Math. Prog. Comp. (2018) 10:187–223
https://doi.org/10.1007/s12532-017-0127-0

FULL LENGTH PAPER

pyomo.dae: a modeling and automatic discretization framework for optimization with differential and algebraic equations

Bethany Nicholson[1] · **John D. Siirola**[2] · **Jean-Paul Watson**[2] · **Victor M. Zavala**[3] · **Lorenz T. Biegler**[1]

Received: 26 May 2016 / Accepted: 22 August 2017 / Published online: 20 December 2017
© Springer-Verlag Berlin Heidelberg and The Mathematical Programming Society 2017

Abstract We describe pyomo.dae, an open source Python-based modeling framework that enables high-level abstract specification of optimization problems with differential and algebraic equations. The pyomo.dae framework is integrated with the Pyomo open source algebraic modeling language, and is available at http://www.pyomo.org. One key feature of pyomo.dae is that it does not restrict users to standard, predefined forms of differential equations, providing a high degree of modeling flexibility and the ability to express constraints that cannot be easily specified in other modeling frameworks. Other key features of pyomo.dae are the ability to specify optimization problems with high-order differential equations and partial differential equations, defined on restricted domain types, and the ability to automatically transform high-level abstract models into finite-dimensional algebraic problems that can be solved with off-the-shelf solvers. Moreover, pyomo.dae users can leverage existing capabilities of Pyomo to embed differential equation models within stochastic and integer programming models and mathematical programs with equilibrium constraint formulations. Collectively, these features enable the exploration of new modeling concepts, discretization schemes, and the benchmarking of state-of-the-art optimization solvers.

Keywords Dynamic optimization · Mathematical modeling · Algebraic modeling language · DAE constrained optimization · PDE constrained optimization

✉ Bethany Nicholson
blnicho@sandia.gov

[1] Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA

[2] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185, USA

[3] Department of Chemical and Biological Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA

Mathematics Subject Classification 49M37 · 68N15 · 90-04 · 93A30 · 90C90

1 Introduction and motivation

Optimization problems constrained by differential and algebraic equations (DAEs) or partial differential and algebraic equations (PDAEs) are ubiquitous in engineering and science. Application domains include aerospace systems, chemical reactor systems, infrastructure (water, gas, electricity, and transportation) networks, economics and finance, robotics, and environmental engineering. Dynamic optimization problems arising in these domains are often expressed in specific canonical forms. For example, a standard form for an optimal control problem is:

$$\min \quad \int_{t_0}^{t_f} \Phi(t, x(t), y(t), u(t))dt + \Psi(t_f, x(t_f), y(t_f)) \tag{1.1a}$$

$$\text{s.t.} \quad \dot{x}(t) = f(t, x(t), y(t), u(t)), \qquad t \in [t_0, t_f] \tag{1.1b}$$

$$0 = g(t, x(t), y(t), u(t)), \qquad t \in [t_0, t_f] \tag{1.1c}$$

$$0 \leq h(t, x(t), y(t), u(t)), \qquad t \in [t_0, t_f] \tag{1.1d}$$

$$x(t_0) = \bar{x}_0 \tag{1.1e}$$

where x, y, and u are differential, algebraic, and control variables respectively. f represents the right-hand sides of explicit ordinary differential equations and g and h represent equality and inequality algebraic constraints respectively. Finally, (1.1e) represents the initial conditions for the differential variables. It is important to note that the differential equations in this standard form are assumed to be first-order and explicit. In other words, each differential equation can be written as a single derivative term set equal to a function of the variables in the model.

Dynamic optimization problems like (1.1) are challenging to solve because off-the-shelf optimization solvers do not directly handle integrals or differential equations. Therefore, these problems must be reformulated in order to use standard optimization tools. This is typically done by discretizing the continuous dynamics using an approach such as single shooting, multiple shooting, or direct transcription [7]. These methods differ in which parts of the model are discretized and the types of dynamic models they can address. This work focuses on the direct transcription approach. The direct transcription method simultaneously discretizes the entire DAE model and approximates the continuous dynamics using algebraic equations. More information on this approach is provided in Sect. 3.

The dynamic optimization solution approaches described above are often implemented manually which is tedious, time-consuming, and error-prone. Furthermore, while many dynamic optimization problems can be formulated in a standard form like (1.1), more complex and exotic (non-canonical) dynamic models including multi-point boundary conditions and PDAEs spanning multiple domains are not uncommon. Manually applying a dynamic optimization solution approach to models in non-standard forms is non-trivial and may require the combination of different techniques and associated experimentation. To avoid such complications, it is desirable to separate the

Table 1 Capabilities of existing tools for specifying and solving optimization problems with differential equations

Tool	Types	Non-canonical forms	Open source	Open language	Solution methods
ACADO [18]	ODE, DAE		X	C++	Shooting methods, collocation
APMonitor [16]	ODE, DAE	X			Collocation
GPOPSII [26]	ODE, DAE				Collocation
gPROMS [27]	ODE, DAE, PDAE	X			Shooting methods, collocation
Optimica and JModelica.org [1]	ODE, DAE		X		Shooting, collocation
SOCS [6] and SOS [5]	ODE,DAE			Fortran	Finite difference, collocation
TACO [22]	ODE, DAE		X		Shooting
Tomlab PROPT [28]	ODE, DAE	X			Collocation
pyomo.dae	ODE, DAE, PDAE	X	X	Python	Finite difference, collocation

dynamic model abstractions from the solution approach so that users can experiment with alternative approaches and assess performance in a more systematic manner.

There are many implementation approaches available to users interested in incorporating differential equations into optimization models. One widely used alternative is simply to apply a discretization manually and then express the resulting finite-dimensional representation in an algebraic modeling language. Algebraic modeling languages allow a user to represent optimization problems using a concise and natural syntax, perform limited model checking and automatic differentiation, and provide interfaces for communicating with optimization solvers. Well-known commercial algebraic modeling languages include GAMS [11] and AMPL [12]. A limitation of these tools is that they define their own proprietary syntax for representing optimization problems and are not open source. Consequently, they have limited extensibility.

In contrast to commercial alternatives, algebraic modeling tools that are embedded in high-level programming languages such as Python, Matlab, and Julia provide more flexibility to incorporate new syntax, components, and processing capabilities. Examples of such tools include FlopC++ [19], PuLP [25], Pyomo [15], and more recently JuMP [23]. While these algebraic modeling languages allow the user to formulate problems in a high-level programming language, they are currently restricted in the classes of optimization problems they can represent. Specifically, only a small subset of these tools provide syntax extensibility and processing (discretization) capabilities for differential equations. An overview of such tools and their capabilities is provided in Table 1. This table summarizes the types of differential equations each tool can express and manipulate, indicates whether the tool is open source, and indicates whether the tool represents models using a high-level programming language (as opposed to a proprietary syntax). The last column of the table lists the solution methods each tool provides to the user. As the table indicates, the new package described in

this paper – `pyomo.dae` – allows for representation of a broad range of differential and algebraic equation components in optimization models, is open source and based on a widely used high-level programming language (Python), and provides a range of discretization capabilities for solution of specified models.

Most of the existing tools that can express and solve optimization models with differential equations require users to formulate models using a proprietary language. Notable exceptions are ACADO and SOCS/SOS, which can be respectively used directly from C++ and Fortran. ACADO is open source and provides both shooting and direct collocation methods for solving specified problems. However, ACADO cannot represent optimal control problems in non-canonical forms. While these tools provide tailored solution algorithms for models with specific classes of differential equations, they are unable to represent and solve models which deviate even slightly from those specific classes. APMonitor offers additional flexibility in expressing differential equations in non-canonical forms and high-index DAE systems, and is freely accessible via the web. However, APMonitor is not open source and uses a proprietary syntax.

Of the tools surveyed here, gPROMS is the only package that provides complete flexibility in the types of differential equations that can be modeled, including support for PDAEs and non-canonical forms. However, gPROMS is a commercial software package and cannot be extended by the user. Tools such as Optimica, TACO, and Tomlab's PROPT are extensions for the existing programming languages and software packages Modelica, AMPL, and Matlab, respectively. None of these tools include support for differential equations in non-standard forms and the extensions are all based on a proprietary syntax.

In contrast to the tools described above, `pyomo.dae` is a flexible and extensible tool for representing systems of ODEs, DAEs, and PDAEs on certain domains and provides capabilities to automatically applying numerical discretization techniques for converting differential equations to algebraic equations. Using `pyomo.dae`, a user may represent ordinary and partial differential equations of arbitrary order, of arbitrary dimension, and in any form – including complex mixed partial derivatives. The `pyomo.dae` package is also capable of expressing and solving optimal control problems in non-canonical forms. We have based `pyomo.dae` on a widely used open source algebraic modeling language, Pyomo, which in turn is written in the high-level programming language Python.

The remainder of this paper is organized as follows. In Sect. 2, we provide a brief overview of Pyomo and describe the new modeling constructs available in `pyomo.dae`. Then, in Sect. 3, we review the automatic discretization schemes that are provided in `pyomo.dae` to transform dynamic models to algebraic approximations, and describe their use within Pyomo and Python. In Sect. 4, we detail how `pyomo.dae` can be extended to support custom discretization schemes. We discuss the expression and solution of several illustrative examples using `pyomo.dae` in Sect. 5. Finally, we conclude in Sect. 6 with a summary of `pyomo.dae` capabilities, our contributions to algebraic modeling languages for optimization, and directions for future research.

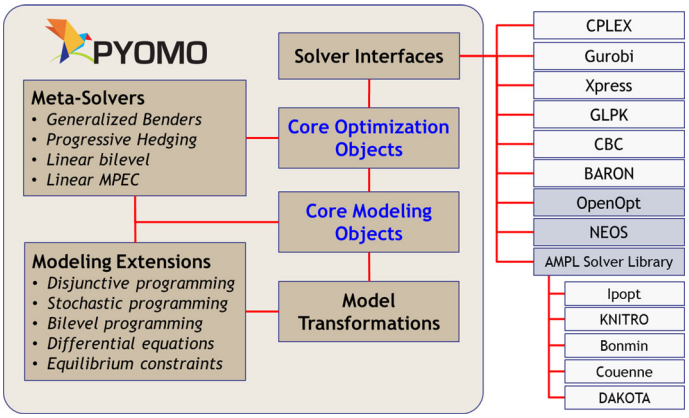

Fig. 1 Summary of Pyomo features

2 Modeling differential equations using `pyomo.dae`

In Sect. 2.1 we provide a brief overview of the Pyomo algebraic modeling language, surveying its core capabilities and justifying our choice of Pyomo as the basis for our DAE and PDAE modeling and transformation capabilities. In Sect. 2.2, we then describe our new modeling components, which integrate directly with and extend the Pyomo library.

2.1 Pyomo: background

Pyomo is an open source algebraic modeling language written in Python [14,15]. Since its introduction, Pyomo has undergone major restructuring and extension, such that Pyomo is now stable, flexible, and widely used. An overview of the current features of Pyomo is shown in Fig. 1. Pyomo supports a wide range of problem types including Linear Programming (LP), Mixed-Integer Programming (MIP), Nonlinear Programming (NLP), and Mixed-Integer Nonlinear Programming (MINLP). Pyomo also provides interfaces to a variety of optimization solvers and provides automatic differentiation (AD) for NLP problems via the open source AMPL Solver Library (ASL).

One of Pyomo's main advantages over other algebraic modeling languages is that it is written in a high-level programming language, Python. Consequently, a user does not have to learn a specialized modeling language in order to formulate and solve optimization problems; a basic understanding of Python is all that is required. Models are represented using Python objects and can be formulated and manipulated in sophisticated ways using simple scripts. Furthermore, Pyomo users have access to a large collection of other Python packages which include tools for plotting, numerical and statistical analysis, and input/output. These capabilities enable the development of novel algorithms, complicated model formulations, and general model transformations. All of these features make Pyomo a promising platform for implementing extensions for problem classes such as dynamic optimization.

