DAE Tools Project

pyDAE

Version 1.0.0

Introduction, Getting Started, Tutorial, User Guide

[d][a][e]
Tools Project

Dragan Nikolić, dnikolic@daetools.com

August 5, 2010

Table of Contents

http://www.bestcode.com/html/syntax_highlighter.php

http://www.andre-simon.de/doku/highlight/en/highlight_demo.html IDE MSVS 2008

1. Introduction

1.1. What is pyDAE?

DAE Tools is a collection of software tools for modelling, simulation and optimization of real-world processes. It is a free software published under the GNU General Public Licence (think of it as in free speech not free beer). Like no other process modelling software it respects your freedom: to run software as you wish, to modify it as and if you wish, to distribute it if and as you wish and to publish modified version when and if you wish.

DAE Tools are initially developed to model and simulate processes in chemical process industry. However, no restrictions are imposed on the kind of phenomena modelled. Thus, DAE Tools can help you develop high-accuracy models of (in general) many different kind of processes/phenomena, simulate them, visualize and analyse results. Its features should be sufficient to enable mathematical description of chemical, physical or socio/economic phenomena.

1.2. History

"*Necessity, who is the mother of invention*".

(Plato, The Republic, Greek philosopher, ~380 BC)

"*Every good work of software starts by scratching a developer's personal itch*".

(Eric S. Raymond, hacker, The Cathedral and the Bazaar, 1997)

This sentence couldn't be more true (although I do not agree with the Open Source views, I am more concerned about freedom thus my support to the Free Software Foundation, but let us leave it beside at the moment). The early ideas of starting a project like this go back into 2007. At that time I have been working on my PhD thesis using one of commercially available process modelling software. It was everything nice and well until I discovered some annoying bugs and lack of certain highly appreciated features. The developers of that proprietary program (as it is a case with all proprietary computer programs) had their own agenda fixing only they wanted to fix and introducing new features that only they anticipated. I have been developing a model of a pressure swing adsorption process (gas separation). Its main characteristic is that it never reaches a steady state – only a cyclic steady state (after a high number of cycles). Although I was able to improve the code and introduce certain features which will help (not only) me in modelling such types of processes I was helpless. The source code was not available and nobody will ever consider giving it to me (to create patches or to fix certain bugs/develop new features – not even if I swear on a holy (c++) bible ;-)). The turning point was somewhere in 2007 with the release of the new versions. Not only weren't there expected bug fixes but the list of new features included something like the following: tool-tips over reserved words (why would one ever need something like that?), a keyboard short-cut Ctrl-P to print the document (uff, what a relief) and finally my favourite - a recognition of Windows XP service pack 2 (wtf!??). Enough is enough! Now, having the service pack successfully recognized the contours of my own (free!) process modelling software began to form. It took me a while until I made a definite plan and initial features. I pondered different ideas. One was to follow Cape-Open standards. But hey, they have been made according to one and only available commercial program. Guess which...? Moreover, even they dont follow it but use its extensions instead. Should I make a clone of it or brainstorm something new. Well, the answer is not difficult... and the new free program has been born. Obviously it took me some time to start actual coding (remember I have been doing it in my spare time) and I had to abandon few initial versions ("*plan to throw one away; you will, anyhow*"; damn you Eric Raymond, interfering with my business again :-)) so the new project was officially born early next year - 2008.

2. Getting started

2.1. Installation

2.1.1. Requirements

- Python (2.6+): *www.python.org*
- Boost libraries (version 1.35+; system, python, thread, asio): *www.boost.org*
- Numpy: *http://numpy.scipy.org*
- Matplotlib: *http://matplotlib.sourceforge.net*
- pyQT4: *http://www.riverbankcomputing.co.uk/software/pyqt*

For more information on how to install packages please refer to the documentation for the specific library.

2.1.2. Getting the packages

First download the appropriate installer for your operating system (GNU/Linux, Windows) and architecture (x86, x86_64, arm):

- .deb for Debian GNU/Linux
- .rpm for rpm based distributions
- .exe for Windows

2.1.3. Installing

2.1.3.1. Debian GNU/Linux

First install necessary dependencies. To do so in Debian GNU/Linux use the following command:

```
$ sudo apt-get install libboost-all-dev python-qt4 python-numpy python-matplotlib
```

Install *DAE Tools* by opening it by *Gdebi* package installer or by typing the following shell command:

```
$ cd Directory_where_you_downloaded_the_package
$ sudo dpkg -i PACKAGE_NAME.deb
```

2.1.3.2. Other GNU/Linux distributions

For other distributions follow their documentation.

