Automatic Differentiation and Continuous System Formal Verification

A research proposal based on literature survey and basic experiments

by

Yan Peng

B.Eng., Zhejiang University, 2012
M.Sc., The University of British Columbia

A REPORT SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE COURSE
CPSC513 FORMAL VERIFICATION

MASTER OF SCIENCE

in

The Faculty of Science

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2013

Table of Contents

List of Tables

List of Figures

Acknowledgements

Thank Prof. Mark Greenstreet for the instructions throughout this project.
Thank Jijie Wei for providing the simple version of MATLAB integrator.

Chapter 1

Problem

1.1 Problem Description

In the world of formal techniques and formal verification, one tradition approach for continuous problems is to model the system as ordinary differential equations or partial differential equations and try finding the dynamical behaviours of the system in order to recognize the pattern of the dynamics and verify certain properties of the system. Many formal techniques has been developed in this field.

In the world of circuits, the problem becomes specific. The variables becomes node voltages of a circuit and the mechanism of circuits, built upon mechanism of NMOS and PMOS, can be modelled by ordinary differential equations. Given different initial voltages to the nodes, the circuit will run in totally different trajectory to some future states.

Verification of a circuit is important. Two examples serve to show my statement. First, chip companies need it. Even if a tiny mistake in the design of a chip circuit can make it useless and once massive production of it is done, the company can only waits to lose money. Second, safety-required circuit design needs it. Imagine there are some safety qualifications for a robot which needs to be implemented in the circuit of it. Other applications could be in public transportation, public electrical devices and so on.

Reachability analysis Reachability analysis asks the question where the region will be after the system starts at some point in the space and run for certain amount of time. The task can be realized through two phases: modelling a group of points and calculating the next states after some time. Possible modelling can be polygons, ellipsoids, and projectagons, etc. For continuous state calculation, there exist exact methods and over-approximate methods.

Finding a Trajectory Based upon reachability analysis, one can seek to find a possible trajectory of a circuit, which can be a possible trajectory that is out of the qualification of the circuit, more directly called, a bug.

In real life companies, testers usually tends to design test cases manually by themselves. Formal methods can provide them a way to find a counter-example automatically.

For example, when given the ODE of a circuit system, one can work out the Jacobian of it by differentiation, and further the sensitivity matrix of the system. The sensitivity matrix tells people how node points' voltages change throughout the whole time in relation of the change in initial inputs. For example, now we know a trajectory is quite close to fail the qualification, by looking at the sensitivity matrix, we know how to change the initial state to generate a counter-example for the system.

Deciding Period of an Oscillator In deciding the period of an oscillator, one technique to use is Newton's method. In the updating of period and initial states, one also needs to figure out the sensitivity matrix of a system, which requires some differentiation to get the Jacobian matrix as stated above.

Automatic Differentiation Automatic differentiation is a single field outside formal verification, however, as one looks into verification of continuous systems, one find automatic differentiation tools of great help.

There exist four ways of differentiation in practise: hand coding when given the specific functions, numerical methods like divided differences, symbolic differentiation which gives one the exact formula of derivatives and at last, automatic differentiation.

Automatic differentiation(AD) is a method different from numerical methods because there is no approximation except for the floating point rounding. It is also different from symbolic methods because results of AD are differentiation values at given points. The method in essence is quite simple and beautiful, it just follows the chain rule of basic calculus: if we have a composition of functions $f(x) = g(h(x))$, the rule says we can calculate the derivative of function f in this way: $\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx}$. There basically exist two methods in AD: forward method and reverse method. The reverse method takes less time than forward method in theory because it doesn't have to carry out calculation for all variables for each sub-function. Other methods may be combining the two approaches or other small justifications to improve performance.

As stated that AD is a useful tool, there exists lots of applications. In optimization problems, often, gradients and Hessians must be calculated. Here, the AD methods must be adapted to higher dimension differentiation, as vectors and matrices. The theory become more difficult to understand. Another possible application is in our problem of formal verification of continuous dynamic systems.

1.2 A literature survey

In this part, I want to do a survey on the development of automatic differentiation algorithms. Automatic differentiation, also called algorithmic differentiation, is good for both its automatism and precision.

