Implementation of Periodic Continuation Algorithms in the Simulation Environment Diana

Anna Guseva
Max-Planck-Institut für Dynamik komplexer technischer Systeme
Sandtorstraße 1, 39106 Magdeburg, Germany
26.11.2008

Introduction

Diana [1] is a software package for dynamic simulation and nonlinear analysis of chemical and biochemical process models that has been developed at the Max-Planck-Institut für Dynamik komplexer technischer Systeme. Diana can be applied to differential algebraic equation systems that are usually generated by the process modeling tool ProMoT [2]. It contains numerical methods for determining steady state solutions, for the solution of initial value problems, for dynamic optimization as well as for continuation and singularity analysis. The objective of this work is to extend the capabilities of Diana in the area of continuation methods. M. Krasnyk developed in his Ph.D. thesis a stand-alone tool for the continuation of periodic solutions in one parameter [3]. In this work, the periodic continuation has been included in Diana and thus has been made available to all Diana users. The following report gives a detailed description of how to use periodic continuation methods in Diana.

Continuation of periodic solutions in Diana

While the numerical core algorithms in Diana are implemented as C++ routines, the user does not get in direct contact with these routines. Instead, a convenient and easy-to-use Python [4] interface is available for all necessary communications between Diana user and numerical kernel. In the following, a simple Python script is presented that allows to solve a periodic continuation problem. It can be easily extended and adapted to other applications.

The first step in a Diana session consists in importing all libraries that will be needed. This can be done by the commands below:

```
# script for testing
import sys, os, diana
from math import *
```

Then you need to create and initialize a model and a solver with indication of the model path and the solver name. In this test script periodic continuation for the HafkeReactor model will be investigated. This model describes the iron-(III)-catalyzed oxidation of ethanol by hydrogen peroxide in a continuous stirred tank reactor [5]. The model comprises an energy balance and five mass balances. It has been found to show a surprisingly complex nonlinear behavior [6].
try:
    os.system("/usr/bin/killall gnuplot_x11 >& /dev/null")
modelPath = "../../models/HafkeReactor/HafkeReactor.so"

dmain = diana.GetDianaMain()
modmanager = dmain.GetModelManager()
solvfactory = dmain.GetSolverFactory()
model = modmanager.CreateModel(diana.CAPE_CONTINUOUS, modelPath)
model.Initialize()
conti = solvfactory.CreateSolver(diana.CAPE_CONTI, model, "periodic.so")
conti.Initialize()

The next step is to specify initial values for the model equations and for the solver. The periodic continuation requires initial values either on a periodic orbit or at a Hopf bifurcation point. In this example, the continuation of the periodic branch is started at a Hopf bifurcation point. The initial values are read from a data file called HafkeReactorHopf.dat. The data file is generated from the HafkeReactorHopf.py script which computes the Hopf bifurcation point. The HafkeReactorHopf.py script will be described in the next section.

    eso = model.GetActiveESO()
    eso.LoadState('HafkeReactorHopf.dat')

In the next step, parameters for the periodic continuation algorithm are set:

    contipar = conti.GetParameters()
    contipar['MaxStepSize'].SetValue(1e+1)
    contipar['ATol'].SetValue(1e-08)
    contipar['EpsH'].SetValue(1e-1)
    contipar['TolR'].SetValue(1e-4)
    contipar['TolX'].SetValue(1e-6)
    contipar['RPM'].SetValue(False)
    contipar['Locking'].SetValue(True)
    contipar['Plotting'].SetValue(True)
    contipar['ResultsDir'].SetValue('results')
    contipar['OutputVars'].SetValue(['tk','c_[1]', 'temp'])
    contipar['Intermediate'].SetValue(False)
    contipar['VerboseLevel'].SetValue(3)

The main solver parameters have the following meaning:

- **MaxStepSize** limits the maximum step size of the continuation algorithm
- **Atol** stands for the integrator's absolute tolerance
- **EpsH** influences the distance of the first computed point on the periodic solution branch from the Hopf bifurcation point.
- **TolR** is the residual tolerance
- **TolX** is the tolerance of the Newton method
The boolean variable \textit{RPM} determines if recursive projection methods are used for the periodic continuation. RPM methods are especially efficient for high order systems. For details see \[3\].

\textit{Plotting} turns the online plot of results on and off.

For getting output variables on each continuation step you need to set the value of parameter \textit{Intermediate} to \textit{True}.

Further solver parameters for periodic continuation not used in this example are:

\begin{itemize}
  \item \texttt{Rtol} - integrator relative tolerance
  \item \texttt{AtolQ} - integrator absolute tolerance for the quadrature variable
  \item \texttt{RtolQ} - integrator relative tolerance for the quadrature variable
  \item \texttt{AtolE} - locking absolute tolerance
  \item \texttt{RtolE} - locking relative tolerance
  \item \texttt{EpsP} - absolute tolerance for the Picard iteration
\end{itemize}

Finally, one has to choose a model parameter that will be varied during the continuation. In this case, the coolant flow rate \textit{qknormal} is used as the continuation parameter:

\begin{verbatim}
  conti.AddFreeParameter('qknormal', 0, 1e-4)
\end{verbatim}

The next line starts the numerical computation:

\begin{verbatim}
  res = conti.Continue()
\end{verbatim}

\begin{verbatim}
except diana.ECapeUser, exceptn:
  print "error: ", exceptn
\end{verbatim}

Figure 1 shows the result of this continuation run. The amplitude of oscillations in the coolant temperature is depicted as a function of the coolant flow. The green curve shows the maximum coolant temperature under periodic conditions, the blue curve shows the temperature minima. The periodic branch starts from the Hopf bifurcation point located at the left boundary of the figure. With increasing coolant flow rate, the amplitude of the oscillations increases continuously, as well.

