gpu: bool = False,
                 sparse: bool = False, reduce_dims: bool | float = False, scale: bool = False):
```

Figure 2.3: Constructor of the Python implementation of the Gaussian process regression submodel.

### 2.1.2 Neural network regression

Neural networks are capable of representing very complex interrelationships. This comes however at the expensive of high degree of difficulty when designing their parameters. The architecture and usage of neural networks has become a major field of research with the developments in machine learning in the last decade. Therefore, their set-up requires profound knowledge and experience to achieve useful results. To accommodate for this, the Python implementation of the library features a 'neural-network-generator', which uses Bayesian optimization[10] to generate a appropriate neural network design based on a given dataset. The generator uses various features such as k-fold validation, regularization, momentum and dropout to prevent overfitting and improve accuracy[11]. The implementation of the submodule is based on *Keras*, an open-source neural network API written in Python that provides a high-level abstraction for building and training deep learning models. At the core, *Keras*[**keras**] also uses the *TensorFlow* library.

[8]A GPU (Graphics Processing Unit) is a specialized electronic circuit designed to rapidly process and manipulate memory to accelerate the creation of images in a frame buffer intended for output to a display.

[9]TensorFlow is an open-source software library developed by Google for numerical computation and large-scale machine learning applications. It features great support for GPU-based computation.

[10]The neural network generator is actually built with the library implementation of this thesis, which accounts for the versatility of the library.

[11]The *Keras* documentation provides extensive information on these techniques

The implementation of the neural network regression has also been realized upon the *model* interface. The generator has been integrated in this submodule. Depending on which arguments are passed when initializing the submodule, either an existing model can be passed and trained or the generator is invoked to create a suitable model for the supplied dataset, as figure 2.4 illustrates.

```python
class neural_network:
    def __init__(self, X, Y,    # compulsory
                 # use these three arguments to pass an existing model
                 model=None, epochs: int = None, batch_size: int = None,
                 # use these arguments to invoke the neural network generator
                 k=5,  # for k-fold cross validation
                 max_layers=5,  # num hidden layers
                 max_probability_dropout=0.5,
                 activation_fxns=None,  # default: ['relu', 'tanh', 'sigmoid', 'elu']
                 max_units_per_layer=8,  # 2**n, e.g. 8 --> 256
                 max_l1=0.01, max_l2=0.01, max_momentum=0.9, max_learning_rate=0.9, max_epochs=300,
                 max_batch_size=7, # 2**n, e.g. 8 --> 256
                 reduce_dims: bool | float = False
                 ):
```

Figure 2.4: Constructor of the Python implementation of the neural network submodel.

# 3 Optimization and control framework

## 3.1 Optimization

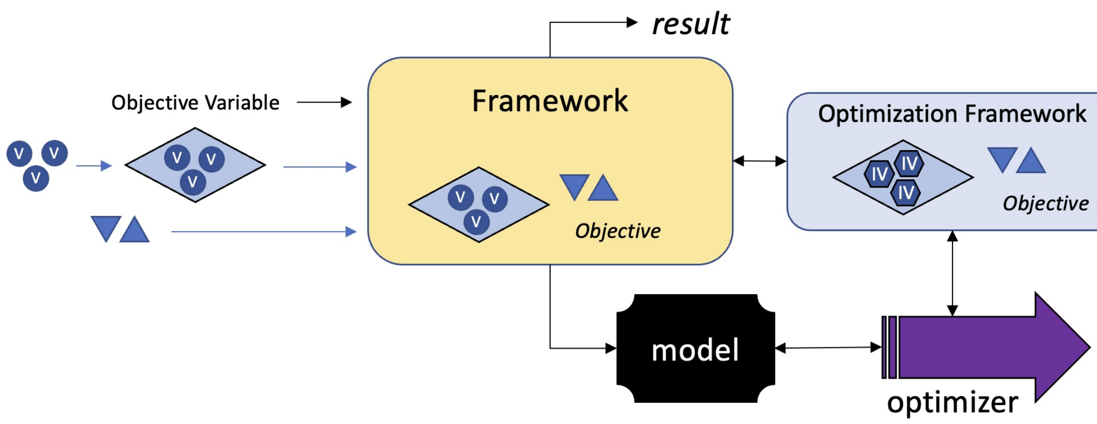


Figure 3.1: Graphic representation of the optimization procedure

Numerous methods for optimization are available. To accommodate their implementation, a shared *optimizer* module has been designed to provide basic optimization functionality. It provides solutions to store data during the optimisation process and offers a communication with the *framework* and the *model*. The *model* subject to optimisation can be any object subclassing the *model* interface, such as for example a *simulation* or a approximation thereof as described in the above section. Furthermore, a naive implementation of parallel execution has been included, in order to use most of the available hardware capabilities. The *framework* communicates with the *optimizer* via a sub-framework containing the index-*variables* presented in section 1. In practice, this means that the optimization is executed using indices, not actual values. The model will in return communicate with the actual *framework* in order to access the the values given by the optimizer in form of indices, as illustrated in figure 3.2. To inspect the optimization progress after completion, a boolean *log*-flag can be set during initialization. The complete implementation of the *optimizer* module including its submodules can be found in the digital appendix for further inspection. Following the same logic as with the *model* interface, the *optimizer* can be subclassed with custom implementations of optimization techniques, allowing for great flexibility.

