Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

H. Olsson, H. Tummescheit, H. Elmqvist
Dynasim AB; Modelon AB, Sweden

Using Automatic Differentiation for Partial Derivatives of Functions in Modelica

pp. 105-112

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Using Automatic Differentiation for Partial Derivatives of Functions in Modelica

Hans Olsson[1] Hubertus Tummescheit[2] Hilding Elmqvist[1]
[1]Dynasim AB, Lund, Sweden (Hans.Olsson@Dynasim.se, Elmqvist@Dynasim.se)
[2] Modelon AB, Lund, Sweden (Hubertus.Tummescheit@Modelon.se)

Abstract

The Modelica language has been enhanced with a notation for partial derivatives of Modelica functions. This paper presents how Dymola [4] enables the use of partial derivatives in certain modeling applications in the Modelica language. It is shown that using partial derivatives is natural and supported in Dymola, and solves several advanced modeling problems.

1 Introduction

Partial derivatives of functions arise naturally in a number of modeling applications. Accurate fluid property functions can be expressed as partial derivatives of a Gibbs- or Helmholtz function with respect to a few variables, e.g. for single-substance fluids as g(T,p), the Gibbs free energy, or f(T,ρ), the Helmholtz energy of the fluid, see [3]. Partial derivatives are also required to handle non-linear constraints in MultiBody mechanics and contact handling. For contact handling involving parametric surface descriptions, the tangents of each surface is required to specify the constraint equations for the contact point. The tangents are the partial derivatives of the parametric surface description function with regards to the two independent parameters.

These examples demonstrate that partial derivatives of functions occur in several modeling domains. Recently, the Modelica Design group took up this need and a language extension has been made to express partial derivatives of functions in the Modelica language. For this to be actually useful, a Modelica tool like Dymola has to have efficient techniques to generate computationally efficient code for the partial derivatives. In the following sections of the paper we are going to elaborate on the necessary techniques of code generation and give a few application examples of that. The examples are using the implementation of partial derivative generation in Dymola.

2 Automatic Differentiation of Modelica Functions

Using the Gibbs-function as an illustrative example, we will explain how partial derivatives are generated and used. Since the other thermodynamic properties of a fluid are described as partial derivatives the most natural way of expressing these partial derivatives is to directly express them in Modelica and let the tool, Dymola, differentiate the expressions.

Some simple examples are given in the table:

Property	**Formula**
Specific volume	$v = (\partial g / \partial p)_T$
Specific entropy	$s = -(\partial g / \partial T)_p$

Modelica 2.2 has thus been extended with the syntax:

```
function Gibbs_pp=der(Gibbs, p, p);
function Gibbs_pT=der(Gibbs, p, T);
```

to express that Gibbs_pp is the partial derivative of the function Gibbs with respect to p and p, and Gibbs_pT is the partial derivative of Gibbs with respect to p and T. Dymola's existing symbolic differentiation of expressions has been extended with a symbolic variant of automatic differentiation [1], which works for almost any Modelica function (including the Gibbs-functions). It uses forward-mode automatic differentiation. For systems of equations and index-reduction Dymola automatically uses additional derivatives (time-derivatives), and for index-reduction they are also constructed automatically. For the future it is planned to also automatically construct these when needed for Jacobians. Note that

even if time-derivatives and partial derivatives are different they share the same underlying framework.

The Gibbs-function is often fitted as a special polynomial in two variables (and some additional expressions). Generating optimized code for these special polynomials require special care, and is currently only implemented for the simple case of positive and negative *integer* powers. Extending it to handle rational powers will be straightforward. Due to the sparseness of the powers a simple use of Horner's scheme is not optimal, and obviously the pow-function is ruled out because of computational cost.

It is important that the automatic differentiation is done symbolically prior to the code generation since the code for special polynomials can reuse expressions for computing powers. Later this will be extended to include intra-function optimizations between the different partial derivatives. Using code optimization is the key to making symbolic differentiation an efficient form of automatic differentiation, see e.g. [2].

