DAE Tools Project Documentation

Release 1.3.1

by Dragan Nikolic

April 22, 2013

CONTENTS

CHAPTER

ONE

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

1.1 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; for complex projects I still prefer the heavy c++ artillery.

1.2 The main concepts

DAE Tools contains 5 modules:

- Core
- Activity
- DataReporting
- IDAS
- Units

1.2.1 Core module

Core module defines the main modelling concepts:

- **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**

    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.

  - **State Transition**

    State transition can be defined as a transition from the current to some other state, subject to given conditions.

  - **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).

- **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

- **Solver**

  Solver is a set of mathematical procedures/algorithms necessary to solve a given set of equations. There are several types of solvers: Linear Algebraic solvers (**LA**), used to solve linear systems of equations; Nonlinear Algebraic solvers (**NLA**), used to solve non-linear systems of equations; Differential Algebraic solvers (**DAE**), used to solve mixed systems of differential and algebraic equations; Nonlinear Programming solvers (**NLP**), used to solve nonlinear optimization problems; Mixed-integer Nonlinear Programming solvers (**MINLP**), used to solve mixed-integer nonlinear optimization problems. In **DAE Tools** it is possible to choose **DAE** (currently only Sundials IDAS), **NLP/MINLP** (currently IPOPT/BONMIN and NLOPT), and **LA** solvers (built-in Sundials LA solvers; Trilinos Amesos; Trilinos AztecOO; SuperLU/SuperLU_MT; Intel MKL; AMD ACML).

- **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.

- **Log**

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

1.2.2 Activity module

1.2.3 DataReporting module

1.2.4 IDAS module

1.2.5 Units module

1.3 Running a simulation

Two steps are needed to run a simulation:

1. Start **daePlotter**:

   - **In GNU/Linux and MacOS:**

     Go to: **Applications/Programming/daePlotter** or type the following shell command: **daeplotter**

- **In Windows:**

  Go to: **Start/Programs/DAE Tools/daePlotter** The **daePlotter** main window should appear (given in *Figure 1.*)

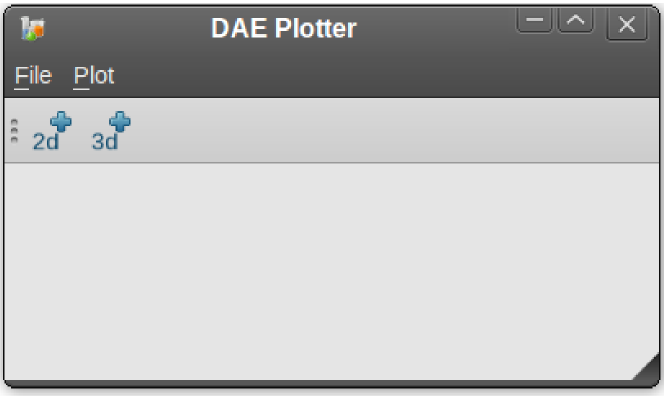

Figure 1.1: **Figure 1.** daePlotter main window.

daePlotter can be also added to a panel. Simply add a custom application launcher (command: daeplotter).

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

   - **In GNU/Linux and MacOS:**

     Go to: **Applications/Programming/DAE Tools Examples** or type the following shell command: **daeexamples**

   - **In Windows:**

     Go to: **Start/Programs/DAE Tools/DAE Tools Examples**

In general, simulations are started by typing the following shell commands (GNU/Linux and Windows): figwidth .. code-block:: bash

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

The main window of **DAE Tools Examples** application is given in *Figure 2a.* while the output from the simulation run in *Figure 2b.*. 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).

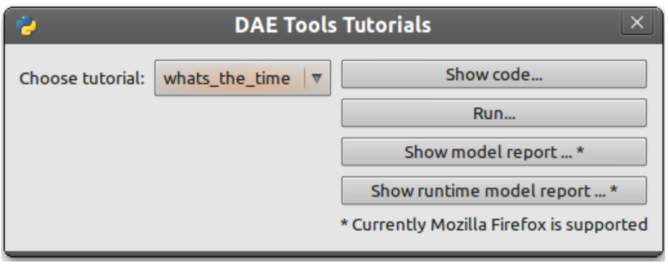

Figure 1.2: **Figure 2a.** DAE Tools Examples main window

The simulation can also be started from the shell. The sample output is given in *Figure 3.*:

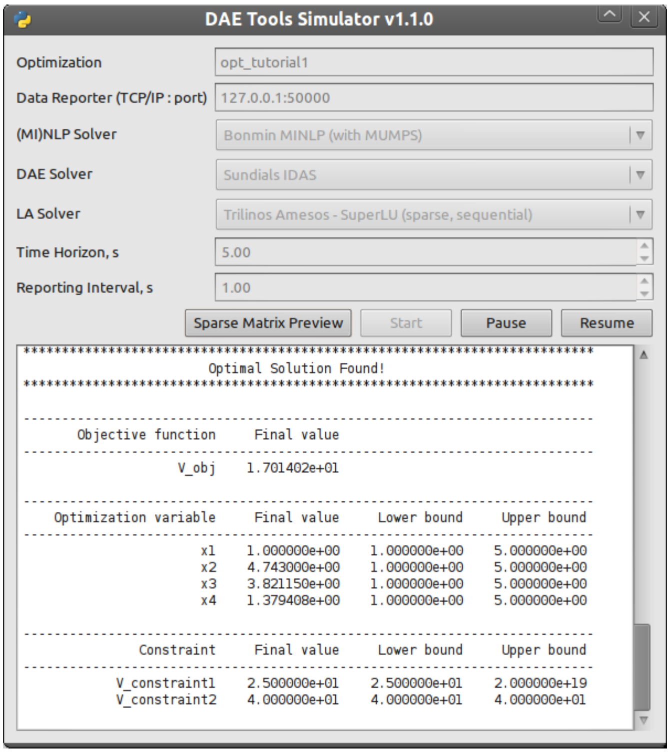

Figure 1.3: **Figure 2b.** A typical optimization output from DAE Tools

1.4 Running an optimization

Running the optimization problems is analogous to running a simulation.

1.5 Model development

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

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

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:

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

[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.

Figure 1.4: **Figure 3.** Shell output from the simulation

- 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).

1.5.1 Models

In **DAE Tools** models are created by defining its parameters, distribution domains, variables, equations, and ports. Models are developed by deriving a new class from the base model class (`pyCore.daeModel`). The process consists of two steps:

1. Declare all domains, parameters, variables and ports in `__init__()` function (the constructor)
2. Declare equations and state transition networks in `DeclareEquations()` function

Models in **pyDAE** (using python programming language) can be defined by the following statement:

```
class myModel(daeModel):
    def __init__(self, Name, Parent = None, Description = ""):
        daeModel.__init__(self, Name, Parent, Description)
        ... (here go declarations of domains, parameters, variables, ports, etc)

    def DeclareEquations(self):
        ... (here go declarations of equations and state transitions)
```

while in **cDAE** (using c++ programming language):

```
class myModel : public daeModel
{
public:
    myModel(string strName, daeModel* pParent = NULL, string strDescription = "")
    : daeModel(strName, pParent, strDescription)
    {
        ... (here go additional properties of domains, parameters, variables, ports, etc)
    }

    void DeclareEquations(void)
    {
        ... (here go declarations of equations and state transitions)
    }

public:
    ... (here go declarations of domains, parameters, variables, ports, etc)
};
```

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*.

1.5.2 Distribution domains

There are two types of domains in **DAE Tools**: simple arrays and distributed domains (commonly used to distribute variables, parameters and equations in space). The distributed domains can have a uniform (default) or a user specified non-uniform grid. At the moment, only the following finite difference methods can be used to calculate partial derivatives:

[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.

- Backward finite difference method (BFD)
- Forward finite difference method (FFD)
- Center finite difference method (CFD)

In **DAE Tools** just anything can be distributed on domains: parameters, variables, equations even models and ports. Obviously it does not have a physical meaning to distribute a model on a domain, However that can be useful for modelling of complex processes where we can create an array of models where each point in a distributed domain have a corresponding model so that a user does not have to take care of number of points in the domain, etc. In addition, domain points values can be obtained as a **NumPy** one-dimensional array; this way **DAE Tools** can be easily used in conjuction with other scientific python libraries NumPy, SciPy, for instance and many other.

Domains in **pyDAE** can be defined by the following statement:

```
myDomain = daeDomain("myDomain", Parent_Model_or_Port, Description)
```

while in **cDAE**:

```
daeDomain myDomain("myDomain", &Parent_Model_or_Port, Description);
```

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

1.5.3 Parameters

There are two types of parameters in **DAE Tools**: ordinary and distributed. Several functions to get a parameter value (function call operator `__call__()`) and array of values (`array()`) have been defined. In addition, distributed parameters have `npyValues` property to get the values as a numpy multi-dimensional array.

Parameters in **pyDAE** can be defined by the following statement:

```
myParam = daeParameter("myParam", eReal, Parent_Model_or_Port, "Description")
```

while in **cDAE**:

```
daeParameter myParam("myParam", eReal, &Parent_Model_or_Port, "Description");
```

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

1.5.4 Variables

There are two types of variables in **DAE Tools**: ordinary and distributed. Functions to get a variable value (function call operator `__call__()`), a time or a partial derivative (`dt()`, `d()`, or `d2()`) or functions to obtain an array of values, time or partial derivatives (`array()`, `dt_array()`, `d_array()`, or `d2_array()`) have been defined. In addition, distributed variables have `npyValues` property to get the values as a numpy multi-dimensional array.