2.1.3.3. Windows

So far it is tested on Windows XP only. First install necessary dependencies. To do so follow the instructions on corresponding web sites (you can also try Python(x,y) at http://www.pythonxy.com/). Then install *DAE Tools* by double clicking it and follow the instructions.

2.1.4. Library structure

pyDAE will be installed in the site-packages folder under python (/usr/lib/python26, C:/Python2.6 for instance). The structure of the folders and files is:

- pyDAE
  - docs
  - examples
  - include
  - lib
  - *pyAmdACML.so*
  - *pyDAE.so*
  - *pyIntelMKL.so*
  - *pyIntelPardiso.so*
  - *pyLapack.so*
  - *pyTrilinosAmesos.so*
  - Various .py files

2.2. Basic concepts

Explain how do I create models and run simulations.

2.3. Your first simulation – “What's the time?”

whats_the_time.py

```
from pyDAE import *
from time import localtime, strftime

typeNone = daeVariableType("None", "-", 0, 1E10,   0, 1e-5)

class modTutorial(daeModel):
    def __init__(self, Name):
        daeModel.__init__(self, Name)

        self.time = daeVariable("Time", typeNone, self)

    def DeclareEquations(self):
        eq = self.CreateEquation("time")
        eq.Residual = self.time.dt() - 1.0

        super(modTutorial, self).DeclareEquations()

class simTutorial(daeDynamicSimulation):
    def __init__(self):
        daeDynamicSimulation.__init__(self)
        self.m = modTutorial("Tutorial")
        self.SetModel(self.m)

    def SetUpVariables(self):
        self.m.time.SetInitialCondition(0, eAlgebraicIC)

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

# Enable reporting of the results
simulation.m.SetReportingOn(True)

# Set the time horizon and the reporting interval
simulation.SetReportingInterval(1)
simulation.SetTimeHorizon(100)

# Connect data reporter
simName = strftime("What's the time [%d.%m.%Y %H:%M:%S]", localtime())
datareporter.Connect("", simName)

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

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

# Run
simulation.Run()
```

Figure 2.3.1 What's the time output

3. Tutorial

3.1. Tutorial 1 – Heat Conduction

Figure 3.1.1 Tutorial 1 output

3.2. Tutorial 2 – Heat Conduction

4. User Guide

DAE Tools consist of a set of libraries:

- Core library
- Solvers
- Activity library
- Datareporting library

What is needed to successfully simulate/optimize some process. Two the most important parts are Model and Activity definition. First one should build a model that is a mathematical description of the process. Another important part is the Activity definition (an operating procedure) that is how the model should be simulated. What is needed are values of the time horizon, the model parameters, initial conditions etc.

4.1. Core module

4.1.1. Models

Model is set of parameters, distribution domains, variables, ports, equations, state transition networks (STN) and child models.

Models have the following properties:

- *Name*: string (read-only; inherited from *daeObject*)
  Defines a name of an object (“Temperature” for instance)
- *CanonicalName*: string (read-only; inherited from *daeObject*)
  It is a method of defining a location of an object (“HeatExchanger.Temperature” for instance means that the object Temperature belongs to the parent object HeatExchanger). Object names are divided by dot symbols (“.”)
- *Domains*: *daeDomain* list
- *Parameters*: *daeParameter* list
- *Variables*: *daeVariable* list
- *Equations*: *daeEquation* list
- *Ports*: *daePort* list
- *ChildModels*: *daeModel* list
- *PortArrays*: *daePortArray* list
- *ChildModelArrays*: *daeModelArray* list
- InitialConditionMode: daeeInitialConditionMode

The most important functions are:

- *ConnectPorts*
- *SetReportingOn*
- *sum*, *product*, *integral*, *min*, *max*
- *CreateEquation*
- *If*, *Else_if*, *Else*, *End_if*
- *Stn*, *State*, *Switch_to*, *End_stn*

Every user model has to implement two functions: *__init__* and *DeclareEquations*. *__init__* is the constructor and all parameters, distribution domains, variables, ports, and child models must be declared here. *DeclareEquations* function is used to declare equations and state transition networks.

```
class myModel(daeModel)
    def __init__(self, Name, Model):
        daeModel.__init__(self, Name, Model)
        ...

    def DeclareEquations(self)
        ...
        super(myModel, self).DeclareEquations()
```

Details of how to declare and use parameters, distribution domains, variables, ports, equations, state transition networks (STN) and child models are given in the following sections.

