Developing a set of Fortran-to-Python wrappers for Rheinboldt's package of DAE solvers

Oscar Utterbäck

Bachelor's thesis
2014:K12

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

Bachelor's Theses in Mathematical Sciences 2014:K12
ISSN 1654-6229

LUTFNA-4002-2014

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/

DEVELOPING A SET OF FORTRAN-TO-PYTHON WRAPPERS FOR RHEINBOLDT'S PACKAGE OF DAE SOLVERS.

OSCAR UTTERBÄCK

Department of Numerical Analysis
Faculty of Engineering
Lund University

Spring 2014

ABSTRACT - ENGLISH

Assimulo is a Python-based workbench for solving differential equations with the goal of making a multitude of solver methods available through a higher-level interface. By generalizing the problem description, a user is able to apply any solver of their choice without having to redefine the original problem. Following these guidelines, a set of wrappers have been developed to extend Assimulo with Werner Rheinboldt's solvers for quasilinear differential-algebraic equations of index 1 to 3. The solver library, written in Fortran, is linked to Python with the help of F2Py, a Fortran to Python interface generator, and then integrated with Assimulo's original code. In addition to the solvers, Assimulo is extended with a new set of problem classes describing quasilinear problems of first and second order.

ABSTRACT - SWEDISH

Assimulo är ett Python-baserat ramverk för lösning av differentialekvationer med målet att låta användaren välja från en mängd olika lösningsmetoder oberoende av problem. Målet med denna avhandling är att utöka Assimulo med Werner Rheinboldts metod för att lösa differential-algebraiska ekvationssystem. Detta har skett i form av utveckling av ett kommunikationslager mellan Rheinboldt's metoder, skrivna i Fortran, och Assimulo, samt ett flertal nya klasser för att beskriva de problem denna metod riktar sig till.

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION

Assimulo is a package for solving differential equations aimed at providing a high-level interface for a wide variety of solver methods. By using a common interface, Assimulo allows the user to try a multitude of solver methods and compare them from different perspectives.

The solver in question is Werner Rheinboldt's solver of quasilinear DAE system of index 1, 2 and 3. Using theory from differential geometry, viewing DAE systems as differential equations on implicitly defined manifolds, existence proofs are presented in a way that exhibits a directly algorithmic approach. The theory was developed in the 1980's but lacked the computational tools for practical usage. This was however solved in the mid-90's, and the algorithm package was published online in 2000 where it has remained largely unnoticed.

The aim of this thesis is to make this set of solvers available for use within Assimulo. Some research has been devoted to achieve a brief understanding of the algorithm used in the solver routine. The core of the work has been dedicated to developing a Fortran wrapper to make the library available from Python. To describe the problems and customize the usage of the solver, corresponding Python classes have been developed. The glue tool F2Py has been used to enable intercommunication between the two languages.

Chapter 2 begins with a brief theoretical overview of DAE systems and some related concepts such as the index parameter needed to understand what types of problems the solver deals with. Then follows a brief description of the solver with some history.

Chapter 3 details the design of the wrapper, the problem classes created to hold the problem data as well as the solver classes.

Chapter 4 contains usage instructions and an example of how to run the solver, and what results it presents.

Chapter 5 summarizes the thesis and presents the achieved results of the development as well as propositions for future development.

2 THEORY

Differential-algebraic equations (DAE's) is the common name for a system consisting of a mixture of differential and non-differential, popularly called algebraic, equations. DAE's are a generalization of the concept of differential equations and they arise naturally from the mathematical modeling of a variety of problems: multibody dynamics, electrical engineering and chemical engineering to name a few.

2.1 DEFINITION OF A DAE SYSTEM

A first-order implicit DAE has the general form

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = 0. \tag{1}$$

This definition makes no claims regarding linearity or whether a differential term is part of a particular row in the system. This form encapsulates every form of first-order differential equation, including linear, explicit ODE's, as well as fully algebraic systems. As our interest lies in systems of mixed differential and algebraic systems, the following partitioning seems natural:

$$\begin{aligned} \boldsymbol{F_1}(t, \boldsymbol{y}, \boldsymbol{y}') &= 0 \\ \boldsymbol{F_2}(t, \boldsymbol{y}) &= 0. \end{aligned} \tag{2}$$

