European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)

vol. 3, no. 1-2, 2008, pp. 167-180
ISSN 1790–8140

Automatic Code Generation and Optimization in Maple[1]

Allan Wittkopf

Maplesoft,
Mathematics Group,
Waterloo, Ontario, Canada

Received 31 October, 2007; accepted in revised form 18 December, 2007

Abstract: In this article we discuss the advantages (and pitfalls) in developing code optimization and Maple to C conversion programs for Maple procedures, specifically those required for numerical differential equation solution. Special treatment related to code optimization for large sets of equations is a major focus. Issues related to conversion of Maple code to C code are discussed, and finally a set of comparisons for code optimization and ODE problems are provided.

Keywords: Maple, C, Code Conversion, Code Optimization

Mathematics Subject Classification: AMS-MOS Subject Classification Index 34-04

1 Introduction

General problem solving environments (PSE) provide rich and varied functionality, including the ability to work with mathematics in natural notation and coverage of many areas of mathematics in a single environment. The Maple PSE has packages and facilities for everything from group theory to high performance numerical linear algebra. As such, they need to be general purpose, and often this is at the expense of the efficiency (or in some cases completion) of tasks related to numerical solution of moderate to large differential equation problems.

The first topic of this paper is an implementation of code optimization, which optimizes Maple code to more efficient Maple code, and is very relevant to working within the Maple PSE, as there are many features one can utilize, but there are just as many pitfalls that one must cautiously navigate.

The second topic is an implementation of a Maple to C code conversion and compilation tool. Though the algorithms used in this implementation are not novel or new, there are several hurdles to overcome to make the process clean and efficient, and there are also improvements that can be done with respect to interfacing the compiled code with existing tools (such as Maple's numerical ODE solvers).

Finally several examples with timings are provided for the use of these implementations. Note that all timings and comparisons were obtained in a development version of Maple 12 on a 64-bit Linux Athlon X2 4400 with 2 Gb of RAM.

[1]Published electronically March 31, 2008

Since this paper concerns two types of code, the code being processed, and the routines performing the processing, the terms *code* and *routine* can be ambiguous. To clarify this, whenever routines in the implementation are being referred to, the term *routine* will be used, while the term *code* will be used exclusively for the code being processed (optimized, compiled, or called).

2 A Maple Primer and Numerical ODEs in Maple

Much of the following material requires some degree of familiarity with the Maple symbolic PSE, so some of the basic information is provided here.

Maple is a symbolic computation system, and can be used to represent mathematics in a more abstract way than is available in compiled programs. It has the concepts of unassigned variables, and functions acting on them, solution of systems of equations (linear and nonlinear), symbolic differentiation and integration, symbolic summation, and many other mathematical operations ranging from group theory computations to exact solutions of ordinary and partial differential equations (to cite a couple more examples). It also has several numerical algorithms to obtain numerical results for many of the operations described above.

Much of Maple has been written in its own interpreted language, and this code can be viewed directly from the interface. Maple coded procedures can themselves be viewed as data objects, and can be examined, manipulated, or as we will see later optimized and compiled.

Within Maple memory is managed using garbage collection, and even routines that are part of Maple can be collected when they are no longer in use. To make the representation of data in Maple more compact, it identifies common subexpressions automatically, and stores the data for these expressions only once (we call this uniquification). For example, the expression $x + y$ and the expression $y + x$ are stored in one of the two orders, and the other is recognized to represent the same structure (but in a different order), so the two expressions are stored only once.

The core interpreter in Maple (the kernel) is a compiled program (C and C++), as well as some functionality that lives in compiled code (shared libraries). For example, double precision linear algebra operations on floating point matrices are performed in compiled code. The compiled form of these procedures can be significantly faster than the equivalent code running inside Maple, as the compiled code has no interpreter overhead, and compiled code memory management must be coded explicitly, so it introduces no garbage collection overhead. In general compiled code is restricted to work with numerical datatypes, though it is possible to work directly with Maple datatypes. In order to do so, one must fully understand the Maple datatype internals, and code cautiously, as any Maple data objects created in the compiled code may generate garbage which will not be collected until the call to the routine returns to Maple.

