A generative approach to a virtual material testing laboratory

A generative approach to a virtual material testing laboratory

By
John McCutchan, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (2007) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: A generative approach to a virtual material testing laboratory

AUTHOR: John McCutchan, B.Sc. (McMaster University)

SUPERVISORS: Dr. Spencer Smith

NUMBER OF PAGES: x, 82

Abstract

This thesis presents a virtual material testing laboratory that is highly generic and flexible in terms of both the material behaviour and experiments that it supports. Generic and flexible material behaviour was accomplished via symbolic computation, generative programming techniques and an abstraction layer that effectively hides the material model specific portions of the numerical algorithms. To specify a given member of the family of material models a domain specific language (DSL) was created. A compiler, which uses the Maple computer algebra system, transforms the DSL into an abstract material class. Three different numerical algorithms, including a return map algorithm, are presented in the thesis to illustrate the advantage of the abstract material model. To accomplish the goal of generic and flexible experiments the finite element method was employed and an API that supports both load and displacement controlled experiments, as well as the capability for the experiments to modify their state over time, was developed. The virtual laboratory provides a family of material models with the following behaviours: elastic, viscous, shear-thinning, shear-thickening, strain hardening, viscoelastic, viscoplastic and plastic. As well, the developed framework, by using the Ruby programming language, provides support for a wide variety of programmable experiments, including: uniaxial, biaxial, multiaxial extension and compression, shear and triaxial.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Spencer Smith, for the endless amount of effort he has exerted on my behalf in the form of guidance, ideas and encouragement over the past three years that we have been working together. I would also like to thank my great friend and roommate, Jeanine Campsall, for her tolerance and encouragement. Finally, I would like to thank my family, the Campsalls and all my friends at McMaster and elsewhere.

> "Please accept my resignation. I don't want to belong to any club that will accept me as a member."
>
> *Groucho Marx*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modelling the response of different materials under various loading histories is of critical importance to scientists and engineers. For example, a geotechnical engineer needs to model the loading characteristics of soil to accurately predict the settlement of a building. Without an accurate model of the soil, serious damage could occur and in extreme cases the building may even collapse. As another example, designers of automobiles need to model material behaviour so that they can predict how much mechanical energy a vehicle frame can absorb during a collision. In this case an accurate material model is vital for passenger safety. These are just two examples where understanding the response of materials under loading is vital.

Modelling the response of materials is potentially complex and challenging. The relationship between the loading and the deformation of a material can rely on multiple non-linear equations, which are potentially dependent on the entire history of the material's deformation and temperature. In addition to the modelling challenge, another challenge exists in constructing and performing the physical experiments needed to determine the values of the model's parameters. Given the great importance of understanding material behaviour, these modelling and experimental challenges need to be overcome. One approach to overcoming these challenges would be a tool to assist scientists in developing new material models. This tool would be even more valuable if it could assist science and

engineering students in learning the complex field of material modelling and experimentation. For instance, the tool could facilitate students gaining a deep understanding of the differences between various materials by allowing them to perform many virtual experiments with a wide variety of materials. This thesis presents such a tool, in the form of two programs: MatGen and MatCalc. Together these programs provide a generic extensible virtual laboratory for material testing and modelling.

The rest of this chapter is divided into several sections. The next section provides background information on material testing. A section on material modelling follows. Thereafter the benefits of a virtual laboratory are discussed. To realize these benefits the techniques of generative programming are utilized, so an overview of this field is provided in the next section. The final section discusses the purpose and scope of the thesis and the tool that was developed.

1.1 Material Testing

When testing materials, the results rely not only on the boundary conditions for the test, but also on the loading and deformation history of the test specimen. For example, if a steel rod is permanently stretched the internal structure of the steel rod has been changed. This internal change will impact how the rod reacts to future loads or deformations. Tests performed in a laboratory require elaborate equipment including a test apparatus and a test specimen in a known state. An example material test apparatus is shown in Figure 1.1.

There are two types of tests: load controlled and displacement controlled. In the first type, a load or force is applied to the test specimen and the resulting deformation or strain is measured. This is called a load controlled experiment because one is controlling the amount of force that the specimen experiences. As an example, an experiment might involve applying a known force or sequence of forces, to a lead cylinder and observing how the cylinder deforms over time. Contrary to load controlled experiments, displacement controlled experiments apply a sequence of deformations to the test specimen and measure

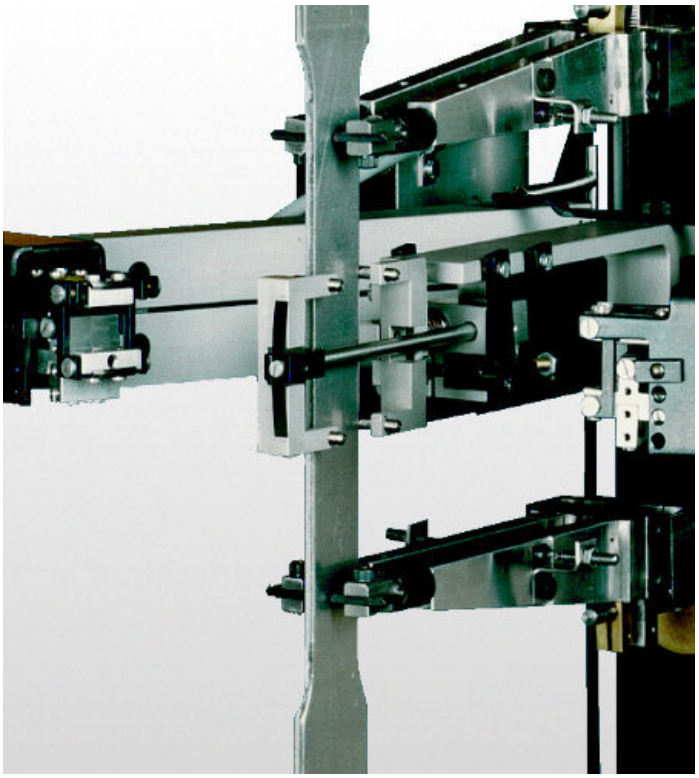

Figure 1.1: Material testing apparatus (Roell, 2007)

the internal force or stress over time.

Within both load and displacement control experiments there is a variety of boundary conditions that control how and where the load or displacement is applied to the test specimen. The simplest is the uniaxial extension or compression test. In this test the test specimen is held fixed at one end and the free end is pulled or pushed along the test specimen axis that is aligned between the two ends. Some example test specimens that are tested in engineering practices include metal or plastic rods, concrete cylinders and soil samples.

Soil samples are typically tested using a triaxial test. In the triaxial test the soil is placed within a cylindrical membrane and a confining pressure is applied, which is intended to approximate the confining pressure the soil would experience in situ. The specimen then has a load applied in the same direction as the length of the cylinder. While the loading is applied the deformation history is measured.

1.2 Material Modelling

Material models provide a relationship between the stress (load) and the strain (deformation). These models are a mathematical approximation of real world material behaviour. There exist many different models for various material behaviours. This thesis is mainly interested in elastic, viscous and plastic material behaviour and combinations of these three behaviours. In this section the differences between elastic, viscous and plastic material behaviours will be presented along with common models of each. The models will be presented using a common experiment, the uniaxial extension of a rod, as shown in Figure 1.2. The test specimen is a rectangular box with the original dimensions of $L_0 \times W_0 \times H_0$. A force F is applied to the free end of the specimen so that it deforms to the new dimensions of $L \times W \times H$. This experiment is 1D and thus allows illustration of the important points, without the need to introduce unnecessary details. Before describing the three material behaviours, a brief introduction to the definition of stress and strain will be presented.

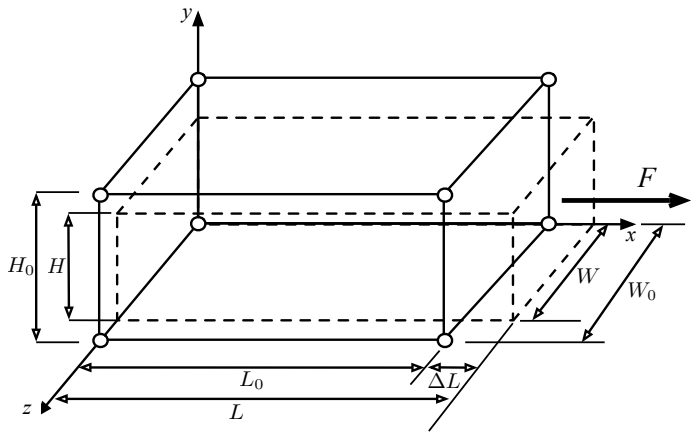

Figure 1.2: Test specimen undergoing uniaxial extension test

1.2.1 Stress and Strain

Consider a uniaxial extension experiment, shown in Figure 1.2. Stress is defined as the force (F) divided by the current (deformed) cross-sectional area ($A = WH$). Stress is denoted by σ.

$$\sigma = \frac{F}{A} \tag{1.1}$$

A distinction should be made between true stress and engineering stress. The above equation is for true stress, but a commonly used simplification is engineering stress (σ^E) on the other hand, references the original undeformed configuration, as follows:

$$\sigma^E = \frac{F}{A_0} \tag{1.2}$$

where $A_0 = W_0 H_0$ is the original cross-sectional area of the rod before loading. The true stress definition takes into account that deformations will occur under loading and thus change the area that the force is applied to. In a typical uniaxial extension experiment deformation will lead to a significant decrease in the cross-sectional area of the member; this phenomenon is known as "neck-in."

Strain, which is used as a dimensionless measure of deformation, is denoted by ϵ. It is easier to define engineering strain before true strain, because of the latter's relative complexity. Engineering strain, denoted here by ϵ^E is defined as the change in the length (Δl) of the rod over the original length (L_0). (i.e. relative change in length.)

$$\epsilon^E = \frac{\Delta l}{L_0} \tag{1.3}$$

Unlike engineering strain, true strain takes into account the history of length changes not just L_0. The true strain is defined by first considering a small strain increment ($d\epsilon$), which is defined as follows:

$$d\epsilon = \frac{dL}{L} \tag{1.4}$$

where dL is the current increment in the length and L is the current rod length. By summing all strain increments over the course of a given deformation the true strain is defined by the following equation:

$$\epsilon = \int_{L_0}^{L_0+\Delta L} \frac{dL}{L} = \ln\left(\frac{L_0 + \Delta L}{L_0}\right) = \ln\left(\frac{L}{L_0}\right) \tag{1.5}$$

where L in the last equation is the final length of the test specimen.

For very small deformations, which are the most common in practice, the approximation of engineering stress and strain are essentially equivalent to the true values and thus remain physically meaningful and useful for most engineering purposes. For a more detailed discussion on stress and strain see Beer and Johnston Jr. (1985).

1.2.2 Elasticity

Elastic materials are materials that deform when loaded and return to their original configuration after the load is removed. Elastic materials can be modelled as springs that follow

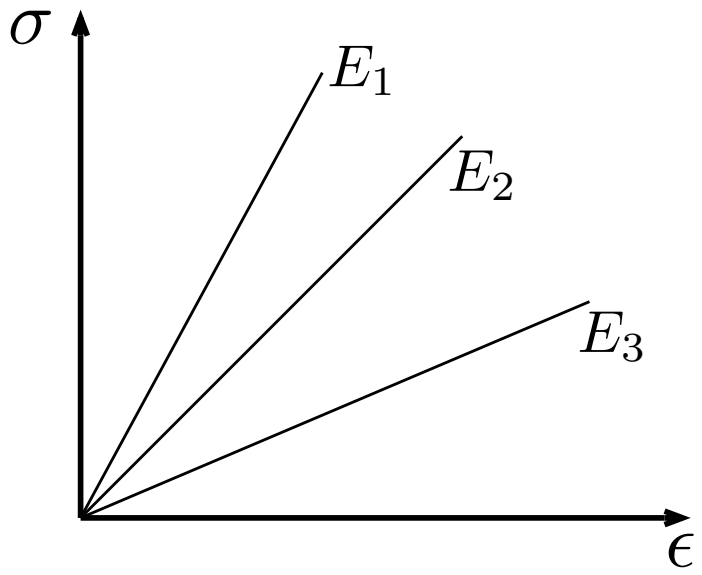

Figure 1.3: Typical elastic stress vs. strain graphs $E_1 > E_2 > E_3$

Hooke's law. The stress in an elastic material is linearly related to the strain of the material, as follows:

(1.6) $\sigma = E\epsilon$

where E is known as Young's modulus. It is easy to see the connection with the spring equation. $F = kx$. An elastic model is generally only accurate for small strains.

A graph of stress vs. strain for an elastic material with three different values of E can be seen in Figure 1.3.

1.2.3 Viscosity

Viscosity describes the stress that develops in a material to resist a given rate of deformation. Viscosity is typically associated with fluids. A material with high viscosity such as honey, resists a higher rate of deformation than a material with a low viscosity such as water. The stress of a viscous material depends on the rate of strain, $\dot{\epsilon}$, and the coefficient

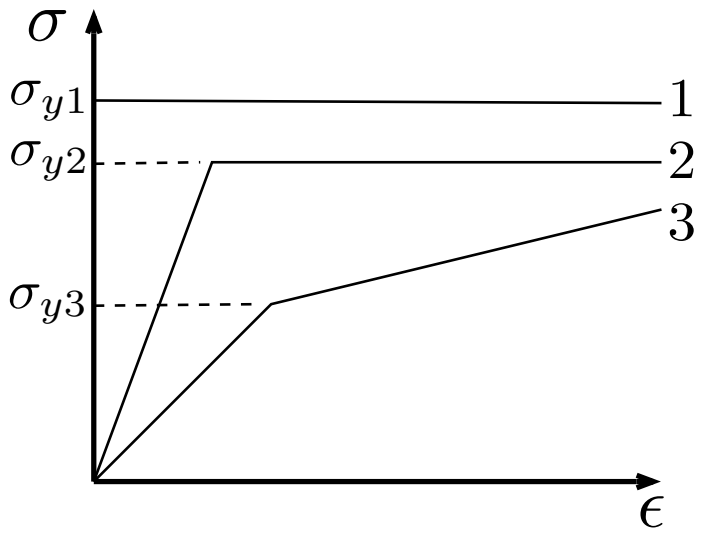

Figure 1.4: Typical plastic stress vs. strain graphs showing: (1) perfectly plastic, (2) elastic perfectly plastic and (3) strain hardening

of viscosity, η, as follows:

(1.7) $\sigma = 2\eta\dot{\epsilon}$

1.2.4 Plasticity

Unlike elastic materials, when a plastic material is loaded it will deform permanently. Elastoplastic materials begins deforming elastically until the material yields, after which the deformation is permanent. Materials which can undergo large plastic deformations without fracturing are described as ductile. In contrast, materials which fracture suddenly, such as concrete, are called brittle materials. Some example plots of stress vs. strain for plastic materials can be seen in Figure 1.4. Plastic deformation begins once the stresses (σ) have reach the yield point, this can be seen in Figure 1.4. Perfectly plastic materials yield immediately and elastoplastic materials first exhibit elastic behaviour and then after yielding, begin to behave plasticly. Strain hardening is the phenomenon of the yield stress getting larger as the material undergoes strain.

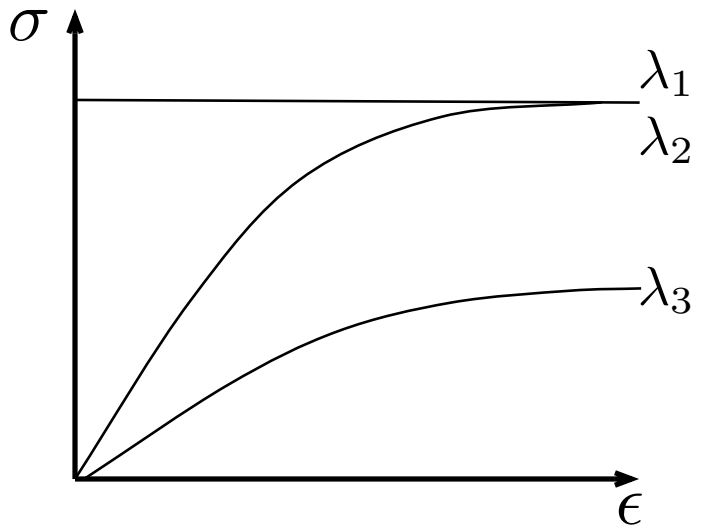

Figure 1.5: Typical viscoplastic stress vs. strain graphs where the relative relaxation times are ordered $\lambda_1 < \lambda_2 < \lambda_3$ (constant strain rate test.)

