

# DAE Tools Project Documentation Release 1.3.1

by Dragan Nikolic

# **CONTENTS**

1	Getti	ng Start	ed with DAE Tools	1
	1.1	Program	nming language	1
	1.2	The ma	in concepts	1
		1.2.1	Core module	2
		1.2.2	Activity module	3
		1.2.3	DataReporting module	3
		1.2.4	IDAS module	3
		1.2.5	Units module	3
	1.3	Runnin	g a simulation	3
	1.4	Runnin	g an optimization	5
	1.5	Model o	development	5
		1.5.1	Models	7
		1.5.2	Distribution domains	7
		1.5.3	Parameters	8
		1.5.4	Variables	8
		1.5.5	Equations	9
		1.5.6	State Transition Networks (discontinuous equations)	10
		1.5.7	Ports	11
		1.5.8	Event Ports	12
		1.5.9	Simulation	12
		1.5.10	Running a simulation	13
		1.5.11	Optimization	15
		1.5.12	Running the optimization	16
		1.5.13	Processing the results	18
		1.5.14	References	18
2	nvD/	AE User	Cuido	19
_	руш	AL USEI	Guide	17
3	pyDA	AE API I	Reference	21
	3.1	Module	pyCore	21
		3.1.1	Overview	21
		3.1.2	Key modelling concepts	21
		3.1.3	Logging support	31
		3.1.4	Autodifferentiation and equation evaluation tree support	32
		3.1.5	Auxiliary classes	34
		3.1.6	Auxiliary functions	35
		3.1.7	Enumerations	35
		3.1.8	Global constants	39
	3.2	Module	pyActivity	40
		3.2.1	Overview	40
	3.3		pyDataReporting	43
		3.3.1	Overview	43
	3.4	Module	pyIDAS	43

			Overview																	
	3.5	Module	e pyUnits .									 								44
		3.5.1	Overview									 								44
		3.5.2	Classes .									 								44
4	Tuto	rials																		45
5	Indic	es and t	ables																	47
Ру	thon I	Module 1	Index																	49
In	dex																			51

# 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 (**DAE**), 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*.)

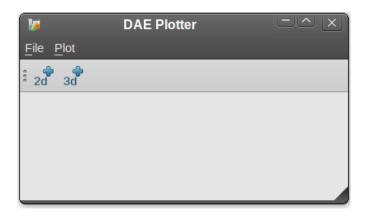


Figure 1.1: **Figure 1.** daePlotter main window.

**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*.:

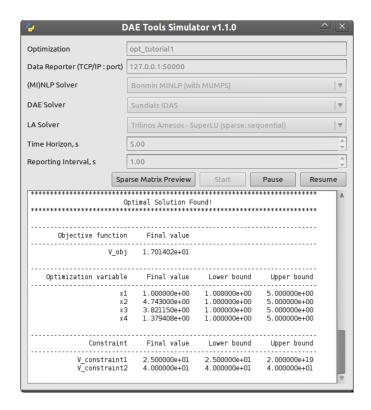


Figure 1.3: Figure 2b. A typical optimization output from DAE Tools

### 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 <sup>1</sup>). 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 (<sup>1</sup>). Also a good overview of the equation-oriented approach and its application in gPROMS is given by Barton & Pantelides (<sup>2</sup>, <sup>3</sup>, <sup>4</sup>).

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

<sup>&</sup>lt;sup>1</sup> Morton, W., Equation-Oriented Simulation and Optimization. *Proc. Indian Natl. Sci. Acad.* 2003, 317-357.

<sup>&</sup>lt;sup>2</sup> 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.

<sup>&</sup>lt;sup>3</sup> 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.

<sup>&</sup>lt;sup>4</sup> 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:

<sup>&</sup>lt;sup>5</sup> 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 distributed 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*..

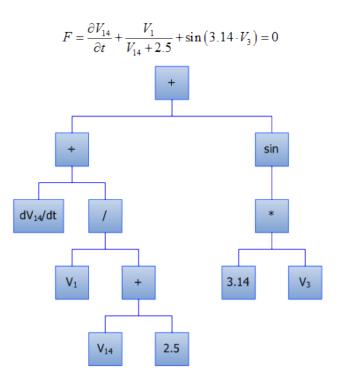


Figure 1.5: Figure 4. DAE Tools equation evaluation tree

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() \leq 2100)
                                        # (Laminar flow)
#... (equations go here)
ELSE_IF(Re() > 2100 and Re() < 10000) # (Transient flow)
#... (equations go here)
                                        # (Turbulent flow)
#... (equations go here)
END_IF()
while in cDAE by:
IF (Re() \leq 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");</pre>
                              switchToState = "Fried");
ON_CONDITION(T() > 110,
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:
```

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:

```
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:
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):

```
# 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
          = daeStdOutLog()
log
solver = daeStdOut = 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<daeDataReporter_t> pDataReporter(daeCreateTCPIPDataReporter());
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
           = 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());
                                        pLog(daeCreateStdOutLog());
boost::scoped_ptr<daeLog_t>
                                       pNLPSolver(new daeBONMINSolver());
boost::scoped_ptr<daeNLPSolver_t>
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
```

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**

# **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
      \underline{\hspace{0.5cm}} init\underline{\hspace{0.5cm}} ((object)arg1) \rightarrow None
            __init__( (object)self,
                                                    (object)units,
                                                                     (float)lowerBound,
                                                                                           (float)upperBound,
                                      (str)name,
           (float)initialGuess, (float)absTolerance) -> None
     AbsoluteTolerance
      InitialGuess
     LowerBound
     Name
      Units
      UpperBound
class pyCore.daeObject
      Bases: Boost.Python.instance
      CanonicalName
     Description
      {\tt GetNameRelativeToParentModel}~((daeObject)self)~\rightarrow {\rm str}
      GetStrippedName((daeObject)self) \rightarrow str
      \texttt{GetStrippedNameRelativeToParentModel}((daeObject)self) \rightarrow str
      ID
      Library
     Model
      Name
      Version
class pyCore.daeDomain
      Bases: pyCore.daeObject
        \_init\__((object)arg1) \rightarrow 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) \rightarrow adouble
      call ((daeDomain)self, (int)index) \rightarrow adouble
      \textbf{CreateArray} ((\textit{daeDomain}) self, (\textit{int}) no \textit{Intervals}) \rightarrow \textbf{None}
      CreateDistributed((daeDomain)self,
                                                        (daeeDiscretizationMethod)discretizationMethod,
                                (int)discretizationOrder, (int)numberOfIntervals, (float)lowerBound,
                                (float)upperBound) \rightarrow None
      DiscretizationMethod
      DiscretizationOrder
      LowerBound
      NumberOfIntervals
     NumberOfPoints
      Points
      Type
```

```
Units
     UpperBound
     npyPoints
class pyCore.daeParameter
     Bases: pyCore.daeObject
     GetValue ((daeParameter)self[, (int)index1[, ...[, (int)index8]]]) \rightarrow 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]]]) \rightarrow 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) \rightarrow 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) \rightarrow None
           Sets all values of the parameter (as float or quantity).
      array((daeParameter)self[, (object)index1[, ...[, (object)index8]]]) \rightarrow 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]]]) <math>\rightarrow 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) \rightarrow None
     Domains
     GetDomainsIndexesMap((daeParameter)self, (int)indexBase) \rightarrow dict
     NumberOfPoints
     ReportingOn
     Units
     npyValues
class pyCore.daeVariable
     Bases: pyCore.daeObject
     GetValue ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]) \rightarrow 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.

3.1. Module pyCore

```
GetQuantity ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]) \rightarrow 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) <math>\rightarrow 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) \rightarrow None
     Sets all values of the variable (as float or quantity).
AssignValue ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]], (object)value) <math>\rightarrow None
AssignValues ((daeVariable)self, (object)values) \rightarrow None
ReAssignValue ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]], (object)value) <math>\rightarrow None
ReAssignValues ((daeVariable)self, (object)values) \rightarrow None
                                                                 \dots, (int)index8]]],
SetInitialCondition ((daeVariable)self |, (int)index1 |,
                            ject)initialCondition) \rightarrow None
SetInitialConditions ((daeVariable)self, (object)initialConditions) <math>\rightarrow None
ReSetInitialCondition((daeVariable)self[, (int)index1[, ...[, (int)index8]]],
                               ject)initialCondition) \rightarrow None
ReSetInitialConditions ((daeVariable)self, (object)initialConditions) \rightarrow None
SetInitialGuess ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]], (object)initialGuess)
                       \rightarrow None
SetInitialGuesses ((daeVariable)self, (object)initialGuesses) <math>\rightarrow None
SetAbsoluteTolerances ((daeVariable)self, (object)tolerances) \rightarrow None
array((daeVariable)self[, (object)index1[, ...[, (object)index8]]]) \rightarrow adouble_array
     Gets the array of variable's values at the specified domain indexes (used to build equation residu-
     als 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]]]) \rightarrow adouble_array
     Gets the array of partial derivatives at the specified domain indexes (used to build equation resid-
     uals 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]]]) \rightarrow 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 pvCore.daeParameter.arrav().
dt_array((daeVariable)self[, (object)index1[, ...[, (object)index8]]]) \rightarrow adouble_array
     Gets the array of time derivatives at the specified domain indexes (used to build equation residu-
     als 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().
  \mathtt{call} ((daeVariable)self[, (int)index1[, ...[, (int)index8]]]) <math>\rightarrow 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.

