Algorithmic Differentiation

Math 337 Final Project Handout

Andy Reagan

April 29, 2014

Contents

1 Introduction

The key question that we attempt to answer when modeling real-world phenomena is: how sensitive is the output of model with respect to the input values and parameters? Most of the methods that we have developed in the class rely on finite differences to answer this question, and hence to make accurate predictions. We judge the accuracy of these methods based on how well they approximate the Taylor series of the given function, giving rise to their local truncation error. Accuracy is not the sole desired property of a numerical scheme, but once convergence to the true solution can be achieved, we seek the most efficient and most accurate determination of this solution.

A promising tool to answer the above question, with arbitrary accuracy, support for extended precision arithmetic, and the ability to apply to comlex, large-scale models is Algorithmic Differentiation. My presentation today outlines the basics of Algorithmic Differentation (AD) technology. Some of the literature refers to AD as Automatic Differentiation, but in my experience the differentiation is still far from automatic.

The core functionality of AD relies upon the fact that any computer program can be broken down into a sequence of elementary operations, for which the derivative can be computed easily. One limitation of this approach is that we are restricted to a range of functions that can be broken down this way, although we will see that this is not as limiting in practical implementations. There are many ways to potentially compute derivatives algorithmically, and we will discuss source code transformation, derivative compilers, and operator overloading techniques.

1.1 Taylor Method

Going back to Day 1, we are interesting in solving problems of the form of Eq (0.1) of the notes:

$$y'(x) = f(x, y).$$

Instead of taking the finite difference, we next consider the Taylor series expansion:

$$\begin{aligned} f(x,y) = f(x_0,y_0) &+ (\Delta x f_x(x_0,y_0) + \Delta y f_y(x_0,y_0)) \\ &+ \frac{1}{2!}\left((\Delta x)^2 f_{xx}(x_0,y_0) + 2\Delta x \Delta y f_{xy}(x_0,y_0) + (\Delta y)^2 f_{yy}(x_0,y_0)\right) + \dots \end{aligned}$$

The computation of the forms of f_x, f_y and so forth requires significantly effort from a computer, and will typically be cumbersome. Therefore, evaluating them will be slow. Algorithm differentiation overcomes this difficulty. We will discuss this in more detail in the following section.

2 Algorithmic Differentiation Basics

First I will note other possibilities for computing derivatives and jacobians, namely: symbolic differentiation, hand-coding, and divided differences. In line with the goals of Bischof and Bucker (2000), we seek to evaluate derivatives in a way that has the following properties:

- **Reliability** The computed derivatives should be accurate to the IEEE floating point arithmetic of the computer.
- **Computational Cost** Computing the derivatives is often the main computational cost. The amount of memory and operations should be bounded.
- **Scalability** The approach should give correct results for a 1-line formula as well as a 100,000-line code.
- **Human Effort** A user should not spend much time preparing a code for differentiation, in particular where code changes frequently.

Handcoding, divided-difference approximations, and symbolic manipulators fall short with respect to the previously mentioned criteria. The main drawbacks of divided-difference approximations are their numerical unpredictability and their computational cost. In contrast, both the handcoding and symbolic approaches suffer from a lack of scalability and require considerable human effort Bischof and Bucker (2000).

To simplify the discussion, I adopt the notation of Jorba and Zou (2005). If $a : t \in I \subset \mathbb{R} \to \mathbb{R}$ is a smooth function, denote its normalized n-th derivative to the value

$$a^{[n]}(t) = \frac{1}{n!} a^{(n)}(t).$$

Assume now that $a(t) = F(b(t), c(t))$ and that we know the values of $b^{[j]}(t)$ and $c^{[j]}(t)$, $j = 0, \ldots, n$ for a given t. I will prove the following on the board:

Proposition 2.1. *If the function a and b are of class C^n, and $\alpha \in \{\mathbb{R} - 0\}$, we have*

