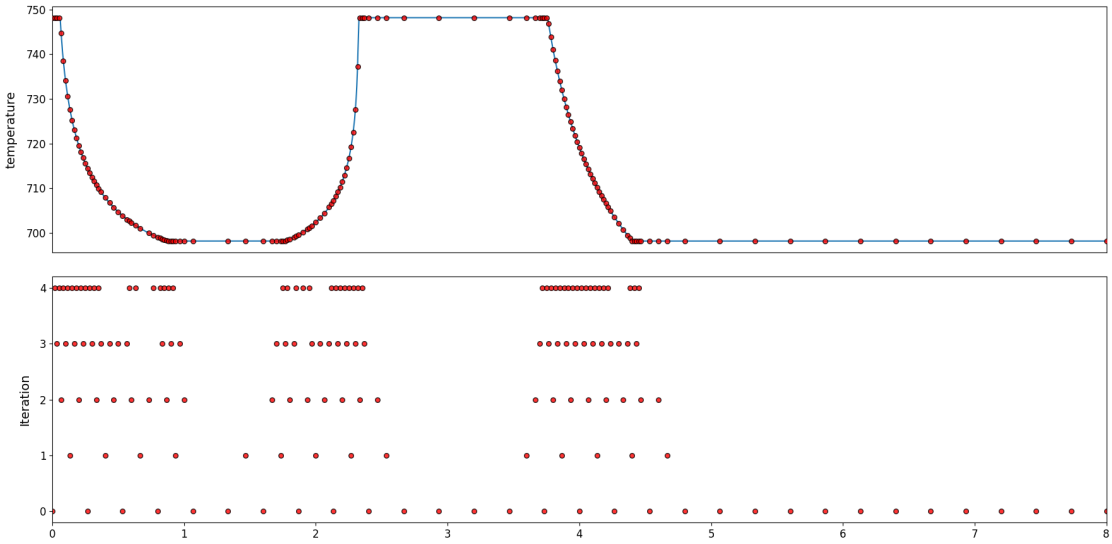
User's Guide:
Dynamic Optimization Framework

A Python-based Environment for Solving Dynamic Models



Author:
Linus Langenkamp

September 22, 2024

Contents

1 Introduction

Dynamic optimization is a branch of mathematical optimization that deals with systems evolving over time. It aims to find the optimal trajectory of a system's state variables by adjusting control inputs while respecting constraints over a defined time horizon. Such problems arise in various fields, including control theory, economics, engineering or industrial biochemistry.

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

1.1 How Does the Framework Solve Dynamic Models?

2 Installation

Clone the repository from GitHub:

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

2.1 Requirements

Ensure that the following dependencies are installed:

- Optimizer
  - IPOPT
- Linear Solver (one of the below)
  - MUMPS
  - HSL solvers, such as MA27, MA57, MA86, ...
- Frontend Python Packages
  - SymEngine
  - NumPy
  - SciPy
  - pandas
  - Matplotlib

3 Getting Started

This section will show 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[1]:

```
from optimization 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

[1] `https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/optimization.html`

- 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 a objective
- Generate the C++ code
- Optimize for a given final time, number of steps and collocation nodes
- Plot the model

3.3 Solution

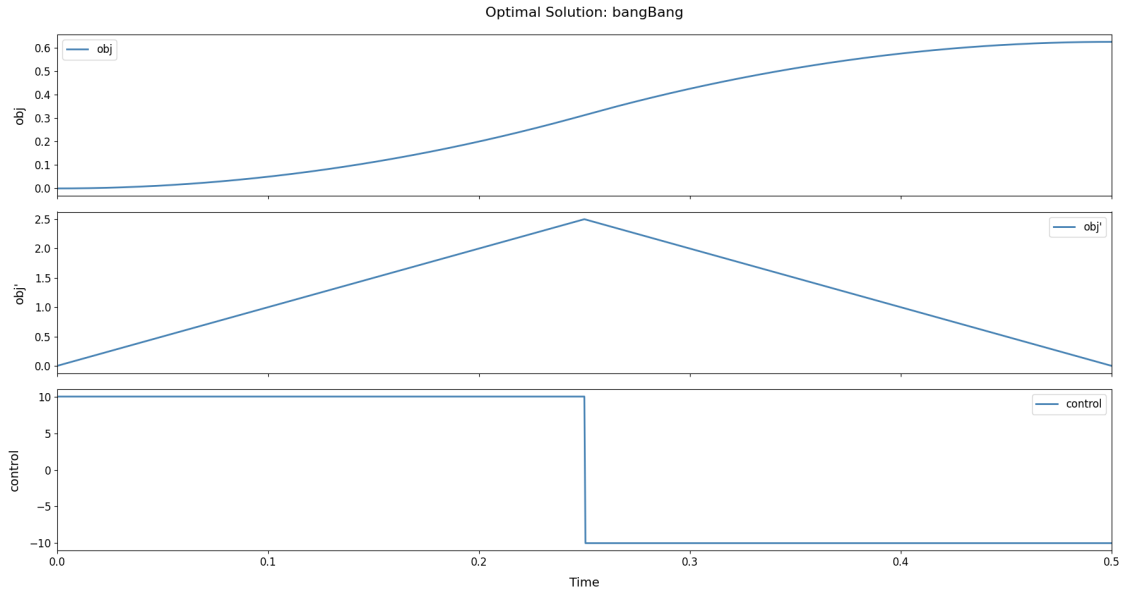


Figure 1: Optimal bang-bang solution provided by the framework

4 Mathematical Problem Formulation

Definition 1 (General Dynamic Optimization Problem)**.**

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

4.1 Variables

4.2 Constraints

4.3 Objective

Variables:
$u(t)$ is the time-dependent control, p is a set of static parameters, t_f is the final time.
Objective:
With the Mayer term $M(\cdot)$ evaluating the final state, and the Lagrange term $L(\cdot)$ aggregating desired combinations of states.
Constraints:
With $\dot{x} = f(\cdot)$ and $x(t_0) = x_0$ as the initial value problem for the states, $g^L \leq g(\cdot) \leq g^U$ as the path constraints for the states and control over the entire time horizon, $h_k^L \leq h_k(\cdot) \leq h_k^U$ as the path constraints for subsets of the time horizon, $r^L \leq r(\cdot) \leq r^U$ as the final constraints for the states and control, and $a^L \leq a(p) \leq a^U$ as the algebraic constraints for the parameters.

5 Modeling

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