4.1.2. Equations

To create equation residuals and to compute Jacobian matrix pyDAE combines the operator overloading concept introduced in ADOL-C, with the concept of representing equations as binary nodes tree. To do so, ADOL-C library has been extended to create these trees, which can be then used to calculate residuals and derivatives or to export equations into MathML or Latex format. To this end *adouble* class from the ADOL-C library has been extended to automatically create these nodes. Function call operators (*operator()*) in daeParameter/daeVariable classes and functions like *dt*, *d*, and *d2* in daeVariable class all return the extended *adouble* objects. By overloading basic mathematical operators and functions to accept adouble objects as their arguments it is possible to build up binary trees and store them in a top level *adouble* object – daeEquation residual.

- *Name*: string (read-only; inherited from daeObject)
- *CanonicalName*: string (read-only; inherited from daeObject)
- *Domains*: daeDomain list (read-only)
- *Residual*: adouble

4.1.2.1. Declaring equations

To declare an ordinary equation:

```
eq = model.CreateEquation("MyEquation")
```

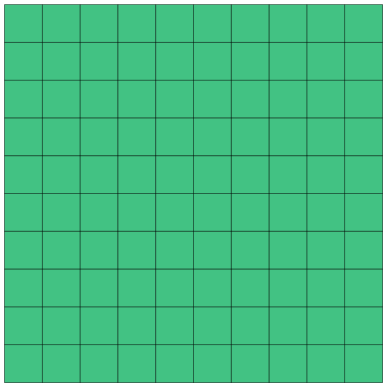
To declare a distributed equation:

```
eq = model.CreateEquation("MyEquation")
md = eq.DistributeOnDomain(myDomain, eClosedClosed)
```

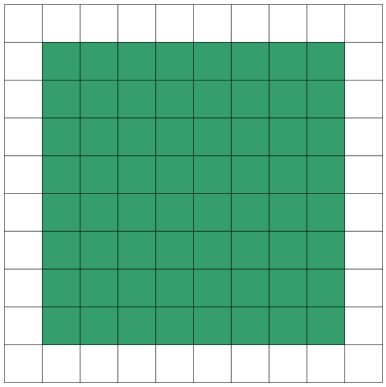
Ovde sad moze onaj grafik sa razni Bounds-ima...

4.1.2.2. Defining equations (building of equation residuals)

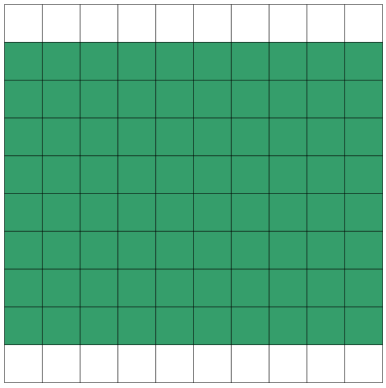
Table 1. Distributed equation bound examples