Variables in **pyDAE** can be defined by the following statement:

```
myVar = daeVariable("myVar", variableType, Parent_Model_or_Port, "Description")
```

while in **cDAE**:

```
daeVariable myVar("myVar", variableType, &Parent_Model_or_Port, "Description");
```

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

1.5.5 Equations

DAE Tools introduce two types of equations: ordinary and distributed. What makes distributed equations special is that an equation expression is valid on every point within the domains that the equations is distriibuted on. Equations can be distributed on a whole domain, on a part of it or on some of the points in a domain. Equations in **pyDAE** can be defined by the following statement:

```
eq = model.CreateEquation("myEquation", "Description")
```

while in **cDAE**:

```
daeEquation* eq = model.CreateEquation("myEquation", "Description");
```

To define an equation expression (used to calculate its residual and its gradient - which represent a single row in a Jacobian matrix) **DAE Tools** combine the operator overloading technique for automatic differentiation (adopted from ADOL-C library) with the concept of representing equations as **evaluation trees**. Evaluation trees are made of binary or unary nodes, itself representing four basic mathematical operations and frequently used mathematical functions, such as `sin, cos, tan, sqrt, pow, log, ln, exp, min, max, floor, ceil, abs, sum, product, ...`. These basic mathematical operations and functions are implemented to operate on **a heavily modified ADOL-C** library class `adouble` (which has been extended to contain information about domains/parameters/variables etc). In adition, a new `adouble_array` class has been introduced to apply all above-mentioned operations on arrays of variables. What is different here is that `adouble`/`adouble_array` classes and mathematical operators/functions work in two modes; they can either **build-up an evaluation tree** or **calculate a value of an expression**. Once built the evaluation trees can be used to calculate equation residuals or derivatives to fill a Jacobian matrix necessary for a Newton-type iteration. A typical evaluation tree is presented in *Figure 4.*.

$$F = \frac{\partial V_{14}}{\partial t} + \frac{V_1}{V_{14} + 2.5} + \sin\left(3.14 \cdot V_3\right) = 0$$

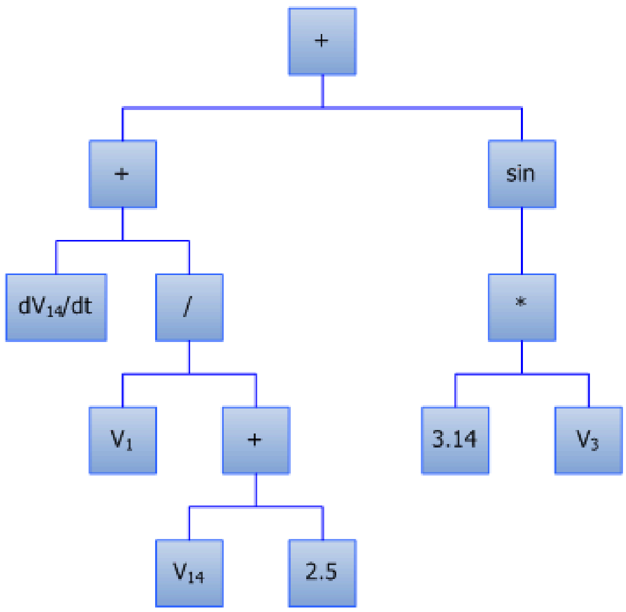

Figure 1.5: **Figure 4.** DAE Tools equation evaluation tree

As it has been noted before, domains, parameters, and variables contain functions that return `adouble`/`adouble_array` objects, which can be used to calculate residuals and derivatives. These functions include functions to get a value of a domain/parameter/variable (function call operator), to get a time or a partial derivative of a variable (functions `dt()`, `d()`, or `d2()`) or functions to obtain an array of values, time or partial derivatives (`array()`, `dt_array()`, `d_array()`, or `d2_array()`). Another useful feature of **DAE Tools** equations is that they can be exported into MathML or Latex format and easily visualized.

For example, the equation *F* (given in *Figure 4.*) can be defined in **pyDAE** by using the following statements:

```
F = model.CreateEquation("F", "F description")
F.Residal = V14.dt() + V1() / (V14() + 2.5) + Sin(3.14 * V3())
```

while in **cDAE** by:

```
daeEquation* F = model.CreateEquation("F", "F description");
F->SetResidal( V14.dt() + V1() / (V14() + 2.5) + sin(3.14 * V3()) );
```

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

1.5.6 State Transition Networks (discontinuous equations)

Discontinuous equations are equations that take different forms subject to certain conditions. For example, if we want to model a flow through a pipe we may observe three different flow regimes:

- Laminar: if Reynolds number is less than 2,100
- Transient: if Reynolds number is greater than 2,100 and less than 10,000
- Turbulent: if Reynolds number is greater than 10,000