This separation leaves us with two sets of functions. $\boldsymbol{F_1}$ is a collection of differential equations and $\boldsymbol{F_2}$ is called the system of **constraint equations**. This part of the system imposes a set of constraints that the solution of the differential equations $\boldsymbol{F_1}$ must satisfy. For demonstration we use the example of the pendulum modeled in cartesian coordinates (with length and mass normalized to 1), documented in [EES08]:

$$\begin{aligned} \dot{p}_1 &= v_1 \\ \dot{p}_2 &= v_2 \\ \dot{v}_1 &= -\lambda p_1 \\ \dot{v}_2 &= -\lambda p_2 - g_{grav} \\ 0 &= p_1^2 + p_2^2 - 1 \end{aligned} \tag{3}$$

The first four rows correspond to $\boldsymbol{F_1}$ where the equations of motion for the pendulum are given as four first order ODE's. The last row is $\boldsymbol{F_2}$, the constraint equation in question. λ is an algebraic variable, denoting the string tension, and results from coupling with the constraint equation. The physical interpretation of this constraint defines the length of the pendulum to be constant. Any solution to the system of differential equations must also satisfy this condition. This is equally relevant for the initial conditions of the system, and as such we refer to **definition 5.1.1** in [EES08], that states: A function $x : [t_0, t_e] \to \mathbb{R}^n$, $x(t) = (p(t), v(t), \lambda(t))$ fulfilling the smoothness requirement $p \in \mathcal{C}^2[t_0, t_e]$, $v \in \mathcal{C}^1[t_0, t_e]$, $\lambda \in \mathcal{C}^0[t_0, t_e]$ is called a solution of (3) if it satisfies (3). The value $x(t_0)$ is then called a **consistent** initial value. Because of the set of constraints we are no longer free to choose (to some degree) arbitrary initial conditions, as is the case for pure ODE systems.

2.2 INDEX OF A DAE SYSTEM

The index is a parameter used to measure the complexity and difficulty of working with a certain DAE system. It varies from system to system, between different solution methods and can even vary between different parts of the solution region within one system [CLP08]. However, there are a few details shared among the definitions [PJR02]. Considering a DAE system that is eventually reducible to an explicit ODE system, then:

- the reduction is performed via a number of recursive operations,
- the index is defined as the number of operations required for the reduction,
- the system has an index of 0 **iff** the system is an ODE.

One of the more common definitions used is the differentiation index. Through repeated differentiation of the algebraic equations, substituting with the differential equations as needed, one will eventually end up with a system of nothing but differential equations. In this case the index is defined as the number of differentiations. Applying this procedure to the last row of (3) results in the following:

$$0 = p_1^2 + p_2^2 - 1 \tag{4}$$

$$0 = p_1 v_1 + p_2 v_2 \tag{5}$$

$$0 = v_1^2 + v_2^2 - \lambda(p_1^2 + p_2^2) - p_2 g_{grav} \tag{6}$$

From top to bottom, the three constraint equations are of index 3, 2 and 1 respectively. Differentiating (6) again and substituting where necessary will yield an explicit ODE of the form $\dot{\lambda} = f(\boldsymbol{p}, \boldsymbol{v})$. This is the **underlying ODE** to the DAE system(3).

2.3 NUMERICAL TREATMENT OF DAE'S

"Differential-algebraic equations are not ODE's" - Linda R. Petzold

The solver package developed by Rheinboldt stems from a series of articles written in the 1980's. Historically, the numerical handling and solution of differential-algebraic systems has been left to methods originally designed for systems of ODE's. It has been shown that DAE's in certain cases can be solved by slight, or advanced, modification of methods designed for systems of stiff ODE's. This procedure is far from ideal and becomes increasingly difficult with higher index systems.

The quote by Petzold is the title of one of her papers [Pet82] on the subject of working on the general first order equation systems of the form

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = \boldsymbol{0}. \tag{7}$$

In her paper she outlines several causes of difficulties that may arise when working with DAE systems as if they were ODE systems. To discussion is brought not only difficulties that occur when working with problems with discontinuities but even with very simple, higher-index systems consisting of smooth functions. The following example is presented in Petzold's article. Applying Euler's backward differentiation formula (BDF)

$$\boldsymbol{y}'(t) = \frac{\boldsymbol{y}(t_n) - \boldsymbol{y}(t_{n-1})}{h} \tag{8}$$

on the simple index 3 system

$$y_2'(t) = y_1(t) \qquad y_3'(t) = y_2(t) \qquad 0 = y_3(t) - g(t) \tag{9}$$

it is found that taking a single step from the exact solution results in an error that is independent of the step size, and as such it is impossible to keep the error small after taking a single step from the exact solution. A naïvely implemented ODE integrator would fail on the first step.