The original idea of automatic differentiation date back to 1950s when Kahrimanian and Nolan write their Master's thesis [6, 7]. The basic idea, as stated in Nolan's thesis, comes from how compilers decompose a large scale program into fragments and solve by part. After that, lots of effort has been put into implementation AD tools on different computer systems using different languages.

One person needed to mention is Prof. Andreas Griewank from Humboldt-University. His main dedication lies in the tremendous books and articles published in this field by him. In his 1989 paper *On Automatic Differentiation*[3], he stated:

> Under quite realistic assumptions the evaluation of a gradient requires never more than five times the effort of evaluating the underlying function by itself.
>
> — A. Griewank(1989).

The statement tells us forward mode AD does too much calculation than it is necessary. We didn't wait for long for reverse mode AD came into people's sight. G. M. Ostrovskii[11] and Bert Speelpenning[9] independently introduced the reverse mode AD.

There exist two ways to calculate the reverse mode AD, source transformation and operator overloading. These are two totally different kind of implementation. Source transformation takes the function for differentiation as input, compile it based on the chain rule to produce the compiled code. The compiled function has the original code in reversed order and calculates the derivatives. The main effort must be spent on writing a compatible compiler. Operator overloading walks through the function in forward order and record the computation on a 'tape'. A related technique is the checkpoint strategy([4]) to determine the upper bound for use of memory. A more detailed list of papers and tools in the history of AD can be found at[2].

Some of the most famous tools in MATLAB are INTLAB[8] which implement forward mode AD using operator overloading method; TOMLAB/MAD which also implement forward AD using operator overloading[10]; ADiMat[1] which implements forward AD using both source transformation and operator overloading method. One can see from the existing tools that it is difficult to implement reverse mode AD in MATLAB directly.

1.3 Problem Statement

In this research proposal, the problem I'm addressing is the application of automatic differentiation in the field of formal verification of continuous dynamical system.

To be more specific, the AD methods with focus is the pure forward and reverse mode methods and the continuous dynamic system falls in the category of analog circuits.

1.4 Problem Solution

My solution to this problem can be accomplished in three steps. First, find out working mechanism of forward and reverse mode methods and do some basic comparison experiments between hand code method, forward method and reverse method. Decide whether AD is better and which mode to use. Second, try some simple verification methods on AD tool and think about possible functions that needs to be supported in the tool in order to meet the requirements of a verification problem. Third, fulfil the implementation of the AD tool and use it in bigger problems.

In order to commit the first step, I did some comparative experiments forward mode AD, reverse mode AD and manual-effort Jacobian method and shows the performance results. After that, I did analytical analysis on the performance of the three methods and use it to explain the performance in experiments.

Then I tried the forward mode AD tool provided by INTLAB library on some simple problems in circuit verification and figured out possible missing functions of existing AD tools which is specifically for out verification problems.

In the chapter of research proposal, I concluded what I have found in the project and proposed a possible M.Sc. research topic and its possible solution based on my observation. Basically, I want to implement an AD tool based upon **reverse mode** method and possibly the other two which can support functions for a large part of the analog circuit verification and use it to verify a big formal verification problem.

Chapter 2

Comparative Experiments on Forward Mode AD, Reverse Mode AD and Manual-effort Jacobian method

As is stated in the first chapter, there exists many possible methods in calculating the derivative of a function. Here, I choose manual-effort Jacobian method in calculating the revised sensitivity matrix as the method in comparison to forward-mode AD and reverse-mode AD. This is because hand code method produces same derivative results as AD methods, but numerical methods produce approximations which is a level lower than AD methods and symbolic methods produce symbolic solution which is a level higher than just derivative value at a specific point. One can see hand code method is more comparable to AD methods.

2.1 Experiment Description and Design

2.1.1 Calculating revised sensitivity matrix

Consider a continuous dynamic system given by its ODE:

$$\begin{aligned} \dot{x} &= f(x) \\ x(0) &= x_0 \end{aligned} \tag{2.1}$$

Here, $x \in \mathscr{R}^n$, $\dot{x}$ is the derivative of $x(t)$, f is a n-dimensional vector of functions of x and $x_0 \in \mathscr{R}^n$ stands for the initial state of the system. The set of ODE functions tells one the derivative value at specific states.