What we can see is that from any of these three states we can go to any other state. This type of discontinuities is called a **reversible discontinuity** and can be described by the `IF()`, `ELSE_IF()`, `ELSE()` state transient network functions. In **pyDAE** it is given by the following statement:

```
IF(Re() <= 2100)                    # (Laminar flow)
#... (equations go here)

ELSE_IF(Re() > 2100 and Re() < 10000) # (Transient flow)
#... (equations go here)

ELSE()                              # (Turbulent flow)
#... (equations go here)

END_IF()
```

while in **cDAE** by:

```
IF(Re() <= 2100);                    // (Laminar flow)
//... (equations go here)

ELSE_IF(Re() > 2100 && Re() < 10000);  // (Transient flow)
//... (equations go here)

ELSE();                              // (Turbulent flow)
//... (equations go here)

END_IF();
```

Reversible discontinuities can be **symmetrical** and **non-symmetrical**. The above example is **symmetrical**. However, if we have a CPU and we want to model its power dissipation we may have three operating modes with the following state transitions:

- Normal mode
  - switch to **Power saving mode** if CPU load is below 5%
  - switch to **Fried mode** if the temperature is above 110 degrees
- Power saving mode
  - switch to **Normal mode** if CPU load is above 5%

  – switch to **Fried mode** if the temperature is above 110 degrees

- Fried mode (no escape from here... go to the nearest shop and buy a new one!)

What we can see is that from the **Normal mode** we can either go to the **Power saving mode** or to the **Fried mode**. The same stands for the **Power saving mode**: we can either go to the **Normal mode** or to the **Fried mode**. However, once the temperature exceeds 110 degrees the CPU dies (let's say we heavily overclocked it) and there is no going back. This type of discontinuities is called an **irreversible discontinuity** and can be described by using `STN()`, `STATE()`, `END_STN()` functions while state transitions using `ON_CONDITION()` function. In **pyDAE** this type of state transitions is given by the following statement:

```
STN("CPU")

STATE("Normal")
#... (equations go here)
ON_CONDITION(CPULoad() < 0.05, switchToState = "PowerSaving")
ON_CONDITION(T() > 110,        switchToState = "Fried")

STATE("PowerSaving")
#... (equations go here)
ON_CONDITION(CPULoad() >= 0.05, switchToState = "Normal")
ON_CONDITION(T() > 110,         switchToState = "Fried")

STATE("Normal")
#... (equations go here)

END_STN()
```

while in **cDAE** by:

```
STN("CPU");

STATE("Normal");
//... (equations go here)
ON_CONDITION(CPULoad() < 0.05, switchToState = "PowerSaving");
ON_CONDITION(T() > 110,        switchToState = "Fried");


STATE("PowerSaving");
//... (equations go here)
ON_CONDITION(CPULoad() >= 0.05, switchToState = "Normal");
ON_CONDITION(T() > 110,         switchToState = "Fried");

STATE("Normal");
//... (equations go here)

END_STN();
```

More information about state transition networks can be found in *pyDAE User Guide* and `pyCore.daeSTN`. Also, do not forget to have a look on *Tutorials*.

1.5.7 Ports

Ports are used to connect two models. Like models, they may contain domains, parameters and variables. For instance, in **pyDAE** ports can be defined by the following statements:

```
class myPort(daePort):
    def __init__(self, Name, Type, Parent = None, Description = ""):
        daePort.__init__(self, Name, Type, Parent, Description)
        #... (here go declarations of domains, parameters and variables)
```

while in **cDAE** by:

```
class myPort : public daePort
{
public:
myPort(string strName, daeePortType eType, daeModel* pParent, string strDescription = "")
        : daePort(strName, eType, pParent, strDescription)
    {
        //... (here go additional properties of domains, parameters and variables)
    }

public:
    //... (here go declarations of domains, parameters and variables)
};
```

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

1.5.8 Event Ports

Event ports are also used to connect two models; however, they allow sending of discrete messages (events) between model instances. Events can be triggered manually or as a result of a state transition in a model. The main difference between event and ordinary ports is that the former allow a discrete communication between model instances while latter allow a continuous exchange of information. A single outlet event port can be connected to unlimited number of inlet event ports. Messages contain a floating point value that can be used by a recipient (these actions are specified in `ON_EVENT()` function); that value might be a simple number or an expression involving model variables/parameters.

