DAE Tools Project Documentation

Release 1.3.1

by Dragan Nikolic

May 22, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is DAE Tools?

DAE Tools is a free/open-source cross-platform object- and equation-oriented process modelling and optimization software. It is not a modelling language, rather a collection of software tools for:

- Modelling development
- Simulation, optimization, and parameter estimation
- Processing of the results (plotting and exporting to various file formats)
- Report generation
- Code generation and model exchange

DAE Tools is initially developed to model and simulate processes in chemical process industry (mass, heat and momentum transfers, chemical reactions, separation processes, thermodynamics). However, **DAE Tools** can be used to develop high-accuracy models of (in general) many different kind of processes/phenomena, simulate/optimize them, visualize and analyse the results.

The following approaches/paradigms are adopted in **DAE Tools**:

- A hybrid approach between general-purpose programming languages (such as c++ and Python) and domain-specific modelling languages (such as Modelica, gPROMS, Ascend etc.) (more information: *The Hybrid approach*)
- An object-oriented approach to process modelling (more information: *The Object-Oriented approach*)
- An Equation-Oriented (acausal) approach where all model variables and equations are generated and gathered together and solved simultaneously using a suitable mathematical algorithm (more information: *The Equation-Oriented approach*)
- Separation of the model definition from the activities that can be carried out on that model. The structure of the model (parameters, variables, equations etc.) is given in the model class while the runtime information in the simulation class. This way we can have a single model definition, but one or more different simulation scenarios, and one or more optimization scenarios
- Core libraries are written in standard c++, however Python is used as the main modelling language (more information: *Programming language*)

Class of problems that can be solved by **DAE Tools**:

- Initial value problems of implicit form, described by a system of linear, non-linear, and (partial-)differential algebraic equations
- Index-1 DAE systems
- With lumped or distributed parameters: Finite Difference or Finite Elements Methods (still experimental)
- Steady-state or dynamic
- Continuous with some elements of event-driven systems (discontinuous equations, state transition networks and discrete events)

Type of activities that can be carried out with models developed in **DAE Tools**:

- Simulation

  - Steady-state or dynamic
  - With simple or complex operating procedures
- Optimization
  - NLP problems
  - MINLP problems
- Parameter estimation
  - The least squares method (Levenberg–Marquardt algorithm)
- XML + MathML report generation
- Code generation for other modelling languages and general purpose programming languages
  - Modelica
  - ANSI C
  - Functional Mockup Interface (FMI)
- Export of the simulation results to various file formats

All core libraries are written in standard ANSI/ISO c++. It is highly portable - it runs on all major operating systems (GNU/Linux, MacOS, Windows) and all platforms with a decent c++ compiler, Boost and standard c/c++ libraries (by now it is tested on 32/64 bit x86 and ARM architectures making it suitable for use in embedded systems). Models can be developed in Python (**pyDAE** module) or c++ (**cDAE** module), compiled into an independent executable and deployed without a need for any run time libraries.

DAE Tools support a large number of solvers. Currently Sundials IDAS solver is used to solve DAE systems and calculate sensitivities, while BONMIN, IPOPT, and NLOPT solvers are used to solve NLP/MINLP problems. **DAE Tools** support direct dense and sparse matrix linear solvers (sequential and multi-threaded versions) at the moment. In addition to the built-in Sundials linear solvers, several third party libraries are interfaced: SuperLU/SuperLU_MT, Intel Pardiso, AMD ACML, Trilinos Amesos (KLU, Umfpack, SuperLU, Lapack), and Trilinos AztecOO (with built-in, Ifpack or ML preconditioners) which can take advantage of multi-core/cpu computers. Linear solvers that exploit general-purpose graphics processing units (GPGPU, such as NVidia CUDA) are also available ([[SuperLU_CUDA]], CUSP) but in an early development stage.

1.2 Licence

DAE Tools is free software and you can redistribute it and/or modify it under the terms of the GNU General Public Licence version 3 as published by the Free Software Foundation (GNU philosophy).

1.3 History

"Necessity, who is the mother of invention" *Plato, Greek author & philosopher (427 BC - 347 BC), The Republic*

"Every good work of software starts by scratching a developer's personal itch" *Eric S. Raymond, hacker, The Cathedral and the Bazaar, 1997*

The latter cannot be more true [1]. The early ideas of starting a project like this go back into 2007. At that time I have been working on my PhD thesis using one of commercially available process modelling software. It was everything nice and well until I discovered some annoying bugs and lack of certain highly appreciated features. The developers of that proprietary program (as it is a case with all proprietary computer programs) had their own agenda fixing only what they wanted to fix and introducing new features that they anticipated. Although I was able to improve the code and introduce certain features which will help (not only) me - I was helpless. The source code was not available and nobody will ever consider giving it to me to create patches with bugs fixes/new features. Not even if I swear on the holy (c++) bible!!

Very soon the contours of a new process modelling software slowly began to form. It took me a while until I made a definite plan and initial features, and I had to abandon a couple of initial versions...

"Plan to throw one away; you will, anyhow" *Eric S. Raymond, hacker, The Cathedral and the Bazaar, 1997*

Damn you Eric Raymond, interfering with my business again! :-) The new project was officially born early next year - 2008.

[1] However, I do not agree with Eric Raymond and the Open Source Iniative views - they miss the point IMO, but let us leave it beside at the moment.

1.4 Acknowledgements

DAE Tools use the following third party free software libraries (GNU GPL, GNU LGPL, CPL, EPL, BSD or some other type of free/permissive/copy-left licences):

- Sundials IDAS: https://computation.llnl.gov/casc/sundials/main.html
- Boost: http://www.boost.org
- ADOL-C: https://projects.coin-or.org/ADOL-C
- Qt and pyQt4: http://qt.nokia.com, http://www.riverbankcomputing.co.uk/software/pyqt/intro
- Numpy: http://numpy.scipy.orghttp://numpy.scipy.org
- Scipy: http://www.scipy.org
- Blas/Lapack/CLapack: http://www.netlib.org
- Minpack: http://www.netlib.org/minpack
- Atlas: http://math-atlas.sourceforge.net
- Trilinos Amesos: http://trilinos.sandia.gov/packages/amesos
- Trilinos AztecOO: http://trilinos.sandia.gov/packages/aztecoo
- SuperLU/SuperLU_MT: http://crd.lbl.gov/~xiaoye/SuperLU/index.html
- Umfpack: http://www.cise.ufl.edu/research/sparse/umfpack
- MUMPS: http://graal.ens-lyon.fr/MUMPS
- IPOPT: https://projects.coin-or.org/Ipopt
- Bonmin: https://projects.coin-or.org/Bonmin
- NLOPT: http://ab-initio.mit.edu/wiki/index.php/NLopt
- CUSP: http://code.google.com/p/cusp-library

DAE Tools can optionally use the following proprietary software libraries:

- AMD ACML linear solver (pyAmdACML module): http://www.amd.com/acml
- Intel MKL linear solvers (pyIntelMKL and pyIntelPardiso modules): http://software.intel.com/en-us/articles/intel-mkl

Please see the corresponding websites for more details about the licences.

1.5 How to cite

If you use **DAE Tools** in your work then please cite it in the following way: D. Nikolic, DAE Tools process modelling software, 2010. http://www.daetools.com

CHAPTER

TWO

PROGRAMMING PARADIGMS

2.1 The Hybrid approach

In general, there are two types of approaches that can be applied to process modelling: Domain Specific Language approach and a general-purpose programming language approach (such as c/c++, Java or Python). A Domain Specific Language (DSL) is a special-purpose programming or specification language dedicated to a particular problem domain and so designed that it directly supports the key concepts necessary to describe the underlying problems. A domain-specific language is created specifically to solve problems in a particular domain and is usually not intended to be able to solve problems outside it (although that may be technically possible in some cases). In contrast, general-purpose languages are created to solve problems in a wide variety of application domains.

Domain-specific languages are languages with very specific goals in design and implementation and commonly lack low-level functions for filesystem access, interprocess control, and other functions that characterize full-featured programming languages, scripting or otherwise.

A good example of general purpose (multi-domain) domain specific language is Modelica while single-domain (chemical processing industry related) DSLs are gPROMS, Ascend, SpeedUp etc.

DAE Tools approach is a sort of the hybrid approach: it applies general-purpose programming languages such as c++ and Python, but offers a class-hierarchy/API that resembles a syntax of a DSL as much as possible, an access to the low-level functions, large number of standard and third party libraries and uses state of the art free/open-source software components to accomplish particular tasks (calculating derivatives and sensitivities, solving systems of differential and algebraic systems of equations and optimization problems, processing and plotting results etc).

API comparison between Modelica, gPROMS and **DAE Tools**:

```
model BufferTank
  /* Import libs */
  import Modelica.Math.*;

  parameter Real Density;
  parameter Real CrossSectionalAre
  parameter Real Alpha;

  Real HoldUp(start = 0.0);
  Real FlowIn;
  Real FlowOut;
  Real Height;

equation
// Mass balance
  der(HoldUp) = FlowIn - FlowOut;

// Relation betwee liquid level an
  HoldUp = CrossSectionalArea * He

// Relation between pressure drop
  FlowOut = Alpha * sqrt(Height);

end BufferTank;
```

a) Modelica

```
PARAMETER
  Density as Real
  CrossSectionalArea as Real
  Alpha as Real

VARIABLE
  HoldUp as Mass
  FlowIn as Flowrate
  FlowOut as Flowrate
  Height as Length

EQUATION
  # Mass balance
  $HoldUp = FlowIn - FlowOut;

  # Relation betwee liquid level a
  HoldUp = CrossSectionalArea * He

  # Relation between pressure drop
  FlowOut = Alpha * sqrt(Height);
```

b) gPROMS

```
class BufferTank(daeModel):
    def __init__(self, Name, Parent = None, Description = ""):
        daeModel.__init__(self, Name, Parent, Description)

        self.Density = daeParameter("Density", unit(), self)
        self.Area    = daeParameter("Area",    unit(), self)
        self.Alpha   = daeParameter("Alpha",   unit(), self)

        self.HoldUp  = daeVariable("HoldUp",  no_t, self)
        self.FlowIn  = daeVariable("FlowIn",  no_t, self)
        self.FlowOut = daeVariable("FlowOut", no_t, self)
        self.Height  = daeVariable("Height",  no_t, self)

    def DeclareEquations(self):
        # Mass balance
        eq = self.CreateEquation("MassBalance")
        eq.Residual = self.HoldUp.dt() - self.FlowIn() + self.Flow

        # Relation between liquid level and holdup
        eq = self.CreateEquation("LiquidLevelHoldup")
        eq.Residual = self.HoldUp() - self.Area() * self.Height() *

        # Relation between pressure drop and flow
        eq = self.CreateEquation("PressureDropFlow")
        eq.Residual = self.FlowOut() - self.Alpha() * Sqrt(self.Hei
```

c) DAE Tools

DAE Tools provide low-level concepts such as parameters, variables, equations, ports, models, state transition networks, discrete events etc. so that the key concepts from new application domains can be added on top of those low level concepts. For instance, the key modelling concepts from the simulator-independent xml-based domain specific language for modelling of biological neural networks NineML such as neurones, synapses, connectivity patterns, populations of neurones, projections etc. are based on **DAE Tools** low-level concepts.

Side-by-side comparison between the DSL approach and the **DAE Tools** hybrid approach:

DSL Approach	DAE Tools Approach
Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain (direct support for all modelling concepts by the language syntax)	Modelling concepts cannot be expressed directly in the programming language and have to be emulated in the API or in some other way
Clean, concise, ellegant and natural way of building model descriptions: the code can be self documenting	The support for modelling concepts is much more verbose and less elegant; however, DAE Tools can generate XML+MathML based model reports that can be either rendered in XHTML format using XSLT transformations (representing the code documentation) or used as an XML-based model exchange language.
Domain-specific languages could enhance quality, productivity, reliability, maintainability and portability	
DSLs could be and often are simulator independent making a model exchange easier	Programming language dependent; however, a large number of scientific software libraries exposes its functionality to Python via Python wrappers
Cost of designing, implementing, and maintaining a domain-specific language as well as the tools required to develop with it (IDE): a compiler/lexical parser/interpreter must be developed with all burden that comes with it (such as error handling, grammar ambiguities, hidden bugs etc)	A compiler/lexical parser/interpreter is an integral part of the programming language (c++, Python) with a robust error handling, universal grammar and massively tested
Cost of learning a new language vs. its limited applicability: users are required to master a new language (yet another language grammar)	No learning of a new language required (everything can get done in a favourite programming language)
Increased difficulty of integrating the DSL with other components: calling external functions/libraries and interaction with other software is limited by the existence of wrappers around a simulator engine (for instance some scripting languages like Python or javascript)	Calling external functions/libraries is a natural and straightforward Interaction with other software is natural and straightforward
Models usually cannot be created in the runtime/on the fly (or at least not easily) and cannot be modified in the runtime	Models can be created in the runtime/on the fly and easily modified in the runtime
Setting up a simulation (ie. the values of parameters values, initial conditions, initially active states) is embedded in the language and it is typically difficult to do it on the fly or to obtain the values from some other software (for example to chain several software calls where outputs of previous calls represent inputs to the subsequent ones)	Setting up a simulation is done programmaticaly and the initial values can be obtained from some other software in a natural way (chaining several software calls is easy since a large number of libraries make Python wrappers available)
Simulation operating procedures are not flexible; manipulation of model parameters, variables, equations, simulation results etc is limited to only those operations provided by the language	Operating procedures are completely flexible (within the limits of a programming language itself) and a manipulation of model parameters, variables, equations, simulation results etc can be done in any way which a user cosiders suitable for his/her problem
Only the type of results provided by the language/simulator is available; custom processing is usually not possible or if a simulator does provide a way to build extensions it is limited to the functionality made available to them	The results processing can be done in any way which a user considers suitable(again within the limits of a programming language itself)

2.2 The Equation-Oriented approach

In general, three approaches to process modelling exist ([1]):

- Sequential Modular (**SeqM**) approach
- Simultaneous Modular (**SimM**) approach
- Equation-Oriented (**EO**) approach

[1] Morton, W., Equation-Oriented Simulation and Optimization. *Proc. Indian Natl. Sci. Acad.* 2003, 317-357.

The pros & cons of the first two approaches are extensively studied in the literature. Under the **EO** approach we generate and gather together all equations and variables which constitute the model representing the process. The equations are solved simultaneously using a suitable mathematical algorithm (Morton, 2003 [1]). Equation-oriented simulation requires simultaneous solution of a set of differential algebraic equations (**DAE**) which itself requires a solution of a set of nonlinear algebraic equations (**NLAE**) and linear algebraic equations (**LAE**). The Newton's method or some variant of it is almost always used to solve problems described by NLAEs. A brief history of Equation-Oriented solvers and comparison of **SeqM** and **EO** approaches as well as descriptions of the simultaneous modular and equation-oriented methods can be found in Morton, 2003 ([1]). Also a good overview of the equation-oriented approach and its application in gPROMS is given by Barton & Pantelides ([2], [3], [4]).

DAE Tools use the Equation-Oriented approach to process modelling, and the following types of processes can be modelled:

- Lumped and distributed
- Steady-state and dynamic

Problems can be formulated as linear, non-linear, and (partial) differential algebraic systems (of index 1). The most common problems are initial value problems of implicit form. Equations can be ordinary or discontinuous, where discontinuities are automatically handled by the framework. A good overview of discontinuous equations and a procedure for location of equation discontinuities is given by Park & Barton ([5]) and in Sundials IDA (used in DAE Tools).

The main characteristics of the Equation-oriented (acausal) approach:

- Equations are given in an implicit form (as a residual):

  $$F(\dot{x}, x, y, p) = 0$$

  where x and $\dot{x}$ are state variables and their derivatives, y are degrees of freedom and p are parameters.

- Input-Output causality is not fixed

  The benefits are:

  - Increased model re-use
  - Support for different simulation scenarios (based on a single model) by specifying different degrees of freedom. For instance, an equation given in the following form:

    $$x_1 + x_2 + x_3 = 0$$

    can be used to determine either `x1`, `x2` or `x3` depending on what combination of variables is known:

    $$\begin{gathered} x_1 = -x_2 - x_3 \\ \vee \\ x_2 = -x_1 - x_3 \\ \vee \\ x_3 = -x_1 - x_2 \end{gathered}$$

2.3 The Object-Oriented approach

The Object-Oriented approach to process modelling is adopted in **DAE Tools**. The main characteristics of such an approach are:

- Everything is an object
- Models are classes derived from the base daeModel class

[2] Pantelides, C. C., and P. I. Barton, Equation-oriented dynamic simulation current status and future perspectives, *Computers & Chemical Engineering*, vol. 17, no. Supplement 1, pp. 263 - 285, 1993.

[3] Barton, P. I., and C. C. Pantelides, gPROMS - a Combined Discrete/Continuous Modelling Environment for Chemical Processing Systems, *Simulation Series*, vol. 25, no. 3, pp. 25-34, 1993.

[4] Barton, P. I., and C. C. Pantelides, Modeling of combined discrete/continuous processes", *AIChE Journal*, vol. 40, pp. 966-979, 1994.

[5] Park, T., and P. I. Barton, State event location in differential-algebraic models", *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 2, New York, NY, USA, ACM, pp. 137-165, 1996.

- Basically all OO concepts supported by the target language (c++, Python) are allowed, except few exceptions: * Multiple inheritance is supported * Models can be parametrized (using templates in c++) * Derived classes always inherit all declared parameters, variables, equations etc. (polymorphism achieved through virtual functions where the declaration takes place) * All parameters, variables, equations etc. remain public
- Hierarchical model decomposition

2.4 Programming language

DAE Tools core libraries are written in standard c++. However, Python programming language is used as the main modelling language. The main reason for use of Python is (as the authors say): "*Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms*" link.

And: *"Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit-test-debug cycle is incredibly fast*" link. Also, please have a look on a comparison to the other languages. Based on the information available online, and according to the personal experience, the python programs are much shorter and take an order of magnitude less time to develop it. Initially I developed daePlotter module in c++; it took me about one month of part time coding. But, then I moved to python: reimplementing it in PyQt took me just two days (with several new features added), while the code size shrank from 24 cpp modules to four python modules only!

"*Where Python code is typically 3-5 times shorter than equivalent Java code, it is often 5-10 times shorter than equivalent C++ code! Anecdotal evidence suggests that one Python programmer can finish in two months what two C++ programmers can't complete in a year. Python shines as a glue language, used to combine components written in C++*" link. Obviously, not everything can be developed in python; a heavy c++ artillery is still necessary for highly complex projects.

CHAPTER

THREE

ARCHITECTURE

DAE Tools consists of several interdependent components:

- Model
- Simulation
- Optimization
- DAE solver
- LA solver
- NLP solver
- Log
- Data reporter
- Data receiver

The components are located in the following modules:

- pyCore module (Model and Log components)
- pyActivity module (Simulation and Optimization components)
- pyIDAS module (DAE solver component)
- pyDataReporting module (Data reporter and Data receiver components)
- pyUnits module
- Large number of third party linear equation solver modules (LA solver component): pySuperLU, pySuperLU_MT, pyTrilinos
- Large number of third party NLP/MINLP solver modules (NLP solver component): pyIPOPT, pyBONMIN, pyNLOPT

An overview of **DAE Tools** components and their interdepedency is presented in the *DAE Tools architecture*.

3.1 pyCore module

pyCore module defines the key modelling concepts such as:

- Model

  A model of the process is a simplified abstraction of real world process/phenomena describing its most important/driving elements and their interactions. In **DAE Tools** models are created by defining their parameters, distribution domains, variables, equations, and ports.

- Distribution domain

  Domain is a general term used to define an array of different objects (parameters, variables, equations but models and ports as well).

- Parameter

  Parameter can be defined as a time invariant quantity that will not change during a simulation.

