Case Studies in Model Manipulation for Scientific Computing

J. Carette, S. Smith, J. McCutchan, C. Anand, and A. Korobkine

Computing and Software Department, McMaster University, Hamilton, ON, CANADA
{carette, smiths, mccutcjs, anandc, korobkao}@mcmaster.ca

Abstract. The same methodology is used to develop 3 different applications. We begin by using a very expressive, appropriate Domain Specific Language, to write down precise problem definitions, using their most natural formulation. Once defined, the problems form 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 that 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 shown in Fig. 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.

For scientific applications, the most appropriate DSL is well-known: *mathematics*. More difficult is 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 allow direct manipulation, in and by the language itself, of mathematical expressions (and more generally of mathematical specifications). The only languages that currently combine the necessary richness and ease of manipulation are the languages of *Computer Algebra Systems*. In our case, because it is the system we are

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

S. Autexier et al. (Eds.): AISC/Calculemus/MKM 2008, LNAI 5144, pp. 24–37, 2008.

1. *Express the Model* - the model is declaratively expressed in a DSL,
2. *Transform the Model* - transform the initial model 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 for the solution.

Fig. 1. Typical model manipulation steps

(by far) the most familiar with, we 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 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, classical code generation techniques can be applied to generate C code 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 [11,22],
2. when a program can be expressed succinctly in a domain-specific language, but requires lengthy and complex code in a mainstream language. [6,7]

The first situation occurs most famously when *automatic differentiation* [13] is required and applicable. There is ample literature (from [25] onwards) that shows that smooth optimization problems are incomparably easier to solve when Jacobians and Hessians are available. 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 as well, which has caused the growing popularity of GUI-builders, lexer and parser generators, Java-from-DTD builders, etc. This trend is also present in the scientific computation community [8,9].

The problems well-suited to our approach are those which:

1. can be succinctly described using mathematics as the "domain language,"
2. needs information, like derivatives, easily obtained from the model, and
3. requires experimentation and manipulation at the "model" level.

The downsides of using a DSL, as given by [7], are not relevant when using a mathematical programming language (such as Maple).

It is worth repeating that the most important step is that of "model manipulation." Our aim is to automate *every other step of the problem-solving process* to ensure that a designer's time is spent thinking about the semantics and the structure of the problem to solve, and not wasted on mundane computational tasks. Eventually, we would hope to provide higher-level abstractions for this step as well. Several reviewers have, naïvely in our opinion, asked why we do not provide more automation for this step. The answer is simple: we only know how to automate very particular cases, and in fact believe that there are no general recipes to follow. This is not to say that particular cases cannot be fully automated, nor even that these particular cases are "rare" – quite the contrary.

Part of our aim is to free our own time, so that we can concentrate on the pure problem-solving parts of scientific software development. Once we have reliably achieved that, we will then strive to develop automated tools, using our scientific and engineering knowledge of typical solutions, to as many problem classes as possible.

We will present 3 applications developed using this methodology: real-time visual tracking of a target, data fitting in model-based time series and material behaviour modelling. To highlight the similarities between the examples, in each case reference will be made to the model manipulation steps shown in Fig. 1. Significantly more details on these examples can be found in [3].

2 Visual Tracking

This example is based on [1]. 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. Here we will focus on recognition of radially-symmetric, essentially compact targets, which we will call *spots*.

2.1 Model of Spot Fitting

We can *Express the Model* of spot recognition as the least squares fit between actual light intensity (ϕ_p) and the equation ($v_1 f(p) + v_0$) describing the spot:

$$\min_{\mathcal{U}} F = \sum_{p \in \Omega} \left(\phi_p - (v_1 f(p) + v_0)\right)^2 \tag{1}$$

where v_0 is the background illumination, v_1 is the brightness of the centre of the spot, p is a pixel, Ω is a region of pixels and $f(p)$ is a polynomial. The function $f(p)$ depends on k_1, k_2, which determine the radial profile of the spot; b_x, b_y, which define the coordinates of the ellipse centre; and finally a_1, a_2, a_3, which define the shape of the elliptical boundary. We minimize F over $\mathcal{U}$, where $\mathcal{U} \subseteq \{v_0, v_1, k_1, k_2, b_x, b_y, a_1, a_2, a_3\}$. Figure 2 shows an example.