1.2.5 Viscoplasticity

Commonly, materials have behaviour that is a combination of elastic, viscous and plastic material behaviours. These types of materials are labelled viscoplastic materials. These materials include metals, soils, and molten polymers. The Maxwell model for viscoelastic material can also describe viscoplastic materials. Maxwell's model can be thought of as a viscoplastic damper connected in series with a purely elastic spring (Shown in Figure 1.6.) A plot of a stress vs. strain for a viscoplastic material can be seen in Figure 1.5. The plot includes three different relaxation times. Relaxation time is the measure of how quickly the elastic stress relaxes. Low values of λ correspond to the viscous behaviour discussed in Section 1.2.3. In Figure 1.5 the smallest relaxation time (λ_1) corresponds with a viscous response under the constant rate of strain uniaxial extension experiment.

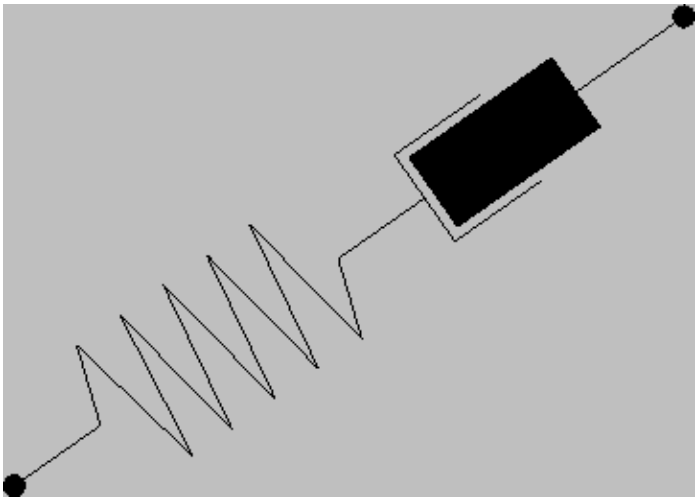

Figure 1.6: Spring and damper connected in series (Radi, 1998)

1.3 Advantages of a Virtual Laboratory

Virtual laboratories are available in a wide variety of fields. For example, ViBE (Subramanian and Marsic, 2001) is a virtual biology laboratory. Additional examples can be found in a recent literature review of virtual laboratories (Ma and Nickerson, 2006). This review gives evidence of virtual laboratories existing in such diverse fields as electrical engineering, telecommunications, and environmental sciences to name just a few. These virtual laboratories are targeted not just at researchers but at students as well. Yaron et al. (2005) discuss a virtual chemistry laboratory designed to aid students learning chemistry and to assist their instructors in presenting the subject.

As in other fields, a virtual laboratory for material testing can provide many advantages. Most of these benefits can be realized by providing a general framework where new experiments and material behaviour models can be easily added.

A virtual material testing laboratory provides an ideal environment for developing an understanding of a given material model. Testing models under a wide variety of situations becomes trivial. Although real experiments are still required to find the parameters for a given material, the virtual experiments provide a means for an in depth exploration of

material models. Typically experiments consist of a set of boundary conditions placed on the test specimen. In a real laboratory these boundary conditions cannot be enforced perfectly. Fortunately, this is not the case in a virtual laboratory. Also, potentially unwanted natural phenomena such as friction and gravity do not interfere in a virtual laboratory. As well, inevitable natural imperfections present in real test specimens cease to be a concern in a virtual laboratory. Therefore, a virtual material testing laboratory can help researchers gain insight into the parameters derived from real laboratory experiments.

A virtual laboratory can be a real boon for researchers modelling materials. Researchers can easily add new material behaviour models to the system and immediately perform experiments on their new models. Comparing their new models to previous models provides insight into which model is best in a particular situation.

Students studying the material sciences can also benefit from a virtual laboratory. The opportunity to easily perform experiments on a wide variety of material models, as they study the models, without requiring them to use laboratory equipment, will aid the student in understanding how different materials react under certain conditions. VizCore (Hashash et al., 2002), is an existing tool which uses visualization techniques to help students evaluate material models. VizCore does not offer the ability to develop or add new material models or experiments to the system.

Virlab (Smith and Gao, 2005; Gao, 2004) was an earlier attempt at providing a virtual laboratory for material testing, but it too was limited to a set of hard coded material behaviours and experiments. To extend Virlab with a new material behaviour an expert in material sciences is required to perform complex mathematical derivations and then write source code implementing the new material behaviour. Virlab failed to offer the kind of framework necessary to truly see the benefits of a virtual material testing laboratory. Virlab has no mechanism for adding new experiments independent of the material model. To overcome this limitation, a virtual laboratory should make the experiments programmable. Removing the need for a material sciences expert to add new material behaviour models to the system is not as simple as adding programmable experiments. Generating a program

based on a high level description of the material model is one way of meeting this need. An earlier attempt using a generative approach can be found in Arnold and Tan (1989) where symbolic derivation of portions of the material model were performed with a LISP program. The program has only one function that is to symbolically differentiate parts of the material model. By combining a virtual laboratory with a programmable experiment system and using generative programming to make adding new material behaviour models easy, a virtual material testing laboratory can provide many benefits to experimenters, researchers, and students.

1.4 Generative Programming

Generative programming is the practice of having one program write the source code for another program. Typically the input to the first program is a simple description of the generated program. As stated in the previous section, one of the negative aspects of Virlab was the need for a material sciences expert to derive and code any new material behaviour added to the system. Ideally the new code should be generated from a minimal amount of user input. However, the problem of generating a new material behaviour model (Section 3.1) is not trivial. For example, it requires taking second order gradients of functions (Section 2.3) and writing source code to evaluate these expressions. The majority of commercial material modelling tools, such as, FEMAP (Engineering, 2007) suffer from the same limitation as VirLab. In particular, FEMAP and other finite element analysis programs require a material expert to derive and implement any material models. FEMAP provides many robust material models and new material models can be added by experts by deriving and coding the new material behaviour, but it does not provide a way for a non-expert to add new materials to the system. One solution to this problem is to use generative programming, which compiles a high level mathematical description of the material model to a programming language. Generative programming is a solution because it removes need of the user to perform work that is complex, tedious and error prone. In the generative

programming community the high level language that the material model is described in, is termed a Domain-Specific Language (DSL). A DSL is a language that is tailored for the problem domain. A literature survey of DSLs and generative programming can be found in van Deursen et al. (2000). In the case of a virtual material testing laboratory, the DSL is a language that can express material models so that they can be compiled to another high level language, and the generated code can be added to the virtual laboratory.

1.5 Purpose & Scope

This thesis presents a virtual material testing laboratory consisting of two programs, MatCalc and MatGen, which together provide a flexible, generic and easy to use virtual material testing laboratory. MatCalc performs material experiments in 3D on elastic, viscous, and viscoplastic isothermal material behaviour models. Experiments run by MatCalc are programmable using the Ruby programming language (Matsumoto, 2007) and allow for both load and displacement controlled experiments. MatCalc can output the results from material experiments to a file or visualize them with a GTK+ (GTK+, 2007) GUI. MatCalc is written in C++ and can be used as a library in a larger material testing environment.

MatGen is a material behaviour model generator. It performs the duties of compiling a simple, high level mathematical description of a material behaviour model to a C++ class, which can then be used in MatCalc. MatGen uses Maple to perform symbolic computation and generate the C expressions from symbolic mathematical expressions. MatGen is also written in C++ (and Maple).

This thesis is divided into six chapters. Chapter 2 covers the theoretical physics background used in MatCalc, including the numerical algorithm used to model the material tests. Chapter 3 discusses the implementation details of MatGen and MatCalc. Chapter 4 explains how to extend MatCalc with new material behaviours and experiments as well as how to use MatCalc as a library in a larger material testing program. Following this, Chapter 5 covers experimental results and explains how the results were verified. Finally,

Chapter 6 presents concluding remarks and future work.

Chapter 2

Background

This chapter introduces background information that is important in understanding the theoretical aspects of a virtual material laboratory. First, the experimental setup is presented. Secondly, the governing equations of the physics are summarized. Finally, the numerical algorithm used to simulate the experiments is detailed.

2.1 Experimental Setup

The virtual laboratory conducts simple experiments that have real world analogues in materials testing. Figure 2.1 shows the idealization of the test specimen that is used for all tests. The initial configuration of the body is a rectangular brick, whose geometry can be described by eight nodes. The specimen will have three degrees of freedom at each node; that is, at node j the degrees of freedom will be the displacements u_j, v_j and w_j, corresponding to the x, y and z directions, respectively. Using the nodal displacements, the displacement of any points within the body can be found using the interpolation (or shape) functions:

$$u = \sum_{j=1}^{8} N_j u_j \tag{2.1}$$

$$v = \sum_{j=1}^{8} N_j v_j \tag{2.2}$$

$$w = \sum_{j=1}^{8} N_j w_j \tag{2.3}$$

To simplify the mathematics, locations are given using the dimensionless coordinates r,s and t. This coordinate system has the origin at the centroid of the test specimen and the values of the coordinates range -1 to 1. Using the dimensionless coordinate systems, the shape functions can be summarized as follows (Zienkiewicz et al., 2005):

$$N_j = \frac{1}{8}(1 + r_j r)(1 + s_j s)(1 + t_j t) \tag{2.4}$$

where r, s and t are the interpolation parameters $\in [-1, 1]$. A matrix $\boldsymbol{N}$ with dimensions 3 x 24 is constructed from the interpolation functions. The matrix $\boldsymbol{N}$ follows:

$$\boldsymbol{N} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & ... & N_8 & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & ... & 0 & N_8 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & ... & 0 & 0 & N_8 \end{bmatrix} \tag{2.5}$$

Equations 2.1, 2.2 and 2.3 can be rewritten in matrix form as the following:

$$\boldsymbol{u} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \boldsymbol{N}\boldsymbol{a} \tag{2.6}$$

where $\boldsymbol{a}$ is defined as the following:

$$\boldsymbol{a} = \begin{bmatrix} u_1 & v_1 & w_1 & ... & u_8 & v_8 & w_8 \end{bmatrix}^T \tag{2.7}$$

In all experiments the three dimensional displacement field and the internal forces will be changing over time. The specific types of the experiments are distinguished by the

boundary conditions applied to the brick. These boundary conditions may be modified as the experiment proceeds. For stability of the test specimen, each test will have the specimen fixed so that it cannot move in certain directions, or for load controlled experiments the specimen will be loaded in such a way to restrict movement. For instance, Figure 2.1 shows a uniaxial tension experiment that fixes node 1, which is at the origin of the coordinate axes, so that it cannot move. Other locations are potentially free to move in some directions, so that a resisting force will not develop in the corresponding direction. For instance, for the uniaxial experiment, node 8 is allowed to move in both the y and z directions, although it is fixed so that it cannot move in the x direction.

The deformation of the specimen over time depends on the fixity of the nodes of the brick and on the prescribed displacements or loads. In a displacement controlled experiment certain nodes are required to move by a set amount, whereas for load controlled experiments known forces are applied to the body. Using various combinations of fixity, displacement and load control, it is possible to construct tests for uniaxial extension/compression, biaxial extension/compression, multiaxial extension/compression, shear and triaxial experiments. Experiments are not restricted to a certain axis, they can be oriented along any of the coordinate axes. For example, a uniaxial extension test could be done in the x, y and z directions. Although it only allows linear interpolation, it is possible to use an 8 noded brick element, since the simple tests to be performed the different stress and strain values will be constant throughout the element. The strain for linear interpolation is constant, since as shown later, the strain is calculated from the gradients of the displacements.

2.2 Governing Equations

The notation used in this section and in the remainder of the paper is similar to the notation often used in finite element analysis (Zienkiewicz et al., 2005). That is, symmetric second order tensors, such as stress and strain, are represented as vectors and the equilibrium and

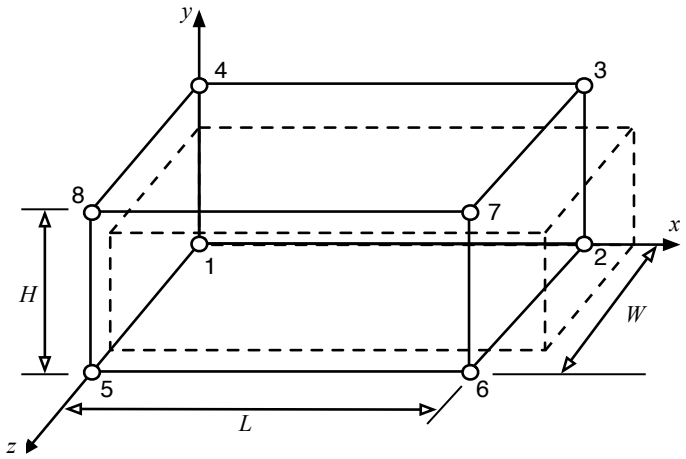

Figure 2.1: Test specimen

constitutive equations are written in matrix form.

2.2.1 Equilibrium Equation

At every instant in time the test specimen (Figure 2.1) must satisfy the equilibrium equation. If inertia, self-weight and other body forces are neglected, then the equilibrium equation can be written as

$$\boldsymbol{L}^T\boldsymbol{\sigma} = \mathbf{0} \tag{2.8}$$

where $\boldsymbol{\sigma}$ is the state of stress, which is a generalization of the 1D concept presented in Section 1.2.1. $\boldsymbol{L}^T$ is the following differential operator:

$$\boldsymbol{L}^T = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \tag{2.9}$$

The state of stress $\boldsymbol{\sigma}$ at a point is summarized using vector notation by six independent components acting on a small cube (Shown in Figure 2.2) centred at a point as follows:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{yz} & \sigma_{xz} \end{bmatrix}^T \tag{2.10}$$

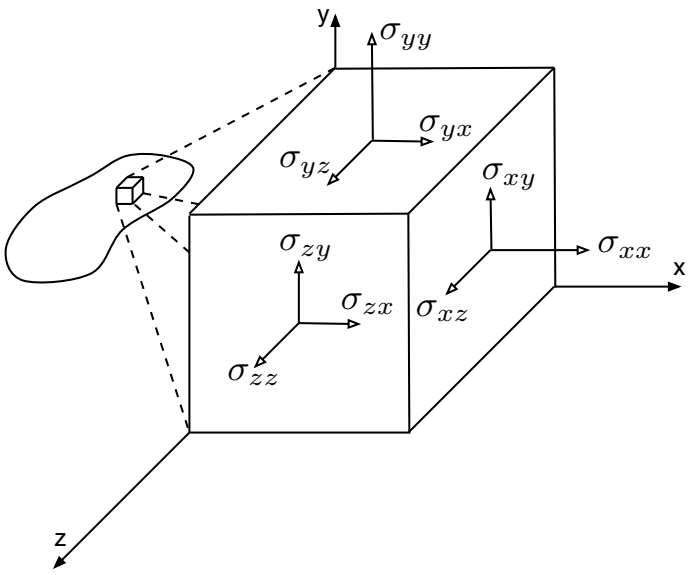

Figure 2.2: Stress tensor

where the subscripts (x, y, and z) refer to the coordinate axes. The first three stress components act normal to the faces of the cube, while the remaining three components are shearing stresses that act across the faces of the cube. Only six components of stress are needed because the remaining three: σ_{yx}, σ_{zy} and σ_{zx} are equal to σ_{xy}, σ_{yz} and σ_{xz} respectively (Beer and Johnston Jr., 1985).

2.2.2 Constitutive Equation

To determine the deformation and forces within a test specimen requires using the equilibrium equation (Equation 2.8). This equation applies to all bodies regardless of the material type. However, the equilibrium equation alone does not provide enough information to determine the specimen's new configuration. A material specific equation, called the closure or constitutive equation, is also needed. For a given material type, the constitutive equation models the relationship between the current stress field and the thermal and deformation history of the body. In the current virtual laboratory the body is assumed to be isothermal, so thermal effects will be neglected. An overview of the theory of constitutive equations

can be found in Malvern (1969) and Mase (1970).

