Model Manipulation as Part of a Better Development Process for Scientific Computing Code

Jacques Carette, Spencer Smith, John McCutchan, Christopher Anand and Alexandre Korobkine

December 2007

SQRL Report 48

McMaster University

Contents

Abstract

We review 3 different applications of the same development methodology. We begin by using a very expressive, appropriate Domain Specific Language (in this case mathematics as embodied in a Computer Algebra System), to write down very precise problem definitions, using their most natural formulation. Once these problems are defined, this forms an implicit definition of a unique solution. From the problem statement, our model, we use mathematical transformations to make the problem simpler to solve computationally. We call this crucial step “model manipulation.” With the model rephrased in more computational terms, we can also derive various quantities directly from this model, which greatly simplify traditional numeric solutions, our eventual goal. From all this data, we then use standard code generation and code transformation techniques to generate lower-level code to perform the final numerical steps. This methodology is very flexible, generates faster code, and generates code which would have been all but impossible for a human programmer to get correct.

1 Introduction

Collectively, the authors have been developing various scientific applications for several decades. Over time, we have independently drifted towards the same development methodology. The basic ingredients involve a (declarative) domain-specific language (DSL) in which to express our model(s)[1], model transformations, code generation and program transformation. The steps involved are theoretically quite straightforward, as shown in Figure 1. Through 3 case studies, we show that the methodology is flexible, generates faster code, and generates code that would have been all but impossible for a human programmer to get correct.

1. *Express the Model* - the model is declaratively expressed in an appropriate DSL,
2. *Transform the Model* - the initial model is transformed into a form more suitable for computational solutions,
3. *Extract Structure* - structure and properties are directly extracted from the model,
4. *Optimize the Computation* - the structure is used to optimize the computational "solution" of the model,
5. *Generate the Code* - low-level code is generated to carry out the solution.

Figure 1: Typical model manipulation steps

In the case of scientific applications, the most appropriate DSL is well-known: *mathematics*. The more difficult aspect involves finding computer-based tools that can easily deal with the kinds of mathematics involved in typical scientific applications. Furthermore, not only does this language need to be "declarative," it should also be possible to manipulate this language directly. The only languages that combine the necessary semantic richness and ease of manipulation are the languages of *Computer Algebra Systems*. In our case, because it is the system we are (by far) the most familiar with, we

[1]Note that where we use "model" here, many mathematicians would use "problem" instead.

have used Maple. It is then straightforward to directly phrase the kinds of models we are most interested in: (solutions of) differential equations, and (solutions of) continuous optimization equations.

We will show that given *explicit* representations of equations whose solution we seek, the *intentional* structure of those equations can be mined to obtain a wealth of information about the structure of the solution. This, in turn, allows one to make better choices about (numerical) solution methods. We call this step "model manipulation." This is the step where human creativity and ingenuity is most needed. This is also the step where the domain expert can bring important insights. We recommend spending relatively more time on model manipulation because an investment of time here makes subsequent steps much simpler to automate.

With a model rephrased in more computational terms, we can apply various well-known techniques (like symbolic differentiation, common subexpression elimination, finding of differential or recurrence relations, etc.) to further optimize the computational structure of the model. At this point, even more classical code generation techniques can be applied to generate C calls with embedded calls to optimized numerical libraries.

In scientific computation, there are at least two circumstances in which code generation has proven to be quite effective:

1. when complex program transformations are needed (Rall, 1981; Griewank, 2000),

2. when a program can be expressed very succinctly in a domain-specific language, but requires lengthy and sometimes very complex code in a mainstream language. (Deursen et al., 2000; Deursen and Klint, 1998)

The first situation occurs most famously when *automatic differentiation* (Kahrimanian, 1953) is both required and applicable. There is now ample literature (from Thames (1969) onwards) that shows that smooth optimization problems are incomparably easier to solve when Jacobians and Hessians are available; on large problems of real interest. However, the functions to differentiate are usually given by very large programs with a multitude of inputs. Computing derivatives numerically is well-known to be a futile task, and computing them by hand (symbolically) is so fraught with error as to be deemed impossible. On the other hand, differentiation is a simple (symbolic) program.

The second situation from the above list is now emerging as rather common, which has caused the growing popularity of GUI-builders, lexer and parser generators, Java-from-DTD builders, and so on. This trend is also present in the scientific computation community, from specific efforts like Fotinatos et al. (2003), to vast projects like Dongarra and Eijkhout (2003b,a) and Kennedy et al. (2001).

The (scientific) problems that are particularly well-suited to being attacked via our approach are those which:

1. can be succinctly described using mathematics as the "domain language,"
2. require information (such as derivatives) that is easy to obtain from the model, and
3. requires experimentation and manipulation at the "model" level.

The downsides of using a DSL, as given by Deursen et al. (2000), are not relevant when using a mathematical programming language (such as Maple) as a DSL.

It is worth repeating that the most important step is that of "model manipulation." Our aim is to automate every other step to ensure that problem-solving time is spent thinking about the semantics and the structure of the problem to solve, and not wasted on mundane computational tasks.

We will present 3 applications developed using this methodology: Section 2 outlines the problem of real-time visual tracking of a target on a satellite, Section 3 shows data fitting in model-based time series obtained from Magnetic Resonance applications, and Section 4 describes material behaviour modelling. To highlight the similarities between the examples, in each case frequent reference will be made to the model manipulation steps shown in Figure 1.

2 Visual Tracking

The example in this section for visual tracking is based on Anand et al. (2004). In visual tracking applications, a series of images captured from CCD (Charge-Coupled Device) cameras must be processed in real-time to extract information about spatial positioning. This information can be used for target identification, object measurement, and closed-loop target acquisition. To compensate for harsh, dynamic lighting conditions, we consider the use of multi-colour, multi-brightness patterns, which provide quantitative information about lighting even for saturated images. The recognition of simple, explicitly designed patterns can be modeled as a constrained, nonlinear optimization problem. Recognition can be implemented as a solver, which in addition to estimating model parameters, can assign a likelihood to the estimates. Advanced model-based controllers make use of the likelihood information to improve the robustness of the controller for random and systematic noise.

Whatever target we choose, we must be able to reliably recognize its translations under affine and perspective transformations. Ideally we would like to be able to robustly identify the transformation between the identified pattern and a base family member–which would give us partial or complete information on the relative position of the target. Here we will focus on a simple family of radially-symmetric, essentially compact targets, which we will call *spots*. Transformed spots will have elliptical equiradiant contours. Given a target image, we are required to estimate parameters for the position, size, orientation and asymmetry (rotation and pitch).

The intended application of the tracking software is remote, unassisted satellite acquisition. If one wishes to capture a satellite and perform maintenance on it, a camera mounted on a robotic arm must detect and track a predefined pattern on the satellite. Significant obstacles to the algorithm include sudden changes of lighting, saturation of a significant part of the pattern, complete or partial "blindness" of the camera from the sunlight or shadow. To overcome these obstacles, we use a model-predictive controller that incorporates confidence information derived in parallel with position estimates, as well as frame-by-frame illumination estimates to be used to control camera gain. In the constrained environment of space, and the impossibility of direct human intervention, it should be possible to extract position information from tens of images per second using significantly less resources (CPU, memory, power) than on a current desktop computer. For control stability, image processing must be fast, estimated at 10-30 frames per second. For the example described here images are streamed from the camera over a FireWire serial interface at 15 frames per second.

In the first subsection below we *Express the Model* for converting the colour space and for optimal fitting of the spot parameters. After this, we *Transform the Model* to the use of Newton's method for the optimization. We then detail the opportunities to improve performance by *Extracting Structure* from the model and we describe the final step of *Generating Code*.

2.1 Model of the Colour Space and Spot Fitting

Simple colour patterns can be used to identify the orientation of the target. Having three spots of different colours, e.g. red, green and blue, the position and size of these three spots yield the position and the orientation of the target. Since differences in hues are orthogonal to brightness, there is no interference between the spots. Fitting each spot is a non-convex problem; however, the problem of locating the centre of each spot is convex.

To simplify *Expressing the Model*, the problem of fitting the spots is decomposed into two parts: conversion to a colour space in which the different spot colours are pairwise orthogonal including identification of the different colour values, followed by the extraction of spot parameters from a real-valued spot. The next two subsections detail each of these two steps. The real-valued spot can be either a gray-scale image or a single component of a multispectral image; we will not make a distinction.

2.1.1 Colour Space

Before processing, the colour values are usually linearly transformed to another colour space and converted to floating point numbers, so the effect of saturation may not be restriction to a cube in $\mathbb{R}^3$ aligned to the coordinate axes. In the target environment, saturation may be quite common. Saturation occurs when the gain adjustment is too high for the given lighting conditions. Pixels with component values of 255 may be clipped to that value from a higher value, since 255 is the maximum value for the images, which are formatted as arrays of 8-bit unsigned RGB values.

