HiT Report No. 5

Anushka Perera

Using CasADi for Optimization and Symbolic Linearization/Extraction of Causality Graphs of Modelica Models via JModelica.Org

Telemark University College

HiT Report No.5

Anushka Perera

Using CasADi for Optimization and Symbolic Linearization/Extraction of Causality Graphs of Modelica Models via JModelica.Org

Telemark University College
Faculty of Technology

HiT Report No. 5

ISBN 978-82-7206-380-0
ISSN 1894-1044

Telemark University College
P.O. Box 203
3901 Porsgrunn

Phone 35 57 50 00
Fax 35 57 50 01
http://www.hit.no/

Preface

My visit to Modelon AB was basically for two purposes; (1) to participate in Dymola Introduction Course I & II (25th 29th Nov. 2013) and (2) an extended stay at Modelon to work on several tasks (till 12th Dec. 2013). This report mainly concerns (2). I had already had some experience with Dymola prior to the course and I had gained a lot at the end of the course. In particular, working with discrete events, troubleshooting models and numerical problems, selecting the right solver, embedding external C and Fortran model components, exporting models as stand-alone executables or DLLs and interfacing Dymola with MATLAB and Simulink were quite interesting.

The tasks to be worked during my extended stay are: (1) linearizing dynamic models which are modeled according to the Modelica standards at a specified operating point and then make system matrices available for linear system analysis (a good possibility is to use Python Control Systems Toolbox. See in `http://python-control.sourceforge.net/manual/intro.html`) (2) extracting the structural information of complex dynamic systems, as this can be useful for example for analyzing structural observability/controllability, decomposing large scale systems into subsystems, etc. using graph-theoretic approaches and (3) accessing IPOPT solver in Python. All tasks have been more or less completed.

Anushka Perera, Porsgrunn, February 04^{th}, 2014.

Acknowledgement

Thanks goes to a few. Firstly, prof. Bernt Lie — my main PhD supervisor — is he who recommended the course at Modelon. Also he communicated with Johan Åkesson who is the technical director at Modelon and arranged an extended stay at Modelon so that I could work on some tasks related to my research work there in Modelon Head Office. The two lecturers Johan Widhal and Stéphane Velut who conducted an interesting, fruitful, as well as a well organized Dymola workshop. Johan Åkesson arranged me to work with Toivo Henningsson and Toivo was very supportive throughout. Fredrik Magnusson should also be mentioned here. Thanks to Joel Andersson, Joris Gillis and others who answered my questions at CasADi FAQ; this was of great help. Finally, thanks a lot Stéphane Velut, I enjoyed the time I spent with you and your family in the evening on Dec. 1^{th}, 2013.

Contents

Chapter 1

Introduction

The Modelica language has become a handy modeling tool for multi-domain complex physical systems. Its object-oriented and equation-based approach eases the modeling process to a great extent. Dynamic models which are scripted according to the Modelica standards demand a simulation environment. There are many simulation tools such as OpenModelica, Dymola, JModelica.org, OPTIMICA Studio, etc. OpenModelica and JModelica are two examples of free Modelica-based simulation environments. Dynamic Optimization is also an important aspect in control engineering. JModelica.org, among others, provides support of Dynamic Optimization and it is free. Apart from being freeware, there are many other attractive features in it. Python, which is again free, is the scripting language used in the JModelica.org platform and it is possible to integrate various Python libraries on demand so as to get required functionalities that we seek. Often used Python libraries are Numpy, Scipy, MatPlotLib, etc. The JModelica.org installer installs those necessary packages automatically. More interestingly, JModelica interfaces to CasADi which is a symbolic framework for Automatic Differentiation and Nonlinear Optimization. The JModelica.org-CasADi interface is at our main interest in this report due to two main reasons: (1) it is possible to translate Modelica/Optimica models into a symbolic representation via the JModelica.org-CasADi interface and use the power of CasADi to find Jacobian matrices both symbolically and numerically — this can be used to linearize dynamic models and in particular symbolic Linearization is possible to use in analyzing structural properties of dynamic systems, and (2) CasADi is already interfaced with state of the art nonlinear optimizers (e.g. IPOPT.), integrators (e.g. SUNDIALS.), etc. and consequently, JModelica.org freely inherit those well-known integrators/nonlinear solvers so that we can use them in Python.

The report treats three main tasks: (1) interfacing nonlinear optimizers and integrators — like IPOPT, CVODES, etc. — into Python, (2) linearization of Modelica models at a given operating point, and (3) extracting causality of Modelica models. All these objectives are achieved via JModelica.org within its limitations. Most of the content of this report depends on the JModelica.org-CasADi interface. Therefore a good understanding of CasADi is a prerequisite. Chapter 2 gives an introduction to basic symbolic manipulation in CasADi with several examples. Here defining and usage of symbolic expressions and functions are explained. A discussion on how to use built-in optimizers and integrators available in CasADi is given in Chapter 3. Chapters 2 and 3 cover tasks (1) and (2). Several modifications are made in the Python script casadi_interface.py (which is available in the JModelica.org installation directory.) to linearize Modelica models both symbolically and numerically as well as to do structural observability analysis, system decompositions, etc. This is given Chapter in 4. It is assumed that JModelica.org has been installed.

Chapter 2

Basics of CasADi

CasADi is a software tool for automatic/algorithmic/computational differentiation [1] and nonlinear optimization. Joel Andersson in answering one of my questions on 24-01-2014 at CasADi's FAQs;

> "But CasADi has come to denote the whole optimization framework and not just the symbolic core. So if you write about it, we'd prefer that you write "the optimization framework CasADi". We avoid to call it either a "CAS" or an "AD-tool" since it's not really intended to replace either."

The CasADi framework may be equipped with for example a Python front-end, and it interfaces with powerful nonlinear optimizers (KINSOL, IPOPT, KNITRO, WORSHOP, etc.), DAE/ODE integrators (CVODES, IDAS, etc.), linear solvers (MUMPS, MA27, etc.), etc. CasADi is easily installed via JModelica.org and may use Python in either Pylab or Ipython mode. This avoids the installation hassle to a great a great extent. Also the latest release of OpenModelica (version 1.9.0) features with CasADi. See in `https://openmodelica.org/`. But this will not be under this report.

There are five main base classes upon which the structure of CasADi is based: SX, SXMatrix, DMatrix, FX (several sub-classes of the FX class are SXFunction, MXFunction, LinearSolver, ImplicitFunction, GenericIntegrator, Simulator, ControlSimulator, NLPSolver, and QP Solver.) and MX. See the class diagram given in Figure 2.1 which is available at `http://casadi.sourceforge.net/api/html/d5/d01/classCasADi_1_1FX.html#diagram`. SX, SXMatrix and MX classes are to represent symbolic variables whilst FX is the base class for all CasADi's functionality. Chapter 2 will discuss SX, SXMatrix, DMatrix, SXFunction, MX and MXFunction classes and Chapter 3 is dedicated to the FX class and its derived classes.

2.1 SX and SXMatrix

Scalar and matrices with scalar entities are created using SX and SXMatrix classes respectively. Consider a sample expression given below;

$$y = f(x_1, x_2) = \cos\left(\left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - \exp(x_2)\right] \cdot \left[\frac{x_1}{x_2} - \exp(x_2)\right]\right)$$

x_1, x_2 and y are floating point numbers. Now, we will see how to represent these variables symbolically. Start Python either in Pylab or IPython mode. Then import CasADi package using the Python command `from casadi import *`. Now you can access the SX class (also other classes: SXMatrix, DMatric, FX and MX). Define x_1 and x_2 then using these symbolic variables, formulate an expression for y as follows;

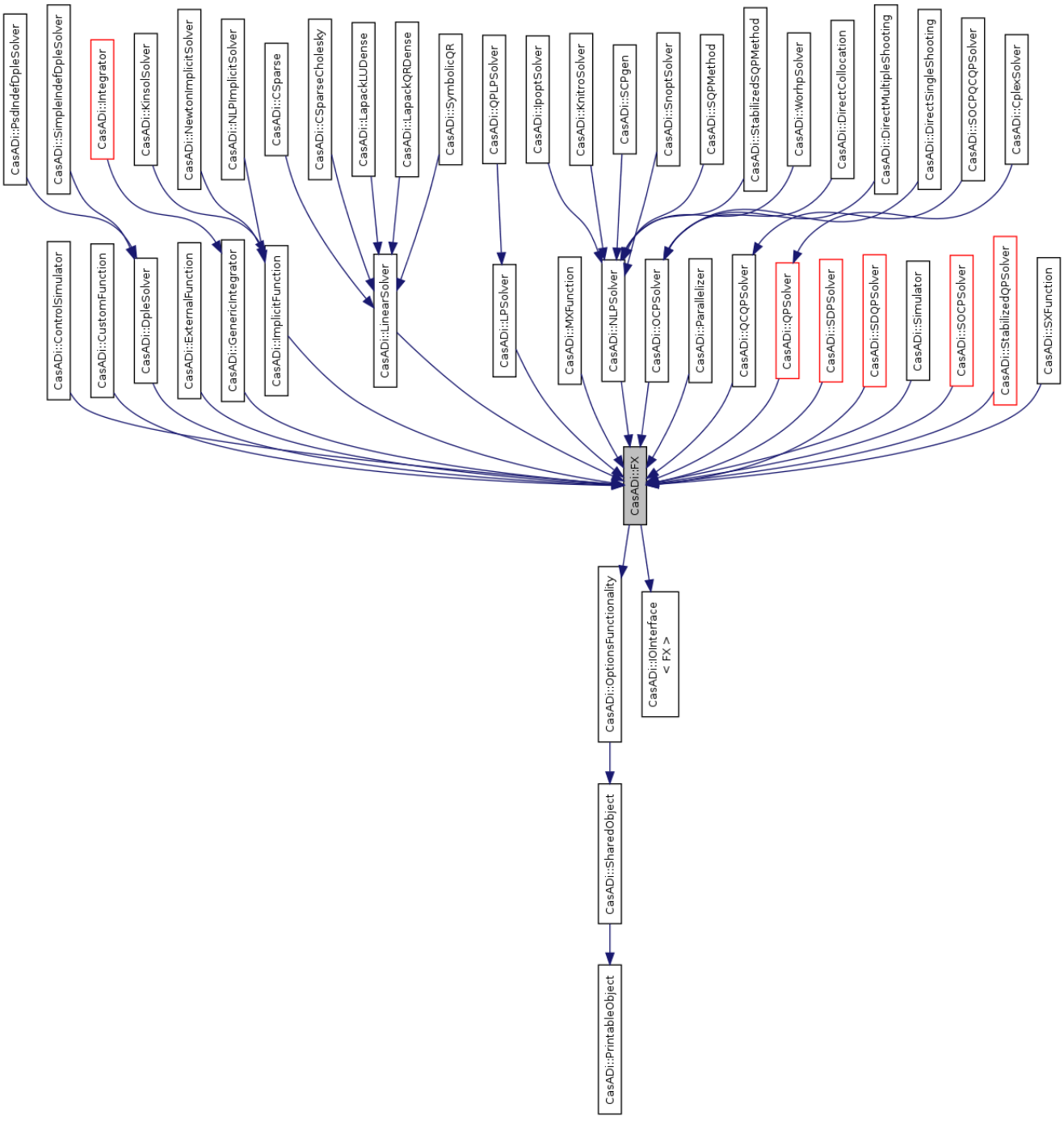

Figure 2.1: The Class Diagram of CasADi. Taken from https://github.com/casadi/casadi/wiki.

```
from casadi import *
from casadi.tools import *
x1 = SX("x1")
x2 = SX("x2")
y = cos((sin(x1/x2)+x1/x2-exp(x2))*(x1/x2-exp(x2)))
dotdraw(y)
printCompact(y)
```

The `dotdraw()` function is used to visualize the evaluation procedures graphically for given symbolic expressions. The Python package `pydot` has to be installed in order to use `dotdraw()` function and also `from casadi.tools import *` command must have been implemented before `dotdraw()`. I am working on Ubuntu 12.04 LTS and I could manage to install `pydot` package just with a click as pydot is available in Ubuntu Software Center. Note that JModelica.org installer doesn't install pydot package. The computational graph of y is constructed using the `dotdraw(y)` command is shown in Figure 2.2 and it depicts $\frac{x_1}{x_2}$ is evaluated 3 times (i.e. 3 edges related to the same evaluation task, $\frac{x_1}{x_2}$) and e^{x_2} twice (i.e. 2 edges for the same task, e^{x_2}). In total, there are 11 elementary operations with some repetitions. We may eliminate common sub-expressions (CSEs)[1] from y, which is something we might do manually. It can be seen that $\frac{x_1}{x_1}$ and e^{x_2} are repeated several time in y, so we can eliminate these common sub-expressions. Define, $z_1 \triangleq \frac{x_1}{x_1}$ and $z_2 \triangleq e^{x_2}$. Then, $y = \cos\left(\left[\sin\left(z_1\right) + z_1 - z_2\right] \cdot \left[z_1 - z_2\right]\right)$. Now run following code and see Figure 2.3:

```
from casadi import * # if you haven't imported already!
from casadi.tools import * # if you haven't imported already!
x1 = SX("x1")
x2 = SX("x2")
z1 = x1/x2
z2 = exp(x2)
y = cos(sin(z1)+z1-z2)*(z1-z2)
dotdraw(y)
printCompact(y)
```

Observe the output from `printCompact(y)` and notice that now x_1/x_2 and e^{x_2} are evaluated only once. The SXMatrix class is used to create symbolic matrices where elements of SXMatrix objects are SX instances. The following example demonstrate how to create symbolic matrices as objects from the SXMatrix class. We can also use `ssym` to create SXMatrices. See the code given below:

```
# Use SX class to create SX objects
x1 = SX("x1")
x2 = SX("x2")
x3 = SX("x3")
x4 = SX("x4")
# Use SXMatrix class and ssym to create SXMatrix objects
X1 = SXMatrix([[x1,x2],[x3,x4]])
X2 = ssym("X2",4,5)
X3 = SXMatrix.ones(3,1)
```

2.2 SXFunction

The SXFunction class is a subclass of the FX class. We can formulate functions that involve only scalar operations as an SXFunction class' object. Let's say that we have $y = f\left(x_1, x_2\right)$. f

[1] Thanks goes to Joel Andersson and Greg Horn, @CasADi FAQs.

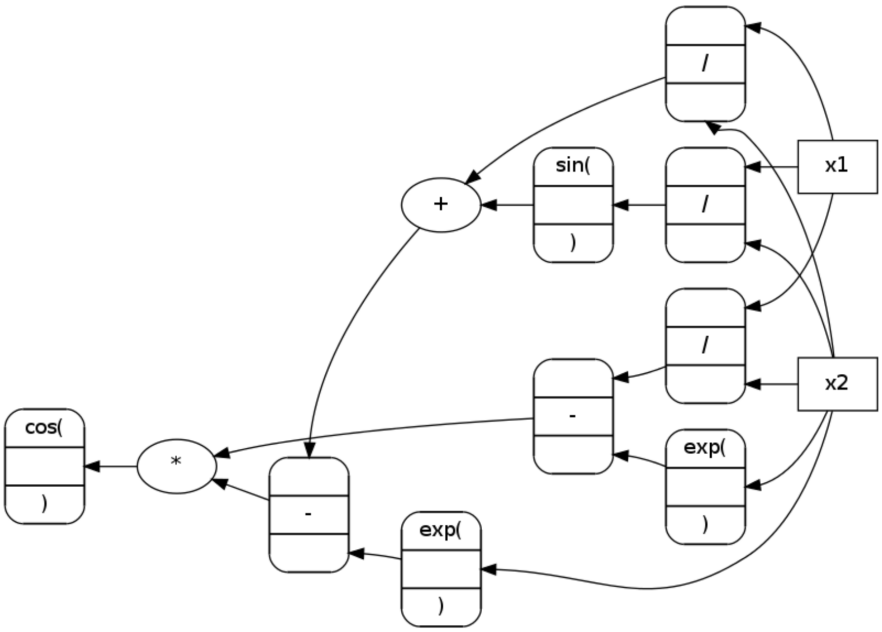

Figure 2.2: The graph of y without CSE.