In general the state of stress depends on the history of deformation, so a measure of deformation must also be introduced. A commonly used measure is the strain tensor $\boldsymbol{\varepsilon}$, which like stress can be generalized to a multidimensional case from the 1D example in Section 1.2.1. In vector form the strain tensor is written as

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{zz} & \gamma_{xy} & \gamma_{yz} & \gamma_{xz} \end{bmatrix}^T \tag{2.11}$$

where the first three components are normal strains and the last three represent shear strains. The strain tensor is related to the three-dimensional displacement field $\boldsymbol{u}$ as follows:

$$\boldsymbol{\varepsilon} = \boldsymbol{L}\boldsymbol{u} = \boldsymbol{L} \begin{bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{bmatrix} \tag{2.12}$$

where L comes from Equation 2.9.

Many different models for constitutive equations exist. One of the simplest, but still very useful in engineering applications, is the linear elastic model. In this model the change in stress $\Delta\boldsymbol{\sigma}$ does not depend on the entire history of deformation, only on the most recent change in the elastic strain $\Delta\boldsymbol{\varepsilon}^e$:

$$\Delta\boldsymbol{\sigma} = \boldsymbol{D}\Delta\boldsymbol{\varepsilon}^e \tag{2.13}$$

where $\boldsymbol{D}$ is known as the elastic constitutive matrix (Beer and Johnston Jr., 1985). $\boldsymbol{D}$ comes from a generalization of Hooke's law. The definition for this matrix is as follows:

$$\boldsymbol{D} = \chi \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} \end{bmatrix} \tag{2.14}$$

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

Another constitutive equation that has proven to be useful in practise is the one that relates the stress and the rate of deformation. In this case the interest is often in the relationship between the deviatoric stress tensor $\boldsymbol{s}$ ($\boldsymbol{s} = \boldsymbol{\sigma} - [\sigma_m \ \sigma_m \ \sigma_m \ 0 \ 0 \ 0]^T$, $\sigma_m = \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz})$) and the rate of viscoplastic strain tensor $\dot{\boldsymbol{\varepsilon}}^{vp}$

$$\dot{\boldsymbol{\varepsilon}}^{vp} = \lambda \boldsymbol{s} \tag{2.15}$$

where $\boldsymbol{s}$ gives the direction for the strain and λ provides the magnitude. The strain is termed viscoplastic because it is rate dependent (viscous) and the deformation is permanent (plastic).

The goal of the virtual laboratory is to use a constitutive equation that can accommodate both elastic and viscoplastic effects. One way that these effects are typically combined is by evoking the additivity postulate:

$$\Delta\boldsymbol{\varepsilon} = \Delta\boldsymbol{\varepsilon}^e + \Delta\boldsymbol{\varepsilon}^{vp} = \Delta\boldsymbol{\varepsilon}^e + \Delta t\dot{\boldsymbol{\varepsilon}}^{vp} \tag{2.16}$$

where $\Delta\boldsymbol{\varepsilon}$ is the total strain and Δt is the time step.

Equation 2.16 can be rearranged to solve for $\Delta\boldsymbol{\varepsilon}^e$ and combined with Equation 2.13 to obtain the following:

$$\Delta\boldsymbol{\sigma} = \boldsymbol{D}(\Delta\boldsymbol{\varepsilon} - \Delta t\dot{\boldsymbol{\varepsilon}}^{vp}) \tag{2.17}$$

A useful form of Equation 2.15 that allows modelling of a wide variety of material behaviour is the form proposed by Perzyna (1966):

$$\dot{\boldsymbol{\varepsilon}}^{vp} = \lambda\frac{\partial Q}{\partial\boldsymbol{\sigma}} = \gamma < \varphi(F) > \frac{\partial Q}{\partial\boldsymbol{\sigma}} \tag{2.18}$$

where γ is a fluidity parameter, F is the yield function, Q is the viscoplastic potential (also referred to as the dynamic loading surface) and

$$< \varphi(F) >= \begin{cases} \varphi(F) & \text{if} \quad F > 0 \\ 0 & \text{if} \quad F \leq 0 \end{cases} \tag{2.19}$$

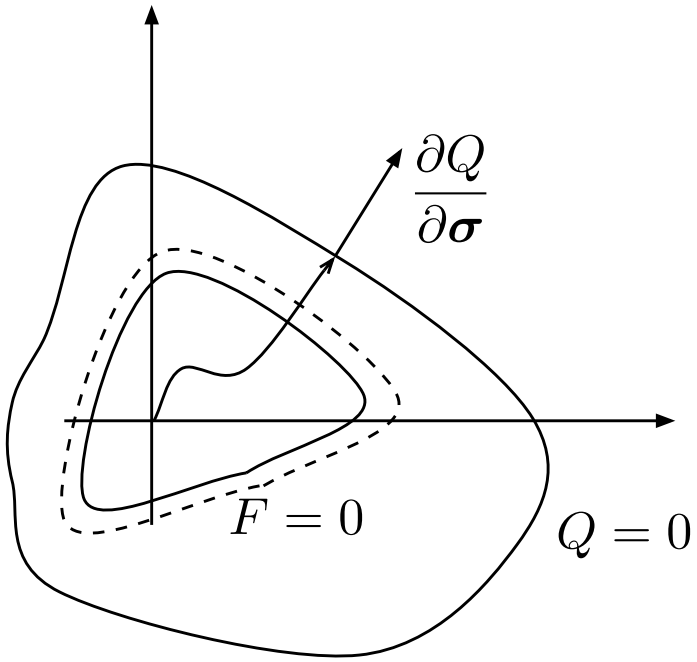

Figure 2.3: Yield function and plastic potential

with $\varphi(F)$ some function of F.

Equation 2.18 is similar to Equation 2.15 in that the viscoplastic strain rate is separated into a magnitude ($\lambda = \gamma < \varphi(F) >$) and a direction ($\frac{\partial Q}{\partial \boldsymbol{\sigma}}$). The occurrence of the yield function (F) in the Perzyna equation allows for the constitutive equation to accommodate elastic, viscous, viscoplastic and viscoelastic effects. The equation $F = 0$ defines a surface in 6 dimensional stress space, which can be visualized by considering the sketch shown in Figure 2.3. Inside the surface $\dot{\boldsymbol{\varepsilon}}^{vp} = 0$, which means, as shown by Equation 2.17, the material response will be purely elastic in this case.

When the material has yielded, which occurs when the stress path reaches the yield surface, as shown in Figure 2.3, the yield surface may change shape. This change in shape can be modelled as strain hardening (or softening) of the material. The new yield surface is shown as a dashed line in Figure 2.3. This behaviour is mathematically represented by having $F = F(\boldsymbol{\sigma}, \kappa)$, where κ is termed a hardening parameter. The parameter κ depends on the accumulated viscoplastic strain ($\kappa = \kappa(\boldsymbol{\varepsilon}^{vp})$). The addition of κ allows

for expansion or contraction of the yield surface depending on the instantaneous value of viscoplastic strains.

Figure 2.3 also shows a sketch of the surface Q, where Q is known as the potential function (Owen and Hinton, 1980). The value of Q depends on the state of stress ($Q = Q(\boldsymbol{\sigma})$). The normal to this plastic potential surface gives the direction of the viscoplastic strain increment. For many materials Q can be obtained from an isotropic expansion of the quasistatic yield surface. In this case, the material is said to obey an associative flow rule.

The above description of the constitutive equation shows several generic terms that need to be concretely specified before one can obtain the equations for a specific material. Four functions need to be given to describe a specific material: F, Q, κ, and φ. A constant γ must also be provided. The power of MatCalc and MatGen comes from postponing the specification of this information. The numerical algorithm (Section 2.3) used to solve for the deformation and stresses in the test specimen is derived using the generic forms, so that specific materials can be added simply by specifying the concrete form.

2.3 Numerical Algorithm

MatCalc uses the finite element method (FEM) to simulate the experiments. This method was selected because FEM naturally accommodates both displacement and load controlled boundary conditions. With the same finite element algorithm, all potential material tests can be simulated; all that changes between experiments is the input describing the boundary conditions. This means that each step through the experiment will solve for 24 displacement degrees of freedom, which will be stored in the vector $\boldsymbol{a}$ shown in Equation 2.7. Details of FEM can be found in Zienkiewicz et al. (2005).

The derivation of the finite element equations for the viscoplastic constitutive equation follows the approach presented by Stolle (1991). To estimate the displacements for the $(i+1)^{th}$ time step the residual for that time step ($\boldsymbol{\Psi}_{i+1}$) should be approximately zero, as

shown below:

$$\boldsymbol{\Psi}_{i+1} = \int_V \boldsymbol{B}^T \boldsymbol{\sigma}_{i+1} dV - \boldsymbol{R}_i = \boldsymbol{0} \tag{2.20}$$

where $\boldsymbol{R}_i$ is the load vector, V is the volume of the body, $\boldsymbol{B}$ is a matrix defined as $\boldsymbol{B} = \boldsymbol{LN}$ such that $\Delta\boldsymbol{\varepsilon} = \boldsymbol{B}\boldsymbol{a}_{i+1}$. The stress change over the time step may be written as

$$\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i + \Delta\boldsymbol{\sigma}_i \tag{2.21}$$

The value of $\Delta\boldsymbol{\sigma}_i$ can be found using Equation 2.17, which shows that the change in stress depends on the rate of viscoplastic straining. Since the numerical algorithm for MatCalc is intended to be stable, the value used for the viscoplastic strain rate is the value at the end of the time step. This makes the algorithm fully implicit and thus improves the stability, which allows MatCalc to handle a large variety of experimental conditions and material types. In particular, MatCalc will be able to handle constitutive equations with higher gradients in stress than it could using an explicit algorithm. In the fully implicit version, Equation 2.18 becomes

$$\Delta\boldsymbol{\varepsilon}_i^{vp} = \Delta t \dot{\boldsymbol{\varepsilon}}_{i+1}^{vp} = \Delta t \lambda_{i+1} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{2.22}$$

where λ_{i+1} is the magnitude of the viscoplastic strain rate at the end of the time step. Using a truncated Taylor's expansion of λ_{i+1} and mathematical manipulation (Smith, 2001), it is possible to derive the following linear system of equations that are solved to find the nodal degrees of freedom ($\boldsymbol{a}$):

$$\int_V \left(\boldsymbol{B}^T \boldsymbol{D}^{vp} \boldsymbol{B} dV\right) \boldsymbol{a}_i = \boldsymbol{R}_i - \int_V \boldsymbol{B}^T \boldsymbol{\sigma}_i dV + \int_V \boldsymbol{B}^T \Delta\boldsymbol{\sigma}^{vp} dV \tag{2.23}$$

with

$$\boldsymbol{D}^{vp} = \boldsymbol{D}\left[\boldsymbol{I} - \Delta t C_1 \lambda' \frac{\partial Q}{\partial \boldsymbol{\sigma}} \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \boldsymbol{D}\right], \lambda' = \frac{d\lambda}{dF} \tag{2.24}$$

$$\Delta\boldsymbol{\sigma}^{vp} = \Delta t C_1 \lambda \boldsymbol{D} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{2.25}$$

$$C_1 = [1 + \lambda' \Delta t (H_e + H_p)]^{-1} \tag{2.26}$$

(2.27) $H_e = \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \boldsymbol{D}(\frac{\partial Q}{\partial \boldsymbol{\sigma}})$

(2.28) $H_p = -\frac{\partial F}{\partial \kappa}\left(\frac{\partial \kappa}{\partial \boldsymbol{\varepsilon}^{vp}}\right)^T \frac{\partial Q}{\partial \boldsymbol{\sigma}}$

where $\boldsymbol{I}$ is the identity matrix.

Solving for $\boldsymbol{a}_i$ in Equation 2.23 provides a first estimate for the displacements. For subsequent passes within an equilibrium iteration loop, the finite element equations, which provide a correction $\Delta\boldsymbol{a}_i$ for $\boldsymbol{a}_i$, simplify to the usual elastic form (Zienkiewicz et al., 2005):

(2.29) $\int_V \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}dV\Delta\boldsymbol{a}_i = \boldsymbol{R}_i - \int_V \boldsymbol{B}^T\boldsymbol{\sigma}_i dV$

The equilibrium iteration loops ceases when the relative changes in displacement become small. This can be defined by the following convergence criterion:

(2.30) $\frac{||\Delta\boldsymbol{a}_i||}{||\boldsymbol{a}_i||} \leq \text{toler}$

where $||\boldsymbol{a}_i||$ represents the Euclidean norm of the vector $\boldsymbol{a}_i$.

After each iteration of solving for the displacements the local stresses and strains are updated. Error can remain in the stress calculation at this point and the return map algorithm (Zienkiewicz et al., 2005, Pages 103-104) is used to correct the stress value. This corrected stress value is used in subsequent iterations of solving for $\boldsymbol{a}$. The return map algorithm solves the following system of nonlinear equations:

(2.31) $\Delta\boldsymbol{\sigma_i} = \boldsymbol{D}(\Delta\boldsymbol{\varepsilon_i} - \Delta t\lambda_{i+1}\frac{\partial Q(\boldsymbol{\sigma})}{\partial \boldsymbol{\sigma}})$

(From Equations 2.21 and 2.22)

(2.32) $\gamma\varphi(F_{i+1}) - \lambda_{i+1} = 0$

(From Equation 2.18) The system of equations can be rewritten as a root finding problem:

(2.33) $f_1(\Delta\boldsymbol{\sigma}, \lambda) = \boldsymbol{D}\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\sigma} - \Delta t\lambda\boldsymbol{D}\frac{\partial Q}{\partial \boldsymbol{\sigma}} = 0$

(2.34) $f_2(\Delta\boldsymbol{\sigma}, \lambda) = \varphi(F) - \dfrac{\lambda}{\gamma} = 0$

Newton's method is used to find the root of the above system. This is formulated as follows:

$$\boldsymbol{J}\delta\boldsymbol{x} = -\boldsymbol{F}(\boldsymbol{x})$$

Where $\boldsymbol{x}$ is the following 7D vector:

$$\begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{yz} & \sigma_{xz} & \lambda \end{bmatrix}^T$$

$\boldsymbol{F}$ is

(2.35) $\begin{bmatrix} f_1 f_2 \end{bmatrix}^T$

$\boldsymbol{J}$ is the Jacobian of $\boldsymbol{F}$:

(2.36) $\begin{bmatrix} \frac{\partial f_1}{\partial \boldsymbol{\sigma}} & \frac{\partial f_1}{\partial \lambda} \\ \frac{\partial f_2}{\partial \boldsymbol{\sigma}} & \frac{\partial f_2}{\partial \lambda} \end{bmatrix}$

Which expands to:

(2.37) $\begin{bmatrix} -\boldsymbol{I} - \Delta t \lambda \boldsymbol{D} \frac{\partial^2 Q}{\partial \boldsymbol{\sigma}^2} & \Delta t \boldsymbol{D} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \\ \frac{d\varphi}{dF} \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T & \frac{-1}{\gamma} \end{bmatrix}$

In which $\frac{\partial^2 Q}{\partial \boldsymbol{\sigma}^2}$ has the following structure:

(2.38) $\begin{bmatrix} \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{xx}} & \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{yy}} & \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{zz}} & \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{xy}} & \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{yz}} & \frac{\partial^2 Q}{\partial \sigma_{xx} \sigma_{xz}} \\ \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{xx}} & \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{yy}} & \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{zz}} & \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{xy}} & \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{yz}} & \frac{\partial^2 Q}{\partial \sigma_{yy} \sigma_{xz}} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{xx}} & \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{yy}} & \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{zz}} & \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{xy}} & \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{yz}} & \frac{\partial^2 Q}{\partial \sigma_{xz} \sigma_{xz}} \end{bmatrix}$

The convergence criterion is defined as follows:

(2.39) $max(\dfrac{||\Delta(\Delta\boldsymbol{\sigma})||}{||\Delta\boldsymbol{\sigma}||}, \dfrac{\Delta\lambda}{\lambda}) \leq \text{toler}$

The increment in the viscoplastic strain is found using Equation 2.22 and the value of λ determined in the return map algorithm. Similarly, the increment in the stress determined by the return map algorithm is used. After solving for the displacements for a given time step the local stresses and strains are updated. Pseudo code for the numerical integration algorithms can be found in Appendix A.

The equations given above are generic for many different viscoplastic material behaviours. F, Q, κ are all used in a generic way. At this point a material modelling expert would normally work out the various gradients by hand and then proceed to the implementation. These derivations are potentially time consuming and error prone and they require a solid understanding of tensors, invariants and vector calculus. The goal of the current work is to automatically go from the equations to the implementation. In the next Chapter we will discuss in detail how this goal was accomplished.