Maple's primary numerical ODE solvers are hybrid programs. The initial processing of the input system, to cast it into $y' = f(t, y(t))$ form is written in Maple code, the core engine that performs the numerical integration is a compiled program, but when the compiled program needs to compute the values from $y' = f(t, y(t))$ it calls back into Maple (from the compiled code) to perform this computation. The code for computing stiff solutions or DAE solutions is similar, but further callbacks are required (to compute Jacobians or residuals).

The goal of the two implementations in this paper is to make the code used to evaluate $y' = f(t, y(t))$ (and the Jacobians or residuals) more efficient, and to eliminate the callback into interpreted Maple by providing compiled functions to the core engine.

3 Code Optimization

The primary goal of the code optimization implementation is *not* to obtain the best possible optimization, but rather to get very close in as little time and memory usage as possible. Note

that the code optimizer produces Maple code from Maple code, but the output code is generated in a form more easily converted to C code (when possible).

There already exists a routine in Maple with the purpose of optimizing Maple code, namely `codegen[optimize]`, but it was found that for several classes of problems the degree of optimization is insufficient, and when requesting stronger optimization (via command options) the command takes a great deal of time. This is demonstrated in the examples (§5.1,§5.2). In addition, the primary goal is to optimize for the type of codes we expect to see when computing solutions of ODE, DAE or PDE problems. This represents a smaller class of problems than the existing code was designed for, allowing some trade-off to gain efficiency at the expense of reducing flexibility.

Two of the key facilities in Maple that one can leverage when implementing a code optimization routine are the Maple `table` data structure (the primary implementation of which is a hash table), and Maple's automatic expression uniquification facilities mentioned in §2. The uniquification makes comparison of expressions trivial, and identification of common subexpressions a snap, because one need not look at the underlying structure, but simply the address at which the expression is stored.

One bane of code optimization from within Maple is common to many interpreted languages (such as Java) and it is the fact that the language uses garbage collection for memory management instead of manual memory management via explicit allocation and freeing. This is actually a benefit from the users' perspective, as they do not need to remember to explicitly clear memory they are no longer using, but it becomes a challenge for programmers when trying to implement code to run in a small footprint. This becomes more of an issue when one creates compiled code to work with Maple expressions, as when one is running the compiled code, the garbage collection mechanism is never triggered.

The types of optimizations performed are fairly standard, and are as follows:

1. Common subexpression optimization
2. Fractional power optimization ($a = x^{1/2}, b = x^{3/2} \Rightarrow a = x^{1/2}, b = xa$)
3. Integer power optimization ($a = x^8, b = x^{10} \Rightarrow t_1 = x^2, t_2 = t_1^2, a = t_2^2, b = at_1$)
4. Sub-product optimization ($a = xyz, b = wyz \Rightarrow t_1 = yz, a = xt_1, b = wt_1$)
5. Sum multiple optimization ($a = x - y, b = 2y - 2x \Rightarrow a = x - y, b = -2a$)
6. Other odd optimizations

At one point partial sum optimization was implemented:

$$a = x + y + z, b = x + y - z \Rightarrow t_1 = x + y, a = t_1 + z, b = t_1 - z,$$

but it was found to be too expensive, and to provide too little benefit for the set of benchmark problems used during the development.

There are several pitfalls to avoid when implementing routines of this type, and some are well known, while others are not. We will describe here a few of the major ones, and describe the method used for a more efficient implementation.

3.1 Lists and Sequences

This issue is quite well known, and involves the method of constructing a list of values inside a loop. Note that lists share the same basic implementation as sequences in Maple (a list is just a wrapped sequence), so we will restrict the discussion and the benchmarks to the use of sequences. This process is used extensively in nearly all of the optimization routines, as looping over the

code statements and replacing, adding, or removing statements as the processing is performed is common to all of them.