2.2 Modeling components for expressing DAEs and PDAEs

One of the main constructs in any algebraic modeling language is the notion of an indexing set. Indexing sets allow users to specify related collections of parameters, variables, and constraints with a concise notation. In standard algebraic modeling languages, indexing sets are assumed to be discrete. This assumption makes it challenging to represent optimization problems with continuous domains and dynamics without first converting the model to a discretized approximation, causing loss of information and transparency with respect to the precise nature of the model. To address these shortcomings, the `pyomo.dae` library allows direct specification of continuous domains and arbitrary derivatives, through the introduction of two new Pyomo modeling components. We now describe these components and provide a brief overview of how they can be used to express a diverse range of algebraic optimization models with differential equations. More details about these components can be found in the online documentation available at http://www.pyomo.org.

2.2.1 The `ContinuousSet` *component*

Analogous to the discrete set construct available in most algebraic modeling languages, we now introduce the notion of a *continuous set*. The `ContinuousSet` construct allows users to represent continuous bounded domains in an algebraic optimization model. Such domains can be interpreted as "spatial" or "time" domains, but such interpretations are not strictly necessary. Rather, a continuous domain can represent any domain of a function, e.g., a parameter domain.

The `ContinuousSet` component is similar in structure to that of Pyomo's `Set` component and can be used to index variables and constraints. Variables and constraints can be defined over an arbitrary number of continuous domains. For instance, the expression

$$w(\ell, x, t): \ \ell \in \mathcal{L} := \{0, 1, \ldots, L\}, \ t \in \mathcal{T} := [0, T], \ x \in \mathcal{X} := [0, X] \qquad (2.1)$$

defines a variable w with indices ℓ spanning the discrete set $\mathcal{L}$, such that each $w(\ell, \cdot, \cdot)$ is defined over the continuous rectangular domain $\mathcal{T} \times \mathcal{X} = [0, T] \times [0, X]$. By default, a `ContinuousSet` will be treated as a closed set. The code to declare variable (2.1) using `pyomo.dae` is shown below (assuming L, T, and X are previously defined constants).

Listing 1 Declaring a variable over discrete and continuous domains

```
1 m = ConcreteModel()
2 m.l = RangeSet(0, L)
3 m.x = ContinuousSet(bounds=(0, X))
4 m.t = ContinuousSet(bounds=(0, T))
5 m.w = Var(m.l, m.x, m.t)
```

A `ContinuousSet` must be initialized with two numeric values that specify the lower and upper bounds of the continuous domain being represented. A user may also specify additional points in the domain that may be communicated to a

transformation method and/or solver. Such data can be used, for instance, to ensure that a discretization mesh includes a subset of points, which in turn may be required to specify path constraints or ensure consistency with experimental data. The following code fragment illustrates this capability:

Listing 2 Declaring continuous domains using `ContinuousSet` components

```
# declare by providing bounds
model.t = ContinuousSet(bounds=(0, 5))
# declare specific points in domain
model.x = ContinuousSet(initialize=[0, 1, 2.5, 3.3, 5])
```

A user may also declare a `ContinuousSet` as part of a purely *abstract* Pyomo model, mirroring AMPL's paradigm of explicitly separating model from data. Generally, most valid methods to declare and initialize a Pyomo `Set` can be used to declare and initialize a `ContinuousSet`.

When an algebraic optimization model with DAEs and/or PDAEs is manually discretized, the indexing set for a continuous variable typically takes the form of an arbitrary set of integers and the user is responsible for converting each index to the desired temporal or spatial quantity. This conversion process is one of the most common places where modeling errors occur, particularly when a more complex discretization scheme is used, e.g., collocation on finite elements. The conversion can be particularly challenging in the case of unequally spaced finite elements. Beyond discretization for a solver, accuracy of this conversion is crucial in visualization and analysis of results. One of the main design strengths of the `ContinuousSet` component in `pyomo.dae` is that this manual conversion step has been eliminated and domain scaling is done directly within the set. In other words, the discrete points in a `ContinuousSet` directly capture actual floating point values from the bounded continuous domain being represented.

The `ContinuousSet` component can also be used to store information from a transformation and/or solver. For instance, if the set is discretized using a collocation method over finite elements, one can interrogate the `ContinuousSet` component to obtain a list of these discretization points in order to implement customized model initializations or constraints. Examples of these capabilities are shown below:

Listing 3 Interrogating a `ContinuousSet` component to get information about the applied discretization scheme

```
m = ConcreteModel()
m.t = ContinuousSet(bounds=(0, 10))
m.v = Var(m.t)

# Discretize model using Radau Collocation over finite elements
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m, wrt=m.t, nfe=100, ncp=3)

# Obtain a Python dictionary containing details about the discretization
# scheme that has been applied to the Continuous Set
disc_info = m.t.get_discretization_info()

# Apply a constraint at finite element boundary points only
def _con(m, i):
    return m.v[i] >= 0
```

```
m.con = Constraint(m.t.get_finite_elements(), rule=_con)

# Obtain the upper and lower finite element boundaries for
# internal collocation points
for i in m.t:
    upper = m.t.get_upper_element_boundary(i)
    lower = m.t.get_lower_element_boundary(i)
```

The `ContinousSet` component provides users with a highly flexible mechanism to impose constraints and objectives in non-standard forms. In particular, we do not impose restrictions on the number of independent dimensions of a variable or constraint, nor do we constrain the order of appearance of continuous and regular sets in an indexed variable or constraint. For example, we enable specifications of the following form:

$$w(\ell, x, y, z, t) : \ \ell \in \mathcal{L}, \ \ x \in \mathcal{X}, \ y \in \mathcal{Y}, \ z \in \mathcal{Z}, \ t \in \mathcal{T}. \tag{2.2}$$

Analogous to the definition of a subset of a discrete set $\bar{\mathcal{L}} \subset \mathcal{L}$ is the definition of a subset of a continuous set $\bar{\mathcal{T}} \subset \mathcal{T}$. Here, the subset $\bar{\mathcal{T}}$ can be either discrete (with elements in $[0, T]$) or continuous. Constraints involving variables defined over a continuous set may be defined over a subset of a continuous set. For instance, if we define $\mathcal{X} = [0, 2]$ and $\mathcal{T} = [1, 10]$, the following constraints are conceptually valid:

$$\begin{aligned} &h(w(\ell, x, t)) = 0, \ x \in [0, 1], \ t \in [5, 10] \\ &h(w(\ell, x, t)) = 0, \ x \in \{0, 0.1, 0.5, 1\}, \ t \in \mathcal{T} \\ &h(w(\ell, x, t)) = 0, \ x \in \{0, 0.1, 0.5, 1\}, \ t \in \{2, 3, 4\}. \end{aligned}$$

Currently, `pyomo.dae` does not allow a `ContinuousSet` component to be defined as a subset of another `ContinuousSet`. This extension is on-going work. Instead, constraints over continuous subsets can currently be implemented using constraint skipping. Examples of declaring constraints over continuous and discrete subsets are shown below:

Listing 4 Declaring constraints over a subset of a continuous set

```
m.xsubset = Set(initialize=[0, 0.1, 0.5, 1])
m.tsubset = Set(initialize=[2, 3, 4])

m.x = ContinuousSet(bounds=(0, 2), initialize=m.xsubset)
m.t = ContinuousSet(bounds=(1, 10), initialize=m.tsubset)

def _con1(m, l, x, t):
    if x <= 1 and t >= 5:
        return m.h[l, x, t] == 0
    else:
        return Constraint.Skip
m.con1 = Constraint(m.l, m.x, m.t, rule=_con1)

def _con2(m, l, x, t):
    return m.h[l, x, t] == 0
```

```
m.con2 = Constraint(m.l, m.xsubset, m.t, rule=_con2)

def _con3(m, l, x, t):
    return m.h[l, x, t] == 0
m.con3 = Constraint(m.l, m.xsubset, m.tsubset, rule=_con3)
```

The ability to impose constraints at discrete points allows the user to specify path and point constraints. This in turn allows the imposition of multi-point boundary conditions including periodicity and delay constraints. Recirculation (cyclic) constraints are typical in chemical reactors, periodicity constraints are typical in model predictive control, and delay constraints are typical in signal and transportation networks. Specifications of these types of constraints take the following form:

$$
\begin{aligned}
h(w_1(x,0)) &= f(w_1(x,T)), \quad x \in \mathcal{X} \\
h(w_1(x,T)) &= f(w_1(x,T-\delta)), \quad x \in \mathcal{X} \\
h(w_1(X,0)) &= f(w_1(X/2,T/2))
\end{aligned}
$$

where $h(\cdot)$ and $f(\cdot)$ are arbitrary functions. The constraints expressed above can be implemented as follows assuming X and T are previously defined constants:

Listing 5 Declaring multi-point boundary conditions

```
m.delta = Param(initialize=5)

m.x = ContinuousSet(bounds=(0, X), initialize=[X/2])
m.t = ContinuousSet(bounds=(0, T), initialize=[T/2, T-m.delta])

m.h = Expression(m.x, m.t, rule=_arbitrary_h_expression)
m.f = Expression(m.x, m.t, rule=_arbitrary_f_expression)

def _con1(m, x):
    return m.h[x, 0] == m.f[x, T]
m.con1 = Constraint(m.x, rule=_con1)

def _con2(m, x):
    return m.h[x, T] == m.f[x, T-delta]
m.con2 = Constraint(m.x, rule=_con2)

def _con3(m):
    return m.h[X, 0] == m.f[X/2, T/2]
m.con3 = Constraint(rule=_con3)
```

We also allow the user to impose constraints between variables and across domains. For instance, the constraint

$$
h(w(\ell,x,T)) = f(w(\ell+1,x,0)), \quad x \in \mathcal{X}
$$

can be implemented as:

Listing 6 Declaring constraints across domains

```
m.l = RangeSet(1, 10)
m.x = ContinuousSet(bounds=(0, X))
m.t = ContinuousSet(bounds=(0, T))

def _con(m, l, x):
    if l == 10:
        return Constraint.Skip
    else:
        return m.h[l, x, T] == m.f[l+1, x, 0]
m.con = Constraint(m.l, m.x, rule=_con)
```

This feature enables the user to couple physical elements described by different sets of differential equations. For instance, multiple domains arise in the modeling and control of lithium-ion batteries, fuel cells, buildings, and gas networks. Coupling across domains is also common in multi-stage optimal control where the horizon is lifted into stages.

The above example illustrates that `pyomo.dae` provides an intuitive, straightforward syntax for formulating and linking dynamic algebraic optimization models over several domains. However, we note that we do not make any claims concerning our ability to *solve* arbitrary dynamic models over continuous domains. Rather, our tool simply allows the user to formulate their problem in a direct, high-level manner.

2.2.2 *The* `DerivativeVar` *component*

The `DerivativeVar` component is used to declare a derivative of a standard Pyomo `Var`. A variable may only be differentiated with respect to a `ContinuousSet`. Furthermore, the `ContinuousSet` must be included as an explicit indexing set of the `Var`. The indexing sets of a `DerivativeVar` are obtained directly from the `Var` it is differentiating. We also allow for specification of derivatives of arbitrary order. In the case of the variable $w(\ell, x, t)$ defined in (2.1), the derivatives

$$\frac{\partial^{\alpha} w(\ell, x, t)}{\partial t^{\alpha}} \tag{2.3a}$$

$$\frac{\partial^{\beta} w(\ell, x, t)}{\partial x^{\beta}} \tag{2.3b}$$

$$\frac{\partial^{\alpha}\partial^{\beta} w(\ell, x, t)}{\partial t^{\alpha}\partial x^{\beta}} \tag{2.3c}$$

can be imposed for $\alpha, \beta \in \{1, 2, \ldots\}$. The following code fragment illustrates the declaration of first-order, second-order, and mixed-order derivatives. The variable being differentiated is supplied as the only positional argument to the `DerivativeVar` constructor and the type of derivative is specified using the "wrt" keyword argument (the more verbose "withrespectto" can also be used).