- Variable

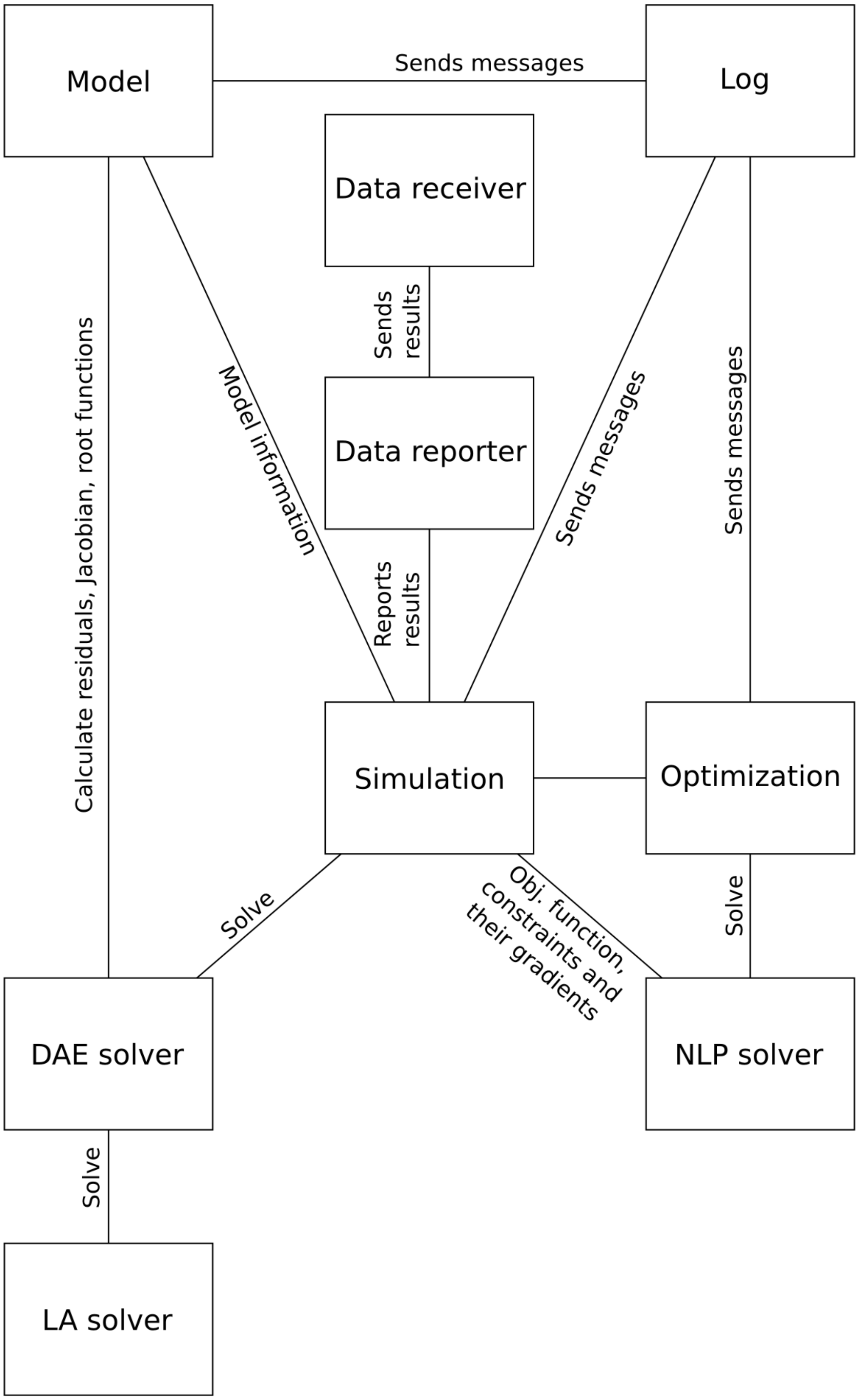

Figure 3.1: **DAE Tools** architecture

Variable can be defined as a time variant quantity, also called a *state variable*.

- Equation

  Equation can be defined as an expression used to calculate a variable value, which can be created by performing basic mathematical operations (+, -, *, /) and functions (such as sin, cos, tan, sqrt, log, ln, exp, pow, abs etc) on parameter and variable values (and time and partial derivatives as well).

- State transition network

  State transition networks are used to model a special type of equations: *discontinuous equation*s. Discontinuous equations are equations that take different forms subject to certain conditions. They are composed of a finite number of *states*.

- State

  States can be defined as a set of actions (in our case a set of equations) under current operating conditions. In addition, every state contains a set of state transitions which describe conditions when the state changes occur.

- OnEvent and OnCondition event handlers

- Port

  Ports are objects used to connect two model instances and exchange continuous information. Like models, they may contain domains, parameters and variables.

- EventPort

  Event ports are objects used to connect two model instances and exchange discrete information (events/messages).

- Log

  Log is defined as an object used to send messages from the various parts of **DAE Tools** framework (messages from solvers or simulation).

3.2 pyActivity module

- Simulation

  Simulation of a process can be considered as the model run for certain input conditions. To define a simulation, several tasks are necessary such as: specifying information about domains and parameters, fixing the degrees of freedom by assigning values to certain variables, setting the initial conditions and many other (setting the initial guesses, absolute tolerances, etc).

- Optimization

  Process optimization can be considered as a process adjustment so as to minimize or maximize a specified goal while satisfying imposed set of constraints. The most common goals are minimizing cost, maximizing throughput, and/or efficiency. In general there are three types of parameters that can be adjusted to affect optimal performance:

  - Equipment optimization
  - Operating procedures
  - Control optimization

3.3 pyDataReporting module

- Data Reporter

  Data reporter is defined as an object used to report the results of a simulation/optimization. They can either keep the results internally (and export them into a file, for instance) or send them via TCP/IP protocol to the **DAE Tools** plotter.

- Data Receiver

Data receiver can be defined as on object which duty is to receive the results from a data reporter. These data can be later plotted or processed in some other ways.

3.4 pyIDAS module

Contains an implementation of the Sundials IDAS DAE solver.

3.5 pyUnits module

Defines two key concepts:

- Unit (SI)

  7 fundamental dimensions (length, mass, time, electrical current, temperature, luminous intensity, amount of substance) * Multiplier * Offset

- Quantity
- Value
- Unit

3.6 NLP/MINLP modules

Contain implementations of various NLP/MINLP solvers:

- IPOPT in the pyIPOPTmodule
- NLOPT in the pyNLOPT module
- BONMIN in the pyBONMIN module

3.7 LA solver modules

Contain implementations of various third party linear equation solvers:

- Trilinos Amesos in the pyTrilinos module
- Trilinos AztecOO in the pyTrilinos module
- SuperLU in the pySuperLU module
- SuperLU_MT in the pySuperLU_MT module
- Intel MKL in the pyTrilinos module

CHAPTER

FOUR

GETTING DAE TOOLS

DAE Tools (pyDAE module) is installed in `daetools` folder within `site-packages` (or `dist-packages`) folder under python (tipically `/usr/local/lib/pythonXY/Lib` or `C:\PythonX.Y\Lib`). The structure of the folders is the following:

- `daetools`
  - `code_generators`
  - `dae_plotter`
  - `dae_simulator`
  - `docs`
  - `examples`
  - `pyDAE`
  - `solvers`
  - `unit_tests`

4.1 System requirements

Supported platforms:

- GNU/Linux (i686, x86_64, arm)
- Windows (32/64 bit)
- MacOS (x86, x86_64)

Supported python versions:

- 2.6 (some older GNU/Linux distributions only)
- 2.7

Supported numpy versions:

- GNU/Linux (1.5, 1.6, 1.7)
- Windows (1.6)
- MacOS (1.6)

Mandatory packages:

- Python (2.7.x): http://www.python.org
- Numpy (1.5.x, 1.6.x, 1.7.x): http://numpy.scipy.org
- Scipy: http://www.scipy.org
- Matplotlib: http://matplotlib.sourceforge.net
- pyQt4 (4.x): http://www.riverbankcomputing.co.uk/software/pyqt

Optional packages (3rd party linear solvers):

- Intel Pardiso (proprietary)
- AMD ACML (proprietary)

For more information on how to install packages please refer to the documentation for the specific library. By default all versions (GNU/Linux, Windows and MacOS) come with the Sundials dense LU and Lapack linear solvers, SuperLU, Trilinos Amesos (with built-in support for KLU, SuperLU and Lapack linear solvers), Trilinos AztecOO (with built-in support for Ifpack and ML preconditioners), NLOPT and IPOPT/BONMIN (with MUMPS linear solver and PORD ordering). Standalone SuperLU_MT is available on GNU/Linux and MacOS versions only. Additional linear solvers: AMD ACML and Intel Pardiso must be downloaded separately and compiled from source since they are not free software.

4.2 Getting the packages

The instalation files can be downloaded from: https://sourceforge.net/projects/daetools/files

Note: From the version 1.2.1 **DAE Tools** use distutils to distribute python packages and extensions.

The naming convention of the installation files:

`daetools-major.minor.build-platform-architecture-python_version.tar.gz`

where `major.minor.build` represents the version (`1.2.1` for instance), `architecture` could be `i686`, `x86_64` or `universal`, and `python_version` can be `py26`, `py27` etc. An example: `daetools-1.2.1-gnu_linux-x86_64-py27.tar.gz` is the version 1.2.1 for 64 bit GNU/Linux with python 2.7.

For the other platforms, architectures and python versions not listed in System requirements daetools must be compiled from the source. The source code can be downloaded either from the subversion tree or from the folder with a particular version (`daetools-1.2.1-source.tar.gz` for instance).

4.3 Installation

4.3.1 GNU/Linux

First install the mandatory packages: python 2.7, numpy 1.5/1.6/1.7, scipy, matplotlib and pyqt4.

In Debian GNU/Linux and derivatives use the Synaptic Package Manager or type the following commands:

```
sudo apt-get install python-numpy python-scipy python-matplotlib python-qt4 mayavi2
```

In Red Hat and derivatives use the package manager or type the following commands:

```
sudo yum install numpy scipy python-matplotlib PyQt4 Mayavi
```

Then unpack the downloaded archive, cd to the `daetools-X.Y.Z` folder and install **DAE Tools** by typing the following shell command:

```
sudo python setup.py install
```

4.3.2 MacOS

First install the mandatory packages: **python 2.7**, **numpy 1.6**, **scipy**, **matplotlib** and **pyqt4**. As a starting point the following links can be used:

- Python 2.7: http://www.python.org/ftp/python/2.7.3/python-2.7.3-macosx10.6.dmg
- Numpy: http://sourceforge.net/projects/numpy/files/NumPy/1.6.2/numpy-1.6.2-py2.7-python.org-macosx10.6.dmg/download
- Scipy: http://sourceforge.net/projects/scipy/files/scipy/0.10.1/scipy-0.10.1-py2.7-python.org-macosx10.6.dmg/download
- Matplotlib: http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.1.0/matplotlib-1.1.0-py2.7-python.org-macosx10.6.dmg/download
- PyQt4: http://www.riverbankcomputing.co.uk/static/Downloads/PyQt4downloadsection

Then unpack the downloaded archive, cd to the `daetools-X.Y.Z` folder and install **DAE Tools** by typing the following shell command:

```
sudo python setup.py install
```

4.3.3 Windows

DAE Tools is compiled and tested on a 32-bit Windows XP and Windows 7. In order to use **DAE Tools** on 64-bit versions of Windows the 32-bit versions of python, pyqt, numpy and scipy packages should be installed. First install the mandatory packages: python 2.7, numpy 1.6, scipy, matplotlib and pyqt4. As a starting point the following links can be used:

- Python 2.7: http://www.python.org/ftp/python/2.7.3/python-2.7.3.msi
- Numpy: http://sourceforge.net/projects/numpy/files/NumPy/1.6.2/numpy-1.6.2-win32-superpack-python2.7.exe/download
- Scipy: http://sourceforge.net/projects/scipy/files/scipy/0.10.1/scipy-0.10.1-win32-superpack-python2.7.exe/download
- Matplotlib: http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.1.0/matplotlib-1.1.0.win32-py2.7.exe/download
- PyQt4: http://www.riverbankcomputing.co.uk/static/Downloads/PyQt4downloadsection

To be able to create 3D plots you need to install Mayavi2 package. Alternatively you can install everything needed through Python(x,y). Finally, install **DAE Tools** by double clicking the file daetools_x.x-x-win32_py27.exe and follow the instructions. To uninstall use the uninstall program in `Start` -> `All Programs` -> `DAE Tools` -> `Uninstall`.

4.4 Compiling from source

To compile the **DAE Tools** the following is needed:

- Installed python, numpy, and scipy modules
- Compiled third party libraries and DAE/LA/NLP solvers: Sundials IDAS, Bonmin, NLopt, Trilinos, SuperLU, SuperLU_MT, Blas/Lapack

All **DAE Tools** modules are developed using the QtCreator/QMake cross-platform integrated development environment. The source code can be downloaded from the SourceForge website or checked out from the DAE Tools subversion repository:

```
svn checkout svn://svn.code.sf.net/p/daetools/code daetools
```

4.4.1 GNU/Linux and MacOS

The easy way

First, install all the necessary dependencies by executing `install_dependencies_linux.sh` shell script located in the `trunk` directory. It will check the OS you are running (currently Debian, Ubuntu, Linux Mint, CentOS and Fedora are supported but other can be easily added) and install all necessary packages needed for **DAE Tools** development.

```
# 'lsb_release' command might be missing on some GNU/Linux platforms
# and has to be installed before proceeding.
# On Debian based systems:
# sudo apt-get install lsb-release
# On red Hat based systems:
# sudo yum install redhat-lsb

cd daetools/trunk
sh install_dependencies_linux.sh
```

Then, compile the third party libraries by executing `compile_libraries_linux.sh` shell script located in the `trunk` directory. The script will download all necessary source archives from the **DAE Tools** SourceForge web-site, unpack them, apply changes and compile them. If all dependencies are installed there should not be problems compiling the libraries.

```
sh compile_libraries_linux.sh
```

Note 1: There is a bug in Sundials IDAS library. When compiling fails, go to the folder `trunk/idas` and change the line 24 (or somewhere around it) in the `Makefile`: `top_builddir = `` to ``top_builddir = ..`

Note 2: There are known problems to compile the older bonmin and trilinos libraries using GNU GCC 4.6. This has been fixed in bonmin 1.5+ and trilinos 10.8+ versions. Therefore, either GCC 4.5 and below or the recent versions of bonmin/trilinos libraries should be used.

Finally, compile the **DAE Tools** libraries and python modules by executing `compile_linux.sh` shell script located in the `trunk` directory. The script accepts one argument specifying projects that should be compiled. Any of the following is accepted: `all`, `core`, `pydae`, `solvers`, `superlu`, `superlu_mt`, `superlu_cuda`, `cusp`, `trilinos`, `bonmin`, `ipopt`, and `nlopt`. If `all` is specified the script will compile `dae`, `superlu`, `superlu_mt`, `trilinos`, `bonmin`, `ipopt`, and `nlopt` projects.

```
sh compile_linux.sh all
# Or for instance:
# sh compile_linux.sh dae superlu nlopt
```

All python extensions should be placed in `trunk/daetools-package/daetools/pyDAE` and `trunk/daetools-package/daetools/solvers` folders. **DAE Tools** can be now installed by using the folowing commands:

```
cd daetools-package
sudo python setup.py install
```

From QtCreator IDE

DAE Tools can also be compiled from within QtCreator IDE. First install dependencies and compile third party libraries (as explained in *The easy way*) and then do the following:

- Do not do the shadow build. Uncheck it (for all projects) and build everything in the release folder
- Choose the right specification file for your platform (usually it is done automatically by the IDE, but double-check it):
- for GNU/Linux use `-spec linux-g++`
- for MacOS use `-spec macx-g++`
- Compile the `dae` project (you can add the additional Make argument `-jN` to speed-up the compilation process, where N is the number of processors plus one; for instance on the quad-core machine you can use `-j5`)
- Compile `SuperLU/SuperLU_MT/SuperLU_CUDA` and `Bonmin/Ipopt` solvers. `SuperLU/SuperLU_MT/SuperLU_CUDA` and `Bonmin/Ipopt` share the same code and the same project file so some hacking is needed. Here are the instructions how to compile them:
- Compiling `libcdaeBONMIN_MINLPSolver.a` and `pyBONMIN.so`:
  - Set `CONFIG += BONMIN` in `BONMIN_MINLPSolver.pro`, run `qmake` and then compile
  - Set `CONFIG += BONMIN` in `pyBONMIN.pro`, run `qmake` and then compile
- Compiling `libcdaeIPOPT_NLPSolver.a` and `pyIPOPT.so`:
  - Set `CONFIG += IPOPT` in `BONMIN_MINLPSolver.pro`, run `qmake` and then compile
  - Set `CONFIG += IPOPT` in `pyBONMIN.pro`, run `qmake` and then compile
- Compiling `libcdaeSuperLU_LASolver.a` and `pySuperLU.so`:
  - Set `CONFIG += SuperLU` in `LA_SuperLU.pro`, run `qmake` and then compile
  - Set `CONFIG += SuperLU` in `pySuperLU.pro`, run `qmake` and then compile
- Compiling `libcdaeSuperLU_MT_LASolver.a` and `pySuperLU_MT.so`:
  - Set `CONFIG += SuperLU_MT` in `LA_SuperLU.pro`, run `qmake` and then compile
  - Set `CONFIG += SuperLU_MT` in `pySuperLU.pro`, run `qmake` and then compile
- Compiling `libcdaeSuperLU_CUDA_LASolver.a` and `pySuperLU_CUDA.so`:
  - Set `CONFIG += SuperLU_CUDA` in `LA_SuperLU.pro`, run `qmake` and then compile

   - Set CONFIG += SuperLU_CUDA in pySuperLU.pro, run qmake and then compile

- Compile the LA_Trilinos_Amesos project

4.4.2 Windows

Necessary tools: QtCreator, Microsoft VC++ and G95 Fortran compiler (Mumps only).

Note: Compiling all third party libraries and **DAE Tools** projects requires a mental gymnastics impossible to describe by any human language so that the pre-compiled libraries are provided in the downloads section (windows libraries).

DAE Tools should be compiled from within QtCreator IDE:

- Unpack the downloaded archive bonmin-trilinos-idas-superlu-nlopt-mumps-g95-msvc-win32.zip into the daetools/trunk folder. All libraries are compiled with MS VC++ 2008 Express edition (the most likely other versions of MS VC++ will also work). Mumps Fortran 95 files are compiled with G95 Fortran compiler.
- Path to libf95.a and libgcc.a libraries should be set in dae.pri config file. For instance, if G95 is installed in c:\g95 set the G95_LIBDIR variable to: G95_LIBDIR = c:\g95\lib\gcc-lib\i686-pc-mingw32\4.1.2
- Follow the instructions for compiling **DAE Tools** described in *From QtCreator IDE* section above.

Note: superlu_mt and superlu_cuda cannot be compiled on Windows at the moment.

DAE Tools can be installed by using the folowing commands:

```
cd daetools-package
sudo python setup.py install
```

CHAPTER

FIVE

GETTING STARTED WITH DAE TOOLS

This chapter gives the basic information about what is needed to develop a model of a process, how to simulate/optimize it and how to obtain and plot the results of a process simulation/optimization. In general, the simulation/optimization of a process consists of three tasks:

1. Modelling of a proces
2. Defining a simulation/optimization
3. Processing the results

5.1 Running tutorials

1. Start `daePlotter`:

   - GNU/Linux:

     Run `Applications/Programming/daePlotter` from the system menu or execute the following shell command:

     `daeplotter`

   - MacOS:

     Execute the following shell command:

     `daeplotter`

   - Windows:

     Run `Start/Programs/DAE Tools/daePlotter` from the Start menu.

   The `daePlotter` main window should appear (given in *daePlotter main window.*)

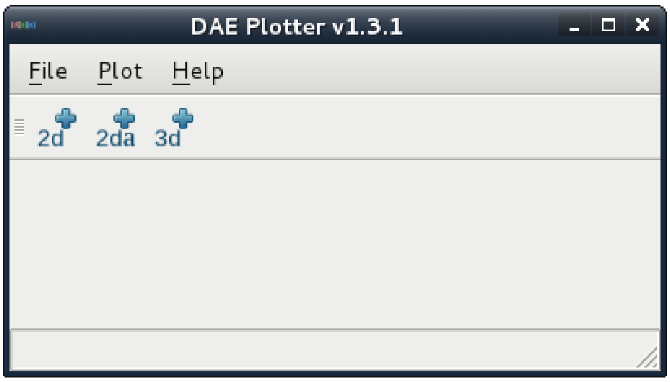

Figure 5.1: daePlotter main window.