As a benchmark problem we construct the sequence of integers from 1 to n inside a loop. It is important to keep in mind that there are much more efficient ways to accomplish this operation (such as the `seq` operator), but in some cases the values being placed in the sequence may depend upon values previously computed in the loop, so use of `seq` may not be possible.

The most straightforward way to accomplish this is to simply append to a sequence in a loop, as follows:

```
vals := NULL:
for i to n do
   vals := vals,i:
end do:
```

This is also the worst way to accomplish this task, as every time a new element is appended to `vals`, a new sequence that is one greater in length is constructed, the old values are copied, the new value is placed in the last element, and the old sequence is marked as garbage. For n values, the expense of this operation is $\mathrm{O}(n^2)$ in both time and memory, but allocated memory will look a bit better as old sequences from prior iterations will be garbage collected as the loop is run.

The next method is most useful when the size of the output is known exactly, and that is to declare an `Array` for storage of the results, and convert to a sequence at the end:

```
vals := Array(1..n):
for i to n do
   vals[i] := i:
end do:
vals := seq(vals[i],i=1..n):
```

The expense of this approach, in time and memory, is $\mathrm{O}(n)$.

The final method is most useful when the size of the output is not known, and that is to declare a Maple `table` for storage of the results, and convert the results to sequence at the end:

```
vals := table():
for i to n do
   vals[i] := i:
end do:
vals := seq(vals[i],i=1..n):
```

The expense of this approach is also $\mathrm{O}(n)$, but the overhead is higher, as the `table` data structure is more flexible.

Table 1 lists time (c.p.u. seconds) and memory usage (Mb allocated) for the above three approaches as a function of the number of elements in the sequence:

So we can see that the advantages of the `Array` and `table` forms of the loop over direct construction of the sequence are significant, as for 50000 elements, the `table` form requires 0.4% of the time and 4% of the memory. We also note that the additional overhead from using the `table` form instead of the `Array` form is moderate, so this is surely a reasonable way to construct a sequence of unknown length.

3.2 Substitution

This next pitfall is not so well known as the first, and involves a mechanism for efficiently performing substitutions into arrays of expressions. This mechanism is heavily used when applying

Number of	Sequence		Array		Table	
Elements	Time	Memory	Time	Memory	Time	Memory
100	0.000	0.00	0.000	0.00	0.000	0.00
200	0.004	0.13	0.000	0.00	0.000	0.00
500	0.004	1.05	0.000	0.00	0.000	0.00
1000	0.016	4.06	0.004	0.13	0.004	0.13
2000	0.048	8.65	0.004	0.26	0.008	0.26
5000	0.332	10.75	0.012	0.66	0.012	0.79
10000	1.264	17.30	0.016	1.18	0.016	1.44
20000	5.036	45.60	0.032	2.23	0.048	2.88
50000	31.522	145.73	0.088	5.90	0.128	5.77

Table 1: Sequence v.s. Array v.s. Table loop construction

optimizations to a sequence of statements. For example, if one added the relation $t_{123} = x^2y + z$ as part of an optimization, then one would want to replace all subsequent occurrences of the subexpression $x^2y + z$ by t_{123}. This operation is so frequent, that it is necessary that it be fast and have a small footprint.

There are two main mechanisms for accomplishing this in Maple, and these are `eval` and `subs`. The *correct* choice here is `subs` because the replacement is structural rather than mathematical (`eval` is designed to mathematically evaluate the object after the replacements are made, which can add to the time).

There are a couple of problems with `subs` relative to code optimization. For efficiency reasons, uniquification is turned **off** while the subs operation is in effect, and this means that when the expressions one is dealing with are large, Maple does not leverage common subexpressions that are formed during the substitution. This can use a great deal of memory when dealing with large or highly nested statement sequences. Another issue is that `subs` seems to be more finely tuned to substitution into lists, and less so for substitution into in-place structures (such as Maple `Arrays`), and further, in our code optimization implementation it is often known in advance that a particular variable or subexpression occurs only in a certain range of entries in the array, and there is no way to efficiently restrict the substitution to that range.