2.1 Basics of automatic differentiation

We will here present the basics of automatic differentiation in a general setting, even though we have only implemented the features needed for partial derivatives of functions. We will then consider the implementation choices, and the special cases in Modelica.

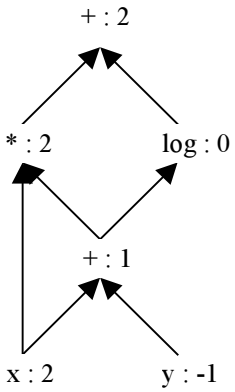
2.1.1 Forward mode

When performing automatic differentiation real variables and expressions are replaced by Taylor/power-series[1] (in one variable 't' – representing a directional derivative) whereas non-reals, e.g. the conditions of if and while-clauses are kept unchanged.

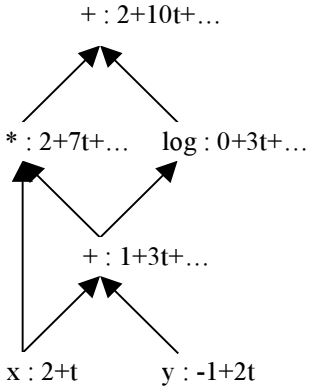
The rules for propagating Taylor-series through functions and expressions can be found in [6], and we will here only consider a trivial example

$$z := x \cdot (x + y) + \log(x + y)$$

The computation of this expression can be computed from its corresponding directed acyclical graph by propagating the numerical values.



Thus if $x = 2, y = -1$ we get $z = 2$. For automatic differentiation we replace the values at the bottom by Taylor-series and propagate these upwards:



Thus if $x = 2, \partial x / \partial t = 1, y = -1, \partial y / \partial t = 2$ we obtain $z = 2, \partial z / \partial t = 10$ and by including higher terms we would get higher order derivatives. For each node we only have to consider the values on the arrows entering it, which lead to efficient computations of directional derivatives, and is thus efficient for computing both partial derivatives and time-derivatives.

The result of this assignment-statement (in terms of Taylor-series) can then be used directly in the next statement in Modelica.

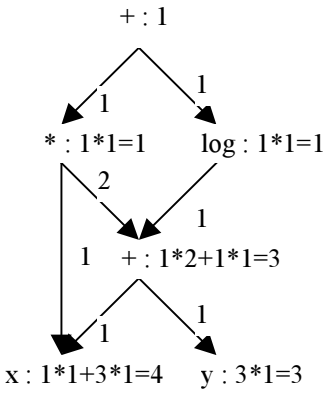
2.1.2 Reverse mode

Reverse-mode automatic differentiation [1] is an efficient technique for computing the derivative of one variable with respect to many. Since it is not as efficient for computing partial derivatives we will only present an example of how it works and not the underlying theory.

Consider the above example and augment the graph with the partial derivatives for each primitive operation (using the numerical values). Then start from the top where $\partial z / \partial z = 1$, and for each node compute the sum of the products of nodes above times the value along the edge.

[1] For higher order differentials a different scaling of the coefficients is more efficient – we will ignore that in this paper.

In order to indicate that values are propagated downwards the arrows have been reversed:



The interpretation of the result is that the value at each node represents the partial derivative of z with respect to this value: $\partial z / \partial x = 4, \partial z / \partial y = 3$ (which is consistent with the result above).

To implement reverse mode one first go through the algorithm once to compute the values for each node, and then once in reverse order using these values. This requires that all intermediate results are stored.

2.1.3 Implementation choices

Automatic differentiation can be implemented in several ways [1, 5], and forward-mode is in general simpler to implement than reverse-mode. We have selected to perform forward-mode symbolically in the Dymola kernel.

Another possibility that has attracted attention recently is to generate code that numerically propagates derivatives e.g. by overloaded operators in C++ or by modifying the code-generation for each primitive operation.

The derivative annotation in Modelica was designed with this in mind and can thus be used when computing Jacobians for non-linear systems using the 'time-derivative' of the function and also internally to compute partial derivatives.