Chapter 3

High Level Design and Implementation

In this chapter the high level design of MatGen and MatCalc are presented. MatGen generates material models from a description of the material model, which is written using a DSL. The new material model can be used by the more complex program MatCalc, which combines together the following: material models, experiments, and numerical integration methods to simulate simple material experiments. This chapter first presents a high level introduction to MatGen and MatCalc, followed by a more detailed description of the implementation of both programs.

3.1 High Level Design of MatGen

MatGen is responsible for automatically generating source code that describes the new material model, which can be used by MatCalc. Currently MatGen only outputs source code implementing the interface expected by MatCalc, but this is not an inherent limitation of MatGen. The generated interface could easily be modified to match the requirements of another material model simulation program. A material model is defined by the four functions F, Q, κ, and φ but the numerical algorithms that are used to simulate the experiments, as described in Chapter 2, require derivatives of these functions, such as $\frac{\partial F}{\partial \boldsymbol{\sigma}}$. So, a useful material model not only includes the four functions but also all needed expressions which

are derived from the functions given by the user. The first and most important design goal of MatGen was that the user not be required to compute all of the expressions needed for a numerical simulation, MatGen should perform this tedious and error prone work for the user. A language is needed which can represent both the functions and any derived expressions. A tool is also needed which can compute the needed derivatives. The second design goal of MatGen was that user would not be required to write a program implementing the material model. This meant that MatGen must output code, such as a C++ class that provides numerical access methods describing the material model sufficiently for the numerical integration algorithms. Another compiler is needed which can translate the material model into source code form. By using a DSL and providing a compiler which can go from the definitions of the four functions to working source code MatGen allows novice material modellers to develop new material models.

3.2 High Level Design of MatCalc

MatCalc is responsible for simulating experiments on material models. MatCalc consists of four main modules: material behaviour, experiment control, numerical integration and data table. Each of these modules define an abstract interface that a concrete implementation must satisfy. Because of the abstract nature of these modules, many different material models, experiments and numerical integration methods can be used together. The first module, the material behaviour module, provides concrete details of a material behaviour model to the numerical integration algorithm. The second module, the experiment control module, provides a set of constraints on the experimental specimen that the numerical integration algorithm uses. The third module, the numerical integration module, simulates the experiments forward in time using the material model and experiment control modules to provide the necessary details. The final module is responsible for storing the output of the numerical integration algorithms in the form of a table. Each row in the table stores the simulation state for a given point in time. Figure 3.1 shows interaction between the

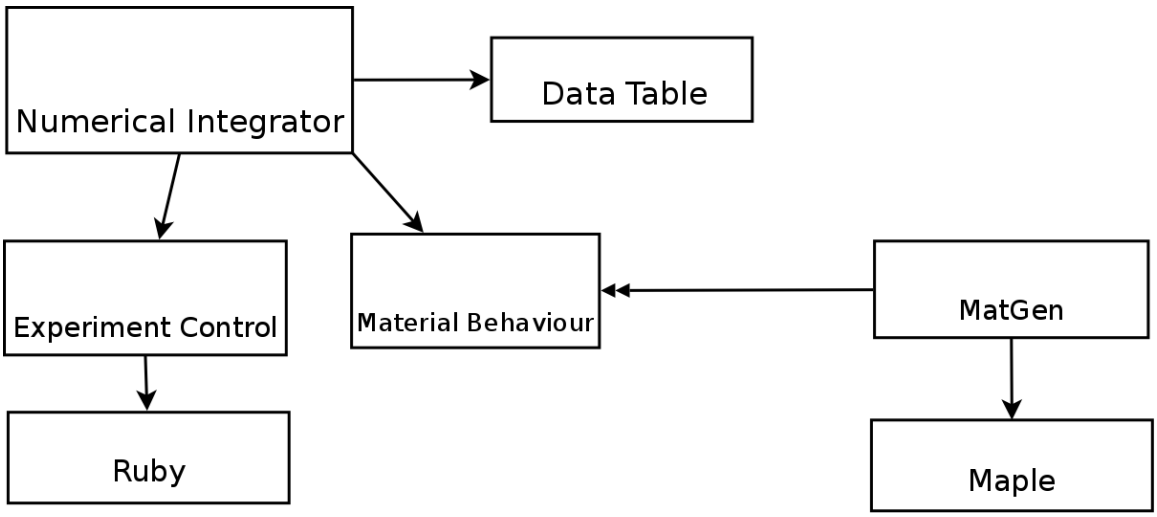

Figure 3.1: Module interaction diagram

modules (single headed arrow implies usage, double headed arrow indicates output from MatGen.) The following subsections will provide more details for each of these modules. Each module is summarized by giving its secret (Parnas et al., 1984) and then a more detailed description.

3.2.1 Material Behaviour

Module Secret: Algorithms to assist with computing the material model's state.

Module Discussion: The material behaviour module offers high level access to material specific terms found in the numerical integration algorithm. The interface was designed by studying the numerical algorithm found in Section 2.3 and providing accessor functions to material dependent expressions found therein. For example, the interface includes a function that provides the numerical integration algorithm with the matrix $\boldsymbol{D}^{vp}$ seen in Equation 2.24. Similarly, other material model specific equations defined in Section 2.3 are provided by the material behaviour interface. The main design goal was to make writing the numerical integration algorithms as easy as possible, so instead of providing access to low level expressions such as $\frac{\partial F}{\partial \sigma_{xx}}$ the interface was designed to provide very high level

access to the sub expressions of the numerical algorithm, such as $\boldsymbol{D}^{vp}$. A consequence of this decision is that it forces a very tight and fragile coupling between the needs of the numerical algorithm and the material behaviour module generated by MatGen. However, because of the generative nature of MatGen, it is relatively easy to add additional methods to the material behaviour module, in the case that they are needed by a new integration algorithm.

3.2.2 Experiment Control

Module Secret: Algorithm controlling the current constraints (boundary conditions) placed on the test specimen.

Module Discussion: This module controls the state of the test specimen during the experiment. Experiments can be load or displacement controlled. As well, the experiment state is not necessarily static and can vary over time. For example, the test specimen might be stretched 2 cm over the first second and then 4 cm for the rest of the experiment. Experiments should provide the ability to control whether or not the test specimen geometry is updated throughout the experiment. By allowing for geometry updates both the engineering and true stress and strain (Section 1.2.1) can be output. Updating is done by applying the predicted displacements to the corresponding coordinates. No updating leads to the engineering values, while updating produces the true values. To be optimally useful, experiments must be programmable. The experiment control interface was designed in such a way that both displacement and load controlled experiments can be implemented and the experiments can track time and dynamically modify the experiment's independent variables.

3.2.3 Data Table

Module Secret: Data structure for storing the state history of the test specimen.

Module Discussion: The data table is a complete history of the state of the test specimen

Time	σ_{xx}	...	ε_{xx}	...
0.01	0.0136	...	0.001	...
0.02	0.0454	...	0.002	...
0.03	0.0812	...	0.003	...
...	...	...	...	...

Table 3.1: Example data table

over the course of the experiment. The numerical integration algorithm fills in this table as the experiment simulation progresses. Each row in the table represents a discrete point in time and includes columns recording such state data as the stress ($\boldsymbol{\sigma}$) and strain ($\boldsymbol{\epsilon}$). An example data table can been seen in Table 3.1. Additionally, the real values assigned to each constant used by the material model and experiment are stored in the data table.

3.2.4 Numerical Integrator

Module Secret: Algorithm for numerically integrating the experiment forward in time. **Module Discussion:** The numerical integration is responsible for simulating the experiment forward in time. It is responsible for the high level simulation control, relying on a material behaviour model and experiment control model to provide the missing concrete details. The numerical integration algorithm builds and maintains a data table (described in Section 3.2.3) describing the history of stress and strain for the experiment specimen. The numerical integration algorithm module is also an abstract interface; this gives the user the ability to use custom numerical algorithms. Users typically would want to replace the numerical algorithm when they know in advance certain characteristics about the material model or experiment that they can exploit to gain performance or accuracy advantages.

3.3 Implementation of MatGen

As stated in Section 3.1 a DSL and compiler are needed that together can represent mathematical expressions, perform differentiation on the mathematical expressions, and compile the expressions into a language used by MatCalc. The DSL accepted by MatGen is defined in Appendix B. It is a very minimal subset of the language accepted by the Maple computer algebra system. Maple acts as a compiler to perform the symbolic differentiation and convert from the mathematical expressions into C expressions. Details of the output can be found in Chapter 4.

MatGen itself is a C++ program that produces a C++ class which defines a material model. The user provided definitions of F, Q, etc, are defined in the DSL. The needed mathematical expressions are built inside Maple. Following this the mathematical expressions are converted into C expressions using the Maple "CodeGeneration" function. These C expressions are inlined into the generated C++ class defining the material model. This C++ class is then compiled alongside MatCalc and used directly inside MatCalc when simulating experiments.

3.4 Implementation of MatCalc

In this section, details will be given on the implementation of MatCalc. MatCalc was designed as a library that can be used by other programs needing material model simulation. Using this library a simple GUI was developed to run experiments. There was a principle design rationale for MatCalc, to be as generic as possible by supporting many different experiments and material models with the same abstract interface. The rest of the section is divided into subsections covering the implementation of the following high level modules of MatCalc: Material Behaviour, Experiment Control and Numerical Integrator.

3.4.1 Material Behaviour

Material behaviour model modules are C++ classes that provide the numerical integration algorithm in MatCalc with concrete details of the material model being simulated. The interface that these classes expose was derived from the needs of the numerical integration algorithm defined in Section 2.3. MatGen can generate these C++ classes automatically or a user could write them manually. The class interface for the material behaviour module is summarized in Section 4.4.

3.4.2 Experiment Control

As stated earlier, the experiment control interface must be scriptable and capable of implementing both displacement controlled and load controlled experiments. The Ruby scripting language was chosen as the scripting language to implement experiments because of the ease with which it can be embedded into an existing C++ program. An instance of the Ruby interpreter is initialized by MatCalc and MatCalc communicates with an experiment instance loaded inside the Ruby interpreter. An added benefit of this is that MatCalc does not need to be recompiled when adding a new experiment. Details of the experiment control can be found in Chapter 4.

3.4.3 Numerical Integrator

There are three numerical integration algorithms implemented in MatCalc: elastic, viscoplastic and the returnmap algorithm. The first integrator, elastic, can only handle purely elastic materials. The second integrator, viscoplastic, can handle viscoplastic materials but does not correct the stresses back to the yield surface. The final integrator, the returnmap, is derived from the viscoplastic integrator but is fully implicit and it corrects the stresses back to the yield surface. The numerical integration algorithm used is decided by the user. A high level description of the viscoplastic and returnmap integration algorithms is found in Section 2.3 and pseudo code of the implementations can be found in Appendix A.

Chapter 4

Programmer's Manual

This chapter is a manual that covers how to use MatCalc and Matgen, how to add new experiments to MatCalc, how to add new material models to MatCalc, and how to use MatCalc as a library in one's own programs.

4.1 Using MatCalc

MatCalc includes a GUI program that allows users to perform experiments, tune experiment parameters, graph experimental data, and save experimental data so that it can be analyzed later. This section covers using this program, a later section covers using the MatCalc library. A screenshot of MatCalc is shown in Figure 4.1. In it, the user has selected a uniaxial extension or compression along the X axis as their experiment. The user has also selected the power law fluid (plfluid) material model. In the right pane the user can fill out any constant values that the experiment or material model requires. The process of using MatCalc to execute an experiment consists of the following steps:

1. Select experiment
2. Select material model
3. Fill in constants needed by experiment and material model

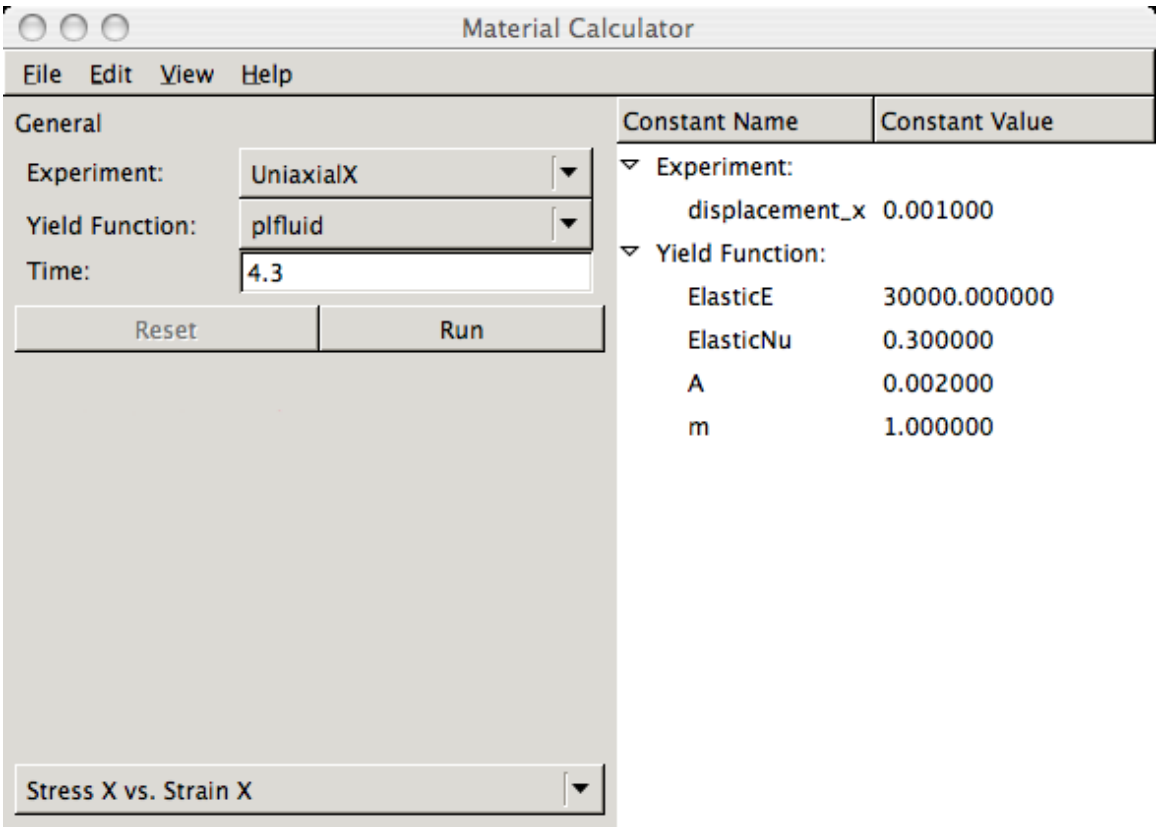

Figure 4.1: MatCalc screenshot

4. Select how long the experiment should be run for

5. Click the "Run" button

Once the simulation is complete the right pane will be replaced with a table of all experimental data, some of which can be graphed inside MatCalc. To graph experimental data, the user selects the type of graph in the lower left of the MatCalc window.

4.2 Using MatGen

MatGen includes a GUI program that allows the user to easily generate new material behaviour models based on the input of the functions F, Q, κ, φ and the constant γ. A

screenshot of MatGen can be seen in Figure 4.2. In the screenshot the user has provided the name "ViscoElastic" for the material model name, the functions F and Q are both J_2 (J_2 is a macro which will be discussed below), κ is 0.0 and φ is F, the constant gamma is $\frac{1}{2\eta}$. At the bottom users can add any constants that the expressions defining the functions rely on. ElasticE (E) and ElasticNu (ν) are always defined because they are needed by all material models. In this example eta (η) is added. The process of using MatGen to generate a new material behaviour model consists of the following steps:

1. Provide material behaviour name

2. Provide expressions for F, Q, κ, φ, and γ

3. Add any constants that are present in the above expressions

4. Click the "Generate" button

After the material model has been generated, C++ source and header files will be written with the file name of the material name followed by "_material." For instance, for the example shown in Figure 4.2, the file names would be "ViscoElastic_material.cc" and "ViscoElastic_material.h."
NOTE: Each function must be written in the DSL defined in Appendix B.

NOTE: You must make sure that all constant names used in function definitions are added to the list of constants or else the generated code will fail to compile.