In another example demonstrating variable step size methods on the same system, the order of the error is worse than expected unless very specific sequences of step sizes are used. In a third example using an index 2 system, the order of the error is again worse than expected, without any obvious remedies.

2.4 DAE SYSTEMS AS ODE'S ON MANIFOLDS

In [Rhe84] a new approach is presented. Instead of treating the system as a special case of ODE's, focus is placed on the constraint equations. Under certain conditions the system of constraint equations implicitly define a manifold on which the solution to the differential equations are **consistent**. On this manifold, the DAE locally reduces to a ODE, and the solution to this ODE gives the solution to the DAE. The local ODE is obtained via a parametrization of the constraint manifold.

Following is a short pseudocode outline of the solver routine.

```
at initial point y_0:
    compute initial parametrization of constraint manifold

let y_i be the computed solution at time t_i
while t_i < t_final:
    solve local ODE system
    use solution to compute y_i
    if chosen point is outside area of validity:
        compute new parametrization\todo{add caption?}
```

Along with the algorithm, some interesting details are brought to light when revisiting problems mentioned in Petzold's paper. One of the theorems used to motivate the algorithm defines the term 'algebraic incompleteness'. In essence, certain systems are only reducible to an ODE if the constraint manifold is restricted to a lower-dimensional space. This is the case for the pendulum in (3). If only the original system is regarded, the existence theorem does not hold for the pendulum, unless certain conditions are met. The first of these is that $p_1v_1 + p_2v_2 = 0$, which is exactly the equation obtained in the first index reduction process using the differentiation index. Augmenting the pendulum with this equation yields a new system with a new condition: $v_1^2 + v_2^2 - \lambda(p_1^2 + p_2^2) - p_2 g_{grav} = 0$. This is again the same result obtained via the differentiation reduction. It is only if these additional constraints are fulfilled that the system has a solution.

2.5 PROBLEM FORMULATIONS

The solver package comes with routines for solving six types of DAE systems. The solver routine is able to handle implicit first-order index 1 systems:

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = 0, \tag{10}$$

quasilinear index 1 and 2 systems:

$$\begin{aligned} \boldsymbol{A}(t,\boldsymbol{y})\boldsymbol{y}' + \boldsymbol{B}(t,\boldsymbol{y})\boldsymbol{\lambda} &= \boldsymbol{f}(t,\boldsymbol{y}) \\ \boldsymbol{g}(t,\boldsymbol{y}) &= \boldsymbol{0}, \end{aligned} \tag{11}$$

where index 1 and 2 corresponds to $\boldsymbol{B} = \boldsymbol{0}$ and $\boldsymbol{B} \neq \boldsymbol{0}$ respectively, and quasilinear index 3 systems:

$$\begin{aligned} \boldsymbol{A}(t,\boldsymbol{y},\boldsymbol{y}')\boldsymbol{y}'' + \boldsymbol{B}(t,\boldsymbol{y},\boldsymbol{y}')\boldsymbol{\lambda} &= \boldsymbol{f}(t,\boldsymbol{y},\boldsymbol{y}') \\ \boldsymbol{g}(t,\boldsymbol{y},\boldsymbol{y}') &= \boldsymbol{0}. \end{aligned} \tag{12}$$

The solver also contains routines for working with two special cases of (12), namely problems with non-holonomical constraints of index 2 as well as Euler-Lagrange equations of index 3. These correspond to systems where the matrix $\boldsymbol{B}$ has a certain relation to the constraint equations. In the case of non-holonomical problems, the system takes the form

$$\begin{aligned} \boldsymbol{A}(\boldsymbol{y})\boldsymbol{y}'' + \frac{d\boldsymbol{g}^T}{d\boldsymbol{y}'}\lambda &= \boldsymbol{f}(t,\boldsymbol{y},\boldsymbol{y}'), \\ \boldsymbol{g}(t,\boldsymbol{y},\boldsymbol{y}') &= \boldsymbol{0}. \end{aligned} \tag{13}$$

