Modelica2Pyomo: a tool to translate Modelica models into Pyomo optimization models

Matteo L. De Pascali[1] Lorenz T. Biegler[2] Emanuele Martelli[3] Francesco Casella[1]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy,
{matteoluigi.depascali,francesco.casella}@polimi.it
[2]Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, United States of America,
lb01@andrew.cmu.edu
[3]Dipartimento di Energia, Politecnico di Milano, Milan, Italy, emanuele.martelli@polimi.it

Abstract

Tasks involving Modelica models often do not simply investigate the dynamic behavior of a system, but rather also want to characterize possible optimal control strategies according to suitable criteria. Unfortunately, since Modelica does not support out-of-the-box optimization features, users are often forced to use other tools to code again the system model for optimization studies. For this reason, the authors present Modelica2Pyomo, an open-source tool to translate Modelica models into Pyomo optimization programs, leveraging on their flat Base Modelica representation. This work illustrates the main features of Modelica2Pyomo, including automatic variables and constraints normalization, expressions manipulation and initialization via Modelica simulation results. To demonstrate the capabilities of this framework, two examples are showcased, including an industrial-grade open-loop optimal control problem of a solid-oxide fuel cell.

Keywords: Modelica, Model Conversion, Base Modelica, Pyomo, Dynamic Optimization

1 Introduction

Over the years, the Modelica language has established itself as one of the main tools for the formulation of system-level models for dynamic simulation purposes. Even though the size and the complexity of the systems that can be simulated with this language has increased dramatically since Modelica's inception, its object-oriented and equation-based nature, together with its drag&drop flowsheeting feature, significantly tames the challenges faced in the formulation of such models. Moreover, the possibility of highly customizing pre-existing and newly coded models, makes Modelica a suitable choice for academy and industrial R&D users that study bleeding edge technologies (see for example De Pascali and Casella (2024)).

Such studies often do not consist solely of dynamic simulations but they rather aim at characterizing both the system behavior and apply optimization routines to formulate, for example, optimized control strategies according to specific criteria. Since Modelica does not provide out-of-the-box optimization features, this second task requires a different framework. Often, users resort to different software and need to create from scratch a copy of the Modelica model in another programming language. This process requires modeling the same system twice and is prone to *manual translation* errors.

For this reason, the Modelica community spent quite some effort to build a bridge between the simulation and the optimization worlds. The main contributions were initially brought by the open-source software JModelica.org (Åkesson et al. 2010) and OpenModelica (Fritzson et al. 2020), both through the implementation of the Optimica framework (Åkesson 2008). Optimica extends Modelica with language constructs that enable the formulation of optimization problems based on Modelica models. Optimica-enabled tools communicate with optimization solvers such as IPOPT directly or through the CasADi framework (see Wächter and Biegler (2006) and Andersson et al. (2019), respectively).

This approach has several advantages over the *manual translation* approach. Most importantly, the system model is written only once and in an equation-based object-oriented framework: this avoids the possibility of introducing errors and makes it easier to model systems that would be too complex to code in a flat equation-based framework. Moreover, the resulting optimization problem benefits from the pre-processing executed by the Modelica compiler on the original Modelica model before its translation, e.g. alias elimination, index reduction, parameter evaluation, etc., to further reduce the size and the complexity of the problem that is fed to the underlying optimization solver. Finally, Modelica GUIs can support the formulation of complex models with the drag&drop flowsheeting feature mentioned above, easing the user task of connecting different units and components to build the final model to optimally control.

Unfortunately, the open-source JModelica.org development was discontinued in 2019 in favor of the commercial cloud-based software Modelon Impact, which includes the Optimica Compiler Toolkit; the development and maintenance of optimization features in OpenModelica mostly stopped around 2016, so there is currently no available industrial-strength open-source solution to solve dynamic optimization problems on Modelica models.

To fill this gap, the authors present Modelica2Pyomo, an open-source tool developed in Python that allows to automatically translate a Modelica model into an equivalent Pyomo model, that can be used to formulate and solve optimization tasks with state-of-the-art solvers.

The input of the tool is a Modelica model, possibly using advanced object-oriented features such as inheritance, a-causal connections, hierarchically structured models with parameter propagation, expandable connectors, stream connectors, inner/outer objects, etc. The model is first flattened by the open-source OpenModelica tool into an equivalent Base Modelica (Kurzbach et al. 2023) representation, that basically contains a set of variable declarations and a corresponding set of (implicit) differential-algebraic equations (DAEs). As a matter of fact, the case of the Modelica2Pyomo tool is a nice application of the main aim of Base Modelica, which is *"to provide a much lower barrier of entry to the Modelica world, since writing a Base Modelica compiler or interpreter is a much easier task than writing a full-fledged Modelica compiler"* (OpenModelica Consortium 2025).

Pyomo (see Bynum et al. (2021)) is an established open-source software package for formulating and solving large-scale optimization problems in Python, with a clear to read semantics. Pyomo's modeling constructs can be used to express a wide range of optimization problems, including nonlinear and mixed-integer programs. In particular, the package Pyomo.DAE by B. Nicholson et al. (2018) allows to formulate the differential-algebraic constraint equations of optimal control problems in a declarative way, which is very close to flat Base Modelica code. The optimal control problem is then automatically transformed by Pyomo into a form suitable for optimization solvers by means of direct collocation or finite difference discretization methods, including the automatic generation of sparse Jacobians and Hessians.

The output of the tool is a configurable Python script containing the generated Pyomo model with the same variables and equations as the original Modelica model. The formulation of the actual optimization model, i.e., the objective function, additional degrees of freedom, and constraints of the optimization problem can be subsequently added manually to perform a wide range of tasks and analyses.