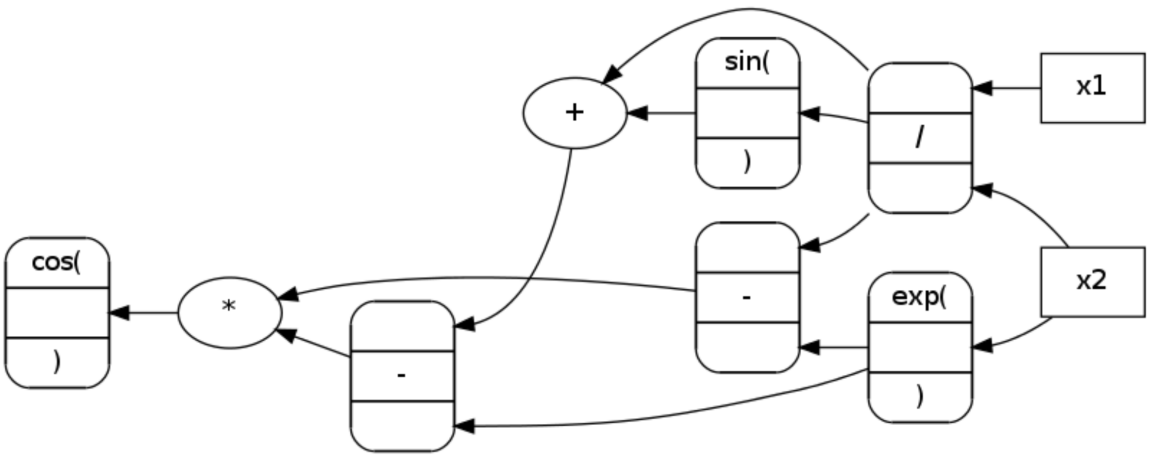

Figure 2.3: The graph of y with CSE.

Figure 2.4: Methods associated with `funName` object.

can be defined as a SXFunction object. Use the syntax `funName = SXFunction([x1,x2],[y])`. If you enter `print funName`, then you will get;

```
Inputs (2):
0. 1-by-1 (dense)
1. 1-by-1 (dense)
Output: 1-by-1 (dense)
Function not initialized
```

It says that ‘‘`Function not initialized`’’. So initialize it before going any further using the command, `funName.init()`.[2]. You may try `type(funName)` and verify that `funName` is an object the class `casadi.casadi.SXFunction`. To see available `funName`-object's methods, enter `funName.` and press `tab` key. See Figure 2.4.

Now try some of its (i.e. `funName`'s) functions; to get input/output expressions number of inputs/outputs, set input values, evaluate the function and get output. Use the following commands, respectively; `funName.inputExpr()`, `funName.outputExpr()`, `funName.getNumInputs()`, `funName.getNumOutput()`, `funName.setInput()`, `funName.evaluate()` and `funName.getOutput()`. See the example given below:

```
# Get input variables;
for i in range(funName.getNumInputs()):
    print funName.inputExpr(i)
# Get output expression;
for i in range(funName.getNumOutputs()):
    print funName.outputExpr(i)
# Set inputs' values;
for i in range(funName.getNumInputs()):
    funName.setInput(i**2+i,i)
# Evaluate funtion;
funName.evaluate()
# Get output values;
for i in range(funName.getNumOutputs()):
    print funName.getOutput(i)
```

[2] "The instantiation of an evaluation program for a certain set of input parameters with nomination independent and dependent variables maps it into an evaluation procedure, since then the control flow is fixed." [1]

Enter `print funName` and you will get the actual evaluation procedures in CasADi. See below:

```
Inputs (2):
0. 1-by-1 (dense)
1. 1-by-1 (dense)
Output: 1-by-1 (dense)
@0 = input[0][0];
@1 = input[1][0];
@2 = (@0/@1);
@3 = (@0/@1);
@3 = sin(@3);
@3 = (@3+@2);
@2 = exp(@1);
@3 = (@3-@2);
@0 = (@0/@1);
@1 = exp(@1);
@0 = (@0-@1);
@3 = (@3*@0);
output[0][0] = @3;
```

It is also possible to symbolically evaluate SXFunctions using `SXFunctionObj.eval()`. Enter the following commands:

```
x3=SX(x3)
x4=SX(x4)
print funName.eval([x3,x4])
```

How to find partial derivatives both symbolically and numerically? Go through the following code and the results are shown in Figure 2.5:

```
from casadi import *
x1=SX("x1")
x2=SX("x2")
y=(sin(x1/x2)+x1/x2-exp(x2))*(x1/x2-exp(x2))
funName=SXFunction([x1,x2],[y])
funName.init()
for i in range(funName.getNumInputs()):
    funGrad=funName.grad(i)
    PD=SXFunction([x1,x2],[funGrad])
    PD.init()
    print "Symbolic partial derivative w.r.t. "+"x"+\
    str(i+1)+" is "+str(PD.outputExpr(0))
    PD.setInput(1,0)
    PD.setInput(2,1)
    PD.evaluate()
    print "Numerica partial derivative w.r.t. "+"x"+\
    str(i+1)+" is "+str(PD.getOutput())
    print "============================="
```

`funName` is a multiple-scalar-input and single-scalar-output function. We can try a multiple-scalar-input and multiple-scalar-output example now. See below:

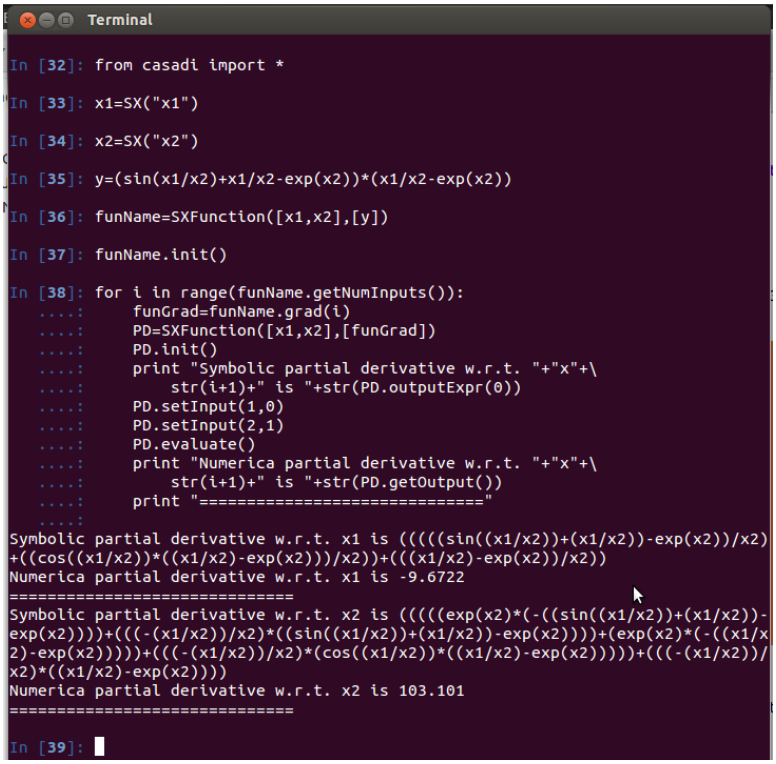

Figure 2.5: How to find partial derivatives of SXFunction's objects.

```
from casadi.casadi import *
x1=SX("x1")
x2=SX("x2")
y2=[x1*x2,x1+x2];
funName2=SXFunction([x1,x2],y2)
funName2.init()
for i in range(funName2.getNumInputs()):
    for j in range(funName2.getNumOutputs()):
        print funName2.grad(i,j)
```

It is also possible to create matrices/vectors with their elements as SX objects using `ssym` (this creates an SXMatrix as pointed out earlier). Reformulate the same function discussed just above, with input variable as an SXMatrix:

```
from casadi import *
x=ssym("x",2,1)
y3=(sin(x[0]/x[1])+x[0]/x[1]-exp(x[1]))*(x[0]/x[1]-exp(x[1]))
funName3=SXFunction([x],[y3])
funName3.init()
funName3.setInput([1,1])
funName3.evaluate()
print funName3.getOutput()
```

Consider an example: solve $\frac{dx}{dt} = f(t, x, u)$ for a given initial condition x_0 using Forward Euler method for $t \in [t_0, t_f]$. Let x and u be scalar variables. Divide the time span into N intervals, hence $dt = \frac{(t_f - t_0)}{N}$. For simplicity, take $f(t, x, u) = ax + u$ where $a < 0$. Look at the code given below:

```
from casadi import *
from casadi.tools import *
#
x0 = 1.0
t0 = 0.0
tf = 1.0
N = 2
dt = (tf -t0)/N
#
xk = ssym("xk") # or SX("xk") or ssym("xk",1,1)
uk = ssym("uk")
tk = ssym("tk")
#
a = -1.0
#
xki = x0
tki = t0
#
for i in range(N):
    tki = tki + dt
    xki = xki + dt*(a*xki+uk)
print xki
```

`xki` gives a symbolic expression. As `x0` is a floating point number, `xki` is a function of `uk` only (note that I have kept `uk` as a constant for $t \in [t_0, t_f]$.). Now we can create an SXFunction keeping `xki` as the dependent variable and `uk` as the independent variable. The syntax is: `giveFunName = SXFunction([uk],[xki])`. By slightly modifying the last code, we can create two numeric arrays to store both time and state data. See the code below (and see Figure 2.6 for the results):

```
from casadi import *
from casadi.tools import *
import numpy as np
import matplotlib.pyplot as plt
#
x0 = 10.0
t0 = 0.0
tf = 10.0
N = 100
t = np.linspace(t0,tf,N+1)
dt = t[1] - t[0]
#
xk = ssym("xk") # or SX("xk") or ssym("xk",1,1)
uk = ssym("uk",N,1)
#
a = -1.0
#
xki = x0
#
X = x0*np.ones(N+1)
U = np.random.rand(N)
#
```

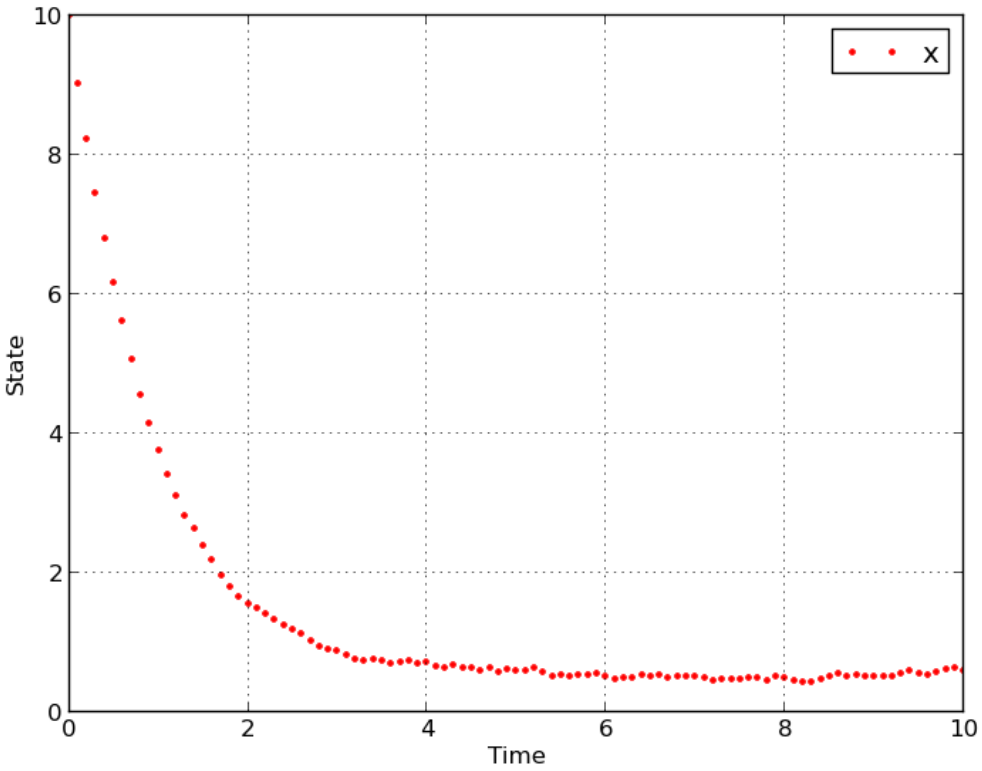

Figure 2.6: Forward Euler solution to Example 1.

```
for i in range(N):
    xki = xki + dt*(a*xki + uk[i])
    fun = SXFunction([uk],[xki])
    fun.init()
    fun.setInput(U)
    fun.evaluate()
    X[i+1] = fun.getOutput()
#
plt.plot(t,X,"r.")
plt.xlabel('Time')
plt.ylabel('State')
plt.grid('on')
plt.legend('x')
plt.show()
```

2.3 MX and MXFunction

We have become familiar with the SX, SXMatrix, SXFunction classes which are used to generate scalar symbolic variables/expressions, and thereby to create symbolic functions of the type SXFunction based on symbolic expressions already created as SX/SXMatrix instances. Any symbolic expression of SX/SXMatrix types associates with 'a graph of SX nodes'[3]. In order to build expressions, a set unary (ex. `()` - cast operator, `+=` - increment operator, etc.) and binary operators (ex. `!=` - inequality, `*` - multiplication, etc.) are used. Example: `x=SX(''x'');y=x*x+1;z=y<(-x)` produces two symbolic expressions, `y` and `z` based on the input node `x` using three binary operators `*`, `+` and `<` also `()` and `-` which are unary operators. For a given symbolic expression of SX objects, say `y`, we can use `countNodes(y)` to get the number of nodes in the graph and `dotdraw(y)` to visually inspect it. Remember to import casadi.tools (use the command `from import casadi.tools import *`) before using `countNodes(y)` and `dotdraw(y)`.

Previously it was pointed out that how to use the SXMatrix instances instead of SX. Example: when we have to define $f = (x, u)$, which is the right hand side of an ordinary differential equation $\frac{dx}{dt} = f(x, u)$, it is convenient to define the state x as `x=ssym('x',n,1)` and access state variables x_n in Python via `x[n-1]` (similarly, for t and u, we have `t=ssym('t',1,1)` and `u=ssym('u',m,1)`.) This is much more handy, rather than defining SX instances as `x1=SX('x');x2=SX('x2''); ...; xn=SX('xn'`. Don't confuse this with that in `varName = SX('displayName')` or `varName = ssym('displayName',m,n)`, it is not necessary `varName` and `displayName` are the same. Example: if `x1=SX('state1')` and `x2=SX('state2')` then `y=x1+x2` is a correct expression while `y=state1+state2` is not. On the other hand, if `x1=SX('x1')` and `x2=SX('x1')` then `y=x1+x2` is syntactically correct. Anyway this sort of practice should be avoided because symbolic expressions are displayed with the display names of symbolic variables involving but not with variable names. Also remember to insert input/output arguments as Python type list in `funName = SXFunction([input list],[output list])`. Inserting input/output arguments as `casadi.casadi.IOSchemeVectorSXMatrix` type, using helper functions is also a possibility, which would come in Chapter 3. Once a symbolic expression of SX/SXMatrix instances is given, we can construct an SXFunction as we have done many times so far. We may also convert an MXFunction object into a SXFunction object using the command: `sxfunctionObject = SXFunction(mxfunctionObject)`. However the converse, `mxfunctionObject = MXFunction(sxfunctionObject)` is not possible![3]

[3]Actually, this makes sense. Consider a matrix expression, $Y = f(X) = X^T.X$ where $X \in \mathbb{R}^{n\times 1}$. X being a scalar is a special case. Hence, if we create an MXFunction for `f`, then it could be converted into SXFunction.

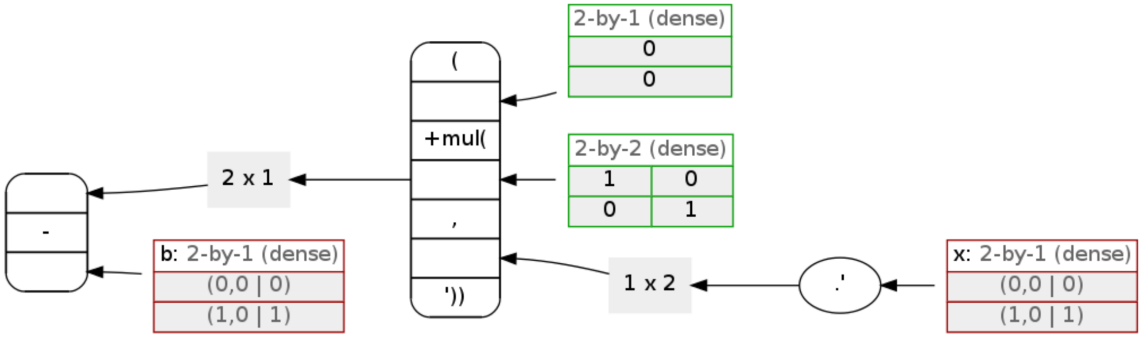

Figure 2.7: Evaluation procedures of e.