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

1.5.9 Simulation

As it was mentioned before, simulation of a process can be considered as the model run for certain input conditions. To define a simulation in **DAE Tools** the following tasks have to be done:

1. Derive a new simulation class
    - Specify a model to simulate
    - Specify its domains and parameters information
    - Fix the degrees of freedom by assigning the values to certain variables
    - Set the initial conditions for differential variables
    - Set the other variables' information: initial guesses, absolute tolerances, etc
    - Specify the 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

Simulations in **pyDAE** can be defined by the following construct:

```python
class mySimulation(daeSimulation):
    def __init__(self):
        daeSimulation.__init__(self)
        self.m = myModel("myModel", "Description")

    def SetUpParametersAndDomains(self):
        ... (here we set up domains and parameters)

    def SetUpVariables(self):
        ... (here we set up degrees of freedom, initial conditions, initial guesses, etc)

    def Run(self):
        ... (here goes a custom operating procedure, if needed)
```

while in **cDAE** by:

```cpp
class mySimulation : public daeSimulation
{
public:
    mySimulation(void) : m("myModel", "Description")
    {
        SetModel(&m);
    }

public:
    void SetUpParametersAndDomains(void)
    {
        ... (here we set up domains and parameters)
    }

    void SetUpVariables(void)
    {
        ... (here we set up degrees of freedom, initial conditions, initial guesses, etc)
    }

    void Run(void)
    {
        ... (here goes a custom operating procedure, if needed)
    }

public:
    myModel m;
};
```

1.5.10 Running a simulation

Simulations in **pyDAE** can be run in two modes:

By using PyQt4 graphical user interface (GUI) # From the shell

1. Running a simulation from the GUI (**pyDAE** only):

```python
# 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 = simTutorial()
```

```
# 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_()
```

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.

2. Running a simulation 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   = simTutorial()

# Set the linear solver (optional)
lasolver     = pyTrilinosAmesos.CreateTrilinosAmesosSolver("Amesos_Superlu")
solver.SetLASolver(eThirdParty, lasolver)

# 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 string)
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()
simulation.Finalize()
```

while in **cDAE** by:

```
// Create Log, Solver, DataReporter and Simulation object
boost::scoped_ptr<daeSimulation_t>     pSimulation(new simTutorial);
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 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 simulation
pSimulation->Initialize(pDAESolver.get(), pDataReporter.get(), pLog.get());

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

// Run
pSimulation->Run();
pSimulation->Finalize();
```

1.5.11 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 is specified in the function **SetUpOptimization** in the **daeSimulation** class. The tasks have to be done are:

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**

      sover 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 is declared in **pyDAE** as the following:

```
class mySimulation(daeSimulation):
    ... (here we set up a simulation)

    def SetUpOptimization(self):
        ... (here goes a declaration of the obj. function, opt. variables and constraints)
```

while in **cDAE** by:

```
class mySimulation : public daeSimulation
{
    ... (here we set up a simulation)

    void SetUpOptimization(void)
    {
        ... (here goes a declaration of the obj. function, opt. variables and constraints)
    }
};
```

1.5.12 Running the optimization

Optimizations, like simulations in **pyDAE** can be run in two modes:

1. By using PyQt4 graphical user interface (GUI)
2. From the shell

1. Running an optimization from the GUI (**pyDAE** only):

```
# 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 = simTutorial()
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_()
```

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.

2. Running a simulation from the shell:

In **pyDAE**:

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

# Create Log, NLPSolver, DAESolver, DataReporter, Simulation and Optimization objects
log          = daePythonStdOutLog()
daesolver    = daeIDAS()
nlpsolver    = daeBONMIN()
datareporter = daeTCPIPDataReporter()
simulation   = simTutorial()
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 simulation
optimization.Initialize(simulation, nlpsolver, daesolver, datareporter, log)

# Set the MINLP solver options (optional)
#nlpsolver.SetOption('OPTION', VALUE)
#nlpsolver.LoadOptionsFile("")

# Save the model report and the runtime model report
simulation.m.SaveModelReport(simulation.m.Name + ".xml")
simulation.m.SaveRuntimeModelReport(simulation.m.Name + "-rt.xml")

# Run
optimization.Run()
optimization.Finalize()
```

while in **cDAE** by:

```
// Create Log, NLPSolver, DAESolver, DataReporter, Simulation and Optimization objects
boost::scoped_ptr<daeSimulation_t>        pSimulation(new simTutorial);
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 simulation
pOptimization->Initialize(pSimulation.get(),
                          pNLPSolver.get(),
                          pDAESolver.get(),
                          pDataReporter.get(),
                          pLog.get());

// Run
pOptimization.Run();
pOptimization.Finalize();
```

More information about simulation can be found in *pyDAE User Guide* and [http://{{SERVERNAME}}/api_ref/activity.html#daeOptimization`. Also, do not forget to have a look on *Tutorials*.

1.5.13 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 **Figure 5.**) dialog appears where a user has to select a variable to plot and specify information about domains - fix some of them while leaving another free by selecting * from the list (to create a 2D plot you need one domain free, while for a 3D plot you need two free domains). Typical 2D and 3D plots are given in **Figures 6. and 7.**

[[Image:Screenshot-ChooseVariable.png|thumb|200px|Figure 5. Choose variable dialog for a 2D plot]]

[[Image:Screenshot-Results.png|thumb|200px|Figure 6. Example 2D plot (produced by Matplotlib)]]

[[Image:Screenshot-3Dplot.png|thumb|200px|Figure 7. Example 3D plot (produced by Mayavi2)]]

1.5.14 References

CHAPTER

TWO

PYDAE USER GUIDE

CHAPTER

THREE

PYDAE API REFERENCE

3.1 Module pyCore

3.1.1 Overview

Trt mrt.