```python
class optimizer:
    def __init__(self, iterations: int, objective_model, framework: fw = None,
                 use_scipy: bool = False, parallel: bool | int = False, log: bool = False):
```

Figure 3.2: Constructor of the *optimizer* module

**Random search** The random search algorithm presented in figure **??** is implemented for two applications. If a mean and a variance of the decision variables are passed, random samples will be drawn as per the normal distribution defined. Otherwise, samples will be drawn uniformly between the specified lower and upper bounds of the respective *variable*. Each sample group will be checked for constraint compliance. In case a group of samples fails to adhere to the constraints, it will be discarded and a a new group will be sampled. This process is limited to $10^7$ iterations, after which an error will be raised in order to avoid infinite loops caused by maliciously defined constraints. Furthermore, the *random search* submodule implements a *generate x*-method that sample a specified amount of samples from the domain space within the bounds and constraints and will return them without evaluation.

**Grid search** As the algorithm in figure **??** describes, the grid search partitions the domain space into a equally spaced parameter grid that will be evaluated by the *model*. To fit continuous *variables* into the parameter grid, they will be discretized into one hundred samples. There are multiple possibilities to initialize a *grid search* submodule:

- When a total amount of iterations is passed, each dimension in the parameter grid is subdivided proportionally to the size respective domain space. This means that *variables* containing a larger sample space will recieve a smaller resolution in the parameter grid.
- The desired number of iterations per grid dimension can be passed is a list directly.
- A boolean flag *sweep* can be set during initialization of the submodule. If the flag is set, no iterations need to be specified. The parameter grid will contain the complete sample space of each *variable*. While this exhaustive search guarantees a global optimum, the computational requirements move from heavy to infeasible quickly.

The areas of the parameter grid that fail to comply with the constraints will be excluded from the parameter grid. As with the *random search* module, the *grid search* module also implements a *generate parameter grid*-method that returns the parameter grid without evaluation.

**Staged search** In order to combine the strenghts of both the *random search*, they can be combined into a *staged search*. With a passed list of iterations per grid dimension, the *grid search* will first broadly evaluate the whole domain space. The most promising results from this search are used to compute a mean and a variance, which will then be passed to a subsequent *random search* for fine-tuning.

**Gradient-based optimization** For the deployment of deterministic optimization methods, the *optimize*-toolkit of *SciPy*[**scipy**] has been implemented as a submodule of the *optimizer*. In the implementation for this thesis, the BFGS algorithm was used, however any method that the *SciPy* toolkit supports can be used. In fact, the exact algorithm supported by the *SciPy* kit is the L-BFGS-B algorithm. It is an evolution of the original BFGS algorithm with improved memory efficiency and the ability to handle bounds. If another method is desired, it needs to be passed when initializing the module. Since the BFGS algorithm was not designed to be used with constraints, the objective function was wrapped around a helper function that assigns a fixed value when the constraints are breached. Since the optimization is being done by the *SciPy* kit, the *log*-flag will be ignored for this optimization method.

**Genetic optimization** Heuristic optimization methods such as genetic algorithms provide a very powerful approach to global optimization of non-convex and mixed-integer problems. Using the *DEAP* [**deap**] Python framework as a basis, a genetic algorithm has been developed to work in conjunction with the library architecture of this thesis. The key is the encoding of the chromosomes. For each decision index *variable*, the number of bits to represent the upper boundary is determined. The bits of each *variable* are then concatenated into a chromosome, the individual. A population of a given number of individuals is randomly generated and evaluated using a tournament. To decode an individual for evaluation, the chromosome is split into its binary variables again and converted to decimal. Due to crossover and mutation, the bits will flip thereby changing the values. The well-performing changes will be applied when creating a new population of individuals. The necessary parameters such as population size, tournament contestants as well as crossover and mutation probability are passed when initializing the submodule. The mutation probability can be either specified as a single number or as a tuple of numbers, which will then set the bounds for a randomly sampled mutation probability for each generation. If an individual breaches the constraints, a dynamic penalty will be imposed depending on how far the bound of a constraint has been crossed. This will ensure that the average fitness of a generation will not be impacted by a minority of individuals being slightly invalid.