```
\mathbf{d} ((daeVariable)self, (daeDomain)domain[, (int)index1[, ...[, (int)index8]]]) \rightarrow 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]]]]) \rightarrow 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.
      \texttt{dt} \; ((\textit{daeVariable}) self \big[, (\textit{int}) \textit{index} 1 \big[, ... \big[, (\textit{int}) \textit{index} 8 \big] \big] \big]) \; \rightarrow \; \text{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.
      \texttt{DistributeOnDomain} ((daeVariable)self, (daeDomain)domain) \rightarrow None
      Domains
      GetDomainsIndexesMap((daeVariable)arg1, (int)self) \rightarrow dict
      NumberOfPoints
      OverallIndex
      ReportingOn
      VariableType
      npyIDs
      npyValues
class pyCore.daeModel
      Bases: pyCore.daeObject
      Base model class.
        \_init\_((object)arg1) \rightarrow 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) \rightarrow None
            Connects two event ports.
      ConnectPorts ((daeModel)self, (daePort)portFrom, (daePort)portTo) \rightarrow None:
            Connects two ports.
      \textbf{CreateEquation} ((daeModel)self, (str)name[, (str)description='`[, (float)scaling=1.0]]) \rightarrow
                             daeEquation:
            Creates a new equation. Used to add equations to models or states in state transition networks
      DeclareEquations ((daeModel)self) \rightarrow None:
                User-defined function where all model equations ans state transition networks are declared.
                Must be always implemented in derived classes.
```

A list of domains in the model.

DeclareEquations( (daeModel)self) -> None

Domains

```
ELSE ((daeModel)self) \rightarrow None:
     Adds the last state to a reversible state transition network.
ELSE_IF ((daeModel)self, (daeCondition)condition, (float)eventTolerance=0.0 \rangle \rightarrow None:
     Adds a new state to a reversible state transition network.
END_IF ((daeModel)self) \rightarrow None:
     Finalises a reversible state transition network.
END_STN ((daeModel)self) \rightarrow 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, (daeModelExportCon-
          text)modelExportContext) \rightarrow None:
ExportObjects ((daeModel)self, (list)objects, (daeeModelLanguage)language) <math>\rightarrow str:
IF ((daeModel)self, (daeCondition)condition[, (float)eventTolerance=0.0]) <math>\rightarrow 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]]]]]] \rightarrow None
                                       (daeEventPort)eventPort[,
                                                                            (list)switchToStates=[],
ON_EVENT ((daeModel)self,
             (list) setVariable Values = [] [, \quad (list) trigger Events = [] [, \quad (list) user Defined Actions = [] ]
             )) \rightarrow \text{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) \rightarrow daeState :
STN ((daeModel)self, (str)stnName) \rightarrow daeSTN:
STNs
```

A list of state transition networks in the model.

```
SWITCH_TO ((daeModel)self, (str)targetState, (daeCondition)condition[, (float)eventTolerance=0.0
                     ) \rightarrow \text{None}:
      SaveModelReport ((daeModel)self, (str)xmlFilename) \rightarrow None:
      SaveRuntimeModelReport ((daeModel)self, (str)xmlFilename) \rightarrow None:
      SetReportingOn ((daeModel)self, (bool)reportingOn) \rightarrow 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) \rightarrow
           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
     \textbf{Export} \; ((\textit{daePort}) \textit{arg1}, \; (\textit{str}) \textit{arg2}, \; (\textit{daeeModelLanguage}) \textit{arg3}, \; (\textit{daeModelExportContext}) \textit{arg4}) \; \rightarrow \;
                None
     Parameters
      \textbf{SetReportingOn} \; ((\textit{daePort}) \textit{arg1}, (\textit{bool}) \textit{arg2}) \; \rightarrow \; \textit{None}
      Type
      Variables
```

3.1. Module pyCore

```
class pyCore.daeEventPort
      Bases: pyCore.daeObject
     EventData
      Events
     ReceiveEvent ((daeEventPort)arg1, (float)arg2) \rightarrow None
      RecordEvents
      SendEvent ((daeEventPort)arg1, (float)arg2) \rightarrow None
      Type
class pyCore.daePortConnection
      Bases: pyCore.daeObject
      Equations
      PortFrom
     PortTo
class pyCore.daeScalarExternalFunction
      Bases: Boost.Python.instance
      \_\_\mathtt{call}\_\_((daeScalarExternalFunction)arg1) \rightarrow adouble
      Calculate ((daeScalarExternalFunction)arg1, (tuple)arg2, (dict)arg3) \rightarrow object
           Calculate( (daeScalarExternalFunction)arg1, (tuple)arg2, (dict)arg3) -> None
     Name
class pyCore.daeVectorExternalFunction
      Bases: Boost.Python.instance
      __call__((daeVectorExternalFunction)arg1) \rightarrow adouble_array
      Calculate ((daeVectorExternalFunction)arg1, (tuple)arg2, (dict)arg3) \rightarrow list
           Calculate( (daeVectorExternalFunction)arg1, (tuple)arg2, (dict)arg3) -> None
     Name
class pyCore.daeDomainIndex
      Bases: Boost.Python.instance
      \underline{\hspace{0.5cm}} init\underline{\hspace{0.5cm}} ((object)arg1) \rightarrow 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
      \underline{\hspace{0.5cm}} init\underline{\hspace{0.5cm}} ((object)arg1) \rightarrow 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
      \underline{\hspace{0.5cm}} init\underline{\hspace{0.5cm}} ((object)arg1) \rightarrow None
          __init__( (object)self, (daeDomainIndex)domainIndex) -> None
          __init__( (object)self, (daeIndexRange)indexRange) -> None
     DomainIndex
     NoPoints
     Range
     Type
class pyCore.daeDEDI
     Bases: pyCore.daeObject
     \_\_\mathtt{call}\_\_((daeDEDI)self) \rightarrow \mathtt{adouble}
     Domain
     DomainBounds
     DomainPoints
class pyCore.daeAction
     Bases: pyCore.daeObject
     Execute ((daeAction)arg1) \rightarrow None
          Execute( (daeAction)arg1) -> None
     RuntimeNode
     STN
     SendEventPort
     SetupNode
     StateTo
     Type
     VariableWrapper
class pyCore.daeOptimizationVariable
     Bases: \verb"pyCore.daeOptimizationVariable_t"
     LowerBound
     Name
     StartingPoint
     Type
     UpperBound
     Value
```

```
{\bf class} \; {\tt pyCore.daeObjectiveFunction}
     Bases: \verb"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
{\bf class} \; {\tt pyCore.daeEquationExecutionInfo}
     Bases: Boost.Python.instance
     EquationType
    Node
     VariableIndexes
```

#### **Functions**

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

```
\begin{array}{l} {\tt pyCore.d}\,((adouble)arg1,(daeDomain)ad) \, \to {\tt adouble} \\ {\tt pyCore.dt}\,((adouble)ad) \, \to {\tt adouble} \\ {\tt pyCore.Time}\,() \, \to {\tt adouble} \\ {\tt pyCore.Constant}\,((float)value) \, \to {\tt adouble} \\ {\tt Constant}(\,({\tt object}){\tt value}) \, \to {\tt adouble} \\ {\tt pyCore.Array}\,((list)values) \, \to {\tt adouble\_array} \\ {\tt pyCore.Sum}\,((adouble\_array)adarray) \, \to {\tt adouble} \\ \end{array}
```

```
pyCore.Product ((adouble_array)adarray) \rightarrow adouble
pyCore.Integral ((adouble_array)adarray) \rightarrow adouble
pyCore.Average ((adouble_array)adarray) \rightarrow adouble
```

#### 3.1.3 Logging support

daeLog_t
daeBaseLog
daeFileLog
daeStdOutLog
daeTCPIPLog
daeTCPIPLogServer