Colour conversion is achieved by taking the "real colours"

$$A = \begin{pmatrix} r_0 & r_1 & r_2 \\ g_0 & g_1 & g_2 \\ b_0 & b_1 & b_2 \end{pmatrix} \tag{1}$$

and the spot colours as α, β and γ,

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} r_0 & r_1 & r_2 \\ g_0 & g_1 & g_2 \\ b_0 & b_1 & b_2 \end{pmatrix}^{-1} \begin{pmatrix} r_{real} \\ g_{real} \\ b_{real} \end{pmatrix}, \tag{2}$$

where r_{real}, g_{real} and b_{real} are the colour values of each pixel from the original image. Now the array of α, β and γ values of each pixel is processed by the algorithm.

2.1.2 Model for Fitting Spots

Let the two-dimensional array $\phi_{x,y} \in \mathbb{R}$ represent the stored image. If the colour information is introduced, 3D $\phi_{x,y,c}$ is used, where the components x and y define a pixel p on the image and $c \in \{r, g, b\}$ defines its colour.

A model of a spot is fitted over a small region of pixels $\Omega \subset \mathbb{Z}^2$,which is believed to contain a light spot. We use a basic polynomial model of a spot (Figure 2),

$$f = k_0 + k_1 s + k_2 s^2 + k_3 s^3, \tag{3}$$

$$s = \begin{pmatrix} x - b_x \\ y - b_y \end{pmatrix}^T \begin{pmatrix} a_1 & a_2 \\ a_2 & a_3 \end{pmatrix} \begin{pmatrix} x - b_x \\ y - b_y \end{pmatrix} \tag{4}$$

where

k_0, k_1, k_2, k_3	-	determine the radial profile
a_1, a_2 and a_3	-	determine the extent and eccentricity (shape of the elliptical boundary)
b_x and b_y	-	x and y coordinates (in pixels) of the ellipse centre

In addition, we use the following constraints and conventions

$s \leq 1$	-	spot extent
$f\|_{s=1} = 0$	-	background value for spot exterior
$f\|_{s=0} = 1$	-	ideal brightness value at spot centre
a positive definite	-	so we get a spot

We use these constraints to eliminate the parameters $k_0 = 1$ and $k_3 = -(k_0 + k_1 + k_2)$. Variations of spot and background illumination are represented in the complete model,

$$v_1 f(p) + v_0, \tag{5}$$

by v_0 and v_1.

Using the least squares method, the best fit of this model to the actual light intensity function ϕ of the chosen area can be found.

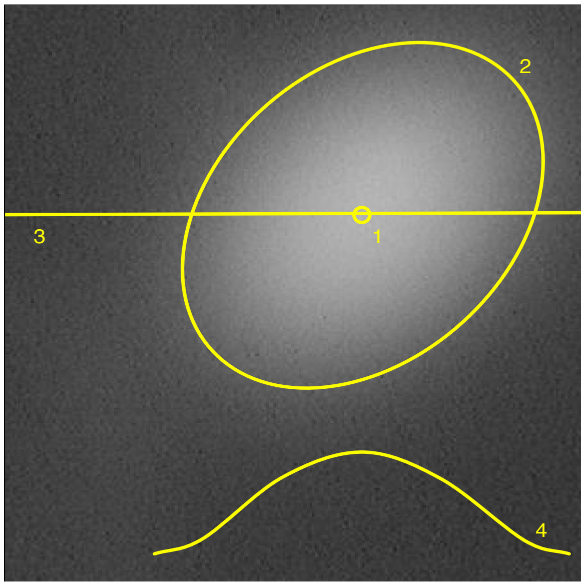

Figure 2: Actual image of a gray-scale target,showing the spot's centre (1), a level set representative of the shape given by a (2), and, at a cross-section (3), the profile (4) determined by k. The background illumination is determined by v_0 and the brightness of the centre of the spot is v_1.

$$\min_{\mathcal{U}} F = \sum_{p \in \Omega} (\phi_p - (v_1 f(p) + v_0))^2 \tag{6}$$

where

$$\Omega = \{(x, y)|s \leq 1\}. \tag{7}$$

We minimize F over $\mathcal{U}$, where $\mathcal{U} \subseteq \mathcal{V} = \{v_0, v_1, k_1, k_2, b_x, b_y, a_1, a_2, a_3\}$. Equation 6 shows the first step in the model manipulation process of *Expressing the Model*. This equation actually represents a family of models, distinguished by the particular value of $\mathcal{U}$. The DSL (Maple code) for this problem specifies the required value of $\mathcal{U}$ for a specific solver. The reason for the family of solvers is that a naive implementation, where we simultaneously optimize all variables, will fail, except possibly when we are tracking an essentially stationary spot from frame to frame, uninterrupted by lighting changes. The reason for this failure is that the target recognition problem is not convex. However, we empirically determined that the problem for finding the *center* (b_x, b_y) of a single spot is convex, as is the problem of finding the *shape* (a_1, a_2, a_3). However, the joint problem is not convex (the Hessian has two negative eigenvalues with magnitude 1×10^5).

Due to lack of convexity when solving for the position of the centre and the shape of the spot in a single stage, it is necessary to have multiple solver stages. Assuming that the size of the spot is larger than the whole captured image, during the first stage the location of spot is found (b_x and b_y). Having an exact position of the centre of the spot makes the fitting of a_1, a_2 and a_3 a convex problem. So in a second stage, in just a few iterations, the shape of the spot can be found.

2.1.3 Saturation

Before fitting of the parameters defining the location and the shape of the spot can be performed, detection of saturation of the captured image must be carried out. If the image is found to be saturated, saturated pixels are removed when F is computed. Furthermore, if $\mathcal{S}$ is the set of saturated pixels, set Ω now becomes:

$$\Omega_{without\ saturation} = \Omega - \{(x, y)|(x, y) \in \mathcal{S}\}. \tag{8}$$

To perform the computation of F excluding saturated pixels, which contain little useful information about illumination, we must conditionalize the processing pixel by pixel. This will increase execution time of the algorithm. Since saturation of the image will not always occur, for each set of parameters, two versions of the solvers will be produced–one with and one without saturation control. The likelihood of saturation is calculated each frame based on the estimate for v, and this information is used to decide which solver to use for the next frame.

2.2 Transformed Model (Newton's Method)

The *Transformed Model* for finding the minimum in Equation 6 consists of searching for a common zero of all the partial derivatives with respect to all the parameters of $\mathcal{U}$, using Newton's method. Let $\mathbf{J}_{\mathcal{U}}$ be the Jacobian of F and $\mathbf{H}_{\mathcal{U}}$ be the Hessian of F with respect to the variables $\mathcal{U}$. $\mathbf{u}_n$ is a solution for the nth step (vector-column of the $\mathcal{U}$ parameters). The Newton iteration is defined by the recursion:

$$\mathbf{u}_{n+1} = \mathbf{u}_n - \mathbf{H}_{\mathcal{U}}(\mathbf{u}_n)^{-1}\mathbf{J}_{\mathcal{U}}(\mathbf{u}_n) \qquad (9)$$

Newton's method is used for the fitting, which means the initial starting point is very important. We use heuristics to determine reasonable initial guesses. From the image we can extract initial guesses for the average radius of the light spot on the image, the values for v_0 and v_1, and the position of the centre.

The profile of every spot should be the same, thus the same values of k are used for all the spots. v_0 describes a background. b_x and b_y can be chosen as a centre of the image fragment. We have found it sufficient to use as a first guess for the parameters a_1, a_2 and a_3, which determine the shape of the spot is a circle, values which ensure that the initial spot covers the entire image fragment.

2.3 Extracting Structure and Generating Code

To improve the performance of the algorithm, we can *Extract Structure* from the transformed model. From the model, we can easily compute the Jacobian and Hessian matrices. The manipulations that optimize the computation of these matrices provide examples of the kind of model manipulation that so often arises in scientific computing. For instance, we know that large arrays are needed to store the information for the captured images and that calculation of the sum over elements in arrays is an expensive procedure. This

	jointly optimized Jacobian and Hessian		separately optimized Jacobian and Hessian	
	+tryhard		+tryhard	
b	78	112	97	152
a	88	135	117	176
a,b	205	325	220	396
a,b,v	230	394	284	461

Table 1: Number of flops per pixel in generated solvers

implies that efficient use of the cache is required to minimize the execution time, which will be bounded by memory accesses. The easiest way to ensure this is to group all accesses to one pixel of data (within a solver iteration) together. From their definitions we know that Jacobian and Hessian matrices will contain many common subexpressions, therefore optimization on "the inner sum" is crucial. We also know that since Hessian matrices are symmetric, we only need to calculate the upper triangular portion of them. Using this information suggests that the for the Jacobian and the Hessian we should jointly *Generate the Code*. Measurements of floating point operations in code generated using these optimization strategies confirms our expectations (see Table 1).