The main advantage of the Pyomo framework compared to other solutions (e.g., CasADi and Optimica Compiler Toolbox) lies in the wide range of supported optimization solvers, both open-source and commercial, which allows to pick the best-suited one for the specific case at hand. The approach proposed in this work thus offers more flexibility compared to the available alternatives. In some cases, commercial optimization solvers may be required to tackle particularly challenging problems; on the other hand, the remaining part of the required toolchain is entirely open-source.

A limitation of the current implementation of Modelica2Pyomo is that it does not yet benefit from the symbolic manipulations such as alias elimination and index reduction, since the Base Modelica code can currently only be generated by the OpenModelica front-end, whereas these operations are carried out by the back-end. Nonetheless, the complex examples presented in this work can be successfully run without such manipulations. Recents results on the impact of symbolic manipulation of the model equations on the efficiency and convergence robustness of optimization solvers could also be considered, see e.g., (Naik et al. 2025) and (Parker et al. 2022). Such manipulations could be performed on the Pyomo side, but some of them could also be carried out on the Modelica tool side, if they are already implemented for the optimization of simulation code.

This paper is organized as follows: Section 2 describes the Modelica2Pyomo tool in terms of both usage and implementation. Section 3 describes two test cases, showcasing the flexibility of the tool and demonstrating the level of complexity of the Modelica models that can be successfully handled. Section 4 draws the conclusion of this work and outline possible future developments.

The code of Modelica2Pyomo, together with the code of the examples showed in this paper, is available in (De Pascali and Casella 2025).

2 The Modelica2Pyomo tool

The Modelica2Pyomo tool is a Python script that translates a Modelica model into an equivalent Pyomo model. In its current form, the tool accepts as input the flattened Base Modelica representation of a square Modelica simulation model with as many equations as unknown variables, with all the input variables bound to some expressions that only depend on time. The corresponding generated Pyomo model is a degenerate dynamic optimization model with a trivial objective function, no free inputs, and no inequality constraints, whose solution corresponds to the simulation of the original Modelica model. The comparison between the simulation results of the Modelica model and the solution of this degenerate dynamic optimization problem allows to verify that the DAEs have been correctly transcribed by the whole toolchain.

One can then formulate actual dynamic optimization problems by manually editing the generated Pyomo code:

- replacing the objective function with the desired one;
- removing the equations that determine the time-dependent system inputs, thus freeing the inputs that should be determined as a result of the optimization problem;
- optionally adding inequality constraints to the system variables.

2.1 Tool inputs

To generate the equivalent Pyomo model, Modelica2Pyomo requires as input the flattened Base Modelica code of the Modelica model and optionally the result file of a Modelica simulation of the model to provide an initial guess to the solver. Employing Base Modelica code as input allows to process a flat representation of the Modelica model which only contains lists of variables, functions and record declarations, equations and initial equations, without bothering about object-oriented features such as modularity, connectors, inheritance, etc. This greatly simplifies the writing of the Modelica2Pyomo tool.

Additionally, in order to minimize the tool development work, the generated BaseModelica code needs to fulfill some additional requirements, which are obtained by flattening the model with OpenModelica and setting some special flags in the compiler:

- `-d=evaluateAllParameters` replaces all parameter instances in the equations with their literal numerical value; this allows to skip the parameter section of the model entirely, as well as the handling of dependencies among parameters enforced by non-trivial parameter-binding expressions; it also prevents conditional equations and expressions depending only on parameters to show up in the equation that will be passed to Pyomo;
- `--frontendInline` enables inlining of functions (i.e., the body of a function is inserted in all places where the function is called, swapping the formal inputs with the actual inputs), to make the resulting model fully equation-based.
- `--baseModelicaOptions=moveBindings` moves all the binding equations of Real variables into the `equation` section, so there is no need to parse binding equations separately from regular equations;
- `--baseModelicaOptions=scalarize` rolls out array variable declarations and array equations (including `for` loops) into their scalar elements, so that each variable declaration corresponds to a scalar variable and each equation is a scalar equation in the Base Modelica code; this also minimizes the tool development effort, because there is no need to handle arrays, slicing, iterators, etc., which are all handled upstream by the OpenModelica tool.

2.2 Tool capabilities and limitations

Modelica2Pyomo can handle Modelica models of arbitrary complexity from an object-oriented point of view: features such as models defined by instantiating components, parameter passing, a-causal connectors, composite connectors, stream connectors, expandable connectors, inner/outer components, inheritance, replaceable classes and components, N-dimensional arrays of variables and components, are all supported. The availablity of a Base Modelica interface with a state-of-the-art compiler such as OpenModelica makes it possible to off-load the processing of all these features onto the Modelica tool, so that the implementation of the Modelica2Pyomo tool can be kept very lean, while remaining always be up-do-date with the latest additions to the Modelica language, as long as an updated version of the Modelica tool is used.

This approach can make Modelica2Pyomo a very attractive tool for the Pyomo community. Pyomo indeed includes several features for the handling of modular, structured models, but they are by no means as powerful as those made available by the Modelica language. In the authors' opinion, it would be a waste of time to re-invent yet another full-fledged object-oriented modelling language for Pyomo – a much more efficient approach is to re-use Modelica as a modelling front-end (including drag-and-drop graphical user interfaces such as OMEdit) and then use OpenModelica for flattening and Pyomo as an optimization back-end. Modelica2Pyomo is meant to be a first step in this direction.

The main limitation on the models that Modelica2Pyomo can handle stems from the capabilities of the Pyomo.DAE tool it is based upon. Pyomo.DAE can only take as input purely equation-based index-1 DAEs, described by scalar variables and equations. Hence, the upstream Modelica tool must be able to reduce the original model to that form. This entails some restrictions in the set of supported models.