**Bayesian optimization** The recent advances in machine learning have made Bayesian optimization a popular choice for hyperparameter optimization of machine learning models. It provides an intelligent way to navigate through a large domain space and thereby reducing the execution of a costly-to-evaluate function. By default, the implementation of the submodule will use a Gaussian process as a function approximator. However any *model* submodule can be passed under the premise that it supports a method that will return the predicted mean and variance of the approximated function at a given point. Furthermore, an optimization method for the acquisition function needs to be specified, which can be any of the above *optimizers*. The submodule features an *early stopping* flag, which is set to True by default. When the optimizer of the acquisition function chooses the same point twice for evaluation, a global optimum is reached since there is no other room for improvement left. Therefore, the optimization can be stopped without evaluating the remain iterations, as the optimizer would evaluate the same point over and over again. The *early stopping* flag can be switched to False during initialization.

## 3.2 Optimal control

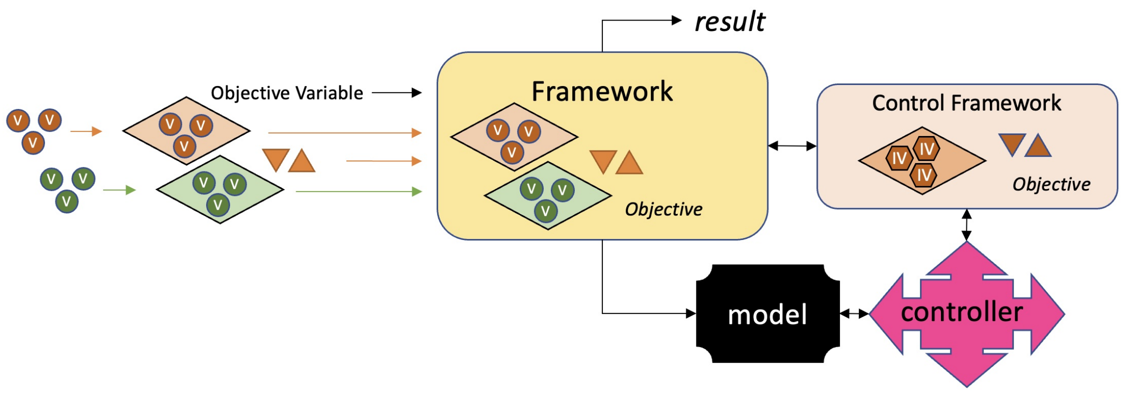


Figure 3.3: Graphic representation of the optimal control procedure

The *optimizers* presented in the above section can also be applied on optimal control scenarios, determining optimal values for each time step in a simulation. For this application, the *controller* module has been designed. As the *optimizer*, it serves the purpose of storing data during the process and handling the communication with the *framework* and the objective *model*. For the purpose of this library, two types of controllers have been implemented, which will be elaborated in the following part of this section. In the same fashion as the *model* and *optimizer*, the *controller* can be extended with custom implementations in any manner. The *controller* communicates with the control-subframework instantiated by the *framework* with the index-*variables* in the same fashion as the *optimizer*. A key difference between the *optimizer* and the *controller* is the presence of state *variables* in the latter. In contrast to a global parameter optimization, the control objectives and their constraints can be influenced by other variables of the simulation. Usually, these dependencies are defined in a state-space system. Furthermore, the *controller* is designed to be a subclass of the simulation. This enables the conjunction of an *optimizer* and a *controller* for dynamic optimization, as will be discussed in the following section.

**Optimal step-control** To apply the optimizers presented in a scenario where *variables* need to be optimized at each time-step of the simulation, the *model* is called step by step instead of being executed as one. The objective of the *optimizer* passed during initialization is the difference of the objectives between two steps.

**Genetic control** Instead of calling the *model* for each step, the genetic *optimizer* presented in the above section has been adapted to for control. In this scenario, the chromosome stores a value for each *variable* and each iteration of the model, resulting in only one call of the *model* to evaluate all of the values for the *variables* in the individual. The *optimizer* will then continue refining the *variables* based on the objective *model*. A key strenght of the genetic *controller* is that it allows for planning, as it is evaluated over the whole simulation horizon, in contrast to the step-wise optimal controller. Therefore it is able to find solutions that may be inferior at the current time-step, but lead to better values in the future. The arguments for the construction of the genetic *controller* are identical to the arguments of the genetic *optimizer*, since it is based on it.

## 3.3 Dynamic optimization

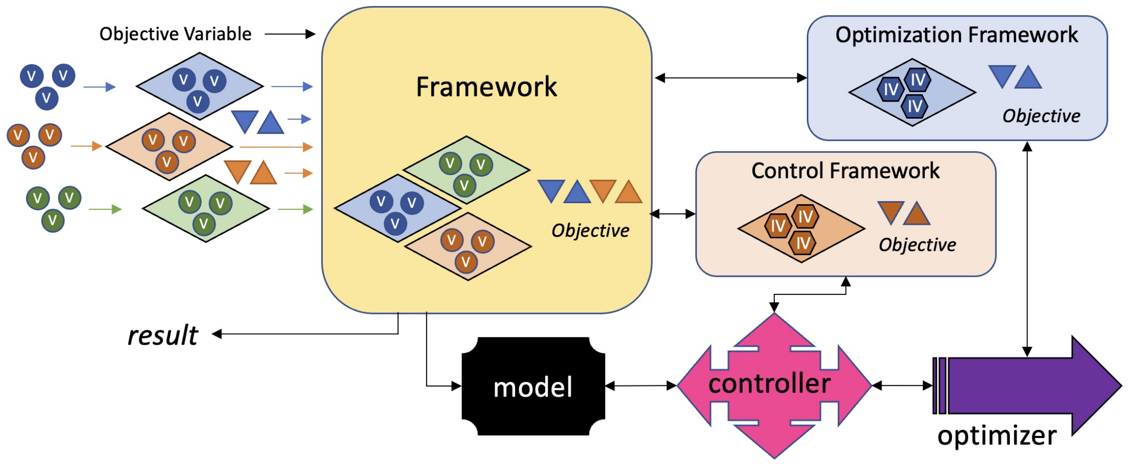


Figure 3.4: Graphic representation of the dynamic optimization procedure