There is an advantage to the joint optimization of the Jacobian and the Hessian matrices, which would not be feasible without *Generating the Code*. Maple's `codegen[optimize]` function eliminates common subexpressions very effectively when these matrices are generated together. If the optimization of code is performed separately for Jacobian and Hessian matrices and the results are concatenated (which is closer to the code that would be obtained without using the model manipulation process), both the length of the solver and the number of flops per pixel are roughly doubled. This does not reflect the equally important reduction in memory traffic and reduction in local variables by jointly calculating the Jacobian and Hessian in one loop. The improvements in the performance from joint optimization and calculation of the Jacobian and Hessian corresponds to the fourth step in the model manipulation process: *Optimizing the Computation*.

The complexity of the Maple-generated Newton solvers is found to be directly proportional to the complexity of the model. The length of the typical Maple-generated Newton solver (for 2 or 3 variables) is approximately

180 lines of code and 200 flops per pixel. Solvers are generated by executing `codegen[optimize]` function with the `tryhard` option. Without the `tryhard` option, generated solvers slightly increase in length and number of floating point operations per pixel (230 lines of code, 250 flops).

Besides potentially improving the performance of the Newton solvers, the approach of generating members of a family of Newton solvers (indexed by the power set of the set of model parameters), allows for the development of a staged algorithm that would be very difficult to derive another way. To find a multi-stage non-linear solver with good convergence properties involved using heuristics and benchmarking. Without the ability to easily generate the solvers, the experimentation necessary to produce this overall algorithm would have been very difficult.

The details of the implementation in Maple can be obtained from the original paper (Anand et al., 2004). As a brief summary, fitting of the spots at run time is performed as follows. An image is read pixel by pixel into a 2D array. The Maple-generated functions that calculates Jacobian and Hessian matrices for given optimized variables are used with the Newton's method from LAPACK. Two functions in particular, `sgetrf_` for factoring the $\mathbf{H}$ matrix and `sgetrs_` to solve the $\mathbf{Hx} = -\mathbf{J}$ system. A better approximation to the initial guess is calculated and is used in Maple-generated C function to obtain yet "better" values for the optimized variables and the procedure is repeated. Depending on the timing requirements of the application, either the difference between two consecutive approximations of Newton's method or the number of iterations can be used as termination criteria for the algorithm. This section has outlined a specific optimization algorithm. The next section will present a generalization of this algorithm.

3 Parameter Estimation in Model-Based Time Series

This example of parameter estimation from time-series data is extracted from Anand et al. (2005). Parameter estimation is important in many problem domains including determination of rate constants in pharmaceutical drug transport, decomposing audio signals and voice recognition, and measurement of metabolite levels in Magnetic Resonance Spectroscopy and Relaxometry.

Here we describe a model-based method of decomposing signals. Our model class will have a high degree of structure, namely:

1. the objective will be a linear superposition of the simple functions,
2. these functions will satisfy a linear recurrence relation over time,
3. these functions will satisfy a (system) of linear ordinary differential equations in the parameters.

Note that in practice this is actually more general than the currently used model classes. Bachmann et al. (1994) provides details of how recurrences form a very general class. In fact, since models are commonly composed of combinations of closed-form analytic functions, the assumption that they satisfy a linear ordinary differential equation (ODE) in the parameters is more natural than restrictive.

There are a multitude of physical processes involving decay equations that can be efficiently computed using recurrence relations, and some of them generate quantities of data that pose a computational challenge, even with modern processors. Among these are Magnetic Resonance Spectroscopy (MRS) and Magnetic Resonance Relaxometry. The first is used to identify the chemical composition within living tissue, and can be used both in clinical diagnosis and biomedical research. Relaxometry measures tissue properties that depend on changes in pH and temperature, and has been proposed as a method of diagnosis and treatment monitoring, in particular, non-invasive, real-time temperature monitoring.

The presentation of the model manipulation for the parameter estimation problem begins with *Expressing the Model*, which in this case is a more general version of the model presented in the previous section. The *Transformed Model*, which is an optimization via Newton's method, is also similar to the previous example. After describing the models, the *Extract Structure* step is performed. The structure shows recurrence relations, which can be used to *Optimize the Computation* in the *Generate Code* step. The end of this subsection includes two examples that highlight the advantages of the model manipulation approach.

3.1 Expressing the Mathematical Model

A common method of parameter estimation for time series data involves modelling signal sources, $f(x_1, x_2, \ldots, x_n, t)$, (where the x_i are the model parameters and f is in general a vector-valued function) and fitting a superposition of the various sources to the measured data. Through minimization of an objective function F an optimal set of parameters may be determined:

$$\min_{x_1^1,x_2^1,\ldots,x_n^1,\ldots,x_n^s} \sum_t \left\| y(t) - \sum_{s\in\{\text{sources}\}} a_s f_s(x_1^s, x_2^s, \ldots, x_n^s, t) \right\|^2 . \tag{10}$$

where x_j^s denotes the x_j'th parameter of peak s. Equation 10 *Expresses the Model* for parameter estimation of a time series. The Maple code DSL for this model should explicitly declare the class of functions f, which parameters to optimize for, and how many superpositions of the basis function should be used for fitting.

As our objective functions will all be analytic, we can safely use Newton's method to solve the minimization problem; therefore, Newton's method once again forms the *Transformed Model*, as it did for the last example. The structure of the Newton solver, and the issues that arise, are the same as for the previous application (see subsection 2.2).

3.1.1 Recurrence Relations

The goal of *Extracting Structure* from the model shows the frequent occurrence of recurrence relations, because in many time-series models, a simple time evolution exists. This allows the use of recurrence relations instead of explicit calculations of the model function. This greatly increases the efficiency of objective function evaluations, as well as the calculation of the Jacobian and Hessian on each solver iteration. For instance, in the case of an exponentially damped oscillatory signal, $ae^{-(d+if)t}$ of frequency f, amplitude a, and damping coefficient d, the sequence $a, ae^{-(d+if)}, ae^{-2(d+if)}, \ldots$ can be calculated using the recursion

$$z_0 = a, \quad z_{i+1} = kz_i, \quad k = e^{-(d+if)}. \tag{11}$$

This requires two real variables for each constant k, two variables for the most recent value z_i, and one temporary variable to do the multiplication. If $d > 0$, the sequence converges to 0 geometrically in norm. In this case, the numerical errors do not accumulate appreciably, in the absolute sense, although they will be a relative accumulation of errors. Since errors in the measurements are commonly assumed to be uniformly and independently distributed, we are usually only concerned with absolute errors. When f is vector-valued, the same ideas usually work component-wise.

3.1.2 Differential Equations

If the model happens to have a simple dependence on the parameters, then it is usually the case that the derivatives that appear in the Jacobian and Hessian of Equation 10 are simply expressible in terms of the model itself. Here we only illustrate what happens for a first-order dependence, which can be used to simplify both the Jacobian and the Hessian. If there is a second-order dependence, then that can be used to simplify the Hessian.

Considering a simple model with first-order dependence on a parameter b,

$$f(b) = ae^{bp(x)} \tag{12}$$

then the derivative can be re-expressed in terms of f as follows.

$$\frac{\partial f}{\partial b} = p(x)ae^{bp(x)} = p(x)f(b) \tag{13}$$

The above can be deduced by constructing the ODE that $f(b)$ satisfies, namely

$$\frac{\partial f}{\partial b} - p(x)f(b) = 0. \tag{14}$$

If the dependence is algebraic - which can be considered to be a zeroth order differential equation - this can also be used for simplifications. As such dependencies are sources of redundant computations in the resulting code, it is important to factor them out.

3.2 Model Manipulation

Abstractly, what we really want to do is to be able to solve any parameter-fitting problem, such as Equation 10, by describing our class of functions f, which parameters to optimize for, and how many superpositions of the basis function should be used for fitting. We can then symbolically obtain

1. The recurrence equation satisfied by the model f with respect to the main variable t.

2. The differential equation(s) satisfied by the model f with respect to the parameters x_i.

3. The Jacobian of the fitting Equation 10 with respect to all the parameters x_i.

4. The (upper triangle of the) Hessian of the fitting Equation 10 with respect to all of the parameters x_i.

Furthermore, *Extracting the Structure* using the recurrence and the differential equations, we obtain a simplification of the Jacobians and the Hessians with respect to the structure of the model f and the linearity of Equation 10.