Listing 7 Declaring derivatives

```
m.dwdt = DerivativeVar(m.w, wrt=m.t)
m.dwdx2 = DerivativeVar(m.w, wrt=(m.x, m.x))
m.dwdxt = DerivativeVar(m.w, wrt=(m.x, m.t))
```

We assume that a variable defined over a continuous set is sufficiently smooth, such that any defined derivatives exist. The user is responsible for ensuring that these requirements are fulfilled. As is the case with the `ContinuousSet` component, the specification syntax is agnostic to the specifics of any transformation and/or solution method that may be employed.

Derivatives can be referenced in the body of any constraint. This is an important feature, as we do not impose any predefined structure on differential models. This enables the expression of DAEs with complex mass matrices or boundary conditions. For instance, consider the following constraints:

$$\frac{\partial w(\ell, x, t)}{\partial x} = f(w(\ell, x, t)),\ \ell \in \mathcal{L},\ x \in \mathcal{X},\ t \in \mathcal{T}$$
$$0 = \frac{\partial w(\ell, x, t)}{\partial x} g\left(w(\ell, x, t)\right),\ \ell \in \mathcal{L},\ x \in \mathcal{X},\ t \in \mathcal{T}.$$

The code implementing such constraints has the form:

Listing 8 Declaring constraints with derivatives

```
m.f = Expression(m.l, m.x, m.t, rule=_arbitrary_f_expression)
m.g = Expression(m.l, m.x, m.t, rule=_arbitrary_g_expression)

def _con1(m, l, x, t):
    return m.dwdx[l, x, t] == m.f[l, x, t]
m.con1 = Constraint(m.l, m.x, m.t, rule=_con1)

def _con2(m, l, x, t):
    return 0 == m.dwdx[l, x, t] * m.g[l, x, t]
m.con2 = Constraint(m.l, m.x, m.t, rule=_con2)
```

The code fragment below provides a more explicit and complete example of how to specify `DerivativeVar` components:

Listing 9 Declaring derivatives using `DerivativeVar` components

```
model = ConcreteModel()
model.s = Set(initialize=['a', 'b'])
model.t = ContinuousSet(bounds=(0, 5))
model.l = ContinuousSet(bounds=(-10, 10))
model.x = Var(model.t)
model.y = Var(model.s, model.t)
model.z = Var(model.t, model.l)

# Declare first derivative of model.x with respect to model.t
model.dxdt = DerivativeVar(model.x, withrespectto=model.t)

# Declare second derivative of model.y with respect to model.t
# Note that this DerivativeVar will be indexed by both model.s and model.t
model.dydt2 = DerivativeVar(model.y, wrt=(model.t, model.t))
```

```
15
16 # Declare partial derivative of model.z with respect to model.l
17 # Note that this DerivativeVar will be indexed by both model.t and model.l
18 model.dzdl = DerivativeVar(model.z, wrt=model.l, initialize=0)
19
20 # Declare mixed second order partial derivative of model.z with respect
21 # to model.t and model.l and set bounds
22 model.dz2 = DerivativeVar(model.z, wrt=(model.t, model.l), bounds=(-10,10))
```

The design of the `DerivativeVar` component diverges from that of core Pyomo components in its use of positional arguments. Typically, positional arguments in Pyomo are used to specify indexing sets of a particular component. However, for `DerivativeVar` components, a variable is supplied as a positional argument. This design choice was intended to codify the notion of a derivative being an operation on a variable rather than a distinct object. We note that the "initialize" keyword argument shown above will initialize the value of a derivative, rather than specifying an initial condition or boundary constraint.

After the derivatives in a model have been declared using `DerivativeVar` components, differential equations are declared as standard Pyomo constraints and are not required to have any particular form. The following code fragment illustrates how one might declare an ordinary or partial differential equation using one or more of the derivatives defined in the previous code fragment:

Listing 10 Declaring differential equations

```
1  # An ordinary differential equation
2  def _ode_rule(m, t):
3      if t == m.t.first():
4          return Constraint.Skip
5      return m.dxdt[t] == m.x[t]**2
6  model.ode = Constraint(model.t, rule=_ode_rule)
7
8  # A partial differential equation
9  def _pde_rule(m, t, l):
10     if t == m.t.first() or l == m.l.first() or l == m.l.last():
11         return Constraint.Skip
12     return m.dzdl[t, l] == m.dz2[t, l]
13 model.pde = Constraint(model.t, model.l, rule=_pde_rule)
```

A modeler may not want to define a differential equation at one or both boundaries of a continuous domain. This choice can be specified explicitly in the `Constraint` declaration using the "Constraint.Skip" return value, as shown above. By default, a constraint declared over a `ContinuousSet` will be applied at every discretization point contained in the set. One can also think of this as the distinction between applying a constraint over an open or closed set. Finally, for the case of boundary conditions of PDAEs, we note that the user is responsible for providing consistent expressions and `pyomo.dae` provides syntax to communicate such constraints (e.g., derivative objects and access to specific points).

3 Discretization transformations

Before a Pyomo model with `ContinuousSet` and `DerivativeVar` components can be processed by a finite dimensional solver it must first be discretized. This transformation process converts a dynamic optimization model to a purely algebraic optimization model using a simultaneous discretization approach. Specifically, the continuous domains in the dynamic optimization model are discretized and any derivatives are approximated using algebraic equations defined at the discretization points. Two families of discretization schemes are currently implemented in `pyomo.dae`: finite differences and collocation.

By separating models from discretization schemes we enable users to easily experiment with alternative discretization schemes, and to identify the one that works best with their particular problem. This separation also provides users the freedom to combine discretization schemes in non-standard ways, e.g., using collocation to discretize a spatial domain and finite difference method to discretize in time or vice versa.

One of the key differences between `pyomo.dae` and similar tools is that the discretized model is returned to the user after each transformation is applied. This design feature allows the user to interrogate the discretized model and examine the discretization equations that were added to the model. Additionally, users can further modify the model following discretization.

Applying one of the discretization schemes available in `pyomo.dae` to a differential equation is analogous to approximating the solution of that differential equation using a numerical method. Numerical methods vary in terms of accuracy and the type of problems to which they can be applied. While we provide a brief overview of the methods implemented in `pyomo.dae` below, a detailed description of these methods is outside the present scope. For more information on these techniques and details of their applicability, we defer to [3,4,9,10].

3.1 Finite difference transformation

Finite difference methods are the simplest discretization schemes to apply manually, and approximate the derivative at a particular point using a difference equation. The `dae.finite_difference` transformation in `pyomo.dae` includes implementations of several finite difference methods. The most commonly-used finite difference method is the backward difference method, also referred to as implicit or backward Euler. In our implementation, a discretization is applied to a particular continuous domain and propagated to each derivative and constraint over that domain. For instance, consider the following derivative and associated constraint:

$$\left(\frac{dx(t)}{dt}, f(x(t), u(t))\right) = 0, \quad t \in [0, T]. \tag{3.1}$$

After applying the backward difference method to domain t, the resulting derivative and constraint pair is

$$\left.\frac{dx}{dt}\right|_{t_{k+1}} = \frac{x_{k+1} - x_k}{h}, \quad k = 0, \ldots, N-1 \tag{3.2}$$

$$g\left(\left.\frac{dx}{dt}\right|_{t_{k+1}}, f(x_{k+1}, u_{k+1})\right) = 0, \quad k = 0, \ldots, N-1 \tag{3.3}$$

where $x_k = x(t_k), t_k = kh$, and h is the step size between discretization points or the size of each finite element. We note that the discretization scheme is applied to all constraints and variables of the model in a given continuous domain. Higher order derivative terms are approximated using recursive schemes. When a `dae.finite_difference` transformation is applied to a Pyomo model, equations such as (3.2) are automatically generated and added to the resulting discretized Pyomo model as equality constraints. The following Python script applies the backward difference method to a Pyomo model. The code also illustrates how to add a constraint to a model after discretization.

Listing 11 Applying a finite difference discretization to a Pyomo model

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample import model

# Discretize model using Backward Difference method
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(model, nfe=20, wrt=model.time, scheme='BACKWARD')

# Add a constraint to the discretized model
def _sum_limit(m):
    return sum(m.x1[i] for i in m.time) <= 50
model.con_sum_limit = Constraint(rule=_sum_limit)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

There are several options available to a `dae.finite_difference` transformation, which can be specified as keyword arguments to the '`apply_to`' function of the transformation object. These keywords are as follows:

'`nfe`': Specifies the desired number of finite element points to be included in the discretization. The default value is 10.

'`wrt`': Specifies the `ContinuousSet` for which the transformation should be applied. If this keyword argument is not specified then the same scheme will be applied to all continuous sets in the model.

'`scheme`': Specifies which finite difference method to apply. Alternatives are '`BACKWARD`', '`CENTRAL`', or '`FORWARD`'. The default scheme is the backward difference method.

If the existing number of finite element points in a `ContinuousSet` is less than the desired number, new discretization points will be automatically added to the set. If a

user specifies a desired number of finite element points which is less than the number of points already included in the `ContinuousSet` then the transformation will ignore the desired number and proceed with the larger set of points. Discretization points will never be removed from a `ContinuousSet` during the discretization transformation.

3.2 Collocation transformation

Orthogonal collocation (or simply "collocation") over finite elements is another popular numerical method for discretization of differential equations. Collocation works by first breaking a continuous domain into $N-1$ finite elements. Then over each finite element i the profile of the differential variable $x(t)$ is approximated using a polynomial of order $K+1$. This polynomial is defined using K collocation points, which act as additional discretization points within each finite element i. Continuity is enforced at the finite element boundaries for the differential variables x. For example, the discretization equations for a constraint of the form (3.1) are given by

$$\left.\frac{dx}{dt}\right|_{t_{ij}} = \frac{1}{h_i}\sum_{j=0}^{K} x_{ij}\frac{d\ell_j(\tau_k)}{d\tau}, \quad k=1,\ldots,K,\ i=1,\ldots,N-1 \tag{3.4a}$$

$$0 = g\left(\left.\frac{dx}{dt}\right|_{t_{ij}}, f(x_{ik},u_{ik})\right), \quad k=1,\ldots,K,\ i=1,\ldots,N-1 \tag{3.4b}$$

$$x_{i+1,0} = \sum_{j=0}^{K}\ell_j(1)x_{ij}, \quad i=1,\ldots,N-1 \tag{3.4c}$$

where $t_{ij} = t_{i-1} + \tau_j h_i, x(t_{ij}) = x_{ij}$. Further, we note that the solution $x(t)$ is interpolated as follows:

$$x(t) = \sum_{j=0}^{K}\ell_j(\tau)x_{ij}, \quad t\in[t_{i-1},t_i], \quad \tau\in[0,1] \tag{3.5a}$$

$$\ell_j(\tau) = \prod_{k=0,\neq j}^{K}\frac{(\tau-\tau_k)}{(\tau_j-\tau_k)}. \tag{3.5b}$$

The advantage of using a collocation method over a finite difference method is that collocation results in significantly more accurate algebraic approximations. The drawback of collocation is that it is much harder to implement and debug manually, relative to finite difference methods. Furthermore, implementing the collocation discretization equations on higher-order derivatives, partial derivatives, or differential equations that are not in a standard form is non-trivial.

