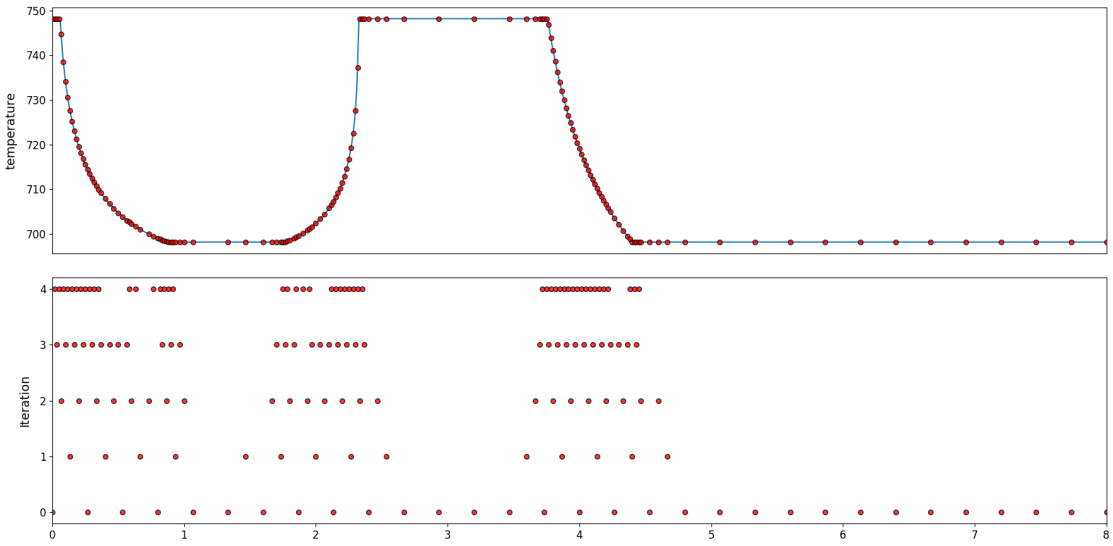
User's Guide:
GDOPT - General Dynamic Optimizer v.0.1.3

A Python Environment for Optimizing Dynamic Models



Author:
Linus Langenkamp

Last updated on:
October 18, 2024

Contents

1 Introduction

Dynamic optimization is a branch of mathematical optimization that deals with systems that evolve over time. The goal is to find optimal state trajectories satisfying given differential equations by adjusting the controls over a certain time horizon. These so-called optimal control problems are of great importance in industrial applications and occur in various areas such as industrial biochemistry, economics and aerospace engineering.

This guide presents a high level Python modeling environment for solving dynamic optimization problems using local collocation methods and adaptive mesh refinement techniques under the hood. The following sections provide step-by-step instructions on setting up, modeling and optimizing dynamic models along with analyzing the optimal solution.

1.1 Overview of the Framework

The dynamic optimization framework, GDOPT, was developed to enable easy modeling and optimization of dynamic systems. It consists of two main components:

A Python front-end that provides the user with a high-level interface to model the continuous problem and set flags, such as enabling mesh refinements or setting tolerances. A C++ back-end that performs the computationally intensive task of solving the resulting large-scale nonlinear problem (NLP) with IPOPT (Interior Point OPTimizer). In addition, a novel mesh refinement algorithm (L2-Boundary-Norm) is implemented that performs an h-method. This leads to high-resolution solutions at discontinuities, steep sections or corners of control variables.

The framework is very flexible and expressive, but retains a simple and accessible syntax. The front-end has many special features that are presented in this guide.

Once a dynamic optimization problem is defined, the framework discretizes it and applies numerical techniques to transform it into a sparse NLP that can be efficiently solved. In the following section, this process is described in principal.

1.2 How the Framework Solves Dynamic Models

The framework solves continuous dynamic optimization problems by discretizing them over the specified time horizon. This is achieved by the following steps:

- Time discretization: The continuous time horizon is discretized into N equidistant grid points.
- Dynamic constraints: Within each of the resulting intervals, a RadauIIA collocation scheme is constructed, inserting $M = 1, \ldots, 36$ nodes on each interval. Consequently, the states of the model are represented by a polynomial of degree M on each interval, discretizing the dynamics.

- Non-dynamic constraints: Non-dynamic constraints are discretized at all nodes according to their time frame. For example, path constraints must apply at every node and final constraints at the very last node.
- Objective function: The Lagrange term, which is an integral over the entire time horizon, is approximated using the quadrature rule, which corresponds to the RadauIIA collocation scheme. The Mayer term, which represents a cost function evaluated at the last time point, is replaced by an evaluation at the last node.
- Transformation to a nonlinear problem: The local collocation-based discretization leads to a large-scale nonlinear problem. This NLP has very sparse Jacobians and Hessians, which makes it well suited for modern optimization algorithms.
- Back-end solution: The C++ back-end of the framework solves this NLP with IPOPT. The Python front-end allows access to this back-end so that the user can interact with the solver via a series of flags. For example, if the number of mesh refinement iterations is set greater than 0, the optimization and refinement are looped, refining and optimizing the mesh sequentially. This provides a very precise solution to the optimization problem.

This overall approach combines the flexibility of Python with the computational efficiency of C++, ensuring that dynamic optimization problems can be solved efficiently.

2 Installation

Depending on your operating system, follow the installation instructions.

2.1 Ubuntu / Debian

Make sure that the dependencies (git, curl, pkgconf, gfortran, LAPACK, CMake, g++, Python3, Python3-Venv, Python3-Build) are installed on the system:

```
sudo apt-get install git curl pkg-config gfortran liblapack-dev cmake
    g++ python3 python3-venv
pip install build
```

Clone the GDOPT repository from GitHub:

```
git clone https://github.com/linuslangenkamp/GDOPT.git
```

To keep the environment isolated, it is advised to create and activate a virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
```

With the virtual environment activated, install the package. This will automatically install IPOPT[1], the linear solver MUMPS[2], and the Python packages used in the frontend, i.e. SymEngine[4], NumPy[5], SciPy[6], pandas[7], and Matplotlib[8]:

```
pip install .
```

Now it is possible to use GDOPT in your Python scripts:

```
import gdopt
```

To test whether the framework is set up correctly, execute the `Hello World` model:

```
gdopt.HelloWorld()
```

In case the plotting features are not working correctly, it is advised to install Qt5:

```
pip install PyQt5
```

2.2 Fedora

Make sure that the dependencies (git, curl, pkgconf, gfortran, LAPACK, CMake, g++, patch, Python3, Python3-Venv, Python3-Pip, Python3-Build) are installed on the system:

```
sudo dnf install git curl pkgconf gfortran lapack-devel cmake gcc-c++
    python3 python3-pip python3-virtualenv patch
pip install build
```

Clone the GDOPT repository from GitHub:

```
git clone https://github.com/linuslangenkamp/GDOPT.git
```

To keep the environment isolated, it is advised to create and activate a virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
```

With the virtual environment activated, install the package. This will automatically install IPOPT[1], the linear solver MUMPS[2], and the Python packages used in the frontend, i.e. SymEngine[4], NumPy[5], SciPy[6], pandas[7], and Matplotlib[8]:

```
pip install .
```

Now it is possible to use GDOPT in your Python scripts:

```
import gdopt
```

To test whether the framework is set up correctly, execute the `Hello World` model:

```
gdopt.HelloWorld()
```

In case the plotting features are not working correctly, it is advised to install Qt5:

```
pip install PyQt5
sudo dnf install qt5-qtbase-gui
```

2.3 HSL Linear Solvers

In addition to MUMPS, linear solvers from the HSL suite such as MA27, MA57, MA77, MA86, MA97[3] may also be used if a license is present. To do this, the environment variable "LIB_HSL" must be set to the corresponding path. The framework can then load the solvers at runtime.

3 Getting Started

This section shows how to set up and solve a simple dynamic optimization problem with the proposed framework.