The first step above is obtained via the `IsHypergeometricTerm` function from the `RationalNormalForms` Maple package. This function uses some advanced symbolic techniques (Abramov et al., 2004) to decide if a given term $f(t)$ is such that $\frac{f(t+1)}{f(t)}$ is a rational function of t, and returns this rational function if this is the case. We can thus handle any model family f that is a *Hypergeometric term* in t. This includes functions such as $\Gamma(a*t+b)$ (and thus factorial), the pochhammer symbol $(a)_t$ (or rising factorial $a^{\overline{t}}$), the falling factorial $a^{\underline{t}}$, as well as polynomials, rational functions, and linear exponentials e^{at+b}, as well as finite products and ratios of any of the aforementioned. Our method easily generalizes to higher order recurrences, but we are not aware of a simple way to obtain these recurrences using the current version of Maple.

The second step is obtained via `gfun[holexprtodiffeq]`. The abbreviations stand respectively for *generating function* and *holonomic expression to differential equation*. The package `gfun` is described in Salvy and Zimmermann (1994), while the theory of holonomic (or D-finite) functions is described in Chyzak and Salvy (1998). For the time being, we can only take advantage of either zero-th and first-order differential dependence on parameters. For example, for arbitrary functions f, g and h, we can handle models that look like $g(a) \times h(t) + f(b)$ as well as $e^{h(a)t+g(b)}$ for parameters a, b.

3.3 Code Generation

The *Generate Code* step involves first turning the mathematics of the previous sections into (pseudo) code. We are looking to generate something like the following:

```
procedure GeneratedCode(y, n)
integer n, t
real f_1, ..., f_k, h_1, ..., h_k, F
real array y, Jacobian, Hessian
```

```
begin
  f_1 := f_1(0);
  h_1 := recurrence ratio of f_1;
  ...
  f_k := f_k(0);
  h_k := recurrence ratio of f_k;
  F := 0;
  Jacobian := 0;
  Hessian := 0;
  for t := 0 to (n-1) begin
    F := F + (y[t] - sum(f_i, 1 to k))^2;
    Jacobian := Jacobian + Jacobian at t;
    Jacobian := Hessian + Hessian at t;
    f_1 := h_1 * f_1;
    ...
    f_k := h_k * f_k;
  end;
return F, Jacobian, Hessian;
end;
```

In other words, we need to generate a procedure that computes F, its Jacobian and Hessian, taking full advantage of the fact that F is a sum, and that all of its sub-terms satisfy a recurrence. Using this structure allows us to *Optimize the Computation.*

The generation algorithm can be explained as

1. get recurrence relation for f on t (via `IsHypergeometricTerm`),

2. construct the Jacobian and the Hessian for the model function f in terms of f,

3. if f is a complex function, split the above into real and imaginary parts,

4. generate code to calculate the initial value of f, the recurrence ratio h, as well as code to calculate successive terms using h and the last calculated term; do this for each superposition of f;

5. generate code to calculate, by summing in a loop, F, Jacobian(F), Hessian(F); use previously computed relations on derivatives of f (from step (2)), as well as re-using the recurrence for f;

6. the above code uses local variables (in the generated code) to store the Jacobian and Hessian, to enable common-subexpression elimination (as it cannot be done on Matrix/Vector entries).

7. generate "cleanup" code to assign locally stored Jacob(F) and Hess(F) to arrays that are "returned"

8. wrap F, Jacob(F), Hess(F) and recurrence code in a loop on t and apply sub-expression elimination optimization

9. "paste" code together and transform to C code

The user inputs the model function f, the main variable t, the parameters to optimize α, and the number of superpositions k to fit. In the above f is represented abstractly in the intermediate steps of generating the code. This is useful because we know that the information derived from f is correct generically, and the details of f would actually hinder rather than help these computations. All parameters of f are indexed by the superposition to which they belong.

The Jacobian of f with respect to the parameters α is computed symbolically, using the previously computed differential relations. If the differential equation technique fails for any $a \in \alpha$, that partial derivative is computed by direct symbolic differentiation. Direct symbolic differentiation is then used on the Jacobian to get the Hessian. Any occurrence of $\frac{\partial f}{\partial a}$ in the Hessian is replaced by the appropriate Jacobian entry. Since the Jacobian is given in terms of f the Hessian will be as well.

If f is a complex (vector) function, f, h (the recurrence multiplier), and the Jacobian and Hessian of f are separated into real and imaginary parts at this point. We must eventually convert all our computations to real computations only, and this point in the algorithm is where we gain the most benefit: previous computations are simpler on the complex function, while more common sub-expressions can be pulled out from the expanded version.

For each superposition of the model function, code is generated to calculate the initial term of the recurrence of f, the recurrence ratio h, and successive terms. If f and y are assumed to take values in $\mathbb{R}^m$, α_p is a short-hand for the parameter vector, the (2-norm) error function F and its Jacobian are given respectively by

$$\sum_{t=0}^{n-1} \left\| y(t) - \sum_{p=1}^{k} f(t, \alpha_p) \right\|^2 = \sum_{t=0}^{n-1} \sum_{s=0}^{m-1} \left(y_s(t) - \sum_{p=1}^{k} f_s(t, \alpha_p) \right)^2 \qquad (15)$$

$$J_{\alpha_q^i} = 2\sum_{t=0}^{n-1}\sum_{s=0}^{m-1}\left(\sum_{p=1}^{k} f_s(t,\alpha_p) - y_s(t)\right)\frac{\partial f_s(t,\alpha_q)}{\partial \alpha_q^i} \tag{16}$$

Formulas for the Hessian can be similarly derived. It is very important to note that the formulas for the Jacobian above are completely uniform in the parameter i. This means that instead of computing each component as a sum over very similar entries, it is more efficient to compute the Jacobian as a sum of vectors, as this allows significantly more common computations to be extracted. For every occurrence of $\frac{\partial f}{\partial a}$ in Jacob(F) and Hess(F), the corresponding entries of the pre-computed Jacob(f) and Hess(f) are substituted in.

The code to calculate F, J and H is combined with the code to calculate successive terms of f. This then makes up the body of a loop on the main variable t. Common sub-expression elimination is used on the loop body via `codegen[optimize]` with the `tryhard` option, and the optimized code is wrapped in a loop on t from 0 to $n-1$, where n (number of data points) is an argument of the generated function. The loop is then spliced with the previous code, transformed into a C function with the following signature:

```
(double *y, int n, double *gama, double *J, double *H)
```

and the return value F.

Using model manipulation we have measured a 120-fold reduction in execution time for real valued exponential models when compared to a "vanilla" implementation, and a 540-fold reduction for complex valued exponential models. Although we would not expect this to be the case for all applications, we certainly expect significant gains for many applications.

3.4 Applications

We give two examples from Magnetic Resonance. The first example provided the impetus for this work, and in addition to developing the model functions, we sketch the arguments for using a series of subspace searches to try to find the global minimum for a highly non-convex function. The second example shows another use of real exponentials.

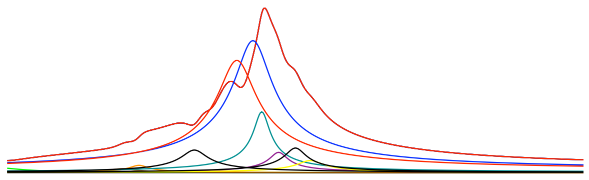

Figure 3: Soya bean oil phantom 1H-MR spectrum (maroon) and component estimates.

3.4.1 Magnetic Resonance Spectroscopy (MRS)

The signal in conventional MRS is attained from hydrogen atoms bound to the molecules of interest. The different bonding patterns found in these molecules slightly alter the base resonance frequency of the hydrogens and lead to an effect referred to as chemical shift. Other factors such as bulk magnetic susceptibility and tissue orientation (with respect to the magnetic field) also contribute to chemical shift, but generally to a lesser extent. Different chemicals can therefore be identified by their frequency shift relative to some reference (usually water). To quantify the amounts of chemical present, we model the signal generation and find the maximum likelihood estimate for the model parameters. Figure 3 shows the resulting decomposition of a measured spectrum.

Unfortunately this objective function is not convex. By approximating the problem by one which is convex, and then successively introducing greater complexity in successive approximations, and switching between the frequency and time domains, we are able to stay within the basin of convergence of Newton's method. We can ignore the signal phase while still fitting the peak positions in the frequency domain, and then switch to the time domain, increasing the accuracy of the determined model parameters. Using prior knowledge of fixed peak resonance frequency relationships of the compounds we expect to find in the samples, we both increase the accuracy of our estimates, and reduce the complexity and memory footprint of the solver. We find that this method fits spectra very quickly and provides good results.