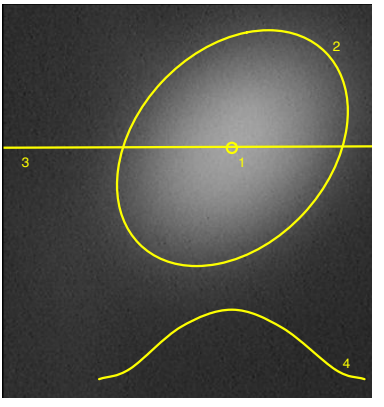

Fig. 2. Actual image of a gray-scale target, showing the spot's centre (1), shape (2) and cross-section (3, 4)

Equation 1 actually represents a family of models, distinguished by the choice of $\mathcal{U}$. Domain-specific information allows us to *Extract Structure* via appropriate choices of $\mathcal{U}$. Note that a naive implementation where we simultaneously optimize all variables will fail because the target recognition problem is not convex, forcing us into multiple solver stages.

2.2 Transformed Model (Newton's Method Solver)

The *Transformed Model* for finding the minimum in Eq. 1 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. Denoting by $\mathbf{J}_{\mathcal{U}}$ the Jacobian of F and $\mathbf{H}_{\mathcal{U}}$ the Hessian of F with respect to the variables $\mathcal{U}$, Newton iteration for an iterative solution vector $\mathbf{u}_n$ is defined by:

$$\mathbf{u}_{n+1} = \mathbf{u}_n - \mathbf{H}_{\mathcal{U}}(\mathbf{u}_n)^{-1}\mathbf{J}_{\mathcal{U}}(\mathbf{u}_n) \tag{2}$$

2.3 Extracting Structure and Generating Code

To improve performance, we can *Extract Structure* (again) from the transformed model. In particular, we know that large arrays are needed to store the captured images and that computing the sum over each elements in those arrays is expensive. Efficient use of cache would help reduce execution time (as this is bounded by memory accesses). This is most easily done by localizing computations within a solver iteration. Furthermore, from their definitions, we know that Jacobian and Hessian matrices will contain many common subexpressions; therefore, optimization on the "the inner sum" is crucial. We also know that since Hessian matrices are symmetric, we only need to calculate their upper triangular portion. Using this information suggests that for the Jacobian and the Hessian we should jointly *Optimize the Code*. Measurements of floating point operations in code generated using these optimization strategies confirms our expectations (see Table 1).

Table 1. Number of flops per pixel in generated solvers

	jointly optimized		separately optimized	
	+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

There is an advantage to the joint optimization of the Jacobian and the Hessian matrices, which would not be feasible without *Optimizing the Code*. Maple's `codegen[optimize]` function, especially with the `tryhard` option, eliminates common subexpressions very effectively when these matrices are generated together. If the optimization of code is performed separately 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.

3 Parameter Estimation in Model-Based Time Series

This example of parameter estimation from time-series data is extracted from [2]. 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 (MRS) and Relaxometry. Figure 3 shows an example from MRS of the decomposition of a measured magnetic resonance spectrum for soya bean oil.

3.1 Expressing the Mathematical Model

Expressing the Model for parameter estimation shows that we have a more general version of the least squares fitting example presented in the previous section. 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

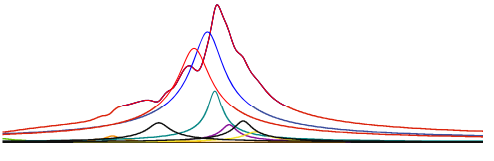

Fig. 3. Soya bean oil spectrum (maroon) and component estimates

various sources to the measured data. Through minimization of an objective function F, an optimal set of parameters may be determined:

$$\min_{x_1^1, x_2^1, \ldots, x_n^1, \ldots, x_n^s} \sum_t \left\| y(t) - \sum_{s \in \{\text{sources}\}} a_s f_s(x_1^s, x_2^s, \ldots, x_n^s, t) \right\|^2 . \tag{3}$$

where x_j^s denotes the x_j'th parameter of peak s. Equation 3 *Expresses the Model* for parameter estimation of a time series. An important part of the mathematical modelling step is to explicitly declare the class of functions f to consider, which parameters to optimize for, and how many superpositions of the basis function should be used for fitting. This gives important structural information from which we can *Extract Structure*.