3.1 Basic Example

Here is a minimal working example for a bang-bang control problem, adapted from the OpenModelica User's Guide [9]

```
from gdopt import *

model = Model("bangBang")

# states x1(t), x2(t) with x1(0) = x2(0) = 0
x1 = model.addState(start=0, symbol="obj")
x2 = model.addState(start=0, symbol="obj'")

# control: u(t) with |u(t)| <= 10
u = model.addControl(lb=-10, ub=10, symbol="control")

# dynamic: x1''(t) = u(t)
model.addDynamic(x1, x2) # x1'(t) = x2(t)
model.addDynamic(x2, u) # x2'(t) = u(t)

# algebraic constraints
model.addPath(x2 * u, lb=-30, ub=30) # -30 <= x2(t) * u(t) <= 30
model.addFinal(x2, eq=0) # x2(tf) = 0

# objective = x1(tf) -> max
model.addMayer(x1, Objective.MAXIMIZE)

# generate the C++ code
model.generate()

# tf=0.5s, 150 intervals, using RadauIIA 3 step order 5 scheme
model.optimize(tf=0.5, steps=150, rksteps=3)

model.plot()
```

3.2 Explanation

Example procedure:

- Import the package and create a `Model` object with a given name
- Add the states with given starting values (and optionally assign a symbol)

- Add the control with lower and upper bounds (and optionally assign a symbol)
- Define the dynamic, path, final constraints
- Add an objective
- Generate the C++ code
- Optimize for a given final time, number of steps and collocation nodes
- Plot the model

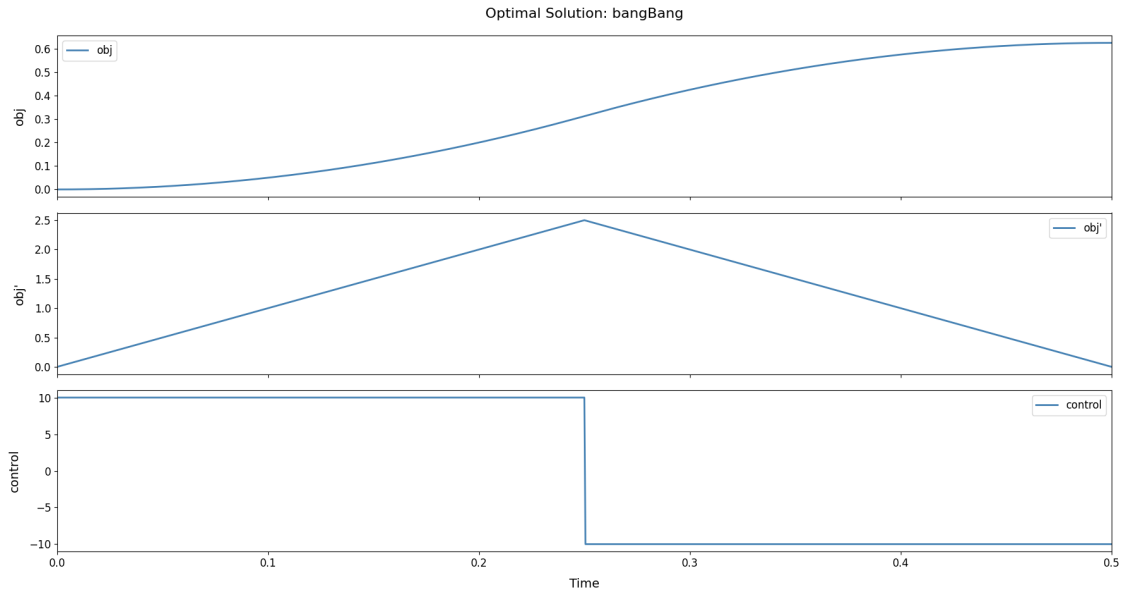


Figure 1: Optimal bang-bang solution provided by the framework

4 Mathematical Problem Formulation

In a **General Dynamic Optimization Problem** (GDOP) the goal is the minimization of a *Mayer term*, which defines the cost at the final time t_f (terminal cost), and a *Lagrange term*, which represents the running cost over the time horizon $[t_0, t_f]$. The problem can be written as follows:

$$\min_{\boldsymbol{u}(t),\boldsymbol{p}} \quad \underbrace{M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)}_{\text{Mayer term}} + \underbrace{\int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\, \mathrm{d}t}_{\text{Lagrange term}}$$

subject to the following constraints:

- *Dynamic constraints*: The states of the system are evaluated by the differential equation $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)$. The differential equation must apply for all times $t \in [t_0, t_f]$ and has the initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$.
- *Path constraints*: For all times $t \in [t_0, t_f]$, the variables must satisfy the inequality $\boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq \boldsymbol{g}^U$. These may represent physical restrictions, for example.
- *Final constraints*: At the final time t_f, $\boldsymbol{r}^L \leq \boldsymbol{r}(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) \leq \boldsymbol{r}^U$ must be fulfilled. These constraints define the desired final configuration of the system.
- *Parametric constraints:* The static parameters must satisfy the algebraic inequality constraints $\boldsymbol{a}^L \leq \boldsymbol{a}(\boldsymbol{p}) \leq \boldsymbol{a}^U$.

Variables and Functions:

- t: Time in the model.
- t_f: Final time of the time horizon.
- $\boldsymbol{x}(t)$: States, which evolve according to the dynamics $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\cdot)$.
- $\boldsymbol{u}(t)$: Time-dependent control variables, also referred to as inputs.
- $\boldsymbol{p}$: Static parameters.
- $M(\cdot)$: Mayer term, a function that evaluates a cost at the final time.
- $L(\cdot)$: Lagrange integrand, a function defining a cost at a specific time t. $\int_{t_0}^{t_f} L(\cdot)\, \mathrm{d}t$ is then referred to as the Lagrange term.
- $\boldsymbol{f}(\cdot)$: Function defining the system dynamics.
- $\boldsymbol{g}(\cdot)$: Path constraint function, which forces restrictions on variables at all times.
- $\boldsymbol{r}(\cdot)$: Final constraint function, which forces restrictions on variables at the final time.
- $\boldsymbol{a}(\cdot)$: Parametric constraint function, which forces restrictions on solely static parameters.

In this problem formulation, the goal is to minimize the combined objective of the Mayer and Lagrange terms while ensuring that the system dynamics, path constraints, final constraints, and parametric constraints are all satisfied.

The entire problem can be formulated in standard notation as

$$
\begin{aligned}
\min_{\boldsymbol{u}(t),\boldsymbol{p}} \; & M(\boldsymbol{x}(t_f),\boldsymbol{u}(t_f),\boldsymbol{p},t_f) + \int_{t_0}^{t_f} L(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t)\,\mathrm{d}t \\
& \text{s.t.} \\
& \dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t)\ \forall t \in [t_0,t_f] \\
& \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\
& \boldsymbol{g}^L \le \boldsymbol{g}(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t) \le \boldsymbol{g}^U\ \forall t \in [t_0,t_f] \\
& \boldsymbol{r}^L \le \boldsymbol{r}(\boldsymbol{x}(t_f),\boldsymbol{u}(t_f),\boldsymbol{p},t_f) \le \boldsymbol{r}^U \\
& \boldsymbol{a}^L \le \boldsymbol{a}(\boldsymbol{p}) \le \boldsymbol{a}^U
\end{aligned}
$$

where $L, M, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{r}, \boldsymbol{a} \in C^2$, i.e. are continuously differentiable twice.

5 Modeling

5.1 Model Creation

The first step is to create a new Python file and to import the gdopt package.

```
from gdopt import *
```

Additionally a `Model` object has to be created, which is the basis for all modeling.

Constructs a `Model` object.

```
model = Model(name="DummyName")
```

Parameters:

- `str name` *(optional)*: The name of the model.

Returns: `Model model`: The `Model` object.

Example:

```
model = Model("Model Name")
```

The variables, constraints and objectives are added to this `Model` object. Note that a `Model` can be printed to the standard output with the native `print` function.

5.2 Variables

5.2.1 States

States are time-dependent functions, that are calculated in the solving process. The value of the state is completely determined by a starting value and an ordinary differential equation.

Adding a state to a `Model` object.

```
x = model.addState(start, symbol=None, lb=-float("inf"),
    ub=float("inf"), nominal=None)
```

Parameters:

- `float start`: The initial value of the state variable.
- `str symbol` *(optional)*: A symbolic representation of the state. The state is represented by this symbol in further analysis.

- `float lb` *(optional)*: Lower bound for the state variable.
- `float ub` *(optional)*: Upper bound for the state variable.
- `float nominal` *(optional)*: A nominal value for scaling.

Returns: `Symbol variable`: The symbolic representation of the state variable, which can be used in constraints.

Aliases: `model.addX`

5.2.2 Inputs

Input or controls are time-dependent functions, that are calculated in the solving process and are not differentiated in the model.

Adding an input / control variable $u(t)$, that changes over time, to a `Model` object.

```
u = model.addInput(symbol=None, lb=-float("inf"), ub=float("inf"),
    guess=0, nominal=None)
```

Parameters:

- `str symbol` *(optional)*: A symbolic representation of the input. The input is represented by this symbol in further analysis.
- `float lb` *(optional)*: Lower bound for the input variable.
- `float ub` *(optional)*: Upper bound for the input variable.
- `Expression guess` *(optional)*: An initial guess for the input. Further explanation in (5.2.2.1).
- `float nominal` *(optional)*: A nominal value for scaling.

Returns: `Symbol variable`: The symbolic representation of the input, which can be used in constraints.

Aliases: `model.addU`, `model.addControl`, `model.addContinuous`

5.2.2.1 Input Guesses The guess parameter in `model.addInput` (5.2.2) can be any `Expression` that contains the global time or final time symbols (7.2), but can also be a regular constant `float` or `int`. If the `"initVars"` flag (5.6.1) is chosen to be a `SOLVE` option (7.6.3), this trajectory is used to solve the dynamic and provide an initial, feasible guess for the state variables. If the option `InitVars.CONST` is set, the guess is used nevertheless, although the states are initialized constant. Note that any valid SymEngine `Expression` can be provided as a guess, although there are several standard functions to provide better initial guesses for the control.

Providing a constant control guess trajectory.

```
guessConstant(const)
```

Parameters:

- `float const`: Constant value for the control guess, i.e. $u(t) \equiv const \; \forall t \in [0, t_f]$.

Remarks: It's also possible to just write `guess=const` in the `model.addInput` arguments.