The approach used here is to re-implement `subs`, in a compiled routine, in such a way that common subexpression uniquification is active, that efficiently handles in-place data structures, and allows for a substitution range (target range) to be specified. There are issues with this, in particular when running in compiled routines, garbage collection never activates, so the routine has been designed so that the memory consumption is monitored, and after memory consumption of approximately 10000 words, the compiled routine exits back to Maple, allowing a garbage collection to occur, then re-enters the compiled routine resuming where it left off. This is a trade-off in that substitution may require more time, but will consume less memory.

In order to demonstrate the issue, the following Maple code was used to generate n random polynomials in $x_1, ..., x_{10}$:

```
vars := [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10]:
rnd := rand(-1..1):
pols := Array(1..n):
for i to n do
   pols[i] := add(rnd()*i+add(rnd()*i*j,j=vars),i=vars):
end do:
```

Table 2 displays the time and memory usage of `eval`, `subs`, and the new `subs` to substitute

$x_1 = y_1$ into the array of polynomials. Included also is the Maple `length` of each polynomial list, which is a measure of how much memory would be used to store the data without uniquification.

Number of	Length	eval		subs		new subs	
Polynomials	(Mwords)	Time	Memory	Time	Memory	Time	Memory
100	0.05	0.000	0.00	0.000	0.00	0.008	0.00
200	0.11	0.000	0.00	0.004	0.00	0.008	0.00
500	0.26	0.004	0.00	0.004	0.00	0.012	0.00
1000	0.52	0.008	0.00	0.016	0.00	0.020	0.00
2000	1.04	0.020	0.00	0.028	0.00	0.028	0.00
5000	2.61	0.048	0.00	0.064	0.00	0.056	0.00
10000	5.23	0.096	2.49	0.156	3.28	0.120	2.23
20000	10.46	0.252	10.35	0.304	12.97	0.244	3.28
50000	26.14	0.644	34.73	0.792	42.46	0.796	5.37

Table 2: eval, subs, and new subs comparison, single relation

Number	Length	eval		subs		new subs	
of Polys	(Mwords)	Time	Memory	Time	Memory	Time	Memory
100	0.05	0.000	0.00	0.000	0.00	0.008	0.00
200	0.11	0.004	0.00	0.004	0.00	0.008	0.00
500	0.26	0.008	0.00	0.012	0.00	0.012	0.00
1000	0.52	0.012	0.00	0.020	0.00	0.020	0.00
2000	1.04	0.028	0.00	0.040	0.00	0.028	0.00
5000	2.61	0.072	0.00	0.128	3.80	0.060	0.00
10000	5.23	0.148	2.62	0.248	14.28	0.144	1.31
20000	10.46	0.380	10.35	0.516	35.12	0.312	2.23
50000	26.14	0.972	34.73	1.320	97.76	0.960	4.46

Table 3: eval, subs, and new subs comparison, multiple relations

The results in Table 3 are for a complete replacement of all variables in the input, i.e. $x_1 = y_1, x_2 = y_2, ..., x_{10} = y_{10}$. Note that for many of the smaller problems the memory usage is zero, but this is simply because Maple allocates memory in batches, and if the memory required by the operation is small enough to fit into the unused memory in the last batch, or into memory that has been garbage collected and made available again for use by Maple, the memory usage will register as zero.

The more critical observations can be made for the larger problems. In Table 2, for a single substitution, the new `subs` routine is a hair slower than `subs`, but the more important factor here is the memory usage, where the new routine uses $< 13\%$ of the memory for the largest problem. In Table 3 the new subs routine is faster and uses $< 5\%$ of the memory for the largest problem. The fact that only a fraction of the memory is used is critical for optimization of large problems.

We mentioned earlier that `subs` was preferred over `eval` for this type of operation, but we can see from the timings that the performance of `eval` surpasses that of `subs`, even though it has the additional evaluation to perform after the substitution is performed. This can be explained by the fact that the evaluation process does trigger some common subexpression uniquification, so less garbage is generated, and less time is spent collecting it, so the timing is better as well.