Furthermore the 'time-derivative' functions (i.e. functions propagating a numerical directional derivative) can be constructed automatically by Dymola by automatic differentiation. This could have been implemented by modifying the code generation for each operation to also numerically propagate directional derivatives.

However, the symbolic variant has the advantage [5] that:

- Expressions independent of e.g. T do not have to propagate the derivative of T.
- The symbolic derivative is a new function that can be manipulated further by Dymola's kernel (e.g. to compute another derivative).
- No need to modify the code-generation in Dymola, and the generated code can be compiled with compilers on realtime platforms where C++-compilers are not always available.

The fact that the symbolic derivative is a new function is also used in this paper since it allows us to present the result of automatic differentiation as Modelica functions.

2.1.4 Special cases in Modelica

In Modelica, functions can contain simple expressions, matrix expressions, expression with iterators and if-, while- and for-clauses. The symbolic differentiation handles all of them, which has required special care, e.g. the rules for simple differentiable expressions with iterators are (where we use the special notation x'= $\partial x / \partial t$):

- {x(j)+x'(j)*t for j in 1:n}=
  {x(j) for j in 1:n}+{x'(j) for j in 1:n}*t
- sum(x(j)+x'(j)*t for j in 1:n)=
  sum(x(j) for j in 1:n)+sum(x'(j) for j in 1:n)*t
- product(x(j)+x'(j)*t for j in 1:n)=
  product(x(j) for j in 1:n)+sum(product(if j==k
  then x'(j) else x(j) for k in 1:n) for j in 1:n)*t+…

The Modelica expert will note that we have not included the rules for min- and max-expressions with iterators, since these are more complex to compute, often discontinuous, and currently not needed.

If the function being differentiated contains calls of other functions the directional derivative is propagated through it by its derivative-function, which is either specified in the derivative-annotation, or constructed by Dymola through automatic differentiation of the function (the latter case assumes the function is non-external).

2.1.5 Non-differentiable functions

A basic limitation of automatic differentiation is that it can provide a derivative even at points where the function does not have a derivative. Verifying that a function with branches (if-statements, if-expressions, or while-statements) is continuous is a difficult problem, see [7]. Furthermore in this reference it is shown that automatic differentiation may produce incorrect results for specific inputs if the function contains equality tests on real values.

For functions declared as partial derivatives (as is the focus of this paper) one can view the continuity as

the responsibility of the modeler declaring the partial derivative function. When using automatic differentiation for index-reduction this is not feasible, and the declarative approach in Modelica is that automatic differentiation for index reduction requires the function to specify the degree of continuity.

3 Fluid Property Modeling using Gibbs- or Helmholtz functions

The Gibbs-function is often fitted as a polynomial in two variables:

$$g(T,p) = \Sigma a_i p^{J_i} T^{K_i}$$

This type of function can be differentiated efficiently with the implementation of automatic differentiation in Dymola. The code for expressing the other thermodynamic properties has become much shorter than in previous implementations of fluid properties. Furthermore, conditions such as the phase equilibrium are also expressed as an equation involving the partial derivatives. Thus it is possible to describe phase equilibrium conditions, e.g. between gas and liquid phases, in a completely declarative way, without resorting to special algorithms. Initial numerical experiments seem to indicate that this works for typical working fluid in thermodynamic cycles, e.g. water or refrigerant R134a.

3.1 Definition of Thermodynamic Properties

A complete application example is the definition of phase equilibrium in two phase fluids. In particular when validity above the critical point is necessary, Helmholtz functions are used to describe high accuracy thermodynamic surfaces. A typical equation from [3] is the one for R134a using a dimensionless Helmholtz energy composed of an ideal (a_{id}) and a residual term (a_{res}) with the general form:

$$\frac{a_{res}(\tau,\delta)}{RT} = \sum_{i}^{1-4} \sum_{j}^{k_i - k_{i+1}} n_j \tau^{t_j} \delta^{d_j} \exp(-\delta^{d_i})$$

$$\frac{a_{id}(\tau,\delta)}{RT} = \ln(\delta) + a_1 \ln(\tau) + \sum_{i=1}^{3} a_{i+1} \tau^{t_i}$$