Providing a linear control guess trajectory. The provided values are interpolated accordingly.

```
guessLinear(u0, uf)
```

Parameters:

- `float u0`: Value for the control guess at $t = 0$, i.e. $u(0)$.
- `float uf`: Value for the control guess at $t = t_f$, i.e. $u(t_f)$.

Providing a quadratic control guess trajectory. The provided values are interpolated accordingly.

```
guessQuadratic(u0, um, uf)
```

Parameters:

- `float u0`: Value for the control guess at $t = 0$, i.e. $u(0)$.
- `float um`: Value for the control guess at $t = \frac{t_f}{2}$, i.e. $u(\frac{t_f}{2})$.
- `float uf`: Value for the control guess at $t = t_f$, i.e. $u(t_f)$.

Providing an exponential control guess trajectory. The provided values are interpolated accordingly.

```
guessExponential(u0, uf)
```

Parameters:

- `float u0`: Value for the control guess at $t = 0$, i.e. $u(0)$.
- `float uf`: Value for the control guess at $t = t_f$, i.e. $u(t_f)$.

Providing a piecewise defined control guess trajectory.

```
guessPiecewise(*args)
```

Parameters:

- `*(Expression, Condition) *args`: Arbitrary number of tuples. The first argument in each tuple has to be an `Expression` and the second argument is the corresponding interval its defined on.

Example: `guessPiecewise((0.6, t <= 0.5), (2 + t**2, 0.5 < t))` represents the initial control guess $u(t) = \begin{cases} 0.6, & t \leq \frac{1}{2} \\ 2 + t^2, & t > \frac{1}{2} \end{cases}$. Note that these expressions can be standard expressions such as `guessQuadratic`, `guessLinear` or any arbitrary SymEngine `Expression`.

5.2.3 Parameters

Parameters are time-invariant constants, that are calculated in the solving process.

Adding a parameter variable, that is time-invariant constant, to a `Model` object.

```
p = model.addParameter(symbol=None, lb=-float("inf"),
    ub=float("inf"), guess=0, nominal=None)
```

Parameters:

- `str symbol` *(optional)*: A symbolic representation of the parameter. The parameter is represented by this symbol in further analysis.
- `float lb` *(optional)*: Lower bound for the parameter variable.
- `float ub` *(optional)*: Upper bound for the parameter variable.
- `Expression guess` *(optional)*: An initial guess for the parameter.
- `float nominal` *(optional)*: A nominal value for scaling.

Returns: `Symbol variable`: The symbolic representation of the parameter, which can be used in constraints.

Aliases: `model.addP`

5.2.4 Runtime Parameters

Runtime parameters are time-invariant global constants, that can be changed after code generation and compilation via `model.setValue`. The value of the runtime parameter will be set at runtime. These represent usual parameters of standard modeling environments. A runtime parameter can be used literally anywhere in the `Model` object, e.g. in objectives or constraints, as a starting value, a lower or upper bound and even as a nominal value.

Adding a runtime parameter to the model. This is a constant in the model, that will be substituted by its associated value at runtime time.

```
rp = model.addRuntimeParameter(default, symbol=None)
```

Parameters:

- `float default`: A default value for the runtime parameter.
- `str symbol` *(optional)*: A symbolic representation of the runtime parameter.

Returns: `Symbol variable`: The symbolic representation of the runtime parameter.

Aliases: `model.addRP`

Changing the associated value of a runtime parameter.

```
model.setValue(runtimeParameter, value)
```

Parameters:

- `Symbol runtimeParameter`: A symbolic representation of the runtime parameter.
- `float value`: The new associated value for the runtime parameter.

5.3 Constraints

Any constraint can contain runtime constants such as the global final time (7.2) or runtime parameters (5.2.4).

5.3.1 Dynamic Constraints

Dynamic constraints are explicit ordinary differential equations with the derivative of a state on the left hand side and the corresponding rate of change on the right hand side. This equation must hold at every moment.

Adding a dynamic constraint $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\ \forall t \in [t_0, t_f]$ to the `Model` object, where $y(t)$ is a previously added state.

```
model.addDynamic(diffVar, expr, nominal=None)
```

Parameters:

- `Symbol diffVar`: The state variable that gets differentiated, i.e. $y(t)$.
- `Expression expr`: The right hand side of the ordinary differential equation, i.e. $f(\cdot)$.
- `float nominal` *(optional)*: A nominal value of $f(\cdot)$ for scaling.

Aliases: `model.addF`, `model.addOde`

Example: `model.addDynamic(x1, x1 * u1 + p + t)` represents the differential equation $\frac{\mathrm{d}x_1(t)}{\mathrm{d}t} = x_1(t)u_1(t) + p + t$. (Using the global time symbol (7.2))

5.3.2 Path Constraints

Path constraints are algebraic constraints on states, inputs, parameters and time, that must hold at every moment.

Adding a path constraint $g^L \leq g(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq g^U \; \forall t \in [t_0, t_f]$ to the `Model` object.

```
model.addPath(expr, lb=-float("inf"), ub=float("inf"), eq=None,
    nominal=None)
```

Parameters:

- `Expression expr`: The algebraic expression, i.e. $g(\cdot)$.
- `float lb` *(optional)*: Lower bound for the path constraint.
- `float ub` *(optional)*: Upper bound for the path constraint.
- `float eq` *(optional)*: Equality parameter for the path constraint. Can not be chosen simultaneously with `lb` nor `ub`.
- `float nominal` *(optional)*: A nominal value of $g(\cdot)$ for scaling.

Aliases: `model.addG`

Example: `model.addPath(x1**2 + u1**2, ub=5)` represents the path constraint $x_1^2 + u_1^2 \leq 5$.

5.3.3 Final Constraints

Final constraints are algebraic constraints on states, control and parameters, that must hold at the final time t_f.

Adding a final constraint $r^L \leq r(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) \leq r^U$ to the `Model` object.

```
model.addFinal(expr, lb=-float("inf"), ub=float("inf"), eq=None,
    nominal=None)
```

Parameters:

- `Expression expr`: The algebraic expression, i.e. $r(\cdot)$.
- `float lb` *(optional)*: Lower bound for the final constraint.
- `float ub` *(optional)*: Upper bound for the final constraint.
- `float eq` *(optional)*: Equality parameter for the final constraint. Can not be chosen simultaneously with `lb` nor `ub`.

- `float nominal` *(optional)*: A nominal value of $r(\cdot)$ for scaling.

Aliases: `model.addR`

Example: `model.addFinal(x1 + u1 - p, eq=0)` represents the final constraint $x_1(t_f) + u_1(t_f) - p = 0$.

5.3.4 Parametric Constraints

Parametric constraints are algebraic constraints that can contain only parameters.

Adding a parametric constraint $a^L \leq a(\boldsymbol{p}) \leq a^U$ to the `Model` object.

```
model.addParametric(expr, lb=-float("inf"), ub=float("inf"), eq=None,
    nominal=None)
```

Parameters:

- `Expression expr`: The algebraic expression, i.e. $a(\cdot)$.
- `float lb` *(optional)*: Lower bound for the parametric constraint.
- `float ub` *(optional)*: Upper bound for the parametric constraint.
- `float eq` *(optional)*: Equality parameter for the parametric constraint. Can not be chosen simultaneously with `lb` nor `ub`.
- `float nominal` *(optional)*: A nominal value of $a(\cdot)$ for scaling.

Aliases: `model.addA`

Example: `model.addParametric(sin(p1) - p2, lb=0)` represents the parametric constraint $0 \leq \sin(p_1) - p_2$.

5.4 Objective

Any objective can contain runtime constants such as the global final time (7.2) or runtime parameters (5.2.4).

5.4.1 Mayer Term

The Mayer term penalizes the final configuration of the system.

Adding a Mayer term $M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)$ to the `Model` object.

```
model.addMayer(expr, obj=Objective.MINIMIZE, nominal=None)
```

Parameters:

- `Expression expr`: The Mayer term, i.e. $M(\cdot)$.
- `Objective obj` *(optional)*: The objective goal for the Mayer term, corresponding to an element of the objective enumeration (7.6.1).
- `float nominal` *(optional)*: A nominal value of $M(\cdot)$ for scaling.

Aliases: `model.addM`

Example: `model.addMayer(x1 + x2**2, obj=Objective.MINIMIZE)` represents the Mayer term $x_1 + x_2^2$.

5.4.2 Lagrange Term

The Lagrange term defines a cumulative cost of the system over the entire time horizon.

Adding a Lagrange term $\int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\,\mathrm{d}t$ to the `Model` object.

```
model.addLagrange(expr, obj=Objective.MINIMIZE, nominal=None)
```

Parameters:

- `Expression expr`: The Lagrange integrand, i.e. $L(\cdot)$.
- `Objective obj` *(optional)*: The objective goal for the Lagrange term, corresponding to an element of the objective enumeration (7.6.1).
- `float nominal` *(optional)*: A nominal value of $L(\cdot)$ for scaling.

Aliases: `model.addL`

Example: `model.addLagrange(x1 * u1 + t, obj=Objective.MINIMIZE)` represents the Lagrange term $\int_{t_0}^{t_f} x_1(t)u_1(t) + t\,\mathrm{d}t$.