The time-domain signal that is measured is the free induction decay (FID), which is the superposition of signals from several different tissues, each having the form

$$ae^{-(d+if)t} \tag{17}$$

where $a \in \mathbb{C}$ is the signal strength and phase, d is the damping and f is the frequency (peak position). The Fourier Transform of one signal is a Lorentzian:

$$\frac{1}{d + i(f-x)} = \frac{d}{d^2 + (f-x)^2} + i\frac{x-f}{d^2+(f-x)^2}.$$

A delay in sampling will create a complex phase ramp e^{ikt} where k is related to the delay. Since spectroscopy pulse sequences are complicated, including water suppression, the phases of the different resonances are unlikely to be exactly the same.

The squared norm of the spectrum is not affected by delays in sampling, nor by the overall phase of the signal. If peaks are well-separated, or have similar phase, the spectrum is not affected by the phase differences between different peaks either. Under these assumptions, we can fit the peaks without worrying about phase. This reduces the dimensionality of the problem and eliminates a large source of non-convexity caused by the ambiguity in phase angles.

Fitting a single peak is equivalent to minimizing the convex objective:

$$\min_{a,d,f} \sum_x \left(\mathcal{F}y(x) - \frac{a\bar{a}}{d^2 + (f-x)^2} \right)^2.$$

Once we add a series of peaks, we must take the norm of the sum of the real and imaginary parts, and not the sum of the norms. This makes the model function more complicated, and again breaks the convexity of the objective function.

To do unattended peak fitting we approximate the problem with a (simplified) convex problem; its solution can then be used as a good starting point for Newton's method to converge to the global minimum of the non-convex objective function. This is achieved by smoothing out the peaks until the spectrum itself becomes convex. We add in a damping factor k and multiply the time-domain data by e^{-kt}, and then try to fit it with a sum of Lorentzians with d replaced by kd. As the Newton method converges, the damping is iteratively decreased until the original problem is recovered.

Time-domain fitting,

$$\min_{a_0,d_0,f_0,\ldots,d_k,f_k} \sum_t \left\| y(t) - \sum_{s\in\{\text{sources}\}} a_p e^{-(d_p + i f_p)t} \right\|^2, \tag{18}$$

(where $y(t)$ is the complex sample at time t), has the advantage that there is no Gibbs ringing, it is easier to fit asymmetric echos, and the model functions can be calculated using only additions and multiplications via the recurrence relations given in Equation (11).

In the frequency domain, the magnitude of the peaks is the easiest and most intuitive to fit. In the model we use for our specific application, there are theoretically 12 distinct signals that compose the FID. Some of these peaks will always have fixed frequency offsets from one another, and generally all 12 peaks should appear in more or less fixed locations. This allows us to greatly simplify the process of finding an initial guess by first fitting the location of the set of fixed peaks. In a second step, we can individually fit the frequencies in the time domain. Fitting the complex data in the time domain is most useful for determining the peak area, as the complex area values carry the signal phase information with them. This becomes a linear problem with a quadratic objective function, yielding a best fit solution in only one iteration of the solver. Performing this operation in the frequency domain would require guessing the phase angles. In addition to fitting the area, attaining accurate values for damping is also simplified in the time domain. In our method, the complex data is iteratively fit for area, phase and damping, until the change in residual area after each iteration is very small.

3.4.2 Magnetic Resonance Relaxometry

A real-valued example problem can be drawn from MR Relaxometry, where the purpose of the experiment is not to acquire an image or FID spectrum as in MRI or MRS, but to determine the time constants of the signal decay, which vary due to chemical environment, including pH levels and temperature, making relaxometry a useful non-invasive tool for determination of these quantities *in-vivo* in real time. Changes in time constants are also useful as indicators of disease not apparent when examining MR images, such as susceptibility for seizure (Kanner, 2004).

The time constants arise from the Bloch equations (Bloch, 1946), which govern the response of a proton in a magnetic field. The decay constant $T1$ relates to the longitudinal relaxation (spin-lattice interactions) of the protons, and manifests itself in the re-growth of the proton magnetization in the direction of the main field. The decay constant $T2$, or transverse relaxation is a measure of the rate of signal de-phasing. Depending on the set-up of the MR Relaxometry experiment, one can measure either constant.

The typical MR Relaxometry experiment involves exciting the sample, waiting, and then measuring the magnitude of the MR signal. This is repeated with a small number (typically 6-10) of different measurement delays, which have the form of a real-valued, damped exponential. When information is required about the decay constants of multiple species present in one sample, the problem becomes very similar to that of spectroscopy, where the objective function has the form:

$$\min_{a_0,d_0,\ldots,a_k,d_k} \sum_t \left\| y(t) - \sum_{s\in\{\text{sources}\}} a_p e^{-d_p t} \right\|^2 . \tag{19}$$

3.5 Results

Performance testing of the generated code for calculation of the Hessian and Jacobian was performed for both a real and a complex model function (Equations (19) and (18)) with 12 signal sources. The execution time was measured as the average of 1000 iterations of the solver on a 1.33 GHz PowerPC G4 processor, over 1024 pseudo-random sample data points. All C code was compiled with GCC 3.3, with full optimizations (-O3) enabled.

The benchmarks were run with different combinations of Maple-based optimizations enabled: no Maple-level optimizations, incorporation of the recurrence relation (R), incorporation of the symbolic differential equations (D), and both (R + D). In addition, different common sub-expression elimination routines in Maple were tested, as detailed in the tables.

Tables 2 and 3 present the data for the complex-valued objective, but with derivatives that are taken with respect to the real d_i terms in the exponential. The data represents the run time and relative speed improvements (with respect to a non-optimized version) of the generated code. Very similar results are obtained in the other scenarios.

In all cases, a large benefit is to be gained by exploiting the problem structure. Not surprisingly, Maple's common sub-expression elimination by itself yields a large improvement in execution time, as the 'base' code contains sums of constant exponential terms (as well as sines and cosines in the complex case). While the recurrence relation and differential equation optimizations alone lend a small speed increase, the maximum speedup is gained when both are put to work. This jump in performance is because, when working together, all exponential and trigonometric function calls are eliminated from the loop over the data points, leaving only addition and multiplication operations.

Table 2: Time for computation complex model function (18)

	Time per iteration (seconds).			
	Base	Base + R	Base + D	Base + R + D
No optim.	1.1237	0.4037	0.5712	0.0037
[optimize]	0.0231	0.0196	0.0198	0.0021
'tryhard'	0.0227	0.0197	0.0191	0.0021

Table 3: Average speedup for complex model function (18)

	Relative speedup due to optimizations.			
	Base	Base + R	Base + D	Base + R + D
No optim.	1	3	2	301
[optimize]	49	57	57	541
'tryhard'	50	57	59	546

4 Material Behaviour Modelling

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.

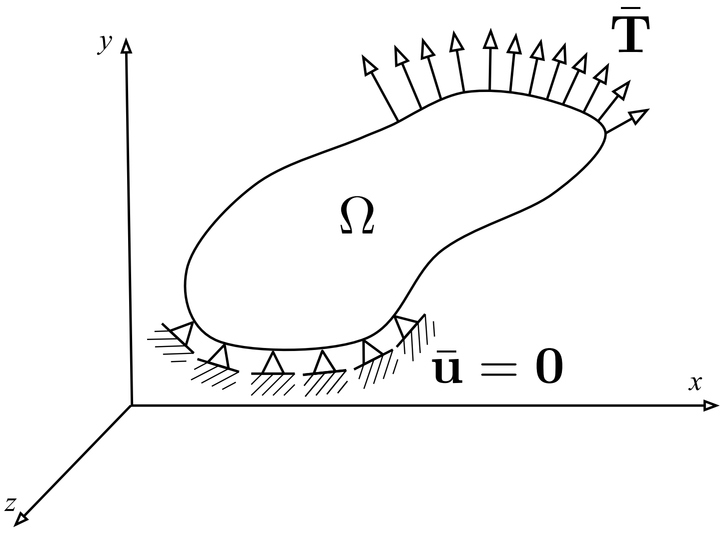

Figure 4: A typical boundary valued problem with prescribed displacements and an applied surface traction

In continuum and computational mechanics the goal is often to solve for the changes of deformation and stress within a given body Ω, as depicted in Cartesian (x-y-z) space in Figure 4. To solve for the histories of deformation and stress it is necessary to satisfy the governing partial differential equation for equilibrium, subject to initial conditions and boundary conditions. The initial conditions could involve setting initial stress or strain fields, while the boundary conditions could involve specifying the surface traction $\bar{\mathbf{T}}$ or the prescribed displacements $\bar{\mathbf{u}}$. The equilibrium equation alone does not provide enough information (equations) to solve for all of the unknowns, so the constitutive equation is added. The constitutive equation postulates a dependence of the stress on the history of deformation, which allows for a solution to the overall problem.