Expressions that are based on SX/SXMatrix symbolics are limited to scalar operations. So as to facilitate matrix operations, MX symbolics are used. The syntax to generate MX objects is: `x=MX('x',m,n)` or `x=msym('x',m,n)`. After defining MX objects, we can create matrix expressions. A simple example: let $e = Ax - b$, where $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^{n \times 1}$ and $b \in \mathbb{R}^{n \times 1}$. Find a symbolic expression for e. See the code below;

```
from casadi import *
from casadi.tools import *
import pydot
import numpy as np
n = 2
x=msym("x",n,1)
A = DMatrix(np.eye(n))
b = msym("b",n,1)
e = mul(A,x) - b
dotdraw(e)
```

A couple of comments: the DMatrix class is for creating matrices with elements that are floating point numbers. Here the `numpy` package is used to create A using the command `A = DMatrix(np.eye(n))`, but there are other ways of creating such matrices. Refer to [4] (available at `http://casadi.sourceforge.net/users_guide/casadi-users_guide.pdf`) for more details. `A*x` gives an error, but `mul(A,x)` is fine. `*` operator in CasADi is identical with `.*` in MATLAB. In order to get matrix multiplication in CasADi, use the CasADi function `mul()`. You may check the graph of `e` using `dotdraw(e)`, as we did for SX symbolics. See Fig. 2.7. In the graph, notice that there are 2 input nodes which are MX objects (for x and b in red color), and one DMatrix object for A (in green). Also the output node is again an MX object and there is one intermediate MX node for Ax.

2.3.1 Solve $\frac{dx}{dt} = f(t, x, u)$ for a given initial condition x_0 using a user defined integrator function for $t \in [t_0, t_f]$

In this case, the integrator is based on the forward Euler method. We use SXFunction's `call()` method with its inputs being MX types. See the code given below (for further details refer [2]):

```
from casadi import *
import numpy as np
import matplotlib.pyplot as plt
# Define f
x = ssym("x",3,1)
u = ssym("u",1,1)
f = [(1-x[1]*x[1])*x[0]-x[1]+u, x[0], x[0]*x[0]+x[1]*x[1]+u*u]
#
t0 = 0.0
tf = 10.0
x0 = [0.0,1.0,0.0]
n = len(x0)
#
N = 100
t = np.linspace(t0,tf,N+1)
dt = t[1] - t[0]
# Create SXFunction
fcn = SXFunction([x,u],f)
fcn.init()
#
xStart = msym("xStart",3,1)
uStart = msym("uStart",1,1)
#
funList = list()
#
for i in range(n):
    fcnStart = fcn.call([xStart,uStart])[i]
    xEnd = xStart[i] + dt*fcnStart
    funList.append(xEnd)
# Define integrator
integrator = MXFunction([xStart,uStart],funList)
integrator.init()
# Allocation of storage for states and define input vectors
X = np.ones((N+1,3))
X[0,:] = x0
U = np.random.rand(N,1)
#
xki = x0
# Integrate
for i in range(N):
    integrator.setInput(xki,0)
    integrator.setInput(U[i],1)
    integrator.evaluate()
    for j in range(n):
        X[i+1,j] = integrator.getOutput(j)
        xki = X[i+1,:]
```

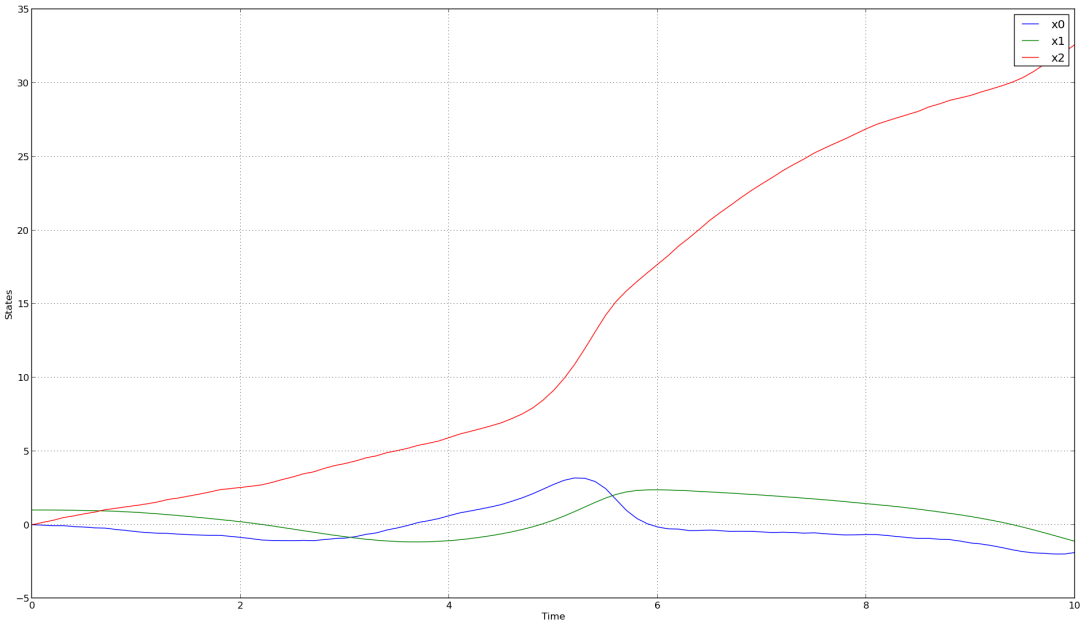

Figure 2.8: The Forward Euler solution to Example 2.

```
# Plot results
legendList = list()
plt.close()
plt.figure(0)
for i in range(n):
    plt.plot(t,X[:,i])
    plt.hold('True')
    legendList.append('x'+str(i))
plt.xlabel('Time')
plt.ylabel('States')
plt.grid('on')
plt.legend(legendList)
plt.show()
```

See Figure 2.8 for the results. What is new in this code, compared to the example given in Section 2.2, is that here we have defined an integrator function (an SXFunction object) based on the forward Euler method which has `xStart` and `uStart` as input arguments and `funList` as the output argument which is the right hand side of the ODE given. Evaluating the integrator function just defined, using `integrator.evaluate()`, after setting inputs via `integrator.setInput()`, will calculate the state at the next sample time instance. Note that `fcn.call([xStart,uStart])` gives a tuple consisting `casadi.casadi.MX` objects (and `len(fcn.call([xStart,uStart])==len(x0)` is equal to `true`). I have used a for-loop to unpack the output of `fcn.call([xStart,uStart])`.

To consult documentation of the SXFunction class, enter `help(SXFunctonObject)`. You will see that under '`List of available options`', there are set of options which can be set using `SXFunctonObject.setOption()`. Example: create a SXFunction object named `sxfunctionName` and enter `sxfunctionName.setOption("name", "MyFunctionName")`. Now the `"name"` prop-

erty of `sxfunctionName` object is set to `"MyFunctionName"`. Verify this fact using the command `sxfunctionName.getOption("name")`. In order to have a complete list of all available options, enter `sxfunctionName.getOptionNames()`. You will get:

```
'ad_mode'
'gather_stats'
'inputs_check',
'jacobian_generator'
'just_in_time'
'just_in_time_opencl'
'just_in_time_sparsity'
'live_variables'
'max_number_of_adj_dir'
'max_number_of_fwd_dir'
'monitor'
'name'
'number_of_adj_dir'
'number_of_fwd_dir'
'numeric_hessian'
'numeric_jacobian'
'regularity_check'
'sparse'
'sparsity_generator'
'store_jacobians'
'topological_sorting'
'user_data'
'verbose'
```

If you want to get a brief description about any of these SXFunction's options, use the command `solver.getOptionDescription(propery-name. Ex.'ad_mode')`. Example: the result of `solver.getOptionDescription('ad_mode')` will be: `'How to calculate the Jacobians'`. The FX class and its derived classes or sub-classes (ex. SXFuncion, etc.) share some common options. `help(FX)` gives this common list of options as FX is the base class. Sub-classes of FX may have options specific to them. Example: compare `help(FX)` and `help(IpoptSolver)`, and you will see that there are additional options available in the IpoptSolver class.

2.3.2 A simple discrete state space model

Consider a discrete state space model, $x_{k+1} = Ax_k + Bu_k$ and x_0, the initial condition is given. Find (1) a symbolic expression for x_k after N sample intervals, (2) define an MXFunction and (3) evaluate it. The code is given below, also See Figure 2.9;

```
from casadi import *
from casadi.tools import *
import pydot
import numpy as np
#
x0 = [1.0,0.0]
n = len(x0)
m = 2 # number of inputs
#
N = 2
#
```

```
# Define input. transpose(u) = [u(0),u(1),...,u(N-1)]
u = msym("u",N,m)
#
# Define A and B matrices
A = DMatrix(np.eye(n))
B = DMatrix(np.eye(n,m))
#
x = x0
#
for i in range(N):
    x = mul(A,x) + mul(B,trans(u[i,:]))
# Graph of x
dotdraw(x)
#
# Create a MXFunction
mfun = MXFunction([u],[x])
mfun.init()
#
# Set inputs
u_ = np.ones((N,m))
mfun.setInput(u_)
#
# Evaluate
mfun.evaluate()
#
# Get output
print mfun.getOutput()
```

The syntax of defining MXFunctions is more or less the same for SXFuntions. The only difference is that instead of SX/SXMatrix lists, two lists of MX objects are used for the MXFunction's input/output arguments. The syntax is: `mxfunName = MXFunctioin([list of MX objects],[list of MX objects])`. Function `mxfunName.call()` is often a useful tool. The syntax is: `[f]=mxfunName.call([list of input arguments])`. See the example given below. `[ffun] = fun.call([XX])` generates an expression (which is `ffun`) in MX symbolics. Noice that `fun.call([XX])` gives a tuple type, so it has to be unpacked so that we will avoid type incompatibilities in the expression `YY = XX + ffun`. Unpacking is done by `[ffun] = fun.call([XX])[index]` or `[ffun1,ffun2,...] = fun.call([XX])`. Note in this particular case, `fun.call([XX])` is a tuple with single element, hence merely `[ffun] = fun.call([XX])` is enough (i. e. no need to specify the `index`). `XX` and `YY` may be used to define a new function, which is `fun2` in this case. Note: the `call()` function comes in SXFunction class too. We could use `sxfunctionName.call(lsit of MX objects)`. Additionally, an MXFunction object can be mapped into an SXFunction object by `sxfunName = SXFunction(mxfunction)`.

```
from casadi import *
from casadi.tools import *
import pydot
import numpy as np
X = MX("x",2,2)
Y = sin(X) + mul(X,X)
fun = MXFunction([X],[Y])
fun.init()
XX = msym("XX",2,2)
```

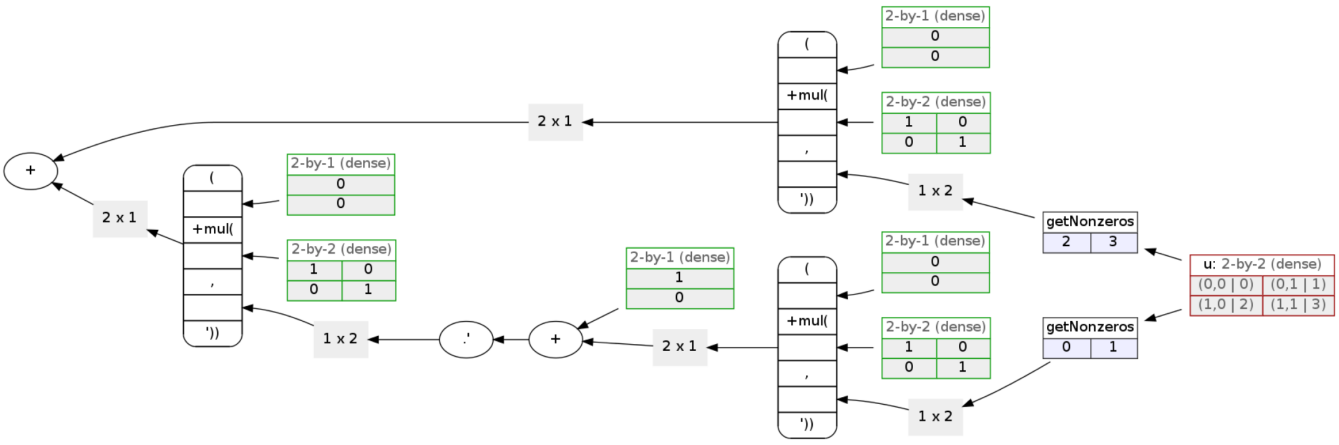

Figure 2.9: The evaluation procedure of `mfun`.

```
[ffun] = fun.call([XX])
YY = XX + ffun
fun2 = MXFunction([XX],[YY])
fun2.init()
```

2.3.3 A simple optimal control problem

As a motivating example (inspired by [2]), I will discuss example 4.2 given in [5]. A nonlinear dynamic model is given below:

$$\begin{aligned}\dot{x}_1 &= -k_1 x_1 - k_3 x_1^2 + (v - x_1)u \\ \dot{x}_2 &= k_1 x_1 - k_2 x_2 - x_2 u \\ y &= x_2\end{aligned}$$

Define a cost function, say, $I = (y_f - r_f)^2$. $t \in [t_0, t_f]$ and $y_f = y\,(t = t_f)$. t_0 and t_f are initial and final time instances. The objective is to find a piece-wise constant control signal such that I is minimum. We can divide the time span into N intervals (then $dt = \frac{t_f - t_0}{N}$) and $u(t) \triangleq u_k$, $t \in (dt \cdot k, dt \cdot (k+1)]$ and $k = 0, 1, \ldots, N-1$. Once x_0 is given, it is possible to find a symbolic expression for $x_f == x\,(t = t_f)$, thereby y_f with respect to piece-wise constant input signals u_k. For simplicity assume that the reference signal r is constant. Hence, $r = r_f, \forall t \in [t_0, t_f]$. Consequently, we have I as a nonlinear function of u_k. Also assume that the disturbance signal v is a constant. For given u_k's (for $k = 0, 1, \ldots, N-1$), we symbolically find x_k's and y_k's (for $k = 1, 2, \ldots, N$.) To do this we need an integrator. Two options for doing this: (1) use built-in integrators already interfaced into CasADi (ex. CVodesIntegrator, IdasIntegrator, etc.), or (2) use a user define integrator (ex. RK method, etc.) If you use a user defined integrator, some care must to be made about the performance of it. Example: step time, dt should be small enough and hence should be carefully selected. Chapter 3 will explain how to use built-in integrators. If we write the optimization problem, it would be: minimize $I(U)$ such that

$[u_{min}, u_{min}, \ldots, u_{min}]^T \leq U \leq [u_{max}, u_{max}, \ldots, u_{max}]^T$ and $U = [u_0, u_1, \ldots, u_{N-1}]^T$. I will use a simple explicit ODE solver based on a Runge Kutta method. See the code below and the results are in Figure 2.10 and Figure 2.11.

```
from casadi import *
from casadi.tools import *
import numpy as np
import matplotlib.pyplot as plt
#
k1 = 50.
k2 = 100.
k3 = 10.
rf = 0.0
#
t0 = 0.0
tf = 1.0
N = 300
dt = (tf-t0)/N
#
x0 = [2.5,1.0]
n = len(x0)
u0 = 25.0
v0 = 10.0
#
u_max = 30.*np.ones((N,1))
u_min = 20.*np.ones((N,1))
#
x = ssym("x",n,1)
u = ssym("u")
v = v0
#
dxdt = vertcat([-k1*x[0]-k3*x[0]**2+(v-x[0])*u,k1*x[0]-k2*x[1]-x[1]*u])
#
fun_dxdt = SXFunction([x,u],[dxdt])
fun_dxdt.init()
#
U = msym("U",N,1)
# Define a simple RK integrator
xk = msym("xk",n,1)
uk = msym("uk")
[a1] = fun_dxdt.call([xk,uk])
[a2] = fun_dxdt.call([xk+(dt/2)*a1,uk])
[a3] = fun_dxdt.call([xk+(dt/2)*a2,uk])
[a4] = fun_dxdt.call([xk+dt*a3,uk])
xkj = xk + (dt/6)*(a1+2*a2+2*a3+a4)
RK_integrator = MXFunction([xk,uk],[xkj])
RK_integrator.init()
# Find xf
xf = x0
for j in range(N):
    [xf] = RK_integrator.call([xf,U[j]])
```

```
# Find yf
yf = xf[1]
# Find I
I = (yf-rf)**2
# Create a MXFunction object using helper function for
# nonlinear programming. I.e. nlpIn() and nlpOut()
fun_nlp = MXFunction(nlpIn(x=U),nlpOut(f=I))
# Create a solver object based on IpoptSolver class
solver = IpoptSolver(fun_nlp)
solver.init()
# Set constraints
solver.setInput(u_max,"ubx")
solver.setInput(u_min,"lbx")
# Solve
solver.solve()
# Get results
U_ = solver.getOutput()
U_opt = U_.toArray().squeeze().tolist()
#
t = np.linspace(t0,tf,N+1).tolist()
U_opt.append(U_opt[-1])
#
yf_opt = list()
yf_opt.append(x0[1])
xf_ = x0
xf = list([x0])
for k in range(N):
RK_integrator.setInput(xf_,0)
RK_integrator.setInput(U_opt[k],1)
RK_integrator.evaluate()
xf_ = RK_integrator.getOutput()
xf_ = xf_.toArray().squeeze().tolist()
xf.append(xf_)
xf = np.array(xf)
# Plot results
plt.close()
plt.figure(0)
plt.plot(t,xf[:,0],'.',t,xf[:,1],'.')
plt.grid('on')
plt.xlabel('Time')
plt.ylabel('x1,x2')
plt.legend(('x1','x2'))
plt.show()
plt.figure(1)
plt.plot(t,U_opt,'.')
plt.grid('on')
plt.xlabel('Time')
plt.ylabel('u')
plt.legend('u')
plt.show()
```