5.4.3 Combined Objectives

If both the Mayer and Lagrange term are contained in a model, there are specific factors that must be taken into account. The objective goal `obj` is multiplying the associated term with a factor of -1, if maximization is chosen. If these goals differ between both terms, e.g. $\max M(\cdot)$ and $\min \int_{t_0}^{t_f} L(\cdot)\,\mathrm{d}t$, then the full objective is given by $\min -M(\cdot) + \int_{t_0}^{t_f} L(\cdot)\,\mathrm{d}t$ and vice versa. Additionally nominal values are added if both terms have assigned nominals. If only one term has a nominal value, then this value is used for the full objective.

Combined objectives can be added individually with `model.addMayer` (5.4.1) and `model.addLagrange` (5.4.2) or with the method `model.addObjective`.

Adding a Mayer $M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)$ and Lagrange term $\int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\,\mathrm{d}t$ to the `Model` object.

```
model.addObjective(mayer, lagrange, obj=Objective.MINIMIZE,
    nominal=None)
```

Parameters:

- `Expression mayer`: The Mayer term, i.e. $M(\cdot)$.
- `Expression lagrange`: The Lagrange integrand, i.e. $L(\cdot)$.
- `Objective obj` *(optional)*: The objective goal for the combined cost of $M(\cdot)$ + $\int_{t_0}^{t_f} L(\cdot)\,\mathrm{d}t$, corresponding to an element of the objective enumeration (7.6.1).
- `float nominal` *(optional)*: A nominal value of the full objective for scaling.

Example: `model.addObjective(x1, x2 * u1, obj=Objective.MINIMIZE)` represents the full objective $x_1(t_f)$ + $\int_{t_0}^{t_f} x_2(t)u_1(t)\,\mathrm{d}t$.

5.5 Code Generation

If the modeling process is done, it is possible to generate and compile the C++ code of the model. Therefore, the 1st and 2nd derivatives of the objectives and all constraints are calculated symbolically with SymEngine. After that, the framework will search for common subexpressions (CSE) in each derivative, using the symbolic capabilities of SymEngine. The resulting expressions as well as all other problem defining structures are generated to a C++ file. This file makes use of runtime constants / model parameters (e.g. final time, runtime parameters, flags, paths, …), that will be set later in the `model.optimize` pipeline (5.6). The resulting C++ file is compiled, while linking against the dynamic optimization back-end.

If several optimizations have to be executed, where only flags and runtime parameters are changed, it is advised to call `model.generate` only once, since all flags will be set in a later stage. Thus, obsolete recalculations can be avoided.

Calculates the 1st and 2nd derivatives, generates all the problem defining structures to a .cpp file, and compiles the code.

```
model.generate()
```

5.6 Optimization

After code generation and compilation, all model parameters (e.g. final time, runtime parameters, flags, paths, …) have to be written to a configuration file. The already existing executable will read in the configuration and then solve the NLP. Additionally the optimal

solution as well as some metadata can be written to text files. These files can be analyzed with the presented framework (6).

Sets all model parameters (e.g. final time, runtime parameters, flags, paths, ...) in the configuration file and runs the optimization.

```
model.optimize(tf=1, steps=1, rksteps=1, flags={}, meshFlags={},
    resimulate=False, recompile=False)
```

Parameters:

- `float tf` *(optional)*: The final time t_f of the model.
- `int steps` *(optional)*: The number of intervals.
- `int rksteps` *(optional)*: The number of collocation nodes of the RadauIIA integrator per interval. Valid range: $1 \leq$ `rksteps` ≤ 36.
- `dict<str, arg> flags` *(optional)*: A dictionary of flags. All possible flags are presented in (5.6.1).
- `dict<str, arg> meshFlags` *(optional)*: A dictionary of mesh flags. All possible mesh flags are presented in (5.6.2).
- `bool resimulate` *(optional)*: If set to `true`, all arguments and flags of this method will be taken from the previous optimization.

Example:

```
model.optimize(
        tf=1,
        steps=250,
        rksteps=3,
        flags={
                "linearSolver": LinearSolver.MUMPS,
                "initVars": InitVars.SOLVE_EXPLICIT,
                "exportJacobianPath": "/tmp",
        },
        meshFlags={
                "algorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "iterations": 6
        }
)
```

If `optimize` has to be called only once, it is also possible to use `model.solve`. This is a basic wrapper, that calls `generate` and `optimize` sequentially and shares the parameters with `optimize`.

Generates all problem defining structures to a .cpp file, compiles, sets all model parameters (e.g. final time, runtime parameters, flags, paths, ...), and runs the optimization.

```
model.solve(tf=1, steps=1, rksteps=1, flags={}, meshFlags={},
    resimulate=False)
```

Parameters:

- `float tf` *(optional)*: The final time t_f of the model.
- `int steps` *(optional)*: The number of intervals.
- `int rksteps` *(optional)*: The number of collocation nodes of the RadauIIA integrator per interval. Valid range: $1 \leq$ `rksteps` ≤ 36.
- `dict<str, arg> flags` *(optional)*: A dictionary of flags. All possible flags are presented in (5.6.1).
- `dict<str, arg> meshFlags` *(optional)*: A dictionary of mesh flags. All possible mesh flags are presented in (5.6.2).
- `bool resimulate` *(optional)*: If set to `true`, all arguments and flags of this method will be taken from the previous optimization.

If the `Model` has to be optimized several times, e.g. for benchmarks, one can call `execute`. This method executes the previously compiled model using the existing initial states and configuration file. Obviously `solve` or `optimize` has to be called previously. Additionally command-line arguments can be provided to the back-end.

Runs the optimization given a previously compiled model, calculated initial states, if `"initVars"` is set to `InitVars.SOLVE` (5.6.1), and an existing configuration. Command-line arguments can be provided to `execute` via the `args` argument (currently in development).

```
model.execute(args="")
```

Parameters:

- `string args` *(optional)*: Command line arguments for the back-end.

5.6.1 Flags

The entire flag dictionary and therefore all flags are optional and do not have to provided by the user. Nevertheless, some flags set values that are mandatory to the solver. The corresponding attributes have internal default values, which are presented below.

- `"tolerance"` → `float` (*default*=`1e-14`): Sets the optimization tolerance in IPOPT. The acceptable tolerance is always defined as `1e3` · "tolerance".
- `"linearSolver"` → `LinearSolver` (*default*=`LinearSolver.MUMPS`): Sets the linear solver used in IPOPT. The linear solver has to be an element of the `LinearSolver` enumeration (7.6.2).

- `"initVars"` → `InitVars` (*default*=`InitVars.SOLVE`): Sets the way in which the variables are initialized. The value has to be an element of the `InitVars` enumeration (7.6.3).
- `"ivpSolver"` → `IVPSolver` (*default*=`IVPSolver.RADAU`): Sets the `SciPy` integrator, that solves the arising initial value problem, if `"initVars"` is set to `InitVars.SOLVE`. The value has to be an element of the `IVPSolver` enumeration (7.6.4).
- `"maxIterations"` → `int` (*default*=`5000`): Sets the maximum number of iterations in IPOPT.
- `"kktMuGlobalization"` → `bool` (*default*=`True`): Turns the IPOPT adaptive mu globalization based on KKT errors on or off. In many cases, it is advantageous to use this, which is why this strategy is set as the default. For poorly conditioned problems, it is advised to disable this.
- `"outputPath"` → `str` (*default* = `"cwd\.generated\modelname\modelname {it}.csv"`): Sets the path to which the optimal solution is output as a text file. If this value is not provided, the optimal solution for every mesh iteration $it = 0, \ldots, maxMeshIteration$ will be written to the .generated directory.
- `"disableOutput"` → `bool` (*default* = `False`): If set to true, the output of the optimal solution will be disabled. This may be useful when benchmarks are carried out. Note that the analysis (6) can not be accessed, if `"disableOutput"` is set to true.
- `"ipoptPrintLevel"` → `int` (*default*=5): Sets the verbosity level for console outputs of IPOPT. The default value of 5 provides pretty detailed information about the solving process. The valid range is $0 \leq$ `"ipoptPrintLevel"` ≤ 12, while a value of 0 corresponds to no output at all.
- `"exportHessianPath"` → `str`: Sets the path to which the Hessian sparsity pattern is output as a text file. If this value is not provided, the sparsity can not be accessed.
- `"exportJacobianPath"` → `str`: Sets the path to which the Jacobian sparsity pattern is output as a text file. If this value is not provided, the sparsity can not be accessed.
- `"initialStatesPath"` → `str` (*default*=`"/.generated"`): Sets the path to which the solution of the initial value problem, which arises if `"initVars"` is set to `InitVars.SOLVE`, is written.
- `"linearObjective"` → `bool` (*default*=`False`): Tells the solver that the Mayer term and Lagrange integrand are at most linear. This can improve the runtime, but be aware that this flag should only be set, if the criterion is met.
- `"quadraticObjective"` → `bool` (*default*=`False`): Tells the solver that the Mayer term and Lagrange integrand are at most quadratic. This can improve the runtime, but be aware that this flag should only be set, if the criterion is met.
- `"linearConstraints"` → `bool` (*default*=`False`): Tells the solver that all constraints, i.e. $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{r}, \boldsymbol{a}$, are at most linear. This can improve the runtime, but be aware that this flag should only be set, if the criterion is met.