```
class pyCore.daeLog_t
```

Bases: Boost.Python.instance

**DecreaseIndent**  $((daeLog\_t)arg1, (int)arg2) \rightarrow None$  DecreaseIndent( $(daeLog\_t)arg1, (int)arg2) \rightarrow None$ 

ETA

Enabled

IncreaseIndent  $((daeLog\_t)arg1, (int)arg2) \rightarrow None$ IncreaseIndent( $(daeLog\_t)arg1, (int)arg2) \rightarrow None$ 

Indent

IndentString

**JoinMessages**  $((daeLog\_t)arg1, (str)arg2) \rightarrow str$  JoinMessages $((daeLog\_t)arg1, (str)arg2) \rightarrow None$ 

**Message**  $((daeLog\_t)arg1, (str)arg2, (int)arg3) \rightarrow None$  Message $((daeLog\_t)arg1, (str)arg2, (int)arg3) \rightarrow None$ 

PercentageDone

PrintProgress

**Progress** 

#### ${f class}$ pyCore. ${f daeBaseLog}$

Bases: pyCore.daeLog\_t

**DecreaseIndent**  $((daeBaseLog)arg1, (int)arg2) \rightarrow None$ 

IncreaseIndent  $((daeBaseLog)arg1, (int)arg2) \rightarrow None$ 

**Message**  $((daeBaseLog)arg1, (str)arg2, (int)arg3) \rightarrow None$  Message $((daeBaseLog)arg1, (str)arg2, (int)arg3) \rightarrow None$ 

 $\begin{tabular}{ll} SetProgress ((daeBaseLog)arg1, (float)arg2) \rightarrow None \\ SetProgress ( (daeBaseLog)arg1, (float)arg2) \rightarrow None \\ \end{tabular}$ 

class pyCore.daeFileLog

 $Bases: \verb"pyCore.daeBaseLog"$ 

**Message** ( $(daeFileLog)arg1, (str)arg2, (int)arg3) \rightarrow None$  Message( $(daeFileLog)arg1, (str)arg2, (int)arg3) \rightarrow None$ 

class pyCore.daeStdOutLog

 $Bases: \verb"pyCore.daeBaseLog"$ 

**Message**  $((daeStdOutLog)arg1, (str)arg2, (int)arg3) \rightarrow None$ Message $((daeStdOutLog)arg1, (str)arg2, (int)arg3) \rightarrow None$ 

#### class pyCore.daeTCPIPLog

Bases: pyCore.daeBaseLog

**Message** ( $(daeTCPIPLog)arg1, (str)arg2, (int)arg3) \rightarrow None Message( (daeTCPIPLog)arg1, (str)arg2, (int)arg3) -> None$ 

#### class pyCore.daeTCPIPLogServer

Bases: Boost.Python.instance

**MessageReceived** ( $(daeTCPIPLogServer)arg1, (str)arg2) \rightarrow None$  MessageReceived(  $(daeTCPIPLogServer)arg1, (str)arg2) \rightarrow None$ 

#### 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

 $Bases: { t Boost.Python.instance}$ 

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) \rightarrow int:
```

Returns the size of the adouble\_array object.

```
\_\_getitem\_ ((adouble_array)self, (int)index) \rightarrow adouble:
```

Gets an adouble object at the specified index.

 $\_$  setitem $\_$  ((adouble\_array)self, (int)index, (adouble)value)  $\rightarrow$  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
```

```
\texttt{pyCore.Sin} \, ((\textit{adouble}) \textit{arg1}) \, \rightarrow \text{adouble}
      Sin( (adouble_array)arg1) -> adouble_array
pyCore.Cos ((adouble)arg1) \rightarrow adouble
      Cos( (adouble array)arg1) -> adouble array
pyCore. Tan ((adouble)arg1) \rightarrow adouble
      Tan( (adouble_array)arg1) -> adouble_array
pyCore.ASin ((adouble)arg1) \rightarrow adouble
      ASin((adouble_array)arg1) -> adouble_array
pyCore.ACos ((adouble)arg1) \rightarrow adouble
      ACos( (adouble_array)arg1) -> adouble_array
pyCore.ATan ((adouble)arg1) \rightarrow adouble
      ATan( (adouble_array)arg1) -> adouble_array
pyCore. Sinh ((adouble)arg1) \rightarrow adouble
pyCore.Cosh((adouble)arg1) \rightarrow adouble
pyCore. Tanh ((adouble)arg1) \rightarrow adouble
pyCore. ASinh ((adouble)arg1) \rightarrow adouble
pyCore. ACosh((adouble)arg1) \rightarrow adouble
pyCore.ATanh ((adouble)arg1) \rightarrow adouble
\texttt{pyCore.ATan2} \; ((adouble)arg1, (adouble)arg2) \; \rightarrow \text{adouble}
pyCore.Ceil ((adouble)arg1) \rightarrow adouble
      Ceil( (adouble_array)arg1) -> adouble_array
pyCore.Floor ((adouble)arg1) \rightarrow adouble
      Floor((adouble_array)arg1) -> adouble_array
pyCore. Pow ((adouble)arg1, (float)arg2) \rightarrow adouble
      Pow( (adouble)arg1, (adouble)arg2) -> adouble
      Pow( (float)arg1, (adouble)arg2) -> adouble
pyCore.Abs ((adouble)arg1) \rightarrow adouble
      Abs( (adouble_array)arg1) -> adouble_array
pyCore. Min ((adouble)arg1, (adouble)arg2) \rightarrow adouble
      Min((float)arg1, (adouble)arg2) -> adouble
      Min( (adouble)arg1, (float)arg2) -> adouble
      Min( (adouble_array) adarray) -> adouble
pyCore. Max ((adouble)arg1, (adouble)arg2) \rightarrow 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='']) \rightarrow 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) \rightarrow object
       \_setitem\_ ((daeConfig)self, (object)propertyPath, (object)value) \rightarrow None
      GetBoolean((daeConfig)self, (str)propertyPath[, (bool)defaultValue]) \rightarrow bool
      GetFloat((daeConfig)self, (str)propertyPath[, (float)defaultValue]) \rightarrow float
      GetInteger ((daeConfig)self, (str)propertyPath[, (int)defaultValue]) \rightarrow int
       GetString ((daeConfig)self, (str)propertyPath[, (str)defaultValue]) \rightarrow str
      Reload((daeConfig)self) \rightarrow None
       \textbf{SetBoolean} ((\textit{daeConfig}) \textit{self}, (\textit{str}) \textit{propertyPath}, (\textit{bool}) \textit{value}) \rightarrow \textbf{None}
       \textbf{SetFloat} ((\textit{daeConfig}) \textit{self}, (\textit{str}) \textit{propertyPath}, (\textit{float}) \textit{value}) \rightarrow \textbf{None}
       SetInteger ((daeConfig)self, (str)propertyPath, (int)value) \rightarrow None
       SetString ((daeConfig)self, (str)propertyPath, (str)value) \rightarrow None
      has\_key((daeConfig)self, (object)propertyPath) \rightarrow object
```

# 3.1.6 Auxiliary functions

daeGetConfig
daeVersion
daeVersionMajor
daeVersionMinor
daeVersionBuild

```
\label{eq:pycore.daeGetConfig} \begin{split} &\operatorname{pyCore.daeGetConfig}() \to \operatorname{object} \\ &\operatorname{pyCore.daeVersion}(\big[ (bool) includeBuild = False \big]) \to \operatorname{str} \\ &\operatorname{pyCore.daeVersionMajor}() \to \operatorname{int} \\ &\operatorname{pyCore.daeVersionBuild}() \to \operatorname{int} \end{split}
```

## 3.1.7 Enumerations

daeeDomainType

Continued on next page

3.1. Module pyCore

	4.	•	
Table 3.8 -	– continued t	rom previous	nage

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

 $\verb"eBool"=pyCore.daeeParameterType.eBool"$ 

eInteger = pyCore.daeeParameterType.eInteger

ePTUnknown = pyCore.daeeParameterType.ePTUnknown

eReal = pyCore.daeeParameterType.eReal

#### class pyCore.daeePortType

Bases: Boost.Python.enum

 $\verb|eInletPort| = pyCore.daeePortType.eInletPort|$ 

eOutletPort = pyCore.daeePortType.eOutletPort

 $\verb"eUnknownPort" = pyCore.daeePortType.eUnknownPort"$ 

## class pyCore.daeeDiscretizationMethod

Bases: Boost.Python.enum

eBFDM = pyCore.daeeDiscretizationMethod.eBFDM

eCFDM = pyCore.daeeDiscretizationMethod.eCFDM

 $\verb"eCustomDM" = pyCore.daeeDiscretizationMethod.eCustomDM"$ 

 $\verb"eDMUnknown" = pyCore.daeeDiscretizationMethod.eDMUnknown"$ 

eFFDM = pyCore.daeeDiscretizationMethod.eFFDM

# ${\bf class} \; {\tt pyCore.daeeDomainBounds}$

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
```