First and foremost, the original Modelica model must be purely continuous-time; hence, it should not contain discrete or clocked variables. Additionally, Pyomo.DAE expects the equations to be continuously twice-differentiable; this also rules out conditional expression and equations, which are potentially not differentiable, and which in any case have no counterpart in Pyomo.DAE.

Conditional expression could be supported by Modelica2Pyomo by replacing expressions such as `if a > b then c else d` with analytic expressions such as

$$\frac{1}{2}\left[d+c+\tanh\left(\frac{a-b}{\varepsilon}\right)(d-c)\right], \tag{1}$$

where ε is a suitably small constant; with the same logic, conditional equations such as

```
if a > b then
  x = y;
else
  z = w;
end if;
```

could be approximated with the following smoothed approximation to a complementary condition:

$$z+x-w-y+\tanh\left(\frac{a-b}{\varepsilon}\right)(z+y-w-x)=0. \tag{2}$$

This feature is not yet implemented at the moment, but it could be easily added in the future.

Last, but not least, Pyomo can only handle index-1 DAE models; at the time of this writing this implies that the original Modelica model also has to be index-1, because OpenModelica currently outputs the flat Base Modelica code immediately after the flattening phase, prior to structural analysis and index reduction. It is possible to check that this condition holds by compiling the Modelica model for simulation with the flags `--preOptModules-=removeSimpleEquations -d=bltdump`, which reports whether index reduction is necessary to obtain an index-1 system. This restriction may be lifted in the future if OpenModelica becomes capable of outputting index-reduced Base Modelica code.

As mentioned at the beginning of Section 2, the originally supplied Modelica model must have prescribed input variables, which will then be freed and determined as the result of the optimization problem. It is possible, and in fact recommended, to also provide a reference simulation result file, obtained with those prescribed input values, which will be employed to provide an initial guess for the optimal solution, considerably improving the chances of convergence of the optimization problem. If no result file is supplied, the variables will be initialized to zero.

As a final remark, the Modelica2Pyomo tool was mainly designed to solve *dynamic* optimization problems, where the constraint equations are DAEs. However, as a special case, it can also solve *static* optimization problems, where all the constraint equations are purely algebraic. This can be helpful in a number of application fields; for example, in the case of thermal power generation systems, it can be first used to compute optimal (i.e., maximum efficiency) steady-state off-design operating points, and later to compute optimal (i.e., fast) transients to move between them at maximum speed while respecting operational constraints such as maximum temperature gradients.

2.3 Notable Modelica2Pyomo features

A crucial feature offered by Modelica2Pyomo is the automatic normalization of variables, constraints and objective function.

One aspect of declarative equation-based modeling is that the use of dimensionally consistent SI units for the physical variables is preferable; however, in many application areas this can lead to systems of equations which are badly scaled from a numerical point of view. As explained by Casella and Braun (2017), even though this is usually not declared explicitly in the literature and in the documentation about optimization solvers, the implicit assumption taken by their developers is that the unknowns of the problem have an order of magnitude close to unity or at least not too far from that, e.g. in the range $10^{-3} - 10^{3}$. The optimization of badly scaled models of significant complexity are inevitably bound to fail, a fact further confirmed by the authors' experience with convergence failures before scaling was included in Modelica2Pyomo.

For these reasons, the normalization procedures described in Casella and Braun (2017) are implemented in Modelica2Pyomo. For the variables, the normalization value is chosen as the maximum value between the assigned nominal value and the initialization value assigned to the variable in the first time instant.

Regarding the constraint residuals and the objective function, again as suggested by Casella and Braun (2017), they are scaled by the factor

$$f_{norm} = \max_j \frac{df}{dx_j} x_{j,norm} \tag{3}$$

where x_j are the unknown variables and $x_{j,norm}$ are their normalization values. The lines of code responsible for the variable normalization are placed after each variable declaration while the code for the constraints and objective function normalization is included in the Pyomo script right before the solver declaration at the end of the file.

Modelica2Pyomo offers also an expression manipulation feature that consists in the replacement of logarithm expressions with a more suited equivalent formulation. Since logarithms are not defined for arguments equal or less than zero, the solvers will encounter issues if their argument become negative during Newton-Raphson iterations. An alternative and equivalent formulation can be introduced to overcome this issue:

$$\begin{cases} \ln(x) \to u \\ \exp(u) - x = 0 \end{cases} \tag{4}$$

The logarithm is replaced by an auxiliary variable u and the second equation, which does not suffer from the same domain limitations of the logarithmic function, is enforced to replace the logarithm expression.

Concerning the initial guess values for the optimization problem, Modelica2Pyomo offers three options:

1. no initial guesses are employed (all initial guesses are zero);
2. each variable is initialized to a constant value obtained from the results file of the Modelica simulation at a specified time instant;
3. each variable is initialized with the time evolution of the corresponding Modelica variable obtained from the results file of a suitable reference simulation.

Finally, if a dynamic optimization problem is formulated, two different options are provided to specify its initial conditions:

1. employ the initial equations of the Modelica model;
2. fix the initial values of the variables appearing in the Modelica derivative operator according to the values reported in the results file of the simulation.

The first option is in principle more flexible, since it covers those cases where the initial condition is not fixed a priori but is also part of the optimization problem; on the other hand, difficult-to-solve initialization problem could lead to convergence problems of the optimization solver.

The second option only works in the case of initial conditions that are fixed for the optimization problem, with the advantage that it can leverage on advanced features of the Modelica tool (e.g., homotopy) to successfully solve hard initialization problems. This is essential in cases such as the steady-state initialization of thermal power generation systems.