5.6.2 Mesh Flags

- `"algorithm"` → `MeshAlgorithm` (*default*=`MeshAlgorithm.L2_BOUNDARY_NORM`): Sets the mesh algorithm, which is used to update the mesh and consequently create high resolution solutions at important sections. The mesh algorithm has to be an element of the `MeshAlgorithm` enumeration (7.6.5).
- `"iterations"` → `int` (*default*=`0`): Sets the number of mesh refinement iterations. The default value is 0, implying that no mesh refinement is executed. A typical range for `"iterations"` is between 3 and 10.
- `"refinementMethod"` → `RefinementMethod` (*default*=`RefinementMethod.LINEAR_SPLINE`): Sets the refinement method, which is used to interpolate the optimal solution from the previous optimization and provide new initial values. The refinement method has to be an element of the `RefinementMethod` enumeration (7.6.6).
- `"level"` → `float` (*default*=0): Sets the level of `MeshAlgorithm.L2_BOUNDARY_NORM`. This is only used, if `"algorithm"` is set accordingly. The `"level"` describes how sensitive the algorithm is to abnormal behavior on an interval. Positive values make the algorithm more sensitive, and negative values less so. The `"level"` operates on a logarithmic scale with base 10, meaning that an increase by 1 corresponds to a 10-fold increase in sensitivity. The typical range for `"level"` is between -2.5 and 2.5.
- `"cTol"` → `float` > 0 (*default*=0.1): Sets the P1-error threshold / corner tolerance in `MeshAlgorithm.L2_BOUNDARY_NORM`. This is only used, if `"algorithm"` is set accordingly. This argument specifies a threshold for the error in derivatives of adjacent intervals at the boundary. Large values make the algorithm less sensitive and values close to 0 make it more sensitive. The typical range for `"cTol"` is between 0.025 and 0.5.
- `"sigma"` → `float` > 0 (*default*=2.5): Sets the sigma in `MeshAlgorithm.BASIC_STRATEGY`. Indicates how many standard deviations away from the mean value is considered abnormal behavior. This is only used, if `"sigma"` is set accordingly. The typical range for `"sigma"` is between 1.5 and 5.

6 Analysis

The gdopt package offers the possibility to directly analyze the optimal solution with pandas and Matplotlib functions. To access the features described in this section, it is mandatory that "disableOutput" (5.6.1) is not set to True. Otherwise the analysis can not be accessed.

6.1 Results

In any case the framework generates a metadata file "\tmp\modelinfo.txt", which includes basic information about the solving process. This file is automatically read in as a dict after a successful optimization and includes the time taken by the solver as well as the number of mesh iterations. The modelinfo can be accessed via model.modelInfo, e.g.

```
print(model.modelInfo)
out: {"maxMeshIteration": 6, "totalTimeInSolver": 0.6753,
    "actualTimeInSolver": 0.600671, "totalTimeInIO": 0.0746288}
```

The file's main purpose is to do benchmarks, although it is used internally as well. The elements of the dictionary are

- "maxMeshIteration": Number of mesh iterations that have been executed
- "totalTimeInSolver": Total time taken in the back-end
- "totalTimeInIO": Total time to read and write files in the back-end
- "actualTimeInSolver": Time taken in back-end excluding read and write operations

If the "disableOutput" flag (5.6.1) is not set to True, the framework can directly import the optimal solution from the back-end as a pandas.DataFrame. Note that the names of the variables are taken from the symbol argument in addState (5.2.1), addInput (5.2.2) and addParameter (5.2.3). If no symbol is provided, a dummy name will be used.

Reads in the result file from a specific mesh iteration and saves it to the dictionary model.resultHistory at the key meshIteration. The value model.resultHistory [meshIteration], that is set in the process, is returned. If no mesh iteration is provided, the last mesh iteration is used automatically.

```
model.getResults(meshIteration=None)
```

Parameters:

- int meshIteration *(optional)*: The mesh iteration from which the results are to be analyzed.

Returns: pandas.DataFrame model.resultHistory[meshIteration]

It is also possible to read in the results from all mesh iterations with getAllResults.

Reads in the result file from all mesh iterations and saves them to the dictionary `model.resultHistory` at the keys `0, . . . , maxMeshIteration`. The entire dictionary `model.resultHistory` is returned in the process.

```
model.getAllResults()
```

Returns: `dict<int, pandas.DataFrame> model.resultHistory`

Example:

```
# read in all result files and set the resultHistory
model.getAllResults()
# e.g. get the results from mesh iteration 2
results = model.resultHistory[2]
```

The results can be printed with the two following method.

Prints the results from a given mesh iteration to the console. If no mesh iteration is provided, the last mesh iteration is used automatically.

```
model.printResults(meshIteration=None)
```

Parameters:

- `int meshIteration` *(optional)*: The mesh iteration from which the results have to be printed.

```
# example soultion from the Batch Reactor
# printing the last mesh iteration
model.printResults()
out:
          time   Reactant    Product         u
0   0.00000000 1.00000000 0.00000000 0.74354337
1   0.00620204 0.99367796 0.00460616 0.74654578
2   0.02579796 0.97375329 0.01908967 0.75623418
3   0.04000000 0.95936074 0.02951972 0.76344758
4   0.04620204 0.95308805 0.03405685 0.76665199
...
```

Prints the optimal parameters from a given mesh iteration to the console. If no mesh iteration is provided, the last mesh iteration is used automatically.

```
model.printResultParameters(meshIteration=None)
```

Parameters:

- `int meshIteration` *(optional)*: The mesh iteration from which the parameters have to be printed.

6.2 Plotting

The framework includes a lot of functions to draw standard plots of states and inputs, sparsity patterns, parametric plots and of the mesh refinement process as well.

Draws a plot of the optimal solution.

```
model.plot(specifCols=None, meshIteration=None, interval=None,
    dots=Dots.OFF)
```

Parameters:

- `list<str> specifCols` *(optional)*: Sets the list of variables that will be drawn. These strings have to match the columns names in `model.resultHistory` or equivalently the `symbol` argument of the variables. If no value is provided, all variables will be drawn in the plot.
- `int meshIteration` *(optional)*: Sets the mesh iteration of which the optimal values originate. If no value is provided, the last iteration is used.
- `[float, float] interval` *(optional)*: Sets the time range of the plot. Defaults to the interval $[t_0, t_f]$.
- `Dots dots` *(optional)*: Adds dots of the discrete solution to the plot. This has to be an element of the `Dots` enumeration (7.6.8) and in the default case no dots are added.

Special cases: There are two methods that use `model.plot` directly. These share all arguments with the standard plot function but only plot the state variables or the input variables respectively.

```
model.plotStates(meshIteration=None, interval=None, dots=Dots.OFF)
```

```
model.plotInputs(meshIteration=None, interval=None, dots=Dots.OFF)
```

Example:

```
model.plot(specifCols=["Product", "u"], interval=[0.5, 1],
    dots=Dots.ALL)
```

