Using Automatic Differentiation to Implement a Family of Material Models

Robert Mastragostino and W. Spencer Smith

January 2015

Eng 1 Technical Report

McMaster University

Contents

Abstract

Virtlab is a virtual material testing laboratory that works with a variety of material models. These materials specify functions as part of their definition, some of which must be differentiated for use in the simulator. Virtlab can be extended with new models by the user, and as a result all functions needed will not be known at development time. To prevent the user from calculating large error-prone expressions by hand, general purpose methods for calculating these derivatives must be considered. Previously, Maple's symbolic computation and code generation facilities were used for this task. However, this creates a dependency on a commercial software package and an associated maintenance challenge if the Maple interface should be modified in the future. The goal of the current project is to use FADBAD++, an automatic differentiation package, for the same task and to compare the two methods. The refactored code proved to be faster in some cases, and slower in others. Enough changes were made to the underlying code that the particular reasons for this are unknown, and a more detailed analysis is needed to determine the direction of future development.

1 Introduction to Virtlab

The focus of this project is on the Virtlab virtual material testing simulation software [2, 5, 6]. Virtlab simulates various material tests using a variety of constitutive equations and can be extended to easily include new models and materials.

This ability to implement user-created models not known at development time necessitates general purpose methods for computing various quantities, especially derivatives. This motivated the use of Automatic Differentiation, which is described in Section 2. The use of Automatic Differentiation resulted in necessary changes to the interfaces used to define materials and run experiments, which is described in Section 3. The results of these modifications are discussed in the Section 4 and the conclusions are given in Section 5.

1.1 Experimental Setup

Virtlab uses the finite element method on a single hexahedral element to simulate an experiment. The state of the element is defined by its eight vertices, which have various boundary conditions (displacements and/or forces) applied to them, according to the experiment being run. These boundary conditions can be changed over the course of the experiment, allowing for more complicated tests.

Tests can generally be classified as either displacement controlled or load controlled, though Virtlab allows for arbitrary combinations of these, if desired. Experiments will typically require various parameters, which must be supplied in the input file (Section 3.2). The test used for the experiments done in this report was a uniaxial extension test in the x-direction, as shown in Figure 1.

All motion in the x-direction is controlled: the nodes in the yz plane are fixed in the x-direction, while the other nodes are pulled along the x-axis at a fixed displacement rate. Movement is otherwise unconstrained.

1.2 Physical Background

The presentation in this section is based on the physics presented in [2]. Assuming other body forces (gravity, inertia, etc.) are ignored, the block must always satisfy the equilibrium equation:

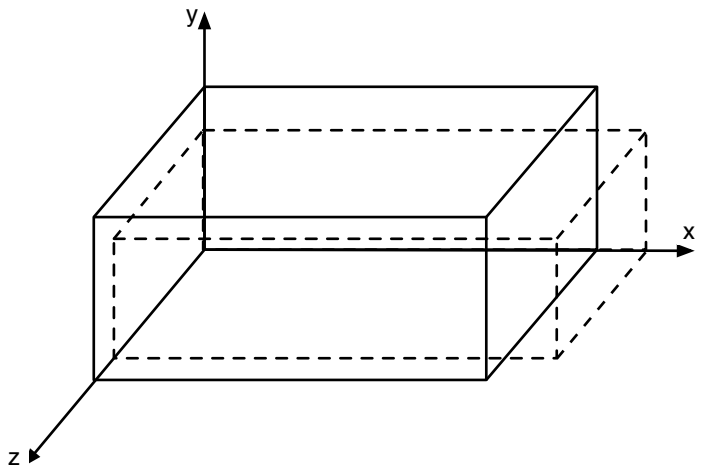

Figure 1: Uniaxial extension test (from [2]).

$$\boldsymbol{L}^T \boldsymbol{\sigma} = 0 \tag{1}$$

where the differential operator $\boldsymbol{L}^T$ is defined as

$$\boldsymbol{L}^T = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \tag{2}$$

The stress tensor is represented here by a 6 dimensional vector as

$$\boldsymbol{\sigma} = [\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{xz}]^T \tag{3}$$

where the first three stress components are normal to the block faces, and the last three are the shear components.