Each function F,Q, etc receives different function arguments. Care must be taken that variables are used only when available. Variables available to each function are listed in the following tables: Table 4.1, Table 4.2, Table 4.3, Table 4.4 for F, Q, κ and φ, respectively. Usage of variables not included in a functions list can cause the generated code to fail to compile.

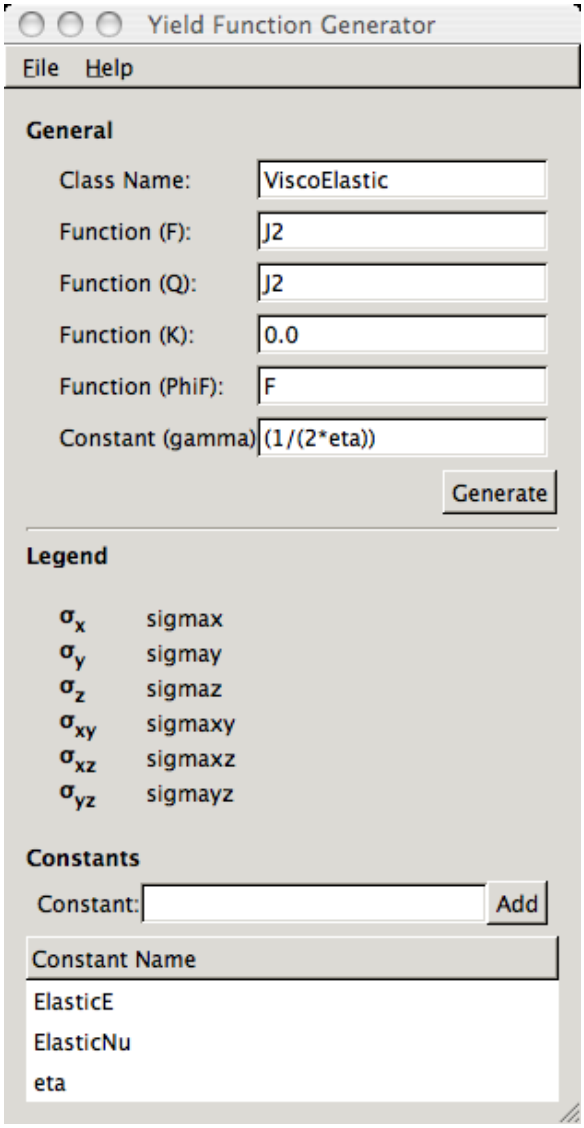

Figure 4.2: MatGen screenshot

Input Name	Input Description
SigmaXX	σ_{xx}
SigmaYY	σ_{yy}
SigmaZZ	σ_{zz}
SigmaXY	σ_{xy}
SigmaYZ	σ_{yz}
SigmaXZ	σ_{xz}
Kappa	κ

Table 4.1: Variables available to function F

Input Name	Input Description
SigmaXX	σ_{xx}
SigmaYY	σ_{yy}
SigmaZZ	σ_{zz}
SigmaXY	σ_{xy}
SigmaYZ	σ_{yz}
SigmaXZ	σ_{xz}

Table 4.2: Variables available to function Q

Input Name	Input Description
EpsilonVPXX	ε_{xx}^{vp}
EpsilonVPYY	ε_{yy}^{vp}
EpsilonVPZZ	ε_{zz}^{vp}
EpsilonVPXY	ε_{xy}^{vp}
EpsilonVPYZ	ε_{yz}^{vp}
EpsilonVPXZ	ε_{xz}^{vp}

Table 4.3: Variables available to function Kappa (κ)

Input Name	Input Description
F	F

Table 4.4: Variables available to function Phi (φ)

To simplify the notation, the DSL for MatGen also includes many macros, which get expanded into expressions that rely on the simulation variables. The current macros are summarized in Table 4.5. Again, care must be taken not to use a macro that includes a variable that is not available to the function being defined. The included macros are ones that have are used heavily in the field of continuum mechanics and are offered as a convenience as it is common for writing on the subject to make use of them. Table 4.5 is written using Maple syntax, the macro definitions in conventional mathematical notation typically used in the domain can be found in Section 5.4.1. In addition, each function has all of the user provided constants available as well. The C++ class interface that is generated by MatGen is given in detail in Section 4.5.

4.3 Adding an Experiment to MatCalc

This section discusses what must be done to add a new experiment to MatCalc. Since experiments in MatCalc are implemented by Ruby scripts adding an experiment consists of developing a Ruby class. The experiment class provides a way for the numerical integration algorithm to query which degrees of freedom are fixed and which are free, the prescribed displacements assigned to each node and the loads at each node. The numerical integration algorithm also calls the "setup" function in the experiment class immediately before beginning an experiment. At each discrete time point during the run of the simulation, the function "tick" is called. Experiment classes must implement the following interface:

– initialize: The initialize function is the constructor that all Ruby classes must implement.

Macro Name	Expansion
Sxx	(SigmaXX - (1/3)*(SigmaXX + SigmaYY + SigmaZZ))
Syy	(SigmaYY - (1/3)*(SigmaXX + SigmaYY + SigmaZZ))
Szz	(SigmaZZ - (1/3)*(SigmaXX + SigmaYY + SigmaZZ))
Sxy	(SigmaXY)
Syz	(SigmaYZ)
Sxz	(SigmaXZ)
Sm	(1/3)*(SigmaXX + SigmaYY + SigmaZZ)
J2	(1/2)*(Sxx^2 + Syy^2 + Szz^2 + 2 * (Sxy^2 + Sxz^2 + Syz^2))
J3	SigmaXX * SigmaYY * SigmaZZ - SigmaXX * SigmaYZ^2 + 2*SigmaXY *SigmaYZ *SigmaXZ - SigmaXY^2 * SigmaZZ - SigmaYY * SigmaXZ^2
q	(3 * J2)^1/2)
EVPxx	(EpsilonVPXX - (1/3)*(EpsilonVPXX + EpsilonVPYY + EpsilonVPZZ))
EVPyy	(EpsilonVPYY - (1/3)*(EpsilonVPXX + EpsilonVPYY + EpsilonVPZZ))
EVPzz	(EpsilonVPZZ - (1/3)*(EpsilonVPXX + EpsilonVPYY + EpsilonVPZZ))
EVPxy	(EpsilonVPXY)
EVPyz	(EpsilonVPYZ)
EVPxz	(EpsilonVPXZ)
J2EVP	(1/2) * (EVPxx^2 + EVPyy^2 + EVPzz^2 + 2 * (EVPxy^2 + EVPxz^2 + EVPyz^2))
EqVP	((4/3) * J2EVP)^1/2)

Table 4.5: Macros available to material model functions

- setup (dataset): The setup method is called once right before the experiment is to begin. This method takes a single argument pointing to the dataset (Section 3.2.3) that will be used in the experiment about to be run. It has no return value.

- tick(datset, dt, ct): The tick method is called once at the beginning of each discrete time step during the simulation. The tick is used to facilitate experiments dynamically modifying their state based on the amount of time that has passed since the beginning of the experiment. This method takes three arguments: An instance of the experiment dataset, the change in time and the current elapsed time of the experiment. It has no return value.

- get_constraints (dataset, dt): The get_constraints method is called whenever the numerical integration algorithm needs to check which nodal degree of freedom is free or constrained. A constrained nodal degree of freedom implies that the displacement value obtained from the get_displacements method must be used in the right hand side of Equation 2.23. A 24D vector containing boolean values must be returned. True is interpreted as constrained.

- get_displacements (dataset, dt): The get_displacements method is called whenever the numerical integration algorithm needs the nodal displacements of the constrained degrees of freedom. It takes two arguments: An instance of the experiment dataset and the change in time. A 24D vector must be returned with the displacement constraints that are applied to each nodal degree of freedom. In the case of a purely force controlled experiment the zero vector can be returned because the prescribed displacements are not used. Entries in this vector are used as the right hand side of Equation 2.23.

- get_forces (dataset): The get_forces method is called whenever the numerical integration algorithm needs the nodal force loads. It takes a single argument consisting of an instance of the experiment dataset. A 24D vector must be returned with the force load that is applied to each nodal degree of freedom. In the case of a purely

displacement controlled experiment the zero vector must be returned so that the force vector used by the numerical integration algorithm is zero.

– get_constants: The get_constants method is called whenever there is a need for the list of constant names that this experiment requires. The return value is an array containing strings naming each required constant. The values assigned to the constants are stored as a mapping between the names returned from this function and the values assigned to them in the data table. The names are used for the GUI and for consistency checking.

– update_geometry (dataset): The update_geometry method returns true or false controlling whether or not the numerical integration algorithm should update the test specimen geometry at the end of each time step. When the numerical integrator updates the test specimen geometry it is computing the true stress and strain values because it is taking into consideration the current configuration of the test specimen and not the original.

These methods define an interface that allows a wide variety of experiments to be implemented. Including displacement controlled, load controlled, or a combination of the two. Experiments can change their state over the course of time and control whether or not the experiment subject geometry is updated. Using the Ruby programming language allows experiments to be as complex as the user desires. An extreme example would be for the virtual experiment to communicate with a real experiment and attempt to mimic the real test specimen state. Example experiments include uniaxial extensions and compressions. Many concrete examples can be found in the source code for MatCalc. As an example, the uniaxial extension experiment follows:

```
class UniaxialX
        def initialize
                @Node1 = 0 # @NodeX is shorthand for referencing
                @Node2 = 3 # a specific degree of freedom
```

```
        @Node3 = 6
        @Node4 = 9
        @Node5 = 12
        @Node6 = 15
        @Node7 = 18
        @Node8 = 21
        @Xaxis = 0
        @Yaxis = 1
        @Zaxis = 2
end
def setup (ds)
end
def tick (ds, dt, ct)
end
def get_displacements (ds, dt)
        xyz = MVector.new 24
        dx = ds.get_constant("displacement_x")
        for i in 0..23
                xyz[i] = 0.0;
        end
        xyz[@Node2 + @Xaxis] = dx * dt
        xyz[@Node3 + @Xaxis] = dx * dt
        xyz[@Node6 + @Xaxis] = dx * dt
        xyz[@Node7 + @Xaxis] = dx * dt
        return xyz
end
def get_constraints (ds, dt)
        constraints = Array.new(24)
        for i in 0..23
                constraints[i] = true;
```

```
            end
            # Node N3,N4,N7,N8 can move in the Y direction
            constraints[@Node3 + @Yaxis] = false
            constraints[@Node4 + @Yaxis] = false
            constraints[@Node7 + @Yaxis] = false
            constraints[@Node8 + @Yaxis] = false
            # Node N5,N6,N7,N8 can move in the Z direction
            constraints[@Node5 + @Zaxis] = false
            constraints[@Node6 + @Zaxis] = false
            constraints[@Node7 + @Zaxis] = false
            constraints[@Node8 + @Zaxis] = false
            return constraints
        end
        def get_forces (ds)
            xyz = MVector.new 24
            for i in 0..23
                xyz[i] = 0.0;
            end
            return xyz
        end
        def get_constants
            constants = Array.new;
            constants.push("displacement_x")
            return constants
        end
        def update_geometry (ds)
            return false
        end
end
```

4.4 Adding a Material Model to MatCalc

All material behaviour classes must be compiled and linked into MatCalc so the process of adding a new material class is slightly more complex than that of adding an experiment. Adding a new material class requires working with the build system used by MatCalc. MatCalc uses Automake and Autoconf. Commonly referred together as Autotools or the GNU Build Tools (FSF, 2007). What follows is a very limited set of instructions to add a new material to the build system. Assuming that one has name_material.h and name_material.cc, the following steps should be taken:

- Step 1) Add your material class files to the build system:

  Add name_material.h and name_material.cc to MATERIAL_CLASSES variable in the source Makefile.am file

- Step 2) Add your material class to the internal list:

  In material_class_list.cc increment NUM_MATERIALS

  Add the line '{ "name" name_material_generator },' to the default material array

4.5 Writing a Material Class by Hand

In some cases the user may want to write a material model class by hand instead of using MatGen. A material class written by hand can be more efficient and handle some more complex constitutive equations than MatGen can generate code for. The material class interface includes the following functions:

- array of string get_constants()

  This function must return a vector of strings naming all the constants that this material class needs to function. The names are used for the GUI and consistency checks.

- scalar KappaF (dataset, epsilonVP)

  This function must return the scalar value obtained from evaluating the κ function.

– scalar F (dataset, stress, kappa)

This function must return the scalar value obtained from evaluating the F function.

– scalar Q (dataset, stress)

This function must return the scalar value obtained from evaluating the Q function.

– scalar get_gamma (dataset)

This function must return the scalar value of gamma.

– scalar PhiF (dataset, F)

This function must return the scalar value obtained from evaluating the φ function.

– vector depsilonvp (dataset, dt, F, stress, kappa)

This function must return the vector value obtained from evaluating $\Delta\varepsilon^{vp}$. This vector is defined in Equation 2.15.

– vector depsilonvpNL (dataset, dt, F, stress, kappa, lambda)

This function must return the vector value obtained from evaluating $\Delta\varepsilon^{vp}$. Takes a numerical version of lambda.

– vector dstressvp (dataset, dt, F, stress, kappa, epsilonVP)

This function must return the vector value obtained from evaluating $\Delta\sigma^{vp}$. This vector is defined in Equation 2.25.

– vector dstressvpNL (dataset, dt, F, stress, kappa, epsilonVP, lambda)

This function must return the vector value obtained from evaluating $\Delta\sigma^{vp}$. This vector is defined in Equation 2.25. Takes a numerical version of lambda.

– matrix get_De (dataset)

This function must return the 6D elastic matrix defined in Equation 2.13.

– matrix get_Dvp (dataset, dt, F, stress, kappa, epsilonVP)

This function must return the 6D Viscoplastic matrix defined in Equation 2.24.

Argument Name	Type
dataset	dataset
epsilonVP	m_vector
stress	m_vector
kappa	scalar
dt	scalar
F	scalar
lambda	scalar
dstress	m_vector
dstrain	m_vector

Table 4.6: Types of function arguments to material class

– matrix get_RM_Jacobian (dataset, dt, F, kappa, stress, lambda, epsilonVP)
  This function must return the Jacobian matrix used by the returnmap stress correction algorithm. This matrix is defined in Equation 2.36.

– vector get_RM_F (dataset, dt, F, stress, dstrain, dstress, lambda)
  This function must return the F vector used by the returnmap stress correction algorithm. This vector is defined in Equation 2.35.

The above interface was derived from the numerical integration algorithm presented in Section 2.3. It does not provide accessor functions for low level derivatives of the input functions such as $\frac{\partial F}{\partial \sigma_{xx}}$; instead it provides high level access which makes implementing the numerical algorithm easier. The functions which have the postfix "NL" (numerical lambda) use a numerical version of the variable λ provided by the numerical integration algorithm instead of the equation $\lambda = \gamma < \varphi(F) >$. This is needed for the Return Map integration algorithm. The types of the function arguments are given in Table 4.6.

4.6 Writing a MatCalc Based Driver Program

MatCalc was designed as a library to be used through a driver program. Below is a minimal driver program which illustrates how easy it is to incorporate into a larger program.

```
1  #include "brick_element.h"
2  #include "returnmap_integrator.h"
3  #include "dataset.h"
4  #include "experiment_list.h"
5  #include "material_class_list.h"
6
7  static material_class_list* mcl = NULL;
8  static experiment_list* el = NULL;
9
10 int
11 main (int argc, char **argv)
12 {
13 mcl = new material_class_list ();
14 el = new experiment_list ();
15 brick_element element;
16 m_vector xyz = get_brick_element_xyz (1.0, 0.25, 0.25);
17 dataset r;
18 r.add_constant("ElasticE", 0.26);
19 r.add_constant("ElasticNu", 0.30);
20 experiment_runner* experiment = el->get_instance ("Experiment");
21 material_class* material = mcl->get_instance ("Material");
22 returnmap_integrator rmi;
23 rmi.integrate (r, xyz, material, element, experiment, time);
24 }
```

For a program using the MatCalc library to function usefully it needs a dataset, test specimen, test specimen geometry, integrator, material class and an experiment. Lines 1

through 5 include the necessary header files: brick element, return map integration algorithm, data table, list of experiments and list of material behaviour modules, respectively. Lines 7 and 8 declare the lists of material classes and experiments and they are initialized on lines 13 and 14. Lines 15 and 16 setup the test specimen and the test specimen geometry. Line 17 initializes the dataset and lines 18 and 19 assign values for the constants "ElasticE" and "ElasticNu." An experiment runner (a class which handles communication between ruby and C++) is initialized in line 20 by asking the experiment list for the experiment named: "Experiment." Similarly the material model class is initialized in line 21 by asking the material model list for the material named: "Material." The numerical integration algorithm is initialized in line 22 and it is run in line 23. As the sample driver program illustrates it is straight forward to get an instance for most of these objects. Only the material class and experiment require more work than simply defining some variables. The recommended approach to obtain a material and experiment instance is to use their respective lists. There are separate material class and experiment lists and you can get an instance of any registered material class or experiment simply by passing the text name to the get_instance function. It is important to note that constants are stored in the dataset prior to running the experiment (see lines 18 and 19).