In the second case, the system is usually given by

$$\begin{aligned} \boldsymbol{A}(\boldsymbol{y})\boldsymbol{y}'' + \frac{d\boldsymbol{g}^T}{d\boldsymbol{y}}\lambda &= \boldsymbol{f}(\boldsymbol{y},\boldsymbol{y}'), \\ \boldsymbol{g}(\boldsymbol{y}) &= \boldsymbol{0}. \end{aligned} \tag{14}$$

These systems frequently arise from the modeling of multibody systems. In the case of Euler-Lagrange problems the system typically has a different form, and the interested reader may find the derivation from the original equations in [CKL12]. This system is generally of index 3 making it extra interesting as higher-index problems are typically more difficult to deal with compared to their lower-index counterparts.

2.6 SOLVER DETAILS

The solver performs computations using the different parts of the DAE system separately, so it requires access to $\boldsymbol{A}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{B}$. It also works with the derivative of the constraint equation $\boldsymbol{dg}$, and in the case of index 3 problems it uses the second derivative with the state derivative applied twice: $\boldsymbol{d^2g}(\boldsymbol{y})(\boldsymbol{y}', \boldsymbol{y}')$. The second derivative, $\boldsymbol{d^2g}(\boldsymbol{y})$, is a rank 3 tensor where the elements are $\boldsymbol{d^2g}_{ijk} = \frac{\partial^2 g_i}{\partial y_j \partial y_k}$. The application of the state derivative yields first a rank 2 tensor, and subsequently the second application yields a rank 1 tensor.

The solver also requires a certain structure regarding the derivative matrices. Regardless of whether or not the system is time-invariant, the solver requires the derivative with regard to time. As such, time-invariant problems will end up with matrices containing columns of zeros. In a theoretical, time-independent system with three constraint equations and two state variables, the matrix would take the following form:

$$\begin{pmatrix} \frac{\partial g_1}{\partial t} & \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} \\ \frac{\partial g_2}{\partial t} & \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} \\ \frac{\partial g_3}{\partial t} & \frac{\partial g_3}{\partial y_1} & \frac{\partial g_3}{\partial y_2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} \\ 0 & \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} \\ 0 & \frac{\partial g_3}{\partial y_1} & \frac{\partial g_3}{\partial y_2} \end{pmatrix} \tag{15}$$

In a similar manner, the derivative vector applied to the tensor will contain $\frac{dt}{dt}$ in the first position as either a 1 or a 0 depending on if the problem is time-dependent.

3 WRAPPER DESIGN AND IMPLEMENTATION

Rheinboldt's solver package comes as a freestanding Fortran module with the intention that the user writes a Fortran program containing the necessary functions and data pertaining to the problem. Following is a brief outline of the prerequisites needed for the solver to work. This exists in a more detailed description in the readme[Rheoo] .

- A main program that:
  - sets problem data (i.e. number of dimensions, initial state)
  - sets solver settings (e.g. tolerance, initial step, final integration time)
  - calls the solver routine
  - calls the cleanup routine
- A module for global data pertaining to the problem
- A subroutine `daefct` that takes a function name and the state vectors as input, and returns the computed function value as output
- A subroutine `solout` that handles output at intermediate points during the integration

Once the solver routine is called, it runs from the initial state until termination. A fully successful integration runs until the final integration time. It may however be terminated earlier, for example due to inconsistent initial values or singularities at some point in the solution.

3.1 WRAPPER OUTLINE

The above mentioned prerequisites give a general idea of how the wrapper should be implemented. The wish is to perform a function call from Python, supplying problem information and solver settings as input parameters. This function should return information and statistics regarding the simulation, as well as a multidimensional array containing the solutions for each time step.

To enable communication between Fortran and Python, the Fortran-to-Python interface generator F2Py is used which allows us to compile fortran modules into python modules. F2Py is included in the package numpy. F2Py takes a number of Fortran files, both source files and compiled modules, and generates a callable Python module which allows the user to reach variables and functions contained within the Fortran module.

The main program is embodied in the file `rbdae_wrapper.f95`. This is compiled into a Python module by F2Py and enables access to a method `main` that takes solver options, problem callback functions and initial states as input, and returns the simulation statistics as output. The resulting simulation states are available in the $n_{steps} \times n_{vars}$ array `x_sol` inside the module.