Where τ is a reduced inverse temperature and δ is a reduced density. This Helmholtz function uses fractional powers and for the test implementation it was more efficient to transform all exponents into integers via a variable substitution. All properties of interest are then computed by automatic differentiation using the new syntax form, and by computing the partial derivatives of $a_r(\tau,\delta)$ and $a_i(\tau,\delta)$ first, e.g.:

```
function ar_t=der(ar, tau);
function ar_tt=der(ar, tau, tau);
function ar_d=der(ar, delta);
```

and so on for all partial derivatives up to order two. The properties themselves are then defined as functions of these partial derivatives, e.g. the pressure

$$p = RT\rho\left(1 + \rho \frac{\partial}{\partial \delta} a_{res}(\tau,\delta)\right)$$

Which results in a nice, compact definition in Modelica:

```
function pressure "pressure"
  input SI.Temperature T;
  input SI.Density d;
  output SI.Pressure p;
protected
  Real delta = d/DCRIT "dim-less density";
  Real tau = TCRIT/T
   "dimensionless inverse temperature";
algorithm
  p := R*T*d*(1+delta*ar_d(tau,delta));
end pressure;
```

The Gibbs energy is computed as:

```
function gibbsEnergy "Gibbs free energy"
  input SI.Temperature T;
  input SI.Density d;
  output SI.SpecificEnergy g;
protected
protected
  Real delta =  d/DCRIT "dim-less density";
  Real tau = TCRIT/T
     "dimensionless inverse temperature";
algorithm
  g := R*T*(1+a0(tau,delta)+ar(tau,delta)
              + delta*ar_d(tau,delta));
end gibbsEnergy;
```

In an equivalent manner, all other properties of interest are defined.

3.2 Declarative Definition of Phase Equilibrium conditions

All current Modelica libraries define two phase fluids for dynamic simulation via auxiliary equations, e.g. splines generated from accurate phase boundary data, that are a very good approximation to the correct thermodynamic equilibrium conditions. There is one fundamental drawback to that approach: the approximation accuracy is fixed and has to be chosen quite high to prevent numerical inconsistencies at tight solver tolerances. From a perspective of a de-

clarative, equation based language, a declarative definition of the phase equilibrium condition has the advantage that it is always solved to the current accuracy of the numerical solver. This means that in contrast to current implementations that have a maximum accuracy that is limited by the accuracy of the phase boundary approximation, a declarative definition does not have this drawback. Even though in Modelica it would be possible to define an iterative scheme to compute phase equilibrium conditions, the algorithm would need the current solver tolerance as a user-defined input, again an undesirable drawback. The declarative definition for phase equilibrium are the two equations:

$$p(T, \rho_{liquid}) = p(T, \rho_{vapour})$$

$$g(T, \rho_{liquid}) = g(T, \rho_{vapour}),$$

i.e. equality of the pressures and Gibbs energies computed from the same saturation temperatures and the liquid and vapour densities ρ_{liquid} and ρ_{vapour} respectively.

With the property functions defined in the last sections, the phase equilibrium conditions only need the variables and equations in the following code fragment:

```
  SI.Density dl(start = 1500.0) "liquid";
  SI.Density dv(start = 5.0) "vapour";
  SI.Temperature T(start = 270.0);
equation
  p = pressure(T,d);
  h = enthalpy(T,d);
  pressure(T,dv) = pressure(T,dl);
  gibbsEnergy(T,dl) = gibbsEnergy(T,dv);
```

From these equations the non-linear solver will compute the liquid and vapour densities for the saturation temperature T. The equations are taken from the context of a dynamic control volume model that assumes the pressure p and the enthalpy h as dynamic states.

3.3 Discussion

These equations have so far been tested in simple setups, and robustness, speed and convergence were excellent, provided that the initial values for ρ_{liquid} and ρ_{vapour} were close to the initial equilibrium point.