1. *If $a(t) = b(t) \pm c(t)$, then $a^{[n]}(t) = b^{[t]}(t) \pm c^{[t]}(t)$.*
2. *If $a(t) = b(t)c(t)$, then $a^{[n]}(t) = \sum_{j=0}^{n} b^{[n-j]}(t)c^{[j]}(t)$.*

3. *If* $a(t) = \frac{b(t)}{c(t)}$, *then* $a^{[n]}(t) = \frac{1}{c^{[0]}(t)} \left[b^{[n]}(t) - \sum_{j=1}^{n} c^{[j]}(t) a^{[n-j]}(t) \right]$.

4. *If* $a(t) = b(t)^{\alpha}$, *then* $a^{[n]}(t) = \frac{1}{n b^{[0]}(t)} \sum_{j=0}^{n-1} (n\alpha - j(\alpha+1))\, b^{[n-j]}(t) a^{[j]}(t)$.

5. *If* $a(t) = e^{b(t)}$, *then* $a^{[n]}(t) = \frac{1}{n} \sum_{j=0}^{n-1} (n-j)\, b^{[n-j]}(t) a^{[j]}(t)$.

6. *If* $a(t) = \ln b(t)$, *then* $a^{[n]}(t) = \frac{1}{b^{[0]}(t)} \left[n^{[n]}(t) - \frac{1}{n} \sum_{j=1}^{n-1} (n-j)\, b^{[j]}(t) a^{[n-j]}(t) \right]$.

7. *If* $a(t) = \cos c(t)$ *and* $b(t) = \sin c(t)$, *then*

$$a^{[n]}(t) = -\frac{1}{n} \sum_{j=1}^{n} j b^{[n-j]}(t) c^{[j]}(t), \qquad b^{[n]}(t) = \frac{1}{n} \sum_{j=1}^{n} j a^{[n-j]}(t) c^{[j]}(t).$$

Remark It is possible to derive similar forumulas for other functions, line inverse trigonometric functions.

Corollary 2.2. *The number of arithmetic operations to evaluate the normalized derivation of a function up to order* n *is* $O(n^2)$.

I will prove this on the board as well.

2.1 Short Example

I base my example on that of Jorba and Zou (2005), and apply the rules of Proposition 2.1 to the Van der Pol equation:

$$\begin{aligned} x' &= y \\ y' &= (1 - x^2)y - x \end{aligned}$$

We decompose this function in the following elementary operations:

$$
\begin{aligned}
u_1 &= x \\
u_2 &= y \\
u_3 &= u_1 u_1 \\
u_4 &= 1 - u_3 \\
u_5 &= u_2 u_4 \\
u_6 &= u_5 - u_1 \\
x' &= u_2 \\
y' &= u_6
\end{aligned}
$$

And applying Proposition 2.1 to the above, we have

$$
\begin{aligned}
u_1^{[n]}(t) &= x^{[n]}(t) \\
u_2^{[n]}(t) &= y^{[n]}(t) \\
u_3^{[n]}(t) &= \sum_{i=0}^{n} u_1^{[n-i]}(t) u_1^{[i]}(t) \\
u_4^{[n]}(t) &= 1 - u_3^{[n]}(t) \\
u_5^{[n]}(t) &= \sum_{i=0}^{n} u_2^{[n-i]}(t) u_4^{[i]}(t) \\
u_6^{[n]}(t) &= u_5^{[n]}(t) - u_1^{[n]}(t) \\
x^{[n+1]}(t) &= \frac{1}{n+1} u_2^{[n]}(t) \\
y^{[n+1]}(t) &= \frac{1}{n+1} u_6^{[n]}(t)
\end{aligned}
$$

2.2 High accuracy computations

To achieve a given level of accuracy, we can choose both the step size h and the order p. Of course, we need to be cognizant of the radius of covergence of the power series we create and the effect that this has on the stability of the step. Since we can control both order and step size, we attempt the dual goals of the desired accuracy and efficient computation.