Chapter 5

Verification and Case Studies

This chapter will discuss the various methods used to verify both MatGen and MatCalc. As well, it will discuss case studies developed with MatCalc. Multiple methods of verification are needed to determine if MatGen and MatCalc are correct and performing adequately. MatGen has symbolic expressions that need to be verified as well as the generated code, which needs to be tested. MatCalc must be tested in a variety of ways as well. The low level modules of MatCalc are tested and a variety of case studies are developed to show the versatility as well as the accuracy of MatCalc. Sample experiments must be tested within MatCalc to verify that the numerical algorithm is functioning correctly.

This chapter will discuss all of these verification methods and case studies in the coming sections. First, verification of the symbolic mathematical expressions that MatGen generates will be discussed. Second, verifying the generated code at a unit level will be detailed. Third, low level unit testing and regression testing of MatCalc will be presented. Finally, case studies, which show the large variety of material models, that MatCalc can handle will be summarized. The case studies will also demonstrate the accuracy of MatCalc when this is possible, that is, when the true solution is known or when a separate program is available whose output can be compared to the output of MatCalc.

5.1 Verifying Symbolic Expressions

All of the symbolic processing is hidden from the user, thus we must be certain that it is functioning correctly. If the symbolic results are incorrect the side effects will cascade through MatGen and into MatCalc. The verification was performed by comparing symbolic output from Maple to hand derived versions of the same expressions. When deriving the expressions by hand many short cuts were taken by using the chain rule and invariants in place of the fully expanded expression. In many cases Maple was able to simplify the final expressions to be identical to the hand derived expressions. An example where this is the case is the derivation of the H_e term for a sample material model, as shown in Appendix C. In these cases it was trivial to verify that the symbolic computation was correct. In other cases Maple was unable to simplify the expressions to be identical. In these cases the expressions were found to be equivalent by verifying that they numerically agreed.

5.2 Verifying Numerical Results at a Low Level

MatGen generates C++ code that evaluates to the numeric result of the symbolic expressions. The generated source code needs to be tested for correctness. This testing was done by first developing a unit test fixture for the C++ unit testing framework (Robbins, 2007). This test fixture compares the numerical output of each method of two different material model classes. The list of methods checked can be found in Section 4.5. All the mathematical expressions needed to implement a material model were derived by hand and then a C++ class was written, also by hand. MatGen generated a class for the same material model. The material model defined in Appendix C was used. The relative difference was calculated using the following formula:

$$RelativeDifference = \frac{||Generated - ByHand||}{||ByHand||} \tag{5.1}$$

Where $||\cdot||$ is the euclidean norm. All methods tested had a relative difference of

10^{-6} or less. These values are small enough that they can be considered irrelevant in most contexts.

5.3 Unit and Regression Testing

As stated in the previous section, a unit test fixture was developed so that two different material model class implementations can be compared with each other. This same test fixture allows for regression testing to be performed between successive versions of MatGen. The version of the material model class generated by MatGen version N-1 can be compared to the material model class generated by MatGen version N. This was used as a safety check during the development of MatGen. The comparison is performed with random input. Also, a unit test fixture was developed for performing unit tests and regression testing on the low level matrix and vector classes. All public methods exposed by both of those classes are tested (a distinct test case exists for each method) and compared with "true" solutions found using MatLab.

5.4 Verifying Case Studies

As a final form of verification, case studies were developed as a means to both verify that the numerical algorithm was working correctly and to demonstrate the large range of material models that MatGen and MatCalc can handle. Some of the case studies in the following subsections have known closed form solutions. For those case studies it was found that the numerical algorithm used in MatCalc gave correct results. For those cases studies where there is no known closed form solution two testing methodologies were used. First, in some case studies the shape of the stress vs. strain graph is well known and this shape can be used as a qualitative measure of correctness. Second, for remainder of the case studies the output of a separate program, NonIso, developed as part of Smith (2001), was compared to MatCalc's output. NonIso is only capable of simulating one specific material

Name	Value	Meaning	Units
L	0.100	Length of brick along x axis	m
H	0.050	Length of brick along y axis	m
W	0.050	Length of brick along z axis	m
ν	0.300	Poisson's Ratio	-
ΔX	0.001	Displacement per second along X axis	$\frac{m}{s}$
$\dot{\varepsilon}_{xx}$	0.01	Constant Strain Rate	$\frac{1}{s}$

Table 5.1: Common material properties

model and does so in quasi 3 dimensions, but under the correct setup, that is both NonIso and MatCalc are simulating a uniaxial extension, the results from NonIso and MatCalc are comparable. Parallel testing was performed between MatCalc and NonIso (more detail can be found in Section 5.4.2).

5.4.1 Case Studies

A uniaxial extension along the x axis was performed for the following material models: elastic, viscoelastic, power-law viscosity and strain hardening. A relaxation experiment was also performed. All experiments were at a constant rate of strain, except for the relaxation test, which started at a constant rate of strain, but then held the specimen at its final length without allowing further straining.

For each of the proceeding case studies the material definitions that are given as input to MatGen, as well as the values assigned for the needed constants, are defined. For each case study the graph obtained from MatCalc is given as well as a discussion of the results in more detail. All the case studies share some material constants, dimensions and experimental parameters. These common values are listed in Table 5.1.

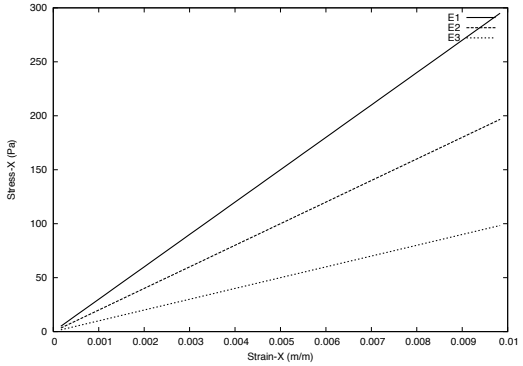

Figure 5.1: Linear Elasticity $E_1 > E_2 > E_3$

5.4.1.1 Elastic Case Study

Figure 5.1 shows an elastic material model with three different values of E. So that yielding will not occur, in the elastic case the yield stress is set very high; that is, F will remain below zero for the entire test. The material constants used are $E_1 = 30000.0Pa$, $E_2 = 20000.0Pa$, and $E_3 = 10000.0Pa$. The slope of stress vs. strain graph can be used to verify the results. The true slope should be equal to E_1, E_2 and E_3 and in all three cases the relative error was zero; that is, there was perfect agreement with the true slope. This perfect agreement is not surprising considering that the material behaviour is strictly linear.

5.4.1.2 Viscoelastic Case Study

Figure 5.2 shows viscoelastic material behaviour, with three different relaxation times (λ). Relaxation time is the measure of how quickly the elastic stress relaxes. The smaller the relaxation time the closer the material is to a true viscous fluid. The formula for λ is defined as:

$$\lambda = \frac{2.0\eta}{E} = \frac{6000.0}{E}$$

The relaxation times used are $\lambda_1 = 0.2$, $\lambda_2 = 2.0$, and $\lambda_3 = 20.0$. To determine several λ_i values we assumed η as $3000.0Pa \cdot s$ and then varied E.

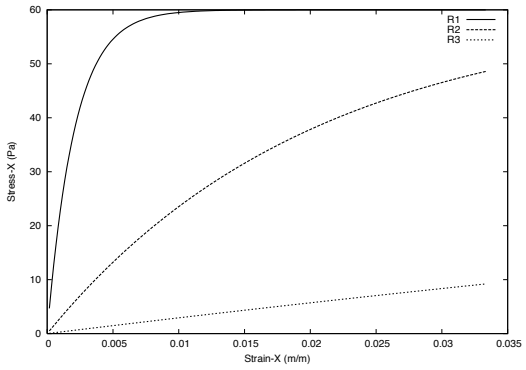

Figure 5.2: Viscoelasticity $\lambda_1 < \lambda_2 < \lambda_3$

The material model definition provided to MatGen is as follows:

$$F = \sqrt{3J_2}$$

$$Q = \sqrt{3J_2}$$

$$\varphi = F$$

$$\gamma = \frac{1}{2\eta}$$

Where J_2 is defined in Equation C.1. As the function F is greater than zero for any stress, this material model yields immediately upon loading.

The true solution in this experiment is given by the following formula:

$$\sigma_{xx} = 2\eta\dot{\varepsilon}_{xx}\left(1 - e^{\left(\frac{-t}{\lambda}\right)}\right)$$

The relative error for the three experiments λ_1, λ_2 and λ_3 was 0.53%, 0.23% and 0.04% respectively.

A relaxation time experiment was also performed on this material model, where the uniaxial extension test was performed for one second and then held still for the remainder of the experiment time. The results can be seen in Figure 5.3. The relaxation time used

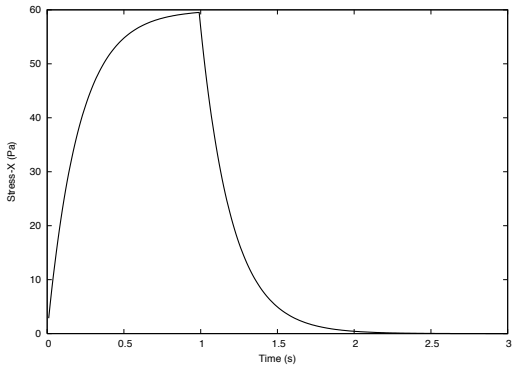

Figure 5.3: Relaxation time experiment

in this experiment was $\lambda = 0.2$. A true solution for the amount of time for the stresses to relax is known and the relative error was found to be 1.68%.

5.4.1.3 Power-Law Viscosity Case Study

Figure 5.4 shows a Power-law viscous material behaviour with three different powers (m) applied. The material definition provided to MatGen is as follows:

$$F = \sqrt{3J_2}$$

$$Q = \sqrt{3J_2}$$

$$\varphi = F^m$$

$$\gamma = A$$

The material constants are as follows: $A = 0.0002$, $m_1 = 1.4$ (shear thinning), $m_2 = 1.0$ (Newtonian viscosity), and $m_3 = 0.75$ (shear thickening.) A true solution for the asymptote of this material is known. The relative error between the experimental asymptote and the true solution was computed. The relative error measured for the three cases (m_i) were 0.00% 0.00% 1.53% repsectively.

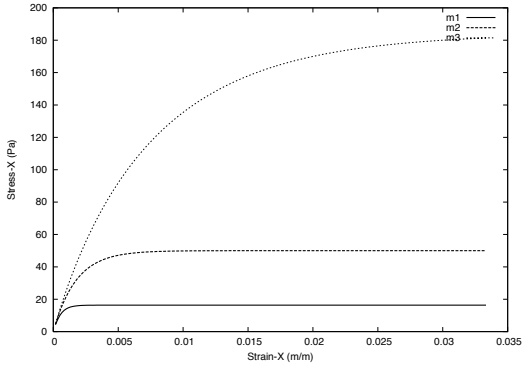

Figure 5.4: Power-law viscosity $m_1 > m_2 > m_3$

5.4.2 Case Study Comparing NonIso and MatCalc

This section consists of case studies that were run using both NonIso and MatCalc. The results are presented in the form of graphs where the results from both NonIso and MatCalc are graphed on the same axis. The material model includes a phenomenon known as strain hardening. Strain hardening is where as the viscoplastic strain accumulates the material becomes harder to deform. There is an analogue of strain softening in which the material becomes easier to deform as the viscoplastic strain accumulates. The material model definition given as input to MatGen follows:

$$F = nA^{\frac{1}{n}}\sqrt{3J_2}^{\frac{m}{n}}\kappa^{\frac{n-1}{n}}$$

$$Q = \sqrt{3J_2}$$

$$\kappa = \varepsilon_q^{vp}$$

$$\varphi = F$$

$$\gamma = 1.0$$

Where $\varepsilon_q^{vp} = \left(\frac{4}{3}J_2^{evp}\right)^{\left(\frac{1}{2}\right)}$ (effective viscoplastic strain) and

$$J_2^{evp} = \frac{1}{2}\left(\varepsilon_{xx}^{evp2} + \varepsilon_{yy}^{evp2} + \varepsilon_{zz}^{evp2} + 2\left(\varepsilon_{xy}^{evp2} + \varepsilon_{xz}^{evp2} + \varepsilon_{yz}^{evp2}\right)\right)$$

Name	Value	Meaning	Units
L	0.100	Length of brick along x axis	m
H	0.500	Length of brick along y axis	m
W	0.100	Length of brick along z axis	m
ν	0.300	Poisson's Ratio	-
ΔX	0.01	Displacement per second along X axis	$\frac{m}{s}$
$\dot{\varepsilon}_{xx}$	10.0	Constant Strain Rate	$\frac{1}{s}$

Table 5.2: Common material properties for NonIso vs. MatCalc case study

Also,

$$\varepsilon_{xx}^{evp} = \varepsilon_{xx}^{vp} - \frac{1}{3}\left(\varepsilon_{xx}^{vp} + \varepsilon_{yy}^{vp} + \varepsilon_{zz}^{vp}\right)$$
$$\varepsilon_{yy}^{evp} = \varepsilon_{yy}^{vp} - \frac{1}{3}\left(\varepsilon_{xx}^{vp} + \varepsilon_{yy}^{vp} + \varepsilon_{zz}^{vp}\right)$$
$$\varepsilon_{zz}^{evp} = \varepsilon_{zz}^{vp} - \frac{1}{3}\left(\varepsilon_{xx}^{vp} + \varepsilon_{yy}^{vp} + \varepsilon_{zz}^{vp}\right)$$
$$\varepsilon_{xy}^{evp} = \varepsilon_{xy}^{vp}$$
$$\varepsilon_{yz}^{evp} = \varepsilon_{yz}^{vp}$$
$$\varepsilon_{xz}^{evp} = \varepsilon_{xz}^{vp}$$

The constant values for all experiments can be found Table 5.2.

Four different experiment runs are given. Two of the four runs do not update the test specimen geometry (Section 3.2.2), and thus do not measure true stress and strain. The other two do update the test specimen geometry, and thus do measure true stress and strain. The value of n is also changed. The lower value of n was experimentally determined to be 0.6. The graphs are given in Figures: 5.5 and 5.6. The results from NonIso and MatCalc are coincident in the graphs. A difference measurement is given for each experiment by measuring the relative difference of the stress values over the entire experiment. The relative difference was computed with the following formula:

Description	**Relative Difference (%)**
$n = 1.5$ No Geometry Update	0.254
$n = 0.6$ No Geometry Update	0.295
$n = 1.5$ Geometry Update	0.302
$n = 0.6$ Geometry Update	0.285

Table 5.3: Relative difference for NonIso vs. MatCalc case study

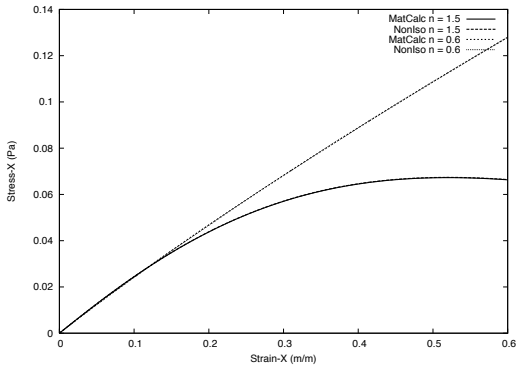

Figure 5.5: NonIso vs. MatCalc

$$RelativeError = \frac{||MatCalc - NonIso||}{||NonIso||} * 100\%$$

where, again, $|| \cdot ||$ is the euclidean norm. The relative errors are given in Table 5.3.