The equilibrium equation is not enough to solve for the unknown displacements and stresses. To have enough equations for a solution, one can introduce the constitutive equation, which specifies how the stresses respond to deformation. To represent deformation we introduce the strain tensor:

$$\boldsymbol{\varepsilon} = [\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \gamma_{xy}, \gamma_{yz}, \gamma_{xz}]^T \tag{4}$$

whose components are defined analogously to those in the stress tensor.

Virtlab is intended to model both elastic and viscoplastic behaviour. Elastic behaviour exists when the stress tensor is proportional to the strain tensor, defined by

$$\Delta\boldsymbol{\sigma} = \boldsymbol{D}\Delta\boldsymbol{\varepsilon}^e \tag{5}$$

$\Delta\boldsymbol{\varepsilon}^e$ is the most recent change in the elastic strain. The $\boldsymbol{D}$ matrix is defined as follows:

$$\boldsymbol{D} = \chi \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 \frac{1-2\nu}{2} & \end{bmatrix} \tag{6}$$

where $\chi = \frac{E}{(1+\nu)(1-2\nu)}$. E is Young's Modulus and ν is Poisson's ratio.

Viscoplastic behaviour occurs when stress (specifically the deviatoric stress tensor) instead relates to the strain *rate*. The deviatoric stress tensor $\boldsymbol{s}$ is defined as $\boldsymbol{s} = \boldsymbol{\sigma} - [\sigma_m, \sigma_m, \sigma_m, 0, 0, 0]^T$, where $\sigma_m = \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz})$. The relationship between the viscoplastic strain rate and this tensor is assumed to be proportional:

$$\dot{\boldsymbol{\varepsilon}}^{vp} = \lambda\boldsymbol{s} \tag{7}$$

Since we want Virtlab to handle materials that may express both elastic and viscoplastic behaviour, we use the superposition principle and define

$$\Delta\boldsymbol{\varepsilon} = \Delta\boldsymbol{\varepsilon}^e + \Delta\boldsymbol{\varepsilon}^{vp} = \Delta\boldsymbol{\varepsilon}^e + \Delta t\dot{\boldsymbol{\varepsilon}}^{vp} \tag{8}$$

Here $\Delta\boldsymbol{\varepsilon}$ is the total strain from both effects, and Δt is the timestep. We can combine Equation 5 and Equation 8 to find

$$\Delta\boldsymbol{\sigma} = \boldsymbol{D}(\Delta\boldsymbol{\varepsilon} - \Delta t\dot{\boldsymbol{\varepsilon}}^{vp}) \tag{9}$$

This equation gives us the stress change in terms of a given strain increment. However, different materials can have very different viscoplastic strain rates, and for our purposes this is what sets material models apart. A good family of models that can represent a wide variety of elastic and viscoplastic behaviours has been presented by Perzyna [3]:

$$\dot{\boldsymbol{\varepsilon}}^{vp} = \lambda \frac{\partial Q}{\partial \boldsymbol{\sigma}} = \gamma < \phi(F) > \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{10}$$

Where Q is the viscoplastic potential, γ is a fluidity parameter, F is the yield function and

$$< \phi(F) >= \begin{cases} 0 & F \leq 0 \\ \phi(F) & F > 0 \end{cases} \tag{11}$$

$\phi(F)$ determines how the material behaves after it yields, and $\dot{\boldsymbol{\varepsilon}}^{vp}$ is 0 if the material has not yielded (i.e. it is entirely elastic within the yield surface). It is the yield function F that lets the equation represent elastic, viscous, viscoplastic and viscoelastic effects. Q depends entirely on $\boldsymbol{\sigma}$, while F depends on both $\boldsymbol{\sigma}$ and a hardening parameter $\kappa(\boldsymbol{\varepsilon}^{vp})$. This lets us model strain hardening and softening materials as well. Each component of this equation is individual to a given material model. So to specify a model the user must give Virtlab expressions for F, Q, ϕ, κ and the constant γ. These expressions are part of the input to Virtlab.

An informative example is the specification of a power law fluid:

$$\begin{aligned} F(\boldsymbol{\sigma}, \kappa) &= \sqrt{3J_2} \\ Q(\boldsymbol{\sigma}) &= \sqrt{3J_2} \\ \phi(F) &= F^m \\ \kappa(\boldsymbol{\varepsilon}^{vp}) &= 0 \\ \gamma &= A \end{aligned} \tag{12}$$

where J_2 is the second invariant of the deviatoric stress tensor, and A and m are material constants. While this is a relatively simple example, the functions involved can be of widely varying and complicated forms (especially D and Q), requiring a method that can work without knowing the particular form of these functions.