3.2 PROBLEM CLASSES

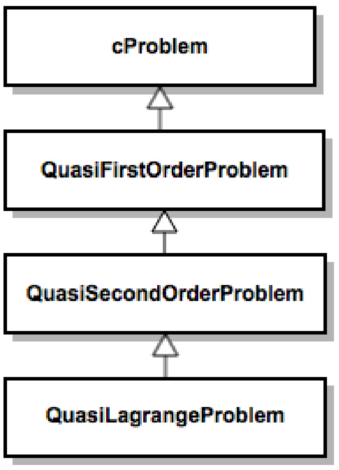

Figure 1: Inheritance between the problem classes.

To represent the problems outlined in section 2.5 a set of problem classes have been created in Python. These are

- `QuasiFirstOrderProblem`
- `QuasiSecondOrderProblem`

- `QuasiLagrangeProblem`

`QuasiFirstOrderProblem` is used for the index 1 and 2 problems in (11), `QuasiSecondOrderProblem` is the base class for all the quasilinear second order problems corresponding to (12) and is inherited by `QuasiLagrangeProblem` as shown in figure 1. The topmost base class is Assimulo's `cProblem`.

To make use of these classes, the user needs to specify the problem functions, the initial states and the number of algebraic variables in the problem. To identify the algebraic variables in the problem, use is made of Assimulo's `algvar` construct which is saved as a member variable. If this is omitted in the initialization of the class, all variables are treated as differential. `algvar` is a boolean vector where `True` at position `k` denotes that y_k is a differential variable, whereas `False` would denote an algebraic variable.

In certain cases certain problem functions are unused. For example in the non-holonomic and the Euler-Lagrange case, the solver does not use $\boldsymbol{B}$ explicitly. In this case, the callback sent to the wrapper is replaced with the private function `_no_cb_defined`. This is a technicality which lets the wrapper accept any combination of callback functions without complaining.

The solver requires the user to specify which problem type is being worked with. This is done through the member variable `index` which as of now can take the values `1, 2, 3, eulag` corresponding to the quasilinear problems of these indices. This is done automatically by the problem class upon initialization.

3.3 SOLVER CLASSES

To use the solvers, a set of classes are developed to handle the different problem classes.These are

- `RBDAEImplicit`
- `RBDAEQuasi`
- `RBDAELagrange`

The solvers inherit from Assimulo's `Implicit_ODE` as can be seen in figure 2. This class contains a method `simulate` that is used to perform a simulation. The calling of the solver is done inside `simulate` via an internal function `integrate` which takes the

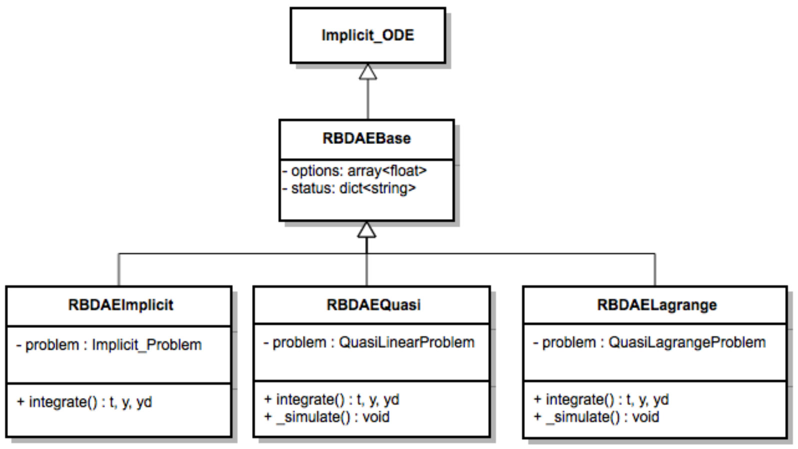

Figure 2: Relationship between the solver classes.

current state and the final integration time as input and returns the resulting computed states for all points in time. To incorporate the solver in Assimulo, each of the solvers are provided with their own implementation of `integrate` that calls the wrapper. The inheritance of `Implicit_ODE` gives access to a multitude of functionality. Of these, the solvers use Assimulo's plotting and statistics functions.