Dynamic optimization is an iterative approach to solving optimal control problems where the global parameters of the model are optimized concurrently with the determination of the optimal control inputs. In contrast to standard approaches that utilize fixed predefined parameters, dynamic optimization tunes the model parameters specifically to optimize the performance of the controller. This is achieved by first defining a model that includes both global parameters to optimize as well as the constraints and objectives of the optimal control problem. Then, in each iteration, the optimal control problem is solved using the current global parameter estimates. After solving the optimal control problem, the global parameters are updated to improve the objective function based on the previous solution. This coupled process of optimizing the parameters and solving the optimal control problem is repeated, dynamically adapting the global model until convergence criteria are met. The key benefit of dynamic optimization is that the model is tuned specifically for the control task at hand rather than based on generic offline parameter identification. This enables co-optimization of the model parameters along with the control inputs tailored to the specific optimal control problem objectives.

Because the *controller* module is subclassed from the *model* object, it allows to be used within an *optimizer* as an objective model, while the controller communicates with the actual *model*. The *optimizer* and *controller* communicate with the *framework* through their respective sub-*frameworks*, as illustrated in figure 3.4.

# 4 The library in application

The usage of the library developed in this thesis is relatively straightforward. A short instruction guide on the intended use of the library will now conclude this section presenting its functionality. This guide is accompanied by a graphical example in figure 4.1. It is to be noted that the purpose of the code in figure 4.1 is solely to showcase the library, there is no further meaning in the set numerical values. For actual working examples, please refer to the 'examples' folder.

**(1) Define the *variables*** Firstly, the decision and state variables for a given optimization scenario need to be defined. It can sometimes be helpful to store values like boundaries in separately and define helper functions to avoid having to make multiple changes in each *variable* definition when updating values, as is done the first few lines of code in figure 4.1. Then create instances of *variables* according to section 1. Whether optimization, control or state variables are necessary is dependant on the optimization scenario. Group the related *variables* in a list. Finally, store the name of the objective variable as a string.

**(2) Define *constraints*** Create the matrices $A$ for linear constraints and define the necessary functions for nonlinear constraints. Boolean constraints can be realized by defining the constraint and returning the boolean value, then initializing lower and upper boundaries equal to $1$. Just as with the declaration of *variables*, it has to be distinguished between optimization constraints and control constraints. For optimization constraints, the shape of a matrix $A$ as well as the input argument $x$ of a constraint function need to have equal dimensionality as the array of optimization *variables*. On the other hand, the shape of matrix $A$ and the input argument $x$ equals the dimensionality of the concatenation of the control and state *variables*. Therefore, in the context of control constraints, the state *variables* may also be used.

**(3) Initialize the *framework*** By passing the lists of *variables* defined earlier on as input arguments, an instance of a *framework* can be created. If for example no state *variables* have been created, they can simply be omitted as an argument when creating a *framework* instance. After instancing the *framework*, use the methods *add_optimization_constraint* and *add_control_constraint* to add optimization and control constraints respectively to the *framework*. In case several constraints have been designed, they can either be passed as a list or by multiple calls of the add-method.

**(4) Create *model* and *optimizer* instances** The model subject to optimization can either be called through a *simulation* instance or an approximation thereof as introduced in section 2.1. This instance will then be passed when creating an *optimizer* or *controller* instance of choice, as explained in section 3.

**(5) Run the optimization** The optimization will be executed by calling the appropriate method of the *framework* as described in section 3 and passing the *optimizer* or *controller* instance to it. If the optimization process is to be displayed, the log-flag can be set to true when calling the execution-method of the *framework*. The result of the optimization process will be returned from the called method of the *framework*.