As our objective functions will all be analytic, we can safely use Newton's method to solve the minimization problem; therefore, Newton's method forms the *Transformed Model*, as it did for the last example (see subsection 2.2).

3.2 Extracting Structure

Extracting Structure from the model shows the frequent occurrence of recurrence relations, since 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 $z_0 = a, \quad z_{j+1} = kz_j, \quad k = e^{-(d+if)}$.

We symbolically obtain the recurrence equation satisfied by the model f with respect to the main variable t via the `IsHypergeometricTerm` function from the `RationalNormalForms` Maple package. This function uses advanced symbolic techniques 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.

Further efforts to *Extract Structure* show that 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 are simply expressible in terms of the model itself. Considering a simple model with first-order dependence on a parameter b,

$$f(b) = ae^{bp(x)} \text{ and } \frac{\partial f}{\partial b} = p(x)ae^{bp(x)} = p(x)f(b) \tag{4}$$

which shows that the derivative can be expressed in terms of f. 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, it is important to factor them out.

`gfun[holexprtodiffeq]` is used to determine the differential equation(s) satisfied by the model f. The abbreviations stand respectively for *generating function* and *holonomic expression to differential equation*. The package `gfun` and the theory of holonomic (or D-finite) functions are described in [24] and [4], respectively.

3.3 Code Generation

We need to *Generate Code* 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 Jacobian 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 Jacobian entry.

If f is a complex (vector) function, then f, 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.

The code to calculate F, $\mathbf{J}$ and $\mathbf{H}$ 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 and transformed into a C function.

The generation algorithm can be explained more specifically 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-sub-expression 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

This is about 500 lines (counting comments and blank lines) of very clear Maple code. The core ideas fit in about 50 lines, with the rest needed to get around various idiosyncracies, keep the code modular and clean, and simply further automate the process.

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.

4 Material Behaviour Modelling

Modelling the response of materials under loading is of critical importance to scientists and engineers. To model the deformation and stress within a solid body, we turn to the constitutive equation, which postulates a dependence of the stress on the history of deformation. A wide range of varied and complex constitutive equations are used in practise. Although the behaviour of these models can vary greatly, the underlying mathematics is very similar. Using the correct abstraction, a wide range of material behaviours form a family of material models. Using model manipulation we can quickly generate code for a specific member of this family.

4.1 The Mathematical Model Relating Stress and Deformation

The goal of material modelling is to find the stress ($\boldsymbol{\sigma} : \mathbb{R}^6$) as a function of time ($t : \mathbb{R}$). That is, to return the function $\boldsymbol{\sigma}(t) : \{t : \mathbb{R} | t_{beg} \leq t \leq t_{end}\} \rightarrow \mathbb{R}^6$, where t_{beg} and t_{end} delimit the duration of the simulation. The stress can be found by solving the constitutive equation, which in rate form is:

$$\dot{\boldsymbol{\sigma}} = \mathbf{D}\left(\dot{\boldsymbol{\epsilon}} - \gamma < \phi(F(\boldsymbol{\sigma}, \kappa)) > \frac{\partial Q(\boldsymbol{\sigma})}{\partial \boldsymbol{\sigma}}\right) \text{ and } \boldsymbol{\sigma}(t_{beg}) = \boldsymbol{\sigma}_0 \tag{5}$$

where $< \phi(F) >= \phi(F)$, if $F > 0$, and 0 otherwise. This equation is based on the viscoplastic constitutive equation presented by Perzyna [19], which 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 Eq. 5, the condition $F = 0$ defines a surface in 6 dimensional stress space, which can be visualized as in Fig. 4. Inside the surface ($F < 0$) the material response will be purely elastic, and outside the response is viscoplastic. When the material has yielded, which occurs when the stress path

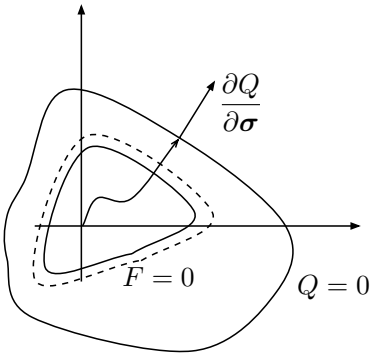

Fig. 4. Yield Function, Hardening and the Plastic Potential in Stress Space

reaches the yield surface, this surface may change shape, as shown in Fig. 4 by the dashed line. Details on material behaviour modelling can be found in [16].