There are many variations of collocation methods, differing primarily in the functional representation of the state profile over each finite element and how the collocation points are defined. The current version of `pyomo.dae` includes two types of collocation methods, both using Lagrange polynomials to represent the state profiles

but differing in the choice of collocation points τ_k. One variant uses shifted Gauss-Radau roots and the other one uses shifted Gauss-Legendre roots. The term "shifted" refers to the fact that the collocation points τ are defined in the domain $\tau \in [0, 1]$ rather than $\tau \in [-1, 1]$. For more information on orthogonal collocation we refer the reader to Chapter 10 of [7].

The following Python script applies the collocation method with Lagrange polynomials and Gauss-Radau roots to a Pyomo model, and additionally illustrates how to add an objective function to the resulting discretized model.

Listing 12 Applying a collocation discretization to a Pyomo model

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample2 import model

# Discretize model using Radau Collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(model, nfe=20, ncp=6, scheme='LAGRANGE-RADAU')

# Add objective function after model has been discretized
def obj_rule(m):
    return sum((m.x[i]-m.x_ref)**2 for i in m.time)
model.obj = Objective(rule=obj_rule)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

The collocation discretization transformation in `pyomo.dae` allows the user to specify the number of finite elements, '`nfe`', independently of the number of collocation points per finite element, '`ncp`'. The discretization options available to a `dae.collocation` transformation are the same as those described above for the `dae.finite_difference` transformation, with the following additions:

'`scheme`': Specifies the desired collocation scheme, either '`LAGRANGE-RADAU`' or '`LAGRANGE-LEGENDRE`'. The default is '`LAGRANGE-RADAU`'.

'`ncp`': Specifies the number of collocation points within each finite element. The default value is 3.

We note that any points that exist in a `ContinuousSet` before a transformation is applied will be used as finite element boundaries and not as collocation points. The locations of the collocation points cannot be specified by the user. Rather, they must be generated by the transformation.

3.3 Applying multiple transformations

Discretization transformations can be applied independently to each `ContinuousSet` in a Pyomo model, providing great flexibility to users. For example, the same scheme can be applied with different resolutions in different domains, as follows:

Listing 13 Applying different discretization resolutions to different continuous domains

```
discretizer = TransformationFactory('dae.finite_difference')
discretize.apply_to(model, wrt=model.t1, nfe=10)
discretize.apply_to(model, wrt=model.t2, nfe=100)
```

This feature can be used, for instance, when discretizing different pipelines in a network that exhibit different dynamic behavior or discretizing different stages in an optimal control model with different resolutions.

Different schemes can also be applied to different domains. For example, we can apply a forward difference method to one `ContinuousSet` and the central finite difference method to another `ContinuousSet`:

Listing 14 Applying different finite difference transformations to different continuous domains

```
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(model, wrt=model.t1, scheme='FORWARD')
discretizer.apply_to(model, wrt=model.t2, scheme='CENTRAL')
```

Furthermore, users may combine finite difference and collocation discretizations, e.g., as follows:

Listing 15 Combining finite difference and collocation schemes

```
discretizer_fe = TransformationFactory('dae.finite_difference')
discretizer_fe.apply_to(model, wrt=model.t1, nfe=10)
discretizer_col = TransformationFactory('dae.collocation')
discretizer_col.apply_to(model, wrt=model.t2, nfe=10, ncp=5)
```

To apply the same discretization to all `ContinuousSet` components in a model, users simply specify a single discretization transformation without the '`wrt`' keyword argument. This approach will apply the selected scheme to all `ContinuousSet` components in the model that have not already been discretized.

4 Package implementation and extensibility

One of the main implementation goals for automatic discretization in `pyomo.dae` is extensibility, as there are a plethora of discretization schemes documented in the literature and some specialized schemes might be required in certain applications. As part of our implementation we have developed a general transformation framework along with certain utility functions so that advanced users may easily implement their own custom discretization schemes while reusing much of the `pyomo.dae` code. The transformation framework consists of the following steps:

1. Specify Discretization Options
2. Discretize the Continuous Sets
3. Update Model Components
4. Add Discretization Equations
5. Return Discretized Model

If a user would like to create a custom finite difference scheme then they only have to re-implement step (4) in the framework. The discretization equations for a

particular scheme have been isolated from of the rest of the code for implementing the transformation. For example, the specific function for the forward finite difference method is:

Listing 16 `pyomo.dae` implementation of the forward finite difference scheme

```
def _forward_transform(v,s):
    """
    Applies the Forward Difference formula of order O(h) for first derivatives
    """
    def _fwd_fun(i):
        tmp = sorted(s)
        idx = tmp.index(i)
        return 1/(tmp[idx+1]-tmp[idx])*(v(tmp[idx+1])-v(tmp[idx]))
    return _fwd_fun
```

In this function, '`v`' represents the continuous variable or function that the method is being applied to while '`s`' represents the set of discrete points in the continuous domain. In order to implement a custom finite difference method, a user would have to duplicate the above function and just replace the equation next to the first return statement with their method. After implementing a custom finite difference method using the above function template, the only other change that must be made is to add the custom method to the '`all_schemes`' dictionary in the `Finite_Difference_Transformation` class.

In the case of a custom collocation method, changes will be made in steps (2) and (4) of the transformation framework. The code below shows the function specific to a collocation scheme with Lagrange interpolation polynomials and Radau collocation points.

Listing 17 `pyomo.dae` implementation of collocation using Lagrange polynomials and Radau roots

```
def _lagrange_radau_transform(v,s):
    ncp = s.get_discretization_info()['ncp']
    adot = s.get_discretization_info()['adot']
    def _fun(i):
        tmp = sorted(s)
        idx = tmp.index(i)
        if idx == 0: # Don't apply this equation at initial point
            raise IndexError("list index out of range")
        low = s.get_lower_element_boundary(i)
        lowidx = tmp.index(low)
        return sum(v(tmp[lowidx+j])*adot[j][idx-lowidx] \
                   *(1.0/(tmp[lowidx+ncp]-tmp[lowidx])) for j in range(ncp+1))
    return _fun
```

In addition to implementing the discretization equations, the user would have to ensure that the desired collocation points are added to the `ContinuousSet` being discretized. The collocation transformation in `pyomo.dae` calculates Radau or Legendre collocation points following chapter 10 in [7] if the user has the Python package Numpy installed, otherwise it uses precomputed values (for up to 10^{th}-order polynomials).

So far we have shown that `pyomo.dae` is extensible in terms of implementing custom discretization schemes but it is also extensible in a more general sense. The modeling abstraction introduced here allows for general implementations of any model transformation or operation typically applied to optimal control problems. For

instance, the control profiles can be constrained to follow predefined functions. To illustrate this, we extended the dae.collocation transformation with a function that forces a control variable to have a certain profile; for example, piecewise constant or piecewise linear. This is useful, for instance, in optimal control problems where the control variable is often restricted to be constant over each finite element while the other variables are not. This function can be implemented by reducing the number of free collocation points for a particular variable. The reduce_collocation_points() function is specified using the following keywords:

'var': The variable being restricted to fewer collocation points
'contset': The continuous set indexing the specified 'var' that was previously discretized.
'ncp': The new number of collocation points. Must be at least 1 and less than the number of collocation points used to discretize the 'contset'.

The function may only be applied to a model after a collocation discretization transformation has been called to discretize the specified ContinousSet. The function works by adding constraints to the discretized model which force any extra, undesired collocation points to be interpolated from the others. These constraints have the following form over each finite element:

$$u_i = \sum_{j=1}^{\bar{K}} \gamma_{ij} u_j, \; i = \bar{K}+1, \ldots, K \tag{4.1}$$

where K is the original number of collocation points and $\bar{K}$ is the reduced number of points. A sample implementation is shown below.

Listing 18 Using the reduce_collocation_points() function

```
# Discretize model using Radau Collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(model, wrt=model.time, nfe=20, ncp=6)

# Control variable u made constant over each finite element
discretizer.reduce_collocation_points(var=model.u,contset=model.time,ncp=1)
```

We also note that pyomo.dae does not distinguish between state and control variables. This means that the reduce_collocation_points() function can be applied just as easily to impose a path constraint on a continuous state variable.

The modular nature of Pyomo also allows for extensions in terms of solvers for dynamic optimization. For example, a typical workflow with the current implementation of pyomo.dae is shown in Fig. 2. A straightforward extension would be to replace the 'Write .nl File' and 'AMPL Solver Library' blocks with another tool for automatic differentiation such as CasADi [2]. Or the workflow could be extended to output the dynamic model, before applying a discretization scheme, and then solving the problem using a shooting method. Finally, there are several parallel solvers available for dynamic optimization problems that rely on the discretized model being decomposed into subproblems at the finite element boundaries. These methods solve each subproblem independently and then apply one of several methods to force

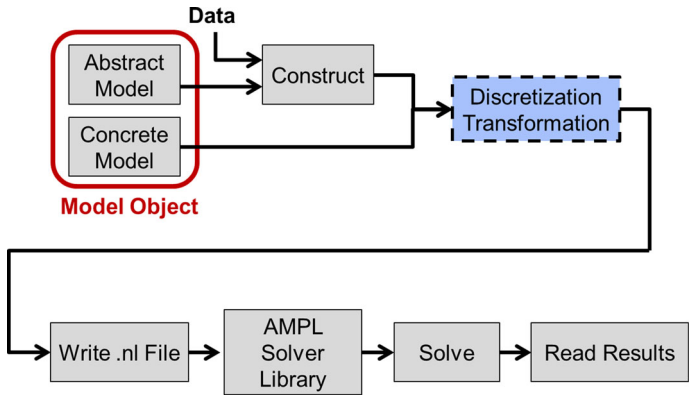

Fig. 2 Pyomo workflow including discretization transformations

continuity at the boundaries of consecutive subproblems [13,32,34]. A general implementation of this decomposition approach could easily be added to `pyomo.dae`.

5 Illustrative applications

We now provide application examples that illustrate the use and flexibility of the modeling and transformation capabilities provided by `pyomo.dae`. We begin with a detailed presentation of a simple heat transfer problem. We then consider a small optimal control example, which illustrates a common application for dynamic optimization. Next, we examine a medium-scale parameter estimation problem for a dynamic disease transmission model, which demonstrates how to integrate time-dependent data into a optimization model using `pyomo.dae`. We conclude with a more complex example involving stochastic optimal control of natural gas networks, which (1) includes all key modeling and transformation elements found in `pyomo.dae` and (2) demonstrates the scalability of `pyomo.dae`. Reported results use Pyomo version 4.3.11388, Python version 2.7.6, and Ipopt version 3.12.4 on a desktop computer running Red Hat Enterprise Linux 6.7. We note that the syntax used in these examples is compatible with more recent Pyomo releases including Pyomo 5.1. Code for these examples can be accessed at https://github.com/Pyomo/pyomo/tree/master/examples/dae.

5.1 Heat transfer

Our first application example is taken from [24] and demonstrates the solution of a one-dimensional heat equation. The equation, initial condition, and boundary conditions are as follows:

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{5.1a}$$

$$u(x, 0) = \sin(\pi x) \tag{5.1b}$$

$$u(0, t) = 0 \tag{5.1c}$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0 \tag{5.1d}$$

$$x \in [0, 1], \ t \in [0, T]. \tag{5.1e}$$

This model has a number of interesting features, including first and second order partial derivatives, boundary conditions involving both the variable and its partial derivative, and nonlinearity induced by a trigonometric function. In Listing 19, we present a Pyomo model that implements a manual discretization, without `pyomo.dae`. We discretized this model in space using the backward difference method and in time using orthogonal collocation with Radau points. The first line in the model imports the required Pyomo packages. Lines 4–6 declare the number of discretization points and lines 9–11 declare sets specifying indices for all the discretization points. Lines 14–20 define the collocation matrix for 4th order Lagrange polynomials with Radau roots. Computing this matrix by hand is often one of the more tedious parts of implementing a dynamic algebraic optimization model. Lines 22–23 define the spatial and time scaling for each finite element and lines 25-30 define the model variables, including the differential variables.