2 Automatic Differentiation

As the previous section suggests, many derivatives need to be calculated during the simulation of a given experiment. Often these are derivatives of functions that are specific to a material model. Since Virtlab should be capable of easily incorporating new models, these will not all be known at development time. A generic method for calculating these derivatives

is desired so that the end user does not have to compute them by hand. Previously a tool called MatGen [2] was implemented for this purpose using Maple's symbolic differentiation and code generation facilities. To create a material model with MatGen, a user specifies definitions of $F, Q, \kappa, \phi, \gamma$ and any needed material constants. The elastic modulus and Poisson's ratio are material properties that are always included, since they are needed to define the elastic constitutive matrix $\boldsymbol{D}$. Functions were specified in a high-level domain-specific language, featuring many macros that are common in material definitions, such as J_2. Maple then take these definitions, calculates the various derivatives, and uses this information to generate a C++ file describing the material that MatCalc can then use to simulate an experiment.

The code generation in MatGen was done in a naive way, such that the derivatives were included in the file (and therefore recalculated) every time they were needed, rather than every time they were changed. This method also subjected Virtlab to versioning issues: when Maple 15 was used instead of Maple 14, the generated code failed to give proper answers. It also created a large dependency on commercial software: only a small fraction of Maple's tools are needed for this project, so this dependency may unnecessarily limit future usage of Virtlab. These issues justify a search for a new approach. In the current work, the new approach involves automatic differentiation and the FADBAD++ package [7].

2.1 An Overview of Automatic Differentiation

Automatic differentiation (AD) is a method of computing derivatives without creating large unwieldy expressions (like symbolic differentiation) or introducing numerical inaccuracies (like numeric differentiation). To implement AD we used the FADBAD++ program. FADBAD stands for "Forward Automatic Differentiation and Backward Automatic Differentiation," which represents the main two types of AD that FADBAD++ is capable of. Forward AD was used for the majority of the project. To see why and how AD works, we give a brief introduction to dual numbers.

To construct the dual numbers, one augments the real number system ($\mathbb{R}$) with a new element ϵ, such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. The usual rules of algebra otherwise apply [1]. A dual number can then be represented as

$$x + x'\epsilon \tag{13}$$

where $x, x' \in \mathbb{R}$, while ϵ is not and is instead a new construction. The goal of dual numbers in this context is to capture the behaviour given by truncating the Taylor series to first order, which is the motivation for the definition of ϵ. This implies that if we consider the x' to be the derivative of x, the standard rules of differentiation follow:

$$(x + x'\epsilon) + (y + y'\epsilon) = (x + y) + (x' + y')\epsilon \tag{14}$$

Here we see that the sum of derivatives is the derivative of the sum.

The product rule similarly arises naturally from multiplication, as follows:

$$(x + x'\epsilon) \cdot (y + y'\epsilon) = xy + xy'\epsilon + x'y\epsilon + x'y'\epsilon^2 = (xy) + (xy' + yx')\epsilon \tag{15}$$

The real components are multiplied normally, while the computation of the new dual component follows the same pattern as for the product rule. Second order behaviour is ignored, thanks to the definition of ϵ.

The quotient rule can also be determined using algebraic manipulation, as follows:

$$\frac{x + x'\epsilon}{y + y'\epsilon} = \frac{(x + x'\epsilon)(y - y'\epsilon)}{y^2 - y'^2\epsilon^2} = \frac{x}{y} + \frac{x'y - y'x}{y^2}\epsilon \tag{16}$$

These results can be extended to any smooth real function using its Taylor series expansion of a dual number. As an example, one can consider $\sin(x + x'\epsilon)$, for which the Maclaurin series expansion is:

$$\sin(x + x'\epsilon) = \sum_{n=0}^{\infty} \frac{(-1)^n (x + x'\epsilon)^{2n+1}}{(2n+1)!} \tag{17}$$

From [1, p. 2], we have the formula for raising a dual number $(x + x'\epsilon)$ to the power m:

$$(x + x'\epsilon)^m = x^m + mx'x^{m-1}\epsilon \tag{18}$$

This formulae, using $m = 2n + 1$, can be substituted into Equation 17, to yield:

$$\sum_{n=0}^{\infty} \frac{(-1)^n [x^{2n+1} + (2n+1)x'x^{2n}\epsilon]}{(2n+1)!} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} + \epsilon x' \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \tag{19}$$

The first series is the Maclaurin series for $\sin(x)$ and the second can be recognized as the Maclaurin series for $\cos(x)$. Therefore,

$$\sin(x + x'\epsilon) = \sin(x) + \cos(x)x'\epsilon \tag{20}$$

Since it is arbitrary, we can select the value of x' to be 1. Using the value of 1, a function applied on a dual number will return the derivative of the function in the second/dual component.