3.1.2 Key modelling concepts

...

Classes

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

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

Bases: `Boost.Python.instance`

`__init__`(*(object)arg1*) → None

__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)arg1*) → None

__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

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*]]], *(object)value*) → None

Sets the value of the parameter at the specified domain indexes (as `float` or `quantity`). 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 (as `float` or `quantity`).

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

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*]]], *(object)value*) → None
Sets the value of the variable at the specified domain indexes (as `float` or `quantity`). How many arguments `index1, ..., index8` are used depends on the number of domains that the variable is distributed on.

SetValues (*(daeVariable)self*, *(object)values*) → None
Sets all values of the variable (as `float` or `quantity`).

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

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

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

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

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

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

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

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

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

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

SetAbsoluteTolerances (*(daeVariable)self*, *(object)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)arg1*, *(int)self*) → dict

NumberOfPoints

OverallIndex

ReportingOn

VariableType

npyIDs

npyValues

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

Base model class.

__init__ (*(object)arg1*) → None

__init__((object)self, (str)name [, (daeModel)parent=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*[, *(str)switchTo=''*[, *(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 :
.

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)arg1, (daeDomain)arg2, (daeeDomainBounds)arg3*) → daeDEDI
DistributeOnDomain((daeEquation)arg1, (daeDomain)arg2, (list)arg3) -> daeDEDI

DistributedEquationDomainInfos

EquationExecutionInfos

EquationType

Residual

Scaling

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

Equations

NestedSTNs

StateTransitions

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

Actions

Condition

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

Domains

Export (*(daePort)arg1, (str)arg2, (daeeModelLanguage)arg3, (daeModelExportContext)arg4*) → None

Parameters

SetReportingOn (*(daePort)arg1, (bool)arg2*) → None

Type

Variables

class pyCore.**daeEventPort**
Bases: pyCore.daeObject

EventData

Events

ReceiveEvent (*(daeEventPort)arg1*, *(float)arg2*) → None

RecordEvents

SendEvent (*(daeEventPort)arg1*, *(float)arg2*) → None

Type

class pyCore.**daePortConnection**
Bases: pyCore.daeObject

Equations

PortFrom

PortTo

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

__call__ (*(daeScalarExternalFunction)arg1*) → adouble

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

Name

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

__call__ (*(daeVectorExternalFunction)arg1*) → adouble_array

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

Name

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

__init__ (*(object)arg1*) → None
__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)arg1*) → None
__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)arg1*) → None
__init__((object)self, (daeDomainIndex)domainIndex) -> None

__init__((object)self, (daeIndexRange)indexRange) -> None

DomainIndex

NoPoints

Range

Type

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

`__call__`(*(daeDEDI)self*) → adouble

Domain

DomainBounds

DomainPoints

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

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

RuntimeNode

STN

SendEventPort

SetupNode

StateTo

Type

VariableWrapper

class `pyCore.`**`daeOptimizationVariable`**
Bases: `pyCore.daeOptimizationVariable_t`

LowerBound

Name

StartingPoint

Type

UpperBound

Value

class `pyCore.`**`daeObjectiveFunction`**
Bases: `pyCore.daeObjectiveFunction_t`

`Gradients`

`Name`

`Residual`

`Value`

class `pyCore.`**`daeOptimizationConstraint`**
Bases: `pyCore.daeOptimizationConstraint_t`

`Gradients`

`Name`

`Residual`

`Type`

`Value`

class `pyCore.`**`daeMeasuredVariable`**
Bases: `pyCore.daeMeasuredVariable_t`

`Gradients`

`Name`

`Residual`

`Value`

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

`EquationType`

`Node`

`VariableIndexes`

Functions

`d`	
`dt`	
`Time`	
`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

3.1.3 Logging support

daeLog_t
daeBaseLog
daeFileLog
daeStdOutLog
daeTCPIPLog
daeTCPIPLogServer

class pyCore.**daeLog_t**

Bases: Boost.Python.instance

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

ETA

Enabled

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

Indent

IndentString

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

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

PercentageDone

PrintProgress

Progress

class pyCore.**daeBaseLog**

Bases: pyCore.daeLog_t

DecreaseIndent (*(daeBaseLog)arg1, (int)arg2*) → None

IncreaseIndent (*(daeBaseLog)arg1, (int)arg2*) → None

Message (*(daeBaseLog)arg1, (str)arg2, (int)arg3*) → None
Message((daeBaseLog)arg1, (str)arg2, (int)arg3) -> None