2. Start `DAE Tools Examples` program to try some examples:

   - GNU/Linux:

Run `Applications/Programming/DAE Tools Examples` from the system menu or execute the following shell command:

```
daeexamples
```

- MacOS:

  Execute the following shell command:

  ```
  daeexamples
  ```

- Windows:

  Run `Start/Programs/DAE Tools/DAE Tools Examples` from the Start menu.

The main window of `DAE Tools Examples` application is given in *DAE Tools Examples main window* while the output from the simulation run in *A typical optimization output from DAE Tools*. Users can select one of several tutorials, run them, and inspect their source code or model reports. Model reports open in a new window of the system's default web browser (however, only Mozilla Firefox is currently supported because of the MathML rendering issue).

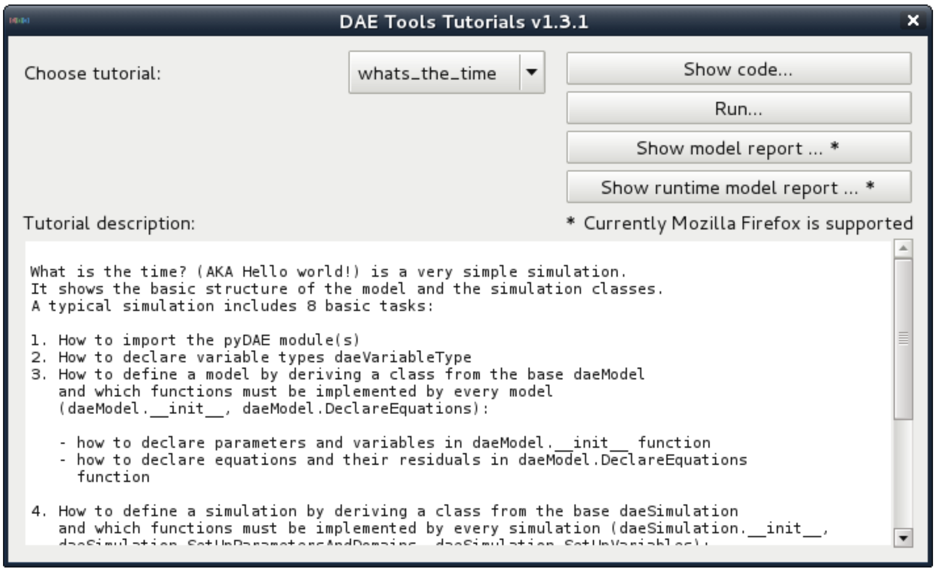

Figure 5.2: DAE Tools Examples main window

Tutorials can also be started from the shell:

```
cd /usr/local/lib/python2.7/dist-packages/daetools/examples
# Or in windows:
# cd C:\PythonX.Y\Lib\site-packages\daetools\examples

python tutorial1.py console
```

The sample output is given in *Shell output from the simulation*:

5.2 Models

5.2.1 Developing a model

In **DAE Tools** models are developed by deriving a new class from the base model class (`daeModel`). The process consists of two steps:

1. Declare all domains, parameters, variables, ports etc.:
   - In **pyDAE** declare and instantiate in the `__init__()` function
   - In **cDAE** declare as class data members and instantiate in the constructor
2. Declare equations and state transition networks in the `DeclareEquations()` function

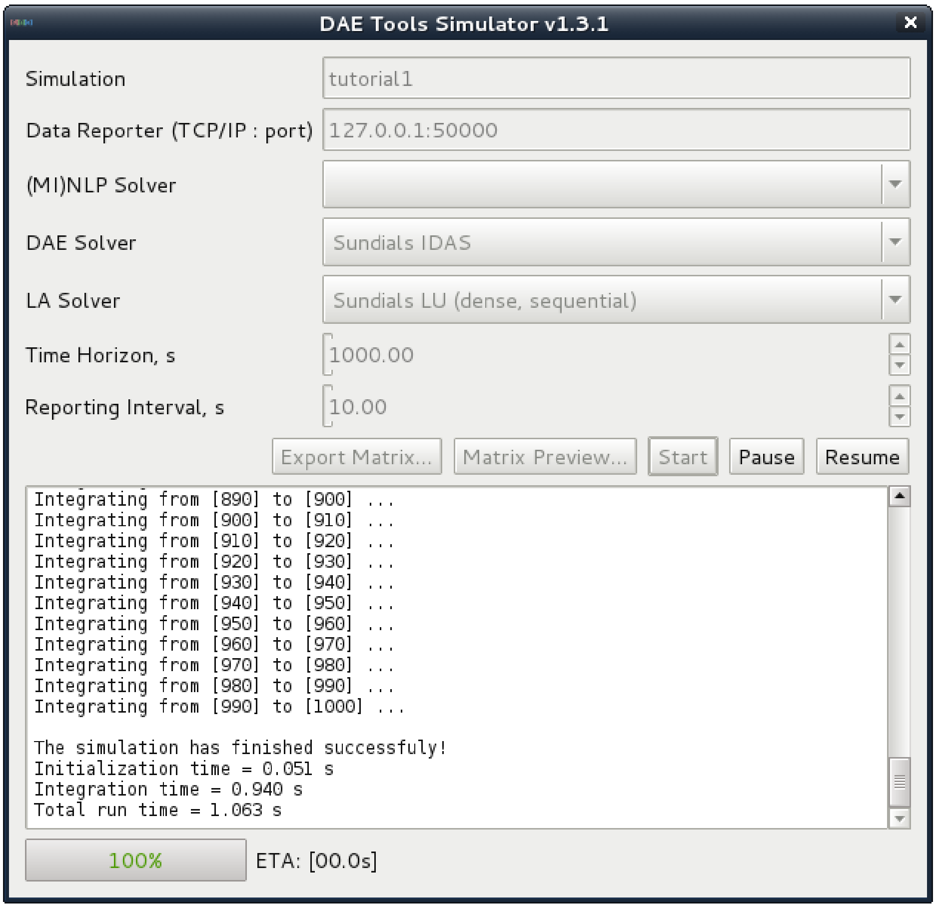

Figure 5.3: A typical optimization output from DAE Tools

An example model developed in **pyDAE** (using python programming language):

```
class myModel(daeModel):
    def __init__(self, name, parent = None, description = ""):
        daeModel.__init__(self, name, parent, description)

        # Declaration/instantiation of domains, parameters, variables, ports, etc:
        self.m     = daeParameter("m",        kg,           self, "Mass of the copper plate")
        self.cp    = daeParameter("c_p",      J/(kg*K),     self, "Specific heat capacity of the plate
        self.alpha = daeParameter("&alpha;", W/((m**2)*K), self, "Heat transfer coefficient")
        self.A     = daeParameter("A",        m**2,         self, "Area of the plate")
        self.Tsurr = daeParameter("T_surr",   K,            self, "Temperature of the surroundings")

        self.Qin   = daeVariable("Q_in",   power_t,       self, "Power of the heater")
        self.T     = daeVariable("T",      temperature_t, self, "Temperature of the plate")

    def DeclareEquations(self):
        # Declaration of equations and state transitions:
        eq = self.CreateEquation("HeatBalance", "Integral heat balance equation")
        eq.Residual = self.m() * self.cp() * self.T.dt() - self.Qin() + self.alpha() * self.A() * (se
```

The same model developed in **cDAE** (using c++ programming language):

```
class myModel : public daeModel
{
public:
    // Declarations of domains, parameters, variables, ports, etc:
    daeParameter mass;
    daeParameter c_p;
    daeParameter alpha;
    daeParameter A;
    daeParameter T_surr;
    daeVariable Q_in;
```

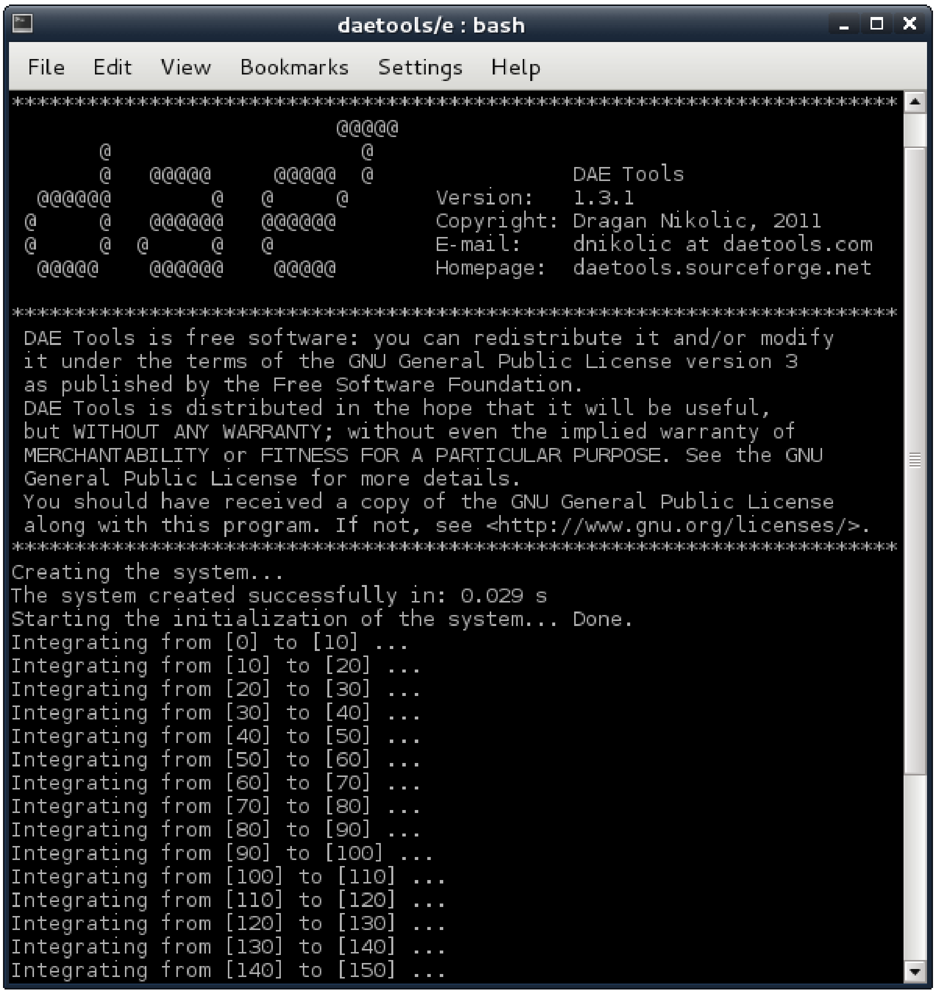

Figure 5.4: Shell output from the simulation

```
    daeVariable T;

public:
    myModel(string strName, daeModel* pParent = NULL, string strDescription = "")
    : daeModel(strName, pParent, strDescription),

    // Instantiation of domains, parameters, variables, ports, etc:
    mass   ("m",        kg,            this, "Mass of the copper plate"),
    c_p    ("c_p",      J/(kg*K),      this, "Specific heat capacity of the plate"),
    alpha  ("&alpha;",  W/((m^2) * K), this, "Heat transfer coefficient"),
    A      ("A",        m ^ 2,         this, "Area of the plate"),
    T_surr ("T_surr",   K,             this, "Temperature of the surroundings"),
    Q_in   ("Q_in",     power_t,       this, "Power of the heater"),
    T      ("T",        temperature_t, this, "Temperature of the plate")
    {
    }

    void DeclareEquations(void)
    {
        // Declaration of equations and state transitions:
        daeEquation* eq = CreateEquation("HeatBalance", "Integral heat balance equation");
        eq->SetResidual( mass() * c_p() * T.dt() - Q_in() + alpha() * A() * (T() - T_surr()) );
    }
};
```

More information about developing models can be found in *pyDAE User Guide* and `pyCore.daeModel`. Also, do not forget to have a look on *Tutorials*.

5.3 Simulation

5.3.1 Setting up a simulation

Definition of a simulation in **DAE Tools** requires the following steps:

1. Deriving a new simulation class from the base simulation class (`daeSimulation`)

- Specification of a model to be simulated
- Setting the values of parameters
- Fixing the degrees of freedom by assigning the values to certain variables
- Setting the initial conditions for differential variables
- Setting the other variables' information: initial guesses, absolute tolerances, etc
- Specifation of an operating procedure. It can be either a simple run for a specified period of time (default) or a complex one where various actions can be taken during the simulation

2. Specify DAE and LA solvers
3. Specify a data reporter and a data receiver, and connect them
4. Set a time horizon, reporting interval, etc
5. Do the initialization of the DAE system
6. Save model report and/or runtime model report (to inspect expanded equations etc)
7. Run the simulation

An example simulation developed in **pyDAE**:

```
class mySimulation(daeSimulation):
    def __init__(self):
        daeSimulation.__init__(self)

        # Set the model to simulate:
        self.m = myModel("myModel", "Description")
```

```
    def SetUpParametersAndDomains(self):
        # Set the parameters values:
        self.m.cp.SetValue(385 * J/(kg*K))
        self.m.m.SetValue(1 * kg)
        self.m.alpha.SetValue(200 * W/((m**2)*K))
        self.m.A.SetValue(0.1 * m**2)
        self.m.Tsurr.SetValue(283 * K)

    def SetUpVariables(self):
        # Set the degrees of freedom, initial conditions, initial guesses, etc.:
        self.m.Qin.AssignValue(1500 * W)
        self.m.T.SetInitialCondition(283 * K)

    def Run(self):
        # A custom operating procedure, if needed.
        # Here we use the default one:
        daeSimulation.Run(self)
```

The same simulation in **cDAE**:

```
class mySimulation : public daeSimulation
{
public:
    myModel m;

public:
    mySimulation(void) : m("myModel", "Description")
    {
        // Set the model to simulate:
        SetModel(&m);
    }

public:
    void SetUpParametersAndDomains(void)
    {
        // Set the parameters values:
        model.c_p.SetValue(385 * J/(kg*K));
        model.mass.SetValue(1 * kg);
        model.alpha.SetValue(200 * W/((m^2)*K));
        model.A.SetValue(0.1 * (m^2));
        model.T_surr.SetValue(283 * K);
    }

    void SetUpVariables(void)
    {
        // Set the degrees of freedom, initial conditions, initial guesses, etc.:
        model.Q_in.AssignValue(1500 * W);
        model.T.SetInitialCondition(283 * K);
    }

    void Run(void)
    {
        // A custom operating procedure, if needed.
        // Here we use the default one:
        daeSimulation::Run();
    }
};
```

Simulations in **pyDAE** can be set-up to run in two modes:

1. From th PyQt4 graphical user interface (**pyDAE** only):

   Here the default log, and data reporter objects will be used, while the user can choose DAE and LA solvers and specify time horizon and reporting interval.

   ```
   # Import modules
   import sys
   from time import localtime, strftime
   ```

```
from PyQt4 import QtCore, QtGui

# Create QtApplication object
app = QtGui.QApplication(sys.argv)

# Create simulation object
sim = mySimulation()

# Report ALL variables in the model
sim.m.SetReportingOn(True)

# Show the daeSimulator window to choose the other information needed for simulation
simulator  = daeSimulator(app, simulation=sim)
simulator.show()

# Execute applications main loop
app.exec_()
```

2. From the shell:

   In **pyDAE**:

```
# Import modules
import sys
from time import localtime, strftime

# Create Log, Solver, DataReporter and Simulation object
log          = daeStdOutLog()
solver       = daeIDAS()
datareporter = daeTCPIPDataReporter()
simulation   = mySimulation()

# Report ALL variables in the model
simulation.m.SetReportingOn(True)

# Set the time horizon (1000 seconds) and the reporting interval (10 seconds)
simulation.SetReportingInterval(10)
simulation.SetTimeHorizon(1000)

# Connect data reporter (use the default TCP/IP connection settings: localhost and 50000 port)
simName = simulation.m.Name + strftime(" [m.%Y %H:%M:%S]", localtime())
if(datareporter.Connect("", simName) == False):
    sys.exit()

# Initialize the simulation
simulation.Initialize(solver, datareporter, log)

# Solve at time = 0 (initialization)
simulation.SolveInitial()

# Run
simulation.Run()

# Clean up
simulation.Finalize()
```

   In **cDAE**:

```
// Create Log, Solver, DataReporter and Simulation object
boost::scoped_ptr<daeSimulation_t>    pSimulation(new mySimulation());
boost::scoped_ptr<daeDataReporter_t>  pDataReporter(daeCreateTCPIPDataReporter());
boost::scoped_ptr<daeIDASolver>       pDAESolver(daeCreateIDASolver());
boost::scoped_ptr<daeLog_t>           pLog(daeCreateStdOutLog());

// Report ALL variables in the model
pSimulation->GetModel()->SetReportingOn(true);

// Set the time horizon (1000 seconds) and the reporting interval (10 seconds)
```

```
    pSimulation->SetReportingInterval(10);
    pSimulation->SetTimeHorizon(1000);

    // Connect data reporter (use the default TCP/IP connection settings: localhost and 50000 port)
    string strName = pSimulation->GetModel()->GetName();
    if(!pDataReporter->Connect("", strName))
        return;

    // Initialize the simulation
    pSimulation->Initialize(pDAESolver.get(), pDataReporter.get(), pLog.get());

    // Solve at time = 0 (initialization)
    pSimulation->SolveInitial();

    // Run
    pSimulation->Run();

    // Clean up
    pSimulation->Finalize();
```

5.3.2 Running a simulation

Simulations are started by executing the following shell commands:

```
cd "directory where simulation file is located"
python mySimulation.py
```

5.4 Optimization

5.4.1 Setting up an optimization

To define an optimization problem it is first necessary to develop a model of the process and to define a simulation (as explained above). Having done these tasks (working model and simulation) the optimization in **DAE Tools** can be defined by specifying the objective function, optimization variables and optimization constraints. It is intentionally chosen to keep simulation and optimization tightly coupled. The optimization problem should be specified in the function `SetUpOptimization()`.

Definition of an optimization in **DAE Tools** requires the following steps:

1. Specify the objective function

- Objective function is defined by specifying its residual (similarly to specifying an equation residual); Internally the framework will create a new variable (V_obj) and a new equation (F_obj).

2. Specify optimization variables

- The optimization variables have to be already defined in the model and their values assigned in the simulation; they can be either non-distributed or distributed.
- Specify a type of optimization variable values. The variables can be `continuous` (floating point values in the given range), `integer` (set of integer values in the given range) or `binary` (integer value: 0 or 1).
- Specify the starting point (within the range)

3. Specify optimization constraints

- Two types of constraints exist in DAE Tools: `equality` and `inequality` constraints To define an `equality` constraint its residual and the value has to be specified; To define an `inequality` constraint its residual, the lower and upper bounds have to be specified; Internally the framework will create a new variable (V_constraint[N]) and a new equation (F_constraint[N]) for each defined constraint, where N is the ordinal number of the constraint.