Constitutive equations can potentially be complex. The relationship between stress and deformation can rely on multiple non-linear equations, which can conceivably involve the entire history of deformation. A wide range of constitutive equations are used in engineering applications. For instance, materials may be modelled as elastic, viscous, viscoelastic, plastic or viscoplastic, or they may be modelled as a combination of these behaviours. New constitutive equations are currently being developed to better describe existing engineering materials, like soil, concrete and steel, and to describe newly developed materials, such as new plastics and composite materials. Although the behaviour of these different types of materials can exhibit great variation, the mathematics used to describe them is often very similar. Using the correct abstraction it is possible to consider a wide range of material behaviours within one family of material models. Using the model manipulation techniques described below it is possible to quickly generate code for a specific member of this family.

We first *Express the Model* by presenting the equations shared by all of the family members and by giving a sample of the DSL used to describe a specific material model. After this, we show how to *Transform the Model* into a finite element algorithm. This transformed model is written in a generic form so that it applies to all members of the family of material models. The next step described is how to *Extract the Structure* (generic parts) from the algorithm so that they may be replaced by a specific material described via the DSL. Finally, details are given for the *Generate Code* step, which uses the DSL to fix the generic parts in the algorithm. The notation used in the sections below 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 constitutive equations are written in matrix form.

4.1 The Mathematical Model Relating Stress and Deformation

The goal in modelling material behaviour is to determine the internal stress within a material particle given the deformation history of that particle. The state of stress $\boldsymbol{\sigma}$ at a point is given by components of force per unit area acting on the faces of an infinitesimal cube centred at the point, as shown in Figure 5. To maintain equilibrium of the cube, several of the stress components are equal, with the result that there are six independent components of the stress tensor, which can be summarized using vector notation (Zienkiewicz et al., 2005):

$$\boldsymbol{\sigma} = \left[\begin{array}{cccccc} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{yz} & \sigma_{xz} \end{array} \right]^T \qquad (20)$$

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.

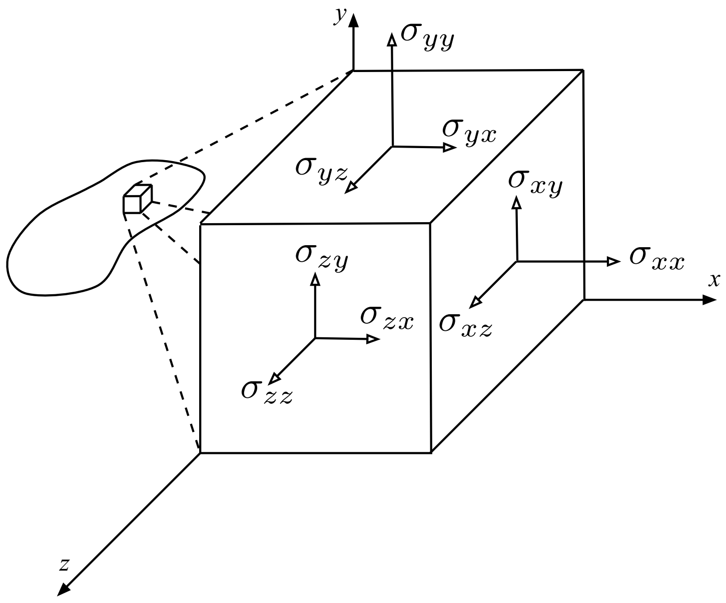

Figure 5: Stress tensor for a point within a body

As stated previously, the stress within a particle depends on the deformation history of that particle. Although there are many measures of deformation, the one that will be adopted here is the rate of natural strain tensor $\dot{\boldsymbol{\epsilon}}$, which, like the stress tensor, can be summarized by six corresponding independent components:

$$\dot{\boldsymbol{\epsilon}} = \left[\begin{array}{cccccc} \dot{\epsilon}_{xx} & \dot{\epsilon}_{yy} & \dot{\epsilon}_{zz} & \dot{\gamma}_{xy} & \dot{\gamma}_{yz} & \dot{\gamma}_{xz} \end{array} \right]^T \tag{21}$$

When simulating material behaviour several input values must be provided at run-time, including the initial stress ($\boldsymbol{\sigma}_0 : \mathbb{R}^6$), the time at which the simulation starts ($t_{beg} : \mathbb{R}$) and the time at which the simulation ends ($t_{end} : \mathbb{R}$). The deformation history must also be provided by giving the rate of natural strain as a function of time ($\dot{\boldsymbol{\epsilon}}(t) : \{t : \mathbb{R} | t_{beg} \leq t \leq t_{end}\} \rightarrow \mathbb{R}^6$). The final run time values that must be specified for a simulation are the material properties, which for the family of models under consideration will always include the elastic modulus ($E : \{x : \mathbb{R} | x \geq 0\}$) and Poisson's ratio ($\nu : \{x : \mathbb{R} | 0 < x \leq 0.5\}$). Additional material properties will depend on the specific material. Each of these properties will be named, and to each name will correspond a value ($\in \mathbb{R}$); these values will be assembled in a *property vector*.

In an actual simulation the rate of natural strain ($\dot{\boldsymbol{\epsilon}}(t)$) will rarely be specified explicitly, rather it will be implicitly specified through the equilibrium equation and the given boundary conditions and initial conditions on the material body. At every instant in time the body must satisfy the equilibrium equation. If inertia, self-weight and other body forces are neglected, then the equilibrium equation can be written as

$$\mathbf{L}^T \boldsymbol{\sigma} = \mathbf{0} \tag{22}$$

where $\mathbf{L}^T$ is the following differential operator:

$$\mathbf{L}^T = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \tag{23}$$

The equilibrium condition (Equation 22) alone is not enough to solve for the stress as a function of time ($\boldsymbol{\sigma}(t) : \{t : \mathbb{R} | t_{beg} \leq t \leq t_{end}\} \rightarrow \mathbb{R}^6$, , where t_{beg} and t_{end} delimit the duration of the simulation.) A constitutive equation relating stress to the history of deformation must also be added. A wide range of material behaviours can be represented by the following constitutive equation written in rate form:

$$\dot{\boldsymbol{\sigma}} = \mathbf{D}\left(\dot{\boldsymbol{\epsilon}} - \gamma < \phi(F(\boldsymbol{\sigma}, \kappa)) > \frac{\partial Q(\boldsymbol{\sigma})}{\partial \boldsymbol{\sigma}}\right) \text{and } \boldsymbol{\sigma}(t_{beg}) = \boldsymbol{\sigma}_0 \tag{24}$$

where

$$< \phi(F) >= \begin{cases} \phi(F) & \text{if} \quad F > 0 \\ 0 & \text{if} \quad F \leq 0 \end{cases} \tag{25}$$

The above equation is based on the viscoplastic constitutive equation presented by Perzyna (1966). The governing differential equation depends on the elastic constitutive matrix ($\mathbf{D} : \mathbb{R}^{6\times6}$), the fluidity parameter ($\gamma : \mathbb{R}$), the function ϕ ($\phi : \mathbb{R} \rightarrow \mathbb{R}$), the yield function ($F(\boldsymbol{\sigma}, \kappa) : \mathbb{R}^6 \times \mathbb{R} \rightarrow \mathbb{R}$), the plastic potential function ($Q(\boldsymbol{\sigma}) : \mathbb{R}^6 \rightarrow \mathbb{R}$), the stress tensor ($\boldsymbol{\sigma} : \mathbb{R}^6$), the strain rate tensor ($\dot{\boldsymbol{\epsilon}} : \mathbb{R}^6$) and the hardening parameter ($\kappa : \mathbb{R}^6 \rightarrow \mathbb{R}$), which measures the accumulated strain. In the constitutive equation the condition $F = 0$ defines a surface in 6 dimensional stress space, which can be visualized by considering the sketch shown in Figure 6. Inside the surface ($F < 0$) the

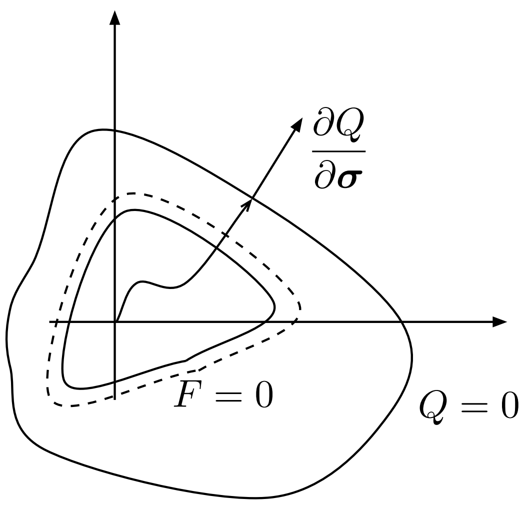

Figure 6: Yield Function, Hardening and the Plastic Potential in Stress Space

material response will be purely elastic, outside the surface the response is viscoplastic. When the material has yielded, which occurs when the stress path reaches the yield surface, as shown in Figure 6, the yield surface may change shape. This change in shape is caused by the strain hardening (or softening) of the material. The new yield surface is shown in Figure 6 as a dashed line. This behaviour is mathematically represented in the yield function by its dependence on the instantaneous values of the hardening parameter κ. The parameter κ depends on the accumulated viscoplastic strain ($\kappa = \kappa(\boldsymbol{\epsilon}^{vp})$). The plastic potential Q (Figure 6) gives the direction of the viscoplastic strain increment. Details on material behaviour modelling can be found in Malvern (1969); Mase (1970) and Zienkiewicz and Taylor (2005).