The relative difference between NonIso and MatCalc is a result of two important differences between the programs. First, NonIso is implemented in 2.5D, not 3D like MatCalc. Second, they use two different numerical algorithms.

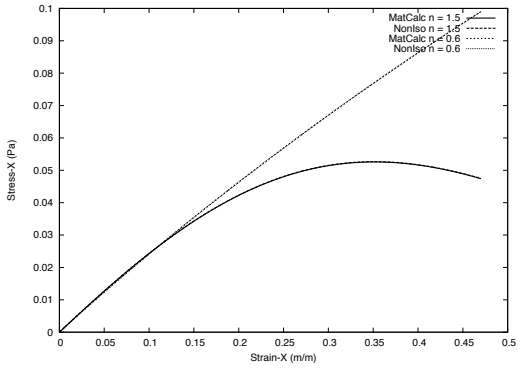

Figure 5.6: NonIso vs. MatCalc (Geometry Update)

Chapter 6

Conclusion

This chapter is divided into two sections. The first section will summarize the work found in this thesis as well as highlight key contributions therein. The second section will introduce some potential items for future work.

6.1 Contributions

This thesis has presented two programs: MatGen and MatCalc. The programs together offer a highly flexible, generic, scriptable virtual material testing laboratory that can be employed by users who have a second year engineering mechanics background as opposed to graduate level computational mechanics. This thesis has presented a generative approach to virtual material testing. As stated, being able to model the response of different materials under various loading histories is of critical importance to scientists and engineers. Scientists and engineers rely on having an accurate understanding of how materials behave for their work. A virtual laboratory, MatCalc, was developed to aid researchers in understanding material behaviour. MatCalc allows researchers to quickly perform simple experiments on a virtualized test specimen for a variety of material models. MatCalc can also be of benefit to students studying material sciences by allowing them to perform experiments outside of a real laboratory. This freedom can quicken the rate at which students develop an un-

derstanding of material behaviours. The numerical integration algorithms implemented in MatCalc support the following material behaviours: elastic, viscous, shear-thinning, shear-thickening, strain hardening, viscoelastic, viscoplastic and plastic. MatCalc also includes a programmable experiment system, experiments are written in the Ruby programming language and allow for experiments that can dynamically modify their state and can be both load and displacement controlled. Although there are examples of virtual material testing laboratories such as VizCore (Hashash et al., 2002) and VirLab (Smith and Gao, 2005), they do not allow for new material models to be added to the system without reprogramming it. To remove this limitation the numerical integration algorithms are written abstractly and make use of an API to access concrete material model details when needed.

Even with the abstract material model, a material sciences expert is needed to derive key expressions for the numerical simulation and write a program implementing these expressions. To alleviate this, MatGen was created, where MatGen is a tool that automatically generates C++ source code for new material behaviours from simple material definitions provided in a subset of Maple syntax. MatGen uses Maple to perform the symbolic computation necessary and to generate C code from the symbolic expressions. MatCalc and MatGen together provide an easy to use and flexible virtual material testing laboratory. This is accomplished by using the power of symbolic computation and bridging a gap between three different programming languages (C++, Ruby and Maple). The flexibility and accuracy of MatGen and MatCalc are demonstrated with the various case studies presented in this thesis. A refined list of contributions that came from the development of these two programs follows:

– Generic material model abstract class specification – The abstract class interface that MatCalc uses can simulate a large variety of material behaviours such as: elastic, viscous, plastic, strain hardening, strain softening, shear thinning and shear thickening. This interface can be used by other programs in the field.

– Material model compiler – MatGen is a material model compiler which takes advantage of Maple to perform both the symbolic differentiation of, and code generation

for the generic material model class.

– Implementation of generic numerical integration algorithms – Generally numerical algorithms implemented in this field are written for a particular material model. It was necessary to develop the algorithm in terms of pseudo code that makes use of the generic material model to simulate a wide variety of material models. These algorithms can cover a large family of material models with no changes.

– A domain specific language (DSL) capable of defining the variety of material models – This DSL is sufficient to define all of the material models present in this thesis and many more.

– A generic framework for experiments via the finite element method – The use of a single 8-noded finite element can use the same algorithm to accommodate all load and displacement controlled experiments when the stress and the strain are constant throughout the specimen.

– A scriptable experiment control interface – This interface is capable of implementing experiments that can dynamically modify their state as well as supporting both load and displacement controlled experiments.

6.2 Future Work

There are many avenues for future work. Some of these will be discussed in the following list.

Inconsistency checking Before an experiment is executed it can be checked for consistency with the rules that govern the experiments. For example, if a nodal degree of freedom is constrained to a particular displacement the load for the degree of freedom should be zero and vice versa. This would require adding an error checker with domain specific knowledge to the experiment control code.

Investigate other numerical algorithms Other numerical algorithms should be investigated and implemented alongside the three that are currently included in MatCalc. Certain numerical algorithms can be advantageous for certain material models. It should also be determined which models work best with which integration algorithms.

Model fitting The material model could be fit to experimental data to help determine material model parameters that match the experimental data.

Generate code for low level access to material model The numerical algorithms access the material model at a high level. This makes the implementation of numerical algorithms trivial but has some potentially negative side effects, which were discussed in Section 3.2.1. The code generator could be extended to include more fine grained access to the material model.

Generate code targeting other material simulation engines MatGen could be modified so that it could generate code that targets material simulation engines other than MatCalc. These simulation programs could be for general purpose engineering computation and not just for simple laboratory experiments.

Temperature controlled experiments Experiments where the temperature is varied could be added allowing for non isothermal material models.

Visualization and animation The visualization and animation of the test specimen including its geometry, stresses, and strains could be added. Also, visualization and animation of the numerical algorithm itself, including correcting stresses back to the yield surface would be interesting.

New classes of material models Additional classes of material models could be added, such as integral constitutive equations, differential constitutive equations and hyperelastic constitutive equations.

Develop DSL for simple experiments Develop a DSL which can describe simple experiments. This DSL could be compiled into a Ruby script implementing the experiment.

Enhance symbolic differentation with Maple Currently symbolic differentation with Maple is done in a brute-force fashion. More subtle and complex methods could be used.

Improve GUI usability Improve the usability of the GUI programs by performing usability studies.

Appendix A

Numerical Algorithm Pseudo Code

This appendix includes pseudo code for both the viscoplastic and return map numerical integration algorithms discussed in Section 2.3.

A.1 Visoplastic Integration Algorithm

This section shows detailed pseudo code for the viscoplastic algorithm.

$ct = 0$

$\boldsymbol{D}e = material.get\_De()$

$\boldsymbol{R} = \mathbf{0}$

$\boldsymbol{\sigma} = \mathbf{0}$

$\boldsymbol{\varepsilon} = \mathbf{0}$

$\boldsymbol{\varepsilon}^{VP} = \mathbf{0}$

while $ct + dt \leq T$ **do**

$ct = ct + \Delta t$

$\boldsymbol{\sigma}^{TR} = \mathbf{0}$

$\Delta\boldsymbol{\varepsilon} = \mathbf{0}$

$\Delta\boldsymbol{\sigma} = \mathbf{0}$

$\Delta\boldsymbol{a} = \mathbf{0}$

$\boldsymbol{displacements} = \mathbf{0}$

$\boldsymbol{constraints} = \mathbf{0}$

$experiment.get\_displacements(\boldsymbol{displacements})$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int(\boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B})dV$

$\boldsymbol{internalF} = \int(\boldsymbol{B}^T\boldsymbol{\sigma})dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$ using displacements

$\Delta\boldsymbol{a} = K\backslash rhs$

$\Delta\boldsymbol{\varepsilon} = \boldsymbol{B}\Delta\boldsymbol{a}$

$\Delta\boldsymbol{\sigma} = \boldsymbol{D}\Delta\boldsymbol{\varepsilon}$

$\boldsymbol{\sigma}^{TR} = \boldsymbol{\sigma} + \Delta\boldsymbol{\sigma}$

$\kappa = material.KappaF(ds, \boldsymbol{\varepsilon}^{VP})$

$F = material.F(ds, \boldsymbol{\sigma}^{TR}, \kappa)$

if $F > 0.0$ **then**

$\boldsymbol{\varepsilon}_0 = \boldsymbol{\varepsilon}$

$\boldsymbol{\varepsilon}_0^{VP} = \boldsymbol{\varepsilon}^{VP}$

$\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}$

$\Delta\boldsymbol{a} = \mathbf{0}$

$error = 0$

$converged = false$

$implicitVP = true$

repeat

if $implicitVP$ **then**

$\boldsymbol{displacements} = \mathbf{0}$

$\boldsymbol{constraints} = \mathbf{0}$

$\boldsymbol{D_{vp}} = material.get\_Dvp(ds, dt, F, \boldsymbol{\sigma}, \kappa)$

$experiment.get\_displacements(\boldsymbol{displacements})$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int(\boldsymbol{B}^T \boldsymbol{D}vp\boldsymbol{B})dV$

$\boldsymbol{internalF} = \int(\boldsymbol{B}^T\boldsymbol{\sigma})dV$

$\Delta\boldsymbol{\sigma}^{VP} = material.dstressvp(F, \boldsymbol{\sigma}, \kappa)$

$\boldsymbol{FVP} = \int(\boldsymbol{B^T}\Delta\boldsymbol{\sigma}^{VP})dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF} + \boldsymbol{FVP}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$

$implicitVP = false$

else

$\boldsymbol{displacements} = \mathbf{0}$

$\boldsymbol{constraints} = \mathbf{0}$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int(\boldsymbol{B}^T \boldsymbol{D}e\boldsymbol{B})dV$

$\boldsymbol{i}nternalF = \int(\boldsymbol{B}^T\boldsymbol{\sigma})dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$ with zero displacements

end if

$\boldsymbol{d\Delta a} = \boldsymbol{K}/\boldsymbol{rhs}$

$error = \frac{|d\Delta a|}{|\Delta \boldsymbol{a}|}$

$converged = error < \epsilon$

$\Delta\boldsymbol{a} = \Delta\boldsymbol{a} + d\Delta\boldsymbol{a}$

$\Delta\boldsymbol{\varepsilon} = \boldsymbol{Bda}$

$\lambda = \gamma * \varphi(F)$

$\Delta\boldsymbol{\varepsilon}^{VP} = material.depsilonVP(F, \boldsymbol{\sigma}, kappa, \lambda)$

$\Delta\boldsymbol{\sigma} = \boldsymbol{D}(\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^{VP})$

$\Delta\boldsymbol{\varepsilon}^{VP} = material.depsilonvp(F, \boldsymbol{\sigma}, kappa)$

$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_0 + \Delta\boldsymbol{\varepsilon}$

$\boldsymbol{\varepsilon}^{VP} = \boldsymbol{\varepsilon}_0^{VP} + \Delta\boldsymbol{\varepsilon}^{VP}$

$\boldsymbol{\sigma} = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}$

$\boldsymbol{\sigma}^{TR} = \boldsymbol{\sigma}$

$\kappa = material.KappaF(ds, \boldsymbol{\varepsilon}^{vp})$

$F = material.F(ds, \boldsymbol{\sigma}, \kappa)$

until $converged$

else

$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon} + \Delta\boldsymbol{\varepsilon}$

$\boldsymbol{\sigma} = \boldsymbol{\sigma}^{TR}$

end if

end while

A.2 Return Map Integration Algorithm

This section shows detailed pseudo code for the return map algorithm.

$ct = 0$

$\boldsymbol{D}e = material.get\_De()$

$\boldsymbol{R} = \mathbf{0}$

$\boldsymbol{\sigma} = \mathbf{0}$

$\boldsymbol{\varepsilon} = \mathbf{0}$

$\boldsymbol{\varepsilon}^{VP} = \mathbf{0}$

while $ct + dt \leq T$ **do**

$ct = ct + dt$

$\boldsymbol{\sigma}^{TR} = \mathbf{0}$

$\Delta\boldsymbol{\varepsilon} = \mathbf{0}$

$\Delta\boldsymbol{\sigma} = \mathbf{0}$

$\Delta \boldsymbol{a} = \mathbf{0}$

$\boldsymbol{displacements} = \mathbf{0}$

$\boldsymbol{constraints} = \mathbf{0}$

$experiment.get\_displacements(\boldsymbol{displacements})$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int (\boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B}) dV$

$\boldsymbol{internalF} = \int (\boldsymbol{B}^T \boldsymbol{\sigma}) dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$ using displacements

$\Delta \boldsymbol{a} = K \backslash rhs$

$\Delta \boldsymbol{\varepsilon} = \boldsymbol{B} \Delta \boldsymbol{a}$

$\Delta \boldsymbol{\sigma} = \boldsymbol{D} \Delta \boldsymbol{\varepsilon}$

$\boldsymbol{\sigma}^{TR} = \boldsymbol{\sigma} + \Delta \boldsymbol{\sigma}$

$\kappa = material.KappaF(ds, \boldsymbol{\varepsilon}^{VP})$

$F = material.F(ds, \boldsymbol{\sigma}^{TR}, \kappa)$

if $F > 0.0$ **then**

  $\boldsymbol{\varepsilon}_0 = \boldsymbol{\varepsilon}$

  $\boldsymbol{\varepsilon}_0^{VP} = \boldsymbol{\varepsilon}^{VP}$

  $\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}$

  $\Delta \boldsymbol{a} = \mathbf{0}$

  $error = 0$

  $converged = false$

  $implicitVP = true$

  repeat

    if $implicitVP$ **then**

      $\boldsymbol{displacements} = \mathbf{0}$

      $\boldsymbol{constraints} = \mathbf{0}$

$\boldsymbol{D_{vp}} = material.get\_Dvp(ds, dt, F, \boldsymbol{\sigma}, \kappa)$

$experiment.get\_displacements(\boldsymbol{displacements})$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int(\boldsymbol{B}^T \boldsymbol{D}vp\boldsymbol{B})dV$

$\boldsymbol{internalF} = \int(\boldsymbol{B}^T\boldsymbol{\sigma})dV$

$\Delta\boldsymbol{\sigma}^{VP} = material.dstressvp(F, \boldsymbol{\sigma}, \kappa)$

$\boldsymbol{FVP} = \int(\boldsymbol{B}^T\Delta\boldsymbol{\sigma}^{VP})dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF} + \boldsymbol{FVP}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$

$implicitVP = false$

else

$\boldsymbol{displacements} = \mathbf{0}$

$\boldsymbol{constraints} = \mathbf{0}$

$experiment.get\_constraints(\boldsymbol{constraints})$

$\boldsymbol{K} = \int(\boldsymbol{B}^T \boldsymbol{D}e\boldsymbol{B})dV$

$\boldsymbol{internalF} = \int(\boldsymbol{B}^T\boldsymbol{\sigma})dV$

$\boldsymbol{rhs} = \boldsymbol{R} - \boldsymbol{internalF}$

constrain $\boldsymbol{K}$

constrain $\boldsymbol{rhs}$ with zero displacements

end if

$d\Delta\boldsymbol{a} = \boldsymbol{K}/\boldsymbol{rhs}$

$error = \frac{|d\Delta\boldsymbol{a}|}{|\Delta\boldsymbol{a}|}$

$converged = error < \epsilon$

$\Delta\boldsymbol{a} = \Delta\boldsymbol{a} + d\Delta\boldsymbol{a}$

$\Delta\boldsymbol{\varepsilon} = \boldsymbol{B}\Delta\boldsymbol{a}$

$rm\_converged = false$

$rm\_error = 0.0$

$\lambda = \gamma * \varphi(F)$

$\Delta\boldsymbol{\varepsilon}^{VP} = material.depsilonvp(F, \boldsymbol{\sigma}, kappa, \lambda)$

$\Delta\boldsymbol{\sigma} = \boldsymbol{D}(\Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^{VP})$

$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_0 + \Delta\boldsymbol{\varepsilon}$

$\Delta\boldsymbol{\varepsilon}^{VP} = material.depsilonvp(F, \boldsymbol{\sigma}, kappa, \lambda)$

$\boldsymbol{\varepsilon}^{VP} = \boldsymbol{\varepsilon}_0^{VP} + \Delta\boldsymbol{\varepsilon}^{VP}$