4. Specify NLP/MINLP solver

- Currently BONMIN MINLP solver and IPOPT and NLOPT solvers are supported (the BONMIN solver internally uses IPOPT to solve NLP problems)

5. Specify DAE and LA solvers
6. Specify a data reporter and a data receiver, and connect them
7. Set a time horizon, reporting interval, etc
8. Set the options of the (MI)NLP solver
9. Initialize the optimization
10. Save model report and/or runtime model report (to inspect expanded equations etc)
11. Run the optimization

`SetUpOptimization()` function should be declared in the simulation class:

In **pyDAE**: .. code-block:: python

> **class mySimulation(daeSimulation):** ...
>
> > **def SetUpOptimization(self):** # Declarations of the obj. function, opt. variables and constraints:

In **cDAE**:

```
class mySimulation : public daeSimulation
{
    ...

    void SetUpOptimization(void)
    {
        // Declarations of the obj. function, opt. variables and constraints:
    }
};
```

Optimizations, like simulations can be set-up to run in two modes:

1. From the PyQt4 graphical user interface (**pyDAE** only)

   Here the default log, and data reporter objects will be used, while the user can choose NLP, DAE and LA solvers and specify time horizon and reporting interval:

   ```
   # Import modules
   import sys
   from time import localtime, strftime
   from PyQt4 import QtCore, QtGui

   # Create QtApplication object
   app = QtGui.QApplication(sys.argv)

   # Create simulation object
   sim = mySimulation()
   nlp = daeBONMIN()

   # Report ALL variables in the model
   sim.m.SetReportingOn(True)

   # Show the daeSimulator window to choose the other information needed for optimization
   simulator = daeSimulator(app, simulation=sim, nlpsolver=nlp)
   simulator.show()

   # Execute applications main loop
   app.exec_()
   ```

2. From the shell:

   In **pyDAE**:

   ```
   # Create Log, NLPSolver, DAESolver, DataReporter, Simulation and Optimization objects
   log          = daePythonStdOutLog()
   daesolver    = daeIDAS()
   nlpsolver    = daeBONMIN()
   datareporter = daeTCPIPDataReporter()
   simulation   = mySimulation()
   ```

```
optimization = daeOptimization()

# Enable reporting of all variables
simulation.m.SetReportingOn(True)

# Set the time horizon and the reporting interval
simulation.ReportingInterval = 10
simulation.TimeHorizon = 100

# Connect data reporter
simName = simulation.m.Name + strftime(" [m.%Y %H:%M:%S]", localtime())
if(datareporter.Connect("", simName) == False):
    sys.exit()

# Initialize the opimization
optimization.Initialize(simulation, nlpsolver, daesolver, datareporter, log)

# Run
optimization.Run()

# Clean up
optimization.Finalize()
```

In **cDAE**:

```
// Create Log, NLPSolver, DAESolver, DataReporter, Simulation and Optimization objects
boost::scoped_ptr<daeSimulation_t>        pSimulation(new mySimulation());
boost::scoped_ptr<daeDataReporter_t>      pDataReporter(daeCreateTCPIPDataReporter());
boost::scoped_ptr<daeIDASolver>           pDAESolver(daeCreateIDASolver());
boost::scoped_ptr<daeLog_t>               pLog(daeCreateStdOutLog());
boost::scoped_ptr<daeNLPSolver_t>         pNLPSolver(new daeBONMINSolver());
boost::scoped_ptr<daeOptimization_t>      pOptimization(new daeOptimization());

// Report ALL variables in the model
pSimulation->GetModel()->SetReportingOn(true);

// Set the time horizon and the reporting interval
pSimulation->SetReportingInterval(10);
pSimulation->SetTimeHorizon(100);

// Connect data reporter
string strName = pSimulation->GetModel()->GetName();
if(!pDataReporter->Connect("", strName))
    return;

// Initialize the optimization
pOptimization->Initialize(pSimulation.get(),
                          pNLPSolver.get(),
                          pDAESolver.get(),
                          pDataReporter.get(),
                          pLog.get());

// Run
pOptimization.Run();

// Clean up
pOptimization.Finalize();
```

More information about simulation can be found in *pyDAE User Guide* and `daeOptimization`. Also, do not forget to have a look on *Tutorials*.

5.4.2 Starting an optimization

Starting the optimization problems is analogous to running a simulation.

5.5 Processing the results

The simulation/optimization results can be easily plotted by using **DAE Plotter** application. It is possible to choose between 2D and 3D plots. After choosing a desired type, a **Choose variable** (given in *Choose variable dialog for a 2D plot*) dialog appears where a variable to be plotted can be selected and information about domains specified - some domains should be fixed while leaving another free by selecting $\star$ from the list (to create a 2D plot one domain must remain free, while for a 3D plot two domains).

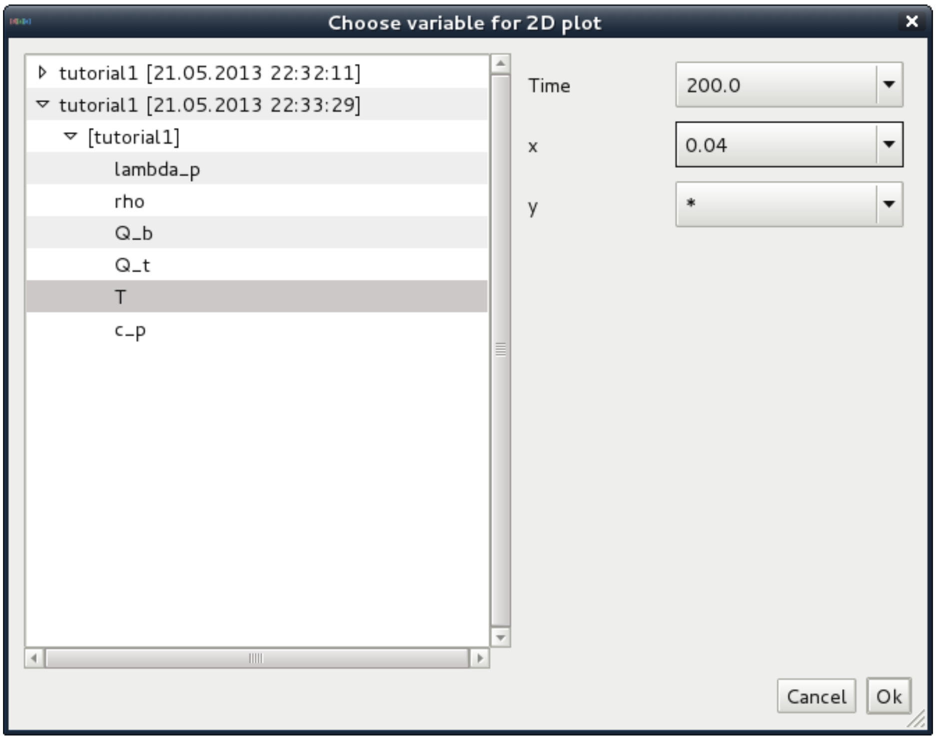

Figure 5.5: Choose variable dialog for a 2D plot

Typical 2D and 3D plots are given in *Example 2D plot (produced by Matplotlib)* and *Example 3D plot (produced by Mayavi2)*.

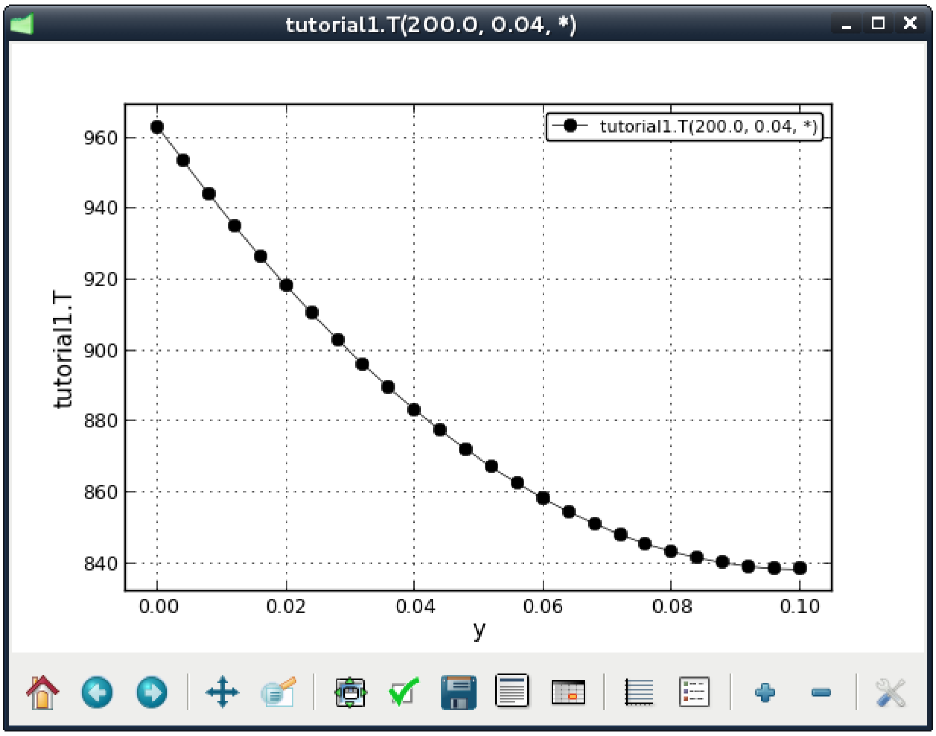

Figure 5.6: Example 2D plot (produced by Matplotlib)

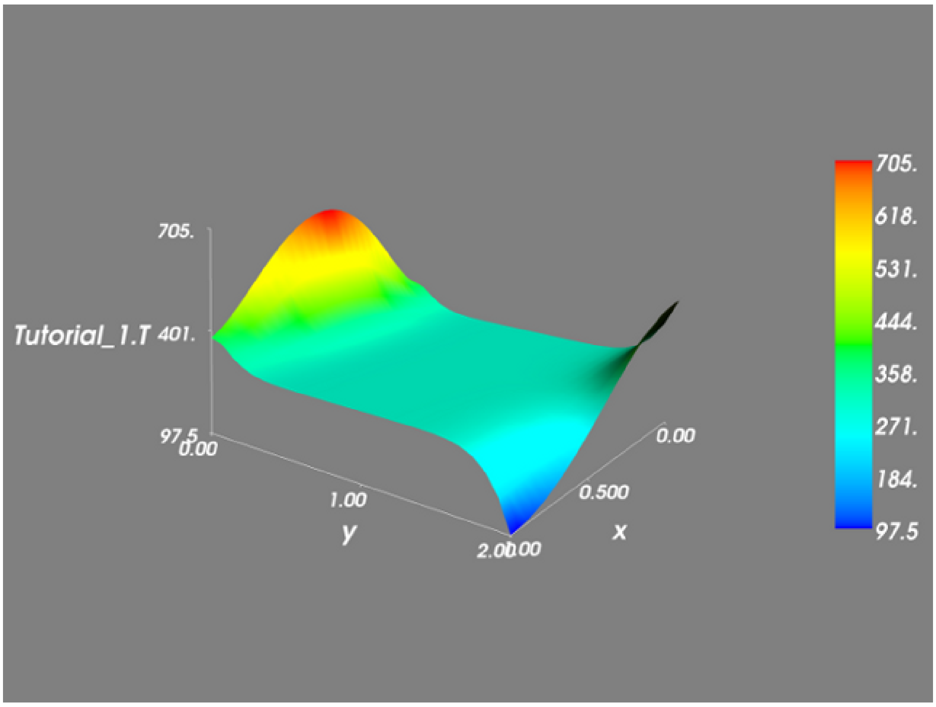

Figure 5.7: Example 3D plot (produced by Mayavi2)

CHAPTER

SIX

PYDAE USER GUIDE

6.1 The main concepts

CHAPTER

SEVEN

PYDAE API REFERENCE

7.1 Module pyCore

7.1.1 Overview

7.1.2 Key modelling concepts

...

Classes

daeVariableType	
daeDomain	
daeParameter	
daeVariable	
daeModel	Base model class.
daeSTN	
daeIF	
daeEquation	
daeState	
daePort	
daeEventPort	
daePortConnection	
daeScalarExternalFunction	
daeVectorExternalFunction	
daeDomainIndex	
daeIndexRange	
daeArrayRange	
daeDEDI	
daeAction	
daeOnConditionActions	
daeOnEventActions	
daeOptimizationVariable	
daeObjectiveFunction	
daeOptimizationConstraint	
daeMeasuredVariable	
daeEquationExecutionInfo	

class `pyCore.`**`daeVariableType`**

Bases: `Boost.Python.instance`

`__init__`((*object*)*self*, (*str*)*name*, (*object*)*units*, (*float*)*lowerBound*, (*float*)*upperBound*, (*float*)*initialGuess*, (*float*)*absTolerance*) → None

`AbsoluteTolerance`

InitialGuess

LowerBound

Name

Units

UpperBound

class `pyCore.`**`daeObject`**

Bases: `Boost.Python.instance`

CanonicalName

Description

GetNameRelativeToParentModel (*(daeObject)self*) → str

GetStrippedName (*(daeObject)self*) → str

GetStrippedNameRelativeToParentModel (*(daeObject)self*) → str

ID

Library

Model

Name

Version

class `pyCore.`**`daeDomain`**

Bases: `pyCore.daeObject`

__init__ (*(object)self, (str)name, (daeModel)parentModel, (object)units*[, *(str)description=''*]) → None

__init__((object)self, (str)name, (daePort)parentPort, (object)units [, (str)description='']) -> None

__getitem__ (*(daeDomain)self, (int)index*) → adouble

__call__ (*(daeDomain)self, (int)index*) → adouble

CreateArray (*(daeDomain)self, (int)noIntervals*) → None

CreateDistributed (*(daeDomain)self, (daeeDiscretizationMethod)discretizationMethod, (int)discretizationOrder, (int)numberOfIntervals, (float)lowerBound, (float)upperBound*) → None

DiscretizationMethod

DiscretizationOrder

LowerBound

NumberOfIntervals

NumberOfPoints

Points

Type

Units

UpperBound

npyPoints

class `pyCore.`**`daeParameter`**

Bases: `pyCore.daeObject`

__init__ (*(object)self, (str)name, (object)units, (daePort)parentPort*[, *(str)description=''*[, *(list)domains=*[ ]]]) → None

__init__((object)self, (str)name, (object)units, (daeModel)parentModel [, (str)description='' [, (list)domains=[]]]) -> None

GetValue ((daeParameter)self[, (int)index1[, ...[, (int)index8]]]) → float
Gets the value of the parameter at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the parameter is distributed on.

GetQuantity ((daeParameter)self[, (int)index1[, ...[, (int)index8]]]) → quantity
Gets the value of the parameter at the specified domain indexes as the `quantity` object (with value and units). How many arguments `index1, ..., index8` are used depends on the number of domains that the parameter is distributed on.

SetValue ((daeParameter)self[, (int)index1[, ...[, (int)index8]]], (float)value) → None
Sets the value of the parameter at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the parameter is distributed on.

SetValue ((daeParameter)self[, (int)index1[, ...[, (int)index8]]], (quantity)value) → None
Sets the value of the parameter at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the parameter is distributed on.

SetValues ((daeParameter)self, (float)values) → None
Sets all values of the parameter.

SetValues ((daeParameter)self, (quantity)values) → None
Sets all values of the parameter.

array ((daeParameter)self[, (object)index1[, ...[, (object)index8]]]) → adouble_array
Gets the array of parameter's values at the specified domain indexes (used to build equation residuals only). How many arguments `index1, ..., index8` are used depends on the number of domains that the parameter is distributed on. Argument types can be one of the following:

- `daeIndexRange` object
- plain integer (to select a single index from a domain)
- python `list` (to select a list of indexes from a domain)
- python `slice` (to select a range of indexes from a domain: start_index, end_index, step)
- character `'*'` (to select all points from a domain)
- integer `-1` (to select all points from a domain)
- empty python list `[]` (to select all points from a domain)

__call__ ((daeParameter)self[, (int)index1[, ...[, (int)index8]]]) → adouble
Gets the value of the parameter at the specified domain indexes (used to build equation residuals only). How many arguments `index1`, ..., `index8` are used depends on the number of domains that the parameter is distributed on.

DistributeOnDomain ((daeParameter)self, (daeDomain)domain) → None

Domains

GetDomainsIndexesMap ((daeParameter)self, (int)indexBase) → dict

NumberOfPoints

ReportingOn

Units

npyValues

class `pyCore.`**daeVariable**
Bases: `pyCore.daeObject`

__init__ ((object)self, (str)name, (daeVariableType)variableType, (daeModel)parentPort[, (str)description='‘[, (list)domains=[ ]]]) → None
__init__((object)self, (str)name, (daeVariableType)variableType, (daePort)parentModel [, (str)description=” [, (list)domains=[]]]) -> None

GetValue ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]) → float
Gets the value of the variable at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on.

GetQuantity (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]]) → quantity
Gets the value of the variable at the specified domain indexes as the `quantity` object (with value and units). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on.

SetValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)value*) → None
Sets the value of the variable at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on.

SetValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)value*) → None
Sets the value of the variable at the specified domain indexes. How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on.

SetValues (*(daeVariable)self*, *(float)values*) → None
Sets all values of the variable.

SetValues (*(daeVariable)self*, *(quantity)values*) → None
Sets all values of the variable.

AssignValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)value*) → None

AssignValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)value*) → None

AssignValues (*(daeVariable)self*, *(float)values*) → None

AssignValues (*(daeVariable)self*, *(quantity)values*) → None

ReAssignValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)value*) → None

ReAssignValue (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)value*) → None

ReAssignValues (*(daeVariable)self*, *(float)values*) → None

ReAssignValues (*(daeVariable)self*, *(quantity)values*) → None

SetInitialCondition (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)initialCondition*) → None

SetInitialCondition (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)initialCondition*) → None

SetInitialConditions (*(daeVariable)self*, *(float)initialConditions*) → None

SetInitialConditions (*(daeVariable)self*, *(quantity)initialConditions*) → None

ReSetInitialCondition (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)initialCondition*) → None

ReSetInitialCondition (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)initialCondition*) → None

ReSetInitialConditions (*(daeVariable)self*, *(float)initialConditions*) → None

ReSetInitialConditions (*(daeVariable)self*, *(quantity)initialConditions*) → None

SetInitialGuess (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(float)initialGuess*) → None

SetInitialGuess (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]], *(quantity)initialGuess*) → None

SetInitialGuesses (*(daeVariable)self*, *(float)initialGuesses*) → None

SetInitialGuesses (*(daeVariable)self*, *(quantity)initialGuesses*) → None

SetAbsoluteTolerances (*(daeVariable)self*, *(float)tolerances*) → None

array (*(daeVariable)self*[, *(object)index1*[, ...[, *(object)index8*]]]) → adouble_array
Gets the array of variable's values at the specified domain indexes (used to build equation residuals only). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on. Argument types are the same as those described in `pyCore.daeParameter.array()`

d_array (*(daeVariable)self*[, *(object)index1*[, ...[, *(object)index8*]]]) → adouble_array
Gets the array of partial derivatives at the specified domain indexes (used to build equation residuals only). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on. Argument types are the same as those described in `pyCore.daeParameter.array()`.

d2_array (*(daeVariable)self*[, *(object)index1*[, ...[, *(object)index8*]]]) → adouble_array
Gets the array of partial derivatives of the second order at the specified domain indexes (used to build equation residuals only). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on. Argument types are the same as those described in `pyCore.daeParameter.array()`.

dt_array (*(daeVariable)self*[, *(object)index1*[, ...[, *(object)index8*]]]) → adouble_array
Gets the array of time derivatives at the specified domain indexes (used to build equation residuals only). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on. Argument types are the same as those described in `pyCore.daeParameter.array()`.

__call__ (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]]) → adouble
Gets the value of the variable at the specified domain indexes (used to build equation residuals only). How many arguments `index1`, ..., `index8` are used depends on the number of domains that the variable is distributed on.

d (*(daeVariable)self*, *(daeDomain)domain*[, *(int)index1*[, ...[, *(int)index8*]]]) → adouble
Gets the partial derivative of the variable at the specified domain indexes (used to build equation residuals only). How many arguments `index1`, ..., `index8` are used depends on the number of domains that the variable is distributed on.

d2 (*(daeVariable)self*, *(daeDomain)domain*[, *(int)index1*[, ...[, *(int)index8*]]]) → adouble
Gets the partial derivative of second order of the variable at the specified domain indexes (used to build equation residuals only). How many arguments `index1`, ..., `index8` are used depends on the number of domains that the variable is distributed on.

dt (*(daeVariable)self*[, *(int)index1*[, ...[, *(int)index8*]]]) → adouble
Gets the time derivative of the variable at the specified domain indexes (used to build equation residuals only). How many arguments `index1`, ..., `index8` are used depends on the number of domains that the variable is distributed on.

DistributeOnDomain (*(daeVariable)self*, *(daeDomain)domain*) → None

Domains

GetDomainsIndexesMap (*(daeVariable)self*, *(int)indexBase*) → dict

NumberOfPoints

OverallIndex

ReportingOn

VariableType

npyIDs

npyValues

class pyCore.**daeModel**
Bases: `pyCore.daeObject`

Base model class.