SetProgress (*(daeBaseLog)arg1, (float)arg2*) → None
SetProgress((daeBaseLog)arg1, (float)arg2) -> None

class pyCore.**daeFileLog**

Bases: pyCore.daeBaseLog

Message (*(daeFileLog)arg1, (str)arg2, (int)arg3*) → None
Message((daeFileLog)arg1, (str)arg2, (int)arg3) -> None

class pyCore.**daeStdOutLog**

Bases: pyCore.daeBaseLog

Message (*(daeStdOutLog)arg1*, *(str)arg2*, *(int)arg3*) → None
Message((daeStdOutLog)arg1, (str)arg2, (int)arg3) -> None

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

Message (*(daeTCPIPLog)arg1*, *(str)arg2*, *(int)arg3*) → None
Message((daeTCPIPLog)arg1, (str)arg2, (int)arg3) -> None

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

MessageReceived (*(daeTCPIPLogServer)arg1*, *(str)arg2*) → None
MessageReceived((daeTCPIPLogServer)arg1, (str)arg2) -> None

3.1.4 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.

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.

__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

3.1.5 Auxiliary classes

`daeVariableWrapper`
`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

3.1.6 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

3.1.7 Enumerations

daeeDomainType

Continued on next page

Table 3.8 – continued from previous page

daeeParameterType
daeePortType
daeeDiscretizationMethod
daeeDomainBounds
daeeInitialConditionMode
daeeDomainIndexType
daeeRangeType
daeIndexRangeType
daeeOptimizationVariableType
daeeModelLanguage
daeeConstraintType
daeeUnaryFunctions
daeeBinaryFunctions
daeeSpecialUnaryFunctions
daeeLogicalUnaryOperator
daeeLogicalBinaryOperator
daeeConditionType
daeeActionType
daeeEquationType
daeeModelType

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

3.1.8 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.

3.2 Module pyActivity

3.2.1 Overview

Trt mrt.

`daeSimulation`
`daeOptimization`

daeSimulation

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

`AbsoluteTolerances`

`ActivityAction`

`CleanUpSetupData` (*(daeSimulation)arg1*) → None
CleanUpSetupData((daeSimulation)arg1) -> None

`Constraints`

`CreateEqualityConstraint` (*(daeSimulation)arg1*, *(str)arg2*) → object

`CreateInequalityConstraint` (*(daeSimulation)arg1*, *(str)arg2*) → object

`CurrentTime`

`DAESolver`

`DataReporter`

`Finalize` (*(daeSimulation)arg1*) → None

`IndexMappings`

InitialConditionMode

InitialDerivatives

InitialValues

Initialize (*(daeSimulation)arg1*, *(object)arg2*, *(object)arg3*, *(object)arg4*[, *(bool)CalculateSensitivities=False*]) → None

InputVariables

Integrate (*(daeSimulation)arg1*, *(daeeStopCriterion)arg2*[, *(bool)ReportDataAroundDiscontinuities=True*]) → float

IntegrateForTimeInterval (*(daeSimulation)arg1*, *(float)arg2*[, *(bool)ReportDataAroundDiscontinuities=True*]) → float

IntegrateUntilTime (*(daeSimulation)arg1*, *(float)arg2*, *(daeeStopCriterion)arg3*[, *(bool)ReportDataAroundDiscontinuities=True*]) → float

LoadInitializationValues (*(daeSimulation)arg1*, *(str)arg2*) → None

Log

MeasuredVariables

Model

ModelParameters

NextReportingTime

NumberOfEquations

NumberOfObjectiveFunctions

ObjectiveFunction

ObjectiveFunctions

OptimizationVariables

Pause (*(daeSimulation)arg1*) → None

ReRun (*(daeSimulation)arg1*) → None

RegisterData (*(daeSimulation)arg1*, *(str)arg2*) → None

Reinitialize (*(daeSimulation)arg1*) → None

RelativeTolerance

ReportData (*(daeSimulation)arg1*, *(float)arg2*) → None

ReportingInterval

ReportingTimes

Reset (*(daeSimulation)arg1*) → None

Resume (*(daeSimulation)arg1*) → None

Run (*(daeSimulation)arg1*) → None
Run((daeSimulation)arg1) -> None

SetBinaryOptimizationVariable (*(daeSimulation)arg1*, *(object)arg2*, *(bool)arg3*) → object
SetBinaryOptimizationVariable((daeSimulation)arg1, (object)arg2, (bool)arg3) -> object

SetContinuousOptimizationVariable (*(daeSimulation)arg1*, *(object)arg2*, *(float)arg3*, *(float)arg4*, *(float)arg5*) → object
SetContinuousOptimizationVariable((daeSimulation)arg1, (object)arg2, (float)arg3, (float)arg4, (float)arg5) -> object

SetInputVariable (*(daeSimulation)arg1*, *(object)arg2*) → object
SetInputVariable((daeSimulation)arg1, (object)arg2) -> object