The above equations, input requirements and output specification constitute the first step in the model manipulation process of *Expressing the Model*. An examination of the model shows that much of it will be common between different problems involving the determination of deformation and stress for a material body under loading. For instance, the equilibrium equation (Equation 22) will always apply. Moreover, the form of the constitutive equation (Equation 24) will remain unchanged. However, before solving a specific problem the material model must be specified; that is, the following six variabilities need to be fixed: F, Q, ϕ, κ, γ and the property vector.

These variabilities can be explicitly declared in a DSL that describes (declaratively) a specific model from the family of material behaviour models. An expression is specified for each function and a value is given for the constant γ. The property vector is implicitly built through writing the expressions for the other variabilities. A portion of the DSL, which is a small subset of Maple, is listed below for the expression for F. The listing is in Backus-Naur form (BNF), extended with some regular expression operations. The extended BNF syntax includes [...] to denote optional portions of expressions and + to denote one or more repetitions. Bold font is used to represent the terminal tokens.

⟨expression⟩→⟨number⟩|
(⟨expression⟩)|
⟨expression⟩ˆ⟨expression⟩|
⟨expression⟩∗⟨expression⟩|
⟨expression⟩/⟨expression⟩|
⟨expression⟩+⟨expression⟩|
⟨expression⟩−⟨expression⟩|
−⟨expression⟩|
sin(⟨expression⟩)|arcsin(⟨expression⟩)|cos(⟨expression⟩)|arccos(⟨expression⟩)|
ln(⟨expression⟩)|log(⟨expression⟩)|
⟨simulation-variable-F⟩|⟨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-F⟩→**Kappa**|⟨simulation-variable-stress⟩|⟨simulation-variable-stress-macros⟩
⟨simulation-variable-stress⟩→**SigmaXX**|**SigmaYY**|**SigmaZZ**|**SigmaXY**|
SigmaYZ|**SigmaXZ**
⟨simulation-variable-stress-macros⟩→**Sxx**|**Syy**|**Szz**|**Sxy**|**Syz**|**Sxz**|**Sm**|**J2**|**J3**|**q**
⟨user-defined-constants⟩→⟨string⟩

The expression for $F = F(\boldsymbol{\sigma}, \kappa)$ is a function that can only depend on the 6 components of the stress tensor $\boldsymbol{\sigma}$ (**SigmaXX**, **SigmaYY**, etc.), the hardening parameter (**Kappa** (κ)), stress macros (**Sxx**, **Syy**, etc.) and user defined constants. The stress macros are functions that often arise in continuum mechanics, such as the stress invariants and the deviatoric stress invariants. These macros only reference the allowed stress components. As an example, **Sm** = **SigmaXX** + **SigmaYY** + **SigmaZZ**. The user defined constants correspond to the material properties (property vector) needed by the material model.

4.2 Transformed Model (Finite Element Algorithm)

This section, which presents a numerical algorithm to solve for the stress and deformation within a body, constitutes the second step in the model manipulation process of *Transforming the Model.* The common parts from the previous section are transformed into their finite element method equivalents. At this point the variabilities are left as unspecified; therefore, the algorithm will remain generic and thus be applicable to any material in this family of materials. The transformed model uses the finite element (FE) method (Zienkiewicz et al., 2005) to find the deformation of a material body. As shown below the FE algorithm involves vector and matrix operations and the calculation of the gradients of F and Q with respect to $\boldsymbol{\sigma}$.

In the case where the body follows a Perzyna constitutive equation an implicit time-stepping algorithm, similar to that presented by Stolle (1991), can be derived. 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:

$$\boldsymbol{\Psi}_{i+1} = \int_V \mathbf{B}^T \boldsymbol{\sigma}_{i+1} dV - \mathbf{R}_i = \mathbf{0} \tag{26}$$

where $\mathbf{R}_i$ is the load vector, V is the volume of the body, $\mathbf{B}$ is the kinematic matrix such that $\Delta\boldsymbol{\epsilon} = \mathbf{B}\mathbf{a}_{i+1}$, where $\Delta\boldsymbol{\epsilon} = \Delta t\dot{\boldsymbol{\epsilon}}$. This equation is the weighted integral equivalent of the equilibrium equation (Equation 22). The stress change over the time step may be written as

$$\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i + \Delta\boldsymbol{\sigma}_i \tag{27}$$

The value of $\Delta\boldsymbol{\sigma}_i$ can be found using Δt (the time step size) times Equation 24, which shows that the change in stress depends on the rate of viscoplastic straining. Since the numerical algorithm 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. In the fully implicit version the increment of viscoplastic strain $\Delta\boldsymbol{\epsilon}_i^{vp}$ becomes

$$\Delta\boldsymbol{\epsilon}_i^{vp} = \Delta t \dot{\boldsymbol{\epsilon}}_{i+1}^{vp} = \Delta t \lambda_{i+1} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{28}$$

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 system of equations that can be solved to find the finite element's degrees of freedom ($\mathbf{a}$):

$$\mathbf{K}\mathbf{a} = \mathbf{F} \tag{29}$$

where $\mathbf{K}$ is known as the stiffness matrix and $\mathbf{F}$ as the load vector. Neither of these quantities depends on $\mathbf{a}$, which makes this a linear system of equations. For the first iteration of the algorithm, the values of $\mathbf{K}$ and $\mathbf{F}$ are as follows:

$$\mathbf{K} = \int_V \mathbf{B}^T \mathbf{D}^{vp} \mathbf{B} dV; \mathbf{F} = \mathbf{R}_i - \int_V \mathbf{B}^T \boldsymbol{\sigma}_i dV + \int_V \mathbf{B}^T \Delta\boldsymbol{\sigma}^{vp} dV \tag{30}$$

with

$$\mathbf{D}_{vp} = \mathbf{D}\left[\mathbf{I} - \Delta t C_1 \lambda' \frac{\partial Q}{\partial \boldsymbol{\sigma}} \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{D}\right], \lambda' = \frac{d\lambda}{dF} \tag{31}$$

$$\Delta\boldsymbol{\sigma}^{vp} = \Delta t C_1 \lambda \mathbf{D} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{32}$$

$$C_1 = [1 + \lambda' \Delta t (H_e + H_p)]^{-1} \tag{33}$$

$$H_e = \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{D}(\frac{\partial Q}{\partial \boldsymbol{\sigma}}) \tag{34}$$

$$H_p = -\frac{\partial F}{\partial \kappa} \left(\frac{\partial \kappa}{\partial \boldsymbol{\epsilon}^{vp}}\right)^T \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{35}$$

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

For subsequent passes within an equilibrium iteration loop, the finite element equations, which provide a correction $\Delta\mathbf{a}_i$ for $\mathbf{a}_i$, simplify to

$$\mathbf{K} = \int_V \mathbf{B}^T \mathbf{D} \mathbf{B} dV; \mathbf{F} = \mathbf{R}_i - \int_V \mathbf{B}^T \boldsymbol{\sigma}_i dV \tag{36}$$

The equilibrium iteration loops ceases when the convergence criteria satisfies a given tolerance (toler) as follows:

$$\frac{||\Delta\mathbf{a}||}{||\mathbf{a}||} \leq \text{toler} \tag{37}$$

where $||\mathbf{a}||$ represents the Euclidean norm of the vector $\mathbf{a}$. After solving for the displacements for a given time step the local stresses and strains are updated using a return map algorithm (Zienkiewicz and Taylor, 2005), which is described in McCutchan (2007).

4.3 Extracting Structure and Code Generation

The equations given in the previous section are generic for any Perzyna type material because F, Q, κ, ϕ, γ and the property vector 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 be able to automatically go from the equations to the implementation. This is done by *Extracting the Structure* from the *Transformed Model* and then *Generating Code* using a DSL specification to replace the generic parts with material specific code.