Unfortunately, there are still some unsolved problems that prevent to use this formulation in many applications: above the critical point these equations loose a meaning and it was numerically not possible to obtain meaningful results for dynamic simulations that come from a supercritical state and go to a subcritical state. The main problem here is that the equilibrium conditions, for the Helmholtz equation above, has several non-physical, numerically valid solutions in the unstable region inside the two-phase dome.

Dymola can handle inequality-constraints on the solutions of non-linear systems, but we have not yet determined the best way specify these inequalities in Modelica, because we need to be sure to find only the thermodynamically stable solutions outside of the spinoidal lines. There are inequality conditions on some partial derivatives for these non-physical solutions, see [3], that could be used to disambiguate unwanted solutions.

A combination of some auxiliary functions for start values and to disambiguate non-physical solutions with the declarative definition of the phase boundary through equations is likely to be a compromise that works robustly under all conditions and avoids the disadvantages of both approaches.

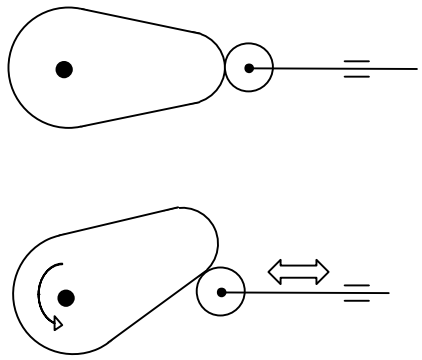
In the test implementation, Jacobians for the non-linear equations have not yet been derived automatically. This would further improve the robustness and solution speed. In dynamic simulation, Dymola uses the last solution point as a start for the next iteration and that makes the otherwise time consuming equilibrium iterations quite fast. For large systems of non-linear equations arising from steady-state problems, the method of using auxiliary functions for the saturation pressure and temperature is likely to be more robust and more flexible. Often the auxiliary functions can be chosen in a way that explicit evaluation is possible in cases when the proper thermodynamic definition inevitably leads to a non-linear equation system.

The main advantage of automatic differentiation for medium properties is the economy of code: the thermodynamic surface for R134a, the computation of the phase boundary and all derived properties is less than 5% of the amount of code with conventional programming “by hand” and auxiliary functions representing the phase boundary.

4 Non-linear Constraints in Multi-Body Mechanics and Contact handling

Partial derivatives are sometimes required to handle non-linear constraints for contact handling in Multi-Body mechanics, see also [9].

As a detailed example, we will consider the CAM mechanism shown below, [8]. The CAM has straight flanks between the circle segments. The follower is roller-ended.



The CAM shape will be described by a replaceable function defining the two dimensional position vector of every point of the circumference as a function of a free parameter theta.

```
partial function shapeFunction
  input Real theta;
  output Real r[2];
end shapeFunction;
```

The straight flanks CAM can be defined as follows:

```
function straightFlanksCam
  extends shapeFunction;
  input Real R1=1   "Base circle radii";
  input Real R2=0.5 "Nose radii";
  input Real d=2    "Centre distance";
  import Modelica.Math.*;
protected
  constant Real pi=Modelica.Constants.pi;
  Real fi0;
  Real fi1;
  Real fi2;
  Real L;
  Real x;
  Real y;
  Real thetamod;
algorithm
  thetamod := atan2(sin(theta),
     cos(theta))
    "to get angle in interval -pi..pi";
  fi0 := asin((R1 - R2)/d);
  if thetamod > 0 then
    fi1 := pi/2 - fi0 - thetamod;
  else
    fi1 := - pi/2 + fi0 - thetamod;
  end if;
  fi2 := atan2(R2*cos(fi0), d +
           R2*sin(fi0));
  if abs(thetamod) > pi/2-fi0 then
    x :=R1*cos(theta);
    y :=R1*sin(theta);
  elseif abs(thetamod) >= fi2 then
    L := R1/cos(fi1);
    x := L*cos(theta);
    y := L*sin(theta);
  else
    L := d*cos(abs(thetamod)) +
     sqrt(R2^2 - d^2*sin(abs(thetamod))^2);
    x := L*cos(theta);
    y := L*sin(theta);
  end if;
  r := {x,y};
end straightFlanksCam;
```