AD builds on the idea that an algebra can be done directly on (*value*, *derivative*) pairs. We adjoin a new element (multiple elements in the multivariate case) and apply the derivative rules of each operation and function as they are called.

The theory of forward AD makes extensive use of dual numbers. We would like to generalize these results to compute the gradients of multivariable functions. Fortunately this is straightforward using the FADBAD++ approach.

2.2 FADBAD++

FADBAD++ includes a template that will modify some other type (typically a double or float) into a type suitable for automatic differentiation. In this way, a variable in FADBAD++ consists of a value and a gradient array. When initialized, the array is empty. Before we use these variables in functions, we must initialize these "gradient arrays" so that the partial derivatives are calculated properly. As a simple example, consider the following code, which is explained below:

```
fadbad:F<double> x=5,y=3;

x.diff(0,2);
y.diff(1,2);

fadbad::F(double)v=x*y;

cout<<v.val()<<endl;
cout<<v.d(0)<<endl;
cout<<v.d(1)<<endl;
```

We can break this code down line by line:

```
fadbad:F<double> x=5,y=3;
```

At this point, `x=5 []`, and `y=3 []`. The gradient arrays are currently empty.

```
x.diff(0,2);
y.diff(1,2);
```

Now `x=5 [1,0]`, and `y=3 [0,1]`. The first argument to `diff` is the position in the array that this variable will occupy (and therefore the component which is initialized to 1). The second is the length of the array. Two variables must have the same length of array to be compatible. If two variables are initialized to occupy the same position, the code will run but the results will be incorrect, as the derivative calculations will overwrite each other at various points.

```
fadbad::F(double)v=x*y;
```

This is where the actual work happens. the $*$ operator has been overloaded to apply the product rule to each element of the array. So after this line we find that `v=15 [3,5]`. Other operators and functions have been overloaded to work similarly.

```
cout<<v.val()<<endl;
cout<<v.d(0)<<endl;
cout<<v.d(1)<<endl;
```

This are how we access the computed values. The first line gets the value of v, which is 15. The second gets $\frac{\partial v}{\partial x}$ (since x was initialized with position 0 in the gradient array), which is 3. The third line gets $\frac{\partial v}{\partial y} = 5$ similarly.

AD includes the virtues of both symbolic differentiation and numerical differentiation. It gains the accuracy of symbolic derivatives, but eliminates large expressions by using numerical values at every step. It gains the simplicity of numerical differentiation for similar reasons, but does not suffer from inaccuracies, due to its use of the chain rule and the symbolic derivative formulas for each individual operation.

3 User Interface and Compatibility

In this section we first look at how the original symbolically generated code for creating material models was modified to use AD. We then look at how the steps for actually running a material test have been changed.

3.1 Creating a Material Model

AD computes the gradient of a function alongside the function value itself. Unlike the case for symbolic computation, no actual expression for the derivative is created; the calculations must be done at runtime and cannot be hard-coded in advance. This necessitates treating the derivative as a black box whenever it appears in a larger calculation, which in turn suggests rewriting the code in a manner more similar to the mathematical formulas describing it. For example, the code representing the Perzyna Equation (Equation 10) is as follows:

```
m_vector AD_plfluid_material::depsilonvp (const dataset& _dset, scalar _d
{
    m_vector _r(6);
    for(int i=0;i<6;i++)
    {
        _r[i]=_dt*get_gamma(_dset)*PhiF(_dset,_F)*Q_gradient[i];
    }
    return _r;
}
```