Extracting the Structure from the finite element equations shows that the following terms use the material model dependent variabilities: F, Q, κ, ϕ, γ, $\Delta\boldsymbol{\epsilon}^{vp}$, $\Delta\boldsymbol{\sigma}^{vp}$, $\mathbf{D}$, and $\mathbf{D}^{vp}$. A program called MatGen (McCutchan, 2007) was developed to automatically generate source code for these terms, given a specific material model. The material model is defined using the previously presented DSL. The code generator MatGen needs to express the extracted expressions in terms of the variables that will be known by the finite element algorithm, namely the stress and accumulated viscoplastic strain components. This involves the calculation of several partial derivatives, such as $\frac{\partial F}{\partial \boldsymbol{\sigma}}$. A compiler is needed that can compute the needed derivatives and output source code. As for the other examples in this paper, the Maple computer algebra system provided such a compiler. Maple performs the necessary calculations, in particular the necessary symbolic differentiation, and then converts from the mathematical expressions into C expressions using the "CodeGeneration" function. These C expressions are inlined into the

C++ class defining the material model. This C++ class can then be used by a program that implements the finite element algorithm presented in the previous section.

The interface provided by the generated code, which matches the extracted structure listed above, is intended to match the needs of the numerical algorithm. The goal was automatic generation of the code necessary to computationally solve the problem. At this point in time, the fourth step in the model manipulation process (*Optimize the Computation*) was not emphasized. Instead the goal was to automatically generate code for new constitutive equations in a manner that is considerably less time consuming and error prone than using hand calculations. Moreover, the automatic code generation facilitates non-experts in computational mechanics easily experimenting with new and different materials.

We illustrate the simplicity and effectiveness of MatGen by considering the calculation of one example term, H_e (Equation 34), and how MatGen reduces the potential for error and the time needed for the calculation. Comparing the symbolic output from Maple to hand derived versions of H_e for a viscoelastic fluid shows the same result that $H_e = 3G$, where G is the shear modulus (McCutchan, 2007). However, the hand derived version was complex, took a nontrivial amount of time, and required expert knowledge. In particular, the by hand derivation takes 5 pages of equations and explanation in (McCutchan, 2007, pages 77–81). The derivation uses the chain rule of calculus, several stress invariants, the Einstein index notation (Einsten, 1916), vector calculus, and knowledge from continuum mechanics, such as the fact that the trace of the deviatoric stress tensor is zero. The MatGen version on the other hand only required using the DSL to specify the model for a viscous fluid, as follows:

$$F = Q = q; \phi = F; \kappa = 0; \gamma = 1/2\eta \tag{38}$$

where q is the effective stress, which is provided by a macro in MatGen, and η is the viscosity, which is the one necessary material property. The calculation of other terms in the finite element algorithm are at least as complex, time consuming and error prone, as the calculation of H_e. In these other cases MatGen was just as simple and effective, although Maple was unable to simplify these other expressions to be identical to the hand derived versions. In these cases though the expressions were found to be equivalent by verifying their numerical agreement.

5 Conclusion

We have presented three examples of the model manipulation process in scientific computing. Each of the examples began with the same step of *Expressing the Model*, where each model was expressed in an appropriate DSL. The models were initially expressed in declarative form using the mathematics of the problem domain. At this stage the model can be understood by anyone with knowledge of the problem domain, without the need for expertise in computation. In the first two examples the model consisted of a minimization problem and in the third example an ordinary differential equation. In each case, a DSL could be used to specify the variabilities between models, where the variabilities include the following: the class of functions, which parameters to optimize for, the number of superpositions of basis functions, the yield function, material properties, etc.

The proposed second step was *Transforming the Model*, where the initial model is transformed into a form more suitable for computational solutions. The first two examples transformed the model to use Newton's method and the third example used a transformation to a finite element algorithm. In each case the transformed model remains generic with respect to the variabilities identified in the first step; therefore, the transformed models represent families of algorithms.

At this point in the process we *Extract Structure* from the model. For instance, we recognized that the Jacobian and the Hessian share many common subexpressions, the Hessian is upper triangular, recurrence relations frequently occur in time-series models, the derivatives in the Jacobian and the Hessian can be expressed in terms of the model itself, and in the material modelling algorithm only a small set of terms use the material model dependent variabilities. In the first two examples the extracted structure is used to *Optimize the Computation* of the "solution" of the model. For instance, improved performance is possible by joint optimization of the Jacobian and Hessian, sub-expression elimination, recurrence relation and differential equation optimizations. In each case Maple was used to *Generate the Code* that was actually used to carry out the solution. The code generation transforms the generic algorithm from the second step into a specific algorithm by using the values of the variabilities specified using the DSL.

We have demonstrated that the approach where model manipulation is used as a first step before generating code in a numerical setting has several advantages for different modelling activities.

1. The conventional approach, for example where the various gradients are worked out by hand in advance of implementation, is difficult and error prone. Replacing this step by symbolic processing reduces the workload, allows non-experts to deal with new problems, and increases reliability.

2. Although the generated code is for a particular numerical algorithm, given the existing framework it is straightforward to generate new programs that meet the needs of other algorithms.

3. Any additional information available at the symbolic processing stage can be used to improve performance. Whether it be because of a known differential or recurrence relation in the model, or it is known that the model is planar or has a natural symmetry, all of this can be used for optimizing the resulting code.

4. In certain situations, the performance gains from taking advantage of the problem structure can be impressive.

We believe that we are discovering a new development methodology for high-level scientific applications that can really leverage Domain Specific Languages, model transformations and program transformation to yield a process which is friendlier to the domain expert, yields further insights into the original problem, and produces faster and more reliable code.

Acknowledgements

The financial support of the Natural Sciences and Engineering Research Council (NSERC) is gratefully acknowledged.

References

Sergei A. Abramov, Jacques Carette, Keith O. Geddes, and Ha Q. Le. Telescoping in the context of symbolic summation in Maple. *Journal of Symbolic Computation*, 38(4):1303–1326, October 2004.

C. Anand, J. Carette, and A. Korobkine. Target recognition algorithm employing Maple code generation. In *Maple Summer Workshop*, 2004.

C. K. Anand, J. Carette, A. Curtis, and D. Miller. COG-PETS: Code generation for parameter estimation in time series. In *Maple Conference 2005 Proceedings*, pages 198–212. Maplesoft, 2005.

Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation*, pages 242–249, 1994. URL `citeseer.ist.psu.edu/bachmann94chains.html`.

F. Bloch. Nuclear induction. *Phys. Rev.*, 70:460, 1946.

Frédéric Chyzak and Bruno Salvy. Non-commutative elimination in Ore algebras proves multivariate holonomic identities. *Journal of Symbolic Computation*, 26(2):187–227, 1998.

A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

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

Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science, June 2–4 2003, St. Petersburg (Russia) and Melbourne (Australia)*, 2003a.

Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *Int. J. High Perf. Comput. Appl.*, 17:125–131, 2003b. also Lapack Working Note 157, ICL-UT-02-07.

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

Jason Fotinatos, Ryan Deak, and Thomas Ellman. Automated synthesis of numerical programs for simulation of rigid mechanical systems in physics-based animation. *Automated Software Engineering*, 10(4):367–398, 2003.

Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000. ISBN 0–89871–451–6.

H. G. Kahrimanian. Analytical differentiation by a digital computer. Master's thesis, Temple University, May 1953.

Andres M. Kanner. Abnormalities identified with t2 relaxometry in hippocampi remote from the seizure focus: Do they mean anything?. *Epilepsy Currents*, 4(3):120–121, 2004.

Ken Kennedy, Bradley Broom, Keith D. Cooper, Jack Dongarra, Robert J. Fowler, Dennis Gannon, S. Lennart Johnsson, John M. Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *J. Parallel Distrib. Comput.*, 61(12):1803–1826, 2001.

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.

John McCutchan. A generative approach to a virtual material testing laboratory. Master's thesis, McMaster University, 2007.

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

Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981. ISBN 0–540–10861–0.

Bruno Salvy and Paul Zimmermann. Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

W. Spencer Smith. *Simulating the Cast Film Process Using an Updated Lagrangian Finite Element Algorithm*. PhD thesis, McMaster University, Hamilton, ON, Canada, 2001.

Dieter 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.

Joseph M. Thames. SLANG, a problem-solving language for continuous-model simulation and optimization. In *Proceedings of the ACM 24th National Conf.* ACM, New York, 1969.

O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method For Solid and Structural Mechanics.* Elsevier Butterworth-Heinemann, 6th edition, 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.