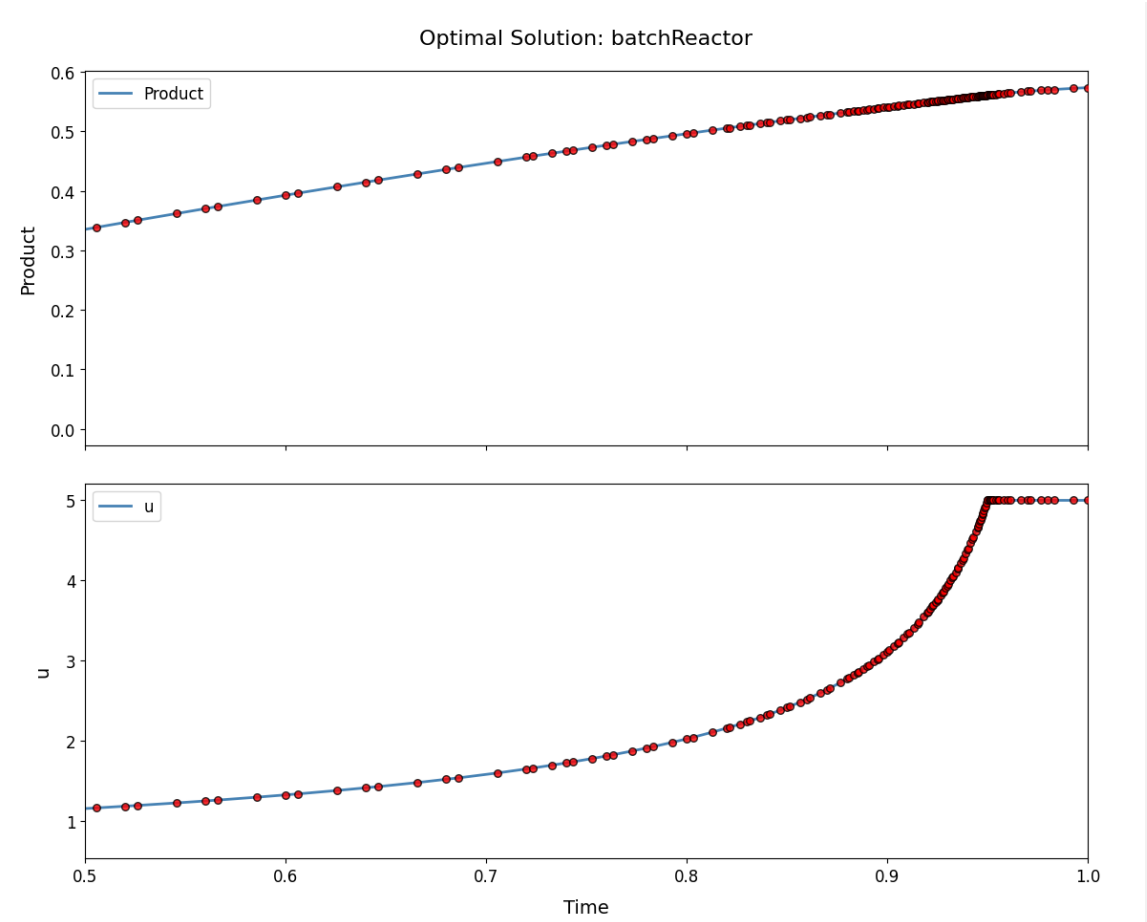


Figure 2: Optimal product and control the "Batch Reactor" example

Draws a parametric plot of the optimal solution.

```
model.parametricPlot(varX, varY, meshIteration=None, interval=None,
    dots=Dots.OFF)
```

Parameters:

- `Symbol varX`: Symbol of the variable on the x-axis.
- `Symbol varY`: Symbol of the variable on the y-axis.
- `int meshIteration` *(optional)*: Sets the mesh iteration of which the optimal values originate. If no value is provided, the last iteration is used.
- `[float, float] interval` *(optional)*: Sets the time range of the plot. Defaults to the interval $[t_0, t_f]$.
- `Dots dots` *(optional)*: Adds dots of the discrete solution to the plot. This has to be an element of the `Dots` enumeration (7.6.8) and in the default case no dots are added.

Example:

```
model.parametricPlot(xpos, ypos)
```

Draws a plot of the mesh refinement process.

```
model.plotMeshRefinement(interval=None, markerSize=30, dots=Dots.BASE)
```

Parameters:

- `[float, float] interval` *(optional)*: Sets the time range of the plot. Defaults to the interval $[t_0, t_f]$.
- `float markerSize` *(optional)*: Sets the size of the dots in the plot.
- `Dots dots` *(optional)*: Sets which points are displayed in the plot. This has to be an element of the `Dots` enumeration (7.6.8) (excluding `Dots.NONE`) and in the default case base points are added as a dot.

Example:

```
model.plotMeshRefinement()
```

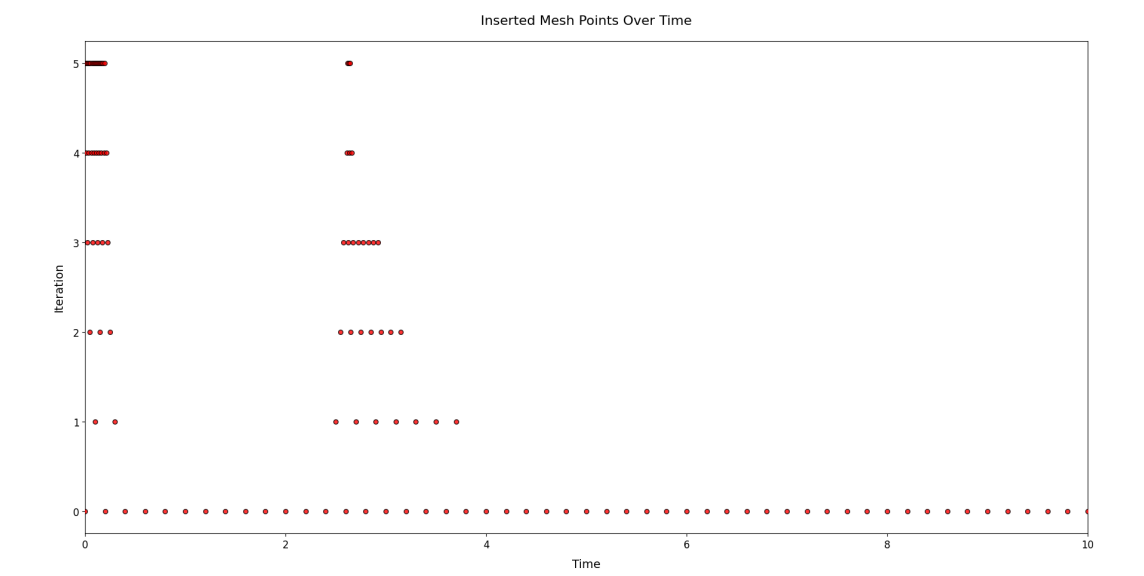


Figure 3: Mesh refinement in the "Van der Pol" example

The plot of the mesh refinement is not very useful on its own. It is therefore possible to combine it with standard plots. In this way, the sections that have been recognized as important or conspicuous by the algorithm can be investigated further.

Draws a combined plot of the mesh refinement and given standard plots.

```
model.plotVarsAndRefinement(meshIteration=None, interval=None,
    specifCols=None, markerSize=30, dotsMesh=Dots.BASE,
    dotsGraph=Dots.OFF)
```

Parameters:

- `int meshIteration` *(optional)*: Sets the mesh iteration of which the optimal values for the standard plots originate. If no value is provided, the last iteration is used.
- `[float, float] interval` *(optional)*: Sets the time range of the plot. Defaults to the interval $[t_0, t_f]$.
- `list<str> specifCols` *(optional)*: Sets the list of variables that will be drawn. These strings have to match the columns names in `model.resultHistory` or equivalently the `symbol` argument of the variables. If no value is provided, all variables will be drawn in the plot.
- `float markerSize` *(optional)*: Sets the size of the dots in the refinement plot.
- `Dots dotsMesh` *(optional)*: Sets which points are displayed in the refinement plot. This has to be an element of the `Dots` enumeration (7.6.8) (excluding `Dots.NONE`) and in the default case base points are added as a dot.
- `Dots dotsGraph` *(optional)*: Adds dots of the discrete solution to the standard plot. This has to be an element of the `Dots` enumeration (7.6.8) and in the default case no dots are added.

Special cases: There is also a special case for input variables. This shares all arguments with `model.plotVarsAndRefinement`, but only contains the plots of the input variables by default.

```
model.plotInputsAndRefinement(meshIteration=None, interval=None,
    markerSize=30, dotsMesh=Dots.BASE, dotsGraph=Dots.BASE)
```

Example:

```
model.plotInputsAndRefinement()
```

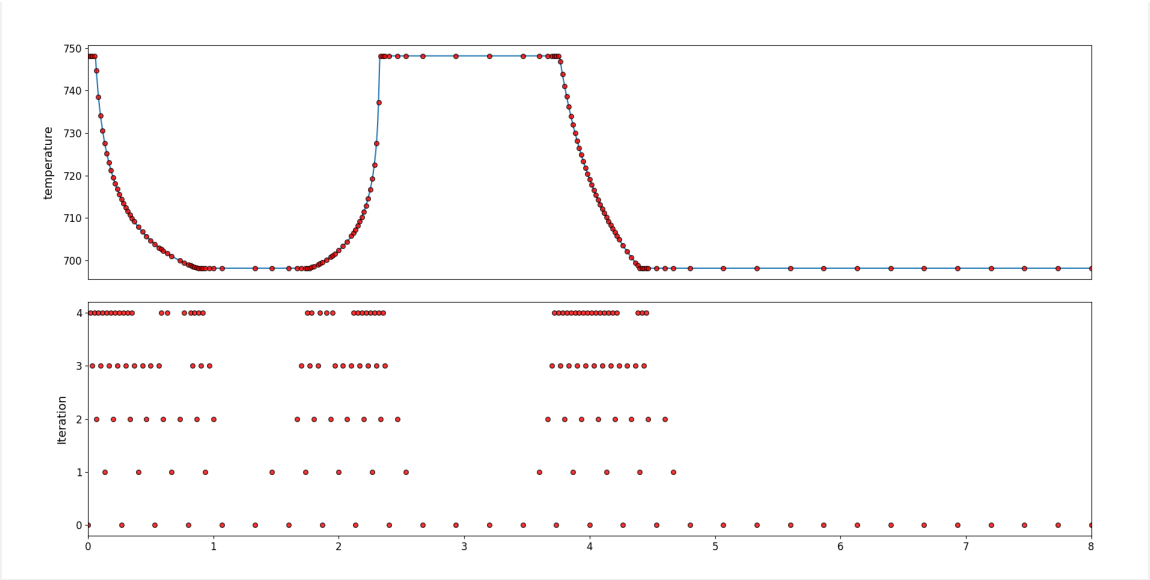


Figure 4: Mesh refinement and temperature control in an "Oil Shale Pyrolysis"

Additionally, the framework is capable of drawing sparsity patterns. To be able to draw a given sparsity pattern the flag "exportJacobianPath" (5.6.1) or "exportHessianPath" (5.6.1) has to be set.