The above constitutive equation, together with the equilibrium equation, are the *Expression of the Model*. The model is very similar between different problems. The only variabilities, which need to be set for a specific material before solving a given problem, are the following: F, Q, ϕ, κ, γ and the property vector, where the property vector consists of the material properties. These variabilities can be explicitly specified in a DSL that describes (declaratively) a particular material model from the family.

4.2 Transformed Model (Finite Element Algorithm)

The second step in the model manipulation process is *Transforming the Model*. In this example, the common parts of the model are transformed into their finite element (FE) method [27] equivalents. This step leaves the variabilities as unspecified; therefore, the algorithm will remain generic and thus be applicable to any material in the family. At the moment, there is no clear algorithm to automatically transform Eq. 5 to an FE equivalent. Currently the transformation seems to require human insight and expert knowledge of the available family of algorithms. However, by keeping the algorithm generic, multiple instances that apply to a variety of materials can quickly be generated.

The FE algorithm selected is a fully implicit time-stepping algorithm that includes a correction back to the yield surface when this is required. The algorithm involves vector and matrix operations and the calculation of the gradients of F and Q with respect to $\boldsymbol{\sigma}$ [3]. The FE equation to solve for the displacement degrees of freedom ($\mathbf{a}$) is as follows:

$$\mathbf{Ka} = \mathbf{F} \tag{6}$$

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:

$$\mathbf{K} = \int_V \mathbf{B}^T \mathbf{D}^{vp} \mathbf{B} dV; \mathbf{F} = \mathbf{R}_i - \int_V \mathbf{B}^T \boldsymbol{\sigma}_i dV + \int_V \mathbf{B}^T \Delta\boldsymbol{\sigma}^{vp} dV \tag{7}$$

with

$$\mathbf{D}_{vp} = \mathbf{D}\left[\mathbf{I} - \Delta t C_1 \lambda' \frac{\partial Q}{\partial \boldsymbol{\sigma}} \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{D}\right], \lambda' = \frac{d\lambda}{dF} \tag{8}$$

$$\Delta\boldsymbol{\sigma}^{vp} = \Delta t C_1 \lambda \mathbf{D} \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{9}$$

$$C_1 = [1 + \lambda' \Delta t (H_e + H_p)]^{-1} \tag{10}$$

$$H_e = \left(\frac{\partial F}{\partial \boldsymbol{\sigma}}\right)^T \mathbf{D}\left(\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right) \tag{11}$$

$$H_p = -\frac{\partial F}{\partial \kappa} \left(\frac{\partial \kappa}{\partial \boldsymbol{\epsilon}^{vp}}\right)^T \frac{\partial Q}{\partial \boldsymbol{\sigma}} \tag{12}$$

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

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

$$\mathbf{K} = \int_V \mathbf{B}^T \mathbf{D} \mathbf{B} dV; \mathbf{F} = \mathbf{R}_i - \int_V \mathbf{B}^T \boldsymbol{\sigma}_i dV \tag{13}$$

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

$$\frac{||\Delta\mathbf{a}||}{||\mathbf{a}||} \leq \text{toler} \tag{14}$$

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 [26], which is described in [17].

4.3 Extracting Structure and Code Generation

After *Extracting the Structure*, which consists of the terms involving F, Q etc., from the *Transformed Model*, the next step is *Generating Code* using a DSL specification to replace the generic parts with material specific code. A program called MatGen [17] was developed to do this. MatGen needs to calculate the required derivatives and output source code for terms such as H_e (Eq. 11). Like the other examples in this paper, Maple was used to do this. Maple performs the necessary symbolic computations and is then used to convert from mathematical expressions into C expressions using the "CodeGeneration" package. These C

expressions are inlined into a C++ class defining the material model. This class can then be used by an FE analysis program.