3.1. Module pyCore

```
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
    eSgrt = pyCore.daeeUnaryFunctions.eSgrt
    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

 $int(x[, base]) \rightarrow integer$ 

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

 $int(x[, base]) \rightarrow integer$ 

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  $((daeSimulation)arg1) \rightarrow None$ CleanUpSetupData( $(daeSimulation)arg1) \rightarrow None$ 

#### Constraints

 $CreateEqualityConstraint((daeSimulation)arg1, (str)arg2) \rightarrow object$ 

 $\textbf{CreateInequalityConstraint} \; ((\textit{daeSimulation}) \\ \textit{arg1}, (\textit{str}) \\ \textit{arg2}) \; \rightarrow \text{object}$ 

CurrentTime

**DAESolver** 

DataReporter

**Finalize**  $((daeSimulation)arg1) \rightarrow None$ 

IndexMappings

```
InitialConditionMode
InitialDerivatives
InitialValues
                                                                                  (object)arg4,
Initialize((daeSimulation)arg1,
                                           (object)arg2,
                                                               (object)arg3,
               (bool)CalculateSensitivities=False ) \rightarrow None
InputVariables
Integrate ((daeSimulation)arg1, (daeeStopCriterion)arg2[, (bool)ReportDataAroundDiscontinuities=True
                                                                                    (float)arg2[,
IntegrateForTimeInterval ((daeSimulation)arg1,
                                   (bool)ReportDataAroundDiscontinuities=True ) \rightarrow float
IntegrateUntilTime ((daeSimulation)arg1,
                                                     (float)arg2,
                                                                      (daeeStopCriterion)arg3,
                           (bool)ReportDataAroundDiscontinuities=True ) \rightarrow float
\textbf{LoadInitializationValues} ((\textit{daeSimulation}) \textit{arg1}, (\textit{str}) \textit{arg2}) \rightarrow \textit{None}
MeasuredVariables
Model
ModelParameters
NextReportingTime
NumberOfEquations
NumberOfObjectiveFunctions
ObjectiveFunction
ObjectiveFunctions
OptimizationVariables
Pause ((daeSimulation)arg1) \rightarrow None
ReRun ((daeSimulation)arg1) \rightarrow None
RegisterData ((daeSimulation)arg1, (str)arg2) \rightarrow None
Reinitialize ((daeSimulation)arg1) \rightarrow None
RelativeTolerance
ReportData ((daeSimulation)arg1, (float)arg2) \rightarrow None
ReportingInterval
ReportingTimes
Reset ((daeSimulation)arg1) \rightarrow None
Resume ((daeSimulation)arg1) \rightarrow None
Run ((daeSimulation)arg1) \rightarrow None
     Run( (daeSimulation)arg1) -> None
SetBinaryOptimizationVariable ((daeSimulation)arg1, (object)arg2, (bool)arg3) \rightarrow ob-
                                          ject
     SetBinaryOptimizationVariable((daeSimulation)arg1, (object)arg2, (bool)arg3) -> object
SetContinuousOptimizationVariable ((daeSimulation)arg1, (object)arg2, (float)arg3,
                                                (float)arg4, (float)arg5) \rightarrow object
     SetContinuousOptimizationVariable( (daeSimulation)arg1, (object)arg2, (float)arg3, (float)arg4,
     (float)arg5) -> object
```

```
SetInputVariable ((daeSimulation)arg1, (object)arg2) \rightarrow object
           SetInputVariable( (daeSimulation)arg1, (object)arg2) -> object
      SetIntegerOptimizationVariable ((daeSimulation)arg1,
                                                                                (object)arg2,
                                                                                                  (int)arg3,
                                                     (int)arg4, (int)arg5) \rightarrow object
            SetIntegerOptimizationVariable( (daeSimulation)arg1, (object)arg2, (int)arg3, (int)arg4, (int)arg5) ->
            object
      SetMeasuredVariable ((daeSimulation)arg1, (object)arg2) \rightarrow object
           SetMeasuredVariable( (daeSimulation)arg1, (object)arg2) -> object
      SetModelParameter ((daeSimulation)arg1, (object)arg2, (float)arg3, (float)arg4, (float)arg5) \rightarrow
                                 object
            SetModelParameter((daeSimulation)arg1, (object)arg2, (float)arg3, (float)arg4, (float)arg5) -> object
      SetUpOptimization ((daeSimulation)arg1) \rightarrow None
           SetUpOptimization( (daeSimulation)arg1) -> None
      \texttt{SetUpParameterEstimation}((daeSimulation)arg1) \rightarrow \texttt{None}
           SetUpParameterEstimation((daeSimulation)arg1) -> None
      {\tt SetUpParametersAndDomains} \ ((\textit{daeSimulation}) \textit{arg1}) \ \rightarrow None
            SetUpParametersAndDomains( (daeSimulation)arg1) -> None
      SetUpSensitivityAnalysis ((daeSimulation)arg1) \rightarrow None
           SetUpSensitivityAnalysis( (daeSimulation)arg1) -> None
      SetUpVariables ((daeSimulation)arg1) \rightarrow None
           SetUpVariables((daeSimulation)arg1) -> None
      SimulationMode
      SolveInitial ((daeSimulation)arg1) \rightarrow None
      StoreInitializationValues ((daeSimulation)arg1, (str)arg2) \rightarrow None
      TimeHorizon
      TotalNumberOfVariables
      VariableTypes
      \underline{\hspace{0.5cm}} init\underline{\hspace{0.5cm}} ((object)arg1) \rightarrow None
       instance size =440
      __module__ = 'pyActivity'
      __reduce__()
      m
      model
daeOptimization
class pyActivity.daeOptimization
      Bases: pyActivity.daeOptimization_t
      Finalize ((daeOptimization)arg1) \rightarrow None
      Initialize ((daeOptimization)arg1, (daeSimulation_t)arg2, (object)arg3, (object)arg4, (ob-
                      ject)arg5, (object)arg6) \rightarrow None
      Run ((daeOptimization)arg1) \rightarrow None
      \underline{\hspace{0.5cm}} init \underline{\hspace{0.5cm}} ((object)arg1) \rightarrow 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  $((daeDataReporter\_t)arg1, (str)arg2, (str)arg3) \rightarrow bool$ Connect( $(daeDataReporter\_t)arg1, (str)arg2, (str)arg3) \rightarrow None$ 

**Disconnect** ((daeDataReporter\_t)arg1) → bool Disconnect( (daeDataReporter\_t)arg1) -> None

$$\label{eq:continuous_problem} \begin{split} \textbf{EndOfData} & ((\textit{daeDataReporter\_t}) \textit{arg1}) \rightarrow \textit{bool} \\ & EndOfData( (\textit{daeDataReporter\_t}) \textit{arg1}) \rightarrow \textit{None} \end{split}$$

**EndRegistration**  $((daeDataReporter\_t)arg1) \rightarrow bool$  EndRegistration $((daeDataReporter\_t)arg1) \rightarrow None$ 

 $\begin{tabular}{l} \textbf{IsConnected} ((\textit{daeDataReporter\_t}) \textit{arg1}) \rightarrow bool \\ \textbf{IsConnected} ((\textit{daeDataReporter\_t}) \textit{arg1}) \rightarrow bool \\ \end{tabular}$ 

 $\label{eq:RegisterDomain} \textbf{RegisterDomain} ((daeDataReporter\_t)arg1, (daeDataReporterDomain)arg2) \rightarrow bool \\ RegisterDomain( (daeDataReporter\_t)arg1, (daeDataReporterDomain)arg2) \rightarrow None \\ \textbf{None} = (daeDataReporterDomain)arg2) \rightarrow (daeDataReporterDo$ 

**RegisterVariable** ( $(daeDataReporter\_t)arg1$ ,  $(daeDataReporterVariable)arg2) \rightarrow bool$  RegisterVariable( $(daeDataReporter\_t)arg1$ , (daeDataReporterVariable)arg2) -> None

**SendVariable** ( $(daeDataReporter\_t)arg1$ ,  $(daeDataReporterVariableValue)arg2) \rightarrow bool$  SendVariable( $(daeDataReporter\_t)arg1$ , (daeDataReporterVariableValue)arg2) -> None

 $\begin{tabular}{ll} \textbf{StartNewResultSet} & ((daeDataReporter\_t)arg1, (float)arg2) & \rightarrow bool \\ \textbf{StartNewResultSet} & (daeDataReporter\_t)arg1, (float)arg2, (float)arg2, (float)arg2, (float)arg3, (float)arg$ 

 $\begin{tabular}{ll} \textbf{StartRegistration} & ((\textit{daeDataReporter\_t}) \textit{arg1}) & \rightarrow \textbf{bool} \\ \textbf{StartRegistration} & ((\textit{daeDataReporter\_t}) \textit{arg1}) & \rightarrow \textbf{None} \\ \end{tabular}$ 

# 3.4 Module pyIDAS

## 3.4.1 Overview

Trt mrt.