Models that are discretized manually typically involve discretization points that lack any physical meaning, i.e., time/distance scaling is handled separately from the definition of the discretization points. This approach often leads to modeling errors and confusion especially when more complicated discretization schemes such as collocation are considered. The model below tries to overcome this difficulty by introducing an additional differential variable `m.t` that is used to calculate the properly scaled time points. Lines 32–54 define constraints representing the heat transfer model (5.1a). The discretization schemes are then applied in lines 56–93. Finally, a "dummy" objective function (because the example involves no optimization objective) is declared in line 95 and the model is solved using Ipopt in lines 97–98. The entire implementation requires fewer than 100 lines of code. While changing the number of finite element points is straightforward, changing the number of collocation points or either of the discretization schemes would require significant changes.

Listing 19 Manually discretized heat transfer model in Pyomo

```
1  from pyomo.environ import *
2
3  m = ConcreteModel()
4  m.nfet = Param(initialize=20)    # Number of finite elements in time
5  m.ncp = Param(initialize=4)      # Number of collocation points per finite element
6  m.nfex = Param(initialize=25)    # Number of finite elements in space
7  m.pi = Param(initialize=3.1416) # Pi
8
9  m.tfe = RangeSet(0, value(m.nfet)-1) # Set of finite element points in time
10 m.tcp = RangeSet(1, value(m.ncp))    # Set of collocation points in time
11 m.x = RangeSet(0, value(m.nfex))     # Set of finite element points in space
12
13 # Collocation Matrix
14 radau_adot={(1,1):-9.0000000000000,(1,2):-4.1393876913398,(1,3):1.7393876913398,\
15         (1,4):-2.9999999999999,(2,1):10.0488093998274,(2,2):3.2247448713915,\
16         (2,3):-3.5678400846904,(2,4):5.5319726474218,(3,1):-1.3821427331607,\
17         (3,2):1.1678400846904,(3,3):0.7752551286084,(3,4):-7.5319726474218,\
18         (4,1):0.3333333333333,(4,2):-0.2531972647421,(4,3):1.0531972647421,\
19         (4,4):5.0000000000000}
20 m.adot = Param(m.tcp, m.tcp, initialize=radau_adot)
21
```

```
m.ht = Param(initialize=2.0/m.nfet) # Length of time finite element
m.hx = Param(initialize=1.0/m.nfex) # Length of spatial finite element

m.u = Var(m.x, m.tfe, m.tcp, initialize=0.2)
m.t = Var(m.tfe, m.tcp)

m.dudx = Var(m.x, m.tfe, m.tcp)
m.dudx2 = Var(m.x, m.tfe, m.tcp)
m.dudt = Var(m.x, m.tfe, m.tcp)

def _pde(m, i, j, k):
    if i == 0 or i == value(m.nfex) or k == 0 :
        return Constraint.Skip
    return m.pi**2*m.dudt[i,j,k] == m.dudx2[i,j,k]
m.pde = Constraint(m.x, m.tfe, m.tcp,rule=_pde)

def _initcon(m,i):
    if i == 0 or i == value(m.nfex):
        return Constraint.Skip
    return m.u[i,0,1] == sin(m.pi*i*m.hx)
m.initcon = Constraint(m.x, rule=_initcon)

def _lowerbound(m, j, k):
    return m.u[0,j,k] == 0
m.lowerbound = Constraint(m.tfe, m.tcp, rule=_lowerbound)

def _upperbound(m, j, k):
    return m.pi*exp(-m.t[j,k])+m.dudx[value(m.nfex),j,k] == 0
m.upperbound = Constraint(m.tfe, m.tcp, rule=_upperbound)

def _init_t(m):
    return m.t[0,1] == 0
m._init_t = Constraint(rule=_init_t)

# Apply finite difference discretization equations
def _dudx_backwardDifference(m, i, j, k):
    if i == 0:
        return Constraint.Skip
    return (m.u[i,j,k]-m.u[i-1,j,k])/m.hx == m.dudx[i,j,k]
m.dudx_backwardDifference = Constraint(m.x, m.tfe, m.tcp, rule=_dudx_backwardDifference)

def _dudx2_backwardDifference(m, i, j, k):
    if i == 0 or i == 1:
        return Constraint.Skip
    return (m.u[i-2,j,k]-2*m.u[i-1,j,k]+m.u[i,j,k])/m.hx**2 == m.dudx2[i,j,k]
m.dudx2_backwardDifference = Constraint(m.x, m.tfe, m.tcp, rule=_dudx2_backwardDifference)

# Apply collocation discretization equations
def _dudt_collocation(m, i, j, k):
    if k == 1:
        return Constraint.Skip
    return sum(m.u[i,j,s]*m.adot[s,k] for s in m.tcp) == m.ht*m.dudt[i,j,k]
m.dudt_collocation = Constraint(m.x, m.tfe, m.tcp, rule=_dudt_collocation)

def _dt_collocation(m, j, k):
    if k == 1:
        return Constraint.Skip
    return sum(m.t[j,s]*m.adot[s,k] for s in m.tcp) == m.ht
m.dt_collocation = Constraint(m.tfe, m.tcp, rule=_dt_collocation)

# Apply collocation continuity equations
def _u_continuity(m, i, j):
    if j == value(m.nfet)-1:
        return Constraint.Skip
    return m.u[i,j+1,1] == m.u[i,j,4]
m.u_continuity = Constraint(m.x, m.tfe, rule=_u_continuity)

def _t_continuity(m, j):
    if j == value(m.nfet)-1:
        return Constraint.Skip
    return m.t[j+1,1] == m.t[j,4]
m.t_continuity = Constraint(m.tfe, rule=_t_continuity)

```

```
m.obj = Objective(expr=1)

solver = SolverFactory('ipopt')
results = solver.solve(m,tee=True)
```

We now contrast the Pyomo model with manual discretization to one that is discretized using pyomo.dae as shown in Listing 20 below. The model requires one additional package import in order to access the modeling components and transformations available in pyomo.dae. Lines 4-8 define the model sets and variables. Note that temporal and spatial scaling is handled explicitly via the defined ContinuousSet components. Specifically, after applying a discretization scheme, the components 'm.t' and 'm.x' will contain actual points in time and space, respectively. This approach eliminates the need for an additional differential variable to represent time, as was used in the manually discretized model. Lines 10-13 define the derivatives that appear in the model, lines 15-34 define the equations associated with the heat transfer model, and line 36 defines the objective function. Lines 38-42 apply discretization schemes defined in pyomo.dae to the model. Line 41 applies the backward difference method to the spatial ContinuousSet 'm.x' and line 42 applies Radau collocation to the temporal ContinuousSet 'm.t'.

Listing 20 Automatically discretized heat transfer model in Pyomo using pyomo.dae

```
from pyomo.environ import *
from pyomo.dae import *

m = ConcreteModel()
m.pi = Param(initialize=3.1416)
m.t = ContinuousSet(bounds=(0, 2))
m.x = ContinuousSet(bounds=(0, 1))
m.u = Var(m.x,m.t)

# Declare derivatives in the model
m.dudx = DerivativeVar(m.u, wrt=m.x)
m.dudx2 = DerivativeVar(m.u, wrt=(m.x, m.x))
m.dudt = DerivativeVar(m.u, wrt=m.t)

# Declare PDE
def _pde(m, i, j):
    if i == 0 or i == 1 or j == 0 :
        return Constraint.Skip
    return m.pi**2*m.dudt[i,j] == m.dudx2[i,j]
m.pde = Constraint(m.x, m.t, rule=_pde)

def _initcon(m, i):
    if i == 0 or i == 1:
        return Constraint.Skip
    return m.u[i,0] == sin(m.pi*i)
m.initcon = Constraint(m.x, rule=_initcon)

def _lowerbound(m, j):
    return m.u[0,j] == 0
m.lowerbound = Constraint(m.t, rule=_lowerbound)

def _upperbound(m, j):
    return m.pi*exp(-j)+m.dudx[1,j] == 0
```

```
34 m.upperbound = Constraint(m.t, rule=_upperbound)
35
36 m.obj = Objective(expr=1)
37
38 # Discretize using Finite Difference and Collocation
39 discretizer = TransformationFactory('dae.finite_difference')
40 discretizer2 = TransformationFactory('dae.collocation')
41 discretizer.apply_to(m, nfe=25, wrt=m.x, scheme='BACKWARD')
42 discretizer2.apply_to(m, nfe=20, ncp=3, wrt=m.t)
43
44 solver = SolverFactory('ipopt')
45 results = solver.solve(m, tee=True)
```

The implementation using `pyomo.dae` requires only half the number of lines of the manually discretized model, leading to a much more concise and consequently comprehensible model. The applied discretization schemes are completely independent of the model, and as a result, the correspondence between the algebraic model and the mathematical model is significantly more clear. The automatically discretized model does not have any of the discretization-specific parameters used in the manually discretized model. Furthermore, changing the number of finite elements, the number of collocation points, or the entire discretization scheme is simply accomplished by modifying one of the 4 lines of code defining the discretization. To illustrate this point, some alternative approaches to discretize the heat transfer model are as follows:

Listing 21 Alternate discretizations for heat the transfer model in Pyomo using `pyomo.dae`

```
1 # Discretize entire model using Finite Difference Method
2 discretizer = TransformationFactory('dae.finite_difference')
3 discretizer.apply_to(m, nfe=25, wrt=m.x, scheme='BACKWARD')
4 discretizer.apply_to(m, nfe=20, wrt=m.t, scheme='FORWARD')
5
6 # Discretize entire model using Collocation
7 discretizer = TransformationFactory('dae.collocation')
8 discretizer.apply_to(m, nfe=10, ncp=3, wrt=m.x, scheme='LAGRANGE-LEGENDRE')
9 discretizer.apply_to(m, nfe=20, ncp=3, wrt=m.t, scheme='LAGRANGE-RADAU')
```

With `pyomo.dae`, experimenting with the type of discretization scheme in addition to the resolution becomes accessible and systematic. However, we observe that not every discretization scheme is appropriate for every model. Different boundary and initial conditions may need to be applied, depending on the model and the discretization scheme used. The `pyomo.dae` package does not perform any model checking to ensure appropriateness of the model to which a discretization scheme is applied. Rather, such checking is the responsibility of the user. The `pyomo.dae` package, however, provides the necessary constructs to enable the user to easily tailor the specification of boundary conditions.

5.2 Optimal control

The purpose of optimal control problems is to find a sequence of inputs to a dynamic system that optimize some system performance metric. For example, a typical optimization objective involves minimizing or maximizing the value of a state variable at the end of a fixed time horizon. Consider the following small example, taken from [20]:

$$\begin{aligned}
\min \quad & x_3(t_f) && \text{(5.2a)}\\
\text{s.t.} \quad & \dot{x}_1 = x_2 && \text{(5.2b)}\\
& \dot{x}_2 = -x_2 + u && \text{(5.2c)}\\
& \dot{x}_3 = x_1^2 + x_2^2 + 0.005 \cdot u^2 && \text{(5.2d)}\\
& x_2 - 8 \cdot (t - 0.5)^2 + 0.5 \leq 0 && \text{(5.2e)}\\
x_1(0) = 0,\ & x_2(0) = -1,\ x_3(0) = 0, t_f = 1 && \text{(5.2f)}
\end{aligned}$$

This example consists of three state variables x_1, x_2, x_3 and one control variable u. The following code fragment illustrates the implementation of some of the more interesting features of this model, including the objective function (5.2a) and the complex path constraint (5.2e).