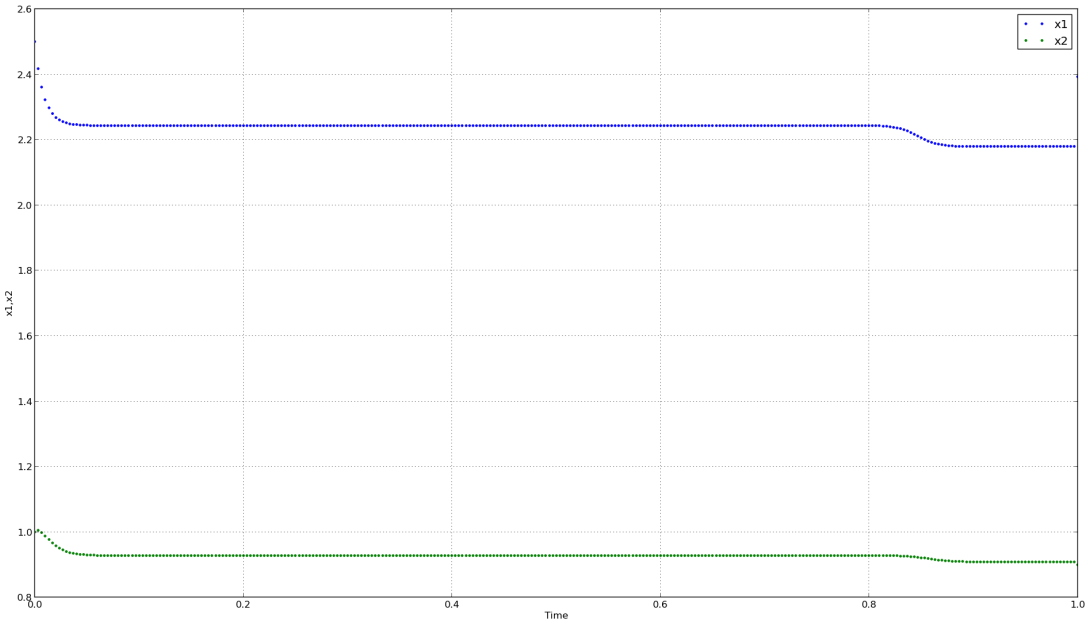

Figure 2.10: A solution to the optimal control problem - Example 4 - states.

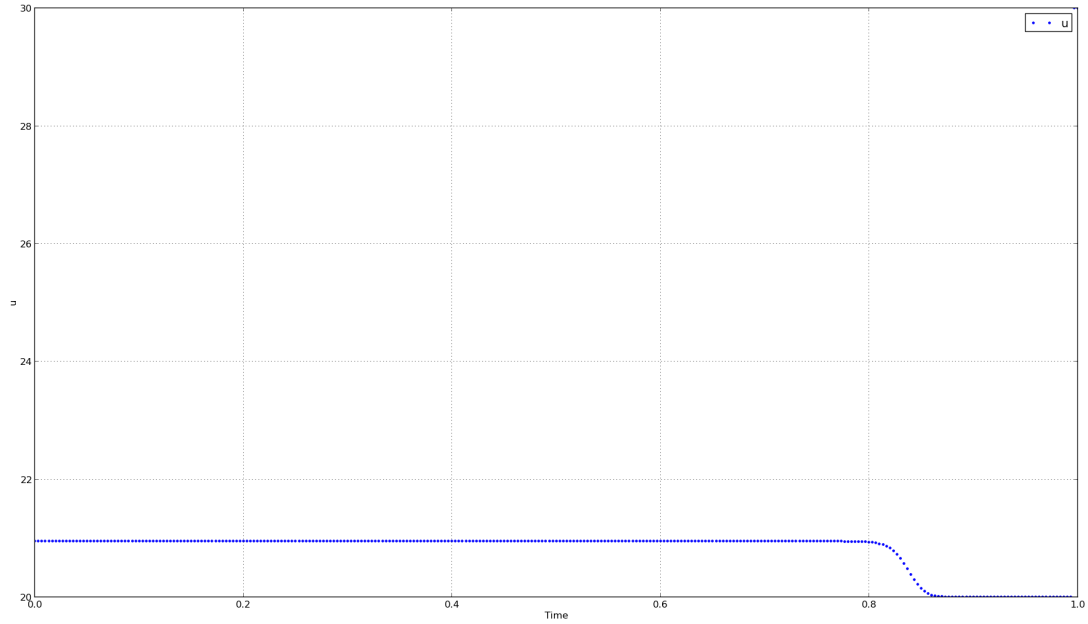

Figure 2.11: A solution to the optimal control problem - Example 4 - input signal.

Here we used Ipopt solver. IPOPT (Interior Point Optimizer) is an open source software package which solves nonlinear optimization problems of the form:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i^L \leq g_i(x) \leq g_i^U; i = 1, 2, \ldots, m. \\ & x^L \leq x \leq x^U \end{aligned}$$

f is a scalar valued function and $g_i(x) \in \mathbb{R}^m$. f and g_i are twice differentiable. To handle any equality constraint $g_j(x) = 0$, set $g_j^L = g_j^U$. IPOPT is a local optimizer. For a detailed discussion refer to [13][12] [14]. There are many options that we can set before solving nonlinear problems. To get the names of available options and their respective descriptions use `solver.getOptionNames()` and `solver.getOptionDescription(''OptionName'')`. To set an option use `solver.setOption(''OptionName'',''OptionValue'')`. Casadi API documentation, example: `help(solver)` diverts us to Ipopt documentation available at `http://www.coin-or.org/Ipopt/documentation/`. You may also try `solver.printOptions()` to get options with more details. To get `''OptionValue''`, you should refer `ipopt`'s `''Options Reference''` section in the URL given above or try the link `http://www.coin-or.org/Ipopt/documentation/node39.html`. Ex. to know the alternatives for `''OptionValue''` for the option `''nlp_scaling_method''` go to `''NLP Scaling''` section in the last URL mentioned and click on `''nlp_scaling_method''`. Then you will see there are 4 alternatives available: `''none''`, `''user-scaling''`, `''gradient-based''`, and `''equilibration-based''`.

Chapter 3

FX Derived Functions

3.1 An Overview

In Chapter 2, creating SX/MXFunction classes' instances using symbolic expressions is discussed. Let us step forward with other functionalities of FX class (integrators, nonlinear solvers, etc.). Create a sample SXFunction instance, `f` and enter `f.printOptions()` in the command line. This will give set options that relate to SXFunction objects which we can set via `f.setOptions()`. Example: we can set function's the name property, automatic differentiation mode, etc. via `f.setOption(name,myfun)`, `f.setOption(ad_mode,forward)`, and so on. This is something we discussed already at the end of Chapter2. See the code below:

```
x = SX("x")
y = sin(x)
f = SXFunction([x],[y])
f.setOption("name","myfunc")
f.setOption("ad_mode","forward")
f.getOption("name")
f.getOption("ad_mode")
```

Syntax for creating SX/MXFunctions are `funName = SXFunction([list of inputs],[list of outputs])` and `funName = MXFunction([list of inputs],[list of outputs])`. Upcoming sub-sections will use a slightly different syntax. The IPOPT solver and two DAEs integrators will be discussed. A given optimization problem is formulated as either using SXFunction or MXFunction, the choice depends on the problem context. For simplicity I will brake down usage of FX derived classes into 3 parts; (1) define the problem description symbolically (defining the cost function and constraints), (2) create an SX/MXFunction (to be used as an input argument to a solver) using relevant helper functions to create its input and output arguments, (3) select a suitable solver and solve the problem. What is a "helper function"? If the problem is, among other choices like linear optimization, quadratic optimization, etc., for example a nonlinear optimization problem then the corresponding SX/MXFuction, created in step (2), should reflect the characteristics of the nonlinear optimization problem. An example is given to elaborate this:

```
x = SX("x")
y = x*x + x
fun = SXFunction([x],[y])
fun.init()
```

`fun` is just a function based on the symbolic expression `y` based on `x`. Say, we want to minimize `y` (this is quadratic) and use a built-in solver (ex. the IpoptSolver class). IpoptSolver, like other solvers, accepts `fun` as an input argument, i. e. `solver=IpoptSolver(fun)`.

But this will not work! `fun` must be compatible with what the `IpoptSolver` input argument demands. When we define `fun`, we should have used two helper functions `nlpIn()` and `nlpOut()` which goes with nonlinear solvers, to define input/output arguments to `fun`. Type `?nlpIn` and `?nlpOut` to access the documentation. So the correct way to define the function, `fun` is `fun = SXFunction(nlpIn(nlpIn_arguments),nlpOut(nlpOut_arguments))` and `solver = IpoptSolver(fun)`. Here `nlpIn_arguments` and `nlpOut_arguments` are taken from the documentation of `IpoptSolver` (enter `help(IpoptSolver)` and read through 'Input scheme' and 'Output scheme'. So always remember to define the SX/MXFunction function compatible with the solver requirements. The helper function's output is in the type of `casadi.casadi.IOSchemeVectorSXMatrix`. See below for a list of available helper functions for various solvers.[1] One more example: if you want to solve a quadratic programming problem using `QPSolver`, then the corresponding helper functions are `qpIn()` and `qpOut()`.

```
ControlSimulator --> controlsimulatorIn, controlsimulatorOut
To define DAEs to be used with integrators --> DAEInput, DAEOutput
DPLE --> not available in my installation.
GradF --> gradFIn, gradFOut
HessLag --> hessLagIn, hessLagOut
Integrator --> integratorIn, integratorOut
JacG --> jacGIn, jacGOut
LPSolver --> lpIn, lpOut
Linsol --> linsolIn, linsolOut
Mayer --> mayerIn
NLP --> nlpIn, nlpOut
NLPSolver --> nlpSolverIn, nlpSolverOut
OCP --> ocpIn, ocpOut
QCQPSolver --> qcqpIn, qcqpOut
QPSolver --> qpIn, qpOut
RDAE --> rdaeIn, rdaeOut
SDP --> sdpIn, sdpOut
SDQP --> sdqpIn, sdqpOut
SOCP --> socpIn, socpOut
StabilizedQPSolver --> not available in my installation.
```

3.2 Nonlinear Programming

Several examples are given how to solve nonlinear optimization problems using the NLPSolver class. There are 5 subclasses of the NLPSolver class, namely: IpoptSolver, KnitroSolver, SCPgen, SnoptSolver and SQPMethod. Only IpoptSolver will be discussed because more or less the same procedure will apply for other solvers.

3.2.1 The Rosenbrock's Function

The problem is taken from page 1-3 of [6]. The problem description is:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100 \cdot \left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 \\ \text{subject to} \quad & x_1^2 + x_2^2 - 1 \leq 0 \end{aligned}$$

Define $x \triangleq [x_1, x_2]^T$. Take the initial guess, $x_0 = [0, 0]^T$. I will recap the step to be followed: (1) symbolically define cost and constraints functions, (2) define an SX/MXFunction object using

[1] Thanks goes to Joris Gillis, he mentioned this list answering one of my questions @CasADi FAQs on January 22, 2014.

`nlpIn` and `nlpOut` helper functions, and (3) use `IpoptSolver` to solve the problem. We can handle this problem in three different ways: method 1 – using SX and SXFunction, method 2 – using SXMatrix and SXFunction, and method 3 – using MX and MXFunction. See the codes given below.

Method 1: the answer is `[0.786415,0.617698]`.

```
from casadi import *
x1 = SX("x1")
x2 = SX("x2")
# Define cost function
f = 100*(x2-x1**2)**2 + (1-x1)**2
# Define constraint
g = x1**2+x2**2-1
# Define a SXFunction to be used with IpoptSolver
nlp = SXFunction(nlpIn(x=SXMatrix([x1,x2])),nlpOut(f=f,g=g))
nlp.init()
# Define a IpoptSolver object
solver = IpoptSolver(nlp)
solver.init()
# Set inputs and solve
solver.setInput([0.0],"ubg")
solver.solve()
# Print solution
print solver.getOutput("x")
```

Method 2: the answer is `[0.786415,0.617698]`.

```
from casadi import *
x = ssym("x",2,1)
# Define cost function
f = 100*(x[1]-x[0]**2)**2 + (1-x[0])**2
# Define constraint
g = x[0]**2+x[1]**2-1
# Define a SXFunction to be used with IpoptSolver
nlp = SXFunction(nlpIn(x=x),nlpOut(f=f,g=g))
nlp.init()
# Define a IpoptSolver object
solver = IpoptSolver(nlp)
solver.init()
# Set inputs and solve
solver.setInput([0.0],"ubg")
solver.solve()
# Print solution
print solver.getOutput("x")
```

Method 3: the answer is `[0.786415,0.617698]`.

```
from casadi import *
x = msym("x",2,1)
# Define cost function
f = 100*(x[1]-x[0]**2)**2 + (1-x[0])**2
# Define constraint
```

```
g = x[0]**2+x[1]**2-1
# Define a MXFunction to be used with IpoptSolver
nlp = MXFunction(nlpIn(x=x),nlpOut(f=f,g=g))
nlp.init()
# Define a IpoptSolver object
solver = IpoptSolver(nlp)
solver.init()
# Set inputs and solve
solver.setInput([0.0],"ubg")
solver.solve()
# Print solution
print solver.getOutput("x")
```

3.2.2 The problem given in page 6-50 of [7]

Just try the code given below. The answer is `[-9.64096,1.14096]`.

```
from casadi import *
x = ssym("x",2,1)
# Define cost function
f = exp(x[0])*(4*x[0]**2+2*x[1]**2+4*x[0]*x[1]+2*x[1]+1)
# Define constraint
g = vertcat([x[0]*x[1]-x[0]-x[1]+1.5,-x[0]*x[1]-10])
# Define a MXFunction to be used with IpoptSolver
nlp = SXFunction(nlpIn(x=x),nlpOut(f=f,g=g))
nlp.init()
# Define a IpoptSolver object
solver = IpoptSolver(nlp)
solver.init()
# Set inputs and solve
solver.setInput([-1.0,1.0],"ubg")
solver.solve()
# Print solution
print solver.getOutput("x")
```

3.3 Integration of DAEs/ODEs

3.3.1 Solve the system of ODEs given in page 10-16 of [7] using CVodesIntegrator

Solve the following systems of ODEs using the built-in integrator CVodesIntegrator;

$$\begin{aligned}\dot{y}_1 &= y_2 y_3 \\ \dot{y}_2 &= -y_1 y_3 \\ \dot{y}_3 &= -0.51 y_1 y_3\end{aligned}$$

with the initial condition $y_0 = [0, 1, 1]^T$ for $t \in [0, 12]$. See the code given below. The results are in Figure 3.1. The necessary helper functions for `fun` and creating an integrator object using `CVodesIntegrator` are quite clear by now. Use `integrator.getOptionNames()` to get available options that we may set. Look at the two options `"t0"` and `"tf"`. Set them into 0 and 12 respectively. Once we have set these two parameters, the `integrator` can be used to integrate for $t \in [t_1, t_2]$ such that $t_0 < t_1 \leq t_2 \leq t_f$.

```
from casadi import *
from casadi.tools import *
import numpy as np
import matplotlib.pyplot as plt
#
y = msym("y",3,1)
dydt = vertcat([y[1]*y[2],-y[0]*y[2],-0.51*y[0]*y[1]])
fun = MXFunction(daeIn(x=y),daeOut(ode=dydt))
fun.init()
#
integrator = CVodesIntegrator(fun)
#
t0 = 0.
tf = 12.
N = 100
dt = (tf-t0)/N
x0 = [0.,1.,1.]
#
x = list()
#
integrator.setOption("t0",t0)
integrator.setOption("tf",tf)
# Always initialize after integrator.setOption(.)
integrator.init()
integrator.setInput(x0,"x0")
#
integrator.evaluate()
integrator.reset()
#
tspan = np.linspace(t0,tf,N+1)
#
for t in tspan:
    integrator.integrate(t)
    x0 = integrator.getOutput().toArray().squeeze()
    x.append(list(x0))
#
X = np.array(x)
#
plt.close()
plt.plot(tspan,X[:,0],'-',tspan,X[:,1],'-,',tspan,X[:,2],'.')
plt.grid('on')
plt.xlabel('Time')
plt.ylabel('States')
plt.legend(('y1','y2','y3'))
plt.show()
```