The solvers are initialized with their corresponding problem class via the factory class `RBDAE`. This class examines the problem type and returns the proper solver.

The solver for implicit systems does not need a special problem class. It may make use of Assimulo's `Implicit_Problem`. It does however at minimum require the jacobian of the problem function.

3.4 CALLBACK FUNCTIONS

The solver requires certain components of the DAE system supplied as callback functions. These are the functions specified in section 2.5. For quasilinear index 3 problems:

$$\begin{aligned} A(t,y,y')y'' + B(t,y,y')\lambda &= f(t,y,y') \\ g(t,y,y') &= \mathbf{0}. \end{aligned} \tag{16}$$

In this case we require the functions A, g, dg and f. In the event that the problem is of index 3, the solver uses the second derivative of the constraint equations $\boldsymbol{d^2g}$ with the state derivative $\boldsymbol{y}'$ applied twice.

The solver routine is programmed in such a way that that it always assumes the variable t to be a part of the state vector. This imposes two demands on the callback functions.

- $\boldsymbol{dg}$ and $\boldsymbol{d^2g}$ must contain the derivative of t regardless of the problems dependence on t.
- The state vector applied to $\boldsymbol{d^2g}$ contains $\frac{dt}{dt}$ in the first position.

If the problem is independent of the time, $\frac{dt}{dt}$ is defined to be 0, otherwise 1. The column with derivatives with regards to t will be entirely zeros.

4 USAGE INSTRUCTIONS AND EXAMPLES

4.1 INSTALLATION

The wrapper package was developed on a 64bit OSX platform using the following packages and softwares:

- gfortran 4.8.1
- Python 2.7.6
- numpy 1.8.0
- Assimulo 2.6

It has been tested to work with Assimulo 2.5 and 2.6. To compile the program from source, precompile the modified files containing the solver routines before using F2Py to create the module:

```
$ gfortran -c dae_lib.f95 dae_solver.f95
$ f2py -m rheinboldt -c rbdae_wrapper.f95 dae_lib.o dae_
    solver.o
```

The `-m` flag decides the module name to be included in Python. Place the resulting `rheinboldt.so` in the same directory as `RBDAE.py` and `QuasiLinearProblem.py`.

4.2 USAGE EXAMPLE

To enable the usage of the solvers, import the solver factory class `RBDAE` from `RBDAE.py`, and the problem class corresponding to the problem from `QuasiLinearProblem.py`.

```
from RBDAE import RBDAE
from QuasiLinearProblem import QuasiFirstOrderProblem,
    QuasiSecondOrderProblem, QuasiLagrangeProblem
```

Following is an example of simulating the Pendulum as an Euler-Lagrange system. The first order system in (3) written in its original second order form:

$$\begin{aligned}\ddot{p}_1 + \lambda p_1 &= 0 \\ \ddot{p}_2 + \lambda p_2 &= -g_{grav} \\ (p_1^2 + p_2^2 - 1)\frac{1}{2} &= 0.\end{aligned} \tag{17}$$

$\boldsymbol{A}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{dg}$ can all be identified directly. The differentiation of $\boldsymbol{g}$ yields the vector $[0, p_1, p_2]$ where the first element is $\frac{\partial g}{\partial t}$. The multiplication of the constraint equation by 0.5, handles the factor of 2 in front of the derivative without changing the solution. The constant g_{grav} is given the value 13.7503671 to make the period of the pendulum as close as possible to 2.

The second derivative $\boldsymbol{d^2g}$ is also required, and results in a 3×3 matrix with zeros in the first column. This is then multiplied twice with the extended derivative vector $[\frac{dt}{dt}, \boldsymbol{y}']$ with $\frac{dt}{dt} = 0$.

Using the command `sim.plot()` we have easy access to plotting the resulting figures.