2.4 Modelica2Pyomo script overview

Modelica2Pyomo is organized in several functions, which are all executed through the *m2p()* function. The arguments that should be passed to this function, together with the paths of the Modelica model and the optional results file, are the following:

- a string with the name of the Python file that will contain the Pyomo model;
- a string with the name of the solver to be employed;
- a string specifying whether the generated problem will be *Static* or a *Dynamic* optimization program;
- in case of a dynamic optimization problem, a string specifying whether the initial equation of the Modelica model should be translated or the initial values of the states should be retrieved from the results file;
- in case of a dynamic optimization problem, the start and finish time instants of the simulation;
- in case of a dynamic optimization problem, a dictionary containing information about the discretization method (e.g. finite differences or collocation) and the respective settings (e.g., number of finite elements, discretization scheme, etc.);
- a flag to enforce the *min*/*max* attributes in the imported reference simulation file;
- a flag to enable expressions manipulation, e.g., logarithms transformations;
- a string containing the Pyomo code to specify the objective function;
- optional strings containing additional constraints and utility Python commands to be inserted before and after the solver definition lines of the Pyomo model.

Once it is run, the tool performs the set of operations that is briefly described below. Each step roughly corresponds to a Python function, which adapts to the user requirements according to the input described above.

1. read the Base Modelica file and store each line in a list;
2. store in a Python dictionary with the DyMat Python package by Jörg (2011) the numerical results of a Modelica simulation of the system;
3. remove from the Base Modelica code comments and other characters which are not related to the model description;
4. identify, if present, the lines related to *TimeTables* and *CombiTimeTables* declarations, store the tables and instantiate a string containing their Pyomo code;
5. analyze the variable section of the Base Modelica model and store name, min, max and nominal values of each variable;
6. employ the information of the previous step and the DyMat results dictionary to build a string containing the necessary code to instantiate the variables of the model, setting min/max bounds, initial values and normalization value employing the Pyomo syntax;
7. analyze the equation section of the Base Modelica model and instantiate a string containing the equations of the problem using the proper Pyomo syntax. In addition, look for the following cases:
    - if a variable is set equal to a number, use the Pyomo *fix* variable attribute to model the equation instead of the original formulation;
    - if a logarithm expression is present and the optional flag is enabled, store the logarithm argument and replace the logarithm as shown in Equation (4);
    - if a time derivative operator is present, store the associated variable to create a list of the states;
8. analyze the initial equation section and instantiate a string with additional equations to define the Modelica parameter characterized by the *fixed = true* attribute, which, otherwise, would be missing their defining equation;
9. use the list of state variables created in the previous step to instantiate the respective time derivatives with the Pyomo *DerivativeVar* object;
10. analyze the initial equation section to instantiate a string featuring the initial equations of the Pyomo optimization problem;
11. write on a .py file the Pyomo code contained in the string created in the previous steps together with the objective function, constraints and objective function normalization code, solver settings and other custom lines included as arguments for the tool.

To execute many of the steps listed above, Modelica2Pyomo employs regular expressions through the Python native re and regex (Matthew 2025) packages to search for predefined patterns in the rows of the Base Modelica file and stores information about variables and constraints according to the specification of the Base Modelica language. Moreover, the quoted identifiers used for Base Modelica variables are adapted to remove the quotes, the Modelica dot notation, square brackets and commas, which are all replaced with underscores in the Pyomo model in order to comply with Python syntax rules.

As an example, Appendix A reports a simple, first-order model formulated in Modelica, its Base Modelica flat representation, and the corresponding Pyomo script generated by the Modelica2Pyomo tool.

3 Test cases

In this Section, two examples of Modelica model translation are showcased; both models are converted into open-loop dynamic optimal control problems, which are then successfully solved by the Pyomo framework. The first example is a simple, conceptual model that can be useful to understand how the tool works. The second example, instead, is a full-fledged industrial-grade system model, aimed at demonstrating the capability of the tool to handle real-life complexity models.

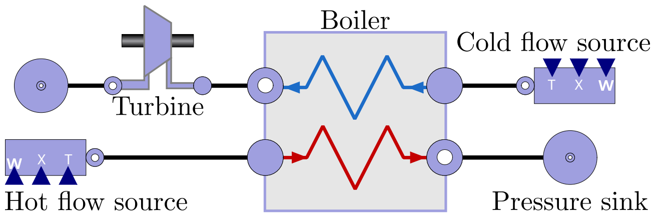

Figure 1. Modelica diagram of the s-CO$_2$ boiler and turbine model.

The corresponding BaseModelica flat models, which are the actual inputs of the tool, are available in the Examples directory of the Modelica2Pyomo package (De Pascali and Casella 2025).

3.1 Conceptual s-CO$_2$ boiler-turbine system

The first test features a conceptual Modelica model of a supercritical CO$_2$ boiler-turbine system. Figure 1 shows the diagram of the model. A hot flue gas stream heats up a cold stream of supercritical CO$_2$ which is expanded in a turbine with a prescribed outlet pressure to produce electrical power. The two streams are generated by ideal mass flow rate sources and their flow rate and temperatures can be changed arbitrarily. The medium model for supercritical CO$_2$ is the implicit Peng-Robinson equation of state, written in the equation section of the model.

In the first test case, the flow rates are fixed to constant values and are embedded in the optimization problem with the Pyomo *fix* attribute. The selected simulation time is 500 s and direct collocation with 100 finite elements, each with 3 collocation points, is the employed discretization method. Since the Modelica model is a square system having the same number of variables and equations, the translated optimization problem does not feature degrees of freedom, thus its optimal solution coincides with the simulated Modelica trajectory.

The coincidence of the two solutions demonstrates that the Modelica model was successfully translated into an equivalent Pyomo model, having the same solution.