3.3.2 Solve the system of ODEs (the van der Pol system) given in page 10-13 of [7] using IdasIntegrator

Just run the code below. See also Figure 3.2.

```
from casadi import *
```

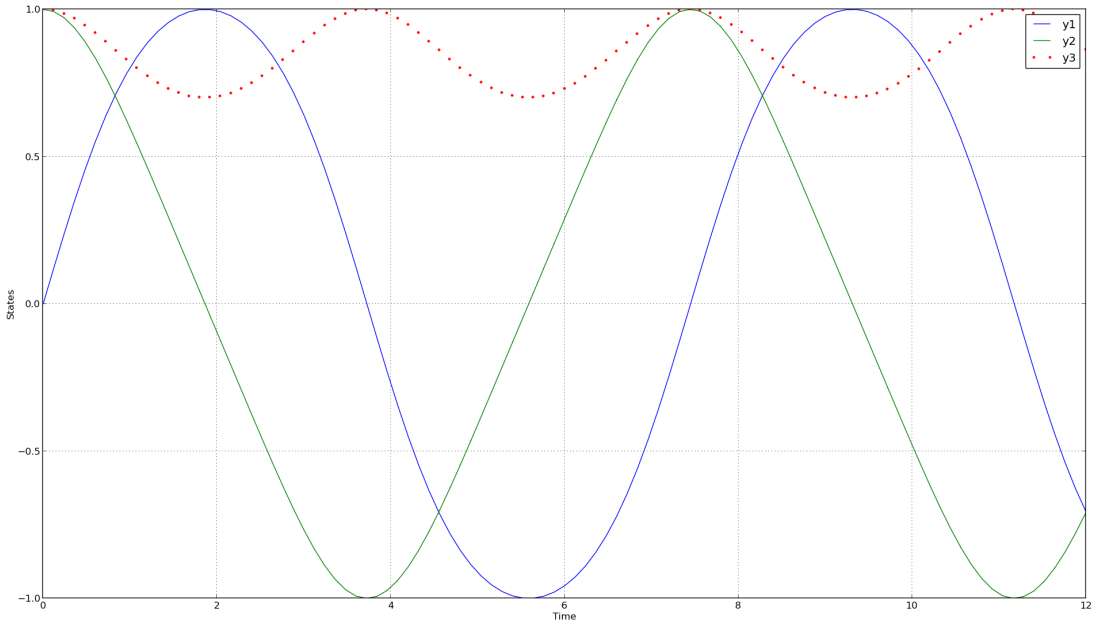

Figure 3.1: A solution to Example 7 - CVodesIntegrator.

```
from casadi.tools import *
import numpy as np
import matplotlib.pyplot as plt
#
y = msym("y",2,1)
dydt = vertcat([y[1],1000.0*(1-y[0]**2)*y[1]-y[0]])
fun = MXFunction(daeIn(x=y),daeOut(ode=dydt))
fun.init()
#
integrator = IdasIntegrator(fun)
#
t0 = 0.
tf = 3000.
N = 100
dt = (tf-t0)/N
x0 = [2.,0.]
#
x = list()
#
integrator.setOption("t0",t0)
integrator.setOption("tf",tf)
integrator.init()
#
integrator.setInput(x0,"x0")
integrator.evaluate()
integrator.reset()
#
```

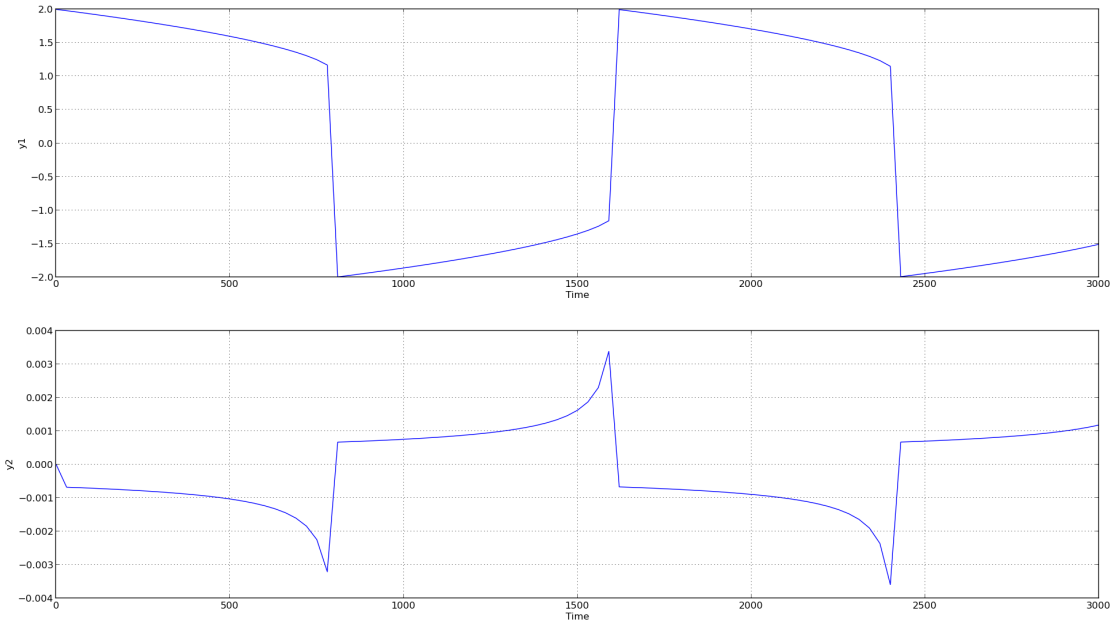

Figure 3.2: A solution to Example 8 - using IdasIntegrator.

```
tspan = np.linspace(t0,tf,N+1)
#
for t in tspan:
    integrator.integrate(t)
    x0 = integrator.getOutput().toArray().squeeze()
    x.append(list(x0))
#
X = np.array(x)
#
plt.close()
plt.figure()
plt.subplot(211)
plt.plot(tspan,X[:,0])
plt.xlabel('Time')
plt.ylabel('y1')
plt.grid('on')
plt.subplot(212)
plt.plot(tspan,X[:,1],'.')
plt.xlabel('Time')
plt.ylabel('y2')
plt.grid('on')
plt.show()
```

Chapter 4

Linearization of ODEs and Extraction of Causality of Modelica Models

4.1 Symbolic/Numeric Linearization

Consider the simple dynamic model given in Chapter 2.11. First, construct a Modelica package which may contain several Modelica models (ex. 'SimpleNonLinearModel1', 'SimpleNonLinear-Model1', etc.). The Modelica script is given below. It is saved with the file name same with the package name and the file extension should be '.mo'.

```
package MyModels
// Start SimpleNonLinearModel1
model SimpleNonLinearModel1
# Define model parameters
parameter Real k1 = 50.0;
parameter Real k2 = 100.0;
parameter Real k3 = 10.0;
# Define state variables
Real x1(start = 2.5, fixed = true);
Real x2(start = 1.0, fixed = true);
# Define input variables
input Real u;
input Real v;
equation
# Define differential equations
der(x1) = -k1*x1 -k3*x1^2 + (v-x1)*u;
der(x2) = k1*x1 - k2*x2 -x2*u;
# Define algebraic equations
y = x1;
end SimpleNonLinearModel1;
// Start SimpleNonLinearModel2
model SimpleNonLinearModel2
// To be defined
equation
// To be defined
end SimpleNonLinearModel2;
//
```

```
end MyModels;
```

For comparison purposes, analytical Jacobian matrices are derived. The state space form is written as follows:

$$\dot{x} = f(x, u, v) = [f_1(x, u, v), f_2(x, u, v)]^T$$

u and v are scalars. $x = [x_1, x_2]^T$. Define, $A \triangleq \frac{\partial f}{\partial x}$, $B \triangleq \frac{\partial f}{\partial u}$, and $L \triangleq \frac{\partial f}{\partial v}$. Then

$$A = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -(k_1 + 2k_3x_1 + u) & 0 \\ k_1 & -(k_2 + u) \end{bmatrix}$$

$$B = \begin{bmatrix} (v - x_1) \\ -x_2 \end{bmatrix}$$

$$L = \begin{bmatrix} u \\ 0 \end{bmatrix}$$

Let us find those Jacobian matrices symbolically. Run the following Python script. Note that `MyModels.mo` and the Python script files should be in the same directory. The results are given in Figure 4.1 and note that it shows identical results with analytical Jacobian matrices found above.

```
# Import casadi packages
from casadi import *
from casadi.tools import *
# Import compilers
from pymodelica import compile_fmu
from pymodelica import compile_jmu
from pymodelica import compile_fmux
# To load FMUs, JMUs, and FMUXs, use these
from pyfmi import load_fmu
from pyjmi import JMUModel
from pyjmi import CasadiModel
# Modelica moodel's details
model_name = 'JModelica.SimpleNonLinearModel1'
model_file = 'MyModels.mo'
# Compile
model_fmu = compile_fmu(model_name,model_file)
model_jmu = compile_jmu(model_name,model_file)
model_fmux = compile_fmux(model_name,model_file)
# Load models: FMUs, JMUs, FMUXs
fmuModel = load_fmu('JModelica_SimpleNonLinearModel1.fmu')
fmuxModel = JMUModel('JModelica_SimpleNonLinearModel1.jmu')
fmuxModel = CasadiModel('JModelica_SimpleNonLinearModel1.fmux')
## ===This Section Is For CasadiModel Objects Only!===
# Make fmuxModel model explicit
fmuxModel.ocp.makeExplicit()
# Access ocp
a = fmuxModel.ocp
# Get RHS of explicit ODE
rhs = a.ode
# Get state
n_x = len(a.x) # number of states
x = list()
```

```
for i in range(n_x):
    x.append(a.x[i].var())
# Get [u,v] as input
input = a.u
# Get u
n_u = 1 # number of inputs
u = list()
for j in range(n_u):
    u.append(input[j].var())
# Get v
v = list()
for k in range(n_u,len(input)):
    v.append(input[k].var())
# Create a SXFunction object
f = SXFunction([vertcat(x),vertcat(u),vertcat(v)],[rhs])
f.init()
# Find symbolic Jacobian, A
A = f.jac(0)
print "================================"
print A
# Find symbolic Jacobian, B
B = f.jac(1)
print "================================"
print B
# Find symbolic Jacobian, L
L = f.jac(2)
print "================================"
print L
## ===End Section On CasadiModel Objects ===
```

A couple of things should be explained before going further. We have had a quite detailed discussion on SX, SXMatrix, SXFunction, MXFunction, MX, FX and FX derived classes. The idea behind `from casadi import *` and `from casadi.tools import *` is obvious by now. We cannot use the function, `vertcat()` without importing `casadi.tools` for example. Note also that `vertcat()` gives an SXMatrix instance. The reason why I have used `vertcat(x)`, `vertcat(u)` and `vertcat(v)` in `f = SXFunction([vertcat(x),vertcat(u),vertcat(v)],[rhs])` is that input arguments SXFunction, should be given as a Python list of SX/SXMatrix instances. So, for example `f = SXFunction([vertcat(x),u,v],[rhs])` is wrong as `u` and `v` are of the type Python lists, but which should have been SX/SXMatrix class instances! `rhs` already is in SXMatrix type, therefore putting the output argument as `[rhs]` is correct.

There are several new concepts which need to be discussed. The JModelica.org compilers deal with code written according Modelica and Optimica standards. Optimica will not be discussed here. So first, we should have a dynamic model which has been modeled in Modelica. It is worth to mention that JModelica.org doesn't support Modelica code containing specific functions, e.g. `delay()`, etc. Python is the scripting language used with JModelica.org. JModelica.org, among other modeling and simulation tools, support FMI-Functional Mock-up Interface import and export, hence JModelica can export FMUs as well as FMUs could be imported into Python using the PyFMI (`https://pypi.python.org/pypi/PyFMI`) module. According to FMI standards, a compiler which supports FMI standards will generate, a '.xml' file containing data (ex. parameter/variable names, units, simulation start/end time, etc.) needed for program execution and several C-codes or binary which represent the mathematical model. Compiled

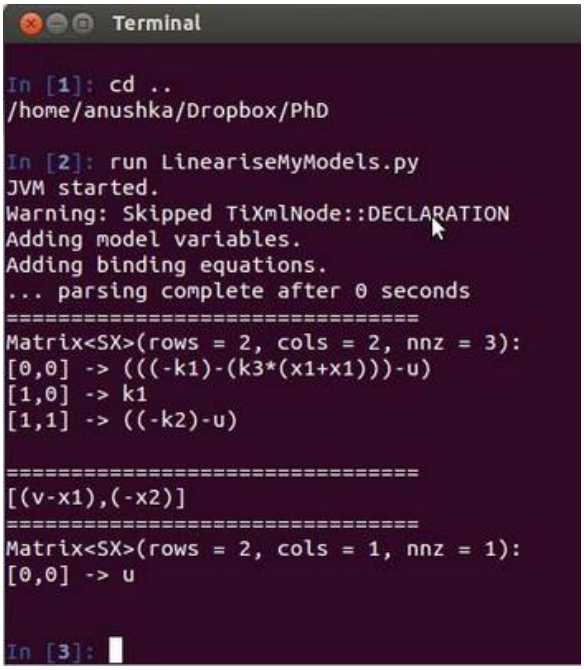

Figure 4.1: Symbolic Jacobian matrices A, B, and L.

model is stuffed in a '.zip' file (so called a FMU-Functional Mock-up Unit.) with the extension '.fmu'. Further details are available in [8]. Also look in `https://fmi-standard.org/`. We can access JModelica.org compilers via the PyModelica module. We may use these compilers (for more options see [9].): (1) `compile_fmu`, (2) `compile_jmu` and (3) `compile_fmux` and these will import FMUs (use FMUModel to import - available in pyFMI), JMUs (use JMUModel to import - available in pyJMI) and FMUXs (use CasadiModel to import - available in pyJMI). Actually, the last option, importing models as `CasadiModel` objects is at our main interest. As the name indicates, it links to CasADi. I will come to this later.

This is how we import compilers:

For FMU

```
from pymodelica import compile_fmu
```

ForJMU

```
from pymodelica import compile_jmu
```

For FMUX

```
from pymodelica import compile_fmux
```

Then to compile Modelica models and export them as FMUs, JMUs, and FMUXs the following commands are used:

For FMU export

```
model_fmu = compile_fmu(model_name, model_file)
```

For JMU export

```
model_jmu = compile_jmu(model_name, model_file)
```

For FMUX export

```
model_fmux = compile_fmux(model_name, model_file)
```

According to the example given above, `model_name = 'JModelica.SimpleNonLinearModel1'` and `model_file = 'MyModels.mo'`. In order to import we may use:

For FMU import

```
from pyfmi import load_fmu
fmuModelObject = load_fmu(enter '.fmu' file name as a string)
```

For JMU import

```
from pyjmi import JMUModel
jmuModelObject = JMUModel(enter '.jmu' file name as a string)
```

For FMUX import

```
from pyjmi import CasadiModel
casadiModelObject = CasadiModel(enter '.fmux' file name as a string)
```

Note that FMUs are only for simulation purposes while JMUs/FMUXs can be used for both simulation and optimization. In FMI standards, the model is transferred into an ODE model while in JMUs it is a Differential Algebraic Equations (DAEs). [9] stated that for simulation FMUs is better than JMUs in some aspects. But when the collocation algorithms to be used for optimization, simulating the model as DAE is a necessity, hence JMUs with DAEs integrating ability is very useful.

The goal is to linearize a given nonlinear model both numerically and symbolically. The code given above demonstrates how to evaluate Jacobian matrices symbolically. It is obvious that we can easily go from symbolic Jacobian to numeric Jacobian. We use `compile_fmux` compiler and `CasadiModel` for model import. As we import the model as a `CasadiModel` object, it comes with structural information of the system. `fmuxModel.ocp` is the one we are going use here.[1] There are some limitation of doing this, too (see Chapters 3, 6, and 12; Chapter 12 is completely about the limitations of JModelica platform, of [9]). Example: "A limitation of the algorithm is that it currently does not support record and function constructs in the Modelica code."