The mechanism can be described by the following equations (three dimension vectors are used for convenience although the mechanism is planar).

$$\mathbf{r}(\theta) = shape(\theta)$$

$$\mathbf{T}(\varphi)\cdot(\mathbf{r}(\theta) + \mathbf{n\_n}(\theta)\cdot(R + dist)) = \{x, y, z\}$$

$$\mathbf{r}_\theta(\theta) = shape\_theta(\theta)$$

$$\mathbf{n}(\theta) = \mathbf{r}_\theta(\theta) \times \{0, 0, 1\}$$

$$\mathbf{n\_n}(\theta) = \frac{\mathbf{n}(\theta)}{\sqrt{\mathbf{n}(\theta)\ \mathbf{n}(\theta)}}$$

$$\mathbf{T}(\varphi)\cdot F\_n\cdot\mathbf{n\_n}(\theta) = \{Fx, Fy, Fz\}$$

where

$\mathbf{r}$	vector to closest point
θ	angle to closest point
φ	angle of CAM
$\mathbf{T}$	Transformation matrix for rotation around z axis
x,y,z	position of center of follower
R	radii of follower
dist	distance between CAM and follower
$\mathbf{n}$	normal of CAM shape
$\mathbf{n\_n}$	normalized normal
F_x, F_y, F_z	force on follower
F_n	normal force

The partial derivative of the replaceable shape function is needed. The Modelica language has recently been extended to allow the der-operator to define partial derivatives of Modelica functions.

```
replaceable function shape =
  shapeFunction;
function shape_theta =
  der(shape, theta);
```

This allows to write equations in the following form:

```
r = shape(theta);
r_theta = shape_theta(theta);
```

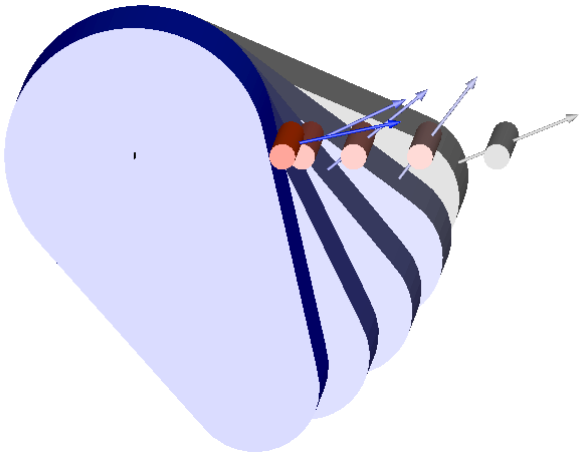
A tool needs to use automatic differentiation to obtain the required partial derivative. Dymola generates the derivative function in Modelica format:

```
function shape_theta
  input Real theta;
protected
  Real r[2];
public
  input Real R1 := 1 "Base circle radii";
  input Real R2 := 0.5 "Nose radii";
  input Real d := 2 "Centre distance";
protected
  constant Real pi := 3.14159265358979;
  Real fi0;
  Real fi1;
  Real fi2;
  Real L;
  Real x;
  Real y;
  Real thetamod;
  Real theta_d13 := 1;
public
  output Real r_d13[2];
protected
  Real fi0_d13, fi1_d13, fi2_d13;
  Real L_d13, x_d13, y_d13;
  Real thetamod_d13;
algorithm
  thetamod_d13 := theta_d13;
  thetamod := arctan2(sin(theta), cos(theta));
  fi0_d13 := 0;
  fi0 := arcsin((R1-R2)/d);
  if (thetamod > 0) then
    fi1_d13 :=  -(fi0_d13+thetamod_d13);
    fi1 := 0.5*pi-fi0-thetamod;
  else
    fi1_d13 := fi0_d13-thetamod_d13;
    fi1 := fi0-0.5*pi-thetamod;
  end if;
  fi2_d13 := -((d+R2*sin(fi0))*R2*fi0_d13*
    sin(fi0)+R2*cos(fi0)*R2*fi0_d13*cos(fi0))/
    ((R2*cos(fi0))^2+(d+R2*sin(fi0))^2);
  fi2 := arctan2(R2*cos(fi0), d+R2*sin(fi0));
  if (abs(thetamod) > 0.5*pi-fi0) then
    x_d13 := -R1*theta_d13*sin(theta);
    x := R1*cos(theta);
    y_d13 := R1*theta_d13*cos(theta);
    y := R1*sin(theta);
  elseif (abs(thetamod) >= fi2) then
    L_d13 := R1*fi1_d13*sin(fi1)/cos(fi1)^2;
    L := R1/cos(fi1);
    x_d13 := L_d13*cos(theta)-L*theta_d13*
      sin(theta);
    x := L*cos(theta);
    y_d13 := L_d13*sin(theta)+L*theta_d13*
      cos(theta);
    y := L*sin(theta);
  else
    L_d13 := -(d*thetamod_d13*simplesign(thetamod)*
      sin(abs(thetamod))+d^2*sin(abs(thetamod))*
      thetamod_d13*simplesign(thetamod)*
      cos(abs(thetamod))/sqrt(R2^2-
      d^2*sin(abs(thetamod))^2));
    L := d*cos(abs(thetamod))+sqrt(R2^2-
      d^2*sin(abs(thetamod))^2);
    x_d13 := L_d13*cos(theta)-L*theta_d13*sin(theta);
    x := L*cos(theta);
    y_d13 := L_d13*sin(theta)+L*theta_d13*cos(theta);
    y := L*sin(theta);
  end if;
  r_d13 := {x_d13, y_d13};
end shape_theta;
```