Because of this abstraction, material models now only differ in a few places: namely in the definitions of material-specific functions and wherever material constants are called. This motivated the decision to consolidate the perviously completely distinct material files into one, which could then have these smaller parts replaced by a smaller file. Macros were used to construct a simple generic material creation interface, intended as a proof of concept in place of MatGen. Continuing with the example of the power law fluid, the new interface is as follows:

```
#include "general_material_macros.h"

//defining constants
#define __NUMBER_OF_CONSTANTS__ 2
#define __CONST_1__ A
#define __CONST_2__ m

#define __Q_FUNCTION__ sqrt(3*_J2_)
#define __F_FUNCTION__ sqrt(3*_J2_)
#define __GAMMA_FUNCTION__ A
#define __PHI_FUNCTION__ pow(F,m)
#define __KAPPA_FUNCTION__ 0.0
```

A material is defined by a header file such as this, which is then included

in the generic material class. This header defines the macros in the generic file to represent a given material. The user must specify the number of constants (up to four), and their names. The constant macros are of the form `__CONST_n__`, and must be given in numerical order (`__CONST_2__` cannot be a specified if `__CONST_1__` is not). The definitions of F, Q, κ, ϕ and γ must also be given. Defintions of the same macros used in MatGen are given to help with this.

3.2 Running an Experiment

The input file used to run a given experiment is unchanged from that used previously [2].

```
# MatCalc Simulation Data
#
# Experiment:                       UniaxialX
# Yield function:                   generic_material
# Algorithm:                        Radial return map
# Time step:                        2.000e-03
# Time span:                        3.000e-00
# Specimen dim. X:                  1.000e-00
# Specimen dim. Y:                  1.000e-00
# Specimen dim. Z:                  1.000e-00
# Experiment constants
#              displacement_rate_x: 1.000e-02
# Material constants
#                         ElasticE: 3.000e+04
#                        ElasticNu: 3.000e-01
#                                A: 2.000e-04
#                                m: 1.000e-00
#
# Data records:                     0
#
# EOF
```

The usage of the yield function and algorithm options have changed slightly. In the original code, the derivatives were expanded symbolically where needed, but these derivatives did not depend on all of the same inputs as the function they were being called in, and so could have been calculated less often. To do this in the revised AD code, we used members of the material class to store the values and gradients of the Q and F functions. Other gradients that were only called once per step were not stored this way. The integrator and material class were then rewritten, so that the gradients are now calculated directly after the stress is updated, and the values are then

accessed when needed. This reduced the number of gradient calculations in half. However, as a side effect different parts of MatCalc are now incompatible. Both the elastic and viscoplastic integrators are compatible with the old materials, but not the new. Currently the radial return map algorithm is the only one that works for the generic material file. The integrator had to be modified to remove the redundant function calculations, breaking the interface. The radial return map algorithm is the most accurate and can handle all materials, so in the interests of time the other integrators were not similarily modified to work with the generic material file.

A minor issue is that a material is no longer specified before running a given experiment: to switch materials, the material-specific header that is included in the generic material file must be changed. The user must then go to `.../3.0/matcalc` and run `make` to rebuild MatCalc. This is an inconvenience. A better system that avoids this and the error-related issues inherent to macros could ideally be developed in the future.

4 Results

The simulation results for the AD code versus the symbolic code are very similar, as presented in the next section. The benchmark experiments for speed show more variety in the results, as discussed below.

4.1 Accuracy

Experiments done on the power law fluid and strain hardening material showed negligible deviations from the results of the old code. The relative difference between the stress vectors were calculated using the ℓ^2 norm as

$$\frac{||\boldsymbol{\sigma}_{new} - \boldsymbol{\sigma}_{old}||_2}{||\boldsymbol{\sigma}_{old}||_2} \times 100\% \tag{21}$$

The relative difference between the strain vectors is calculated similarly.

4.1.1 Power Law Fluid

Figure 2 shows the simulations results for a uniaxial extension of a power law fluid, with three different values of the power m. The parameters used to define the experiment are listed below.