__init__ (*(object)self*, *(str)name*[, *(daeModel)parentModel=0*[, *(str)description=''*]]) → None :
Constructor...

ComponentArrays
A list of arrays of components in the model.

Components
A list of components in the model.

ConnectEventPorts (*(daeModel)self*, *(daeEventPort)portFrom*, *(daeEventPort)portTo*) → None :
Connects two event ports.

ConnectPorts (*(daeModel)self*, *(daePort)portFrom*, *(daePort)portTo*) → None :
Connects two ports.

CreateEquation (*(daeModel)self*, *(str)name*[, *(str)description=''*[, *(float)scaling=1.0*]]) → daeEquation :
Creates a new equation. Used to add equations to models or states in state transition networks

DeclareEquations (*(daeModel)self*) → None :

User-defined function where all model equations ans state transition networks are declared. Must be always implemented in derived classes.

DeclareEquations((daeModel)self) -> None

Domains
A list of domains in the model.

ELSE (*(daeModel)self*) → None :
Adds the last state to a reversible state transition network.

ELSE_IF (*(daeModel)self*, *(daeCondition)condition*[, *(float)eventTolerance=0.0*]) → None :
Adds a new state to a reversible state transition network.

END_IF (*(daeModel)self*) → None :
Finalises a reversible state transition network.

END_STN (*(daeModel)self*) → None :
.

Equations
A list of equations in the model.

EventPorts
A list of event ports in the model.

Export (*(daeModel)self*, *(str)content*, *(daeeModelLanguage)language*, *(daeModelExportContext)modelExportContext*) → None :
.

ExportObjects (*(daeModel)self*, *(list)objects*, *(daeeModelLanguage)language*) → str :
.

IF (*(daeModel)self*, *(daeCondition)condition*[, *(float)eventTolerance=0.0*]) → None :
Creates a reversible state transition network and adds the first state.

InitialConditionMode
A mode used to calculate initial conditions ...

IsModelDynamic
Boolean flag that determines whether the model is synamic or steady-state.

ModelType
A type of the model ().

ON_CONDITION (*(daeModel)self*, *(daeCondition)condition*[, *(list)switchToStates=[]*[, *(list)setVariableValues=[]*[, *(list)triggerEvents=[]*[, *(list)userDefinedActions=[]*[, *(float)eventTolerance=0.0*]]]]]) → None :
.

ON_EVENT (*(daeModel)self*, *(daeEventPort)eventPort*[, *(list)switchToStates=[]*[, *(list)setVariableValues=[]*[, *(list)triggerEvents=[]*[, *(list)userDefinedActions=*[ ]]]]]]) → None :
.

OnConditionActions
A list of OnCondition actions in the model.

OnEventActions
A list of OnEvent actions in the model.

Parameters
A list of parameters in the model.

PortArrays
A list of arrays of ports in the model.

PortConnections
A list of port connections in the model.

Ports
A list of ports in the model.

STATE (*(daeModel)self*, *(str)stateName*) → daeState :
.

STN (*(daeModel)self, (str)stnName*) → daeSTN :
.

STNs
A list of state transition networks in the model.

SWITCH_TO (*(daeModel)self, (str)targetState, (daeCondition)condition*[, *(float)eventTolerance=0.0*]) → None :
.

SaveModelReport (*(daeModel)self, (str)xmlFilename*) → None :
.

SaveRuntimeModelReport (*(daeModel)self, (str)xmlFilename*) → None :
.

SetReportingOn (*(daeModel)self, (bool)reportingOn*) → None :
Switches the reporting of the model variables/parameters to the data reporter on or off.

Variables
A list of variables in the model.

class `pyCore.`**daeSTN**
Bases: `pyCore.daeObject`

ActiveState

States

class `pyCore.`**daeIF**
Bases: `pyCore.daeSTN`

class `pyCore.`**daeEquation**
Bases: `pyCore.daeObject`

DistributeOnDomain (*(daeEquation)self, (daeDomain)domain, (daeeDomainBounds)domainBounds*) → daeDEDI
DistributeOnDomain((daeEquation)self, (daeDomain)domain, (list)domainIndexes) -> daeDEDI

DistributedEquationDomainInfos

EquationExecutionInfos

EquationType

Residual

Scaling

class `pyCore.`**daeState**
Bases: `pyCore.daeObject`

Equations

NestedSTNs

OnConditionActions

OnEventActions

class `pyCore.`**daePort**
Bases: `pyCore.daeObject`

__init__ (*(object)self, (str)name, (daeePortType)type, (daeModel)parentModel*[, *(str)description=''*]) → None

Domains

Export (*(daePort)self, (str)content, (daeeModelLanguage)language, (daeModelExportContext)context*) → None

Parameters

SetReportingOn (*(daePort)self, (bool)reportingOn*) → None

Type

Variables

class pyCore.**daeEventPort**

Bases: pyCore.daeObject

__init__(*(object)self*, *(str)name*, *(daeePortType)type*, *(daeModel)parentModel*[, *(str)description=''*]) → None

EventData

Events

ReceiveEvent(*(daeEventPort)self*, *(float)data*) → None

RecordEvents

SendEvent(*(daeEventPort)self*, *(float)data*) → None

Type

class pyCore.**daePortConnection**

Bases: pyCore.daeObject

Equations

PortFrom

PortTo

class pyCore.**daeScalarExternalFunction**

Bases: Boost.Python.instance

__init__(*(object)self*, *(str)name*, *(daeModel)parentModel*, *(object)units*, *(dict)arguments*) → None

__call__(*(daeScalarExternalFunction)self*) → adouble

Calculate(*(daeScalarExternalFunction)arg1*, *(tuple)self*, *(dict)values*) → object

Calculate((daeScalarExternalFunction)arg1, (tuple)arg2, (dict)arg3) -> None

Name

class pyCore.**daeVectorExternalFunction**

Bases: Boost.Python.instance

__init__(*(object)self*, *(str)name*, *(daeModel)parentModel*, *(object)units*, *(int)numberOfXXX*, *(dict)arguments*) → None

__call__(*(daeVectorExternalFunction)self*) → adouble_array

Calculate(*(daeVectorExternalFunction)arg1*, *(tuple)self*, *(dict)values*) → list

Calculate((daeVectorExternalFunction)arg1, (tuple)arg2, (dict)arg3) -> None

Name

class pyCore.**daeDomainIndex**

Bases: Boost.Python.instance

__init__(*(object)self*, *(int)index*) → None

__init__((object)self, (daeDEDI)dedi) -> None

__init__((object)self, (daeDEDI)dedi, (int)increment) -> None

__init__((object)self, (daeDomainIndex)domainIndex) -> None

DEDI

Increment

Index

Type

class pyCore.**daeIndexRange**

Bases: Boost.Python.instance

__init__(*(object)self*, *(daeDomain)domain*) → None

__init__((object)arg1, (daeDomain)arg2, (list)arg3) -> object

__init__((object)self, (daeDomain)domain, (int)startIndex, (int)endIndex, (int)step) -> None

Domain

EndIndex

NoPoints

StartIndex

Step

Type

class pyCore.**daeArrayRange**
Bases: Boost.Python.instance

__init__(*(object)self, (daeDomainIndex)domainIndex*) → None
__init__((object)self, (daeIndexRange)indexRange) -> None

DomainIndex

NoPoints

Range

Type

class pyCore.**daeDEDI**
Bases: pyCore.daeObject

__init__()
Raises an exception This class cannot be instantiated from Python

__call__(*(daeDEDI)self*) → adouble

Domain

DomainBounds

DomainPoints

class pyCore.**daeAction**
Bases: pyCore.daeObject

__init__(*(object)self*) → None

Execute(*(daeAction)self*) → None
Execute((daeAction)arg1) -> None

RuntimeNode

STN

SendEventPort

SetupNode

StateTo

Type

VariableWrapper

class pyCore.**daeOnEventActions**
Bases: pyCore.daeObject

Actions

EventPort

Execute(*(daeOnEventActions)arg1*) → None

UserDefinedActions

class pyCore.**daeOnConditionActions**
Bases: pyCore.daeObject

Actions

Condition

Execute(*(daeOnConditionActions)arg1*) → None

UserDefinedActions

class pyCore.**daeOptimizationVariable**

Bases: pyCore.daeOptimizationVariable_t

__init__(*(object)self*) → None

LowerBound

Name

StartingPoint

Type

UpperBound

Value

class pyCore.**daeObjectiveFunction**

Bases: pyCore.daeObjectiveFunction_t

__init__(*(object)self*) → None

Gradients

Name

Residual

Value

class pyCore.**daeOptimizationConstraint**

Bases: pyCore.daeOptimizationConstraint_t

__init__(*(object)self*) → None

Gradients

Name

Residual

Type

Value

class pyCore.**daeMeasuredVariable**

Bases: pyCore.daeMeasuredVariable_t

__init__(*(object)self*) → None

Gradients

Name

Residual

Value

class pyCore.**daeEquationExecutionInfo**

Bases: Boost.Python.instance

EquationType

Node

VariableIndexes

Functions

d
dt
Time

Continued on next page

Table 7.2 – continued from previous page

Constant	Constant((object)value) -> adouble
Array	
Sum	
Product	
Integral	
Average	

pyCore.**d**(*(adouble)arg1*, *(daeDomain)ad*) → adouble

pyCore.**dt**(*(adouble)ad*) → adouble

pyCore.**Time**() → adouble

pyCore.**Constant**(*(float)value*) → adouble
Constant((object)value) -> adouble

pyCore.**Array**(*(list)values*) → adouble_array

pyCore.**Sum**(*(adouble_array)adarray*) → adouble

pyCore.**Product**(*(adouble_array)adarray*) → adouble

pyCore.**Integral**(*(adouble_array)adarray*) → adouble

pyCore.**Average**(*(adouble_array)adarray*) → adouble

7.1.3 Autodifferentiation and equation evaluation tree support

Classes

adouble	Class adouble operates on values/derivatives of domains, parameters and variables.
adouble_array	Class adouble_array operates on arrays of values/derivatives of domains, parameters and variables.
daeCondition	

class pyCore.**adouble**

Bases: Boost.Python.instance

Class adouble operates on values/derivatives of domains, parameters and variables. It supports basic mathematical operators (+, -, , /, *), comparison operators (<, <=, >, >=, ==, !=), and logical operators (and, or, not). Operands can be instances of adouble or float values.

__init__(*(object)self*[, *(float)value=0.0*[, *(float)derivative=0.0*[, *(bool)gatherInfo=False*[, *(adNode)node=0*]]]]) → None

Derivative
Derivative

GatherInfo
Internally used by the framework.

Node
Contains the equation evaluation node.

Value
Value

class pyCore.**adouble_array**

Bases: Boost.Python.instance

Class adouble_array operates on arrays of values/derivatives of domains, parameters and variables. It supports basic mathematical operators (+, -, , /, *). Operands can be instances of adouble_array, adouble or float values.

__init__(*(object)self*[, *(bool)gatherInfo=False*[, *(adNodeArray)node=0*]]) → None

__len__(*(adouble_array)self*) → int :
Returns the size of the adouble_array object.

__getitem__ (*(adouble_array)self*, *(int)index*) → adouble :
Gets an adouble object at the specified index.

__setitem__ (*(adouble_array)self*, *(int)index*, *(adouble)value*) → None :
Sets an adouble object at the specified index.

GatherInfo
Used internally by the framework.

Node
Contains the equation evaluation node.

Resize (*(adouble_array)self*, *(int)newSize*) → None :
Resizes the adouble_array object to the new size.

items (*(object)arg1*) → object :
Returns an iterator over adouble items in adouble_array object.

class `pyCore.`**daeCondition**
Bases: `Boost.Python.instance`

__or__ (*(daeCondition)self*, *(daeCondition)right*) → daeCondition

Logical operator `or`

__and__ (*(daeCondition)self*, *(daeCondition)right*) → daeCondition

Logical operator `and`

EventTolerance

Expressions

RuntimeNode

SetupNode

Mathematical functions

`Exp`	Exp((adouble_array)arg1) -> adouble_array
`Log`	Log((adouble_array)arg1) -> adouble_array
`Log10`	Log10((adouble_array)arg1) -> adouble_array
`Sqrt`	Sqrt((adouble_array)arg1) -> adouble_array
`Sin`	Sin((adouble_array)arg1) -> adouble_array
`Cos`	Cos((adouble_array)arg1) -> adouble_array
`Tan`	Tan((adouble_array)arg1) -> adouble_array
`ASin`	ASin((adouble_array)arg1) -> adouble_array
`ACos`	ACos((adouble_array)arg1) -> adouble_array
`ATan`	ATan((adouble_array)arg1) -> adouble_array
`Sinh`	
`Cosh`	
`Tanh`	
`ASinh`	
`ACosh`	
`ATanh`	
`ATan2`	
`Ceil`	Ceil((adouble_array)arg1) -> adouble_array
`Floor`	Floor((adouble_array)arg1) -> adouble_array
`Pow`	Pow((adouble)arg1, (adouble)arg2) -> adouble
`Abs`	Abs((adouble_array)arg1) -> adouble_array
`Min`	Min((float)arg1, (adouble)arg2) -> adouble
`Max`	Max((float)arg1, (adouble)arg2) -> adouble

`pyCore.`**Exp** (*(adouble)arg1*) → adouble
Exp((adouble_array)arg1) -> adouble_array

`pyCore.Log` ((adouble)arg1) → adouble
Log((adouble_array)arg1) -> adouble_array

`pyCore.Log10` ((adouble)arg1) → adouble
Log10((adouble_array)arg1) -> adouble_array

`pyCore.Sqrt` ((adouble)arg1) → adouble
Sqrt((adouble_array)arg1) -> adouble_array

`pyCore.Sin` ((adouble)arg1) → adouble
Sin((adouble_array)arg1) -> adouble_array

`pyCore.Cos` ((adouble)arg1) → adouble
Cos((adouble_array)arg1) -> adouble_array

`pyCore.Tan` ((adouble)arg1) → adouble
Tan((adouble_array)arg1) -> adouble_array

`pyCore.ASin` ((adouble)arg1) → adouble
ASin((adouble_array)arg1) -> adouble_array

`pyCore.ACos` ((adouble)arg1) → adouble
ACos((adouble_array)arg1) -> adouble_array

`pyCore.ATan` ((adouble)arg1) → adouble
ATan((adouble_array)arg1) -> adouble_array

`pyCore.Sinh` ((adouble)arg1) → adouble

`pyCore.Cosh` ((adouble)arg1) → adouble

`pyCore.Tanh` ((adouble)arg1) → adouble

`pyCore.ASinh` ((adouble)arg1) → adouble

`pyCore.ACosh` ((adouble)arg1) → adouble

`pyCore.ATanh` ((adouble)arg1) → adouble

`pyCore.ATan2` ((adouble)arg1, (adouble)arg2) → adouble

`pyCore.Ceil` ((adouble)arg1) → adouble
Ceil((adouble_array)arg1) -> adouble_array

`pyCore.Floor` ((adouble)arg1) → adouble
Floor((adouble_array)arg1) -> adouble_array

`pyCore.Pow` ((adouble)arg1, (float)arg2) → adouble
Pow((adouble)arg1, (adouble)arg2) -> adouble

Pow((float)arg1, (adouble)arg2) -> adouble

`pyCore.Abs` ((adouble)arg1) → adouble
Abs((adouble_array)arg1) -> adouble_array

`pyCore.Min` ((adouble)arg1, (adouble)arg2) → adouble
Min((float)arg1, (adouble)arg2) -> adouble

Min((adouble)arg1, (float)arg2) -> adouble

Min((adouble_array)adarray) -> adouble

`pyCore.Max` ((adouble)arg1, (adouble)arg2) → adouble
Max((float)arg1, (adouble)arg2) -> adouble

Max((adouble)arg1, (float)arg2) -> adouble

Max((adouble_array)adarray) -> adouble

7.1.4 Auxiliary classes

`daeVariableWrapper`

Continued on next page

Table 7.5 – continued from previous page

daeConfig

class pyCore.**daeVariableWrapper**

Bases: Boost.Python.instance

__init__ (*(object)self, (daeVariable)variable*[, *(str)name=''*]) → None
__init__((object)self, (adouble)ad [, (str)name='']) -> None

DomainIndexes

Name

OverallIndex

Value

Variable

VariableType

class pyCore.**daeConfig**

Bases: Boost.Python.instance

__contains__ (*(daeConfig)self, (object)propertyPath*) → object

__getitem__ (*(daeConfig)self, (object)propertyPath*) → object

__setitem__ (*(daeConfig)self, (object)propertyPath, (object)value*) → None

GetBoolean (*(daeConfig)self, (str)propertyPath*[, *(bool)defaultValue*]) → bool

GetFloat (*(daeConfig)self, (str)propertyPath*[, *(float)defaultValue*]) → float

GetInteger (*(daeConfig)self, (str)propertyPath*[, *(int)defaultValue*]) → int

GetString (*(daeConfig)self, (str)propertyPath*[, *(str)defaultValue*]) → str

Reload (*(daeConfig)self*) → None

SetBoolean (*(daeConfig)self, (str)propertyPath, (bool)value*) → None

SetFloat (*(daeConfig)self, (str)propertyPath, (float)value*) → None

SetInteger (*(daeConfig)self, (str)propertyPath, (int)value*) → None

SetString (*(daeConfig)self, (str)propertyPath, (str)value*) → None

has_key (*(daeConfig)self, (object)propertyPath*) → object

7.1.5 Auxiliary functions

daeGetConfig
daeVersion
daeVersionMajor
daeVersionMinor
daeVersionBuild

pyCore.**daeGetConfig** () → object

pyCore.**daeVersion** ([*(bool)includeBuild=False*]) → str

pyCore.**daeVersionMajor** () → int

pyCore.**daeVersionMinor** () → int

pyCore.**daeVersionBuild** () → int

7.1.6 Enumerations

class `pyCore.`**`daeeDomainType`**

Bases: `Boost.Python.enum`

`eArray` = **pyCore.daeeDomainType.eArray**

`eDTUnknown` = **pyCore.daeeDomainType.eDTUnknown**

`eDistributed` = **pyCore.daeeDomainType.eDistributed**

class `pyCore.`**`daeeParameterType`**

Bases: `Boost.Python.enum`

`eBool` = **pyCore.daeeParameterType.eBool**

`eInteger` = **pyCore.daeeParameterType.eInteger**

`ePTUnknown` = **pyCore.daeeParameterType.ePTUnknown**

`eReal` = **pyCore.daeeParameterType.eReal**

class `pyCore.`**`daeePortType`**

Bases: `Boost.Python.enum`

`eInletPort` = **pyCore.daeePortType.eInletPort**

`eOutletPort` = **pyCore.daeePortType.eOutletPort**

`eUnknownPort` = **pyCore.daeePortType.eUnknownPort**

class `pyCore.`**`daeeDiscretizationMethod`**

Bases: `Boost.Python.enum`

`eBFDM` = **pyCore.daeeDiscretizationMethod.eBFDM**

`eCFDM` = **pyCore.daeeDiscretizationMethod.eCFDM**

`eCustomDM` = **pyCore.daeeDiscretizationMethod.eCustomDM**

`eDMUnknown` = **pyCore.daeeDiscretizationMethod.eDMUnknown**

`eFFDM` = **pyCore.daeeDiscretizationMethod.eFFDM**

class `pyCore.`**`daeeDomainBounds`**

Bases: `Boost.Python.enum`

`eClosedClosed` = **pyCore.daeeDomainBounds.eClosedClosed**

`eClosedOpen` = **pyCore.daeeDomainBounds.eClosedOpen**

eDBUnknown = pyCore.daeeDomainBounds.eDBUnknown

eLowerBound = pyCore.daeeDomainBounds.eLowerBound

eOpenClosed = pyCore.daeeDomainBounds.eOpenClosed

eOpenOpen = pyCore.daeeDomainBounds.eOpenOpen

eUpperBound = pyCore.daeeDomainBounds.eUpperBound

class `pyCore.`**`daeeInitialConditionMode`**
Bases: `Boost.Python.enum`

eAlgebraicValuesProvided = pyCore.daeeInitialConditionMode.eAlgebraicValuesProvided