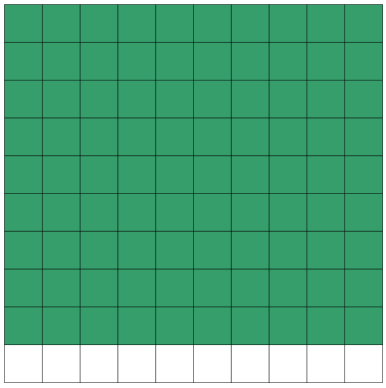
x: eClosedClosed; y: eClosedClosed



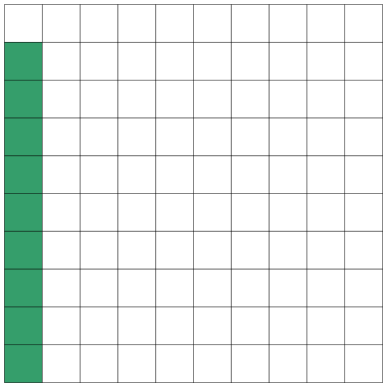
x: eOpenOpen; y: eOpenOpen



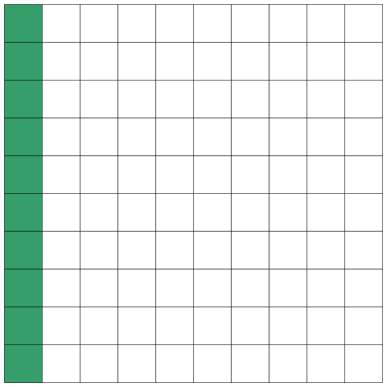
x: eClosedClosed; y: eOpenOpen



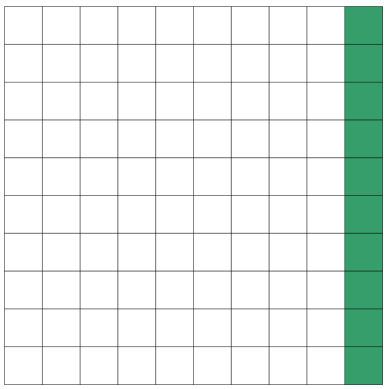
x: eClosedClosed; y: eOpenClosed



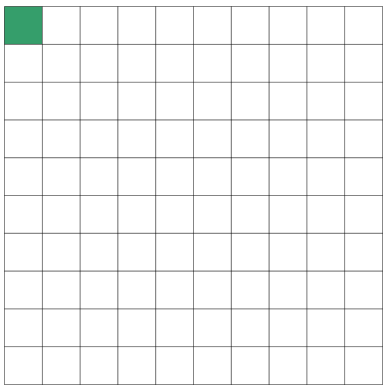
x: eLowerBound; y: eClosedOpen



x: eLowerBound; y: eClosedClosed



x: eUpperBound; y: eClosedClosed



x: eLowerBound; y: eUpperBound

4.1.3. Distribution Domains

Derived from *daeObject*. A distribution domain is a general term used to define an array of different objects. Two types of domains exist: arrays and distributed domains. Arrays are a synonym for a simple array (vector) of objects (parameters, variables, ports, equations, models). Distributed domains are most frequently used to model a spatial distribution of variables, but can be equally used to spatially distribute just any other object (parameters, ports, equations or even models).

Domains have the following properties:

- *Name*: string (read-only; inherited from *daeObject*)
- *CanonicalName*: string (read-only; inherited from *daeObject*)
- *Type*: *daeeDomainType* (read-only)
  It can be array or distributed
- *NumberOfIntervals*: unsigned integer (read-only)
- *NumberOfPoints*: unsigned integer (read-only)
- *LowerBound*: float (read-only)
- *UpperBound*: float (read-only)

Distributed domains also have:

- *DiscretizationMethod*: *daeeDiscretizationMethod* (read-only)
  It can be backward difference, forward difference or central difference method
- *DiscretizationOrder*: unsigned integer (read-only)
  It can be 2, 4 and 6

There is a difference between number of points in domain and number of intervals. Number of intervals is a number of points (if it is array) or finite difference elements (if it is distributed domain). Number of points is actual number of points in the domain. If it is array then they are equal. If it is distributed, and the scheme is one of finite differences for instance, it is equal to the number of intervals + 1.

The most important functions are:

- *operator[]* for getting a value of the point within domain for a given index (used only to construct equation residuals)
- Overloaded *operator[]* for creating daeIndexRange object (used only to construct equation residuals as an argument of functions *array*, *dt_array*, *d_array*, *d2_array*)
- *GetNumPyArray* for getting the point values as a numpy one-dimensional array

The process of creating domains is two-fold: first you declare a domain in the model and then you define it (by assigning its properties) in the simulation.

4.1.3.1. Declaring a domain

To declare a domain:

```
myDomain = daeDomain("myDomain", Parent)
```

4.1.3.2. Defining a domain

To define a distributed domain:

```
# Center finite diff, 2nd order, 10 elements, Bounds: 0.0 to 1.0
myDomain.CreateDistributed(eCFDM, 2, 10, 0.0,  1.0)
```

To define an array:

```
# Array of 10 elements
myDomain.CreateArray(10)
```

4.1.3.3. Using domains

NOTE: It is important to understand that all functions in this chapter are used ONLY to construct equation residuals and NOT no to access the real (raw) data.

I) To get a value of the point within the domain at the given index we can use *operator []*. For instance if we want variable *myVar* to be equal to the sixth point in the domain *myDomain*, we can write:

```
# Notation:
#  - eq is a daeEquation object
#  - myDomain is daeDomain object
#  - myVar is an daeVariable object

eq.Residual = myVar() - myDomain[5]
```

This code translates into:

$$myVar = myDomain[5]$$

II) *daeDomain operator()* returns the *daeIndexRange* object which is used as an argument of functions *array*, *dt_array*, *d_array and d2_array* in *daeParameter* and *daeVariable* classes to obtain an array of parameter/variable values, or an array of variable time (or partial) derivatives.

More details on parameter/variable arrays will be given in the next sections.