To formulate a meaningful optimization problem, first two degrees of freedom are introduced, by *unfixing* the two mass flow rate variables (w_{hot} and w_{cold}), then a suitable cost function is chosen to guide the plant along transient conditions, as follows. The system is required to track a time-varying power output P_{turb} setpoint while keeping the turbine inlet temperature TIT constant. To this end, the following objective function is employed, with overbarred variables being the required setpoints, k indicating the simulation time step and coefficients, A_i being suitable weights for the different components of the expression:

$$\begin{aligned}\min \sum_k \, & (A_1 \cdot (P_{\text{turb}}(k) - \overline{P}_{\text{turb}}(k))^2 \\ & + A_2 \cdot (\text{TIT}(k) - \overline{\text{TIT}}(k))^2 \\ & + A_3 \cdot (w_{\text{hot}}(k) - w_{\text{hot}}(k-1))^2 \\ & + A_4 \cdot (w_{\text{cold}}(k) - w_{\text{cold}}(k-1))^2)\end{aligned} \tag{5}$$

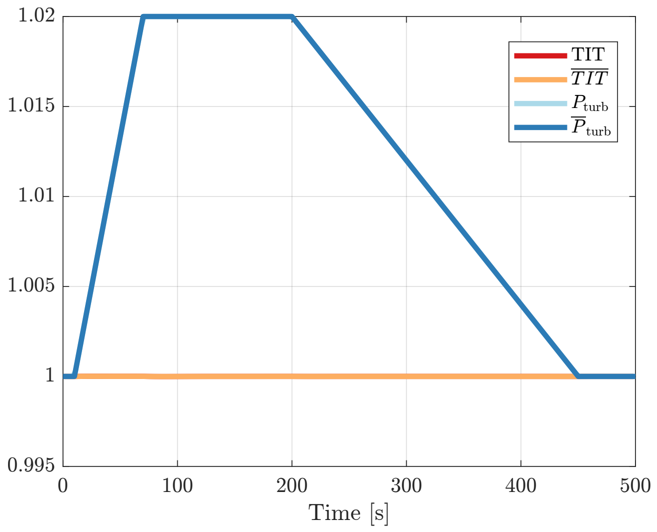

Figure 2. P_{turb} and TIT normalized profiles.

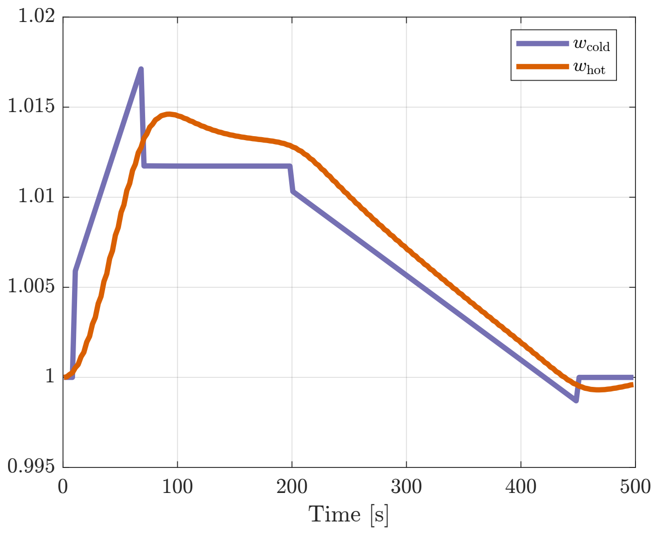

Figure 3. w_{hot} and w_{cold} normalized profiles.

The third and fourth terms of the objective function in (5) limit the rate of change of the optimization variables, moderating the control action.

Figure 2 shows that both P_{turb} and TIT follow accurately their setpoints, while Figure 3 shows the optimization variables w_{hot} and w_{cold}, that, as prescribed by the objective function, do not show abrupt variations. This first example shows that this class of simulation problems can be effectively translated into a nonlinear optimization problem, that features roughly 10000 variables and constraints. The most time consuming operation is the objective function normalization routine, while the Modelica to Pyomo compilation, the variables and constraints instantiation, the constraints normalization and the solver overhead is negligible. Table 1 summarizes these considerations and also shows a comparison between the open-source solver IPOPT and the commercial closed-source solver CONOPT 4.

Table 1. Performance results for the s-CO$_2$ boiler-turbine test

	Time [s]
Modelica2Pyomo model translation	2.3
Pyomo variables and Constraints instantiation	0.2
Constraints normalization	0.01
Objective function normalization	12
Solver: IPOPT	0.3
Solver: CONOPT 4	0.8

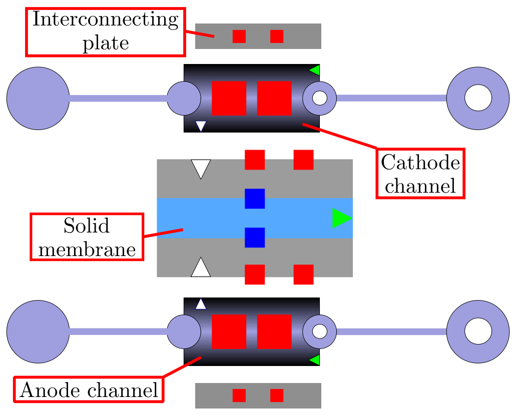

Figure 4. Modelica diagram of the solid-oxide fuel cell model.

3.2 Solid-oxide fuel cell

While the Section above showcased a toy example with relatively few equations, the goal of this Section is to present a relevant industrial case study. To this end, the 1-D solid-oxide fuel cell (SOFC) model presented by De Pascali, Donazzi, et al. (2023) and employed by De Pascali and Casella (2024) to model a complex thermal power generation system, is fed to Modelica2Pyomo to formulate an open-loop optimal control problem. The model has about 9000 variables and equations. Note that the full source code of the model is publicly available on GitHub. The model of the SOFC (see the Modelica diagram in Figure 4) is extremely nonlinear, due to the presence of strongly coupled nonlinear thermal, chemical and electrochemical phenomena. These describe the conversion at high temperatures of natural gas into hydrogen and its oxidation with oxygen ions to produce water, with the final aim of generating electrical power without resorting to fuel combustion.