daeIDAS

#### daelDAS

class pyIDAS.daeIDAS

Bases: pyIDAS.daeDAESolver\_t

```
\label{eq:saveMatrixAsXPM} \begin{subarray}{ll} \textbf{SaveMatrixAsXPM} ((daeIDAS)arg1, (str)arg2) $\rightarrow$ None \\ \textbf{SetLASolver} ((daeIDAS)arg1, (daeeIDALASolverType)arg2) $\rightarrow$ None \\ \textbf{SetLASolver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \begin{subarray}{ll} \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \begin{subarray}{ll} \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \begin{subarray}{ll} \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \textbf{Solver} ((daeIDAS)arg1, (object)arg2) $\rightarrow$ None \\ \begin{subarray}{ll} \textbf{Solver} ((daeIDAS)arg1, (object)arg2, (object)arg2, (object)arg2, (object)arg3, (object)arg3
```

# 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

pyActivity, 40 pyCore, 21 pyDataReporting, 43 pyIDAS, 43 pyUnits, 44

# **INDEX**

Symbols	ActiveState (pyCore.daeSTN attribute), 27
_and() (pyCore.daeCondition method), 33	ActivityAction (pyActivity.daeSimulation attribute), 40
call() (pyCore.daeDEDI method), 29	adouble (class in pyCore), 32
call() (pyCore.daeDomain method), 22	adouble_array (class in pyCore), 32
call() (pyCore.daeParameter method), 23	Array() (in module pyCore), 30
call() (pyCore.daeScalarExternalFunction	array() (pyCore.daeParameter method), 23
method), 28	array() (pyCore.daeVariable method), 24
call() (pyCore.daeVariable method), 24	ASin() (in module pyCore), 34
call() (pyCore.daeVectorExternalFunction	ASinh() (in module pyCore), 34
method), 28	AssignValue() (pyCore.daeVariable method), 24
contains() (pyCore.daeConfig method), 35	AssignValues() (pyCore.daeVariable method), 24
getitem() (pyCore.adouble_array method), 32	ATan() (in module pyCore), 34
getitem() (pyCore.daeConfig method), 35	ATan2() (in module pyCore), 34
getitem() (pyCore.daeConing inethod), 22	ATanh() (in module pyCore), 34
init() (pyActivity.daeOptimization method), 42	Average() (in module pyCore), 31
init() (pyActivity.daeSimulation method), 42	
init() (pyCore.daeArrayRange method), 29	В
init() (pyCore.daeDomain method), 22	baseUnit (pyUnits.unit attribute), 44
init() (pyCore.daeDomainIndex method), 28	
init() (pyCore.daeIndexRange method), 28	C
init() (pyCore.daeModel method), 25	Calculate() (pyCore.daeScalarExternalFunction
init() (pyCore.daeVariableType method), 22	method), 28
init() (pyCore.daeVariableWrapper method), 34	Calculate() (pyCore.daeVectorExternalFunction
instance_size (pyActivity.daeOptimization at-	method), 28
tribute), 42	CanonicalName (pyCore.daeObject attribute), 22
instance_size (pyActivity.daeSimulation attribute),	Ceil() (in module pyCore), 34
42	CleanUpSetupData() (pyActivity.daeSimulation
_len() (pyCore.adouble_array method), 32	method), 40
module (pyActivity.daeOptimization attribute), 42	cnAlgebraic (in module pyCore), 40
module (pyActivity.daeSimulation attribute), 42	cnAssigned (in module pyCore), 40
_or_() (pyCore.daeCondition method), 33	cnDifferential (in module pyCore), 40
_reduce() (pyActivity.daeOptimization method), 42	ComponentArrays (pyCore.daeModel attribute), 25
reduce() (pyActivity.daeSimulation method), 42	Components (pyCore.daeModel attribute), 25
setitem() (pyCore.adouble_array method), 32	Condition (pyCore.daeStateTransition attribute), 27
setitem() (pyCore.daeConfig method), 35	Connect() (pyDataReporting.daeDataReporter_t
	method), 43
A	ConnectEventPorts() (pyCore.daeModel method), 25
Abs() (in module pyCore), 34	ConnectPorts() (pyCore.daeModel method), 25
AbsoluteTolerance (pyCore.daeVariableType attribute),	Constant() (in module pyCore), 30
22	Constraints (pyActivity.daeSimulation attribute), 40
AbsoluteTolerances (pyActivity.daeSimulation at-	Cos() (in module pyCore), 34
tribute), 40	Cosh() (in module pyCore), 34
ACos() (in module pyCore), 34	CreateArray() (pyCore.daeDomain method), 22
ACosh() (in module pyCore), 34	CreateDistributed() (pyCore.daeDomain method), 22
Actions (pyCore.daeStateTransition attribute), 27	CreateEqualityConstraint() (pyActivity.daeSimulation
redons (py corolados and transmon announc), 27	method) 40

CreateEquation() (pyCore.daeModel method), 25 CreateInequalityConstraint() (pyActivity.daeSimulation method), 40	daePort (class in pyCore), 27 daePortConnection (class in pyCore), 28 daeScalarExternalFunction (class in pyCore), 28
CurrentTime (pyActivity.daeSimulation attribute), 40	daeSimulation (class in pyCoie), 28
D	DAESolver (pyActivity.daeSimulation attribute), 40
	daeState (class in pyCore), 27
d() (in module pyCore), 30	daeStateTransition (class in pyCore), 27
d() (pyCore.daeVariable method), 24	daeStdOutLog (class in pyCore), 31
d2() (pyCore.daeVariable method), 25	daeSTN (class in pyCore), 27
d2_array() (pyCore.daeVariable method), 24	daeTCPIPLog (class in pyCore), 32
d_array() (pyCore.daeVariable method), 24	daeTCPIPLogServer (class in pyCore), 32
daeAction (class in pyCore), 29	daeVariable (class in pyCore), 23
daeArrayRange (class in pyCore), 29	daeVariableType (class in pyCore), 22
daeBaseLog (class in pyCore), 31	daeVariableWrapper (class in pyCore), 34
daeCondition (class in pyCore), 33	daeVectorExternalFunction (class in pyCore), 28
daeConfig (class in pyCore), 35	daeVersion() (in module pyCore), 35
daeDataReporter_t (class in pyDataReporting), 43	daeVersionBuild() (in module pyCore), 35
daeDEDI (class in pyCore), 29	daeVersionMajor() (in module pyCore), 35
daeDomain (class in pyCore), 22	daeVersionMinor() (in module pyCore), 35
daeDomainIndex (class in pyCore), 28	DataReporter (pyActivity.daeSimulation attribute), 40
daeeActionType (class in pyCore), 39	DeclareEquations() (pyCore.daeModel method), 25
daeeBinaryFunctions (class in pyCore), 38	DecreaseIndent() (pyCore.daeBaseLog method), 31
daeeConditionType (class in pyCore), 39	DecreaseIndent() (pyCore.daeLog_t method), 31
daeeConstraintType (class in pyCore), 37	DEDI (pyCore.daeDomainIndex attribute), 28
daeeDiscretizationMethod (class in pyCore), 36	Derivative (pyCore.adouble attribute), 32
daeeDomainBounds (class in pyCore), 36	Description (pyCore.daeObject attribute), 22
daeeDomainIndexType (class in pyCore), 37	Disconnect() (pyDataReporting.daeDataReporter_t
daeeDomainType (class in pyCore), 36	method), 43
daeeEquationType (class in pyCore), 39	DiscretizationMethod (pyCore.daeDomain attribute),
daeeInitialConditionMode (class in pyCore), 37	22
daeeLogicalBinaryOperator (class in pyCore), 39	DiscretizationOrder (pyCore.daeDomain attribute), 22
daeeLogicalUnaryOperator (class in pyCore), 38	DistributedEquationDomainInfos (pyCore.daeEquation
daeeModelLanguage (class in pyCore), 37	attribute), 27
daeeModelType (class in pyCore), 39	DistributeOnDomain() (pyCore.daeEquation method),
daeeOptimizationVariableType (class in pyCore), 37	27
daeeParameterType (class in pyCore), 36	DistributeOnDomain() (pyCore.daeParameter method).
daeePortType (class in pyCore), 36	23
daeEquation (class in pyCore), 27	DistributeOnDomain() (pyCore.daeVariable method).
daeEquation (class in pyCore), 27 daeEquationExecutionInfo (class in pyCore), 30	25
daeeRangeType (class in pyCore), 37	Domain (pyCore.daeDEDI attribute), 29
daeeSpecialUnaryFunctions (class in pyCore), 38	Domain (pyCore.daeIndexRange attribute), 28
daeeUnaryFunctions (class in pyCore), 38	DomainBounds (pyCore.daeDEDI attribute), 29
daeEventPort (class in pyCore), 27	DomainIndex (pyCore.daeArrayRange attribute), 29
daeFileLog (class in pyCore), 31	DomainIndexes (pyCore.daeVariableWrapper at-
daeGetConfig() (in module pyCore), 35	tribute), 35
daeIDAS (class in pyIDAS), 43	DomainPoints (pyCore.daeDEDI attribute), 29
daeIF (class in pyCore), 27	Domains (pyCore.daeModel attribute), 25
daeIndexRange (class in pyCore), 28	Domains (pyCore.daeParameter attribute), 23
daeIndexRangeType (class in pyCore), 37	Domains (pyCore.daePort attribute), 27
daeLog_t (class in pyCore), 31	Domains (pyCore.daeVariable attribute), 25
daeMeasuredVariable (class in pyCore), 30	dt() (in module pyCore), 30
= -	dt() (pyCore.daeVariable method), 25
daeModel (class in pyCore), 25	dt_array() (pyCore.daeVariable method), 24
daeObject (class in pyCore), 22	ar_araj () (p) corelate variable method), 2+
daeObjectiveFunction (class in pyCore), 29	E
daeOptimization (class in pyActivity), 42	
daeOptimizationConstraint (class in pyCore), 30	eAbs (pyCore.daeeUnaryFunctions attribute), 38
daeOptimizationVariable (class in pyCore), 29	eAlgebraic (pyCore.daeeEquationType attribute), 39
daeParameter (class in pyCore), 23	