Note that the step *Optimize the Computation* was not emphasized in this example. Instead the goal was to automatically generate code for new constitutive equations in a manner that is simpler, less time consuming and less error prone than using hand calculations. We can illustrate that we reached this goal by considering the calculation of the example term, H_e. Comparing the symbolic output from Maple to a hand derived versions of H_e for a viscoelastic fluid shows the same result that $H_e = 3G$, where G is the shear modulus [17]. However, the hand derivation was complicated, took a nontrivial amount of time, and required expert knowledge. In particular, the hand derivation took 5 pages of equations and explanation [17, pages 77–81]. The derivation used the chain rule of calculus, several stress invariants, the Einstein index notation, 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: $F = Q = q; \phi = F; \kappa = 0; \gamma = 1/2\eta$, where q is the effective stress, which is provided by a macro in MatGen, and η is the material property of viscosity. The calculation of other terms in the FE 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.

Although the *Optimize the Computation* step was not emphasized for the current example, the possibility certainly exists that the FE algorithm can be mined for structure in a manner similar to what was done for the previous two examples. Although the code generated by Maple is not currently efficient, an expert could potentially further *Extract Structure* to improve the efficiency. This possibility illustrates the current need for human insight in the model manipulation process. Additional human creativity and ingenuity at the initial stages of the model manipulation process can facilitate the subsequent automated steps and result in much more efficient code. Further investigation of material behaviour modelling is left as future work.

5 Related Work

The many people working on *Problem Solving Languages* [25] and *Problem Solving Environments* [10,12,14,18,20,21,23] (to cite just a few) implicitly believe in our thesis. By and large, they are however working at creating environments for solving particular problems. For each problem class, the solving methodology is well-enough understood that most of the process can be encapsulated in one piece of software.

Take one of the most impressive examples: SPIRAL [21]. They are essentially following the same approach that we are, but they have concentrated on

documenting different aspects of their work. We have concentrated on pulling out the process, and making sure that we automate all that we can. In the domain of signal processing, they have achieved a high level of automation by doing exactly as we preach: automating the "rest" of the process, and then concentrating on the part where new mathematical insight makes a real difference. We believe that this can be done in general by using "mathematics" as the DSL.

Another approach is code extraction from constructive proofs, most notably from Coq proofs [15]. This is an extremely exciting prospect, but is too far on the leading edge of current research to be properly evaluated at this time. Certainly there are issues [5] where the style of one's proof has dramatic effects on the quality of the extracted software! Of course, there is also the issue that many parts of advanced mathematics (like holonomy) are not yet implemented in Coq, but have quite mature implementations in CASes.

6 Conclusion

We have demonstrated that the model manipulation development methodology for generation of (numeric) solution to scientific computation problems has several advantages.

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. For instance, if there is a known differential or recurrence relation in the model, this can be used for optimizing the code.
4. In certain situations, the performance gains from taking advantage of the problem structure can be impressive.

We have chosen to be pragmatic and reuse a well-known existing tool: Maple. We are well aware that this is a far from optimal choice. A better approach would require the use of a semantically richer tool (as provided by many theorem proving environments); but none of these tools have existing libraries as rich as Maple's. Certainly none of them, to our knowledge, contain tools for dealing with holonomic functions. We look forward to the day where semantically richer environments are as computationally capable as today's CASes.

We believe that we are discovering a new development methodology for high-level scientific applications that leverages DSLs, model transformations and program transformation to yield a process that is friendlier to the domain expert, provides insights into the original problem, and produces faster and more reliable code. We believe that tool developers who keep this process firmly in mind when they design new tools (or improve old ones) can produce environments which will improve the productivity of scientific software developers.