$\boldsymbol{\sigma} = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}$

$\kappa = material.KappaF(ds, \boldsymbol{\varepsilon}^{vp})$

$F = material.F(ds, \boldsymbol{s}igma, \kappa)$

repeat

$\boldsymbol{RM_F} = material.get\_RM\_F(dt, F, \boldsymbol{\sigma}, \Delta\boldsymbol{\varepsilon}, \lambda)$

$\boldsymbol{RM_J} = material.get\_RM\_Jacobian(dt, F, kappa, \boldsymbol{\sigma}, \lambda)$

$\boldsymbol{RM_X} = \boldsymbol{RM_J} / \left(-\boldsymbol{RM_F}\right)$

$\delta\boldsymbol{\sigma} = \boldsymbol{RM_{X\,1..6}}$

$\delta\lambda = \boldsymbol{RM_{X\,7}}$

$\Delta\boldsymbol{\sigma} = \Delta\boldsymbol{\sigma} + \delta\boldsymbol{\sigma}$

$\lambda = \lambda + \delta\lambda$

$\Delta\boldsymbol{\varepsilon}^{VP} = material.depsilonvp(F, \boldsymbol{\sigma}, kappa, \lambda)$

$\boldsymbol{\varepsilon}^{VP} = \boldsymbol{\varepsilon}_0^{VP} + \Delta\boldsymbol{\varepsilon}^{VP}$

$\boldsymbol{\sigma} = \boldsymbol{\sigma}_0 + \Delta\boldsymbol{\sigma}$

$\kappa = material.KappaF(ds, \boldsymbol{\varepsilon}^{vp})$

$F = material.F(ds, \boldsymbol{\sigma}, \kappa)$

$rm_{error} = MAX(\frac{|\delta\boldsymbol{\sigma}|}{|\Delta\boldsymbol{\sigma}|}, \frac{\delta\lambda}{\lambda})$

$rm_{converged} = rm\_error < \epsilon$

until $rm_{converged}$

until $converged$

else

$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon} + \Delta\boldsymbol{\varepsilon}$

$\boldsymbol{\sigma} = \boldsymbol{\sigma}^{TR}$

end if

end while

Appendix B

MatGen DSL

This appendix details the DSL that MatGen accepts as user input when defining the functions F, Q, κ, φ and the constant γ. The DSL is presented in Backus-Naur form, extended with some regular expression operations. Some of the simulation variables and simulation variable macros are only available when used in a function that accepts them as arguments. For example, F takes as arguments the $\boldsymbol{\sigma}$ vector and the value of κ so the expression given for F should only expect the $\boldsymbol{\sigma}$ and κ derived variables and the user provided constants to have meaningful values. For a more detailed explanation of which variables are available to each function see Chapter 4. Each of the four functions is defined by a single expression.

⟨expression⟩→⟨number⟩|
(⟨expression⟩)|
⟨expression⟩^⟨expression⟩|
⟨expression⟩∗⟨expression⟩|
⟨expression⟩/⟨expression⟩|
⟨expression⟩+⟨expression⟩|
⟨expression⟩−⟨expression⟩|
−⟨expression⟩|
sin(⟨expression⟩)|arcsin(⟨expression⟩)|cos(⟨expression⟩)|arccos(⟨expression⟩)|
ln(⟨expression⟩)|log(⟨expression⟩)|

⟨simulation-variable⟩|⟨simulation-variable-macros⟩|⟨user-defined-constants⟩

⟨number⟩→[⟨sign⟩]⟨digit⟩+[⟨decimal-point⟩⟨digit⟩+]

⟨sign⟩→**+**|**-**

⟨decimal-point⟩→**.**

⟨string⟩→⟨character⟩+

⟨character⟩→**a...z**|**A...Z**

⟨digit⟩→**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

⟨simulation-variable⟩→⟨simulation-variable-F⟩|⟨simulation-variable-Q⟩|⟨simulation-variable-Kappa⟩|⟨simulation-variable-Phi⟩

⟨simulation-variable-F⟩→**Kappa**|⟨simulation-variable-stress⟩|⟨simulation-variable-stress-macros⟩

⟨simulation-variable-Q⟩→⟨simulation-variable-stress⟩|⟨simulation-variable-stress-macros⟩

⟨simulation-variable-Kappa⟩→⟨simulation-variable-vp-strain⟩|⟨simulation-variable-vp-strain-macros⟩

⟨simulation-variable-Phi⟩→**F**

⟨simulation-variable-stress⟩→**SigmaXX**|**SigmaYY**|**SigmaZZ**|**SigmaXY**|**SigmaYZ**|**SigmaXZ**

⟨simulation-variable-stress-macros⟩→**Sxx**|**Syy**|**Szz**|**Sxy**|**Syz**|**Sxz**|**Sm**|**J2**|**J3**|**q**

⟨simulation-variable-vp-strain⟩→**EpsilonVPXX**|**EpsilonVPYY**|**EpsilonVPZZ**|**EpsilonVPXY**|**EpsilonVPYZ**|**EpsilonVPXZ**

⟨simulation-variable-vp-strain-macros⟩→**EVPxx**|**EVPyy**|**EVPzz**|**EVPxy**|**EVPyz**|**EVPxz**|**J2EVP**|**EqVP**

⟨user-defined-constants⟩→⟨string⟩

Operators follow the following precedence: (), ˆ, ∗, /, +, −.

The extended BNF syntax includes: [...] which denotes an optional portion of expression and + which denotes one or more repetitions. Bold signifies a terminal token.

Appendix C

Sample Derivation of H_e for a Material Model

This appendix serves two purposes. The first is to show how much manual labour is involved in deriving the necessary mathematical expressions needed to implement a numerical integration algorithm for a single material model. The second is to show that the final expression derived by an expert in the field using a variety of short cuts is (at least in some cases) the same expression that MatGen produces. To demonstrate both of these points the expression for H_e (See Equation 2.27) which is specific to the following material is derived:

$$F = q$$

$$Q = q$$

$$\varphi = F$$

$$\kappa = 0$$

$$\gamma = \frac{1}{2\eta}$$

Where $q = \sqrt{3J_2}$ (effective stress), $J_2 = \frac{1}{2}s_{ij}s_{ij}$ (second invariant of the deviatoric stress tensor) and s_{ij} is $\sigma_{ij} - \frac{1}{3}\sigma_{kk}\sigma_{ij}$ (deviatoric stress tensor). The subscripts here follow

the Einstein summation convention (Einsten, 1916). After some manipulation the expanded form of J_2 looks as follows:

(C.1) $J_2 = \frac{1}{2}(s_{xx}^2 + s_{yy}^2 + s_{zz}^2 + 2(s_{xy}^2 + s_{xz}^2 + s_{yz}^2))$

The expanded form of s_{ij} are as follows:

(C.2) $s_{xx} = (\sigma_{xx} - \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}))$

(C.3) $s_{yy} = (\sigma_{yy} - \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}))$

(C.4) $s_{zz} = (\sigma_{zz} - \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}))$

(C.5) $s_{xy} = (\sigma_{xy})$

(C.6) $s_{xz} = (\sigma_{xz})$

(C.7) $s_{yz} = (\sigma_{yz})$

In the later steps it will prove useful to recognize that, as the above shows,

(C.8) $s_{kk} = s_{xx} + s_{yy} + s_{zz} = 0$

The expression for the abstract version of H_e is $\left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \boldsymbol{D}(\frac{\partial Q}{\partial \boldsymbol{\sigma}})$. F and Q are part of the material model provided above and D is defined fully in Equation 2.13. Because F and Q and thus $\frac{\partial F}{\partial \boldsymbol{\sigma}}$ and $\frac{\partial Q}{\partial \boldsymbol{\sigma}}$ are identical, it is only necessary to derive $\frac{\partial F}{\partial \boldsymbol{\sigma}}$.

The first step in the derivation of H_e is to derive $\frac{\partial F}{\partial \boldsymbol{\sigma}}$. Using the index form, the expression looks as follows:

$$\frac{\partial F}{\partial \sigma_{ij}} = \frac{\partial q}{\partial \sigma_{ij}}$$

Using the chain rule the expression becomes:

$$\frac{\partial F}{\partial \sigma_{ij}} = \frac{\partial q}{\partial J_2}\frac{\partial J_2}{\partial \sigma_{ij}} = \sqrt{3}\left(\frac{\partial \sqrt{J_2}}{\partial J_2}\right)\frac{\partial J_2}{\partial \sigma_{ij}} = \sqrt{3}\left(\frac{1}{2\sqrt{J_2}}\right)\frac{\partial J_2}{\partial \sigma_{ij}}$$

$\frac{\partial J_2}{\partial \sigma_{ij}}$ can be found by again using the chain rule:

$$\frac{\partial J_2}{\partial \sigma_{ij}} = \frac{\partial J_2}{\partial s_{kl}} \frac{\partial s_{kl}}{\partial \sigma_{ij}}$$
$$\frac{\partial J_2}{\partial \sigma_{ij}} = s_{kl} \frac{\partial s_{kl}}{\partial \sigma_{ij}}$$
$$\frac{\partial J_2}{\partial \sigma_{ij}} = s_{kl} \left(\delta_{ki}\delta_{lj} - \frac{1}{3}\delta_{kl}\delta_{ip}\delta_{jp} \right)$$

Where δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ if $i = j$ and 0 otherwise)

The term $\delta_{ip}\delta_{jp}$ is equivelant to δ_{ij} (because in this case $i = j = p$) so,

$$\frac{\partial J_2}{\partial \sigma_{ij}} = s_{kl} \left(\delta_{ki}\delta_{lj} - \frac{1}{3}\delta_{kl}\delta_{ij} \right)$$
$$\frac{\partial J_2}{\partial \sigma_{ij}} = s_{ij} - \frac{1}{3}s_{kk}$$

But $s_{kk} = (s_{xx} + s_{yy} + s_{zz}) = 0$; therefore

$$\frac{\partial J_2}{\partial \sigma_{ij}} = s_{ij}$$

Therefore,

$$\frac{\partial q}{\partial \sigma_{ij}} = \frac{\sqrt{3}}{2\sqrt{J_2}} s_{ij} = \frac{3}{2q} s_{ij}$$

Which written in vector form is,

$$\frac{\partial q}{\partial \boldsymbol{\sigma}} = \frac{3}{2q} \begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & 2s_{xy} & 2s_{yz} & 2s_{xz} \end{bmatrix}^T$$

Therefore,

$$\frac{\partial Q}{\partial \boldsymbol{\sigma}} = \frac{\partial F}{\partial \boldsymbol{\sigma}} = \frac{3}{2q} \begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & 2s_{xy} & 2s_{yz} & 2s_{xz} \end{bmatrix}^T$$

Now that the subterms of H_e have been derived H_e can be constructed. To simplify the derivation, first let

$$\boldsymbol{vec} = \boldsymbol{D} \frac{\partial Q}{\partial \boldsymbol{\sigma}}$$

Performing the multiplication,

$$\boldsymbol{vec} = \frac{3E}{2dq}\begin{bmatrix} (1-\eta)s_{xx} + \eta s_{yy} + \eta s_{zz} \\ \eta s_{xx} + (1-\eta)s_{yy} + \eta s_{zz} \\ \eta s_{xx} + \eta s_{yy} + (1-\eta)s_{zz} \\ (1-2\eta)s_{xy} \\ (1-2\eta)s_{yz} \\ (1-2\eta)s_{xz} \end{bmatrix}$$

$$d = (1+\nu)(1-2\nu)$$

$\boldsymbol{vec}$ can be simplified using the following equations:

$$s_{zz} = -(s_{xx} + s_{yy})$$

$$s_{xx} = -(s_{yy} + s_{zz})$$

$$s_{yy} = -(s_{xx} + s_{zz})$$

Substituting the above equations into $\boldsymbol{vec}$ yields:

$$\boldsymbol{vec} = \frac{3E}{2(1+\nu)q}\begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & s_{xy} & s_{yz} & s_{xz} \end{bmatrix}^T$$

But the shear modulus, $G = \frac{E}{2(1+\nu)}$

Therefore,

$$\boldsymbol{vec} = \frac{3G}{q}\begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & s_{xy} & s_{yz} & s_{xz} \end{bmatrix}^T$$

Now, $H_e = \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \boldsymbol{vec}$, therefore:

$$H_e = \frac{3}{2q}\frac{3G}{q}\begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & 2s_{xy} & 2s_{yz} & 2s_{xz} \end{bmatrix}^T \begin{bmatrix} s_{xx} & s_{yy} & s_{zz} & s_{xy} & s_{yz} & s_{xz} \end{bmatrix}^T$$

Which after performing the dot product becomes,

$$H_e = \frac{3}{2q}\frac{3G}{q}\left(s_{xx}^2 + s_{yy}^2 + s_{zz}^2 + 2\left(s_{xy}^2 + s_{yz}^2 + s_{xz}^2\right)\right)$$

Given the definition of J_2 in Equation C.1, H_e can be further simplified to,

$$H_e = \frac{3J_2}{q}\frac{3G}{q}$$

Using $J_2 = \frac{q^2}{3}$, H_e can be simplified further.

$$H_e = \frac{3q^2}{3}\frac{3G}{q^2}$$

Finally,

$$H_e = 3G$$

As stated at the beginning of the appendix, the second purpose of this appendix is to demonstrate that MatGen is also capable of deriving the same expression for H_e, but without requiring the expert knowledge nor the necessary short cuts and short hand notation that went into the above derivation. When MatGen is given a completely expanded form of the above F and Q functions, it does indeed derive the same final expression for H_e.

Bibliography

S. Arnold and H. Tan. Symbolic derivation of potential based constitutive equations. *Computational Mechanics*, pages 237–246, 1989.

F. P. Beer and E. R. Johnston Jr. *Mechanics of Materials*. McGraw-Hill Higher Education, si metric edition, 1985.

A. Einsten. The foundation of the general theory of relativity. *Annalen der Physik*, 1916.

TEAM Engineering. Femap, 2007.

FSF. *Gnu Build System*, 2007. URL `www.gnu.org/software/autoconf/`.

H. Gao. A framework for a virtual material testing laboratory. Master's thesis, McMaster University, 2004.

GTK+. Gtk+ widget toolkit, 2007. URL `http://www.gtk.org`.

Y.M.A. Hashash, D. Wotring, J.I.-C. Yao, J.-S. Lee, and Q. Fu. Visual framework for development and use of constitutive models. international journal for numerical and analytical methods in geomechanics. 2002.

J. Ma and J. V. Nickerson. Hands-on, simulated, and remote laboratories: A comparative literature review. *ACM Comput. Surv.*, 38(3):7, 2006. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1132960.1132961.

L. E. Malvern. *Introduction to the Mechanics of Continuous Medium*. Prentice Hall, 1969.

G. E. Mase. *Schaum's Outline of Theory and Problems of Continuum Mechanics*. McGraw-Hill Publishing Company, 1970.

Yukihiro Matsumoto. Ruby programming language, 2007. URL `http://www.ruby-lang.org`.

D. R. J. Owen and E. Hinton. *Finite Elements in Plasticity: Theory and Practice*. 1980.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.

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

M. Radi. Image of spring and damper in series, 1998. URL `http://www.iue.tuwien.ac.at/phd/radi/diss.html`.

S. Robbins. Cpp unit testing framework, 2007.

Z. Roell. Image of material testing apparatus, 2007. URL `http://www.zwick.co.uk`.

W. S. Smith. *Simulating the Cast Film Process Using an Updated Lagrangian Finite Element Algorithm*. PhD thesis, McMaster University, Hamilton, ON, Canada, 2001. URL `http://www.cas.mcmaster.ca/ smiths/PhDabstract.html`.

W. S. Smith and H. 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.

D. F. E. Stolle. An interpretation of initial stress and strain methods and numerical stability. *International Journal for Numerical and Analytical Methods in Geomechanics*, 15:399–416, 1991.

R. Subramanian and I. Marsic. Vibe: virtual biology experiments. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 316–325, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-348-0. doi: http://doi.acm.org/10.1145/371920.372076.

A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notice*, 35(6):26–36, June 2000.

D. Yaron, K. L. Evans, and M. Karabinos. Virtual laboratories and scenes to support chemistry instruction. In *About Invention and Impact: Building Excellence in Undergraduate STEM (Science, Technology, Engineering, and Mathematics) Education*, 2005.

O. C. Zienkiewicz, R. L Taylor, and J. Z. Zhu. *The Finite Element Method Its Basis and Fundamentals*. Elsevier Butterworth-Heinemann, 6th edition, 2005.