When Modelica models are formulated in Modelica, the identifier `'input'` is used to distinguish input variables. Example: in above case `input Real u` and `input Real v`. There is no way that we can distinguish them into control and disturbance variables, so we define an augmented input variables vector instead as $[u, v]^T$. So, the code given above is slightly modified and given as follows:

```
# Import casadi packages
from casadi import *
from casadi.tools import *
# Import compilers
from pymodelica import compile_fmu
from pymodelica import compile_jmu
from pymodelica import compile_fmux
# To load FMUs, JMUs, and FMUXs, use these
from pyfmi import load_fmu
from pyjmi import JMUModel
from pyjmi import CasadiModel
# Modelica moodel's details
model_name = 'JModelica.SimpleNonLinearModel1'
model_file = 'MyModels.mo'
```

[1] OCP stands for Optimal Control Problem.

```
# Compile
model_fmu = compile_fmu(model_name,model_file)
model_jmu = compile_jmu(model_name,model_file)
model_fmux = compile_fmux(model_name,model_file)
# Load models: FMUs, JMUs, FMUXs
fmuModel = load_fmu('JModelica_SimpleNonLinearModel1.fmu')
fmuxModel = JMUModel('JModelica_SimpleNonLinearModel1.jmu')
fmuxModel = CasadiModel('JModelica_SimpleNonLinearModel1.fmux')
## ===This Section Is For CasadiModel Objects Only!===
# Make fmuxModel model explicit
fmuxModel.ocp.makeExplicit()
# Access ocp
a = fmuxModel.ocp
# Get RHS of explicit ODE
rhs = a.ode
# Get state
n_x = len(a.x) # number of states
x = list()
for i in range(n_x):
    x.append(a.x[i].var())
# Get [u,v] as input
input = a.u
# Get u
n_u = len(input) # number of inputs,
u = list()
for j in range(n_u):
    u.append(input[j].var())
# Get free parameters
p = list()
n_p = len(fmuxModel.ocp.pi)
for k in range(n_p):
    p.append(fmuxModel.ocp.pi[k].var())
# Create a SXFunction object
f = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[rhs])
f.init()
# Find symbolic Jacobian, A
print "================================"
A = f.jac(0)
print "Jacobian of A is " + str(A)
print "================================"
# Find symbolic Jacobian, B
print "================================"
B = f.jac(1)
print "Jacobian of B is " + str(B)
print "================================"
# Find numeric Jacobians of A
## ===End Section On CasadiModel Objects ===
```

Now according to the dimension of `u` and `v` we could define sub-matrices of `B`. How to calculate numerical Jacobian? The procedure is given below;

```
A_numJac = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[A])
```

```
A_numJac.init()
A_numJac.setInput([1,0],0)
A_numJac.setInput([1,-1],1)
k1 = fmuxModel.get("k1")
k2 = fmuxModel.get("k2")
k3 = fmuxModel.get("k3")
A_numJac.setInput([k1,k2,k3],2)
A_numJac.evaluate()
print "Numerical Jacobian of A is ", str(A_numJac.getOutput())
print "====================================================="
B_numJac = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[B])
B_numJac.init()
B_numJac.setInput([1,0],0)
B_numJac.setInput([1,-1],1)
B_numJac.setInput([k1,k2,k3],2)
B_numJac.evaluate()
print "Numerical Jacobian of B is ", str(B_numJac.getOutput())
```

It is also possible to modify the CasadiModel class to obtain Jacobian matrices. We could compile Modelica models, at the moment with some restrictions, using the compiler `compile_fmux` and then import the compiled Modelica models as a `CasadiModel`. Interestingly, `CasadiModel` objects have access to `ocp` where DAEs are symbolically represented. If we can successfully compile and import a Modelica model as a `CasadiModel` object, then SymbolicOCP (i.e `ocp`) can be used to linearize the nonlinear model for any given state and input variables. In order to do this, we have to modify the CasadiModel class. Go to the JModelica installation directory, try to locate `/.../jmodelica/Python/pyjmi` directory and then Python script `casadi_interface.py`, where CasadiModel class is defined, is modified. To be on the safe side, make a copy of it and save it in a different folder with a different name, say `casadi_interface2.py`. First, importing necessary packages: `from casadi import *` and `from casadi.tools import *`. See Figure 4.2.

Now create CasadiModel class method named `symbolicJacobian()`. The code is given below (and Figure 4.3);

```
def symbolicJacobian(self):
# This function symbolically linearise explicit ODE model.
self.ocp.makeExplicit() # make the model explicit
a = self.ocp # access SymbolicOCP
rhs = a.ode # Get RHS of explicit ODE
# Get state
n_x = len(a.x) # number of states
x = list()
for i in range(n_x):
    x.append(a.x[i].var())
# Get [u,v] as input
input = a.u
# Get u
n_u = len(input) # number of inputs,
u = list()
for j in range(n_u):
    u.append(input[j].var())
# Get free parameters
```

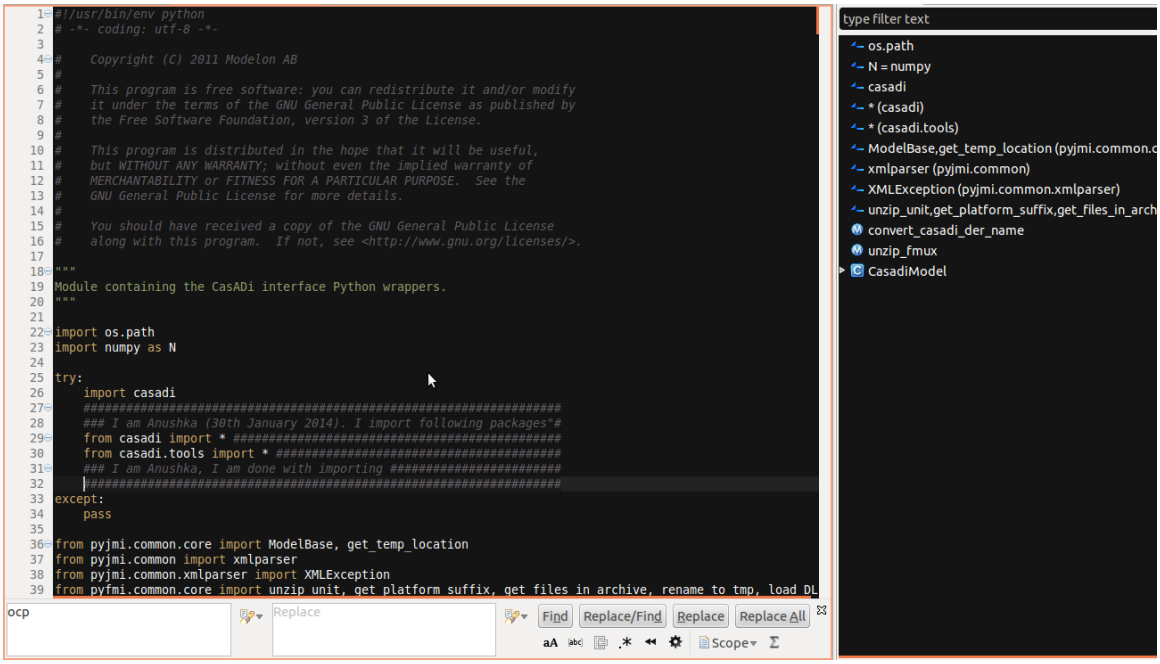

Figure 4.2: Editing Python script, `casadi_interface.py` - importing casadi packages.

```
p = list()
n_p = len(self.ocp.pi)
for k in range(n_p):
    p.append(self.ocp.pi[k].var())
# Create a SXFunction object
f = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[rhs])
f.init()
# Find symbolic Jacobian, A and B
A = f.jac(0)
B = f.jac(1)
return [A,B]
```

We can similarly define another CasadiModel class method, `numericJacobian()` to calculate numeric Jacobian matrices. See the code below. As a summery to this sub-section, the Casadi-Model has been extended so that we can obtain Jacobian matrices of a given dynamic model both numerically and symbolically. This modification will benefit on 'extraction of causality of Modelica models and in analysis of structural properties' of a given dynamic system.

```
def numericLinearisation(self,xk,uk):
[A,B,x,u,p] = self.symbolicLinearisation()
A_numJac = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[A])
A_numJac.init()
A_numJac.setInput(xk,0)
A_numJac.setInput(uk,1)
pValList = list()
for i in range(len(p)):
pValList.append(self.get(["{0}".format(p[i])]))
pValList = N.squeeze(pValList)
A_numJac.setInput(pValList,2)
A_numJac.evaluate()
```

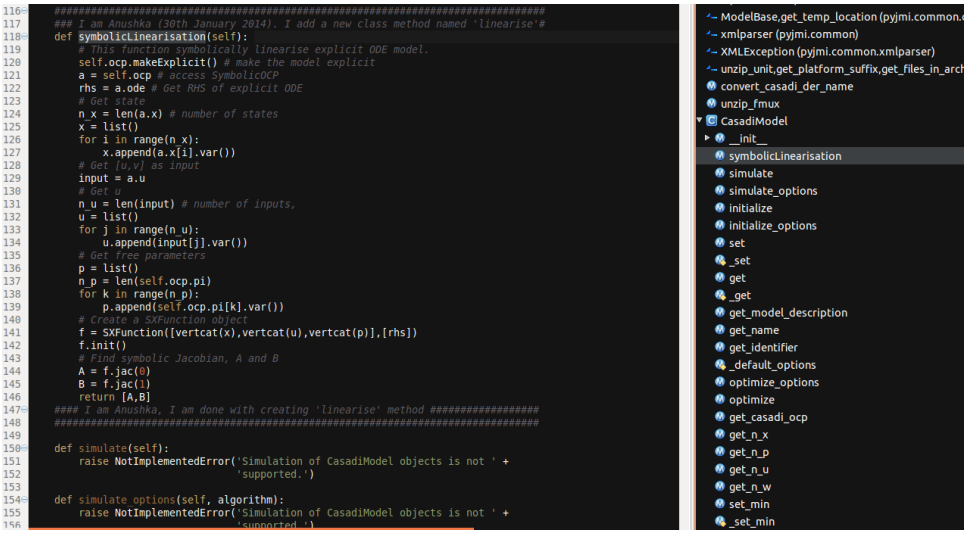

Figure 4.3: Editing Python script, `casadi_interface.py` - creating a class method, `symbolicJacobian()`.

```
Ak = A_numJac.getOutput()
B_numJac = SXFunction([vertcat(x),vertcat(u),vertcat(p)],[B])
B_numJac.init()
B_numJac.setInput(xk,0)
B_numJac.setInput(uk,1)
B_numJac.setInput(pValList,2)
B_numJac.evaluate()
Bk = B_numJac.getOutput()
Ak = N.array(Ak)
Bk = N.array(Bk)
return [Ak,Bk]
```

4.2 Extraction of Causality of Modelica Models and Structural Properties

Consider the dynamic model given in [11] and it has been modeled according to Modelica standards (with a slight modification by including an input variable). See Appendix-A for the Modelica Script. The main idea here is to compile the Modelica model, 'myModel.mo' using `compile_fmux` and load it to Python eviorenment using `CasadiModel` as explained before. Before attending into this, a brief idea is given how DAEs are represented in CasADi. Consider DAEs given below:

$$f_x\left(\dot{x}, x, z, p, t\right) = 0$$
$$f_z\left(x, z, p, t\right) = 0$$

where, x — differential state, z — algebraic state, p — parameter and t — time. Use, for example `help(?CVodesIntegrator)` for more information. The first equation represents a general implicit ODE while the second one an algebraic constraints. Conditionally, it is possible to transform an implicit ODE into its explicit companion, i.e. if $f_x\left(\dot{x}, x, z, p, t\right) = 0$ is such that

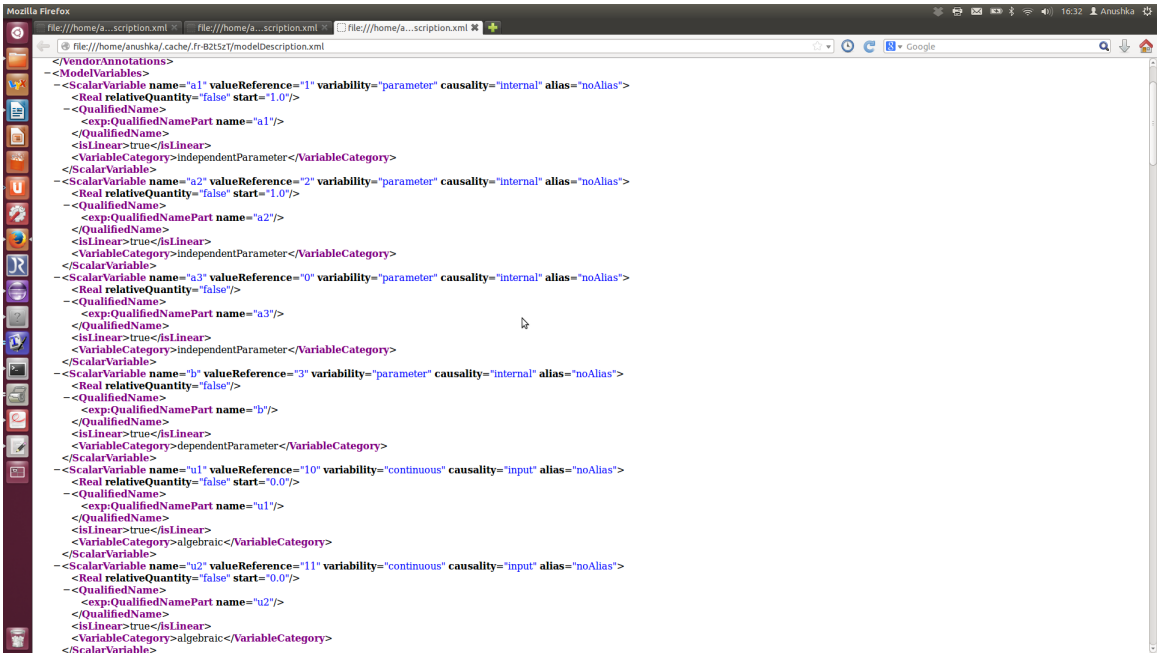

Figure 4.4: A part of the 'modelDescription.xml' file.

$\frac{\partial f_x}{\partial \dot{x}}$ is not singular then we can write,

$$\frac{\partial f_x}{\partial \dot{x}} \cdot \ddot{x} + \frac{\partial f_x}{\partial x} \cdot \dot{x} + \frac{\partial f_x}{\partial z} \cdot \dot{z} + \frac{\partial f_x}{\partial t} = 0$$

and consequently,

$$\ddot{x} + \left[\frac{\partial f_x}{\partial \dot{x}}\right]^{-1} \cdot \frac{\partial f_x}{\partial x} .\dot{x} + \left[\frac{\partial f_x}{\partial \dot{x}}\right]^{-1} \cdot \frac{\partial f_x}{\partial z} \cdot \dot{z} + \left[\frac{\partial f_x}{\partial \dot{x}}\right]^{-1} \cdot \frac{\partial f_x}{\partial t} = 0.$$

In short, we could say, $f_x(\dot{x}, x, z, p, t) = 0 \longrightarrow \dot{\hat{x}} - \hat{f}_x(\hat{x}, z, p, t) = 0$, if $\frac{\partial f_x}{\partial \dot{x}}$ is invertible. Here, x and $\hat{x}$ are may not the same, due to the presence of $\ddot{x}$ and the constraint function, f_z may be differentiated according to the index of the problem. If f_x is in the form of $M(x, z, p, t) \cdot \dot{x} - \tilde{f}_x(x, z, p, t)$ and the matrix $M(x, z, p, t)$ is invertible, then we can easily get the explicit ODE by $\dot{x} = [M(x, z, p, t)]^{-1} \cdot \tilde{f}_x(x, z, p, t)$. In this situation, the number is states is not changed during implicit to explicit transformation.

Once compilation is done using the `compile_fmux` compiler, it produces a folder with the extension of '.fmux'. The file name is the same as the Modelica file name. Example: for 'myModel.mo' it will be 'myModel.fmux'. modelDescription.xml is included in 'myModel.fmux', which contains the Modelica model's information. Some comments about the file 'modelDescription.xml': all the parameters defined in the Modelica code (ex. `parameter Real k1`.) come under two categories: (1) as independent parameters and (2) as dependent parameters. See Figure 4.4.

When we import the compiled the Modelica model (i.e. 'myModel.fmux') by `fmuxModel = CasadiModel('myModel.fmux')`, the model details given in the file 'modelDescription.xml' is used to create DAEs symbolically (use `fmuxModel.xmldoc` to access the '.xml' document.).