Acknowledgements

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

References

1. Anand, C., Carette, J., Korobkine, A.: Target recognition algorithm employing Maple code generation. In: Maple Summer Workshop (2004)
2. Anand, C.K., Carette, J., Curtis, A., Miller, D.: COG-PETS: Code generation for parameter estimation in time series. In: Maple Conference 2005 Proceedings, Maplesoft, pp. 198–212 (2005)
3. Carette, J., Smith, S., McCutchan, J., Anand, C., Korobkine, A.: Model manipulation as part of a better development process for scientific computing code. Technical Report 48, Software Quality Research Laboratory, McMaster University (2007)
4. Chyzak, F., Salvy, B.: Non-commutative elimination in Ore algebras proves multivariate holonomic identities. Journal of Symbolic Computation 26(2), 187–227 (1998)
5. Cruz-Filipe, L., Letouzey, P.: A Large-Scale Experiment in Executing Extracted Programs. In: 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2005 (2005)
6. van Deursen, A., Klint, P.: Little languages: Little maintenance? Journal of Software Maintenance 10, 75–92 (1998)
7. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices 35(6), 26–36 (2000)
8. Dongarra, J., Eijkhout, V.: Self-adapting numerical software for next generation applications. Int. J. High Perf. Comput. Appl. 17, 125–131 (2003); also Lapack Working Note 157, ICL-UT-02-07
9. Fotinatos, J., Deak, R., Ellman, T.: Automated synthesis of numerical programs for simulation of rigid mechanical systems in physics-based animation. Automated Software Engineering 10(4), 367–398 (2003)
10. Gaffney, P.W., Houstis, E.N. (eds.): Programming Environments for High-Level Scientific Problem Solving, Proceedings of the IFIP TC2/WG 2.5 Working Conference on Programming Environments for High-Level Scientific Problem Solving, Karlsruhe, Germany, September 23-27, 1991. IFIP Transactions, vol. A-2. North-Holland, Amsterdam (1992)
11. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. In: Frontiers in Appl. Math., vol. 19. SIAM, Philadelphia (2000)
12. Hunt, K., Cremer, J.: Refiner: a problem solving environment for ode/dae simulations. SIGSAM Bull. 31(3), 42–43 (1997)
13. Kahrimanian, H.G.: Analytical differentiation by a digital computer. Master's thesis, Temple University (May 1953)
14. Kennedy, K., Broom, B., Cooper, K.D., Dongarra, J., Fowler, R.J., Gannon, D., Johnsson, S.L., Mellor-Crummey, J.M., Torczon, L.: Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. J. Parallel Distrib. Comput. 61(12), 1803–1826 (2001)
15. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)

16. Malvern, L.E.: Introduction to the Mechanics of Continuous Medium. Prentice-Hall, Englewood Cliffs (1969)
17. McCutchan, J.: A generative approach to a virtual material testing laboratory. Master's thesis, McMaster University (2007)
18. Parker, S.G., Miller, M., Hansen, C.D., Johnson, C.R.: An integrated problem solving environment: the SCIRun computational steering system. In: 31st Hawaii International Conference on System Sciences (HICSS-31) (1998)
19. Perzyna, P.: Fundamental problems in viscoplasticity. In: Advances in Applied Mechanics, pp. 243–377 (1966)
20. Pound, G.E., Eres, M.H., Wason, J.L., Jiao, Z., Keane, A.J., Cox, S.J.: A grid-enabled problem solving environment (pse) for design optimisation within matlab. In: IPDPS 2003: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, p. 50.1. IEEE Computer Society, Washington (2003)
21. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2) (2005)
22. Rall, L.B.: Automatic Differentiation: Techniques and Applications. LNCS, vol. 120. Springer, Berlin (1981)
23. Rice, J.R., Boisvert, R.F.: From scientific software libraries to problem-solving environments. IEEE Computational Science & Engineering 3(3), 44–53 Fall (1996)
24. Salvy, B., Zimmermann, P.: 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)
25. Thames, J.M.: SLANG, a problem-solving language for continuous-model simulation and optimization. In: Proceedings of the ACM 24th National Conf. ACM, New York (1969)
26. Zienkiewicz, O.C., Taylor, R.L.: The Finite Element Method For Solid and Structural Mechanics, 6th edn. Elsevier Butterworth-Heinemann, Amsterdam (2005)
27. Zienkiewicz, O.C., Taylor, R.L., Zhu, J.Z.: The Finite Element Method Its Basis and Fundamentals, 6th edn. Elsevier Butterworth-Heinemann, Amsterdam (2005)