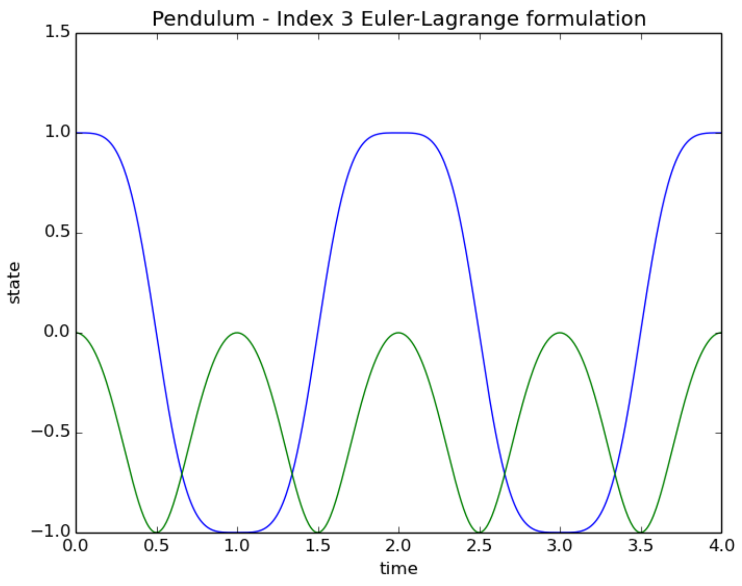

Figure 3: Pendulum states.

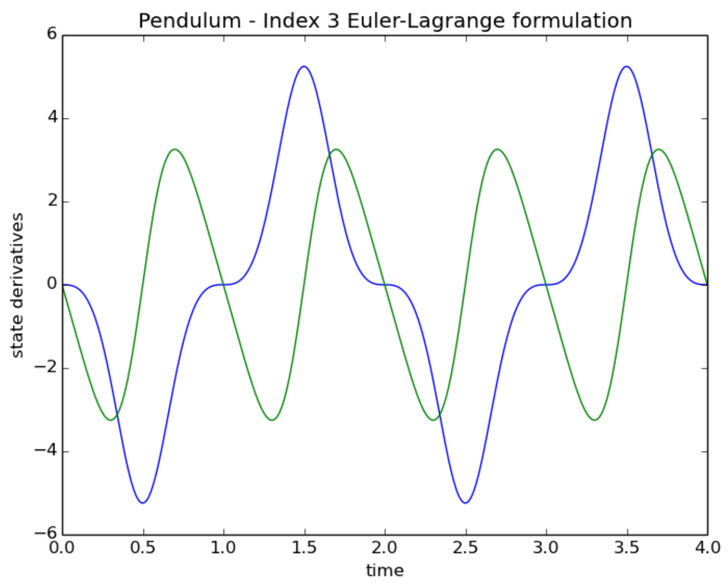

Figure 4: Pendulum states derivatives.

The following code shows how the example above is implemented in Python.

```
from RBDAE import RBDAE
from QuasiLinearProblems import QuasiLagrangeProblem
from matplotlib import pyplot as plt
import numpy as np


def a(t, y):
    return np.eye(2)

def h(t, y, yd):
    rhs_0 = 0.
    rhs_1 = -13.7503671
    return np.array([rhs_0, rhs_1])

def g(t, y):
    ind_3 = (y[0]*y[0] + y[1]*y[1] - 1.) * 0.5
    return np.array([ind_3])

def dg(t, y):
    dg_mat = np.zeros((1, 3))
    dg_mat[0, 0] = 0. # dg/dt
    dg_mat[0, 1] = y[0]
    dg_mat[0, 2] = y[1]
    return dg_mat

def d2g(t, y, yd):
    d2g_mat = np.zeros((3, 3))
    d2g_mat[2, 2] = 1.
    d2g_mat[1, 1] = 1.
    yd = np.concatenate(([0.], yd))
    return np.dot(np.dot(d2g_mat, yd), yd)


y0 = np.array([1., 0., 0.])
yd0 = np.array([0., 0., 0.])
t0 = 0.0
tf = 4.0
algvar = [True, True, False]
euler_prob = QuasiLagrangeProblem(h, g, dg, d2g, a, t0, y0,
    yd0, None, algvar)
euler_prob.name = 'Pendulum - Index 3 Euler-Lagrange
    formulation'

sim = RBDAE(euler_prob)
print 'Starting simulation'
t, y, yd = sim.simulate(tf)
print 'Simulation complete'
```

5 RESULT

The solver package discussed in this thesis originally came in a format that required the user to write a surrounding program for handling input and output in an way dependent on this particular package. The documentation has in some cases been less than helpful in describing the intention of certain parts of the code as well as the mathematics surrounding it. It is clear that the intended audience of this package is not a large one and this may be one of the reasons why this solver has remained largely unnoticed on the netlib repository.