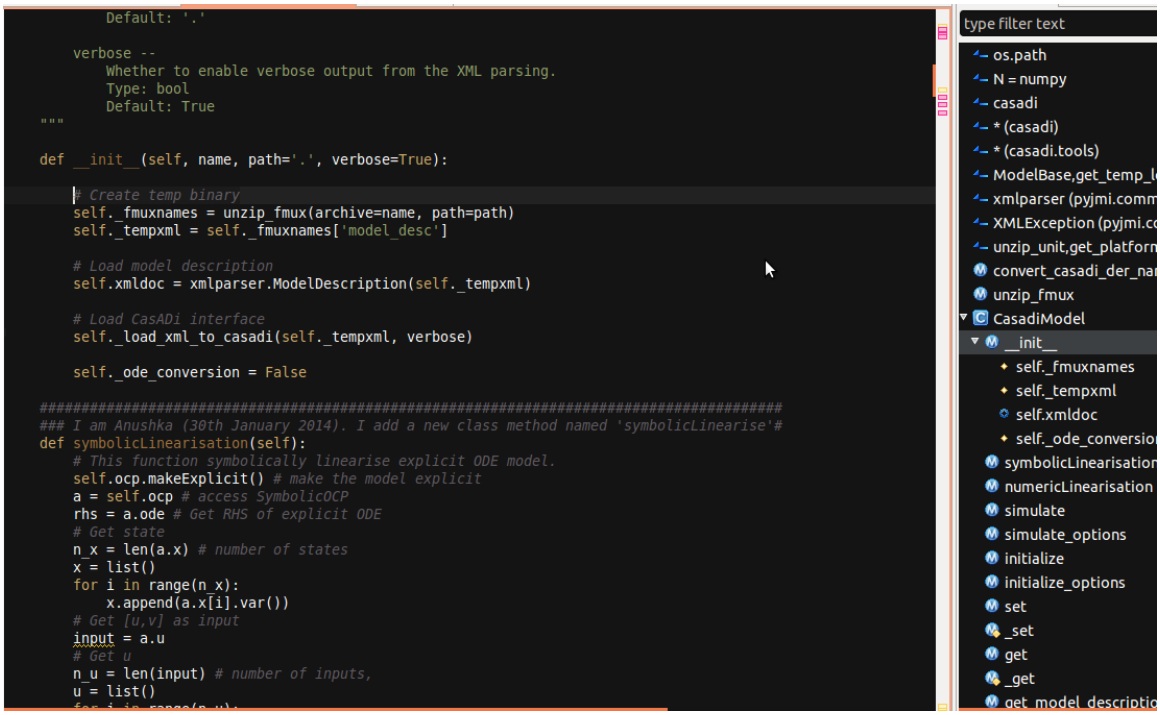

Figure 4.5: Initialization of CasadiModel objects.

`fmuxModel.ocp` gives access to the symbolic formulation of DAEs. For simplicity, say that `symOCP=fmuxModel.ocp`. Symbolic expression for f_x and f_z are given by `symOCP.ode` and `symOCP.alg` respectively. `symOCP.initial` returns specified initial state. In order convert implicit ODEs into explicit we use `symOCP.makeExplicit()`. But this is not always possible, as explained earlier. `symOCP.pi` and `symOCP.pd` gives independent and dependent parameters. Basically, the total number of parameters in the model is equal to `len(symOCP.pi)+len(symOCP.pd)`. The easiest way to get a list of both independent and dependent parameters into a single list, is to use the command, `fmuxModel._parameters`. The state, control input, algebraic states and time are taken respectively from `symOCP.x`, `symOCP.u`, `symOCP.z` and `symOCP.t`. Also `fmuxModel.dx` (but not `symOCP.dx`.) give variables to the corresponding variable, `der(x_n)` in the Modelica code. A summery: (1) create DAEs model in Modelica, (2) compile it through `compile_fmux` — this will create a '.xml' file in a directory with the extension — '.fmux', (3) load compiled model as a CasadiModel object (in `casadi_interface.py`) — when a CasadiModel object initialized, it unpacks the '.fmux' directory, extracts '.xml' file and (3) connects '.xml' file to CasADi via `self._load_xml_to_casadi` — actually, `self._load_xml_to_casadi` creates `self.ocp = casadi.SymbolicOCP()`. See Figure 4.5 and Figure 4.6.

The following discussion will consider a special family of DAEs,

$$\begin{aligned} M(x,z,p,t)\,.\dot{x} - f_x(x,z,p,t) &= 0 \\ z - f_z(x,p,t) &= 0. \end{aligned}$$

Here $M(x,z,p,t)$ is invertible. Hence the ODE given above can be translated into explicit form without adding any additional states which means $\dot{x} = [M(x,z,p,t)]^{-1} \cdot f_x(x,z,p,t))$. Let $\tilde{fx} \triangleq [M(x,z,p,t)]^{-1}.f_x(x,z,p,t)$. $z - f_z(x,p,t) = 0$ means all algebraic variables are explicitly given as a function of x, u, p and t (i.e. index 0.)[2] Now we can define Jacobian

[2]The idea behind taking a special family of DAEs is: implicit to explicit transformation (if this is possible)

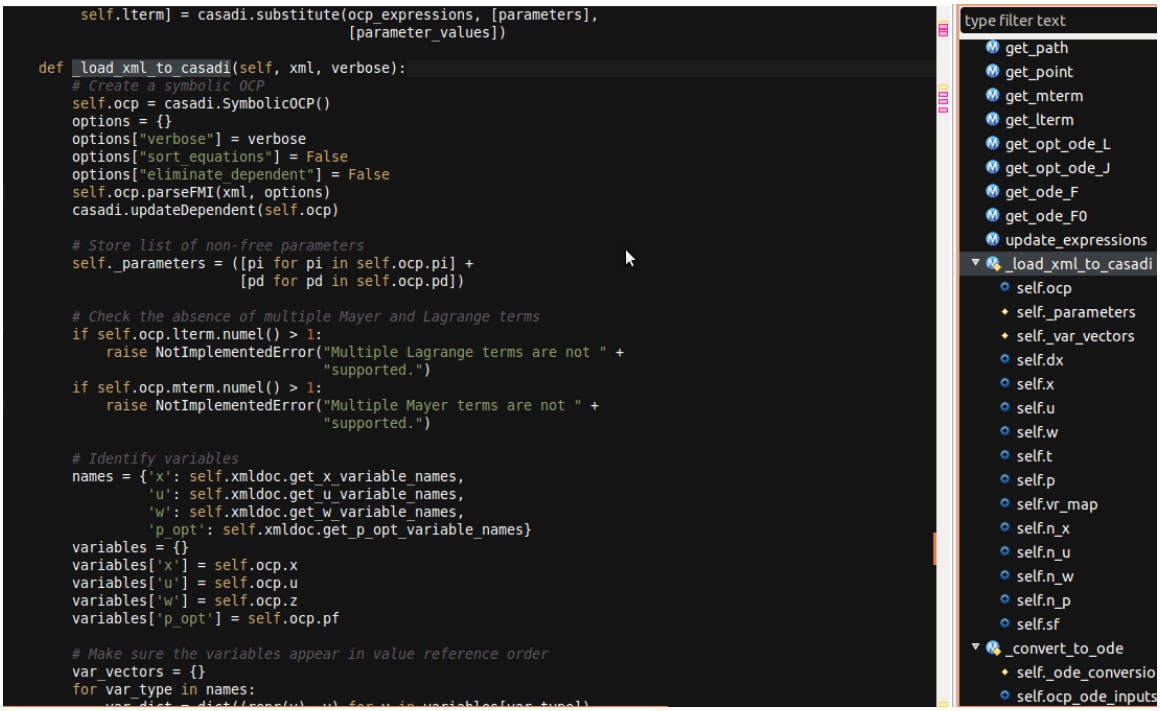

Figure 4.6: Connecting to the CasADi interface.

matrices as follows;

$$A = \frac{\partial \tilde{f}_x}{\partial x} + \frac{\partial \tilde{f}_x}{\partial z} \cdot \frac{\partial f_z}{\partial x}$$
$$B = \frac{\partial \tilde{f}_x}{\partial u} + \frac{\partial \tilde{f}_x}{\partial z} \cdot \frac{\partial f_z}{\partial u}$$

and to get output matrices C and D, we have to isolate sub-matrices from

$$C = \frac{\partial f_z}{\partial x}$$

and

$$D = \frac{\partial f_z}{\partial u}$$

as output variables are included in z.

A couple of couple methods are added to the CasadiModel class in casadi_interface.py. Those are related to finding symbolic/numeric Jacobian matrices and functions needed for structural analysis of dynamic models. See the code given in Appendix B and this function symbolically estimates system matrices A, B, C, and D as well as parameter sensitivity matrices on f_x and f_z. The syntax is: `CasadiModelObj.symbolicLinearization()`. Try the code given below (the Modelica model is given in Appendix A):

of ODEs will ultimately give an ODE in the form of $\dot{x} = f_x(x, z, p, t)$ and constraints function, f_z will transform into $z = f_z(\ldots)$. Note that f_z may contain derivatives of u.

```
import numpy as np
# Import compiler
from pymodelica import compile_fmux
# Import CasadiModel
from casadi_interface_edited import CasadiModel
# Modelica moodel's name and file
model_name = 'myModel'
model_file = 'myModel.mo'
# Compile
compile_fmux(model_name,model_file)
fmuxModel = CasadiModel('myModel.fmux')
# Symbolically linearize
fmuxModel.symbolicJacobian()
#
As = fmuxModel.symJac_A
print As
Bs = fmuxModel.symJac_B
print Bs
P1s = fmuxModel.symJac_P1
print P1s
Cs = fmuxModel.symJac_C
print Cs
Ds = fmuxModel.symJac_D
print Ds
P2s = fmuxModel.symJac_P2
print P2s
```

As, Bs, Cs, Ds, P1s, and P2s follow exactly the analytical results. The next function is to estimate Jacobian matrices numerically. `CasadiModelObj.numericLinearization()` is included. See Appendix C. See a sample code given below (the Modelica model is given in Appendix A):

```
# Import numpy package
import numpy as np
# Import compiler
from pymodelica import compile_fmux
# Import CasadiModel. Note that casadi_interface.py has been modified
# with new functionilitis and saved it in the working directory
# with the name casadi_interface_edited.
from casadi_interface_edited import CasadiModel
# Modelica moodel's name and file
model_name = 'myModel'
model_file = 'myModel.mo'
# Compile
compile_fmux(model_name,model_file)
fmuxModel = CasadiModel('myModel.fmux')
# Linearise numerically
n_x = fmuxModel.n_x
n_u = fmuxModel.n_u
xk = np.arange(n_x)
uk = np.arange(n_u)
[Ak,Bk,Ck,Dk] = fmuxModel.numericLinearization(xk,uk)
#
print Ak,Bk,Ck,Dk
```

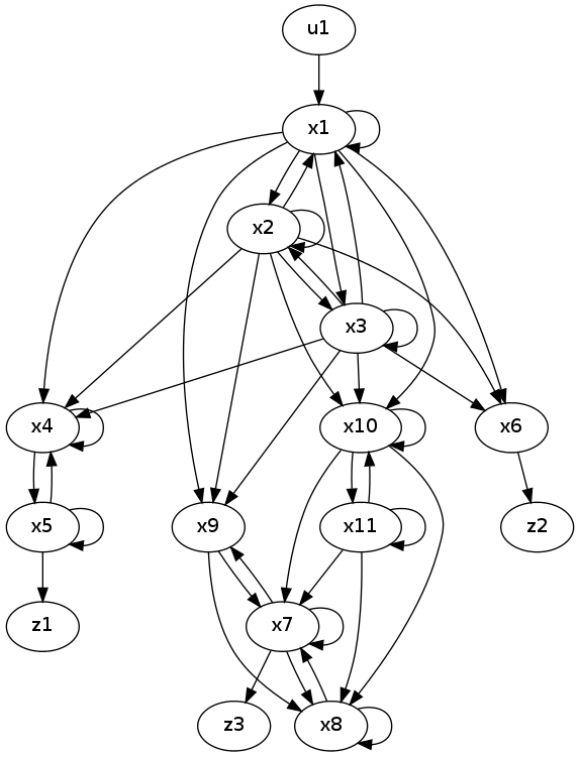

Figure 4.7: The graph of system matrices of the model given in [11].

Once we have symbolic system matrices, it may be possible possible to create graphs representing structure of the system. Refer to [11]. Python packages `pygraphviz` and `networkx` are used here for graphical visualization of graphic networks and for network analysis respectively. Therefore these packages have to be imported in `casadi_interface.py`. Nodes/vertices of graph corresponds to states, inputs (both control inputs and disturbances) and outputs. So the total number of nodes in the graph is equal to $n_x+n_u+n_y$. Edges are constructed as follows (consider the symbolic system matrices A, B, C, D): (1) state interactions — If $A[i][j] \neq 0$ then draw a directed edge from x_j to x_i, (2) state vs. input interactions — If $B[i][j] \neq 0$ then draw a directed edge from u_j to x_i, (3) state vs. output interactions — If $C[i][j] \neq 0$ then draw a directed edge from x_j to y_i, and (4) input vs. output interactions — If $D[i][j] \neq 0$ then draw a directed edge from u_j to y_i. To create nodes and edges, two methods are created in the CasadiModel class. See Appendix D and Appendix E. `CasadiModel.createEdges()` automatically creates directed edges based on symbolic system matrices. To draw a graph `CasadiModel.generateGraph()` method is used and the code is given in Appendix F. See a sample code given below and the result is in Figure 4.7.

```
# Import compilers
from pymodelica import compile_fmux
# Modelica moodel's details
model_name = 'myModel'
model_file = 'myModel.mo'
# Compile
compile_fmux(model_name,model_file)
# Load FMUX model
fmuxModel = CasadiModel('myModel.fmux')
# Create nodes and edges
fmuxModel.createNodes()
```

```
fmuxModel.createEdges()
# Generate a graph and save it as 'graph.png'
fmuxModel.generateGraph('graph')
```

In practice we are encountered large complex dynamic systems which often lead to large number states in the dynamic model. It is not possible to measure all the internal states. What is normally done is that only few state variables are measured and based on those measurements complete state is estimated. What we measure via sensors are called output variables. Generally, states are not independent each other. Therefore it is often enough to measure a sub-set of the state variables. Using a graph-theoretic approach we can systematically identify the minimum number of measurements should be made and which variables should be measured to attain the structural-state observability. More precisely says that structural observability is a necessary condition to the state observability. That means if the system is not structurally observable then the system is not observable. For further details refer to [11] [15] [16]. Let us see how to find minimum number of measurements should be measured so as the dynamic system given in Appendix A to be structurally observable. See the code below (also see Figure 4.8):

```
# Import compilers
from pymodelica import compile_fmux
# Modelica moodel's details
model_name = 'myModel'
model_file = 'myModel.mo'
# Compile
compile_fmux(model_name,model_file)
# Load model
from casadi_interface_edited import CasadiModel
fmuxModel = CasadiModel('myModel.fmux')
# Find strongly connected components
fmuxModel.createNodes()
fmuxModel.createEdges()
fmuxModel.generateGraph('dotFile')
fmuxModel.decomposeGraph()
```

First of all nodes and edges should be created using `fmuxModel.createNodes()` folllowed by `fmuxModel.createEdges()`. See Appendix D and Appendix E. In order to create nodes and edges Python package `pygraphviz` is used. `fmuxModel.generateGraph('dotFile')` generates two files in the working directory: a file with the name 'doFile.dot' and a figure file named 'dotFile.png. You may open 'dotFile.png and observe the state inter-dependencies. The graph (see Figure 4.8) may be decomposed into sub-graphs so called strongly connected components. Here only state nodes are considered. Each state-node in a strongly connected component has a directed path to any other node in the same sub-graph. `fmuxModel.decomposeGraph()` finds strongly connected components. Here we use the Python package `networkx`. Note that when we call `fmuxModel.generateGraph('dotFile')`, it creates a file named 'dotFile.dot as mentioned before and `fmuxModel.decomposeGraph()` reads 'dotFile.dot to create a `networkx` graph object. This is a way of converting `pygraphviz` graph objects into `networkx` graph objects. Strongly connected components are called root strongly connected components if they don't have outcoming edges. Minimum number of sensor measurements should be made to make the system structurally state observable is equal to number of root strongly connected components as well as if a root strongly connected component has more than one node then we can select one of the nodes can be a measurement and if it is a single node one then it must be measured.