4.1.4. Parameters

Parameters are time invariant quantities that will not change during simulation. Usually a good choice what should be a parameter is a physical constant, number of discretization points in a domain etc. Parameters have the following properties:

- *Name*: string (read-only; inherited from *daeObject*)
- *CanonicalName*: string (read-only; inherited from *daeObject*)
- *Type*: *daeeParameterType* (read-only; real, integer, boolean)
- *Domains*: *daeDomain* list

The most important functions are:

- Overloaded *operator()* for getting the parameter value (used only to construct equation residuals)
- Overloaded function a*rray* for getting an array of values (used only to construct equation residuals as an argument of functions like s*um*, p*roduct* etc)
- Overloaded functions *SetValue* and *GetValue* for access to the parameter's raw data
- *GetNumPyArray* for getting the values as a numpy multidimensional array

The process of creating parameters is two-fold: first you declare a parameter in the model and then you define it (by assigning its value) in the simulation.

4.1.4.1. Declaring a parameter

Parameters are declared in a model constructor (*__init__* function). To declare an ordinary parameter:

```
myParam = daeParameter("myParam", eReal, Parent)
```

Parameters can be distributed on domains. To declare a distributed parameter:

```
myParam = daeParameter("myParam", eReal, Parent)
myParam.DistributeOnDomain(myDomain)
```

Here, argument *Parent* can be either *daeModel* or *daePort*.

4.1.4.2. Defining a parameter

Parameters are defined in a simulation class (*SetUpParametersAndDomains* function). To set a value of an ordinary parameter:

```
myParam.SetValue(1.0)
```

To set a value of distributed parameters (one-dimensional for example):

```
for i in range(0, myDomain.NumberOfPoints)
    myParam.SetValue(i, 1.0)
```

4.1.4.3. Using parameters

NOTE: It is important to understand that all functions in this chapter are used ONLY to construct equation residuals and NOT no to access the real (raw) data.

I) To get a value of the ordinary parameter we can use *operator ()*. For instance, if we want variable *myVar* to be equal to the sum of the value of the parameter *myParam* and 15, we can write:

```
# Notation:
#  - eq is a daeEquation object
#  - myParam is an ordinary daeParameter object (not distributed)
#  - myVar is an ordinary daeVariable (not distributed)

eq.Residual = myVar() - myParam() - 15
```

This code translates into:

$$myVar = myParam() + 15$$

II) To get a value of the distributed parameter we can again use *operator ()*. For instance, if we want distributed variable *myVar* to be equal to the sum of the value of the parameter myParam and 15 at each point of the domain *myDomain*, we need an equation for each point in the *myDomain*:

```
# Notation:
#  - myDomain is daeDomain object
#  - n is the number of points in the myDomain
#  - eq is a daeEquation object distributed on the myDomain
#  - d is daeDEDI object (used to iterate through the domain points)
#  - myParam is daeParameter object distributed on the myDomain
#  - myVar is daeVariable object distributed on the myDomain

d = eq.DistributeOnDomain(myDomain, eClosedClosed)
eq.Residual = myVar(d) - myParam(d) - 15
```

This code translates into:

$$myVar(d) = myParam(d) + 15;\ \forall d \in [0, n]$$

which is equivalent to writing (in pseudo-code):

```
for d = 0 to n:
    myVar(d) = myParam(d) + 15
```

and which internally transforms into *n* separate equations:

$$myVar(0) = myParam(0) + 15$$
$$myVar(1) = myParam(1) + 15$$
$$...$$
$$myVar(n) = myParam(n) + 15$$

Obviously, a parameter can be distributed on more than one domain. In that case we can use identical functions which accept two arguments:

```
# Notation:
#  - myDomain1, myDomain2 are daeDomain objects
#  - n is the number of points in the myDomain1
#  - m is the number of points in the myDomain2
#  - eq is a daeEquation object distributed on the domains myDomain1 and myDomain2
#  - d is daeDEDI object (used to iterate through the domain points)
#  - myParam is daeParameter object distributed on the myDomain1 and myDomain2
#  - myVar is daeVariable object distributed on the myDomaina and myDomain2

d1 = eq.DistributeOnDomain(myDomain1, eClosedClosed)
d2 = eq.DistributeOnDomain(myDomain2, eClosedClosed)
eq.Residual = myVar(d1,d2) – myParam(d1,d2) – 15
```

This code translates into:

$$myVar(d1, d2) = myParam(d1, d2) + 15;\ \forall d1 \in [0, n], \forall d2 \in [0, m]$$