Listing 22 Partial implementation of problem (5.2) in Pyomo using `pyomo.dae`

```
# Setting the objective function
m.obj = Objective(expr=m.x3[m.t.last()])

# Declaring the path constraint
def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

Discretizing problem (5.2) using orthogonal collocation with 7 finite elements and 6 collocation points per element will result in the optimal profiles shown in the top portion of Fig. 3. In this solution, the control variable is allowed to vary at every time point. In many applications this sort of continuous change in the control variable is undesirable or impossible to implement. Therefore, in practice, control variables are often restricted to be piecewise constant over a certain time interval. Such restrictions can be easily implemented using `pyomo.dae`, using the `reduce_collocation_points()` method of the `dae.collocation` discretization transformation. We illustrate use of this method in the code fragment presented below. The last line of the fragment restricts the control variable u to only 1 free collocation point per finite element, rendering it piecewise constant. The resulting state and control profiles are shown in the bottom of Fig. 3. If instead we wanted the control variable to be piecewise linear we would only have to modify the 'ncp' keyword argument in the `reduce_collocation_points()` method call to 'ncp=2', which would yield two free collocation points (degrees of freedom) for the control variable per finite element.

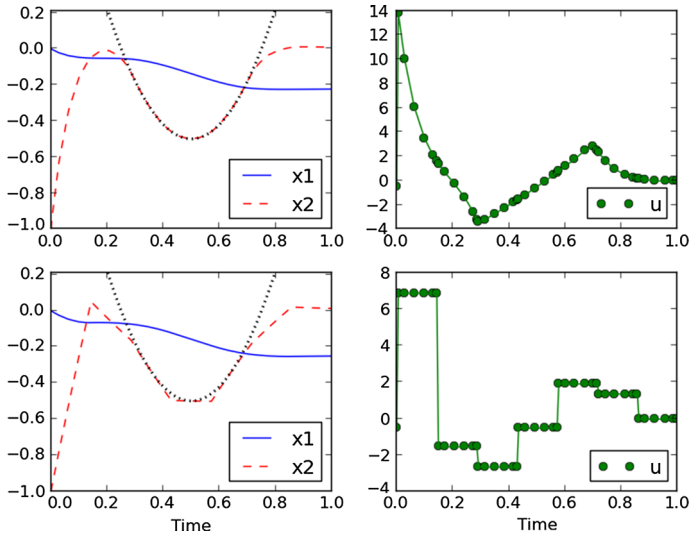

Fig. 3 Solution to the optimal control problem (top) with no restrictions on the control variable and (bottom) restricting the control variable to be piecewise constant. Black dotted line shows the inequality path constraint

Listing 23 Code for discretizing the optimal control model and reducing the degrees of freedom

```
# Discretize model using radau collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(model, wrt=model.t, nfe=7, ncp=6)

# Restrict control to be piecewise constant
discretizer.reduce_collocation_points(model, var=model.u, ncp=1, contset=model.t)
```

5.3 Parameter estimation for disease transmission

Our next application considers a non-linear parameter estimation problem taken from [31]. The objective of this problem is to estimate parameters for a model of childhood infectious disease, using real-world disease case data. The mathematical model for this problem is as follows:

$$\min \quad \omega_I \sum_{i \in \mathcal{F}} (\epsilon_{I_i})^2 + \omega_\phi \sum_{k \in \mathcal{T}} (\epsilon_{\phi_k})^2 \tag{5.3a}$$

$$\text{s.t.} \quad \frac{dS}{dt} = \frac{-\beta(y(t))S(t)I(t)}{N} - \epsilon_I(t) + B(t) \tag{5.3b}$$

$$\frac{dI}{dt} = \frac{\beta(y(t))S(t)I(t)}{N} + \epsilon_I(t) - \gamma I(t) \tag{5.3c}$$

$$\frac{d\phi}{dt} = \frac{\beta(y(t))S(t)I(t)}{N} + \epsilon_I(t) \tag{5.3d}$$

$$R_k^\star = \eta_k(\phi_{i,k} - \phi_{i,k-1}) + \epsilon_{\phi_k}, \; k \in \mathcal{T} \tag{5.3e}$$

$$\beta_k = \beta^{\text{mag}} \cdot \beta_k^{\text{patt}}, \ k \in \mathcal{T} \tag{5.3f}$$

$$1.0 = \frac{\sum_{k\in\mathcal{T}} \beta_k^{\text{patt}}}{\text{len}(\mathcal{T})} \tag{5.3g}$$

$$\bar{S} = \frac{\sum_{i\in\mathcal{F}} S_i}{\text{len}(\mathcal{F})}, \quad \bar{\beta} = \frac{\sum_{k\in\mathcal{T}} \beta_k}{\text{len}(\mathcal{T})} \tag{5.3h}$$

$$0 \le I(t), \quad S(t) \le N, \quad 0 \le \beta(y(t)), \quad 0 \le \phi(t) \tag{5.3i}$$

In this model, S represents the number of people susceptible to the disease, I denotes the number of people with the disease who are infectious, and ϕ denotes the cumulative incidence of the disease. The $R^\star$ quantity corresponds to the reported incidence of the disease and is a known input at a discrete set of reporting times, $\mathcal{T}$. The len() function refers to the length, or number of elements, in a discrete set. For a more detailed description of this model, we refer the reader to [31].

The purpose of this model is to estimate the time-varying disease transmission parameter β under the assumption that β varies seasonally but is constant from year to year (controlling for season). This hypothesis induces the following structural elements to the temporal data: $\mathcal{T}$, the set of reporting times over a single year, and $\mathcal{F}$, the set of all reporting times over a 20 year period. The set $\mathcal{F}$ is represented by the `pyomo.dae` continuous set `model.TIME`. The following code fragment below shows how equation (5.3d) is implemented accounting for these two time sets:

Listing 24 Code for declaring a subset of the components and implementing the differential equation (5.3d) in the disease transmission model

```
1  model.TIME = ContinuousSet(initialize=_TIME_init, bounds=(0,None))
2  model.beta = Var(model.S_BETA, initialize=_init_beta, bounds=(0.01,5))
3  model.S = Var(model.TIME, initialize=_init_S, bounds=_people_bounds)
4  model.I = Var(model.TIME, initialize=_init_I, bounds=_people_bounds)
5  model.phi = Var(model.TIME, initialize=_init_phi, bounds=(0,None))
6  model.Sdot = DerivativeVar(model.S, initialize=_init_Sdot)
7  model.Idot = DerivativeVar(model.I, initialize=_init_Idot)
8  model.phidot = DerivativeVar(model.phi, initialize=_init_phidot, bounds=(-10,None))
9  def _phidot_eq(model,i):
10     if i == 0:
11         return Constraint.Skip
12     fe = model.TIME.get_upper_element_boundary(i)
13     j = model.TIME._fe.index(fe)
14     return model.phidot[i] == model.eps_I[j] \
15       + (model.beta[model.P_BETA_NDX[j]]*model.I[i]*model.S[i])/model.P_POP
16 model.phidot_eq = Constraint(model.TIME, rule=_phidot_eq)
```

In this example, lines 12 and 13 take a time point $i \in \mathcal{F}$ and determine the corresponding upper finite element boundary `fe`. The parameter `model.P_BETA_NDX` then maps this finite element value to the correct index in $\mathcal{T}$ for `model.beta`.

Our infectious disease transmission model has three differential equations and was discretized with 520 finite elements and 3 collocation points per finite element, using orthogonal collocation with Lagrange polynomials and Radau roots. This leads to a non-linear program with 10,458 variables and 9,910 equality constraints. The model was discretized such that all finite element points correspond to a reporting time for $R^\star$. As with our previous application example, the model was implemented twice: manually, as reported in [31], and using `pyomo.dae`, by the authors. Table 2 compares

timing and solution statistics for the two implementations. The creation time refers to the amount of time required to execute the Python script that constructs the model. For the implementation using `pyomo.dae`, the creation time also includes the time for automatic discretization. If we compare the creation time for the two implementations, we observe that the model using `pyomo.dae` takes a few more seconds to build than the manually discretized model implying that there is a slight overhead associated with applying the discretization automatically.

Comparing the solver results for the two implementations, we see analogous results. In particular, the solve time for the two models is practically identical, and Ipopt requires nearly the same number of iterations to locate an optimal solution. While the model creation time results reported in Table 2 show minimal differences between the two implementations, the statistics do not account for one crucial component of the applications development process: the time required to design, implement, debug, and refine the manual discretization. Initial manual implementation and debugging of a discretization scheme like collocation can easily take over an hour for novice modelers. In contrast, incorporating a differential equation into an optimization model using `pyomo.dae` is as simple as writing a regular model constraint and selecting a particular discretization scheme to apply, which can be accomplished in a fraction of the time.

Finally, we consider the issue of model initialization, which is a critical step in solving optimization models with differential equations. The following code fragment shows how the variable S is initialized in our disease transmission model:

Listing 25 Code for initializing S in the disease transmission model

