User's Guide for Local Collocation Dynamic Optimization with Adaptive Mesh Refinement

Linus Langenkamp

September 20, 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, and engineering.

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]:

```python
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
- 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

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

- Optimize for a given final time, number of steps and collocation knots
- Plot the model

3.3 Solution

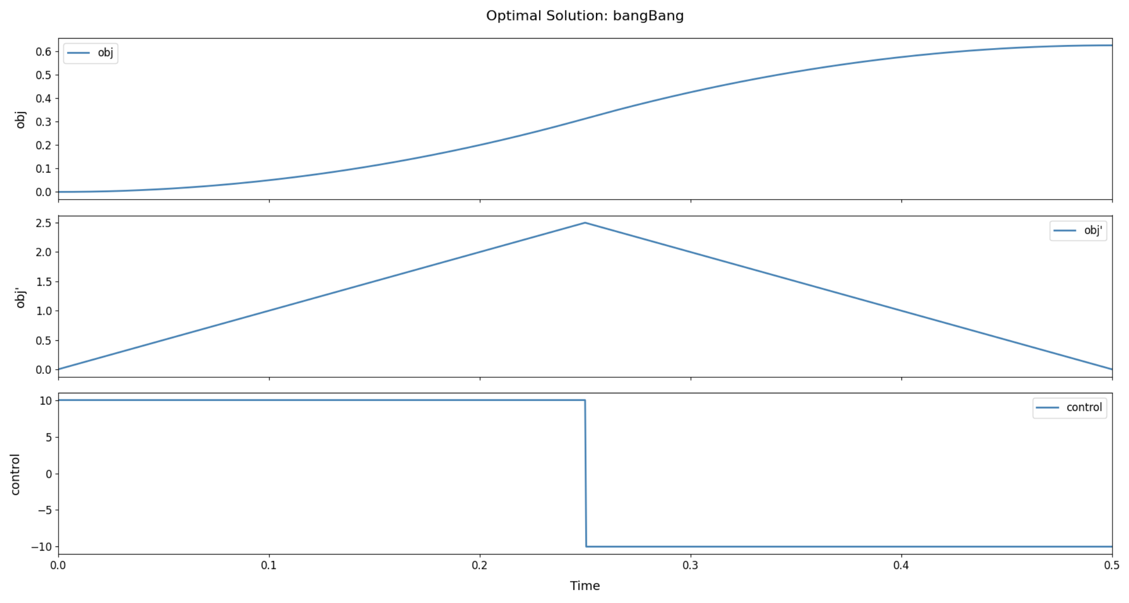

Figure 1: Optimal bang-bang solution provided by the framework

4 Mathematical Problem Formulation

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

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

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.

```python
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.

```python
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-dependant 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-dependant 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

5.6.1.1 is chosen to be `InitVars.SOLVE` 7.3.1. 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 are equivalent to usual parameters in 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. Thus this parameter will not be optimized like the standard parameter.

```
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.

```python
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.

```python
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. These are meant to model e.g. physical limitations of given 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 7.3.2.
- `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 7.3.2.
- `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 adding a factor of -1, if maximization is chosen. If these goals differ between both terms, e.g. $\max M(\cdot)$ and $\min \int_{t_0}^{t_f} L(\cdot)$, then the full objective is given by $\min -M(\cdot) + \int_{t_0}^{t_f} L(\cdot)$. 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 `addMayer` 5.3.1 and `addLagrange` 5.3.2 or with the method `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 $M(\cdot) + \int_{t_0}^{t_f} L(\cdot)$, corresponding to an element of the objective enumeration 7.3.2.
- `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 Constant Derivatives

model.hasQuadraticObjective() model.hasLinearObjective() model.hasLinearConstraints()

5.5 Code Generation

5.6 Optimization

5.6.1 Flags

5.6.1.1 InitVars

5.6.2 Mesh Flags

6 Analysis

7 Utilities

After importing `optimization` the user has 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`, these 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
- pi

7.2 Global Time Symbols

In every `Model` the user can use two global time symbols:

The global time symbol can be used in objectives, constraints or in input guesses. This represents the time in the model.

```
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 of the model. The associated value will be substituted at compile time.

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

Aliases: `finalTime`, `FINAL_TIME_SYMBOL`

7.3 Structures

7.3.1 InitVars

7.3.2 Objective

7.3.3 LinearSolver

7.3.4 MatrixType

7.3.5 Dots

7.3.6 IVPSolver

7.4 Expression Simplification

If the expressions provided by the user are not simplified already, 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`.

8 Conclusion

This guide introduced the main features of the dynamic optimization package, including how to set up, solve, and customize dynamic optimization problems. For more detailed usage and s, refer to the official documentation at `http://your-docs-link.com`.