3 Using Taylor 1.4

Taylor 1.4 is a package that is specifically designed to use AD to integrate systems of equations. The documentation for the package is excellent, in both an extended paper Jorba and Zou (2005), and an online appendix (`http://www.ma.utexas.edu/users/mzou/taylor/manual/manual.html`). I downloaded and built the software using `gcc`, specifically

```
VACC happy% gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/gpfs1/arch/x86_64/gcc4.8.1/libexec/gcc/x86_64-unknown-linux-gnu
    /4.8.1/lto-wrapper
Target: x86_64-unknown-linux-gnu
Configured with: ./configure --prefix=/gpfs1/arch/x86_64/gcc4.8.1 --with-mpc=/gpfs1/arch
    /x86_64 --with-mpc-include=/gpfs1/arch/x86_64/include
Thread model: posix
gcc version 4.8.1 (GCC)
```

I specify the Kepler two-body problem as

```
/* ODE specification: tbp
   Two body problem, discretized into
   a set of four ODE's.  */

diff(x1, t)= x3;
diff(x2, t)= x4;
diff(x3, t)= -x1/((x1^2+x2^2)^(3/2));
diff(x4, t)= -x2/((x1^2+x2^2)^(3/2));

ecc = 0.6; /* 1-ecc, 0, 0, -sqrt((1+ecc)/(1-ecc)); */

initial_values= 0.4, 0, 0, -2;
start_time= 0.0;
stop_time = 500.0;
absolute_error_tolerance = 0.1e-16;
relative_error_tolerance = 0.1e-16;
```

and build the executable with

```
# make the executable
./taylor -name tbp -o tbp.c -jet -step tbp.in
./taylor -name tbp -o taylor.h -header
./taylor -name tbp -o main_tbp.c -main_only tbp.in
gcc -O3 main_tbp.c tbp.c -lm -s
```

4 Comparison to Symplectic Methods

We perform tests with the Kepler two-body problem (Eq (5.30) of the notes):

$$q'' = -\frac{q}{(q^2+r^2)^{3/2}} \quad , \qquad r'' = -\frac{r}{(q^2+r^2)^{3/2}};$$

where q and r are the cartesian coordinates of a certain radius vector relative to the center of mass of the bodies.

Although we could solve this directly, we first put the problem into a system of four ODE's:

$$\begin{aligned} z' &= -\frac{q}{(q^2+r^2)^{3/2}} \\ w' &= -\frac{r}{(q^2+r^2)^{3/2}} \\ q' &= z \\ r' &= w. \end{aligned}$$

For comparison, we apply the Verlet-1 and -2 methods to the Kepler two-body problem. We start with $h = 0.1$ and $t_{\max} = 500$, and decrease h to preserve the Runge-Lenz vector. As the initial condition, use

$$q(0) = 1 - \text{ecc}, \quad r(0) = 0, \quad Q(0) = 0, \quad R(0) = \sqrt{\frac{1+\text{ecc}}{1-\text{ecc}}} \quad \text{for ecc} = 0.6,$$

which corresponds to the exact solution being an elipse with eccentricity 0.6. As we observed in HW5, Bonus 2, the Verlet method loses the orientation of the orbit for large h. Here are three plots for varying h, starting at 0.1 down the 0.005, where the orbit appears to stick around. I didn't spend a long time optimizing the MATLAB code, and it takes quite a long time to run (without making plots):

```
>> tic; andy_hw05_prb12; toc;
Elapsed time is 16.450040 seconds.
```