```
1 from optimization import *
```

Additionally a `Model` object has to be created. This `Model` is the basis for any modeling and has to have an associated model name.

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

The variables, constraints and objectives will be added to this `Model` object.

5.1 Variables

5.1.1 States

States are time-dependent functions, that will be 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 will be 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`

5.1.2 Inputs

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

Adding a input 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 will be 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 5.1.2.1.
- `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`

5.1.2.1 Input Guesses The guess parameter in `model.addInput` (5.1.2) 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`. This guess trajectory is only used if the `"initVars"` flag

5.5.1 is chosen to be `InitVars.SOLVE` 7.6.3. This trajectory will be used to solve the dynamic and provide an initial, feasible guess for the state variables. 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 will be 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 will be 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 a exponential control guess trajectory. The provided values will be 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 with an Expression as the first and the corresponding interval its defined on as second argument.

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`.

5.1.3 Parameters

Parameters are time-invariant constants, that will be 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 will be 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`

5.1.4 Runtime Parameters

Runtime parameter are time-invariant global constants, that can be changed after code generation via `model.setValue`. The value of the runtime parameter will be resolved at

compile time. 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 compile time.

```
rp = model.addRuntimeParameter(default, symbol)
```

Parameters:

- `float default`: A default value for the runtime parameter.
- `str symbol`: 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.

5.2 Constraints

Any constraint can contain compile time constants such as the global final time 7.2 or runtime parameters 5.1.4.

5.2.1 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)

5.2.2 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 $x_1^2 + u_1^2 \leq 5$.

5.2.3 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 $x_1(t_f) + u_1(t_f) - p = 0$.

5.2.4 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) - cos(p2), lb=0)` represents the parametric constraint $0 \leq \sin(p_1) - \cos(p_2)$.

5.3 Objective

Any objective can contain compile time constants such as the global final time 7.2 or runtime parameters 5.1.4.

5.3.1 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 $x_1 + x_2^2$.

5.3.2 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 $\int_{t_0}^{t_f} x_1(t)u_1(t) + t\,\mathrm{d}t$.

5.3.3 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 will be 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` 5.3.1 and `model.addLagrange` 5.3.2 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 $x_1(t_f) + \int_{t_0}^{t_f} x_2(t)u_1(t)\,\mathrm{d}t$.

5.4 Code Generation

If the modeling process is done, it is possible to generate 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 heavy use of preprocessing macros (e.g. final time, runtime parameters, flags, paths, ...), that will be set later in the `model.optimize` pipeline 5.5.

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 will be avoided.

Calculates the 1st and 2nd derivatives. Generates all the problem defining structures to a .cpp file, which includes preprocessing macros.

```
model.generate()
```

5.5 Optimization

After code generation, all model parameters (e.g. final time, runtime parameters, flags, paths, ...) are written as preprocessing macros to a header file. The resulting code consisting of the .cpp and .h files is compiled, while linking against the dynamic optimization backend. The resulting executable will be run, which solves the NLP. Additionally the optimal solution as well as some metadata will 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, ...), compiles and runs the optimization.