eAlgebraicValuesProvided	(py-	eEQ (pyCore.daeeConditionType attribute), 39
Core.daeeInitialConditionMode 37	attribute),	eEqualityConstraint (pyCore.daeeConstraintType attribute), 38
eAllPointsInDomain (pyCore.daeIndexRange tribute), 37	Type at-	eETUnknown (pyCore.daeeEquationType attribute), 39 eExp (pyCore.daeeUnaryFunctions attribute), 38
eAnd (pyCore.daeeLogicalBinaryOperator	attributa)	eExplicitODE (pyCore.daeeEquationType attribute), 39
39		eFFDM (pyCore.daeeDiscretizationMethod attribute),
eArcCos (pyCore.daeeUnaryFunctions attribu		36
eArcSin (pyCore.daeeUnaryFunctions attribut		eFloor (pyCore.daeeUnaryFunctions attribute), 38
eArcTan (pyCore.daeeUnaryFunctions attribu		eGT (pyCore.daeeConditionType attribute), 39
eArray (pyCore.daeeDomainType attribute), 3		eGTEQ (pyCore.daeeConditionType attribute), 39
eAverage (pyCore.daeeSpecialUnaryFuncti tribute), 38	ons at-	eICTUnknown (pyCore.daeeInitialConditionMode attribute), 37
eBFDM (pyCore.daeeDiscretizationMethod 36	attribute),	eImplicitODE (pyCore.daeeEquationType attribute), 39 eIncrementedDomainIterator (py-
eBFUnknown (pyCore.daeeBinaryFunctions	attribute)	eIncrementedDomainIterator (py- Core.daeeDomainIndexType attribute),
38		37
eBinary Variable (pyCore.daeeOptimizationVa attribute), 37	riableType	eInequalityConstraint (pyCore.daeeConstraintType attribute), 38
eBool (pyCore.daeeParameterType attribute),		eInletPort (pyCore.daeePortType attribute), 36
eBOUnknown (pyCore.daeeLogicalBinaryOptribute), 39	erator at-	eInteger (pyCore.daeeParameterType attribute), 36 eIntegerVariable (py-
eCDAE (pyCore.daeeModelLanguage attribut	e), 37	Core.daeeOptimizationVariableType at-
eCeil (pyCore.daeeUnaryFunctions attribute),		tribute), 37
eCFDM (pyCore.daeeDiscretizationMethod 36	attribute),	eIRTUnknown (pyCore.daeIndexRangeType attribute), 37
eChangeState (pyCore.daeeActionType attribution attrib	ute), 39	eLn (pyCore.daeeUnaryFunctions attribute), 38
eClosedClosed (pyCore.daeeDomainBounds		eLog (pyCore.daeeUnaryFunctions attribute), 38
36	,,	eLowerBound (pyCore.daeeDomainBounds attribute),
eClosedOpen (pyCore.daeeDomainBounds	attribute),	37
37		ELSE() (pyCore.daeModel method), 25
eConstantIndex (pyCore.daeeDomainI	ndexType	ELSE_IF() (pyCore.daeModel method), 26
attribute), 37		eLT (pyCore.daeeConditionType attribute), 39
eContinuousVariable	(py-	eLTEQ (pyCore.daeeConditionType attribute), 39
Core.daeeOptimizationVariableType	e at-	eMax (pyCore.daeeBinaryFunctions attribute), 38
tribute), 37	20	eMaxInArray (pyCore.daeeSpecialUnaryFunctions at-
eCos (pyCore.daeeUnaryFunctions attribute),		tribute), 38
39 (pyCore.daeeConditionType	attribute),	eMin (pyCore.daeeBinaryFunctions attribute), 38 eMinInArray (pyCore.daeeSpecialUnaryFunctions at-
eCustomDM (pyCore.daeeDiscretization	onMethod	tribute), 38
attribute), 36		eMinus (pyCore.daeeBinaryFunctions attribute), 38
$eCustomRange\ (pyCore.daeIndexRangeType$	attribute),	eMLNone (pyCore.daeeModelLanguage attribute), 37
37		eMTUnknown (pyCore.daeeModelType attribute), 39
eDAE (pyCore.daeeModelType attribute), 39		eMulti (pyCore.daeeBinaryFunctions attribute), 38
eDBUnknown (pyCore.daeeDomainBounds	attribute),	Enabled (pyCore.daeLog_t attribute), 31
37	(	END_IF() (pyCore.daeModel method), 26
eDifferentialValuesProvided Core.daeeInitialConditionMode	(py-	END_STN() (pyCore.daeModel method), 26
37	attribute),	EndIndex (pyCore.daeIndexRange attribute), 29 EndOfData() (pyDataReporting.daeDataReporter_t
eDistributed (pyCore.daeeDomainType attribu	ite) 36	method), 43
eDITUnknown (pyCore.daeeDomainIndex)		EndRegistration() (pyDataReport-
tribute), 37	.ypc at	ing.daeDataReporter_t method), 43
eDivide (pyCore.daeeBinaryFunctions attribu	te), 38	eNot (pyCore.daeeLogicalUnaryOperator attribute), 38
eDMUnknown (pyCore.daeeDiscretizationM		eNotEQ (pyCore.daeeConditionType attribute), 39
tribute), 36		eODE (pyCore.daeeModelType attribute), 39
eDomainIterator (pyCore.daeeDomainIndex	Type at-	eOpenClosed (pyCore.daeeDomainBounds attribute),
tribute), 37	. •	37
eDTUnknown (pyCore daeeDomainType attri	bute) 36	eOpenOpen (pvCore daeeDomainBounds attribute) 37