The extensive use of object-oriented modeling techniques, that are necessary to tame the complexity of the formulation of such a challenging model, would make an hypothetical *manual translation* into an optimization framework such as Pyomo.DAE quite difficult, time consuming, and error-prone. This is exactly the reason that motivated the work presented in this paper: Modelica2Pyomo is capable of *automatically* converting such a complex Modelica model featuring 10000 equations and 160 states into a working Pyomo model.

As in the previous case, to formulate a dynamic optimization problem, direct collocation is used to discretize in time the simulation: for a 120 s simulation with 35 time steps, each with 3 collocation points, a NLP with roughly 850000 equality constraints is formulated. The goal of this test is to evaluate a trajectory for the SOFC extracted current I to optimally follow an electrical power output P_{SOFC} setpoint, while constraining the maximum allowed rate of change of the temperatures T of the fragile ceramic solid membrane of the SOFC. The following constraints is thus added manually to the Pyomo model for all the temperature variables of the SOFC solid membrane:

$$T(k) - T(k-1) < \frac{dT_{max}}{dt} \cdot \Delta t \tag{6}$$

$$T(k) - T(k-1) > -\frac{dT_{max}}{dt} \cdot \Delta t \tag{7}$$

where dT_{max}/dt is the maximum temperature rate of change prescribed by fuel cell manufacturers and Δt is the length of the simulation time step. In this test, dT_{max}/dt is varied to show how the solver is forced to follow less accurately the power setpoint $\overline{P}_{\mathrm{SOFC}}$ when its variation is too steep. The following objective function is chosen:

$$\begin{aligned}\min \sum_k \big(& B_1 \cdot (P_{\mathrm{SOFC}}(k) - \overline{P}_{\mathrm{SOFC}}(k))^2 \\ & + B_2 \cdot (I(k) - I(k-1))^2 \\ & + B_3 \cdot (T(k) - T(k-1))^2 \big)\end{aligned} \tag{8}$$

Also in this case, the coefficients B_i are employed to weight the contributions of the different terms of the objective function and the last two components are employed to moderate the control action.

Figure 5 shows the SOFC power output considering two different values for dT_{max}/dt: in the stricter case, $dT_{max}/dt = 0.2\,\mathrm{K/s}$, while in the milder case, this value is three times higher. As expected, the control performance are worse in the stricter case, and the power setpoint is followed less accurately. Figure 6 shows the trends of the temperature of the solid membrane at the near the outlet of the SOFC, which respect the prescribed maximum rate of variation. Figure 7 shows the current I extracted from the fuel cell, which is the manipulated variable determined by the optimization problem.

Also in this much more challenging test case, Modelica2Pyomo is able to formulate a meaningful and working Pyomo model with no effort by the end user except for the manual inclusion of additional constraints for the membrane temperatures and the objective function. Compared to the previous example, Table 2 shows comparable compilation and instantiation times, while the time required by the solvers is higher, given the increased size of the optimization problem. Notice that the time employed to normalize the objective function is lower, since this simulation features less time steps, compared to the previous one. Moreover, IPOPT was not able to solve this model,

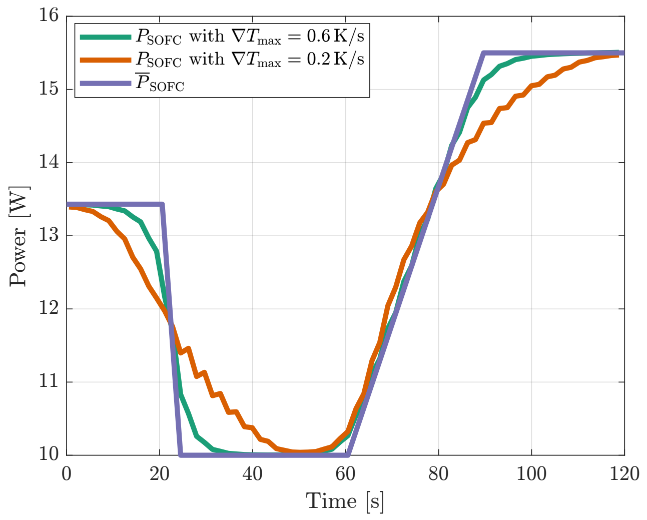

Figure 5. P_{SOFC} at varying dT_{max}/dt.

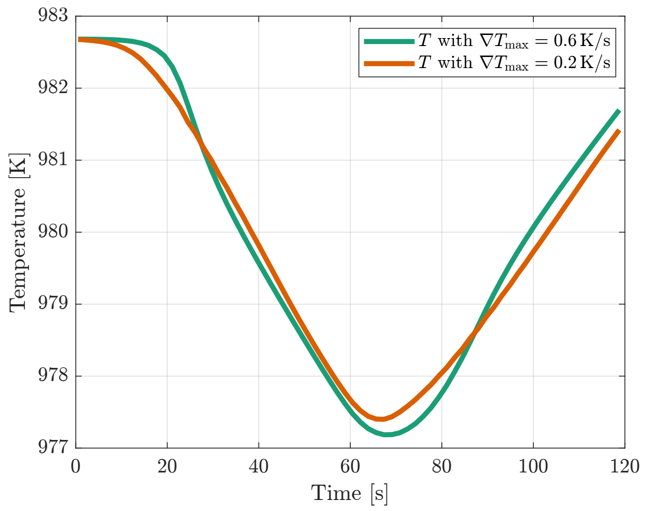

Figure 6. Membrane T at the SOFC discharge at varying dT_{max}/dt.