The Jacobian matrix of the system is calculated in this way:

$$J_{i,j}(f, x) = \frac{\partial f_i(x)}{\partial x_j} \tag{2.2}$$

Here, $J \in \mathscr{R}^{n \times n}$. One can see that it is the dimensions of both x and f make sure that there are n^2 elements in the Jacobian matrix for the ODE system.

The sensitivity matrix of a dynamical system shows how the states change in relation to the change in initial states. Knowing the dimension of the state vector, one can also easily figure out the dimension of the sensitivity matrix S, where:

$$S_{i,j}(t) = \frac{\partial x_i(t)}{\partial x_j(0)} \tag{2.3}$$

According the Jacobian method, the sensitivity matrix can be calculated in below way:

$$\begin{aligned} S(0) &= I \\ \dot{S} &= S \times J(f, x) \end{aligned} \tag{2.4}$$

Now, we may want to know the sensitivity of a state variable at a given time with respect to any variable at any former time, not just for the initial inputs. This way, we want to be able to calculate the revised sensitivity matrix Sr, where:

$$Sr_{i,j}(t_m, t_n) = \frac{\partial x_i(t_m)}{\partial x_j(t_n)} \tag{2.5}$$

We can calculate this matrix by a little change to the algorithm for calculating the original sensitivity matrix. The method is to take the s-tate at time t_n and integrate until time t_m. Another practise is to cal-culate $Sr(t_2, t_1), Sr(t_3, t_2),\ ...\ Sr(t_p, t_{p-1})$first and then calculate the re-vised sensitivity matrix between any two time point in $t_1,\ ...\ , t_p$, t_m and t_n by multiply all revised sensitivity matrix between them: $Sr(t_{m,n}) = Sr(t_{n+1}, t_n) \times Sr(t_{n+2}, t_{n+1}),\ ...\ \times Sr(t_m, t_{m-1})$. This is the approach I used here.

2.1.2 Design comparison experiments

My experiments to conduct here is to calculate the revised sensitivity matrix introduced above using all the three methods: forward mode AD, reverse mode AD and manual-effort Jacobian method.

The continuous dynamical system I used in my experiments is the famous Rambus Oscillator analog circuit. A typical 2-stage Rambus ring oscillator looks like below:

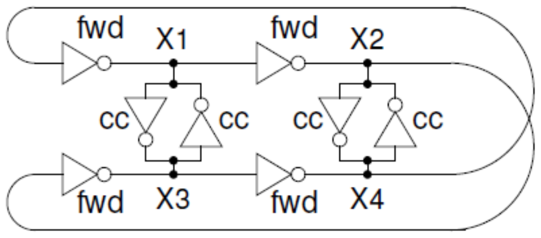

Figure 2.1: Two-Stage Rambus Ring Oscillator

One can see that a 2-stage Rambus oscillator has 4 state variables. Despite some details in the circuit of the oscillator, we use **tanh** function to model the oscillating behaviour of the oscillator:

$$\begin{aligned}
\dot{x_1} &= -tanh(g \cdot x_4) - x_1 + r \cdot (-tanh(g \cdot x_3) - x_1) \\
\dot{x_2} &= -tanh(g \cdot x_1) - x_2 + r \cdot (-tanh(g \cdot x_4) - x_2) \\
\dot{x_3} &= -tanh(g \cdot x_2) - x_3 + r \cdot (-tanh(g \cdot x_1) - x_3) \\
\dot{x_4} &= -tanh(g \cdot x_3) - x_4 + r \cdot (-tanh(g \cdot x_2) - x_4)
\end{aligned} \tag{2.6}$$

Here, g stands for some parameter of the 'forward' inverters and r stands for some parameter of the 'cross-coupling' inverters. I use $g = 4$ and $r = 0.5$ in my program.

hand code Jacobian method The **hand code** here means to calculate the derivative by hand. Here in our problem, we need to calculate the Jacobian matrix, which is a $n \times n$ matrix of functions in x. In example of the Jacobian matrix of the 2-stage Rambus oscillator above:

$$\mathbf{J} = -\begin{pmatrix} r+1 & 0 & rgsech^2(gx_3) & gsech^2(gx_4) \\ gsech^2(gx_1) & r+1 & 0 & rgsech^2(gx_4) \\ rgsech^2(gx_1) & gsech^2(gx_2) & r+1 & 0 \\ 0 & rgsech^2(gx_2) & gsech^2(gx_3) & r+1 \end{pmatrix} \tag{2.7}$$

Once we have the hand code version of the Jacobian of Rambus ring oscillator, we can use the method discussed in before section to integrate out the revised sensitivity matrix.

forward mode AD For the forward mode AD, I use the library provided in INTLAB. The main problem here is that integrator **ode45** in MATLAB does not support data structure except for vector of doubles. Therefore, I used the simplified integrator provided by Jijie Wei and did some modification to it.