```
def _init_S(model,i):
    if i==0:
        return model.init_S_init
    fe = model.TIME.get_upper_element_boundary(i)
    j = model.TIME._fe.index(fe)
    return model.init_S[j]
def _people_bounds(model):
    return (0.0, model.P_POP)
model.S = Var(model.TIME, initialize=_init_S, bounds=_people_bounds)
```

This example is only one illustration of how a `pyomo.dae` model can be initialized, other alternatives may be implemented similarly. In particular, the approach presented above initializes the finite element points using a profile defined by the time-varying parameter `model.init_S` and initializes the intermediate collocation

Table 2 Comparison of two implementations of the disease transmission parameter estimation problem

	Manual discretization	Using `pyomo.dae`
Creation time (s)	1.79	5.29
Solve time (CPU sec)	1.35	0.86
Ipopt iterations	27	26
Objective ($\times 10^{-5}$)	1.4716	1.4716

points by equating them to the value at the upper finite element boundary. We chose this initialization approach because it is sufficiently flexible to handle discretizations with any number of collocation points. The pyomo.dae package does not currently provide any automatic frameworks for model initialization. However, using existing Pyomo and Python constructs, a user can create any requisite initializations.

5.4 Stochastic optimal control of natural gas networks

In our last application example, we consider the problem of optimizing natural gas network inventories while accounting for uncertainties in the system. Specifically, the objective is to satisfy uncertain gas demands in the network by building up inventory in the pipelines in a way that minimizes the required compression power. A model of this problem was proposed in [33], which is as follows:

$$\min \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{S}_n} c_s s_{i,t}\Delta\tau + \sum_{t\in\mathcal{T}}\sum_{\ell\in\mathcal{L}} c_e P_{\ell,t}\Delta\tau + \sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{D}} c_d(d_{j,t}-\bar{d}_{j,t})^2 \tag{5.4a}$$

$$+ \sum_{k\in\mathcal{X}}\sum_{\ell\in\mathcal{L}} c_T(p_{\ell,T,k}-p_{\ell,0,k})^2 + \sum_{k\in\mathcal{X}}\sum_{\ell\in\mathcal{L}} c_T(f_{\ell,T,k}-f_{\ell,0,k})^2 \tag{5.4b}$$

$$\text{s.t. } \frac{\partial p_\ell(x,\tau)}{\partial\tau} = -c_{1,\ell}\frac{\partial f_\ell(x,\tau)}{\partial x}, \quad \ell\in\mathcal{L},\ x\in[0,L_\ell],\ \tau\in[0,T] \tag{5.4c}$$

$$\frac{\partial f_\ell(x,\tau)}{\partial t} = -c_{2,\ell}\frac{\partial p_\ell(x,\tau)}{\partial x} - c_{3,\ell}\frac{f_\ell(x,\tau)|f_\ell(x,\tau)|}{p_\ell(x,\tau)}, \quad \ell\in\mathcal{L},\ x\in[0,L_\ell],\ \tau\in[0,T] \tag{5.4d}$$

$$p_\ell(L_\ell,\tau) = \theta_{rec(\ell)}(\tau),\ \ell\in\mathcal{L},\ \tau\in[0,T] \tag{5.4e}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau),\ \ell\in\mathcal{L}_p,\ \tau\in[0,T] \tag{5.4f}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau) + \Delta\theta_\ell(\tau),\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4g}$$

$$\sum_{\ell\in\mathcal{L}_n^{rec}} f_\ell(L_\ell,\tau) - \sum_{\ell\in\mathcal{L}_n^{snd}} f_\ell(0,\tau) + \sum_{i\in\mathcal{S}_n} s_i(\tau) - \sum_{j\in\mathcal{D}_n} d_j(\tau) = 0, \quad n\in\mathcal{N}\ \tau\in[0,T] \tag{5.4h}$$

$$P_\ell(\tau) = c_p\cdot T\cdot f_\ell(0,\tau)\left(\left(\frac{\theta_{snd(\ell)}(\tau)+\Delta\theta_\ell(\tau)}{\theta_{snd(\ell)}(\tau)}\right)^{\frac{\gamma-1}{\gamma}} - 1\right), \quad \ell\in\mathcal{L},\ \tau\in[0,T] \tag{5.4i}$$

$$\theta_{sup(i),\tau} = \bar{\theta}_i^{sup},\ i\in\mathcal{S},\ \tau\in[0,T] \tag{5.4j}$$

$$0 = -c_{1,\ell}\frac{\partial f_\ell(x,0)}{\partial x},\ \ell\in\mathcal{L},\ x\in[0,L_\ell] \tag{5.4k}$$

$$0 = -c_{2,\ell}\frac{\partial p_\ell(x,0)}{\partial x} - c_{3,\ell}\frac{f_\ell(x,0)|f_\ell(x,0)|}{p_\ell(x,0)},\ \ell\in\mathcal{L},\ x\in[0,L_\ell] \tag{5.4l}$$

$$P_\ell^L \le P_\ell(\tau) \le P_\ell^U,\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4m}$$

$$\theta_\ell^{suc,L} \le \theta_{snd(\ell)}(\tau) \le \theta_\ell^{suc,U},\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4n}$$

$$\theta_{\ell}^{dis,L} \leq \theta_{snd(\ell)}(\tau) + \Delta\theta_{\ell}(\tau) \leq \theta_{\ell}^{dis,U}, \ \ell \in \mathcal{L}_a, \ \tau \in [0, T] \tag{5.4o}$$

This model includes detailed network dynamics captured by the PDEs (5.4c) and (5.4d). Note that these PDEs are indexed by the number of links $\mathcal{L}$ in the network, indicating that optimization of even moderately-sized networks will require the simultaneous solution of many PDEs. Uncertainty is captured by considering multiple scenarios for natural gas demand, and the resulting formulation is a two-stage stochastic optimization problem. Because they describe scenario-specific dynamics, model PDEs are replicated for each scenario. The resulting optimization model is very large, scaling as a function of the number of network links, scenarios, and discretization points. The `pyomo.dae` implementations for the PDEs and the pressure boundary conditions in this model are given in Listings 26 and 27.

Listing 26 Code for implementing the PDE equations (5.4c) and (5.4d)

```
# First PDE for gas network model
def flow_rule(m,j,i,t,k):
    if t == m.TIME.first() or k == m.DIS.last():
        return Constraint.Skip # Do not apply pde at initial time or final location
    return m.dpxdt[j,i,t,k]/3600 + m.c1[i]/m.llength[i]*m.dfxdx[j,i,t,k] == 0
model.flow = Constraint(model.SCEN, model.LINK, model.TIME, model.DIS, rule=flow_rule)

# Second PDE for gas network model
def press_rule(m,j,i,t,k):
    if t == m.TIME.first() or k == m.DIS.last():
        return Constraint.Skip # Do not apply pde at initial time or final location
    return m.dfxdt[j,i,t,k]/3600 == -m.c2[i]/m.llength[i]*m.dpxdx[j,i,t,k] - m.slack[j,i,t,k]
model.press = Constraint(model.SCEN, model.LINK, model.TIME, model.DIS, rule=press_rule)

def slackeq_rule(m,j,i,t,k):
    if t == m.TIME.last():
        return Constraint.Skip
    return m.slack[j,i,t,k]*m.px[j,i,t,k] == m.c3[i]*m.fx[j,i,t,k]*m.fx[j,i,t,k]
model.slackeq = Constraint(model.SCEN, model.LINK, model.TIME, model.DIS, rule=slackeq_rule)
```

Listing 27 Code for implementing the pressure boundary conditions (5.4e), (5.4f), and (5.4g)

```
# boundary conditions pressure, passive links
def presspas_start_rule(m,j,i,t):
    return m.px[j,i,t,m.DIS.first()] == m.p[j,m.lstartloc[i],t]
model.presspas_start = Constraint(model.SCEN, model.LINK_P, model.TIME, rule=presspas_start_rule)

def presspas_end_rule(m,j,i,t):
    return m.px[j,i,t,m.DIS.last()] == m.p[j,m.lendloc[i],t]
model.presspas_end = Constraint(model.SCEN, model.LINK_P, model.TIME, rule=presspas_end_rule)

# boundary conditions pressure, active links
def pressact_start_rule(m,j,i,t):
    return m.px[j,i,t,m.DIS.first()] == m.p[j,m.lstartloc[i],t]+m.dp[j,i,t]
model.pressact_start = Constraint(model.SCEN, model.LINK_A, model.TIME, rule=pressact_start_rule)