```
model.optimize(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 5.5.1.
- `dict<str, arg> mesh flags` *(optional)*: A dictionary of mesh flags. All possible mesh flags are presented in 5.5.2.
- `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={
                "outputPath": "/tmp",
                "linearSolver": LinearSolver.MUMPS,
                "initVars": InitVars.SOLVE_EXPLICIT,
                "exportJacobianPath": "/tmp",
        },
        meshFlags={
                "meshAlgorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "meshIterations": 6
        }
)
```

5.5.1 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.
- `"outputPath"` → `str`: Sets the path to which the optimal solution is output as a text file. If this value is not provided, the optimal solution can not be accessed. This flag does not have a standard path, since I/O operations can slow down the framework and are obsolete in some cases.
- `"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.

5.5.2 Mesh Flags

The `"meshAlgorithm"` flag specifies the algorithm used for mesh generation in the optimization.

The `"meshIterations"` flag defines the number of iterations for refining the mesh.

The `"refinementMethod"` flag specifies the refinement method used for optimizing the mesh.

The `"meshLevel"` flag defines the level of refinement applied to the mesh.

The `"meshCTol"` flag sets the convergence tolerance for the mesh refinement process.

The `"meshSigma"` flag controls the smoothing factor applied during mesh generation.

6 Analysis

6.1 Results

6.2 Plotting

7 Utilities

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

7.1 Special Functions

By importing `optimization`, 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 has to have the fallback case `(0, True)`, i.e. the function is constant 0 anywhere, where it was not explicitly defined. This inconvenience will be 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 compile time.

```
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 1.1 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 will set the corresponding parameters in the backend of the framework and therefore reduce the execution time.

Only call these methods if the requirements are met, otherwise the Solver will converge very slowly or diverge.

Tells the solver that the Mayer term $M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)$ and Lagrange integrand $L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)$ are at most linear.

```
model.hasLinearObjective()
```

Tells the solver that the Mayer term $M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)$ and Lagrange integrand $L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)$ 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 will 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()`)

7.6 Enumerations

7.6.1 Objective

Enumeration for the attribute `obj` of `model.addMayer` 5.3.1, `model.addLagrange` 5.3.2, and `model.addObjective` 5.3.3: 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"` 5.5.1: 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"` 5.5.1: 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` 5.1.2 and `model.addParameter` 5.1.3 respectively.

```
1 InitVars(Enum)
```

Elements:

- `CONST`: The initial states are given by $\boldsymbol{x}(t) \equiv \boldsymbol{x}_0$.
- `SOLVE`: The initial states will be 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 5.5.1, which can be implicit.
- `SOLVE_EXPLICIT`: The initial states will be 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 will be 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"` 5.5.1: Sets the `SciPy` integrator. If the flag `"initVars"` 5.5.1 is set to `InitVars.SOLVE`, this integrator is used to solve the dynamic and compute an initial state solution.

```
1 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 `"meshAlgorithm"` 5.5.2: 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"` 5.5.2: 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 will be 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 will be interpolated by a 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 will be added to the plot as a dot.

```
Dots(Enum)
```

Elements:

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

8 Example Code

8.1 Batch Reactor

```
from optimization 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)

u = model.addInput(symbol="u", lb=0, ub=5, guess=0.5)

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={
                "outputPath": "/tmp",
                "linearSolver": LinearSolver.MUMPS,
                "initVars": InitVars.SOLVE
        },
        meshFlags={
                "meshAlgorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "meshIterations": 6
        }
)

model.printResults()
model.plotInputsAndRefinement()
```

8.2 Oil Shale Pyrolysis

```
from optimization 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={
```

```
                "outputPath": "/tmp",
                "tolerance": 1e-14,
                "linearSolver": LinearSolver.MA57
        },
        meshFlags={
                "meshAlgorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "meshIterations": 5
        }
)

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

8.3 Van der Pol Oscillator

```
from optimization 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={
                "outputPath": "/tmp",
                "linearSolver": LinearSolver.MUMPS
        },
        meshFlags={
                "meshAlgorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
                "meshIterations": 5,
                "refinementMethod": RefinementMethod.POLYNOMIAL
        }
)

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

```
35
36 model.setValue(rp, 0.1)
37
38 model.optimize(resimulate=True)
39 model.plotInputsAndRefinement()
```

8.4 Nominal Batch Reactor

```
1  from optimization import *
2
3  model = Model("nominalBatchReactor")
4
5  NOM_SCALED = 1e10
6
7  y1 = model.addState(symbol="Reactant", start=NOM_SCALED,
       nominal=NOM_SCALED)
8  y2 = model.addState(symbol="Product", start=0, nominal=1 / NOM_SCALED)
9
10 u = model.addInput(symbol="u", lb=0, ub=5, guess=1)
11
12 x1 = y1 / NOM_SCALED
13 x2 = y2 * NOM_SCALED
14
15 model.addDynamic(y1, -(u + u**2 / 2) * x1 * NOM_SCALED,
       nominal=NOM_SCALED)
16 model.addDynamic(y2, u * x1 / NOM_SCALED, nominal=1 / NOM_SCALED)
17
18 model.addMayer(y2, Objective.MAXIMIZE, nominal=1 / NOM_SCALED)
19
20 model.hasLinearObjective()
21
22 model.generate()
23
24 model.optimize(
25         tf=1,
26         steps=500,
27         rksteps=3,
28         flags={
29                 "outputPath": "/tmp",
30                 "linearSolver": LinearSolver.MA57,
31                 "initVars": InitVars.SOLVE,
32                 "tolerance": 1e-14,
33         }
34 )
35
36 model.plot(dots=Dots.BASE)
```