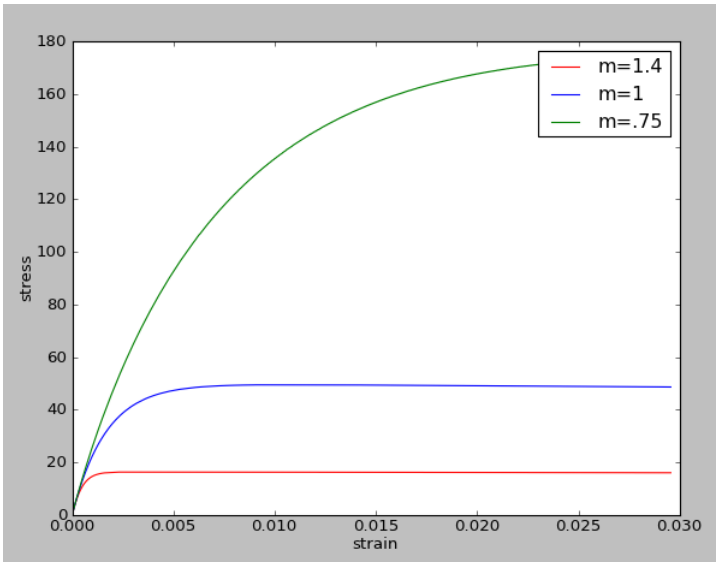

Figure 2: Stress versus train for uniaxial extension of a power law fluid.

- Experiment: Uniaxial extension in the x-direction
- Material Model: Power Law Fluid
- Numerical Integration Algorithm: Radial Return Map
- Time Step: 2×10^{-3} seconds
- Time Span: 3 seconds
- Element Dimensions: 1m by 1m by 1m
- Displacement Rate: 1×10^{-2} metres/second
- Material Constants:
  - Elastic Modulus: $3 \cdot 10^4$ Pa
  - Poisson's Ratio: 0.3
  - A: 2×10^{-4}
  - m: 1.4, 1.0, and 0.75.

The relative difference between the old code and new code is extremely low, as shown in Table 1.

m	$\boldsymbol{\sigma}$ % difference	ε % difference
1.4	$3.76 \cdot 10^{-7}$	0.0
1.0	$5.28 \cdot 10^{-8}$	0.0
0.75	$1.49 \cdot 10^{-9}$	$5.58 \cdot 10^{-12}$

Table 1: Relative difference between symbolic and AD codes for power law fluid experiments.

4.1.2 Strain Hardening Material

Figure 3 shows the simulations results for a strain hardening material with different values of the strain hardening parameter (n). The experimental setup of these experiments were the same as for the power law fluid, given in Section 4.1.1. The material constants used are listed below.

- Elastic Modulus: $3 \cdot 10^4$ Pa
- Poisson's Ratio: 0.3
- A: 2×10^{-4}
- m: 1.00
- n: 0.95, 0.90, 0.85, 0.80

As for the power law fluid, the relative difference between the new and old simulation results is small, as shown in Table 2.

n	$\boldsymbol{\sigma}$ % error	ε % error
0.95	$2.17 \cdot 10^{-5}$	$5.58 \cdot 10^{-12}$
0.90	$7.89 \cdot 10^{-10}$	$1.04 \cdot 10^{-11}$
0.85	$1.05 \cdot 10^{-9}$	$8.36 \cdot 10^{-12}$
0.80	$1.36 \cdot 10^{-9}$	$4.35 \cdot 10^{-12}$

Table 2: Relative difference between symbolic and AD codes for strain hardening experiments.

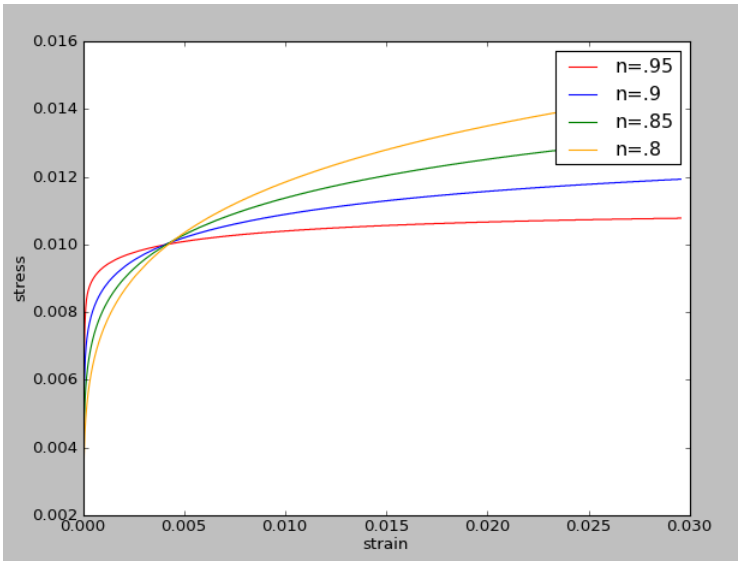

Figure 3: Stress versus train for uniaxial extension of a strain hardening material.