Draws the Jacobian or Hessian sparsity pattern of the resulting large-scale NLP.

```
model.plotSparseMatrix(matrixType)
```

Parameters:

- MatrixType matrixType: Sets the matrix. This has to be an element of the MatrixType enumeration (7.6.7).

Example:

```
model.plotSparseMatrix(MatrixType.JACOBIAN)
```

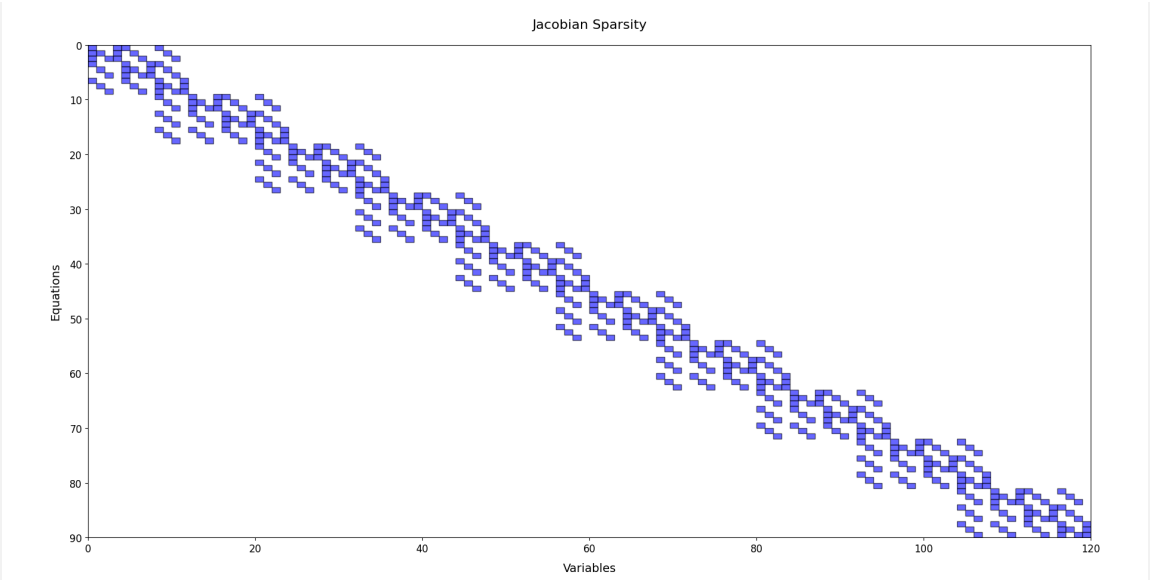


Figure 5: Jacobian sparsity pattern of the "Batch Reactor" example

7 Utilities

The package provides access to many utilities. These contain special functions, global time symbols and many structures to obtain more streamlined flags and arguments.

7.1 Special Functions

By importing `gdopt`, some special functions and symbols get automatically imported from SymEngine. These can be used freely in the modeling process, although the user has to be careful with non differentiable and discontinuous functions.

- `exp`, `log`
- `sin`, `cos`, `tan`
- `sqrt`
- `asin`, `acos`, `atan`
- `sinh`, `cosh`, `tanh`
- `asinh`, `acosh`, `atanh`
- `Abs`
- `Max`, `Min`
- `Piecewise` / `piecewise`
- `pi`

Note that `Piecewise` is the native SymEngine function to define piecewise functions. For code generation to work, this function has to have the fallback case `(0, True)`, i.e. the function is constant 0 anywhere, where it was not explicitly defined. This inconvenience is addressed with the custom `piecewise` function. It is a basic wrapper around `Piecewise`, where the argument `(0, True)` is added implicitly. Piecewise functions allow for greater expressibility in objectives and constraints, e.g.

```
# constraint only has to hold for time < 0.25
binaryTrigger = piecewise((1, t < 0.25), (0, t >= 0.25))
model.addPath(x1 * u1 * binaryTrigger, lb=-30, ub=30)
```

The formulated path constraint is given by $\begin{cases} -30 \leq x_1 u_1 \leq 30, & t < 0.25 \\ -30 \leq 0 \leq 30, & t \geq 0.25 \end{cases}$ and obviously holds for $t \geq 0.25$. It is therefore possible to restrict the domain of constraints to a subinterval $I \subset [t_0, t_f]$. Piecewise functions can be useful in many scenarios, e.g. if the Lagrange integrand only matters on a subset of the entire time horizon, i.e. $\min \int_{t_1}^{t_2} L(\cdot)\, \mathrm{d}t$ with $[t_1, t_2] \subset [t_0, t_f]$.

7.2 Global Time Symbols

In every `Model` two global time symbols can be accessed:

The time symbol can be used in objectives, constraints or in input guesses. This represents the time in the model, i.e. $t \in [t_0, t_f]$.

```
t = Symbol("t")
```

Aliases: `time`, `TIME_SYMBOL`

The constant final time symbol can be used in objectives, constraints, starting values, nominal values, lower and upper bounds, etc. This represents the final time t_f of the model. The associated value will be substituted at runtime.

```
tf = Symbol("FINAL_TIME")
```

Aliases: `finalTime`, `FINAL_TIME_SYMBOL`

7.3 Helper Expressions

Since each expression in the framework is essentially a SymEngine `Expression`, the user is able to use so-called helper expressions throughout the entire modeling process.

E.g. the following dynamic has to be modeled:

$$\dot{x} = \exp\left(x + \frac{u}{1+y^2}\right) + uy$$
$$\dot{y} = \exp\left(x + \frac{u}{1+y^2}\right) + ux$$

By introducing a helper variable, the model becomes way more readable.

```
expTerm = exp(x + u / (1 + y**2))    # helper expression
model.addDynamic(x, expTerm + u * y) # simple dynamic for x
model.addDynamic(y, expTerm + u * x) # simple dynamic for y
```

These helper expressions can also be the Python standard types `float` and `int`. This allows for the modeling of global constants.

E.g. consider this simple free fall dynamic:

```
g = -9.80665               # constant gravitational acceleration
model.addDynamic(v, a + g) # dynamic using the constant
```

7.4 Expression Simplification

If the expressions provided by the user are not simplified yet, it may be advantageous to enable global expression simplification. This is done by adding the line

```
model.setExpressionSimplification(True)
```

directly after the creation of the `Model`.

7.5 Constant Derivatives

As mentioned in chapter (1.2) the framework reduces the continuous formulation of the GDOP (4) to a large scale nonlinear program (NLP). Since the underlying Solver IPOPT requires the 1st and 2nd derivatives of this NLP in every iteration, it is beneficial if these derivatives are constant and thus have to be calculated only once.

The user can call three methods depending on the structure of the continuous problem formulation. These set the corresponding parameters in the back-end of the framework and therefore reduce the execution time.

Only call these methods if the requirements are met, otherwise the Solver converges very slowly or diverges.

Tells the solver that the Mayer term $M(\cdot)$ and Lagrange integrand $L(\cdot)$ are at most linear.

```
model.hasLinearObjective()
```

Tells the solver that the Mayer term $M(\cdot)$ and Lagrange integrand $L(\cdot)$ are at most quadratic.

```
model.hasQuadraticObjective()
```

Tells the solver that all constraints are at most linear, i.e. $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot), \boldsymbol{r}(\cdot), \boldsymbol{a}(\cdot)$ are linear or constant.

```
model.hasLinearConstraints()
```

Depending on the above mentioned methods, the following derivatives of the NLP are assumed to be constant:

- Gradient of the objective function, if `model.hasLinearObjective()`
- Jacobian of the constraint vector, if `model.hasLinearConstraints()`
- Hessian of the augmented Lagrangian, if `model.hasLinearConstraints()` and (`model.hasLinearObjective()` or `model.hasQuadraticObjective()`)

Note that it is also possible to set the flags "linearObjective" (5.6.1), "quadraticObjective" (5.6.1), and "linearConstraints" (5.6.1) to tell the solver that constant derivatives are present.

7.6 Enumerations

7.6.1 Objective

Enumeration for the attribute obj of model.addMayer (5.4.1), model.addLagrange (5.4.2), and model.addObjective (5.4.3): Objective goal.

```
Objective(Enum)
```

Elements:

- MINIMIZE: Minimize the objective.
- MAXIMIZE: Maximize the objective. Will multiply the provided expression with a factor of −1, since the optimizer minimizes.

Aliases: MIN = MINIMIZE, MAX = MAXIMIZE

7.6.2 LinearSolver

Enumeration for the flag "linearSolver" (5.6.1): Specifies the linear solver used by IPOPT.

```
LinearSolver(Enum)
```

Elements:

- MUMPS: MUMPS.
- MA27: MA27 (HSL solver).
- MA57: MA57 (HSL solver).
- MA77: MA77 (HSL solver).
- MA86: MA86 (HSL solver).
- MA97: MA97 (HSL solver).
- PARDISO: Intel PARDISO (currently not supported).

7.6.3 InitVars

Enumeration for the flag "initVars" (5.6.1): Determines how the initial solutions for the state trajectories, that have to be provided to the nonlinear solver, are calculated.