The resulting wrapper has enabled access to the solver package from Python, and as such the need to write and compile a surrounding program handling the input and output has been removed. Python's high-level functionality and easily readable syntax make the solvers easily accessible.

5.1 FUNCTIONALITY PROVIDED

The wrapper developed for this thesis provides the following functionality:

- High-level access to 5 out of 6 solver routines
- Access to Assimulo's plotting commands
- Access to simulation statistics
- Access to choose number of result points within the integration interval.

By giving Assimulo access to the solver the simulation results may now be plotted, saved and reused as needed. The original code wrote the results to a file that would then have to be read manually or parsed, and any data in the memory was discarded as the program terminates. The implementation of the solver for non-holonomical problems should be trivial but was left out in favor of prioritizing other tasks.

5.2 SUGGESTIONS FOR FUTURE DEVELOPMENT

The solvers were developed in a stand-alone manner. Full integration with Assimulo is still on the todo-list, and following is a set of suggestions for future development.

1. Integrate the problem classes with `special_system`
2. Adding the solver class into the library
3. Examine whether three separate problem classes are necessary to cover all the cases
4. Examine the solver classes with the same intention
5. Handle the result of the second derivative in index 3 cases
6. Develop an installation script

5.2.1 *Integration with special_system*

The problem classes are suitable to include in the problem generator `special_system`. This means Assimulo can keep it's generalized problem description so that the user can obtain the problem form suitable for the solver they want to use. Doing this will also remove further obligation from the user with regard to choosing the correct problem class, and they may focus on the mathematical model.

5.2.2 *Integration with the library*

The solver is developed as a stand-alone class. To make it a part of Assimulo it needs to be fitted into the solver library under the package `assimulo.solvers`.

5.2.3 *Examine the need for multiple solver and problem classes*

During the development, the classes describing the problems and the classes handling the solvers have taken various forms as details surrounding both the extension of Assimulo and working with the solver code have emerged. In its current state, the problem classes describe problems from a mathematical point of view with one class for first order problems, one class for second order problems and one for the special case of Euler-Lagrange. It might be of interest to keep the amount of classes as few as possible, especially with regard to the interest of using the `special_system` class to generate the correct problem.

5.2.4 *Handle the result of the second derivative*

In the index 3 cases, the solver also returns results for the second derivative of the state vector, $\boldsymbol{y}''$. Assimulo does not as of now handle these results in any way, and they are left forgotten.

5.2.5 *Develop an installation script*

The compilation of the solver and the generation of the wrapper is as of now done manually via the commands outlined in section 4.2. Fortran is platform-dependent and should preferably be compiled from source. An installation script should be created which can be used by Assimulo.

Part I

APPENDIX

BIBLIOGRAPHY

[CKL12] Sumit Jain C. Karen Liu. A Quick Tutorial on Multibody Dynamics. Online tutorial, June 2012. (Cited on page 7.)

[CLP08] S. L. Campbell, V. Hoang Linh, and L. R. Petzold. Differential-algebraic equations. *Scholarpedia*, 3(8):2849, 2008. revision 91199. (Cited on page 3.)

[EES08] Claus Führer Edda Eich-Soellner. *Numerical Methods in Multibody Dynamics*. Self-published, 2008. (Cited on pages 2 and 3.)

[Pet82] Linda R. Petzold. Differential-Algebraic Equations are not ODE's. *SIAM J. Sci. Stat. Comput.*, 3(3):367–384, 1982. (Cited on page 4.)

[PJR02] Werner C. Rheinboldt Patrick J. Rabier. *HANDBOOK OF NUMERICAL ANALYSIS, VOL VIII*, chapter Theoretical and Numerical Analysis of Differential-Algebraic Equations, pages 183–540. Elsevier Science B.V., 2002. (Cited on page 3.)

[Rhe84] Werner C. Rheinboldt. Differential-Algebraic Systems as Differental Equations on Manifolds. *Mathematics of Computation*, 43(168):473–482, October 1984. (Cited on page 5.)

[Rhe00] Werner C. Rheinboldt. Solver package readme, November 2000. (Cited on page 8.)

Bachelor’s Theses in Mathematical Sciences 2014:K12
ISSN 1654-6229

LUTFNA-4002-2014

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/