4.2 Benchmark Tests for Speed

MatCalc 3.0 was benchmarked against the original MatCalc 2.0 code for comparison. Experiments were run with the following parameters.

- Experiment: Uniaxial extension in the x-direction
- Material Model: Power Law Fluid
- Numerical Integration Algorithm: Radial Return Map
- Time Step: 2×10^{-3} seconds
- Time Span: 3 seconds
- Element Dimensions: 1m by 1m by 1m
- Displacement Rate: 1×10^{-2} metres/second
- Material Constants:
  - Elastic Modulus: $3 \cdot 10^4$ Pa, $3 \cdot 10^3$ Pa, and $3 \cdot 10^2$ Pa

  - Poisson's Ratio: 0.3
  - A: 2×10^{-4}
  - m: Ranging from 1.4 to 0.5, in intervals of 0.1.

For the strain hardening material, the material constants were:

- Elastic Modulus: $3 \cdot 10^4$ Pa
- Poisson's Ratio: 0.3
- A: 2×10^{-4}
- m: 1
- n: Ranging from 0.55 to 0.95, in intervals of 0.05.

The relative differences between experiment execution times for the power law fluid and the strain hardening material are respectively given in Figures 4 and 5. The three different plots in Figure 4 correspond to the different values of E listed under the material constants above. Modifying E has the influence of changing the relaxation time for the power law material. A positive % difference in these plots means that the AD code is slower relative to the symbolic code. A negative % difference means that the AD code is faster.

The data shows some unusual trends. For example, the amount of time taken varies widely with the various material constants. This is likely related to how many times the loops within an integration step are running, though the non-monotonic behaviour suggests that the issue may be subtle. Unfortunately we did not have access to a proper benchmarking tool, which would have allowed for better optimization and more detailed results. Some differences between the AD and symbolic results are notable:

- The AD code typically runs slower for the power law fluid, but typically runs faster for the strain hardening material.
- When AD code *is* faster, it's typically faster by a large margin, while when it's slower it's only by a few percent.
- The new code is less stable. For example, the old code could continue the strain hardening material experiments to lower values of n, but the new code could not do so, unless a smaller timestep was used.

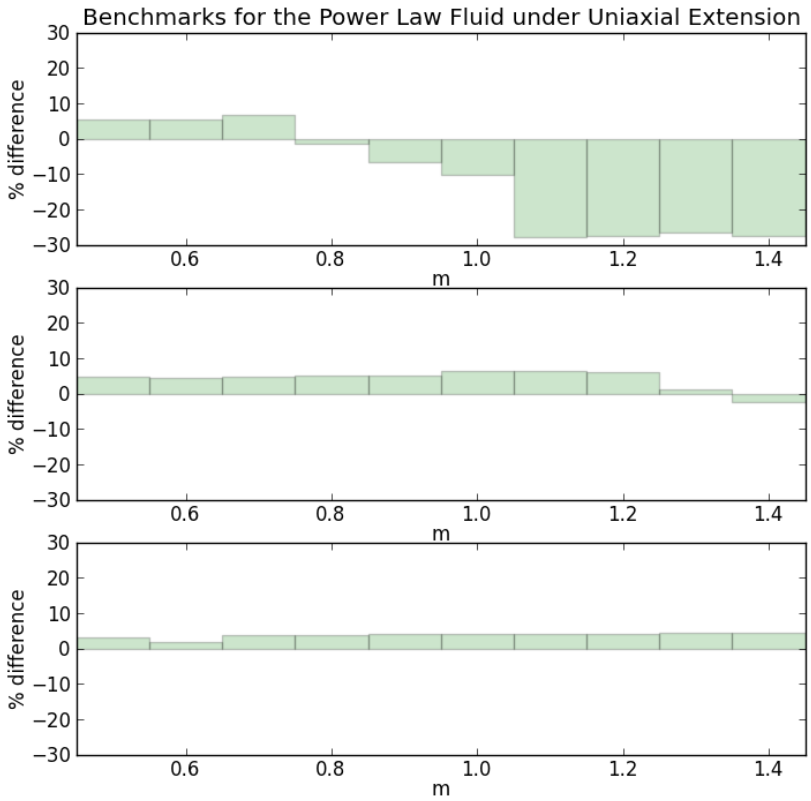

Figure 4: Percentage difference in execution time between AD versus symbolic code for a power law material with different relaxation times.