III) To get an array of parameter values we can use the function *array* which returns the *adouble_array* object. Arrays of values can only be used in conjunction with mathematical functions that operate on *adouble_array* objects: *sum*, *product*, *sin*, *cos*, *min*, *max*, *log*, *log10* etc. For instance, if we want variable *myVar* to be equal to the sum of values of the parameter *myParam* for all points in the domain *myDomain*, we can use the function *sum* (defined in *daeModel* class) which accepts results of the *array* function (defined in *daeParameter* class). Arguments for the *array* function are *daeIndexRange* objects obtained by the call to *daeDomain operator ()*. Thus, we can write:

```
# Notation:
#  - myDomain is daeDomain object
#  - n is the number of points in the domain myDomain
#  - eq is daeEquation object
#  - myVar is daeVariable object
#  - myParam is daeParameter object distributed on the myDomain

eq.Residual = myVar() - sum( myParam.array( myDomain() ) )
```

This code translates into:

$$myVar = \sum_{i=0}^{n} myParam(i)$$

which transforms into:

$$myVar = myParam(0) + myParam(1) + ... + myParam(n)$$

The above example could be also written in an extended form:

```
# points_range is daeDomainRange object
points_range = daeDomainRange(myDomain)

# arr is adouble_array object
arr = myVar2.array(points_range)

# Finally:
```

```
eq.Residual = myVar() - sum(arr)
```

However, the previous formulation is more concise and shorter.

On the other hand, if we want variable *myVar* to be equal to the sum of values of the parameter *myParam* only for certain points in the *myDomain*, there are two ways to do it:

```
# Notation:
#  - myDomain is daeDomain object
#  - n is the number of points in the domain myDomain
#  - eq is a daeEquation object
#  - myVar is an ordinary daeVariable object
#  - myParam is a daeParameter object distributed on the myDomain

# 1) For a given set of points; the points must be in the range [0,n-1]
eq.Residual = myVar() - sum( myParam.array( myDomain( [0, 5, 12] ) ) )

# 2) For a given slice of points in the domain;
#    slices are defined by 3 arguments: start_index, end_index, step
#    in this example: start_index = 1
#                     end_index = 10
#                     step = 2
eq.Residual = myVar() - sum( myParam.array( myDomain(1, 10, 2) ) )
```

The code sample 1) translates into:

$$myVar = myParam(0) + myParam(5) + myParam(12)$$

The code sample 2) translates into:

$$myVar = myParam(1) + myParam(3) + myParam(5) + myParam(7) + myParam(9)$$

NOTE: One may argue that the function *array* calls can be somewhat simpler and directly accept python lists or slices as its arguments. For instance it would be possible to write:

```
eq.Residual = myVar() - sum( myParam.array( [0, 1, 3] ) )
```

or

```
eq.Residual = myVar() - sum( myParam.array( slice(1,10,2) ) )
```

However, that would be more error prone since it does not check whether a valid domain is used and whether specified indexes lay within the domain bounds (which should be done by the user).

4.1.5. Variable Types

Variable types are used to describe variables. The most important properties are:

- *Name*: string
- *Units*: string
- *LowerBound*: float
- *UpperBound*: float
- *InitialGuess*: float
- *AbsoluteTolerance*: float

Declaration of variable types is usually done outside of model definitions (as global variables).

4.1.5.1. Declaring a variable type

To declare a variable type:

```
# Temperature, units: Kelvin, limits: 100 – 1000K, Def.value: 273K, Abs.Tol: 1E-5
typeTemperature = daeVariableType("Temperature", "K", 100, 1000, 273, 1E-5)
```

4.1.6. Variables

Variables are time variant quantities (state variables). However, it is possible to fix the value of certain variables, that is to fix the degrees of freedom of a model.

The most important properties are:

- *Name*: string (read-only; inherited from daeObject)
- *CanonicalName*: string (read-only; inherited from daeObject)
- *Type*: daeVariableType object
- *Domains*: daeDomain list
- *ReportingOn*: boolean

The most important functions are:

- Overloaded *operator()* for getting the variable value/time derivative/partial derivative (used only to construct equation residuals)
- Overloaded functions a*rray*, *dt_array*, *d_array*, and *d2_array* for getting an array of values/time derivatives/partial derivatives (used only to construct equation residuals as an argument of functions like *Sum*, *Product* etc)
- Overloaded functions *AssignValue* to fix degrees of freedom of the model
- Overloaded functions *ReAssignValue* to change a value of a fixed variable
- Overloaded functions *SetValue* and *GetValue* for access to the variable's raw data
- Overloaded function *SetInitialGuess* for setting an initial guess of the variable
- Overloaded function *SetInitialCondition* for setting an initial condition of the variable
- Overloaded function *ReSetInitialCondition* for re-setting an initial condition of the variable
- Overloaded function *SetAbsoluteTolerances* for setting an absolute tolerance of the variable
- *GetNumPyArray* for getting the values as a numpy multidimensional array

The process of creating variables is two-fold: first you declare a variable in the model and then you define it (by assigning its value) in the simulation.