SetIntegerOptimizationVariable (*(daeSimulation)arg1*, *(object)arg2*, *(int)arg3*, *(int)arg4*, *(int)arg5*) → object
SetIntegerOptimizationVariable((daeSimulation)arg1, (object)arg2, (int)arg3, (int)arg4, (int)arg5) -> object

SetMeasuredVariable (*(daeSimulation)arg1*, *(object)arg2*) → object
SetMeasuredVariable((daeSimulation)arg1, (object)arg2) -> object

SetModelParameter (*(daeSimulation)arg1*, *(object)arg2*, *(float)arg3*, *(float)arg4*, *(float)arg5*) → object
SetModelParameter((daeSimulation)arg1, (object)arg2, (float)arg3, (float)arg4, (float)arg5) -> object

SetUpOptimization (*(daeSimulation)arg1*) → None
SetUpOptimization((daeSimulation)arg1) -> None

SetUpParameterEstimation (*(daeSimulation)arg1*) → None
SetUpParameterEstimation((daeSimulation)arg1) -> None

SetUpParametersAndDomains (*(daeSimulation)arg1*) → None
SetUpParametersAndDomains((daeSimulation)arg1) -> None

SetUpSensitivityAnalysis (*(daeSimulation)arg1*) → None
SetUpSensitivityAnalysis((daeSimulation)arg1) -> None

SetUpVariables (*(daeSimulation)arg1*) → None
SetUpVariables((daeSimulation)arg1) -> None

SimulationMode

SolveInitial (*(daeSimulation)arg1*) → None

StoreInitializationValues (*(daeSimulation)arg1*, *(str)arg2*) → None

TimeHorizon

TotalNumberOfVariables

VariableTypes

__init__ (*(object)arg1*) → None

__instance_size__ = 440

__module__ = 'pyActivity'

__reduce__ ()

m

model

daeOptimization

class pyActivity.**daeOptimization**
Bases: pyActivity.daeOptimization_t

Finalize (*(daeOptimization)arg1*) → None

Initialize (*(daeOptimization)arg1*, *(daeSimulation_t)arg2*, *(object)arg3*, *(object)arg4*, *(object)arg5*, *(object)arg6*) → None

Run (*(daeOptimization)arg1*) → None

__init__ (*(object)arg1*) → None

__instance_size__ = 88

__module__ = 'pyActivity'

__reduce__()

3.3 Module pyDataReporting

3.3.1 Overview

Trt mrt.

pyDataReporting

daeDataReporter_t

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

Connect(*(daeDataReporter_t)arg1*, *(str)arg2*, *(str)arg3*) → bool
Connect((daeDataReporter_t)arg1, (str)arg2, (str)arg3) -> None

Disconnect(*(daeDataReporter_t)arg1*) → bool
Disconnect((daeDataReporter_t)arg1) -> None

EndOfData(*(daeDataReporter_t)arg1*) → bool
EndOfData((daeDataReporter_t)arg1) -> None

EndRegistration(*(daeDataReporter_t)arg1*) → bool
EndRegistration((daeDataReporter_t)arg1) -> None

IsConnected(*(daeDataReporter_t)arg1*) → bool
IsConnected((daeDataReporter_t)arg1) -> None

RegisterDomain(*(daeDataReporter_t)arg1*, *(daeDataReporterDomain)arg2*) → bool
RegisterDomain((daeDataReporter_t)arg1, (daeDataReporterDomain)arg2) -> None

RegisterVariable(*(daeDataReporter_t)arg1*, *(daeDataReporterVariable)arg2*) → bool
RegisterVariable((daeDataReporter_t)arg1, (daeDataReporterVariable)arg2) -> None

SendVariable(*(daeDataReporter_t)arg1*, *(daeDataReporterVariableValue)arg2*) → bool
SendVariable((daeDataReporter_t)arg1, (daeDataReporterVariableValue)arg2) -> None

StartNewResultSet(*(daeDataReporter_t)arg1*, *(float)arg2*) → bool
StartNewResultSet((daeDataReporter_t)arg1, (float)arg2) -> None

StartRegistration(*(daeDataReporter_t)arg1*) → bool
StartRegistration((daeDataReporter_t)arg1) -> None

3.4 Module pyIDAS

3.4.1 Overview

Trt mrt.

daeIDAS

daeIDAS

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

SaveMatrixAsXPM(*(daeIDAS)arg1*, *(str)arg2*) → None

SetLASolver(*(daeIDAS)arg1*, *(daeeIDALASolverType)arg2*) → None
SetLASolver((daeIDAS)arg1, (object)arg2) -> None

3.5 Module pyUnits

3.5.1 Overview

Trt mrt.

3.5.2 Classes

unit
quantity

unit

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

baseUnit

unitDictionary

quantity

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

scaleTo(*(quantity)arg1*, *(object)arg2*) → quantity

units

value

valueInSIUnits

CHAPTER

FOUR

TUTORIALS

CHAPTER

FIVE

INDICES AND TABLES

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

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V