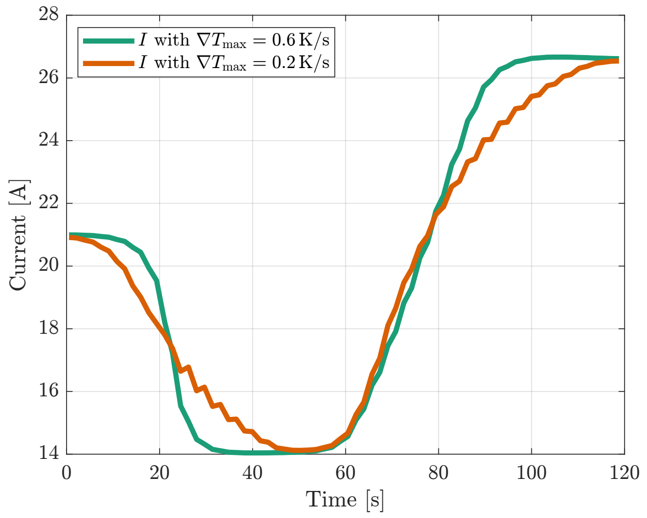

Figure 7. I extracted from the fuel cell at varying dT_{max}/dt.

Table 2. Performance results for the solid-oxide fuel cell test

	Time [s]
Modelica2Pyomo model compilation	3.7
Pyomo variables and Constraints instantiation	6.7
Constraints normalization	1.3
Objective function normalization	0.7
Solver: CONOPT 4	185

possibly indicating that a feasible path, active-set solver like CONOPT might be more suited for this class of problems, compared to an interior point method like IPOPT.

4 Conclusion

In this paper, the Modelica2Pyomo tool was presented. The aim of this tool is to create an open-source, flexible, and lightweight bridge between Modelica and the optimization world.

Modelica2Pyomo is a Python script that translates Modelica models into Pyomo models, exploiting the OpenModelica tool for flattening, the new Base Modelica language for flat Modelica model representation, and the Pyomo.DAE tool for writing DAE constraints for Pyomo in a high-level fashion. The flat, scalarized Base Modelica representation obtained through OpenModelica uses a very nimble set of language constructs, that can be efficiently handled by regular expressions to find Base Modelica language patterns that are converted into Pyomo expressions, without the need of embarking in the traditional workflow of compiler development, with parsers, lexers, abstract syntax trees, etc.

This approach allowed to drastically cut the required development and maintenance effort from the tens of person-years required for the development of a full-fledged Modelica compiler down to a few person-months effort, a nice demonstration of the potential that can be unlocked by the forthcoming Base Modelica standard.

The Modelica2Pyomo tool produces the Pyomo code corresponding to the set of DAE constraint equations, including their proper scaling, the transformation of some expression to more numerically robust equivalent ones, and the capability of importing intial guesses for the NLP solver from reference simulation results. The actual optimization problem (constraints and objective function) is then directly formulated using the Pyomo tool, which allows flexible access to a range of different solvers, both open-source and commercial.

The results obtained in two example test cases were reported, showing the capabilities of the proposed framework, which is able to successfully convert industrially-relevant complex Modelica models into nonlinear optimization programs with satisfying performance.

The code of the tool is available as open source under GPL v3 license. Contributions for its further improvement and developments are welcome.

Future research directions will extend the range of Modelica constructs that can be handled by the tool, e.g., by handling conditional expressions and equations by regularization, improve the numerical robustness of the translated Pyomo.DAE model, as well as test the framework on even more challenging problems, e.g., optimizing transients of an innovative power plant that features the fuel cell model studied in this work as one of its components. One could also experiment with the use of optimization techniques for solving hard steady-state initialization problems, using Pyomo as a protoyping environment. It would also be interesting to explore the optimal control of Modelica systems including time delays and discrete variables, as well as optimization with embedded DAE solvers (shooting methods).

Modelica2Pyomo clearly demonstrates how Base Modelica can successfully enable the development of lean tools that employ Modelica models for uses other than simulation. The authors hope that this example further motivates the standardization effort of the Base Modelica language, as well as its improved support in Modelica tools.

Acknowledgements

This work was supported by the Italian Ministry of University and Research under Programma Operativo Nazionale "Ricerca e Innovazione" PONRI 2014-2020.

References

Åkesson, Johan (2008-03). “Optimica - An Extension of Modelica Supporting Dynamic Optimization”. In: *Proc. 6th Intl. Modelica Conference*. Bielefeld, Germany, pp. 57–66.

Åkesson, Johan et al. (2010). “Modeling and optimization with Optimica and JModelica.org — Languages and tools for solving large-scale dynamic optimization problems”. In: *Computers & chemical engineering* 34.11, pp. 1737–1749.

Andersson, Joel AE et al. (2019). “CasADi: a software framework for nonlinear optimization and optimal control”. In: *Mathematical Programming Computation* 11, pp. 1–36.

Casella, Francesco and Willi Braun (2017-12). “On the importance of scaling in equation-based modelling”. In: *8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2017*. Wessling, Germany, pp. 3–7. DOI: 10.1145/3158191.3158192.

Bynum, Michael L. et al. (2021). *Pyomo-optimization modeling in python*. Vol. 67. s 32. Springer.

De Pascali, Matteo Luigi and Francesco Casella (2024). “Modeling Innovative Thermal Power Generation Systems for the Energy Transition with Modelica: an Open Source and Flexible Solution”. In: *2024 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, pp. 533–538.