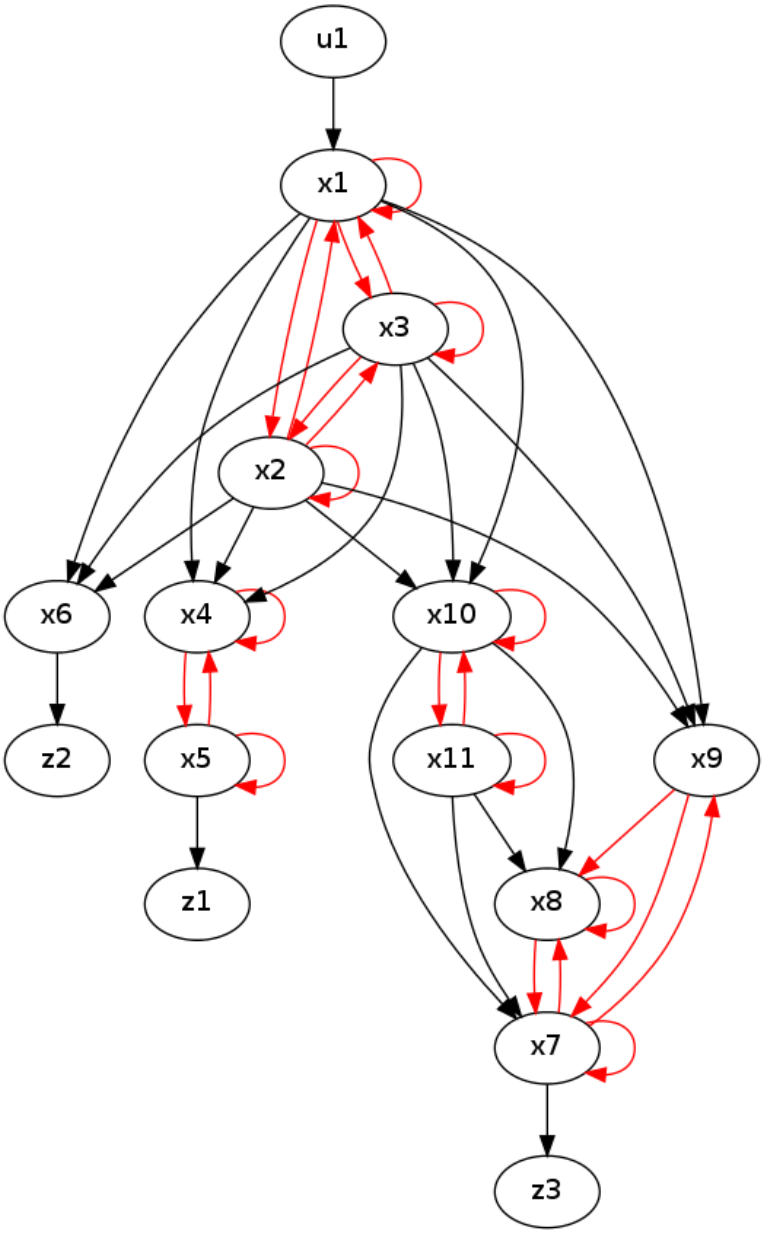

Figure 4.8: Strongly connected components of the model given in [11].

Chapter 5

Conclusion

There are many efficient free software packages available to handle nonlinear optimization problems and to integrate DAEs — Differential Algebraic Equations. It is our interest to use them in Python. IPOPT is such a free nonlinear optimizer and it is our main interest to use it in Python. It is possible to interface IPOPT with Python as with other software tools like MATLAB, C++, C, etc. However, as we already use JModelica.org — which is a free Modelica-based simulation and optimization platform uses Python as the scripting language — there is no need to bother with interfacing IPOPT with Python. The reason is that CasADi — which is a free optimization framework — has already been interfaced to JModelica.org and CasADi comes with the IPOPT solvers and many other solvers, integrators, etc. As a results we can use the IPOPT solver in Python via the JModelica.org-CasADi interface. I have shown in Chapters 3 how to use IPOPT solvers to solve nonlinear optimization problems in Python via JModelica.org. CasADi is a symbolical framework so Chapter 2 gives necessary basics symbolic manipulations in CasADi.

Generally, dynamic models are represented by DAEs and it is possible to map them into a Modelica code. A Modelica model can be translated into a symbolic representation using JModelica.org-CasADi interface. Then it is possible to obtain symbolic and numeric Jacobian matrices easily using CasADi. Section 4.1 shows how to do it for a special class of DAEs such that: (1) constraints function is such that `index = 0`, and (2) ODEs are converted to explicit form without adding additional states. A couple of modifications have been made to JModelica.org to do this. As a future work it is expected to modify JModelica.org so that it will handle any DAE model.

By isolating the structure of a dynamic system it is possible to apply a graph-theoretic approach to analyse generic properties of the system. Example: structural observability, structural controllability, system decomposition, etc. In Section 4.2 it is shown how to decompose a system into strongly connected components and hence deduce minimum number of measurements to be made to make the system to be structurally observable. In the future it is planned to add more functionalities based on graph-theoretic concepts to JModelica.org.

Appendices

Appendix A

The Modelica Model Used in Sub-Section 4.2

```
model myModel
// Author: Anushka Perera, anushka.perera@hit.no
// Telemark University College, Porsgrunn, Norway
// 16th September 2013
// The model is given in;
// Liu, Y.-Y., Slotine, J.-J., and Barabasi, A.-L.,
// ''Observability of Complex Systems,''
// Proceedings of the National Academy of Sciences, 2013.
//
// Define parameters
parameter Real k1 = 1.0;
parameter Real k2 = 2.0;
parameter Real k3 = 3.0;
parameter Real k4 = 4.0;
parameter Real k5 = 5.0;
parameter Real k6 = 6.0;
// Define differential states
Real x1;
Real x2;
Real x3;
Real x4;
Real x5;
Real x6;
Real x7;
Real x8;
Real x9;
Real x10;
Real x11;
// Define algebraic states
Real z1;
Real z2;
Real z3;
equation
// Define algebraic equations
z1 = x5;
z2 = x6;
```

```
z3 = x7;
// Define input variables
input Real u;
// Define differential equations
der(x1) = -k1*x1*x2*x3 + u;
der(x2) = -k1*x1*x2*x3;
der(x3) = -k1*x1*x2*x3;
der(x4) = k1*x1*x2*x3 - k2*x4 + k3*x5;
der(x5) = k2*x4 -k3*x5;
der(x6) = k1*x1*x2*x3;
der(x7) = k4*x8*x9 - k5*x7 + k6*x10*x11;
der(x8) = -k4*x8*x9 + k5*x7 + k6*x10*x11;
der(x9) = -k4*x1*x2*x3 + k5*x7;
der(x10) = k1*x1*x2*x3 - k6*x10*x11;
der(x11) = -k6*x10*x11;
end myModel;
```

Appendix B

The Python script for symbolicLinearization()

```
def symbolicLinearization(self):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Implicit to explicit conversion. I.e. fx(xDot,x,u,p,t)-->xdot = fx(x,u,p,t)
(2) Find symbolic Jacobians.
'''
  # Make the ODE explicit
        self.ocp.makeExplicit()
        # Get RHS of explicit fx
        fx = self.ocp.ode
        # Get state
        n_x = len(self.ocp.x) # number of states
        x = list()
        for i in range(n_x):
            x.append(self.ocp.x[i].var())
        x = vertcat(x)
        # Get u
        n_u = len(self.ocp.u ) # number of inputs,
        u = list()
        for j in range(n_u):
            u.append(self.ocp.u[j].var())
        u = vertcat(u)
        # Get parameters
        p = list()
        n_p = len(self._parameters)
        for k in range(n_p):
            p.append(self._parameters[k].var())
        p = vertcat(p)
        # Get z
        z = list()
        n_z = len(self.ocp.z)
        for l in range(n_z):
```

```
    z.append(self.ocp.z[l].var())
z = vertcat(z)
# Create a SXFunction object for fx
fxfcn = SXFunction([x,u,z,p],[fx])
fxfcn.init()
# Create a SXFunction object for fz
alg = self.ocp.alg
algfcn = SXFunction([x,u,z,p],[alg])
algfcn.init()
# Make fz explicit. I.e. fz(x,u,z,p,t) = 0 --> z = fz(x,u,p,t)
z1 = DMatrix(N.zeros(n_z))
[fz1] = algfcn.eval([x,u,z1,p])
fz1 = fz1*(-1.0) # now fz1 gives an expression for fz w.r.t. x, u, p, and t.
fz = SXFunction([x,u,p],[fz1])
fz.init()
# Replace z from fx. I.e. fx(x,u,z,p,t) --> fx2(x,u,p,t)
[z2] = fz.eval([x,u,p])
[fx1] = fxfcn.eval([x,u,z2,p])
fx2fcn = SXFunction([x,u,p],[fx1])
fx2fcn.init()
# Find symbolic Jacobian matrices, A, B and P1 (parameter sensitivity on fx)
symJac_A = fx2fcn.jac(0)
symJac_B = fx2fcn.jac(1)
symJac_P1 = fx2fcn.jac(2)
# Find symbolic Jacobian matrices, C, D and P2 (parameter sensitivity on fz)
output_eqn = SXMatrix()
for i in range(n_z):
    output_eqn.append(fz1[i])
output_fcn = SXFunction([x,u,p],[output_eqn])
output_fcn.init()
symJac_C = output_fcn.jac(0)
symJac_D = output_fcn.jac(1)
symJac_P2 = output_fcn.jac(2)
# Define new object properties
self.symJac_A = symJac_A
self.symJac_B = symJac_B
self.symJac_P1 = symJac_P1
self.symJac_x = x
self.symJac_C = symJac_C
self.symJac_D = symJac_D
self.symJac_P2 = symJac_P2
self.symJac_u = u
self.symJac_p = p
self.symJac_z = z
```

Appendix C

The Python script for numericLinearization()

```
def numericLinearization(self,xk,uk):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Find numeric Jacobians.
# Call symbolicJacobian()
        self.symbolicJacobian()
        # Extract symbolic matrices
        x = self.symJac_x
        u = self.symJac_u
        z = self.symJac_z
        p = self.symJac_p
        P1 = self.symJac_P1
        A = self.symJac_A
        B = self.symJac_B
        C = self.symJac_C
        D = self.symJac_D
        P2 = self.symJac_P2
        # Get numeric values of parameters
        pValList = list()
        for i in range(p.size()):
            pValList.append(self.get(["{0}".format(p[i])]))
        pValList = N.squeeze(pValList)
        # Find numerical A
        Af = SXFunction([x,u,p],[A])
        Af.init()
        Af.setInput(xk,0)
        Af.setInput(uk,1)
        Af.setInput(pValList,2)
        Af.evaluate()
        numJac_A = Af.getOutput()
        # Find nuerical B
        Bf = SXFunction([x,u,p],[B])
```

```
    Bf.init()
    Bf.setInput(xk,0)
    Bf.setInput(uk,1)
    Bf.setInput(pValList,2)
    Bf.evaluate()
    numJac_B = Bf.getOutput()
# Find numerical C
    Cf = SXFunction([x,u,p],[C])
    Cf.init()
    Cf.setInput(xk,0)
    Cf.setInput(uk,1)
    Cf.setInput(pValList,2)
    Cf.evaluate()
    numJac_C = Cf.getOutput()
    # Find numerical D
    Df = SXFunction([x,u,p],[D])
    Df.init()
    Df.setInput(xk,0)
    Df.setInput(uk,1)
    Df.setInput(pValList,2)
    Df.evaluate()
    numJac_D = Df.getOutput()
    # Convert to numpy arrays
    numJac_A = N.array(numJac_A)
    numJac_B = N.array(numJac_B)
    numJac_C = N.array(numJac_C)
    numJac_D = N.array(numJac_D)
    return [numJac_A ,numJac_B, numJac_C,numJac_D]
```

Appendix D

The Python script for createNodes()

```
def createNodes(self):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Create nodes of Graph.
# Initiate a MultiDiGraph
    self.G=pgv.AGraph(strict=False,directed=True)
    # Number of x's, u's, and y's
    n_x = self.n_x
    n_u =self.n_u
    n_z = self.n_z
    # xi's nodes
    for i in N.arange(0,nx):
        self.G.add_node('x{0}'.format(i+1))
    # ui's nodes
    for i in N.arange(1,self.ocp.u.size()+1):
        self.G.add_node('u{0}'.format(i+1))
# zi's nodes
    for i in N.arange(1,3+1): # n_y = 3
        self.G.add_node('z{0}'.format(i+1))
    # Add G as an object property
```

Appendix E

The Python script for createEdges()

```
def createEdges(self):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Create edges of Graph.
    # Symbolic Jacobians must have evaluated before creating edges!
self.symbolicLinearization()

# If G is not defined, implement self.createNodes()
try:
        self.G
    except NameError:
        self.createNodes()
    else:
        pass
# Extract symbolic matrices
symJac_A = self.symJac_A
    symJac_B = self.symJac_B
    symJac_C = self.symJac_C
    symJac_D = self.symJac_D
    # Find number of nodes
    n_x = self.n_x
    n_u = self.n_u
    n_z = self.n_z
    #
    # xi-xj interactions, symJac_A
    for i in N.arange(0,n_x):
        for j in N.arange(0,n_x):
            aij = symJac_A[i,j].toScalar()
            if isZero(aij) == bool(0):
                self.G.add_edge('x{0}'.format(j+1),'x{0}'.format(i+1))
    # xi-uj interactions, symJac_B
    for i in N.arange(0,n_x):
        for j in N.arange(0,n_u):
            bij = symJac_B[i,j].toScalar()
```

```
        if isZero(bij) == bool(0):
            self.G.add_edge('u{0}'.format(j+1),'x{0}'.format(i+1))
# zi-xj interactions, symJac_C
for i in N.arange(0,n_z):
    for j in N.arange(0,n_x):
        cij = symJac_C[i,j].toScalar()
        if isZero(cij) == bool(0):
            self.G.add_edge('x{0}'.format(j+1),'z{0}'.format(i+1))
# zi-uj interactions, symJac_D
for i in N.arange(0,n_z):
    for j in N.arange(0,n_u):
        dij = symJac_D[i,j].toScalar()
     if isZero(dij) == bool(0):
            self.G.add_edge('u{0}'.format(j+1),'z{0}'.format(i+1))
```

Appendix F

The Python script for generateGraph() and decomposeGraph()

```
def generateGraph(self,toFile):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Generate Graph.
self.G.write('{0}.dot'.format(toFile))
    self.G.layout(prog='dot')
    self.G.draw('{0}.png'.format(toFile))
    self.Gnx = ntwx.read_dot('{0}.dot'.format(toFile))

def decomposeGraph(self):
'''
Author: Anushka Perera, PhD candidate, Telemark University College, Porsgrunn.
Email - anushka.perera@hit.no / anushka_mrt@yahoo.com
Tel - 0047 450 19 636
03rd January 2014
This function does:
(1) Decompose based on stongly connected components.
    # Find strongly connected components
    G0 = self.Gnx
    for m in range(1,self.n_u+1):
        G0.remove_node('u{0}'.format(m))
    for m in range(1,self.n_z+1):
        G0.remove_node('z{0}'.format(m))
    Gnx_scc = ntwx.strongly_connected_component_subgraphs(G0)
    n_scc = len(Gnx_scc)
    for i in range(n_scc):
        G1 = Gnx_scc[i]
        Edges = G1.edges()
        Nodes = G1.nodes()
        #
```

```
        if len(Edges) != 0:
            for j in range(len(Edges)):
            self.G.get_edge(Edges[j][0],Edges[j][1]).attr['color'] \
            = "#%2x0000"%(255/n_scc*i)
            for k in Nodes:
            self.G.get_node(k).attr['style'] = 'filled'
                self.G.get_node(k).attr['fillcolor'] \
                = "#%2x0000"%(255/n_scc*i)
self.generateGraph(self.dotFile)
```

Bibliography

[1] Griewank, A., and Walther, A., "Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation," *SIAM*, 2008.

[2] Andersson, J., Akesson, J., and Diehl, M., "Dynamic Optimization with CasADi," *51st IEEE Conference on Decision and Control*, 2012.

[3] Andersson, J., Houska, B., and Diehl, M., "Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling anguages," *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, 2010.

[4] Andersson, J., Kozma, A., Gillis, J., and Diehl, M., "CasADi Users' Guide (WORKING COPY)," 2014.

[5] Di Ruscio, D., "Model Predictive Control with integral action," *lecture notes on SCE4106-Predictive Control with Implementation*, 2013.

[6] Coleman, T. F., and Zhang, Y., "MATLAB Optimization Toolbox, User Guide," *The MathWorks, Inc.*, 2013

[7] The MathWorks, Inc., "MATLAB, Mathematics," *http://www.mathworks.se/help/releases/R2013b/pdf_doc/matlab/math.pdf*, 2013.

[8] Blochwitz, T. (ITI), and Otter, M. (DLR-RM), "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," *Modelisar*, 2011.

[9] JModelica.org, "JModelica.org User Guide Version 1.12," *Modelon AB, Lund*, 2013.

[10] Lie, B. and Hauge, T. A., "Modeling of an industrial copper leaching and electrowinning process, with validation against experimental data," *In Proceedings SIMS 2008, 49th Scandinavian Conference on Simulation and Modeling, Oslo University college*, Oct 7-8, 2008.

[11] Liu, Y.-Y., Slotine, J.-J., and Barabasi, A.-L., "Observability of Complex Systems," *Proceedings of the National Academy of Sciences*, 2013.

[12] Vigerske, S. and Wächter, A., "Introduction to Ipopt: A tutorial for downloading, installing, and using Ipopt," 2013.

[13] Wächter, A., and Biegler, L. T., "On the implementation of an interior-point lter line-search algorithm for large-scale nonlinear programming," *Springer-Verlag*, 2005.

[14] Nocedal, J., and Wright, S. J., "Numerical Optimization," *2nd Edition*, 2006.

[15] Reinschke, K. J., "Multivariable Control-A Graph Theoretic Approach," 1988.

[16] Daoutidis, P., and Kravaris, C., "Structural Evaluation of Control Configurations for Multivariable Nonlinear Processes," 1991.

HiT Report No. 5

ISBN 978-82-7206-380-0
ISSN 1894-1044

Telemark University College
P.O. Box 203
3901 Porsgrunn

Phone 35 57 50 00
Fax 35 57 50 01
www.hit.no