4.1.6.1. Declaring a variable

Variables are declared in a model constructor (*__init__* function). To declare an ordinary variable:

```
myVar = daeVariable("myVar", variableType, Parent)
```

Variables can be distributed on domains. To declare a distributed variable:

```
myVar = daeVariable("myVar", variableType, Parent)
myVar.DistributeOnDomain(myDomain)
```

Here, argument *Parent* can be either daeModel or daePort.

4.1.6.2. Setting degrees of freedom of the model (fixing the variable value)

Degrees of freedom are fixed in a simulation class (*SetUpVariables* function). To fix the value of ordinary variables:

```
myVar.FixValue(1.0)
```

To fix the value of distributed variables (one-dimensional for example):

```
for i in range(0, myDomain.NumberOfPoints)
    myVar.FixValue(i, 1.0)
```

Changing a value of the fixed variable

4.1.6.3. Accessing a variable raw data

GetValue/*SetValue* functions access the variable raw data and should be used directly with a great care!!!

ONLY USE THIS FUNCTION IF YOU EXACTLY KNOW WHAT ARE YOU DOING AND THE POSSIBLE IMPLICATIONS!!!!

To set the value of ordinary variables:

```
myVar.SetValue(1.0)
```

To set the value of distributed variables (one-dimensional for example):

```
for i in range(0, myDomain.NumberOfPoints)
    myVar.SetValue(i, 1.0)
```

4.1.6.4. Setting an initial guess

Initial guesses are set in a simulation class (*SetUpVariables* function). To set the initial guess of an ordinary variable:

```
myVar.SetInitialGuess(1.0)
```

To set the initial guess of distributed variables:

```
for i in range(myDomain.NumberOfPoints)
    myVar.SetInitialGuess(i, 1.0)
```

To set the initial guess of distributed variables to a single value for all points in all domains:

```
myVar.SetInitialGuesses(1.0)
```

4.1.6.5. Setting an initial condition

Initial conditions are set in a simulation class (*SetUpVariables* function). It is possible to set initial conditions for the variable value or its time derivative. To set the initial condition of an ordinary variable:

```
myVar.SetInitialCondition(1.0, eAlgebraicIC)
```

To set the initial condition of distributed variables:

```
for i in range(myDomain.NumberOfPoints)
    myVar.SetInitialCondition(i, 1.0, eAlgebraicIC)
```

Re-setting an initial condition

4.1.6.6. Setting an absolute tolerance

Absolute tolerances are set in a simulation class (*SetUpVariables* function). To set the absolute tolerance of both ordinary and distributed variables:

```
myVar.SetAbsoluteTolerances(1E-5)
```

4.1.6.7. Getting a variable value

NOTE: ***It is important to understand that all functions in this and all following chapters (4.1.6.7. to Error: Reference source not found) are used ONLY to construct equation residuals and NOT no to access the real (raw) data.***

For the code snippet how to get a variable value see the examples **I - III** in the section 4.1.4.3. O*perator ()* in *daeVariable* behaves in the same way as the *operator ()* in *daeParameter* class.

4.1.6.8. Getting a variable time derivative

I) To get a time derivative of the ordinary variable we can use the function *dt*. For instance, if we want a time derivative of the variable *myVar* to be equal to some constant, let's say 1.0, we can write:

```
# Notation:
#  - eq is a daeEquation object
#  - myVar is an ordinary daeVariable (not distributed)

eq.Residual = myVar.dt() - 1
```

This code translates into:

$$\frac{\partial myVar}{\partial t} = 1$$

II) Getting a time derivative of distributed variables is analogous to getting a parameter value (see the example **II** in the section 4.1.4.3.). The function *dt* accepts the same arguments and it is called in the same way as the *operator ()* in daeParameter class.

III) Getting an array of time derivatives of distributed variables is analogous to getting an array of parameter values (see the example **III** in the section 4.1.4.3.). The function *dt_array* accepts the same arguments and it is called in the same way as the function *array* in daeParameter class.

4.1.6.9. Getting a variable partial derivative

It is possible to get a partial derivative only of the distributed variables and only for a domain which is distributed (not an ordinary array).