The idea of the forward mode AD in INTLAB is to produce a new class instead of traditional vectors of doubles. The new class of objects overloads the basic arithmetic operators for double and some other usual functions, e.g. triangular functions, hyperbolic functions, logarithmic functions, exponential functions etc. The overloading function calculates the value and at the same time calculates the derivative results according to chain rules.

With the simplified integrator Ode_DP, we are able to integrate on the new objects and get the resulting sensitivity matrix in the new objects as we calculate for the function value. Special attention should be taken when initializing the input for integrator and initialize data structures in the integrator.

reverse mode AD For the reverse mode AD, I take the easy way to first hand code the 'program after compilation' and then use it for calculating the derivatives.

I did two examples for the reverse mode AD. One of them is as follows[5]. The example is a function of 3 variables.

$$\begin{aligned} f(a,b,c) &= (w-w_0)^2 \\ w &= ln(v^2+1)+cos(c^2-1) \\ v &= e^{u^2-1}+a^2 \\ u &= sin(ab)+cb^2+a^3c^2 \end{aligned} \tag{2.8}$$

In my implementation, given initial state $(a,b,c) = (a_0,b_0,c_0)$, the forward propagation stage calculates all immediate values in the forward sequence. In the backward adjoint stage, the calculation sequence in reversed order, suppose initially $(da, db, dc) = (0, 0, 0)$, is as follows:

$$\begin{aligned}
df &= 1 \\
dw &= 2(w - w_0)df \\
dc &= dc - 2csin(c^2 - 1)dw \\
dv &= \frac{2v}{v^2 + 1}dv \\
da &= da + 2adv \\
du &= 2ue^{u^2-1}dv \\
da &= da + (bcos(ab) + 3a^2c^2)du \\
db &= db + (acos(ab) + 2cb)du \\
dc &= dc + (b^2 + 2a^3c)du
\end{aligned} \tag{2.9}$$

One can see that reverse mode AD omits many unnecessary calculation of sub-function derivatives. The only difference when calculating derivatives for each variables happens in the last several calculation with the former calculations the same calculated only once.

In this way, I rewrite the code for Rambus oscillator as if it is compiled to calculate the reverse mode AD already. The revised code produce the function value and the function derivative as output, then give it to the integrator.

2.2 Main Technique Problem

All main technique problems lie in the proper use of the new class **gradient**.

Column vector and row vector When initializing a *gradient* object as column, the derivative of it is a two dimensional matrix; but when initializing a *gradient* object as row vector, the derivative of it is still a two dimensional matrix but stored in a three dimensional matrix with the first dimension fixed at 1. Special attention should be taken with this kind of design when necessary.

Data structure initialization When implementing some functions, I encounter some situation when I need to initialize new variables in the middle of code. When this new variable is initialized carelessly as double and is assigned as some *gradient* value, there exists no mechanism to convert the gradient type to double. Therefore, this assignment can not be realised.

My solution is to read through the code and figure out the structure I need here and initialize the *gradient* vectors or matrices accordingly. This requires lots of effort.

Subscript reference When multiple *gradient* objects are produced in a vector, the derivative matrix in the object becomes 3 dimensional matrix and the subscript reference becomes different and there is a difference between column vector of gradient objects and row vector of gradient objects.

2.3 Experiment Results

I conducted two sets of experiments. The first experiment increases the number of stages in the Rambus ring oscillator and the second experiment increases the number of time intervals when calculating the revised sensitivity matrix.