eOr (pyCore.daeeLogicalBinaryOperator attribute), 39 eOutletPort (pyCore.daeePortType attribute), 36	ExportObjects() (pyCore.daeModel method), 26 Expressions (pyCore.daeCondition attribute), 33
ePlus (pyCore.daeeBinaryFunctions attribute), 38	F
ePower (pyCore.daeeBinaryFunctions attribute), 38 eProduct (pyCore.daeeSpecialUnaryFunctions at-	
tribute), 38	Finalize() (pyActivity.daeOptimization method), 42 Finalize() (pyActivity.daeSimulation method), 40
ePTUnknown (pyCore.daeeParameterType attribute),	Floor() (in module pyCore), 34
ePYDAE (pyCore.daeeModelLanguage attribute), 37	G
$eQuasySteadyState\ (pyCore.daeeInitialConditionMode$	GatherInfo (pyCore.adouble attribute), 32
attribute), 37	GatherInfo (pyCore.adouble_array attribute), 32
EquationExecutionInfos (pyCore.daeEquation at-	GetBoolean() (pyCore.daeConfig method), 35
tribute), 27 Equations (pyCore.daeModel attribute), 26	GetDomainsIndexesMap() (pyCore.daeParameter
Equations (pyCore.daeNoder attribute), 28 Equations (pyCore.daePortConnection attribute), 28	method), 23
Equations (pyCore.daeState attribute), 27	GetDomainsIndexesMap() (pyCore.daeVariable
EquationType (pyCore.daeEquation attribute), 27	method), 25 GetFloat() (pyCore.daeConfig method), 35
EquationType (pyCore.daeEquationExecutionInfo at-	GetInteger() (pyCore.daeConfig method), 35
tribute), 30	GetNameRelativeToParentModel() (pyCore.daeObject
eRange (pyCore.daeeRangeType attribute), 37	method), 22
eRangeDomainIndex (pyCore.daeeRangeType at-	GetQuantity() (pyCore.daeParameter method), 23
tribute), 37	GetQuantity() (pyCore.daeVariable method), 23
eRangeOfIndexes (pyCore.daeIndexRangeType attribute), 37	GetString() (pyCore.daeConfig method), 35
eRaTUnknown (pyCore.daeeRangeType attribute), 37	GetStrippedName() (pyCore.daeObject method), 22
eReal (pyCore.daeeParameterType attribute), 36	GetStrippedNameRelativeToParentModel() (py-
eReAssignOrReInitializeVariable (py-	Core.daeObject method), 22 GetValue() (pyCore.daeParameter method), 23
Core.daeeActionType attribute), 39	GetValue() (pyCore.daeVariable method), 23
eSendEvent (pyCore.daeeActionType attribute), 39	Gradients (pyCore.daeMeasuredVariable attribute), 30
eSign (pyCore.daeeUnaryFunctions attribute), 38	Gradients (pyCore.daeObjectiveFunction attribute), 30
eSin (pyCore.daeeUnaryFunctions attribute), 38	Gradients (pyCore.daeOptimizationConstraint at-
eSqrt (pyCore.daeeUnaryFunctions attribute), 38	tribute), 30
eSteadyState (pyCore.daeeModelType attribute), 39 eSUFUnknown (pyCore.daeeSpecialUnaryFunctions	Н
attribute), 38	
eSum (pyCore.daeeSpecialUnaryFunctions attribute),	has_key() (pyCore.daeConfig method), 35
38	
ETA (pyCore.daeLog_t attribute), 31	ID (pyCore.daeObject attribute), 22
eTan (pyCore.daeeUnaryFunctions attribute), 38	IF() (pyCore.daeOoject attribute), 22 IF() (pyCore.daeModel method), 26
eUFUnknown (pyCore.daeeUnaryFunctions attribute),	IncreaseIndent() (pyCore.daeBaseLog method), 31
38 a.I. In Image Action (my Considered Action Time attailute)	IncreaseIndent() (pyCore.daeLog_t method), 31
eUnknownAction (pyCore.daeeActionType attribute),	Increment (pyCore.daeDomainIndex attribute), 28
eUnknownPort (pyCore.daeePortType attribute), 36	Indent (pyCore.daeLog_t attribute), 31
eUOUnknown (pyCore.daeeLogicalUnaryOperator at-	IndentString (pyCore.daeLog_t attribute), 31
tribute), 39	Index (pyCore.daeDomainIndex attribute), 28
eUpperBound (pyCore.daeeDomainBounds attribute),	IndexMappings (pyActivity.daeSimulation attribute), 40
37	
eUserDefinedAction (pyCore.daeeActionType at-	
4.7	InitialConditionMode (pyActivity.daeSimulation
tribute), 39	
tribute), 39 EventData (pyCore.daeEventPort attribute), 28	InitialConditionMode (pyActivity.daeSimulation attribute), 40
tribute), 39 EventData (pyCore.daeEventPort attribute), 28 EventPorts (pyCore.daeModel attribute), 26	InitialConditionMode (pyActivity.daeSimulation attribute), 40 InitialConditionMode (pyCore.daeModel attribute), 26 InitialDerivatives (pyActivity.daeSimulation attribute), 41
tribute), 39 EventData (pyCore.daeEventPort attribute), 28	InitialConditionMode (pyActivity.daeSimulation attribute), 40 InitialConditionMode (pyCore.daeModel attribute), 26 InitialDerivatives (pyActivity.daeSimulation attribute), 41 InitialGuess (pyCore.daeVariableType attribute), 22
tribute), 39 EventData (pyCore.daeEventPort attribute), 28 EventPorts (pyCore.daeModel attribute), 26 Events (pyCore.daeEventPort attribute), 28	InitialConditionMode (pyActivity.daeSimulation attribute), 40 InitialConditionMode (pyCore.daeModel attribute), 26 InitialDerivatives (pyActivity.daeSimulation attribute), 41 InitialGuess (pyCore.daeVariableType attribute), 22 Initialize() (pyActivity.daeOptimization method), 42
tribute), 39 EventData (pyCore.daeEventPort attribute), 28 EventPorts (pyCore.daeModel attribute), 26 Events (pyCore.daeEventPort attribute), 28 EventTolerance (pyCore.daeCondition attribute), 33 Execute() (pyCore.daeAction method), 29 Exp() (in module pyCore), 33	InitialConditionMode (pyActivity.daeSimulation attribute), 40 InitialConditionMode (pyCore.daeModel attribute), 26 InitialDerivatives (pyActivity.daeSimulation attribute), 41 InitialGuess (pyCore.daeVariableType attribute), 22 Initialize() (pyActivity.daeOptimization method), 42 Initialize() (pyActivity.daeSimulation method), 41
tribute), 39 EventData (pyCore.daeEventPort attribute), 28 EventPorts (pyCore.daeModel attribute), 26 Events (pyCore.daeEventPort attribute), 28 EventTolerance (pyCore.daeCondition attribute), 33 Execute() (pyCore.daeAction method), 29	InitialConditionMode (pyActivity.daeSimulation attribute), 40 InitialConditionMode (pyCore.daeModel attribute), 26 InitialDerivatives (pyActivity.daeSimulation attribute), 41 InitialGuess (pyCore.daeVariableType attribute), 22 Initialize() (pyActivity.daeOptimization method), 42

Integrate() (pyActivity.daeSimulation method), 41 IntegrateForTimeInterval() (pyActivity.daeSimulation method), 41	NestedSTNs (pyCore.daeState attribute), 27 NextReportingTime (pyActivity.daeSimulation attribute), 41
IntegrateUntilTime() (pyActivity.daeSimulation method), 41	Node (pyCore.adouble attribute), 32 Node (pyCore.adouble_array attribute), 32
IsConnected() (pyDataReporting.daeDataReporter_t	Node (pyCore.daeEquationExecutionInfo attribute), 30
method), 43 IsModelDynamic (pyCore.daeModel attribute), 26 items() (pyCore.adouble_array method), 33	NoPoints (pyCore.daeArrayRange attribute), 29 NoPoints (pyCore.daeIndexRange attribute), 29 npyIDs (pyCore.daeVariable attribute), 25
J	npyPoints (pyCore.daeDomain attribute), 23 npyValues (pyCore.daeParameter attribute), 23
JoinMessages() (pyCore.daeLog_t method), 31	npyValues (pyCore.daeVariable attribute), 25 NumberOfEquations (pyActivity.daeSimulation at-
L	tribute), 41
Library (pyCore.daeObject attribute), 22 LoadInitializationValues() (pyActivity.daeSimulation	NumberOfIntervals (pyCore.daeDomain attribute), 22 NumberOfObjectiveFunctions (pyActiv- ity.daeSimulation attribute), 41
method), 41 Log (pyActivity.daeSimulation attribute), 41 Log() (in module pyCore), 33	NumberOfPoints (pyCore.daeDomain attribute), 22 NumberOfPoints (pyCore.daeParameter attribute), 23
Log10() (in module pyCore), 33	NumberOfPoints (pyCore.daeVariable attribute), 25
LowerBound (pyCore.daeDomain attribute), 22 LowerBound (pyCore.daeOptimizationVariable at-	0
tribute), 29	ObjectiveFunction (pyActivity.daeSimulation at-
LowerBound (pyCore.daeVariableType attribute), 22	tribute), 41
M	ObjectiveFunctions (pyActivity.daeSimulation attribute), 41
m (pyActivity.daeSimulation attribute), 42	ON_CONDITION() (pyCore.daeModel method), 26
Max() (in module pyCore), 34	ON_EVENT() (pyCore.daeModel method), 26
MeasuredVariables (pyActivity.daeSimulation at-	OnEventActions (pyCore.daeModel attribute), 26
tribute), 41	Optimization Variables (pyActivity.daeSimulation at-
Message() (pyCore.daeBaseLog method), 31	tribute), 41 OverallIndex (pyCore.daeVariable attribute), 25
Message() (pyCore.daeFileLog method), 31	OverallIndex (pyCore.daeVariableWrapper attribute),
Message() (pyCore.daeLog_t method), 31	35
Message() (pyCore.daeStdOutLog method), 31 Message() (pyCore.daeTCPIPLog method), 32	
MessageReceived() (pyCore.daeTCPIPLogServer	P
method), 32	Parameters (pyCore.daeModel attribute), 26
Min() (in module pyCore), 34	Parameters (pyCore.daePort attribute), 27
Model (pyActivity.daeSimulation attribute), 41	Pause() (pyActivity.daeSimulation method), 41
model (pyActivity.daeSimulation attribute), 42	PercentageDone (pyCore.daeLog_t attribute), 31
Model (pyCore.daeObject attribute), 22	Points (pyCore.daeDomain attribute), 22
ModelParameters (pyActivity.daeSimulation attribute),	PortArrays (pyCore.daeModel attribute), 26
41	PortConnections (pyCore.daeModel attribute), 26
ModelType (pyCore.daeModel attribute), 26	PortFrom (pyCore.daePortConnection attribute), 28
N I	Ports (pyCore.daeModel attribute), 26
N	PortTo (pyCore.daePortConnection attribute), 28
Name (pyCore.daeMeasuredVariable attribute), 30	Pow() (in module pyCore), 34
Name (pyCore.daeObject attribute), 22	PrintProgress (pyCore.daeLog_t attribute), 31
Name (pyCore.daeObjectiveFunction attribute), 30	Product() (in module pyCore), 30 Progress (pyCore.daeLog_t attribute), 31
Name (pyCore.daeOptimizationConstraint attribute),	pyActivity (module), 40
30 Name (pyCore.daeOptimizationVariable attribute), 29	pyCore (module), 21
Name (pyCore.daeOptimization variable attribute), 29 Name (pyCore.daeScalarExternalFunction attribute),	pyDataReporting (module), 43
28	pyIDAS (module), 43
Name (pyCore.daeVariableType attribute), 22	pyUnits (module), 44
Name (pyCore.daeVariableWrapper attribute), 35	-
Name (pyCore.daeVectorExternalFunction attribute),	Q
28	quantity (class in pyUnits), 44