3.3 Location

The final mechanism to be discussed in detail in this section is one where we need to find the first occurrence of any of a set of subexpressions in an array of expressions. This is not a difficult task, but is one that will be required with high frequency (for example, when adding in an optimization $t_1 = a^2 + b^2$, one needs to locate the first occurrence of $a^2 + b^2$ both for substitution into the list, and for inserting the new statement into the optimized statement sequence).

For this task, there is no direct mechanism in Maple at this time. The best one can do is to loop through the elements of the array querying whether the current element `has` any subexpressions in the set. This initially appears to be purely a time issue.

This prompted the implementation of a `haswhere` compiled routine for locating the first occurrence of any in a set of subexpressions in an array of expressions.

The test problem is one where we perform 100 searches for a name in an array of univariate polynomials $1..n$ in the variables $x_1..x_n$ respectively, and for each search we randomly (uniform) choose one of the variables from $x_1..x_n$.

The Maple routine is simply:

```
maple_haswhere := proc(xprs,v,n) local i;
   for i to n do if has(v[i],xprs) then return i; end if; end do;
   return 0;
end proc:
```

Number	Maple loop		compiled haswhere	
of Polys	Time	Memory	Time	Memory
100	0.016	0.00	0.000	0.00
200	0.026	0.13	0.000	0.00
500	0.092	0.39	0.008	0.00
1000	0.204	0.79	0.012	0.00
2000	0.460	2.10	0.052	0.00
5000	1.360	7.73	0.168	0.00
10000	2.552	9.44	0.352	0.00
20000	4.992	8.52	0.704	0.00
50000	12.784	11.53	1.700	0.00

Table 4: subexpression location comparison

From Table 4 the expense of the searches is roughly linear with respect to the list being searched, and the compiled implementation is significantly faster (a factor of ≈ 7.5 times) than the Maple loop. What was unexpected was the memory usage in the Maple loop. This can be attributed to the fact that Maple is an interpreted environment, and that execution of routines in an interpreted environment allocates memory, but this is yet another win for the `haswhere` implementation.

3.4 Other Aspects

Several other tasks in the implementation were re-coded as compiled routines, and a couple of the more relevant tasks are described below:

Initial Subexpression Extraction

This is required as the first step of common subexpression detection, and will, for example, replace all occurences of the computation of $sin(x)$ by t_1, adding the statement $t_1 = sin(x)$ before the first occurence.

This is an area where Maple's uniquification mechanism does most of the work for us, and all we need to do is perform a recursive walk of the input statement sequence searching for common subexpressions. This is actually quite easy, as uniquification assures us that if two expressions are the same, they will have the same memory address, so we need only perform an address comparison.

The expressions in the statement sequence are walked recursively from bottom up, with replacements being made in-line, and new statements added as needed. This was re-implemented in a compiled routine, and the win (for a large problem) was approximately a factor of 7 for time, and 2 for memory, but to accomplish the latter, the routine exits back to Maple once a certain amount of memory was consumed (to allow for garbage collection), then is called again resuming where it left off.

Map Build/Substitution Reversal

Ideally an optimizer should have the ability to re-use temporary variables that are artifacts of the optimization process, and to reverse any optimizations that do not actually provide a benefit. The former provides efficiency both for Maple code (as there is some expense associated with each independent local variable in a procedure), as well as Maple to C code conversion (as each variable requires type detection), and the latter provides for clarity and compactness of the output code.

In order to accomplish this, one needs to know the duration and frequency of usage for all the temporary variables in the optimized code.

In order to make this process efficiently implementable, at the end of the optimization process, a map is constructed that provides the location of the origin, the frequency, and the first and last usage of all temporary variables in the optimized code.

This map is constructed in a compiled routine, then with this information a native Maple routine is used to perform the final simplification prior to output. Before implementing this approach, this process consumed more time than the optimization itself when running purely in native Maple.

4 Code Generation and Compilation

Since Maple 10 the `Compiler` package has been available for converting Maple code into C code, compiling that code, and providing a Maple interface to access the compiled code.