2.3.1 Runtime comparison

For the first set of experiments, I increase number of stages of the Rambus oscillator from 8 to 16 and fix number of time intervals at 32. The revised sensitivity matrix I'm calculating is $S(t_4, t_1)$. Below table shows the results:

stages	8	10	12	14	16
FWD AD/s	5.025829	5.114733	5.238651	5.286655	5.926259
RVS AD/s	0.693135	0.718496	0.910159	0.910864	0.981068
JCB/s	0.857869	0.866685	0.959758	0.953296	1.201402

Table 2.1: First experiment with number of stages increasing.

Below is a graph showing the change potential:

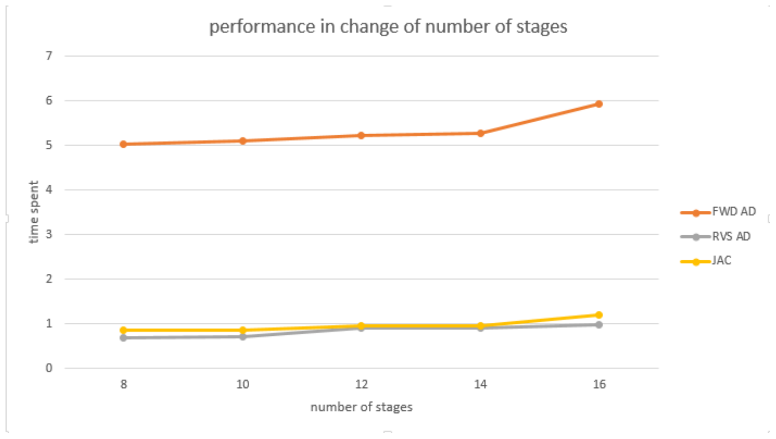

Figure 2.2: Performance in change of number of stages

In order to see the comparison between Jacobian method and reverse mode AD clearly:

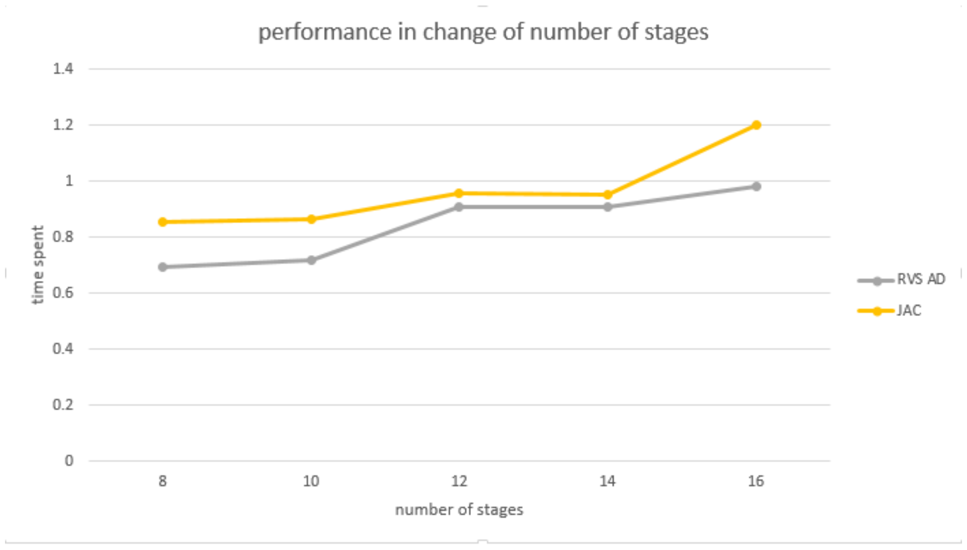

Figure 2.3: Performance in change of number of stages, only JAC and RVS AD

The reason I start at 8 stages is because for stages smaller than 8, the integrator's step length change a lot which makes it difficult to see the influence of stage number on derivative calculation.

For the second set of experiments, I increase number of intervals from 4 to 64 and fix number of stages at 4. The revised sensitivity matrix I'm calculating is also $S(t_4, t_1)$. Below table shows the results:

interval num	4	8	16	32	64
FWD AD/s	0.821029	1.435060	2.687066	5.155991	10.281717
RVS AD/s	0.108053	0.187385	0.346121	0.672408	1.316860
JCB/s	0.137884	0.232701	0.435869	0.834923	1.637169

Table 2.2: Second experiment with number of intervals increasing.