```python
import thesis_library as lib

# declare scenario settings
global_bound, local_bound = 69, 128
interval1, interval2 = 3.145, 2.718
helper_func = lambda gb, i: (gb // i) * i

# define parameters to be optimized
param1 = lib.variable(name='example.var1', lb=0, ub=helper_func(global_bound, interval1),
                  val_type='discrete', scaling='linear', acc=interval1)
param2 = lib.variable(name='example.var2', lb=0, ub=helper_func(global_bound, interval2),
                  val_type='discrete', scaling='exp', acc=interval2)
param3 = lib.variable(name='example.var3', lb=0, ub=local_bound, val_type='real')
param4 = lib.variable(name='example.var4', lb=0, ub=local_bound, val_type='real')
opt_params = [param1, param3]
ctrl_params = [param2]
state_params = [param4]

# define objectives
obj = 'objective.variable'

# define constraints
A1, A2 = [1, 1], [2, -3]
opt_constr1 = lib.linear_constraint(A=A1, lb=0, ub=global_bound)
opt_constr2 = lib.linear_constraint(A=A2, lb=0, ub=global_bound)

def nonlinfxn(x): return math.sin(x) ** 2 + math.cos(x) ** 2
ctrl_constr = lib.nonlinear_constraint(fun=nonlinfxn, lb=-5, ub=5)

# initialize framework
fw = lib.framework(optimisation_variables=opt_params, control_variables = ctrl_params,
                   state_variables = state_params, objective_variables=obj)
fw.add_optimisation_constraint([opt_constr1, opt_constr2])
fw.add_control_constraint(ctrl_constr)

# initialize simulator and optimizer
sim = lib.simulation(
    model_name='ExampleModel', package_name='ExamplePackage', package_path='/path/to/package/package.mo',
    stop_time=666, step_size=1, framework=fw)
ctrl = lib.genetic(sim=sim)
opt = lib.random_search(iterations=42, objective_model=ctrl)

# run optimization
result = fw.dynamic_optimize(optimizer=opt, log=True)
```

Figure 4.1: An example of how to use the Python library to perform a constrained dynamic optimization on a Modelica simulation model