Aside from the fact that the `ToExternal` package pre-dates the `Compiler` package, there are still several compelling reasons why the code generation and compilation provided by `ToExternal` is preferred over that of the `Compiler` package.

1. The implementation needs to be able to generate both a Maple interface, and an external interface (for direct call from compiled code) for a code, and do so efficiently for integer and float datatypes, and vectors, matrices, and arrays of these as well. This is difficult for matrices and arrays, as the data storage ordering becomes an issue.

2. The implementation needs to have support for local array declarations (heavily in use by numerical ODE solution in Maple for dense ODE systems). The `Compiler` package does not currently support this.

3. The implementation needs to be able to handle large problems as efficiently as possible, i.e. it needs to be scalable. This is particularly relevant for solutions using stiff ODE methods where an explicit Jacobian of the system is formed. A moderate ODE system in 20 first-order

variables will require computation of 400 values to compute the Jacobian, and the subsequent code optimization may increase this to 4000 statements (depending on the complexity of the problem).

Note that some experiments were performed using the `Compiler` package and it was found that for moderate to large problems the `ToExternal` package provided far better scalability, and had running times either better than, or on par with the time required to numerically compute the solution, while the `Compiler` package generally took significantly longer. On the flip side of this, it should be explicitly mentioned that the `Compiler` package was designed to handle a far larger class of functions than `ToExternal`, and that class is growing with each release of Maple as the `Compiler` package matures, so it is to be expected that there is some cost associated with this generality. In addition, the `Compiler` package is much easier to use than `ToExternal`, which is why the latter is primarily designed as an under-the-hood tool to be used by Maple's numerical DE solvers.

We should note that it will likely never be possible to convert **all** numerical Maple procedures into compiled code, as some of these may make use of global variables (which a compiled code will not necessarily have access to), or utilize Maple-specific constructs within the code (as an example, consider a piece of code that needs to form a polynomial, and extract a particular coefficient from that polynomial), so any Maple to C code conversion utility will naturally have restrictions and limitations. This is one reason it makes sense to specialize to the class of procedures of interest.

4.1 Implementation Aspects

The `ToExternal` package has been developed with the primary goal that it be fast. The secondary goal is coverage, but so long as the class of problems typically encountered in numerical DEs is covered, this has been given a lower priority. Ease-of-use has the lowest priority, as use of the package requires information that is readily available for numerical DE applications, but otherwise unknown for general procedures.

As speed is an overriding concern, a strong attempt has been made to reduce the number of passes through the code. The core processing steps are as follows:

1. Conversion to inert representation
2. Weak type detection
3. Implicit return replacement
4. Strong type detection
5. C preprocessing
6. C code and wrapper generation
7. Compile and return

In step 1, the Maple procedure is converted to an inert intermediate representation. This representation closely resembles the that obtained from Maple's `codegen[maple2intrep]` routine, where all operations are represented as a function (for example, an assignment is represented by `Assign(target,result)`). In this pass we also collect prototypes for the function itself, and any functions it calls, as well as validating and assigning types to any variables with a declared type on input. This step also creates a Maple table to store any collected data for the function.

In step 2, weak type detection is applied, which means that we determine the types for variables where there is no other option for the type. For example, if a local variable x was assigned the value 3.2, then we know that x must be of float type.

Step 3 can be implemented very efficiently, and is simply the process of making explicit return values that are implicit in the code. For example, a Maple procedure to compute the value of x^2 from x can be written as either

```
f := proc(x) return(x^2); end proc;
```

or

```
f := proc(x) x^2; end proc;
```

where the latter is an implicit return, and the former explicit.

In step 4, strong type detection is applied, and by the end of this step all variables used in the procedure will have a specified type.

Step 5 handles any other miscellaneous preprocessing conversions, such as determining bounds for local arrays, and conversion of piecewise functions to if-then-else constructs.

In step 6, the C code and wrappers are generated (into a string or file), in preparation for compilation. Note that a *wrapper* is simply a piece of interfacing code that performs conversions between Maple data and the data in a form required by the C code, and is needed for calling the code from within Maple.