Below is a graph showing the change potential:

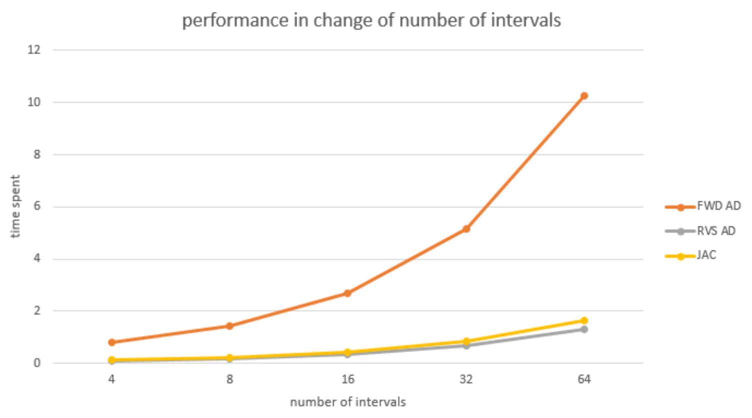

Figure 2.4: Performance in change of number of intervals

2.3.2 Analytical analysis

Suppose the number of stages is n(Therefore, the number of variables is $2n$ and the number of functions is $2n$, too.), the number of intervals is m. Suppose the cost of a function evaluation is F and the number of time intervals in the integrator is v. The analysis is based on the assumption that the number of steps in integrator doesn't change much for each set of parameters.

for forward AD Suppose $p = 4$ is the number of variables per function. The number of partial calculation in forward AD:

$$partial_{FWD} = O(n^2 * p * v * m) = O(4vmn^2) \tag{2.10}$$

for reverse AD The number of partial calculation in reverse AD:

$$partial_{RVS} = O(n^2 * v * m) = O(vmn^2) \tag{2.11}$$

for Jacobian The number of partial calculation in Jacobian method:

$$partial_{JCB} = O(n^2 * v * m) = O(vmn^2) \tag{2.12}$$

For the coarse analysis, we see forward AD's running time must be 4 times larger than reverse AD and Jacobian and the running time for all three methods change linearly with number of time intervals, which matches what we see in the experiment result. But there are also other inconsistency: the running time for all three methods in change to number of stages seems more linearly than quadratically.

I think here, two factors need to be taken into consideration:

a. Steps taken by integrator are the same for different methods, but are different for changing parameters. This factor may influence the result a lot.

b. The analysis is still vague, some small factors are not taken into consideration. These small factors may not be a great deal to the whole landscape, but can be influential to the small set of experiments I did.

Another observation is that there is slightly difference between reverse mode AD and Jacobian method. I think the reason for reverse mode AD to be faster goes into the detail of how my Jacobian matrix is calculated and how the reverse mode derivative is implemented in my code. For coding convenience, I use sparse matrix for Jacobian calculation, but didn't use it for reverse mode AD. I believe sparse matrix operations will be more expensive than ordinary operations although it takes less memory, and if I make it the same, their running time will be even more closer. But for coding convenience and automation, reverse mode AD saves one the time to hand code the Jacobian matrix.

2.3.3 Other attempts on small verification problems

I also did two experiments on small verification problems using the provided library in INTLAB of forward mode AD. In the process, I find some important functions to be supported in the field of verification problems, which fulfils my understanding of a complete AD tool.

One of them is to use the tool to find a possible oscillating trajectory for the Rambus oscillator. The function not supported is *spline* and the sequence of functions called by it:*ppval* and *mkpp*.

The other of them is to use the differentiation tool to find the period of the vdp oscillator. I find that the backslash root finding is not supported by the INTLAB tool.

These two examples give me possible future research ideas.

2.4 Discussion and Conclusion

According to these comparison experiment and analysis, we can see that the reverse mode AD has the best performance, hand code Jacobian method is close to it next and forward mode AD takes more time than the two method to an order.

In comparison of reverse mode AD and hand code Jacobian method, they both have their drawbacks. Obviously, hand code Jacobian takes a lot of human effort in figuring out the Jacobian of a ODE system. If the scale of system is large and sophisticated, it becomes a huge task for the coder and it becomes easy to make mistakes in the process. In this aspect, reverse mode AD is better than Jacobian method. However, one may say that in my experiment, I didn't take the time of compiling out the new function into consideration. Whereas, the time for a computer to 'think' can never be longer than a human. And once it is compiled for a system, this function can be used forever. Therefore, the time spent is not a big deal. One can even give way to exponential compilation time.