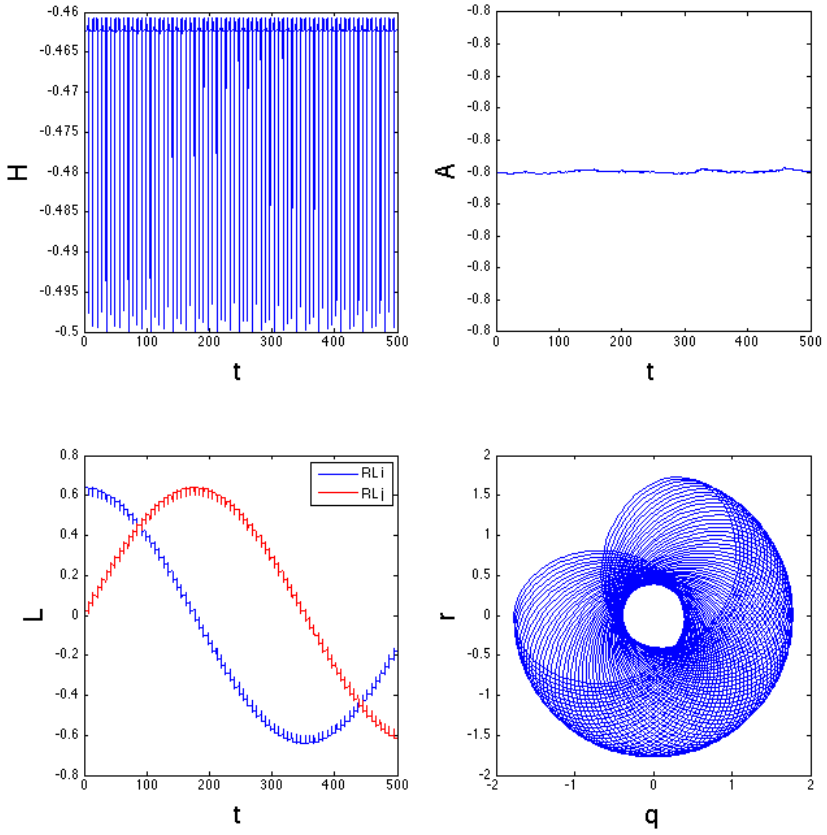

Figure 1: The numerical solution for $h = 0.1$. In the top left panel we plot the Hamiltonian, top right the angular momentum, bottom left the Runge-Lenz vector, and bottom right the phase space plot.