eDifferentialValuesProvided = pyCore.daeeInitialConditionMode.eDifferentialValuesProvided

eICTUnknown = pyCore.daeeInitialConditionMode.eICTUnknown

eQuasySteadyState = pyCore.daeeInitialConditionMode.eQuasySteadyState

class `pyCore.`**`daeeDomainIndexType`**
Bases: `Boost.Python.enum`

eConstantIndex = pyCore.daeeDomainIndexType.eConstantIndex

eDITUnknown = pyCore.daeeDomainIndexType.eDITUnknown

eDomainIterator = pyCore.daeeDomainIndexType.eDomainIterator

eIncrementedDomainIterator = pyCore.daeeDomainIndexType.eIncrementedDomainIterator

class `pyCore.`**`daeeRangeType`**
Bases: `Boost.Python.enum`

eRaTUnknown = pyCore.daeeRangeType.eRaTUnknown

eRange = pyCore.daeeRangeType.eRange

eRangeDomainIndex = pyCore.daeeRangeType.eRangeDomainIndex

class `pyCore.`**`daeIndexRangeType`**
Bases: `Boost.Python.enum`

eAllPointsInDomain = pyCore.daeIndexRangeType.eAllPointsInDomain

eCustomRange = pyCore.daeIndexRangeType.eCustomRange

eIRTUnknown = pyCore.daeIndexRangeType.eIRTUnknown

eRangeOfIndexes = pyCore.daeIndexRangeType.eRangeOfIndexes

class `pyCore.`**`daeeOptimizationVariableType`**
Bases: `Boost.Python.enum`

eBinaryVariable = pyCore.daeeOptimizationVariableType.eBinaryVariable

eContinuousVariable = pyCore.daeeOptimizationVariableType.eContinuousVariable

eIntegerVariable = pyCore.daeeOptimizationVariableType.eIntegerVariable

class `pyCore.`**`daeeModelLanguage`**
Bases: `Boost.Python.enum`

eCDAE = pyCore.daeeModelLanguage.eCDAE

eMLNone = pyCore.daeeModelLanguage.eMLNone

ePYDAE = pyCore.daeeModelLanguage.ePYDAE

class `pyCore.`**`daeeConstraintType`**
Bases: `Boost.Python.enum`

eEqualityConstraint = pyCore.daeeConstraintType.eEqualityConstraint

eInequalityConstraint = pyCore.daeeConstraintType.eInequalityConstraint

class `pyCore.`**`daeeUnaryFunctions`**
Bases: `Boost.Python.enum`

eAbs = pyCore.daeeUnaryFunctions.eAbs

eArcCos = pyCore.daeeUnaryFunctions.eArcCos

eArcSin = pyCore.daeeUnaryFunctions.eArcSin

eArcTan = pyCore.daeeUnaryFunctions.eArcTan

eCeil = pyCore.daeeUnaryFunctions.eCeil

eCos = pyCore.daeeUnaryFunctions.eCos

eExp = pyCore.daeeUnaryFunctions.eExp

eFloor = pyCore.daeeUnaryFunctions.eFloor

eLn = pyCore.daeeUnaryFunctions.eLn

eLog = pyCore.daeeUnaryFunctions.eLog

eSign = pyCore.daeeUnaryFunctions.eSign

eSin = pyCore.daeeUnaryFunctions.eSin

eSqrt = pyCore.daeeUnaryFunctions.eSqrt

eTan = pyCore.daeeUnaryFunctions.eTan

eUFUnknown = pyCore.daeeUnaryFunctions.eUFUnknown

class `pyCore.`**`daeeBinaryFunctions`**
Bases: `Boost.Python.enum`

eBFUnknown = pyCore.daeeBinaryFunctions.eBFUnknown

eDivide = pyCore.daeeBinaryFunctions.eDivide

eMax = pyCore.daeeBinaryFunctions.eMax

eMin = pyCore.daeeBinaryFunctions.eMin

eMinus = pyCore.daeeBinaryFunctions.eMinus

eMulti = pyCore.daeeBinaryFunctions.eMulti

ePlus = pyCore.daeeBinaryFunctions.ePlus

ePower = pyCore.daeeBinaryFunctions.ePower

class `pyCore.`**`daeeSpecialUnaryFunctions`**
Bases: `Boost.Python.enum`

eAverage = pyCore.daeeSpecialUnaryFunctions.eAverage

eMaxInArray = pyCore.daeeSpecialUnaryFunctions.eMaxInArray

eMinInArray = pyCore.daeeSpecialUnaryFunctions.eMinInArray

eProduct = pyCore.daeeSpecialUnaryFunctions.eProduct

eSUFUnknown = pyCore.daeeSpecialUnaryFunctions.eSUFUnknown

eSum = pyCore.daeeSpecialUnaryFunctions.eSum

class `pyCore.`**`daeeLogicalUnaryOperator`**
Bases: `Boost.Python.enum`

eNot = pyCore.daeeLogicalUnaryOperator.eNot

eUOUnknown = pyCore.daeeLogicalUnaryOperator.eUOUnknown

class `pyCore.`**`daeeLogicalBinaryOperator`**
Bases: `Boost.Python.enum`

eAnd = pyCore.daeeLogicalBinaryOperator.eAnd

eBOUnknown = pyCore.daeeLogicalBinaryOperator.eBOUnknown

eOr = pyCore.daeeLogicalBinaryOperator.eOr

class pyCore.**daeeConditionType**
Bases: Boost.Python.enum

eCTUnknown = pyCore.daeeConditionType.eCTUnknown

eEQ = pyCore.daeeConditionType.eEQ

eGT = pyCore.daeeConditionType.eGT

eGTEQ = pyCore.daeeConditionType.eGTEQ

eLT = pyCore.daeeConditionType.eLT

eLTEQ = pyCore.daeeConditionType.eLTEQ

eNotEQ = pyCore.daeeConditionType.eNotEQ

class pyCore.**daeeActionType**
Bases: Boost.Python.enum

eChangeState = pyCore.daeeActionType.eChangeState

eReAssignOrReInitializeVariable = pyCore.daeeActionType.eReAssignOrReInitializeVariable

eSendEvent = pyCore.daeeActionType.eSendEvent

eUnknownAction = pyCore.daeeActionType.eUnknownAction

eUserDefinedAction = pyCore.daeeActionType.eUserDefinedAction

class pyCore.**daeeEquationType**
Bases: Boost.Python.enum

eAlgebraic = pyCore.daeeEquationType.eAlgebraic

eETUnknown = pyCore.daeeEquationType.eETUnknown

eExplicitODE = pyCore.daeeEquationType.eExplicitODE

eImplicitODE = pyCore.daeeEquationType.eImplicitODE

class pyCore.**daeeModelType**
Bases: Boost.Python.enum

eDAE = pyCore.daeeModelType.eDAE

eMTUnknown = pyCore.daeeModelType.eMTUnknown

eODE = pyCore.daeeModelType.eODE

eSteadyState = pyCore.daeeModelType.eSteadyState

7.1.7 Global constants

cnAlgebraic	int(x[, base]) -> integer
cnDifferential	int(x[, base]) -> integer
cnAssigned	int(x[, base]) -> integer

pyCore.**cnAlgebraic = 0**
int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating point argument will be truncated towards zero (this does not include a string representation of a floating point number!) When converting a string, use the optional base. It is an error to supply a base when converting a non-string. If base is zero, the proper base is guessed based on the string content. If the argument is outside the integer range a long object will be returned instead.

pyCore.**cnDifferential = 1**
int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating point argument will be truncated towards zero (this does not include a string representation of a floating point number!) When converting a string, use the optional base. It is an error to supply a base when converting a non-string. If base is zero, the proper base is guessed based on the string content. If the argument is outside the integer range a long object will be returned instead.

pyCore.**cnAssigned = 2**

int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating point argument will be truncated towards zero (this does not include a string representation of a floating point number!) When converting a string, use the optional base. It is an error to supply a base when converting a non-string. If base is zero, the proper base is guessed based on the string content. If the argument is outside the integer range a long object will be returned instead.

7.2 Message logs

7.2.1 Overview

Classes

daeLog_t
daeBaseLog
daeDelegateLog
daeFileLog
daeStdOutLog
daeTCPIPLog
daeTCPIPLogServer

class pyCore.**daeLog_t**

Bases: Boost.Python.instance

__init__()

Raises an exception This class cannot be instantiated from Python

DecreaseIndent(*(daeLog_t)self, (int)offset*) → None

DecreaseIndent((daeLog_t)arg1, (int)arg2) -> None

ETA

Enabled

IncreaseIndent(*(daeLog_t)self, (int)offset*) → None

IncreaseIndent((daeLog_t)arg1, (int)arg2) -> None

Indent

IndentString

JoinMessages(*(daeLog_t)self*[, *(str)delimiter='n'*]) → str

JoinMessages((daeLog_t)arg1, (str)arg2) -> None

Message(*(daeLog_t)self, (str)message, (int)severity*) → None

Message((daeLog_t)arg1, (str)arg2, (int)arg3) -> None

PercentageDone

PrintProgress

Progress

class pyCore.**daeBaseLog**

Bases: pyCore.daeLog_t

__init__(*(object)self*) → None

DecreaseIndent(*(daeBaseLog)self, (int)offset*) → None

IncreaseIndent(*(daeBaseLog)self, (int)offset*) → None

Message(*(daeBaseLog)self, (str)message, (int)severity*) → None

Message((daeBaseLog)self, (str)message, (int)severity) -> None

SetProgress (*(daeBaseLog)self*, *(float)progress*) → None
SetProgress((daeBaseLog)self, (float)progress) -> None

class pyCore.**daeDelegateLog**
Bases: pyCore.daeBaseLog

__init__ (*(object)self*) → None

AddLog (*(daeDelegateLog)self*, *(daeLog_t)log*) → None

Logs

Message (*(daeDelegateLog)self*, *(str)message*, *(int)severity*) → None
Message((daeDelegateLog)self, (str)message, (int)severity) -> None

class pyCore.**daeFileLog**
Bases: pyCore.daeBaseLog

__init__ (*(object)self*, *(str)filename*) → None

Message (*(daeFileLog)self*, *(str)message*, *(int)severity*) → None
Message((daeFileLog)self, (str)message, (int)severity) -> None

class pyCore.**daeStdOutLog**
Bases: pyCore.daeBaseLog

__init__ (*(object)self*) → None

Message (*(daeStdOutLog)self*, *(str)message*, *(int)severity*) → None
Message((daeStdOutLog)self, (str)message, (int)severity) -> None

class pyCore.**daeTCPIPLog**
Bases: pyCore.daeBaseLog

__init__ (*(object)self*) → None

Connect (*(daeTCPIPLog)self*, *(str)tcpipAddress*, *(int)port*) → bool

Disconnect (*(daeTCPIPLog)self*) → bool

IsConnected (*(daeTCPIPLog)self*) → bool

Message (*(daeTCPIPLog)self*, *(str)message*, *(int)severity*) → None
Message((daeTCPIPLog)self, (str)message, (int)severity) -> None

class pyCore.**daeTCPIPLogServer**
Bases: Boost.Python.instance

__init__ (*(object)self*, *(int)port*) → None

MessageReceived (*(daeTCPIPLogServer)self*, *(str)message*) → None
MessageReceived((daeTCPIPLogServer)self, (str)message) -> None

Port

Start (*(daeTCPIPLogServer)self*) → None

Stop (*(daeTCPIPLogServer)self*) → None

Extra logs

class daetools.pyDAE.logs.**daePythonStdOutLog**
Bases: pyCore.daeStdOutLog

Message (*message*, *severity*)

class daetools.dae_simulator.simulator.**daeTextEditLog** (*TextEdit*, *ProgressBar*, *ProgressLabel*, *App*)
Bases: pyCore.daeBaseLog

Message (*message*, *severity*)

SetProgress (*progress*)

7.3 Module pyActivity

7.3.1 Overview

7.3.2 Classes

daeSimulation
daeOptimization

class pyActivity.**daeSimulation**
Bases: pyActivity.daeSimulation_t

Initialization methods

__init__(*(object)self*) → None

Initialize(*(daeSimulation)self*, *(object)daeSolver*, *(object)dataReporter*, *(object)log*[, *(bool)calculateSensitivities=False*]) → None

SolveInitial(*(daeSimulation)self*) → None

m

model

Model

DAESolver

Log

DataReporter

AbsoluteTolerances

RelativeTolerance

TotalNumberOfVariables

NumberOfEquations

Loading/storing the initialization data

LoadInitializationValues(*(daeSimulation)self*, *(str)filename*) → None

StoreInitializationValues(*(daeSimulation)self*, *(str)filename*) → None

InitialValues

InitialDerivatives

Clean up methods

CleanUpSetupData(*(daeSimulation)self*) → None

Finalize(*(daeSimulation)self*) → None

Simulation setup methods

SetUpParametersAndDomains(*(daeSimulation)self*) → None

SetUpVariables(*(daeSimulation)self*) → None

Optimization setup methods

SetUpOptimization (*(daeSimulation)self*) → None

CreateInequalityConstraint (*(daeSimulation)self*, *(str)description*) → object

CreateEqualityConstraint (*(daeSimulation)self*, *(str)description*) → object

SetContinuousOptimizationVariable (*(daeSimulation)self*, *(object)variable*, *(float)lowerBound*, *(float)upperBound*, *(float)defaultValue*) → object
SetContinuousOptimizationVariable((daeSimulation)self, (object)ad, (float)lowerBound, (float)upperBound, (float)defaultValue) -> object

SetIntegerOptimizationVariable (*(daeSimulation)self*, *(object)variable*, *(int)lowerBound*, *(int)upperBound*, *(int)defaultValue*) → object
SetIntegerOptimizationVariable((daeSimulation)self, (object)ad, (int)lowerBound, (int)upperBound, (int)defaultValue) -> object

SetBinaryOptimizationVariable (*(daeSimulation)self*, *(object)variable*, *(bool)defaultValue*) → object
SetBinaryOptimizationVariable((daeSimulation)self, (object)ad, (bool)defaultValue) -> object

OptimizationVariables

Constraints

ObjectiveFunction

Parameter estimation setup methods

SetUpParameterEstimation (*(daeSimulation)self*) → None

SetMeasuredVariable (*(daeSimulation)self*, *(object)variable*) → object
SetMeasuredVariable((daeSimulation)self, (object)ad) -> object

SetInputVariable (*(daeSimulation)self*, *(object)variable*) → object
SetInputVariable((daeSimulation)self, (object)ad) -> object

SetModelParameter (*(daeSimulation)self*, *(object)variable*, *(float)lowerBound*, *(float)upperBound*, *(float)defaultValue*) → object
SetModelParameter((daeSimulation)self, (object)ad, (float)lowerBound, (float)upperBound, (float)defaultValue) -> object

InputVariables

MeasuredVariables

ModelParameters

Parameter estimation setup methods

SetUpSensitivityAnalysis (*(daeSimulation)self*) → None

Operating procedures methods

Run (*(daeSimulation)self*) → None

ReRun (*(daeSimulation)self*) → None

Pause (*(daeSimulation)self*) → None

Resume (*(daeSimulation)self*) → None

ActivityAction

Integrate (*(daeSimulation)self*, *(daeeStopCriterion)stopCriterion*[, *(bool)reportDataAroundDiscontinuities=True*]) → float

IntegrateForTimeInterval (*(daeSimulation)self*, *(float)timeInterval*[, *(bool)reportDataAroundDiscontinuities=True*]) → float

IntegrateUntilTime (*(daeSimulation)self*, *(float)time*, *(daeStopCriterion)stopCriterion*[, *(bool)reportDataAroundDiscontinuities=True*]) → float

Reinitialize (*(daeSimulation)self*) → None

Reset (*(daeSimulation)self*) → None

CurrentTime

TimeHorizon

ReportingInterval

NextReportingTime

ReportingTimes

Data reporting methods

RegisterData (*(daeSimulation)self*, *(str)iteration*) → None

ReportData (*(daeSimulation)self*, *(float)currentTime*) → None

Various information

IndexMappings

InitialConditionMode

SimulationMode

VariableTypes

LastSatisfiedCondition

class pyActivity.**daeOptimization**

Bases: pyActivity.daeOptimization_t

__init__ (*(object)self*) → None

Initialize (*(daeOptimization)self*, *(daeSimulation_t)simulation*, *(object)nlpSolver*, *(object)daeSolver*, *(object)dataReporter*, *(object)log*) → None

Run (*(daeOptimization)self*) → None

Finalize (*(daeOptimization)self*) → None

7.3.3 Enumerations

daeeStopCriterion
daeeActivityAction
daeeSimulationMode

class pyActivity.**daeeStopCriterion**

Bases: Boost.Python.enum

eDoNotStopAtDiscontinuity = **pyActivity.daeeStopCriterion.eDoNotStopAtDiscontinuity**

eStopAtModelDiscontinuity = **pyActivity.daeeStopCriterion.eStopAtModelDiscontinuity**

class pyActivity.**daeeActivityAction**

Bases: Boost.Python.enum

eAAUnknown = **pyActivity.daeeActivityAction.eAAUnknown**

ePauseActivity = **pyActivity.daeeActivityAction.ePauseActivity**

eRunActivity = **pyActivity.daeeActivityAction.eRunActivity**

class pyActivity.**daeeSimulationMode**
Bases: `Boost.Python.enum`

eOptimization = pyActivity.daeeSimulationMode.eOptimization

eParameterEstimation = pyActivity.daeeSimulationMode.eParameterEstimation

eSimulation = pyActivity.daeeSimulationMode.eSimulation

7.4 Module pyDataReporting

7.4.1 Overview

7.4.2 DataReporter classes

daeDataReporter_t
daeDataReporterLocal
daeNoOpDataReporter
daeDataReporterFile
daeTEXTFileDataReporter
daeBlackHoleDataReporter
daeDelegateDataReporter

class pyDataReporting.**daeDataReporter_t**
Bases: `Boost.Python.instance`

Connect (*(daeDataReporter_t)self*, *(str)connectionString*, *(str)processName*) → bool

Disconnect (*(daeDataReporter_t)self*) → bool

IsConnected (*(daeDataReporter_t)self*) → bool

StartRegistration (*(daeDataReporter_t)self*) → bool

RegisterDomain (*(daeDataReporter_t)self*, *(daeDataReporterDomain)domain*) → bool

RegisterVariable (*(daeDataReporter_t)self*, *(daeDataReporterVariable)variable*) → bool

EndRegistration (*(daeDataReporter_t)self*) → bool

StartNewResultSet (*(daeDataReporter_t)self*, *(Float)time*) → bool

SendVariable (*(daeDataReporter_t)self*, *(daeDataReporterVariableValue)variableValue*) → bool

EndOfData (*(daeDataReporter_t)self*) → bool

Data reporters that *do not* send data to a data receiver and keep data locally (*local data reporters*)

class pyDataReporting.**daeDataReporterLocal**
Bases: `pyDataReporting.daeDataReporter_t`

Process

dictDomains

dictVariables

class pyDataReporting.**daeNoOpDataReporter**
Bases: `pyDataReporting.daeDataReporterLocal`

class pyDataReporting.**daeDataReporterFile**
Bases: `pyDataReporting.daeDataReporterLocal`

WriteDataToFile (*(daeDataReporterFile)self*) → None

class pyDataReporting.**daeTEXTFileDataReporter**
Bases: `pyDataReporting.daeDataReporterFile`

WriteDataToFile ((*daeTEXTFileDataReporter)self*) → None

Third-party local data reporters

daePlotDataReporter()	
daeMatlabMATFileDataReporter()	

class daetools.pyDAE.data_reporters.**daePlotDataReporter**
Bases: pyDataReporting.daeDataReporterLocal

Plot (**args*, ***kwargs*)
args can be either:

1.Instances of daeVariable, or

2.Lists of daeVariable instances, or

3.A mixture of both.

Each arg will get its own subplot. The subplots are all automatically arranged such that the resulting figure is as square-like as possible. You can however override the shape by supplying figRows and figCols as keyword args.