Chapter 3

Research Proposal

3.1 Expected Outcome

The experiments I've done shows a possibility of integrating automatic differentiation tools into the verification of continuous dynamical systems. One can see that if reverse mode AD is applied, some amount of time can be saved and a large amount of time can be saved in deriving the Jacobian of a system. Looking at this aspect, it's worse trying.

I think it is important to have a broad goal first, in case any problems happen and I have to switch to other sub-problem, I still have the broad goal. Generally speaking, I expect in the end of my M.Sc. research, I have a fast enough workable AD tool specifically designed for analog circuit formal verification, if possible I want to embed it into COHO.

In detail, I want to implement a AD tool that can take forward, reverse mode AD and hand code Jacobian into consideration, deciding which method to use itself. And then use it on a big enough verification problem to help verification.

Based on my experiments, I think there are several aspects to be done:

a. forward mode AD: Add in new functions to support verification-needed functions.

b. reverse mode AD: Need a compiler for the generation of new function.

c. combination: Need a criterion for deciding which method to use.

d. a verification problem: I expect this problem to be related to part of the PLL.

3.2 Resources Required

I might need INTLAB for its library on forward mode AD. I also need JAVA to write the compiler.

3.3 Key Uncertainties

difficulty in enriching library of forward mode AD Sometimes, it can be very difficult to write a correct function for our use. Therefore, some tricks need to be taken when thinking about calculation of a function. This makes forward mode AD not that 'automatic', because more human-effort derivation must be made.

implementation of reverse mode AD There exists two ways of reverse mode AD implementation. One is called Source Transformation. This is the method I want to use in the future. This method basically means to reorder the source code and 'compile' it to a new code in the reverse order that can calculate the derivatives. However, the other method, Operator Overloading is also a possible method. I might want to try this too, but only time permitted.

whether or not take use of existing software I hope to implement the code myself first. But existing software have more improvements and optimization. Therefore, it's difficult to decide which. However, problems might happen with existing software, e.g. it may be incompatible to my specific problem in formal verification. It's a fine line to take.

An example of existing software to take use: FADBAD++ is a free software that implements forward mode AD, reverse mode AD and Taylor methods using C++ templates and operator overloading.

compatibility with COHO The compatibility problem can happen at any small piece of code. One of the problem I can think of for now is about the forward mode AD provided by INTLAB. This problem is also described in former chapter about main technique difficulties. If new data structure is initialized in the middle of code and assigned to a 'gradient' object, an error will be raised. Thus, to maintain the compatibility, I need a way to cast the new variable to 'gradient' class in a proper way without change my original COHO code to much. But I don't know if it's possible or not.

Bibliography

[1] RWTH Aachen University Andre Vehreschild, Institute for Scientific Computing. Adimat: Automatic differentiation of matlab. `http://www.sc.rwth-aachen.de/adimat/`.

[2] Christian H. Bischof, Paul D. Hovland, and Boyana Norris. On the implementation of automatic differentiation tools. *Higher Order Symbol. Comput.*, 21(3):311–331, September 2008.

[3] Andreas Griewank. On automatic differentiation. In *IN MATHEMATICAL PROGRAMMING: RECENT DEVELOPMENTS AND APPLICATIONS*, pages 83–108. Kluwer Academic Publishers, 1989.

[4] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation, 1991.

[5] C. Homescu. Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance. *ArXiv e-prints*, July 2011.

[6] H. G. Kahrimanian. Analytical differentiation by a digital computer. *M.Sc. Thesis, Temple University*, 1953.

[7] John F Nolan. Analytical differentiation on a digital computer. *M.Sc. Thesis, Massachusetts Institute of Technology*, 1953.

[8] Siegfried M. Rump. Intlab-interval laboratory. `http://www.ti3.tu-harburg.de/rump/intlab/`.

[9] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, January 1980.

[10] TOMLAB. Tomlab /mad. `http://tomopt.com/tomlab/products/mad/`.

[11] Yu. M. Volin and G. M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis. *Computers and Mathematics with Applications*, 11:1099–1114, 1985.