R	SetFloat() (pyCore.daeConfig method), 35
Range (pyCore.daeArrayRange attribute), 29 ReAssignValue() (pyCore.daeVariable method), 24	SetInitialCondition() (pyCore.daeVariable method), 24 SetInitialConditions() (pyCore.daeVariable method), 24
ReAssignValues() (pyCore.daeVariable method), 24	SetInitialGuess() (pyCore.daeVariable method), 24
ReceiveEvent() (pyCore.daeEventPort method), 28	SetInitialGuesses() (pyCore.daeVariable method), 24
RecordEvents (pyCore.daeEventPort attribute), 28	SetInputVariable() (pyActivity.daeSimulation method), 41
RegisterData() (pyActivity.daeSimulation method), 41	SetInteger() (pyCore.daeConfig method), 35
RegisterDomain() (pyDataReport-	SetInteger() (pyCore.daeConing inethod), 35 SetIntegerOptimizationVariable() (pyActiv-
ing.daeDataReporter_t method), 43	ity.daeSimulation method), 42
RegisterVariable() (pyDataReport-	SetLASolver() (pyIDAS.daeIDAS method), 44
ing.daeDataReporter_t method), 43 Reinitialize() (pyActivity.daeSimulation method), 41	SetMeasuredVariable() (pyActivity.daeSimulation
RelativeTolerance (pyActivity.daeSimulation attribute),	method), 42
41	SetModelParameter() (pyActivity.daeSimulation
Reload() (pyCore.daeConfig method), 35	method), 42
ReportData() (pyActivity.daeSimulation method), 41	SetProgress() (pyCore.daeBaseLog method), 31
ReportingInterval (pyActivity.daeSimulation attribute),	SetReportingOn() (pyCore.daeModel method), 27
41	SetReportingOn() (pyCore.daePort method), 27
ReportingOn (pyCore.daeParameter attribute), 23	SetString() (pyCore.daeConfig method), 35
ReportingOn (pyCore.daeVariable attribute), 25	SetupNode (pyCore.daeAction attribute), 29
ReportingTimes (pyActivity.daeSimulation attribute),	SetupNode (pyCore.daeCondition attribute), 33 SetUpOptimization() (pyActivity.daeSimulation
41 P. P. O. (14 division 1981) 41	method), 42
ReRun() (pyActivity.daeSimulation method), 41	SetUpParameterEstimation() (pyActiv-
Reset() (pyActivity.daeSimulation method), 41 ReSetInitialCondition() (pyCore.daeVariable method),	ity.daeSimulation method), 42
24	SetUpParametersAndDomains() (pyActiv-
ReSetInitialConditions() (pyCore.daeVariable method),	ity.daeSimulation method), 42
24	SetUpSensitivityAnalysis() (pyActivity.daeSimulation
Residual (pyCore.daeEquation attribute), 27	method), 42
Residual (pyCore.daeMeasuredVariable attribute), 30	SetUpVariables() (pyActivity.daeSimulation method),
Residual (pyCore.daeObjectiveFunction attribute), 30	42
Residual (pyCore.daeOptimizationConstraint attribute),	SetValue() (pyCore.daeParameter method), 23
30	SetValue() (pyCore.daeVariable method), 24 SetValues() (pyCore.daeParameter method), 23
Resize() (pyCore.adouble_array method), 32	Set Values() (pyCore.daeVariable method), 24
Resume() (pyActivity.daeSimulation method), 41	SimulationMode (pyActivity.daeSimulation attribute),
Run() (pyActivity.daeOptimization method), 42 Run() (pyActivity.daeSimulation method), 41	42
RuntimeNode (pyCore.daeAction attribute), 29	Sin() (in module pyCore), 33
RuntimeNode (pyCore.daeCondition attribute), 33	Sinh() (in module pyCore), 34
runtimer (ode (p) core due condition duribute), 33	SolveInitial() (pyActivity.daeSimulation method), 42
S	Sqrt() (in module pyCore), 33
SaveMatrixAsXPM() (pyIDAS.daeIDAS method), 43	StartIndex (pyCore.daeIndexRange attribute), 29
SaveModelReport() (pyCore.daeModel method), 27	StartingPoint (pyCore.daeOptimizationVariable at-
SaveRuntimeModelReport() (pyCore.daeModel	tribute), 29 StartNewResultSet() (pyDataReport-
method), 27	ing.daeDataReporter_t method), 43
scaleTo() (pyUnits.quantity method), 44	StartRegistration() (pyDataReport-
Scaling (pyCore.daeEquation attribute), 27	ing.daeDataReporter_t method), 43
SendEvent() (pyCore.daeEventPort method), 28	STATE() (pyCore.daeModel method), 26
SendEventPort (pyCore.daeAction attribute), 29	States (pyCore.daeSTN attribute), 27
SendVariable() (pyDataReporting.daeDataReporter_t method), 43	StateTo (pyCore.daeAction attribute), 29
SetAbsoluteTolerances() (pyCore.daeVariable method),	StateTransitions (pyCore.daeState attribute), 27
24	Step (pyCore.daeIndexRange attribute), 29
SetBinaryOptimizationVariable() (pyActiv-	STN (pyCore.daeAction attribute), 29
ity.daeSimulation method), 41	STN() (pyCore.daeModel method), 26
SetBoolean() (pyCore.daeConfig method), 35	STNs (pyCore.daeModel attribute), 26 StoreInitializationValues() (pyActivity.daeSimulation
SetContinuousOptimizationVariable() (pyActiv-	method), 42
ity.daeSimulation method), 41	memod), 72

```
Sum() (in module pyCore), 30
SWITCH_TO() (pyCore.daeModel method), 26
Tan() (in module pyCore), 34
Tanh() (in module pyCore), 34
Time() (in module pyCore), 30
TimeHorizon (pyActivity.daeSimulation attribute), 42
TotalNumberOfVariables (pyActivity.daeSimulation at-
         tribute), 42
Type (pyCore.daeAction attribute), 29
Type (pyCore.daeArrayRange attribute), 29
Type (pyCore.daeDomain attribute), 22
Type (pyCore.daeDomainIndex attribute), 28
Type (pyCore.daeEventPort attribute), 28
Type (pyCore.daeIndexRange attribute), 29
Type (pyCore.daeOptimizationConstraint attribute), 30
Type (pyCore.daeOptimizationVariable attribute), 29
Type (pyCore.daePort attribute), 27
U
unit (class in pyUnits), 44
unitDictionary (pyUnits.unit attribute), 44
Units (pyCore.daeDomain attribute), 22
Units (pyCore.daeParameter attribute), 23
Units (pyCore.daeVariableType attribute), 22
units (pyUnits.quantity attribute), 44
UpperBound (pyCore.daeDomain attribute), 23
UpperBound (pyCore.daeOptimizationVariable
         tribute), 29
UpperBound (pyCore.daeVariableType attribute), 22
Value (pyCore.adouble attribute), 32
Value (pyCore.daeMeasuredVariable attribute), 30
Value (pyCore.daeObjectiveFunction attribute), 30
Value (pyCore.daeOptimizationConstraint attribute), 30
Value (pyCore.daeOptimizationVariable attribute), 29
Value (pyCore.daeVariableWrapper attribute), 35
value (pyUnits.quantity attribute), 44
valueInSIUnits (pyUnits.quantity attribute), 44
Variable (pyCore.daeVariableWrapper attribute), 35
VariableIndexes (pyCore.daeEquationExecutionInfo at-
         tribute), 30
Variables (pyCore.daeModel attribute), 27
Variables (pyCore.daePort attribute), 27
VariableType (pyCore.daeVariable attribute), 25
VariableType (pyCore.daeVariableWrapper attribute),
VariableTypes (pyActivity.daeSimulation attribute), 42
VariableWrapper (pyCore.daeAction attribute), 29
Version (pyCore.daeObject attribute), 22
```