Basic Example:

```
# Create Log, Solver, DataReporter and Simulation object
log = daePythonStdOutLog()
daesolve = daeIDAS()
from daetools.pyDAE.data_reporters import daePlotDataReporter
datareporter = daePlotDataReporter()
simulation = simTutorial()

simulation.m.SetReportingOn(True)
simulation.ReportingInterval = 20
simulation.TimeHorizon = 500

simName = simulation.m.Name + strftime(" [%d.%m.%Y %H:%M:%S]", localtime())
if(datareporter.Connect("", simName) == False):
    sys.exit()

simulation.Initialize(daesolver, datareporter, log)

simulation.m.SaveModelReport(simulation.m.Name + ".xml")
simulation.m.SaveRuntimeModelReport(simulation.m.Name + "-rt.xml")

simulation.SolveInitial()
simulation.Run()

simulation.Finalize()
datareporter.Plot(
    simulation.m.Ci,                          # Subplot 1
    [simulation.m.L, simulation.m.event],     # Subplot 2 (2 sets)
    simulation.m.Vp,                          # Subplot 3
    [simulation.m.L, simulation.m.Vp]         # Subplot 4 (2 sets)
    )
```

class daetools.pyDAE.data_reporters.**daeMatlabMATFileDataReporter**
Bases: pyDataReporting.daeDataReporterLocal

WriteDataToFile ()

Data reporters that *do* send data to a data receiver (*remote data reporters*)

class pyDataReporting.**daeDataReporterRemote**
Bases: pyDataReporting.daeDataReporter_t

SendMessage (*(daeDataReporterRemote)self*, *(str)message*) → bool

class pyDataReporting.**daeTCPIPDataReporter**
Bases: pyDataReporting.daeDataReporterRemote

SendMessage (*(daeTCPIPDataReporter)self*, *(str)message*) → bool

Special-purpose data reporters

class pyDataReporting.**daeBlackHoleDataReporter**
Bases: pyDataReporting.daeDataReporter_t

Data reporter that does not process any data and all function calls simply return `True`. Could be used when no results from the simulation are needed.

class pyDataReporting.**daeDelegateDataReporter**
Bases: pyDataReporting.daeDataReporter_t

A container-like data reporter, which does not process any data but forwards (delegates) all function calls (`Disconnect()`, `IsConnected()`, `StartRegistration()`, `RegisterDomain()`, `RegisterVariable()`, `EndRegistration()`, `StartNewResultSet()`, `SendVariable()`, `EndOfData()`) to data reporters in the containing list of data reporters. Data reporters can be added by using the `AddDataReporter()`. The list of containing data reporters is in the `DataReporters` attribute.

Connect (*(daeDataReporter_t)self*, *(str)connectionString*, *(str)processName*) → Boolean
Does nothing. Always returns `True`.

AddDataReporter (*(daeDelegateDataReporter)self*, *(daeDataReporter_t)dataReporter*) → None

DataReporters

DataReporter data-containers

daeDataReporterDomain
daeDataReporterVariable
daeDataReporterVariableValue

class pyDataReporting.**daeDataReporterDomain**
Bases: Boost.Python.instance

__getitem__ (*(daeDataReporterDomain)self*, *(int)index*) → float

__setitem__ (*(daeDataReporterDomain)self*, *(int)index*, *(float)value*) → None

Name

NumberOfPoints

Points

Type

class pyDataReporting.**daeDataReporterVariable**
Bases: Boost.Python.instance

AddDomain (*(daeDataReporterVariable)self*, *(str)domainName*) → None

Domains

Name

NumberOfDomains

NumberOfPoints

class pyDataReporting.**daeDataReporterVariableValue**
Bases: Boost.Python.instance

__getitem__ (*(daeDataReporterVariableValue)self*, *(int)index*) → float

__setitem__ (*(daeDataReporterVariableValue)self*, *(int)index*, *(float)value*) → None

Name

NumberOfPoints

Values

7.4.3 DataReceiver classes

daeDataReceiver_t
daeTCPIPDataReceiver
daeTCPIPDataReceiverServer

class pyDataReporting.**daeDataReceiver_t**
Bases: Boost.Python.instance

Start (*(daeDataReceiver_t)self*) → bool

Stop (*(daeDataReceiver_t)self*) → bool

Process

class pyDataReporting.**daeTCPIPDataReceiver**
Bases: pyDataReporting.daeDataReceiver_t

Start (*(daeTCPIPDataReceiver)self*) → bool

Stop (*(daeTCPIPDataReceiver)self*) → bool

Process

class pyDataReporting.**daeTCPIPDataReceiverServer**
Bases: Boost.Python.instance

DataReceivers

IsConnected (*(daeTCPIPDataReceiverServer)self*) → bool

Start (*(daeTCPIPDataReceiverServer)self*) → None

Stop (*(daeTCPIPDataReceiverServer)self*) → None

DataReceiver data-containers

daeDataReceiverDomain
daeDataReceiverVariable
daeDataReceiverVariableValue
daeDataReceiverProcess

class pyDataReporting.**daeDataReceiverDomain**
Bases: Boost.Python.instance

__getitem__ (*(daeDataReceiverDomain)self*, *(int)index*) → float

__setitem__ (*(daeDataReceiverDomain)self*, *(int)index*, *(float)value*) → None

Name

NumberOfPoints

Points

Type

class pyDataReporting.**daeDataReceiverVariable**
Bases: Boost.Python.instance

AddDomain (*(daeDataReceiverVariable)self*, *(daeDataReceiverDomain)domain*) → None

AddVariableValue (*(daeDataReceiverVariable)self*, *(daeDataReceiverVariableValue)variableValue*) → None

Domains

Name

NumberOfPoints

TimeValues

Values

class pyDataReporting.**daeDataReceiverVariableValue**
Bases: Boost.Python.instance

__getitem__ (*(daeDataReceiverVariableValue)self*, *(int)index*) → float

__setitem__ (*(daeDataReceiverVariableValue)self*, *(int)index*, *(float)value*) → None

Time

class pyDataReporting.**daeDataReceiverProcess**
Bases: Boost.Python.instance

Domains

FindVariable (*(daeDataReceiverProcess)self*, *(str)variableName*) → daeDataReceiverVariable

Name

RegisterDomain (*(daeDataReceiverProcess)self*, *(daeDataReceiverDomain)domain*) → None

RegisterVariable (*(daeDataReceiverProcess)self*, *(daeDataReceiverVariable)variable*) → None

Variables

dictDomains

dictVariables

7.5 Module pyIDAS

7.5.1 Overview

7.5.2 Classes

daeDAESolver_t
daeIDAS

class pyIDAS.**daeDAESolver_t**
Bases: Boost.Python.instance

InitialConditionMode

Log

Name

NumberOfVariables

RelativeTolerance

class pyIDAS.**daeIDAS**
Bases: pyIDAS.daeDAESolver_t

SaveMatrixAsXPM (*(daeIDAS)self*, *(str)xpmFilename*) → None

SetLASolver (*(daeIDAS)self*, *(daeIDALASolverType)laSolverType*) → None
SetLASolver((daeIDAS)self, (object)laSolver) -> None

7.5.3 Enumerations

daeeIDALASolverType

class pyIDAS.**daeeIDALASolverType**
Bases: Boost.Python.enum

eSundialsGMRES = pyIDAS.daeeIDALASolverType.eSundialsGMRES

eSundialsLU = pyIDAS.daeeIDALASolverType.eSundialsLU

eSundialsLapack = pyIDAS.daeeIDALASolverType.eSundialsLapack

eThirdParty = pyIDAS.daeeIDALASolverType.eThirdParty

7.6 Module pyUnits

7.6.1 Overview

7.6.2 Classes

base_unit
unit
quantity

class pyUnits.**base_unit**
Bases: Boost.Python.instance

__init__(*(object)arg1*) → None
__init__((object)arg1, (float)arg2, (dict)arg3) -> object

__mul__(*(base_unit)arg1*, *(base_unit)arg2*) → object
__mul__((base_unit)arg1, (float)arg2) -> object

__div__(*(base_unit)arg1*, *(base_unit)arg2*) → object
__div__((base_unit)arg1, (float)arg2) -> object

__pow__(*(base_unit)arg1*, *(float)arg2*) → object

__eq__(*(base_unit)arg1*, *(base_unit)arg2*) → object

__ne__(*(base_unit)arg1*, *(base_unit)arg2*) → object

C

I

L

M

N

O

T

multiplier

class pyUnits.**unit**
Bases: Boost.Python.instance

__init__(*(object)arg1*) → None
__init__((object)arg1, (dict)arg2) -> object

__mul__(*(unit)arg1*, *(unit)arg2*) → object
__mul__((unit)arg1, (float)arg2) -> object

__div__ (*(unit)arg1*, *(unit)arg2*) → object
__div__((unit)arg1, (float)arg2) -> object

__pow__ (*(unit)arg1*, *(float)arg2*) → object

__eq__ (*(unit)arg1*, *(unit)arg2*) → object

__ne__ (*(unit)arg1*, *(unit)arg2*) → object

baseUnit

unitDictionary

class pyUnits.**quantity**
Bases: Boost.Python.instance

__init__ (*(object)arg1*) → None
__init__((object)self, (float)value, (unit)unit) -> None

__neg__ (*(quantity)arg1*) → object

__pos__ (*(quantity)arg1*) → object

__add__ (*(quantity)arg1*, *(quantity)arg2*) → object
__add__((quantity)arg1, (float)arg2) -> object

__sub__ (*(quantity)arg1*, *(quantity)arg2*) → object
__sub__((quantity)arg1, (float)arg2) -> object

__mul__ (*(quantity)arg1*, *(quantity)arg2*) → object
__mul__((quantity)arg1, (unit)arg2) -> object

__mul__((quantity)arg1, (float)arg2) -> object

__div__ (*(quantity)arg1*, *(quantity)arg2*) → object
__div__((quantity)arg1, (unit)arg2) -> object

__div__((quantity)arg1, (float)arg2) -> object

__pow__ (*(quantity)arg1*, *(quantity)arg2*) → object
__pow__((quantity)arg1, (float)arg2) -> object

__eq__ (*(quantity)arg1*, *(quantity)arg2*) → object
__eq__((quantity)arg1, (float)arg2) -> object

__eq__((quantity)arg1, (float)arg2) -> object

__ne__ (*(quantity)arg1*, *(quantity)arg2*) → object
__ne__((quantity)arg1, (float)arg2) -> object

__ne__((quantity)arg1, (float)arg2) -> object

__lt__ (*(quantity)arg1*, *(quantity)arg2*) → object
__lt__((quantity)arg1, (float)arg2) -> object

__lt__((quantity)arg1, (float)arg2) -> object

__le__ (*(quantity)arg1*, *(quantity)arg2*) → object
__le__((quantity)arg1, (float)arg2) -> object

__le__((quantity)arg1, (float)arg2) -> object

__gt__ (*(quantity)arg1*, *(quantity)arg2*) → object
__gt__((quantity)arg1, (float)arg2) -> object

__gt__((quantity)arg1, (float)arg2) -> object

__ge__ (*(quantity)arg1*, *(quantity)arg2*) → object
__ge__((quantity)arg1, (float)arg2) -> object

__ge__((quantity)arg1, (float)arg2) -> object

scaleTo (*(quantity)self*, *(object)referrer*) → quantity

units

value

valueInSIUnits

7.7 Variable types

7.7.1 Overview

variable_types.**time_t** = daeVariableType(name=time_t, units=[s], lowerBound=0, upperBound=1e+20, initialGuess=0, al

variable_types.**length_t** = daeVariableType(name=length_t, units=[m], lowerBound=0, upperBound=100000, initialGues

variable_types.**area_t** = daeVariableType(name=area_t, units=[m^2], lowerBound=0, upperBound=100000, initialGuess=

variable_types.**volume_t** = daeVariableType(name=volume_t, units=[m^3], lowerBound=0, upperBound=100000, initialC

variable_types.**velocity_t** = daeVariableType(name=velocity_t, units=[m/s], lowerBound=-1e+10, upperBound=1e+10, i

variable_types.**pressure_t** = daeVariableType(name=pressure_t, units=[Pa], lowerBound=100, upperBound=1e+10, init

variable_types.**temperature_t** = daeVariableType(name=temperature_t, units=[K], lowerBound=0, upperBound=10000

variable_types.**fraction_t** = daeVariableType(name=fraction_t, units=[1], lowerBound=-1e-10, upperBound=1.1, initial

variable_types.**no_t** = daeVariableType(name=no_t, units=[1], lowerBound=-1e+20, upperBound=1e+20, initialGuess=0, a

variable_types.**moles_t** = daeVariableType(name=moles_t, units=[mol], lowerBound=0, upperBound=1e+20, initialGuess

variable_types.**molar_flux_t** = daeVariableType(name=molar_flux_t, units=[mol/m^2], lowerBound=-1e+20, upperBou

variable_types.**molar_concentration_t** = daeVariableType(name=molar_concentration_t, units=[mol/m^3], lowerBou

variable_types.**molar_flowrate_t** = daeVariableType(name=molar_flowrate_t, units=[mol/s], lowerBound=-1e+10, upp

variable_types.**heat_t** = daeVariableType(name=heat_t, units=[J], lowerBound=-1e+20, upperBound=1e+20, initialGuess

variable_types.**heat_flux_t** = daeVariableType(name=heat_flux_t, units=[W/m^2], lowerBound=-1e+20, upperBound=

variable_types.**heat_transfer_coefficient_t** = daeVariableType(name=heat_transfer_coefficient_t, units=[W/(K n

variable_types.**power_t** = daeVariableType(name=power_t, units=[W], lowerBound=-1e+20, upperBound=1e+20, initialG

variable_types.**specific_heat_capacity_t** = daeVariableType(name=specific_heat_capacity_t, units=[J/(K kg)], low

variable_types.**density_t** = daeVariableType(name=density_t, units=[kg/m^3], lowerBound=0, upperBound=1e+20, init

variable_types.**specific_heat_conductivity_t** = daeVariableType(name=specific_heat_conductivity_t, units=[W/(

variable_types.**dynamic_viscosity_t** = daeVariableType(name=dynamic_viscosity_t, units=[Pa s], lowerBound=0, upp

variable_types.**diffusivity_t** = daeVariableType(name=diffusivity_t, units=[m^2/s], lowerBound=0, upperBound=100

variable_types.**amount_adsorbed_t** = daeVariableType(name=amount_adsorbed_t, units=[mol/kg], lowerBound=-1e+2

7.8 Third party solvers

7.8.1 Linear solvers

class pyCore.**daeIDALASolver_t**

Name

SaveAsXPM (*(daeIDALASolver_t)self, (str)xpmFilename*) → int

SuperLU

Instantiation function

pySuperLU.**daeCreateSuperLUSolver** () → daeIDALASolver_t

Classes

class pySuperLU.**daeSuperLU_Solver**
Bases: pySuperLU.daeIDALASolver_t

Options

SaveAsMatrixMarketFile(*(daeSuperLU_Solver)self*, *(str)filename*, *(str)matrixName*, *(str)description*) → int

class pySuperLU.**superlu_options_t**
Bases: Boost.Python.instance

ColPerm

DiagPivotThresh

PrintStat

RowPerm

Enumerations

class pySuperLU.**IterRefine_t**
Bases: Boost.Python.enum

DOUBLE = pySuperLU.IterRefine_t.DOUBLE

EXTRA = pySuperLU.IterRefine_t.EXTRA

NOREFINE = pySuperLU.IterRefine_t.NOREFINE

SINGLE = pySuperLU.IterRefine_t.SINGLE

class pySuperLU.**rowperm_t**
Bases: Boost.Python.enum

LargeDiag = pySuperLU.rowperm_t.LargeDiag

MY_PERMR = pySuperLU.rowperm_t.MY_PERMR

NOROWPERM = pySuperLU.rowperm_t.NOROWPERM

class pySuperLU.**yes_no_t**
Bases: Boost.Python.enum

NO = pySuperLU.yes_no_t.NO

YES = pySuperLU.yes_no_t.YES

class pySuperLU.**colperm_t**
Bases: Boost.Python.enum

COLAMD = pySuperLU.colperm_t.COLAMD

METIS_AT_PLUS_A = pySuperLU.colperm_t.METIS_AT_PLUS_A

MMD_ATA = pySuperLU.colperm_t.MMD_ATA

MMD_AT_PLUS_A = pySuperLU.colperm_t.MMD_AT_PLUS_A

NATURAL = pySuperLU.colperm_t.NATURAL

SuperLU_MT

Instantiation function

pySuperLU_MT.**daeCreateSuperLUSolver**() → daeIDALASolver_t

Classes

class pySuperLU_MT.**daeSuperLU_MT_Solver**
Bases: pySuperLU_MT.daeIDALASolver_t

Options

SaveAsMatrixMarketFile(*(daeSuperLU_MT_Solver)self*, *(str)filename*, *(str)matrixName*, *(str)description*) → int

class pySuperLU_MT.**superlumt_options_t**
Bases: Boost.Python.instance

ColPerm

PrintStat

diag_pivot_thresh

drop_tol

nprocs

panel_size

relax

Enumerations

class pySuperLU_MT.**yes_no_t**
Bases: Boost.Python.enum

NO = pySuperLU_MT.yes_no_t.NO

YES = pySuperLU_MT.yes_no_t.YES

class pySuperLU_MT.**colperm_t**
Bases: Boost.Python.enum

COLAMD = pySuperLU_MT.colperm_t.COLAMD

METIS_AT_PLUS_A = pySuperLU_MT.colperm_t.METIS_AT_PLUS_A

MMD_ATA = pySuperLU_MT.colperm_t.MMD_ATA

MMD_AT_PLUS_A = pySuperLU_MT.colperm_t.MMD_AT_PLUS_A

NATURAL = pySuperLU_MT.colperm_t.NATURAL

Trilinos

Instantiation function

pyTrilinos.**daeTrilinosSupportedSolvers**() → list

pyTrilinos.**daeCreateTrilinosSolver**(*(str)solverName*, *(str)preconditionerName*) → daeIDALASolver_t

Classes

class pyTrilinos.**daeTrilinosSolver**
Bases: pyTrilinos.daeIDALASolver_t

AmesosOptions

AztecOOOptions

IfpackOptions

MLOptions

NumIters

PreconditionerName

PrintPreconditionerInfo (*(daeTrilinosSolver)self*) → None

SaveAsMatrixMarketFile (*(daeTrilinosSolver)self*, *(str)filename*, *(str)matrixName*, *(str)description*) → int

Tolerance

class `pyTrilinos.`**TeuchosParameterList**
Bases: `Boost.Python.instance`

Print (*(TeuchosParameterList)self*) → None

get_bool (*(TeuchosParameterList)self*, *(str)name*) → bool

get_float (*(TeuchosParameterList)self*, *(str)name*) → float

get_int (*(TeuchosParameterList)self*, *(str)name*) → int

get_string (*(TeuchosParameterList)self*, *(str)name*) → str

set_bool (*(TeuchosParameterList)self*, *(str)name*, *(bool)value*) → None

set_float (*(TeuchosParameterList)self*, *(str)name*, *(float)value*) → None

set_int (*(TeuchosParameterList)self*, *(str)name*, *(int)value*) → None

set_string (*(TeuchosParameterList)self*, *(str)name*, *(str)value*) → None

7.8.2 Optimization solvers

class `pyCore.`**daeIDALASolver_t**

Name

Initialize (*(daeIDALASolver_t)self*, *(daeSimulation_t)simulation*, *(daeDAESolver_t)daeSolver*, *(daeDataReporter_t)dataReporter*, *(daeLog_t)log*) → None

Solve (*(daeIDALASolver_t)self*) → None

class `pyIPOPT.`**daeIPOPT**
Bases: `pyIPOPT.daeNLPSolver_t`

ClearOptions (*(daeIPOPT)self*) → None

LoadOptionsFile (*(daeIPOPT)self*, *(str)optionsFilename*) → None

PrintOptions (*(daeIPOPT)self*) → None

PrintUserOptions (*(daeIPOPT)self*) → None

SetOption (*(daeIPOPT)self*, *(str)name*, *(str)value*) → None
SetOption((daeIPOPT)self, (str)name, (float)value) -> None