I) To get a partial derivative of the variable per some domain, we can use functions *d* or *d2* (the function *d* calculates a partial derivative of the first order while the function *d2* calculates a partial derivative of the second order). For instance, if we want a first order partial derivative of the variable *myVar* to be equal to some constant, let's say 1.0, we can write:

```
# Notation:
#  - myDomain is daeDomain object
#  - n is the number of points in the myDomain
#  - eq is a daeEquation object distributed on the myDomain
#  - d is daeDEDI object (used to iterate through the domain points)
#  - myVar is daeVariable object distributed on the myDomain

d = eq.DistributeOnDomain(myDomain, eOpenOpen)
eq.Residual = myVar.d(myDomain, d) - 1
```

This code translates into:

$$\frac{\partial myVar(d)}{\partial myDomain} = 1;\ \forall d \in (0, n)$$

Please note that the function myEquation is not distributed on the whole myDomain (it does not include the bounds).

In the case we want to get a partial derivative of the second order we can use the function *d2* which is called in the same fashion as the function *d:*

```
d = eq.DistributeOnDomain(myDomain, eOpenOpen)
eq.Residual = myVar.d2(myDomain, d) - 1
```

which translates into:

$$\frac{\partial^2 myVar(d)}{\partial myDomain^2} = 1;\ \forall d \in (0, n)$$

II) To get an array of partial derivatives we can use functions *d_array and d2_array* which return the *adouble_array* object (the function *d_array* returns an array of partial derivatives of the first order while the function *d2_array* returns an array of partial derivatives of the second order). Again these arrays can only be used in conjunction with mathematical functions that operate on *adouble_array* objects: *sum*, *product*, etc. For instance, if we want variable *myVar* to be equal to the minimal value in the array of partial derivatives of the variable *myVar2* for all points in the domain *myDomain*, we can use the function *min* (defined in *daeModel* class) which accepts arguments of type *adouble_array*. Arguments for the *d_array* function are *daeIndexRange* objects obtained by the call to *daeDomain operator ()*. In this particular example we need a minimum among partial derivatives for the specified points (0, 1, and 3). Thus, we can write:

```
# Notation:
#  - myDomain is daeDomain object
#  - n is the number of points in the domain myDomain
#  - eq is daeEquation object
#  - myVar is daeVariable object
#  - myVar2 is daeVariable object distributed on myDomain

eq.Residual = myVar() - min( myVar2.d_array(myDomain, myDomain( [0, 1, 3] ) ) )
```

This code translates into:

$$myVar = \frac{\partial myVar(0)}{\partial myDomain} + \frac{\partial myVar(1)}{\partial myDomain} + \frac{\partial myVar(3)}{\partial myDomain}$$

4.1.7. Ports

Ports are used to connect two instances of models. Like models, ports can contain domains, parameters and variables. The most important properties are:

- *Name*: string (read-only; inherited from daeObject)

- *CanonicalName*: string (read-only; inherited from daeObject)
- *Type*: daeePortType (inlet, outlet, inlet-outlet)
- *Domains*: daeDomain list
- *Parameters*: daeParameter list
- *Variables*: daeVariable list

The most important functions are:

- *SetReportingOn*

4.1.8. Advanced Equations

4.1.8.1. More complex equation

4.1.8.2. State Transition Networks

4.2. Simulation module

Ovde mi treba class hierarchy.

4.2.1. Basic use

4.2.2. Advanced use (operating procedures)

4.3. DataReporting module

DataReporting module consists of two parts:

- DataReporter
- DataReceiver

DataReporter (a client) is used by the running activity to send the results. There are two different types of data reporters: Local and Remote. Local do not need the server side (data receiver) and they are capable to process the received data internally (to save them into a file, for instance). Remote need the server to send the data.

The module defines two main interfaces: daeDataReporter_t and daeDataReceiver_t. daeDataReporter_t defines an interface used to perform the following tasks:

- Connect/Disconnect to the corresponding daeDataReceiver_t (if any). A good example is a tcpip client which connects and sends data to a tcpip server.
- Send the info about domains and variables in the activity
- Send the results for specified variables at the given time intervals

daeDataReceiver_t is used to receive the information from the activity (a server) and to store the received data for further processing (plotting, for example). The most often it is a tcp/ip server but in general any type of a client/server communication can be implemented (pipes, shared memory, etc).

4.3.1. Basic use

daeDataReporterLocal

daeFileDataReporter

daeTEXTFileDataReporter

daeHTMLFileDataReporter

daeXMLFileDataReporter

daeDelegateDataReporter

daeHybridDataReporterReceiver

daeDataReporterRemote

daeTCPIPDataReporter

4.3.2. Advanced use

daeTCPIPDataReceiver

4.4. Solver module

Ovde mi treba class hierarchy.

4.4.1. Basic use

4.4.2. Setting a third party linear solver