Note that the initial guess for the parameters and input trajectories are given by the `guess` argument in `model.addInput` (5.2.2) and `model.addParameter` (5.2.3) respectively.

```
InitVars(Enum)
```

Elements:

- `CONST`: The initial states are given by $\boldsymbol{x}(t) \equiv \boldsymbol{x}_0$.
- `SOLVE`: The initial states are computed by solving the dynamic using the guesses for $\boldsymbol{u}(t)$ and $\boldsymbol{p}$. The system is solved by `scipy.integrate.solve_ivp`. The corresponding `SciPy` integrator can be set with (5.6.1), which can be implicit.
- `SOLVE_EXPLICIT`: The initial states are computed by solving the dynamic using the guesses for $\boldsymbol{u}(t)$ and $\boldsymbol{p}$. The system is solved by the classic Runge–Kutta method.
- `SOLVE_EXPLICIT_EULER`: The initial states are computed by solving the dynamic using the guesses for $\boldsymbol{u}(t)$ and $\boldsymbol{p}$. The system is solved by the explicit / forward Euler method.

7.6.4 IVPSolver

Enumeration for the flag `"ivpSolver"` (5.6.1): Sets the `SciPy` integrator. If the flag `"initVars"` (5.6.1) is set to `InitVars.SOLVE`, this integrator is used to solve the dynamic and compute an initial state solution.

```
IVPSolver(Enum)
```

Elements:

- `Radau`: RadauIIA of order 5.
- `BDF`: Implicit multi-step of variable-order (1 to 5).
- `LSODA`: Adams/BDF method.
- `DOP853`: Explicit Runge-Kutta method of order 8.
- `RK45`: Explicit Runge-Kutta method of order 5(4).
- `RK23`: Explicit Runge-Kutta method of order 3(2).

Aliases: `RADAU` = `Radau`

7.6.5 MeshAlgorithm

Enumeration for the mesh flag `"algorithm"` (5.6.2): Sets the mesh algorithm.

```
MeshAlgorithm(Enum)
```

Elements:

- `NONE`: No mesh algorithm at all.
- `BASIC`: A very basic mesh algorithm, which is based on the deviation of input variables to its mean.
- `L2_BOUNDARY_NORM`: An advanced mesh algorithm, which is based on evaluating specific L2-norms and comparing derivatives at the interval boundaries. This algorithm is able to detect jumps, steep sections and corners, while being computationally inexpensive.

7.6.6 RefinementMethod

Enumeration for the mesh flag `"refinementMethod"` (5.6.2): Sets the refinement method, which determines how the initial values for $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ are evaluated from the previous iteration, if a mesh refinement is executed.

```
RefinementMethod(Enum)
```

Elements:

- `LINEAR_SPLINE`: The variables are being interpolated by a linear spline on each interval. The initial solution should contain no oscillations at discontinuities, but might be poor for smooth sections.
- `POLYNOMIAL`: The variables are being interpolated by an interpolating polynomial on each interval. The initial solution is likely to contain oscillations at discontinuities, although being advantageous for smooth sections.

7.6.7 MatrixType

Enumeration that specifies a sparse matrix, i.e. Jacobian or Hessian.

```
MatrixType(Enum)
```

Elements:

- `JACOBIAN`: Jacobian matrix.
- `HESSIAN`: Hessian matrix.

7.6.8 Dots

Enumeration for the `dots` argument in all plotting methods. (6.2) Determines if and which values are added to the plot as a dot.

```
Dots(Enum)
```

Elements:

- `OFF`: No dots are added to the plot.
- `ALL`: The state and input values at all grid points are added to the plot.
- `BASE`: The state and input values at the base point of each interval are added to the plot.

8 Example Code

Note that the previously mentioned GitHub repository offers many more examples to get an easy introduction to modeling with this framework. Nevertheless some basic and instructive examples are also provided in this User's Guide.

8.1 Batch Reactor

```
from gdopt import *

"""
* Batch Reactor from Parallel Multiple-Shooting and Collocation
    Optimization with OpenModelica,
* Bachmann, Ochel, et. al., 2012
"""

model = Model("batchReactor")

x1 = model.addState(symbol="Reactant", start=1)
x2 = model.addState(symbol="Product", start=0)

# a non trivial guess for the control variable
controlGuess = guessPiecewise((0.7, t <= 1/2),
                              (guessQuadratic(0.6, 0.7, 5), 1/2 < t))
u = model.addControl(symbol="u", lb=0, ub=5, guess=controlGuess)

model.addDynamic(x1, -(u + u**2 / 2) * x1)
model.addDynamic(x2, u * x1)

model.addMayer(x2, Objective.MAXIMIZE)

model.hasLinearObjective()

model.generate()

model.optimize(tf=1, steps=250, rksteps=3,
        flags={
                "linearSolver": LinearSolver.MUMPS,
                "initVars": InitVars.SOLVE
        },
        meshFlags={
                "algorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "iterations": 6
        }
)
model.printResults()
model.plotInputsAndRefinement()
```

8.2 Oil Shale Pyrolysis

```
from gdopt import *

model = Model("oilShalePyrolysis")

x1 = model.addState(start=1, symbol="kerogen")
x2 = model.addState(start=0, symbol="bitumen")
x3 = model.addState(start=0, symbol="oil")
x4 = model.addState(start=0, symbol="carbon")

T = model.addInput(lb=698.15, ub=748.15, symbol="temperature")

k1 = exp(8.86 - (20300 / 1.9872) / T)
k2 = exp(24.25 - (37400 / 1.9872) / T)
k3 = exp(23.67 - (33800 / 1.9872) / T)
k4 = exp(18.75 - (28200 / 1.9872) / T)
k5 = exp(20.70 - (31000 / 1.9872) / T)

model.addDynamic(x1, -k1 * x1 - (k3 + k4 + k5) * x1 * x2)
model.addDynamic(x2, k1 * x1 - k2 * x2 + k3 * x1 * x2)
model.addDynamic(x3, k2 * x2 + k4 * x1 * x2)
model.addDynamic(x4, k5 * x1 * x2)

model.addMayer(x2, Objective.MAXIMIZE)

model.generate()

model.optimize(
        tf=8,
        steps=200,
        rksteps=3,
        flags={
                "tolerance": 1e-14,
                "linearSolver": LinearSolver.MA57
        },
        meshFlags={
                "algorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "iterations": 5
        }
)

model.printResults()
model.plot()
```

8.3 Van der Pol Oscillator

```
from gdopt import *

model = Model("vanDerPol")

x1 = model.addState(start=0)
x2 = model.addState(start=1)

u = model.addInput(ub=0.8)

rp = model.addRuntimeParameter(default=1, symbol="RP")

model.addDynamic(x1, (1 - x2**2) * x1 - x2 + u)
model.addDynamic(x2, x1)

model.addLagrange(x1**2 + x2**2 + rp * u**2)

model.generate()

model.optimize(
        tf=10,
        steps=50,
        rksteps=3,
        flags={
                "linearSolver": LinearSolver.MUMPS
        },
        meshFlags={
                "algorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "iterations": 5,
                "refinementMethod": RefinementMethod.POLYNOMIAL
        }
)

model.parametricPlot(x1, x2, dots=Dots.BASE)

model.setValue(rp, 0.1)

model.optimize(resimulate=True)
model.plotInputsAndRefinement()
```

8.4 Nominal Batch Reactor

```
from gdopt import *

model = Model("nominalBatchReactor")

NOM = model.addRuntimeParameter(default=1e-10, symbol="NOM")

y1 = model.addState(symbol="Reactant", start=NOM, nominal=NOM)
y2 = model.addState(symbol="Product", start=0, nominal=1 / NOM)

u = model.addInput(symbol="u", lb=0, ub=5, guess=1)

x1 = y1 / NOM
x2 = y2 * NOM

model.addDynamic(y1, -(u + u**2 / 2) * x1 * NOM, nominal=NOM)
model.addDynamic(y2, u * x1 / NOM, nominal=1 / NOM)

model.addMayer(y2, Objective.MAXIMIZE, nominal=1 / NOM)

model.hasLinearObjective()

model.generate()

model.optimize(
tf=1,
steps=500,
rksteps=3,
flags={"linearSolver": LinearSolver.MA57, "initVars": InitVars.SOLVE,
    "tolerance": 1e-14},
)

model.setValue(NOM, 1e10)
model.optimize(resimulate=True)

model.plot(dots=Dots.BASE)
```

References

[1] Wächter, A. and L. T. Biegler, "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming", *Mathematical Programming*, 106(1):25-57, 2006.

[2] Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., & Koster, J. (2001). A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1), 15-41.

[3] HSL. (2013). A collection of Fortran codes for large-scale scientific computation. `http://www.hsl.rl.ac.uk`.

[4] SymEngine Developers, *SymEngine: A Fast Symbolic Manipulation Library*, GitHub repository, `https://github.com/symengine/symengine`, accessed September 27, 2024.

[5] Harris, C. R., Millman, K. J., et al. (2020). Array programming with NumPy. *Nature*, 585(7825), 357-362.

[6] Virtanen, P., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3), 261-272.

[7] McKinney, W. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51-56).

[8] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90-95.

[9] OpenModelica. (n.d.). OpenModelica User's Guide. Retrieved from `https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/optimization.html`.