def pressact_end_rule(m,j,i,t):
    return m.px[j,i,t,m.DIS.last()] == m.p[j,m.lendloc[i],t]
model.pressact_end = Constraint(model.SCEN, model.LINK_A, model.TIME, rule=pressact_end_rule)
```

The model includes two continuous domains: temporal and spatial. The model was discretized in time using a backward finite difference scheme and was discretized in space using a forward finite difference scheme. Discretization involved 47 time

Table 3 Effect of spatial discretization resolution on computational performance

Scenarios	N_x	Variables	Constraints	Iterations	Creation (s)	Solve (s)
1	2	9686	9144	35	1.34	2.17
1	6	25718	25128	41	3.00	9.69
1	10	41750	41112	43	4.96	15.36
1	20	81830	81072	46	8.98	38.05
1	60	242150	240912	55	27.08	154.98
1	100	402470	400752	71	45.60	395.87
2	2	19371	18508	36	2.43	6.42
2	6	51435	50476	47	5.74	28.86
2	10	83499	82444	44	9.72	45.01
2	20	163659	162364	49	19.79	87.77
2	60	484299	482044	60	56.34	422.33
2	100	804939	801724	82	91.76	1078.56
3	2	29056	27872	36	3.84	11.97
3	6	77152	75824	43	9.26	58.15
3	10	125248	123776	48	14.86	92.68
3	20	245488	243656	60	29.42	222.56
3	60	726448	723176	67	84.29	779.96
3	100	1207408	1202696	71	137.48	1450.17

intervals and multiple numbers of spatial points. Table 3 reports the computational effort required for different spatial discretization resolutions. Figure 4 illustrates how the flow profile converges as the number of discretization points (N_x) increases. The largest problem solved had over 1,200,000 variables and constraints. Note that the creation time for the model is only a small fraction of the overall solution time. Again, the creation time includes the time to apply a discretization scheme automatically. We also see that the creation time does not increase as dramatically as the solve time as the number of discretization points is increased. This result suggests that the automatic discretization routines in `pyomo.dae` are scalable.

We note that the model reported in [33] was first implemented in Pyomo using a manual discretization scheme. During the process of converting the manually discretized model to `pyomo.dae` we actually discovered an "off-by-one" error in the manual discretization. These sort of errors are easy to make and often difficult to find, given the complexity of such models. In contrast, `pyomo.dae` provides a concise and automatic way of discretizing the variables and constraints in a model.

We again highlight that `pyomo.dae` does not perform any model checking. Therefore, users must specify a well-posed model and select discretization schemes that are appropriate for their model. In this particular example, we highlight the need for the user to supress the enforcement of the PDE at certain domain boundaries (as shown in Listing 26) in order to ensure consistency between the specified boundary conditions and the applied discretization scheme.

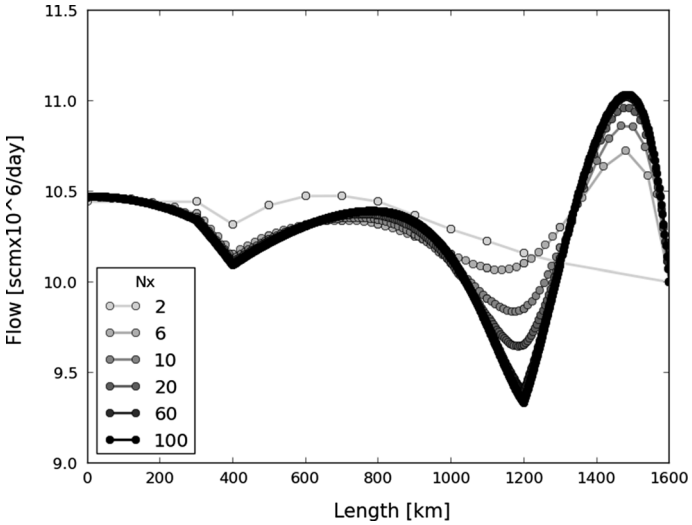

Fig. 4 Optimal axial flow profile in a single-scenario gas network for different resolutions. Darker colors indicate more discretization points

6 Conclusions and future directions

The level of modeling abstraction provided by `pyomo.dae` enables the development of novel formulations and the implementation of generalized analysis frameworks. For example, consider common applications of dynamic optimization, such as model predictive control (MPC). These applications require solution of a sequence of related optimization problems with differential equation components. Using `pyomo.dae`, users can implement such strategies as general frameworks, independent of the system model to which they are applied. In fact, a general implementation of MPC has already been developed using this framework [29].

Furthermore, there is the strong potential for `pyomo.dae` to be used in multigrid / multiscale applications. By separating the model from the discretization scheme, we enable general implementations of algorithms requiring either (1) the sequential solution of problems with different discretization resolutions or (2) the ability to communicate discretization information to specialized solvers.

In addition to the core modeling components available in any algebraic modeling language, Pyomo includes specialized packages for expressing generalized disjunctive programming (GDP) and stochastic programming (SP) problems. These extensions leverage the modular design of Pyomo and can be combined with `pyomo.dae`. For example, Pyomo users can now easily represent GDP problems with differential equations or SP problems incorporating a dynamic model in each scenario. By providing users a straightforward way to represent such complex models, our goal is ultimately to enable the development of new algorithms for solving them.

In the course of developing `pyomo.dae`, our design emphasis was on syntactic flexibility. This does not imply that models with every conceivable differential equation can be solved using this package. Also, because we are not enforcing specific

structure(s) on the differential equations or restricting the order of the derivatives, we do not check for modeling errors or consistency. Rather, the responsibility falls on the user not to implement an inconsistent model. A similar observation also applies to general algebraic modeling languages in which the user is responsible for not implementing models with incompatible constraints or empty feasible regions, for instance. However, with the availability of pyomo.dae, it is now possible to develop templates that create models with certain canonical structures, and as a result prevent users from specifying incompatible models. Further, such templates would allow implementation of solution algorithms tailored for specific structures, to check for modeling errors, and to more easily interface with other existing DAE solvers.

Currently pyomo.dae can only be used to express models with bounded rectangular continuous domains. In the future we plan on extending the package to represent unbounded domains, making it easier to represent control problems over infinite horizons. We also plan on adding a modeling component to represent integrals.

The pyomo.dae package includes a variety of discretization transformations as well as a framework for users to implement their own custom discretization strategies. However, simultaneous discretization is just one of several approaches available. Single and multiple shooting methods are other common solution strategies [8,21,30]. We plan to eventually link pyomo.dae to existing integrators, such as Sundials [17], and then add implementations of shooting methods. An integrator would also allow us to simulate dynamic models in order to initialize our discretized optimization problem or develop hybrid discretization / shooting algorithms.

Another useful extension of pyomo.dae would involve the development of more sophisticated frameworks for model initialization and specification of time dependent data. Moreover, an automatic, element-by-element initialization is another possible extension. In the current implementation a user can provide an initialization for a model after it has been discretized. However, this is not entirely straightforward, and the initialization is tied to the discretization applied. We would also like to implement several data interpolation schemes in order to make this initialization process easier and consistent across any choice of discretization. An interpolation scheme would also be useful for estimating differential variable values at any point in the continuous domain after the model has been solved, especially in the case of a collocation discretization where a high order functional form of the state profile already exists.

Acknowledgements We thank Carl D. Laird for useful technical discussions and for providing the disease transmission model. Victor M. Zavala acknowledges funding from the U.S. Department of Energy Early Career program. The research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under contract number KJ0401000 through the Project "Multifaceted Mathematics for Complex Energy Systems". Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Appendix A: Code structure overview

pyomo.dae introduces only a handful of new files and roughly 2,200 lines of code to the Pyomo code base. The pyomo.dae code can be broken into three categories:

– Class definitions for new modeling components
– Class definitions for discretization transformations
– Miscellaneous utility functions

The files belonging to each category are described below.

A.1 New modeling components

The new modeling components introduced in `pyomo.dae` are derived from standard Pyomo components. This simplifies the implementation, enables code reuse, and ensures that these components are compatible with the downstream Pyomo infrastructure for writing out problems and reading in solutions.

`contset.py`

This file contains the `ContinuousSet` class which is derived from the Pyomo `OrderedSimpleSet` component. This class performs several checks during object construction to ensure a bounded continuous domain is being represented. It also includes access methods for querying the discretization points contained in the `ContinuousSet`. The rest of this component's functionality is inherited.

`diffvar.py`

This file contains the `DerivativeVar` class which is derived from the Pyomo `Var` component. Much of this class consists of checks in the constructor around the state variable that the `DerivativeVar` is differentiating. The rest of the class includes several methods for returning information about the derivative, building up the discretization equations for the derivative, and checking if the `DerivativeVar` is fully discretized.

A.2 Discretization transformations

The discretization transformations included in `pyomo.dae` are subclasses of the Pyomo `Transformation` class and utilize the existing Pyomo plugin system for standardizing transformations across the Pyomo code base. The finite difference and collocation discretizations are implemented in separate files but are very similar in terms of class and method structure. It is important to note that the discretization transformations are performed "in-place" and not on a copy of the model.

`finitedifference.py`

This file contains the implementations of several types of finite difference schemes. There are functions at the top of the file that contain the code specific to a particular discretization scheme. For example,

_forward_transform implements the forward difference equation for first-order derivatives. The discretization transformation is managed by the _apply_to and _transformBlock methods of the Finite_Difference_Transformation class. The _apply_to method checks the user specified options for the discretization and loops through every active Block in the model, passing it to the _transformBlock method. The _transformBlock method calls functions for adding discretization points, builds discretization expressions for every DerivativeVar in the Block, and checks if the DerivativeVar is fully discretized. Fully discretized DerivativeVar components are reclassified as regular Pyomo Var components. This reclassification step ensures that the Pyomo problem writers throw an error when they encounter un-discretized derivative components but support discretized models.

colloc.py

This file containts the implementations of two types of collocation schemes. Similar to the previous file for finite difference discretizations, this file has functions at the top that implement the specific discretization equations for a particular scheme such as _lagrange_radau_transform. The Collocation_Discretization_Transformation class is nearly identical in structure to the Finite_Difference_Transformation class with the addition of extra methods for gathering or computing the collocation constants required in the discretization equations. If the user has Numpy installed then these constants will be computed using several functions included in this file. Finally, the Collocation_Discretization_Transformation class contains a method called reduce_collocation_points for reducing the degrees of freedom for a particular variable. This method does a series of checks on the user specified options and then builds the interpolation equations described in the paper.

A. 3 Utility functions

The last two files included in pyomo.dae contain utility functions and data that are used by the discretization transformations.

misc.py

This file contains miscellaneous functions that are used by every discretization transformation. It includes functions like generate_finite_elements and generate_colloc_points for adding discretization points to a ContinuousSet. In addition it contains the function update_contset_indexed_component which will expand any ContinuousSet indexed components to include indices for any discretization points added. There are private functions for updating each type of component such

as `_update_var` and `_update_constraint`. Finally, this file contains several utility functions for generating and adding discretization equations to a model such as `add_discretization_equations` and `add_continuity_equations` which add discretization and continuity constraints to the model.

`utilities.py`

This file includes Python dictionaries containing pre-computed constants required for the collocation discretization. They are used when a user does not have the Python package Numpy installed, as Numpy is not a required dependency of Pyomo.

References

1. Åkesson, J., Årzén, K.E., Gäfvert, M., Bergdahl, T., Tummescheit, H.: Modeling and optimization with Optimica and JModelica.org languages and tools for solving large-scale dynamic optimization problems. Comput. Chem. Eng. **34**(11), 1737–1749 (2010)
2. Andersson, J.: A General-purpose Software Framework for Dynamic Optimization. PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium (2013)
3. Ascher, U.M., Mattheij, R.M., Russell, R.D.: Numerical Solution of Boundary Value Problems for Ordinary Differential Equations, vol. 13. SIAM, Philadelphia (1994)
4. Ascher, U.M., Petzold, L.R.: Computer Methods for Ordinary Differential Equations and Differential-algebraic Equations, vol. 61. SIAM, Philadelphia (1998)
5. Betts, J.T.: Sparse Optimization Suite (SOS). Applied Mathematical Analysis, LLC (2013)
6. Betts, J.T., Huffman, W.P.: Sparse optimal control software SOCS MEA-LR-085, Boeing Information and Support Services, The Boeing Company, PO Box. Math. Eng. Anal. Tech. Doc. **3707, 98**, 124–2207 (1997)
7. Biegler, L.T.: Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. SIAM, Philadelphia (2010)
8. Bock, H.G., Plitt, K.J.: A multiple shooting algorithm for direct solution of optimal control problems. In: Proceedings of the IFAC World Congress (1984)
9. Brenan, K.E., Campbell, S.L., Petzold, L.R.: Numerical Solution of Initial-value Problems in Differential-Algebraic Equations, vol. 14. SIAM, Philadelphia (1996)
10. Butcher, J.: Numerical Methods for Ordinary Differential Equations. Wiley (2003). https://books.google.com/books?id=nYuDWkxhDGUC
11. Corporation, G.D.: General Algebraic Modeling System (GAMS) Release 24.2.1. Washington, DC, USA (2013). http://www.gams.com/
12. Fourer, R., Gay, D., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Scientific Press (1993). https://books.google.com/books?id=8vJQAAAAMAAJ
13. Frasch, J.: Parallel Algorithms for Optimization of Dynamic Systems in Real-Time. Ph.D. thesis, KU Leuven (2014)
14. Hart, W., Laird, C., Watson, J., Woodruff, D.: Pyomo-optimization Modeling in Python, vol. 67. Springer, Berlin (2012)
15. Hart, W., Watson, J., Woodruff, D.: Pyomo: modeling and solving mathematical programs in Python. Math. Program. Comput. **3**(3), 219–260 (2011)
16. Hendengren, J.: APMonitor Modeling Language (2014). http://APMonitor.com
17. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: SUNDIALS: suite of nonlinear and differential/algebraic equation solvers. ACM Trans. Math. Softw. **31**(3), 363–396 (2005). https://doi.org/10.1145/1089014.1089020
18. Houska, B., Ferreau, H., Diehl, M.: ACADO toolkit—an open source framework for automatic control and dynamic optimization. Opt. Control Appl. Methods **32**(3), 298–312 (2011)
19. Hultberg, T.: FlopC++ an algebraic modeling language embedded in C. In: Waldmann, K.-H. (ed.)Operations Research Proceedings 2006, ser. Operations Research Proceedings, pp. 187–190. Springer, Berlin (2006)

20. Jacobson, D., Lele, M.: A transformation technique for optimal control problems with a state variable inequality constraint. IEEE Trans. Autom. Control **14**(5), 457–464 (1969). https://doi.org/10.1109/TAC.1969.1099283
21. Kraft, D.: On converting optimal control problems into nonlinear programming problems. In: Schittkowski, K. (ed.) Computational Mathematical Programming, pp. 261–280. Springer, Berlin (1985)
22. Leyffer, S., Kirches, C.: TACO–a toolkit for AMPL control optimization. Math. Program. Comput. **5**(3), 227–265 (2013)
23. Lubin, M., Dunning, I.: Computing in operations research using julia. INFORMS J. Comput. **27**(2), 238–248 (2015). https://doi.org/10.1287/ijoc.2014.0623
24. MATLAB: MATLAB Function Reference, chap. pdepe, pp. 6386–6397. The MathWorks Incorporated (2015)
25. Mitchell, S., O'Sullivan, M., Dunning, I.: PuLP: A Linear Programming Toolkit for Python (2011)
26. Patterson, M.A., Rao, A.V.: GPOPS-II: A MATLAB software for solving multiple-phase optimal control problems using hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming. ACM Trans. Math. Softw. **41**(1), 1:1–1:37 (2014). https://doi.org/10.1145/2558904
27. Process Systems Enterprise: gPROMS (1997–2014). www.psenterprise.com/gproms
28. Rutquist, P., Edvall, M.: PROPT—Matlab Optimal Control Software. Tomlab Optimization Inc (2010)
29. Santamara, F.L., Gmez, J.M.: Framework in PYOMO for the assessment and implementation of (as)nmpc controllers. Comput. Chem. Eng. **92**, 93–111 (2016). https://doi.org/10.1016/j.compchemeng.2016.05.005. http://www.sciencedirect.com/science/article/pii/S0098135416301533
30. Sargent, R., Sullivan, G.: The development of an efficient optimal control package. In: Stoer, J. (ed.) Optimization Techniques, Proceedings of the 8th IFIP Conference on Optimization Techniques Würzburg, September 5–9, 1977 pp. 158–168. Springer, Berlin (1978)
31. Word, D.P.: Nonlinear Programming Approaches for Efficient Large-Scale Parameter Estimation with Applications in Epidemiology. PhD thesis, Texas A&M University, Department of Chemical Engineering, College Station, TX (2013)
32. Word, D.P., Kang, J., Akesson, J., Laird, C.D.: Efficient parallel solution of large-scale nonlinear dynamic optimization problems. Comput. Optim. Appl. **59**(3), 667–688 (2014)
33. Zavala, V.M.: Stochastic optimal control model for natural gas networks. Comput. Chem. Eng. **64**, 103–113 (2014). https://doi.org/10.1016/j.compchemeng.2014.02.002. http://www.sciencedirect.com/science/article/pii/S0098135414000349
34. Zavala, V.M., Laird, C.D., Biegler, L.T.: Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. Chem. Eng. Sci. **63**(19), 4834–4845 (2008)