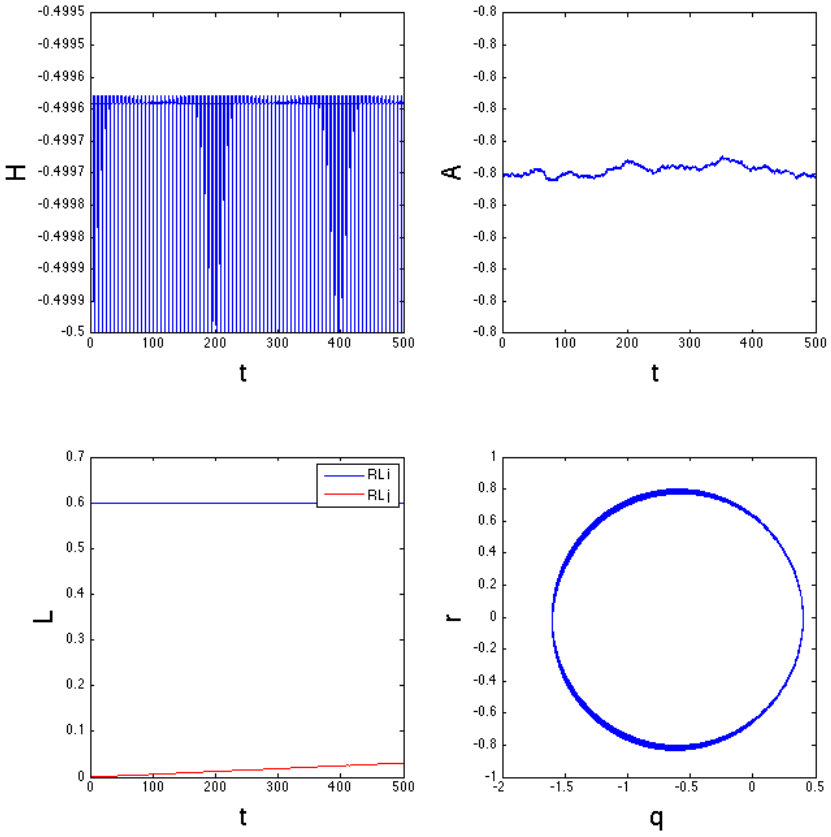

Figure 2: The numerical solution for $h = 0.01$. In the top left panel we plot the Hamiltonian, top right the angular momentum, bottom left the Runge-Lenz vector, and bottom right the phase space plot.

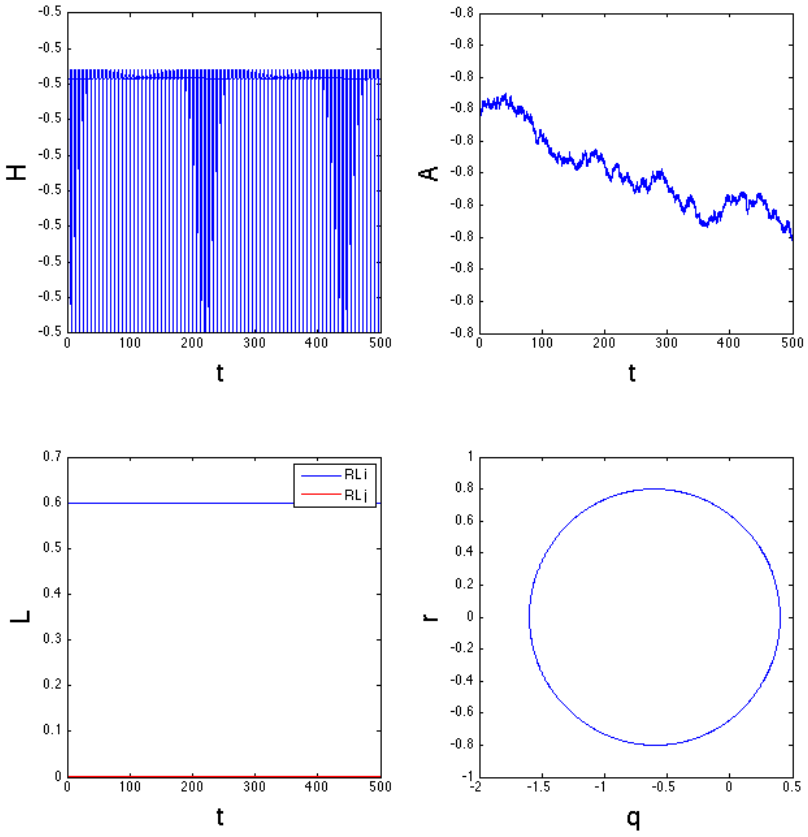

Figure 3: The numerical solution for $h = 0.001$. In the top left panel we plot the Hamiltonian, top right the angular momentum, bottom left the Runge-Lenz vector, and bottom right the phase space plot.

To visualize the output of the ANSI C routine generated by Taylor 1.4 we import the data file into Python, and plot the orbit. The calculation is nearly immediate with Taylor 1.4:

```
VACC happy% time ./a.out >> output.txt

real 0m0.020s
user 0m0.018s
sys  0m0.001s
```

The plot of this solution is:

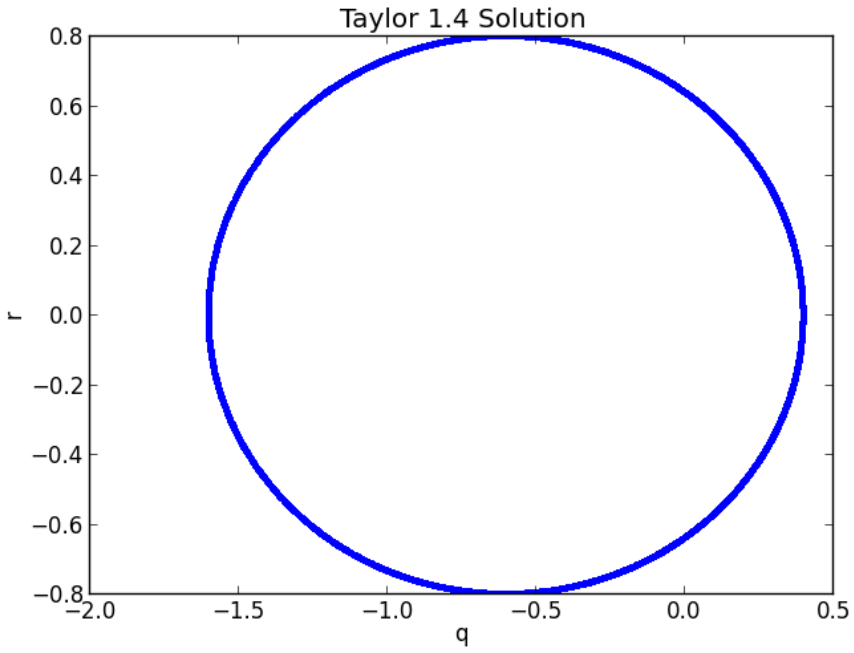

Figure 4: Solution with the Taylor method.

Directly comparing the endpoint of the calculation, for MATLAB we have:

```
>> tvec(end-271)

ans =

     4.997290000000000e+02

>> resultscell{1}(:,end-271)

ans =

  -1.591413885966831
   0.104204021609059
```

```
   0.082063151415239
   0.497324232605820
```

and from Taylor we have:

```
q r Q R t
-1.533946878017245 -0.285929478345275 -0.229056106562621 0.478834122266450
    498.933289591512334
-1.591469349241215 -0.104272042704416 -0.081723967848449 0.497325616300367
    499.304315937556282
-1.590895914077501 0.107704150232526 0.084432121351941 0.497145226861730
    499.729050276503926
```

Of course, we can't really be sure that this means that the solutions are close, because the orbit is periodic (one could have simply lapped the other!).

5 Conclusions

It took a long time to figure out how to get the code to run, again confirming that the process is not automatic, however the speed with which the high accuracy simulation can be coded and computed is impressive. As AD continues to gain popularity, the use cases will grow beyond those in traditional optimization. With the ability to solve equations with arbitrary accuracy (extended precision arithmetic), more research may find applications for the Taylor method.

References

Barrio, R. (2005). Performance of the taylor series method for odes/daes. *Applied Mathematics and Computation 163*(2), 525–545.

Bischof, C. and H. M. Bucker (2000). Computing derivatives of computer programs. *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, NIC Series 3*, 315–327.

Gay, D. M. (2006). Semiautomatic differentiation for efficient gradient computations. In *Automatic differentiation: applications, theory, and implementations*, pp. 147–158. Springer.

Griewank, A. (2003). A mathematical view of automatic differentiation. *Acta Numerica 12*, 321–398.

Jorba, À. and M. Zou (2005). A software package for the numerical integration of odes by means of high-order taylor methods. *Experimental Mathematics 14*(1), 99–117.

Pryce, J. D. and E. M. Tadjouddine (2008). Fast automatic differentiation jacobians by compact lu factorization. *SIAM Journal on Scientific Computing 30*(4), 1659–1677.

Verma, A. (2000). An introduction to automatic differentiation. *CURRENT SCIENCE-BANGALORE- 78*(7), 804–807.

Walther, A. and A. Griewank (2012). Getting started with adol-c. *Combinatorial Scientific Computing*, 181–202.