De Pascali, Matteo Luigi and Francesco Casella (2025). *Modelica2Pyomo Repository*. https://github.com/looms-polimi/Modelica2Pyomo.

De Pascali, Matteo Luigi, Alessandro Donazzi, et al. (2023). “A control-oriented Modelica 1-D model of a planar solid-oxide fuel cell for oxy-combustion cycles”. In: *2023 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, pp. 394–399.

Fritzson, Peter et al. (2020). “The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development”. In: *Modelling, Identification, and Control* 4, pp. 241–285.

Jörg, Rädler (2011). *DyMat*. www.github.com/jraedler/DyMat.

Kurzbach, Gerd et al. (2023). “Design proposal of a standardized Base Modelica language”. In: *Proc. 15th International Modelica Conference*. Aachen, Germany, pp. 469–478.

Matthew, Barnett (2025). *mrab-regex*. www.github.com/mrabarnett/mrab-regex.

Naik, Sakshi et al. (2025). *Variable aggregation for nonlinear optimization problems*. URL: https://arxiv.org/abs/2502.13869.

Nicholson, Bethany et al. (2018). “pyomo.dae: A modeling and automatic discretization framework for optimization with differential and algebraic equations”. In: *Mathematical Programming Computation* 10, pp. 187–223.

OpenModelica Consortium (2025). *Flattening models to BaseModelica*. www.openmodelica.org/doc/OpenModelicaUsersGuide/latest/basemodelica.html.

Parker, R. et al. (2022). “An implicit function formulation for optimization of discretized index-1 differential algebraic systems”. In: *Computers & Chemical Engineering* 168.

Wächter, Andreas and Lorenz T. Biegler (2006). “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming”. In: *Mathematical programming* 106, pp. 25–57.

A Appendix: Source code of a simple first-order example

```
model Test
  parameter Real tau = 5;
  parameter Real mu = 30;
  Real u;
  Real x(nominal = 1, start = 1, fixed = true);
  Real y1(nominal = 50);
  Real y2(nominal = 30) = 0.5*y1;
equation
  u = 2;
  tau*der(x) = (u - x) + (u - x)^2;
  y1 = mu*x;
end Test;
```

```
//! base 0.1.0
package 'Test'
 model 'Test'
  final parameter Real 'tau' = 5.0;
  final parameter Real 'mu' = 30.0;
  Real 'u';
  Real 'x'(nominal = 1.0,
           fixed = true, start = 1.0);
  Real 'y1'(nominal = 50.0);
  Real 'y2'(nominal = 30.0);
 equation
  'y2' = 0.5 * 'y1';
  'u' = 2.0;
  5.0 * der('x') = 'u' - 'x' + ('u' - 'x') ^
      2.0;
  'y1' = 30.0 * 'x';
 end 'Test';
end 'Test';
```

```
import idaes
from pyomo.environ import *
from pyomo.core.expr import identify_variables, differentiate
from pyomo.dae import (ContinuousSet, DerivativeVar)
import numpy as np
import re

m = ConcreteModel()
m.scaling_factor = Suffix(direction=Suffix.EXPORT)
m.time = ContinuousSet(initialize = np.linspace(0,25,10+1))
dt = 25/(10+1)/3
# Writing the code for variable instantiation
m.u = Var(m.time, initialize = 2.0, within = Reals)
m.scaling_factor[m.u] = 1/max(1.0,2.0)
m.x = Var(m.time, initialize = 1.0, within = Reals)
m.scaling_factor[m.x] = 1/max(1.0,1.0)
m.y1 = Var(m.time, initialize = 30.0, within = Reals)
m.scaling_factor[m.y1] = 1/max(50.0,30.0)
m.y2 = Var(m.time, initialize = 15.0, within = Reals)
m.scaling_factor[m.y2] = 1/max(30.0,15.0)
# Instantiating the variables for the time derivatives
m.DERx = DerivativeVar(m.x, initialize = 0.4)
m.scaling_factor[m.DERx] = m.scaling_factor[m.x]*dt
# Applying time discretization through direct collocation to the problem
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m, nfe=10, ncp=3, scheme='LAGRANGE-RADAU')
timeSteps = [h for h in m.time]
# Instantiating the problem constraints
def _constr1(m,t):
return m.y2[t] == 0.5 * m.y1[t]
m.constr1 = Constraint(m.time, rule = _constr1)
m.u.fix(2.0)
def _constr2(m,t):
if t == 0:
return Constraint.Skip
return 5.0 * m.DERx[t] == m.u[t] - m.x[t] + (m.u[t] - m.x[t]) ** 2.0
m.constr2 = Constraint(m.time, rule = _constr2)
def _constr3(m,t):
return m.y1[t] == 30.0 * m.x[t]
m.constr3 = Constraint(m.time, rule = _constr3)
m.obj = Objective(expr = 1, sense = minimize)
# Code for the initial conditions of the dynamic optimization problem
def _init(m):
yield m.x[0] == 1.0
yield ConstraintList.End
m.init_conditions = ConstraintList(rule=_init)
# Constraint normalization
for constr in m.component_objects(Constraint, active=True):
varList = list(identify_variables(constr[list(constr.keys())[0]].body))
listF = []
for var in varList:
varName = re.sub(r'\[.*\]', '', str(var))
varName = "m." + varName
listF.append(abs(differentiate(constr[list(constr.keys())[0]].body, wrt=var)/m.scaling_factor[eval(
    varName)]))
m.scaling_factor[constr] = 1/max(listF)
# Solver settings and problem solution
scaled_model = TransformationFactory('core.scale_model').create_using(m)
solver = SolverFactory("ipopt")
results = solver.solve(scaled_model, tee=True)
TransformationFactory('core.scale_model').propagate_solution(scaled_model, m)
```