- The new code still had periods where it ran much faster even when AD was implemented, but before the redundant calculations were removed (as described in Section 3.2). Removing those calculations resulted in a steady speed increase of roughly 3-6%. This may be cache related, but better diagnostic tools are needed to investigate the difference.

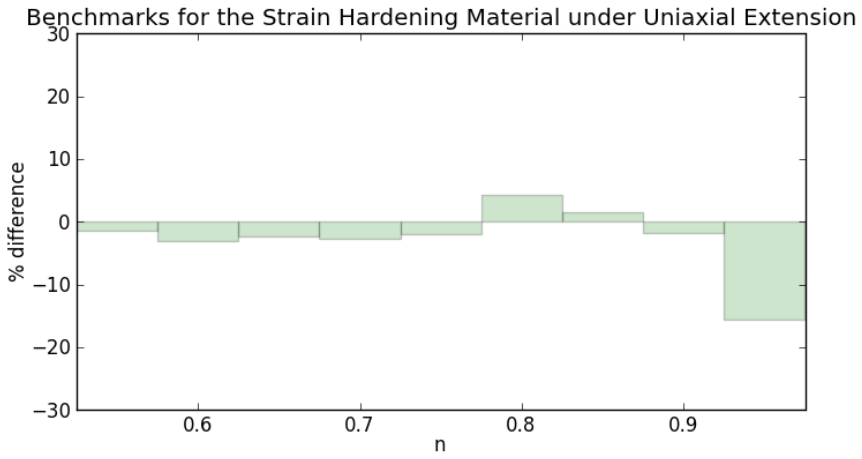

Figure 5: Percentage difference in execution time between AD versus symbolic code for a strain hardening material.

5 Conclusion

Automatic Differentiation is a suitable method for calculating derivatives at runtime in Virtlab. Using FADBAD++ removes the dependency on commercial software, potentially increasing Virtlab's future userbase. Using AD also drastically shortens the amount of code needed to define various materials, shrinking the size of the program. While the macro-based material creation interface is not optimal, it is easy to use and shows that the new method creates no inherent issues for the end user. Given more time, this could be reworked into a more suitable generative approach, potentially integrated into MatGen. However, this is but one of many possible approaches. To determine the best approach for Virtlab in the future, more comparisons need to be done:

- Using free software such as Sage [4] or SymPy [8] would allow computation of symbolic derivatives and code generation without depending on a commercial product.
- Redundant derivative calculations are not inherently required when using symbolic computing. The symbolic method could be rewritten to remove these redundancies.

- Better diagnostic tools, such as Valgrind [9], should be used to better understand the speed differences found and to investigate how each method can be improved.

Acknowledgements

The financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Undergraduate Student Research Award (USRA) program is gratefully acknowledged.

References

[1] Ian Fischer. *Dual-Number Methods in Kinematics, Statics and Dynamics*. Taylor & Francis, 1998. ISBN 9780849391156. URL `http://books.google.com.tr/books?id=hfinyCUHDW0C`.

[2] John McCutchan. A generative approach to a virtual material testing laboratory. Master's thesis, McMaster University, Hamilton, ON, Canada, September 2007.

[3] P. Perzyna. Fundamental problems in viscoplasticity. *Advances in Applied Mechanics*, pages 243–377, 1966.

[4] Sage Development Team. Sage computer algebra system, 2014. URL `http://sagemath.org/index.html`.

[5] Gonzalo Sanchez. Virtual material testing laboratory (version 1.1). Master's thesis, School of Computational Science and Engineering, McMaster University, Hamilton, ON, Canada, April 2010.

[6] W. Spencer Smith and Huanchun Gao. A virtual laboratory for material testing. In N. Callaos, R. H. Chavez, S. Franger, R. Raut, and Z. He, editors, *WMSCI 2005, The 9th World Multi-Conference on Systemics, Cybernetics and Informatics, Volume VI*, pages 273–278, Orlando, Florida, 2005.

[7] Ole Stauning and Claus Bendtsen. FADBAD++ flexible automatic differentiation using templates and operator oveloading in ANSI C++, 2003. URL `http://www.imm.dtu.dk/~kajm/FADBAD/`.

[8] SymPy Development Team. SymPy python symbolic mathematics library, 2013. URL http://www.sympy.org/en/index.html.

[9] Valgrind Development Team. Valgrind diagnostic tool, 2014. URL http://valgrind.org/.