The *dist* variable can be used to define a spring acting when there is penetration:

```
F_n = if dist < 0 then k*(-dist) else 0;
```

Consider the follower connected to a mass which is connected to a spring and damper to ground and that the CAM is rotating at a fixed angular velocity. For high speeds, the follower will leave the nose and bounce back on the flank. Such a case is illustrated below in several frames from an animation.



It should be noted that the redeclared shape function is also used to define the parametric surface used for the animation.

If the contact model also contains damping, the derivative of the shape_theta function is also needed during index reduction. Such contact models are very stiff. In certain cases an idealized contact model with the constraint dist=0 might be sufficient. In such a case, F_n is the constraint force. Index reduction will in that case require the second derivative of shape_theta. It is clear already by inspection of the automatically generated shape_theta function that automatic differentiation to obtain this function and its derivatives saves the modeler much tedious and error-prone work.

5 Conclusions

The paper demonstrates that extending Modelica with partial derivatives of functions is natural and solves a number of advanced modeling problems. Dymola automatically handles the differentiation of the partial derivatives of the functions thus reducing the work needed by the modeler, while preserving the efficiency of the generated simulation code.

References

[1] Juedes D.W. (1991): **Taxonomy of Automatic Differentiation tools**, in Automatic Differentiation of Algorithms. SIAM

[2] Char B.W. (1991): **Computer Algebra as a Toolbox for Program Generation and Manipulation**, in Automatic Differentiation of Algorithms. SIAM

[3] Span R. (2000): **Multiparameter Equations of State, an accurate Source of Thermodynamic Property Data**. Springer Verlag.

[4] Dynasim (2005): **Dymola User's Manual**, www.dynasim.se

[5] Olsson H. (1993): **Applications of automatic and symbolic differentiation in numerical computations**. Master Thesis, Department of Computer Science, Lund Institute of Technology, Lund Sweden.

[6] Rall L.B. (1981): **Automatic differentiation: Techniques and applications**. vol. 120 of Lecture Notes in Computer Science, Springer-Verlag.

[7] Fischer H. (1991): **Special problems in Automatic Differentiation**, in Automatic Differentiation of Algorithms. SIAM.

[8] Hannah J., Stephens R. C. (1972): **Mechanics of Machines**. Second edition, SI units, Edward Arnold.

[9] Otter M., Elmqvist H., Lopez J.L. (2005): **Collision Handling for the Modelica MultiBody Library.** Modelica'2005 conference, Hamburg, March 7-8.