SetOption((daeIPOPT)self, (str)name, (int)value) -> None

class `pyBONMIN.`**daeBONMIN**
Bases: `pyBONMIN.daeNLPSolver_t`

ClearOptions (*(daeBONMIN)self*) → None

LoadOptionsFile (*(daeBONMIN)self*, *(str)optionsFilename*) → None

PrintOptions (*(daeBONMIN)self*) → None

PrintUserOptions (*(daeBONMIN)self*) → None

SetOption (*(daeBONMIN)self*, *(str)name*, *(str)value*) → None
SetOption((daeBONMIN)self, (str)name, (float)value) -> None

SetOption((daeBONMIN)self, (str)name, (int)value) -> None

class pyNLOPT.**daeNLOPT**
Bases: pyNLOPT.daeNLPSolver_t

PrintOptions((daeNLOPT)self) → None

ftol_abs

ftol_rel

xtol_abs

xtol_rel

7.8.3 Parameter estimation solvers

class daetools.solvers.minpack.**daeMinpackLeastSq**

Finalize()

Initialize(*simulation*, *daesolver*, *datareporter*, *log*, ***kwargs*)

Run()

getConfidenceCoefficient(*confidence*)

getConfidenceEllipsoid(*x_param_index*, *y_param_index*, ***kwargs*)

getFit_Dyn(*measured_variable_index*, *experiment_index*, ***kwargs*)

getFit_SS(*input_variable_index*, *measured_variable_index*, ***kwargs*)

7.9 Code generators

7.9.1 Auxiliary classes

class daetools.code_generators.analyzer.**daeCodeGeneratorAnalyzer**
Bases: object

analyzeModel(*model*)

analyzePort(*port*)

analyzeSimulation(*simulation*)

class daetools.code_generators.formatter.**daeExpressionFormatter**
Bases: object

flattenIdentifier(*identifier*)

formatDomain(*domainCanonicalName*, *index*, *value*)

formatIdentifier(*identifier*)

formatNumpyArray(*arr*)

formatParameter(*parameterCanonicalName*, *domainIndexes*, *value*)

formatQuantity(*quantity*)

formatRuntimeConditionNode(*node*)

formatRuntimeNode(*node*)

formatTimeDerivative(*variableCanonicalName*, *domainIndexes*, *overallIndex*, *order*)

formatUnits(*units*)

formatVariable(*variableCanonicalName*, *domainIndexes*, *overallIndex*)

7.9.2 Modelica

class daetools.code_generators.modelica.**daeModelicaExpressionFormatter**
Bases: daetools.code_generators.formatter.daeExpressionFormatter

formatNumpyArray (*arr*)

formatQuantity (*quantity*)

formatUnits (*units*)

class daetools.code_generators.modelica.**daeCodeGenerator_Modelica** (*simulation=None*)
Bases: object

generateModel (*model, filename=None*)

generatePort (*port, filename=None*)

generateSimulation (*simulation, filename=None*)

7.9.3 ANSI C

class daetools.code_generators.ansi_c.**daeANSICExpressionFormatter**
Bases: daetools.code_generators.formatter.daeExpressionFormatter

formatNumpyArray (*arr*)

formatQuantity (*quantity*)

class daetools.code_generators.ansi_c.**daeCodeGenerator_ANSI_C** (*simulation=None*)
Bases: object

generateSimulation (*simulation, **kwargs*)

7.9.4 Functional Mockup Interface (FMI)

class daetools.code_generators.fmi.**daeCodeGenerator_FMI**
Bases: daetools.code_generators.fmi_xml_support.fmiModelDescription

generateSimulation (*simulation, **kwargs*)

7.10 DAE Tools Plotter

7.10.1 Overview

7.10.2 Classes

class daetools.dae_plotter.plotter.**daeMainWindow** (*tcpipServer*)
Bases: PyQt4.QtGui.QMainWindow

plot2D (*updateInterval=0*)

slotAbout ()

slotAnimatedPlot2D ()

slotDocumentation ()

slotPlot2D ()

slotPlot3D ()

daetools.dae_plotter.plotter.**daeStartPlotter** (*port=0*)

class daetools.dae_plotter.plot2d.**dae2DPlot** (*parent, tcpipServer, updateInterval=0*)
Bases: PyQt4.QtGui.QDialog

addLine (*xAxisLabel, yAxisLabel, xPoints, yPoints, domains*)

closeEvent (*event*)

newCurve ()

plotDefaults = [<daetools.dae_plotter.plot2d.daePlot2dDefaults instance at 0x53ac8c0>, <daetools.dae_plotter.plot2d.d

reformatPlot ()

slotExportCSV ()

slotProperties ()

slotRemoveLine ()

slotToggleGrid ()

slotToggleLegend ()

slotViewTabularData ()

updateCurves ()

class daetools.dae_plotter.plot2d.**daePlot2dDefaults** (*color='black'*, *linewidth=0.5*, *linestyle='solid'*, *marker='o'*, *markersize=6*, *markerfacecolor='black'*, *markeredgecolor='black'*)

class daetools.dae_plotter.mayavi_plot3d.**daeMayavi3DPlot** (*tcpipServer*)

newSurface ()

daetools.dae_plotter.plot_options.**col2hex** (*color*)
Convert matplotlib color to hex

daetools.dae_plotter.plot_options.**figure_edit** (*canvas*, *parent=None*)
Edit matplotlib figure options

daetools.dae_plotter.plot_options.**surface_edit** (*canvas*, *parent=None*)
Edit matplotlib figure options

7.11 DAE Tools Simulator

7.11.1 Overview

7.11.2 Classes

class daetools.dae_simulator.simulator.**daeSimulator** (*app*, ***kwargs*)
Bases: PyQt4.QtGui.QDialog

done (*status*)

laAmdACML = 10

laAmesos_Klu = 3

laAmesos_Lapack = 6

laAmesos_Superlu = 4

laAmesos_Umfpack = 5

laAztecOO = 7

laCUSP = 14

laIntelMKL = 9

laIntelPardiso = 8

laLapack = 11

laMagmaLapack = 12

laSundialsLU = 0

laSuperLU = 1

laSuperLU_CUDA = 13

laSuperLU_MT = 2

nlpBONMIN = 2

nlpIPOPT = 0

nlpNLOPT = 1

showMessage (*msg*)

slotExportSparseMatrixAsMatrixMarketFormat ()

slotOpenSparseMatrixImage ()

slotPause ()

slotResume ()

slotRun ()

CHAPTER

EIGHT

TUTORIALS

Note: Currently only Mozilla Firefox and Opera 12+ browsers are supported for viewing model reports (MathML rendering issue).

8.1 What's the time? (AKA: Hello world!)

Description

What is the time? (AKA Hello world!) is a very simple simulation. It shows the basic structure of the model and the simulation classes. A typical simulation includes 8 basic tasks:

1. How to import the pyDAE module(s)
2. How to declare variable types daeVariableType
3. How to define a model by deriving a class from the base daeModel and which functions must be implemented by every model (daeModel.__init__, daeModel.DeclareEquations):
   - how to declare parameters and variables in daeModel.__init__ function
   - how to declare equations and their residuals in daeModel.DeclareEquations function
4. How to define a simulation by deriving a class from the base daeSimulation and which functions must be implemented by every simulation (daeSimulation.__init__, daeSimulation.SetUpParametersAndDomains, daeSimulation.SetUpVariables):
   - how to set specify a model to be used in simulation in daeSimulation.__init__ function
   - how to set values of parameters in daeSimulation.SetUpParametersAndDomains function
   - how to set initial conditions in daeSimulation.SetUpVariables‘ function
5. How to create auxiliary objects required for the simulation (DAE solver, data reporter and logging objects)
6. How to set simulation's additional settings
7. How to connect a data reporter
8. How to run a simulation

Files

Model report	whats_the_time.xml
Runtime model report	whats_the_time-rt.xml
Source code	whats_the_time.py

8.2 Tutorial 1

Description

This tutorial introduces several new concepts:

- Distribution domains
- Distributed parameters, variables and equations
- Boundary and initial conditions

In this example we model a simple heat conduction problem: a conduction through a very thin, rectangular copper plate.

This example should be sufficiently complex to describe all basic DAE Tools features. For this problem, we need a two-dimensional Cartesian grid in X and Y axis (here, for simplicity, divided into 10 x 10 segments):

```
 Y axis
    ^
    |
Ly -| T T T T T T T T T T T
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
    | L + + + + + + + + + R
 0 -| B B B B B B B B B B B
   --|-------------------|-------> X axis
     0                   Lx
```

Points 'B' at the bottom edge of the plate (for y = 0), and the points 'T' at the top edge of the plate (for y = Ly) represent the points where the heat is applied.

The plate is considered insulated at the left (x = 0) and the right edges (x = Lx) of the plate (points 'L' and 'R'). To model this type of problem, we have to write a heat balance equation for all interior points except the left, right, top and bottom edges, where we need to define the Neumann type boundary conditions.

In this problem we have to define the following domains:

- x: X axis domain, length Lx = 0.1 m
- y: Y axis domain, length Ly = 0.1 m

the following parameters:

- ro: copper density, 8960 kg/m3
- cp: copper specific heat capacity, 385 J/(kgK)
- k: copper heat conductivity, 401 W/(mK)
- Qb: heat flux at the bottom edge of the plate, 1E6 W/m2 (or 100 W/cm2)
- Qt: heat flux at the top edge of the plate, here set to 0 W/m2

and the following variable:

- T: the temperature of the plate, K (distributed on x and y domains)

Also, we need to write the following 5 equations:

1. Heat balance:

```
ro * cp * dT(x,y) / dt = k * (d2T(x,y) / dx2 + d2T(x,y) / dy2);  for all x in: (0, Lx),
                                                                  for all y in: (0, Ly)
```

2. Boundary conditions for the bottom edge:

```
-k * dT(x,y) / dy = Qin;  for all x in: [0, Lx],
                          and y = 0
```

3. Boundary conditions for the top edge:

```
-k * dT(x,y) / dy = Qin;  for all x in: [0, Lx],
                          and y = Ly
```

4. Boundary conditions for the left edge:

```
dT(x,y) / dx = 0;  for all y in: (0, Ly),
                   and x = 0
```

5. Boundary conditions for the right edge:

```
dT(x,y) / dx = 0;  for all y in: (0, Ly),
                   and x = Ln
```

Files

Model report	tutorial1.xml
Runtime model report	tutorial1-rt.xml
Source code	tutorial1.py

8.3 Tutorial 2

Description

In this example we use the same conduction problem as in the tutorial 1.

Here we introduce:

- Arrays (discrete distribution domains)
- Distributed parameters
- Number of degrees of freedom and how to fix it
- Initial guess of the variables

Files

Model report	tutorial2.xml
Runtime model report	tutorial2-rt.xml
Source code	tutorial2.py

8.4 Tutorial 3

Description

In this example we use the same conduction problem as in the tutorial 1.

Here we introduce:

- Arrays of variable values
- Functions that operate on arrays of values
- Functions that create constants and arrays of constant values (Constant and Array)

- Non-uniform domain grids

Files

Model report	tutorial3.xml
Runtime model report	tutorial3-rt.xml
Source code	tutorial3.py

8.5 Tutorial 4

Description

In this example we model a very simmple conduction problem where a piece of copper (a plate) is at one side exposed to the source of heat and at the other to the surroundings.

Here we introduce:

- Discontinuous equations (symmetrical state transition networks: daeIF statements)

Here we have a very simple heat balance:

```
ro * cp * dT/dt - Qin = h * A * (T - Tsurr)
```

The process starts at the temperature of the metal of 283K. The metal is allowed to warm up for 200 seconds and then the heat source is removed and the metal cools down slowly to the ambient temperature.

Files

Model report	tutorial4.xml
Runtime model report	tutorial4-rt.xml
Source code	tutorial4.py

8.6 Tutorial 5

Description

In this example we use the same conduction problem as in the tutorial 4.

Here we introduce:

- Discontinuous equations (non-symmetrical state transition networks: daeSTN statements)

Again we have a piece of copper (a plate) is at one side exposed to the source of heat and at the other to the surroundings. The process starts at the temperature of 283K. The metal is allowed to warm up, and then its temperature is kept in the interval [320 - 340] for at 350 seconds. After 350s the heat source is removed and the metal cools down slowly again to the ambient temperature.

Files

Model report	tutorial5.xml
Runtime model report	tutorial5-rt.xml
Source code	tutorial5.py

8.7 Tutorial 6

Description

This is the simple port demo.

Here we introduce:

- Ports
- Port connections
- Units (instances of other models)

A simple port type 'portSimple' is defined which contains only one variable 't'. Two models 'modPortIn' and 'modPortOut' are defined, each having one port of type 'portSimple'. The wrapper model 'modTutorial' instantiate these two models as its units and connects them by connecting their ports.

Files

Model report	tutorial6.xml
Runtime model report	tutorial6-rt.xml
Source code	tutorial6.py

8.8 Tutorial 7

Description

In this example we use the same conduction problem as in the tutorial 1.

Here we introduce:

- Custom operating procedures
- Resetting of degrees of freedom
- Resetting of initial conditions

Here the heat flux at the bottom edge is defined as a variable. In the simulation its value will be fixed and manipulated in the custom operating procedure.

Files

Model report	tutorial7.xml
Runtime model report	tutorial7-rt.xml
Source code	tutorial7.py

8.9 Tutorial 8

Description

In this example we use a similar problem as in the tutorial 5.

Here we introduce:

- Writting Matlab .MAT files (using daeMatlabMATFileDataReporter)
- Custom data reporters

Some time it is not enough to send the result to daePlotter but it is desirable to export them in certain format for use in other programs. Here we show how the custom data reporter can be created. In this example the data reporter simply, after the simulation is finished, save the results into a plain text file. Obviously, the data can be exported to any format. Also some numpy functions that operate on numpy arrays can be used as well. In addition, a new type of data reporters (daeDelegateDataReporter) is presented. It has the same interface and the functionality like all data reporters. However, it does not do any data processing itself but calls the corresponding functions of data reporters which are added to it by using the function AddDataReporter. This way it is possible, at the same time, to send the results to the daePlotter and save them into a file (or process them in some other ways).

Files

Model report	tutorial8.xml
Runtime model report	tutorial8-rt.xml
Source code	tutorial8.py

8.10 Tutorial 9

Description

In this example we use the same conduction problem as in the tutorial 1.

Here we introduce:

- Third party linear equations solvers

Currently there are 3rd party linear equations solvers:

- SuperLU: sequential sparse direct solver defined in pySuperLU module (BSD licence)
- SuperLU_MT: multi-threaded sparse direct solver defined in pySuperLU_MT module (BSD licence)
- Trilinos: sequential sparse direct/iterative solver defined in pyTrilinos module (GNU Lesser GPL)
- IntelPardiso: multi-threaded sparse direct solver defined in pyIntelPardiso module (proprietary)

Files

Model report	tutorial9.xml
Runtime model report	tutorial9-rt.xml
Source code	tutorial9.py

8.11 Tutorial 10

Description

In this example we use the same conduction problem as in the tutorial 1.

Here we introduce:

- daeModel functions d() and dt() which calculate time- or partial-derivative of an expression
- Initialization files
- Domains which bounds depend on parameter values
- How to evaluate integrals of a function

Files

Model report	tutorial10.xml
Runtime model report	tutorial10-rt.xml
Source code	tutorial10.py

8.12 Tutorial 11

Description

This tutorial shows the use of Trilinos group of solvers: Amesos and AztecOO iterative linear equation solvers with different preconditioners (built-in AztecOO, Ifpack or ML) and corresponding linear solver options.

ACHTUNG, ACHTUNG!!

Iterative solvers are not fully working yet and this example is given just as a showcase and for preconditioner options experimenting purposes.

Files

Model report	tutorial11.xml
Runtime model report	tutorial11-rt.xml
Source code	tutorial11.py

8.13 Tutorial 12

Description

As of the version 1.1.1 the main linear algebraic equations solver is superLU.

It comes in three variaants:

- Sequential: superlu
- Multithreaded (OpenMP/posix threads): superlu_MT
- CUDA GPU: superlu_CUDA

The first two are available in daetools with the addition of a new port: superlu_CUDA that works on computers with NVidia CUDA enabled video cards. However, the later is still in an early stage of the development.

In this example, usage and available options of superlu and superlu_MT are explored.

Files

Model report	tutorial12.xml
Runtime model report	tutorial12-rt.xml
Source code	tutorial12.py

8.14 Tutorial 13

Description

In this example we use the same problem as in the tutorial 5.

Here we introduce:

- The event ports
- ON_CONDITION() function showing the new types of actions that can be executed during state transitions
- ON_EVENT() function showing the new types of actions that can be executed when an event is triggered
- User defined actions

Files

Model report	tutorial13.xml
Runtime model report	tutorial13-rt.xml
Source code	tutorial13.py

8.15 Tutorial 14

Description

In this example we use the same conduction problem as in the tutorial 5.

Here we introduce the external functions concept that can handle and evaluate functions in external libraries. Here we use daeScalarExternalFunction class derived external function object to calculate the power.

A support for external functions is still experimental and the goal is to support certain software components such as thermodynamic property packages etc.

Files

Model report	tutorial14.xml
Runtime model report	tutorial14-rt.xml
Source code	tutorial14.py

8.16 Tutorial 15

Description

In this example we use the same problem as in the tutorial 4.

Here we introduce:

- Nested state transitions

Files

Model report	tutorial15.xml
Runtime model report	tutorial15-rt.xml
Source code	tutorial15.py

8.17 Tutorial 16

Description

In this example we use the same conduction problem as in the tutorial 4.

Here we introduce:

- Interactive operating procedures

Files

Model report	tutorial16.xml
Runtime model report	tutorial16-rt.xml
Source code	tutorial16.py

8.18 Tutorial 17

Description

In this example we use the same conduction problem as in the tutorial 4.

Here we introduce:

- TCPIP Log and TCPIPLogServer

Files

Model report	tutorial17.xml
Runtime model report	tutorial17-rt.xml
Source code	tutorial17.py

8.19 Optimization tutorial 1

Description

This tutorial introduces IPOPT NLP solver, its setup and options.

Files

Model report	opt_tutorial1.xml
Runtime model report	opt_tutorial1-rt.xml
Source code	opt_tutorial1.py

8.20 Optimization tutorial 2

Description

This tutorial introduces Bonmin MINLP solver, its setup and options.

Files

Model report	opt_tutorial2.xml
Runtime model report	opt_tutorial2-rt.xml
Source code	opt_tutorial2.py

8.21 Optimization tutorial 3

Description

This tutorial introduces NLOPT NLP solver, its setup and options.

Files

Model report	opt_tutorial3.xml
Runtime model report	opt_tutorial3-rt.xml
Source code	opt_tutorial3.py

8.22 Optimization tutorial 4

Description

This tutorial shows the interoperability between DAE Tools and 3rd party optimization software (scipy.optimize) used to minimize the Rosenbrock function.

DAE Tools simulation is used to calculate the objective function and its gradients, while scipy.optimize.fmin function (Nelder-Mead Simplex algorithm) to find the minimum of the Rosenbrock function.

Files

Model report	opt_tutorial4.xml
Runtime model report	opt_tutorial4-rt.xml
Source code	opt_tutorial4.py

8.23 Optimization tutorial 5

Description

This tutorial shows the interoperability between DAE Tools and 3rd party optimization software (scipy.optimize) used to fit the simple function with experimental data.

DAE Tools simulation object is used to calculate the objective function and its gradients, while scipy.optimize.leastsq function (a wrapper around MINPACK's lmdif and lmder) implementing Levenberg-Marquardt algorithm is used to estimate the parameters.

Files

Model report	opt_tutorial5.xml
Runtime model report	opt_tutorial5-rt.xml
Source code	opt_tutorial5.py

8.24 Optimization tutorial 6

Description

daeMinpackLeastSq module test.

Files

Model report	opt_tutorial6.xml
Runtime model report	opt_tutorial6-rt.xml
Source code	opt_tutorial6.py

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y