![Figure 1: Periodic continuation curves $Tk(qknormal)$](image)
Location of Hopf bifurcation points in Diana

As described above, one needs to know the steady state solution of a system at a Hopf bifurcation point, before the periodic continuation can actually be used. The following python script locates such a bifurcation point and saves the system states at this point to a data file. The script can be downloaded from `/usr/local/diana/tmp/build/diana/test/pytest/data/HafkeReactorHopf.py`.

Similar to the previously described script, the first few line load required libraries and model equations:

```python
import sys, diana
main = diana.GetDianaMain(sys.argv)
# get Diana model manager
mmanger = main.GetModelManager()
# create model
model=mmanger.CreateModel(diana.CAPE_CONTINUOUS, 
"../../models/HafkeReactor/HafkeReactor.so")
model.Initialize()

eso = model.GetActiveESO()

pcoll = eso.GetParameters()

pcoll.ItemByName("q").SetValue(1.84e-06)
pcoll.ItemByName("tkzu").SetValue(2.9165e+02)
pcoll.ItemByName("tzu").SetValue(2.9365e+02)
pcoll.ItemByName("c_zu").SetValue([3.0e+03, 0.0e+0, 0.0e+0, 6.54e+0])
pcoll.ItemByName("c_f_ges").SetValue(6.54e+0)
pcoll.ItemByName("dhr").SetValue([3.025e+02, 3.859e+02, 9.51e+01])
pcoll.ItemByName("e").SetValue([1.055e+02, 1.262e+02, 8.833e+01, 5.569e+01, 4.504e+01])
pcoll.ItemByName("k0").SetValue([3.7e+13, 4.83e+14, 1.65e+10, 3.72e+05, 3.8333e+04])
```

Then the algebraic solver is initialized that will be used for a one parameter continuation of steady states. The name of this solver in Diana is “sstate.so”. The coolant flow rate `qknormal` is again chosen as the continuation parameter.
sfactory = main.GetSolverFactory()
conti = sfactory.CreateSolver(diana.CAPE_CONTI, model, "sstate.so")
conti.Initialize()
# create the reporting interface and determine report variables
ri = main.CreateReportingInterface("basic")
conti.SetReportingInterface(ri)
ri.Add(eso.GetParameters().ItemByName("qknormal"))
ri.Add(eso.GetStateVariables().ItemByName("tk"))
ri.Add(conti.GetParameters().ItemByName("Stability"))
conti.AddFreeParameter("qknormal", 0.5e-5, 5e-5)
conti.GetParameters().ItemByName("VerboseLevel").SetValue(2)
conti.GetParameters().ItemByName("InitialDirection").SetValue(1.0)
conti.GetParameters().ItemByName("ConditionCheck").SetValue(diana.SteadyStateZCE)
conti.GetParameters().ItemByName("StabilityCheck").SetValue(True)
conti.GetParameters().ItemByName("MaxStepSize").SetValue(20.0)
conti.GetParameters().ItemByName("Parameterisation").SetValue("PseudoArclength")
conti.GetParameters().ItemByName("Predictor").SetValue("Tangent")
conti.GetParameters().ItemByName("InitialStepSize").SetValue(10.0)
conti.GetParameters().ItemByName("MaxStepsAmount").SetValue(200)
ret = conti.Continue()

The main steady state continuation solver parameters used above have the following meaning:
- **InitialDirection** specifies if the continuation initially goes in the direction of increasing values of the continuation parameter (then InitialDirection is set to 1.0) or in the direction of decreasing parameter values (then it must be set to -1.0)
- **Parameterisation** specifies the type of the parameterization. Possible values are Local or PseudoArclength. Then either local parameterization or pseudo arc length parameterization will be used. For details see [3].
- **Predictor** specifies the type of the predictor used (Tangent/Chord)
- **ConditionCheck** defines a stopping criterion for the continuation. SteadyStateZCE means that the continuation will be finished as soon as a pair of complex conjugate eigenvalues has crossed the imaginary areas.
- **MaxStepsAmount** is another stopping criterion. The continuation will be finished when more than the given number of steps have been performed.

Auxiliary steady state continuation solver parameters not used above are:
- **Stability** - Stability of the current point
- **ConditionCurrent** - check current continuation condition
- **EigMonitor** - number of eigenvalues to monitor
- **EigConverged** - number of converged eigenvalues
- **InitialCorrection** - initial correction which starts for the model parameters or state variables

The command **continue** in Line 49 finally starts the continuation process.

Note that the script above does not locate the Hopf bifurcation point exactly. It only stops the continuation after the solution branch has passed a Hopf bifurcation point. Therefore the outcome of this computation is only a first approximation of the Hopf bifurcation. To find the exact bifurcation
point, further refinement is needed. For this purpose, another solver called \textit{hopf.so} is initialized. Most solver parameters are identical to those already described. Therefore, the last lines of the script do not need further explanation:

```python
hopf = sfactory.CreateSolver(diana.CAPE_CONTI, model, "hopf.so")
hopf.Initialize()
hr = main.CreateReportingInterface("basic")
hopf.SetReportingInterface(hr)
hr.Add(eso.GetParameters().ItemByName("qknormal"))
hopf.AddFreeParameter("qknormal", 0.5e-5, 5e-4)
hopf.GetParameters().ItemByName("VerboseLevel").SetValue(4)
hopf.GetParameters().ItemByName("InitialDirection").SetValue(1.0)
hopf.GetParameters().ItemByName("MaxStepSize").SetValue(1.0e-3)
hopf.GetParameters().ItemByName("MaxStepsAmount").SetValue(20000)
hopf.GetParameters().ItemByName("Tol").SetValue(1e-12)
hopf.Solve()
eso.SaveState('HafkeReactorHopf.dat')
```

**References**