And finally in step 7 the generated code is compiled into a *call_external* procedure in Maple that can either be used directly in Maple, or can be used to obtain the address of the direct C interface for use by compiled code.

4.2 Efficiency, Efficiency, Efficiency ...

There are many factors that provide a significant hurdle in implementing these routines efficiently. The most difficult is the type detection process. Since Maple is an untyped language, variables in a procedure could be used as integers, floats, or arrays of these with no easy way to distinguish the cases.

We ignore cases where a variable is re-used as an incompatible type with a prior usage (e.g. x used as a float, then later assigned as an array), as these are out of scope for the `ToExternal` package. This class of variable must be detected though (and an error thrown), to prevent an incorrect procedure from being constructed. Also for many procedures, it is not readily apparent into which class a particular variable falls until perhaps its 10th usage, and in some cases it remains unknown after scanning the procedure. This means that each variable instance must be examined for usage (when detecting the type) or consistency (when verifying the type) so this is a significant expense.

After many attempts, and several variations of implementation, it was found that working with a Maple table data structure, that stores type-specific information keyed on the variable names, was the best way to go.

Another aspect of the code generation that was difficult to optimize was the process of generating the C procedure as a string in Maple. If one constructs a string and constantly appends to it, then memory usage will be proportional to the square of the final output size (which can be problematic for a large procedure). If instead one constructs a sequence of strings, then as one appends to the sequence, memory usage will be proportional to the square of the number of strings in use (see §3.1). The current implementation balances between the creation of large strings and sequences of small strings, appending the current completed snippet of code into a Maple table, and performing only one large string concatenation once the code is complete, making the memory usage close to twice the memory required to store the procedure as a string.

As a final aspect of efficiency, Maple's numerical DE solvers are written as compiled routines, so rather than having the solvers call back into Maple, which then calls out to the compiled procedures (thus incurring two conversions to make the call, and possibly two for processing the return value), the solvers are designed so that when `ToExternal` is in use, the compiled solvers can directly call the newly compiled code bypassing the conversions entirely.

5 Examples

In this section we present several examples of our implementations for code optimization and generation, comparing with previous Maple facilities.

5.1 Optimization Example #1: Relativistic system

The first example for code optimization involves a system of four first-order differential equations that describe a relativistic trajectory. The equations are rather large, and so are not provided here, but to give an idea of their size, printing of these equations from a Maple worksheet using a 12-point font covers a little over 21 pages.

	Unoptimized	codegen 1	codegen 2	new optimizer
Additions	486	98	80	78
Multiplications	834	175	124	125
Negations	2	31	24	20
Divisions	0	6	4	0
Exponentiations	288	8	6	5
Function Calls	0	0	2	0
Assignments	4	76	67	54
Time	N/A	0.056	1.308	0.080

Table 5: optimization comparison: relativity problem

In Table 5 there are two result columns for `codegen`, the first is a direct call to `optimize`, while the second also specifies the `tryhard` option to attempt to obtain better optimization via application of more expensive algorithms.

We see that in all three cases, a significant improvement is obtained in optimization of the original problem. The total number of additions, multiplications and exponentiations has dropped by a factor of better than 5 times. The important point here is that the new optimizer has obtained an optimization that is better than `codegen` with `tryhard` in a time that is more comparable with `codegen` without the `tryhard` option.

5.2 Optimization Example #2: Large statement sequence

This example is primarily about scalability. A colleague, Dr. Chad Schmitke, provided a statement sequence of length slightly over 50000 that he was trying to optimize. This was a sub-problem from a vehicle model of a large 6 wheeled truck.

This statement sequence is of such size and complexity, that reading it into Maple consumes 24 Mb memory, and storing it in non-uniquified form requires approximately 700 Mb memory. Attempts to optimize this statement sequence with `codegen[optimize]` took a fair amount of time and memory, and gave a moderate degree of optimization. Initial attempts to optimize it with an earlier version of the new optimizer were unsuccessful (the subs uniquification issue from §3.2 was the primary problem) until the implementation of the strategies outlined in §3. The attempt to optimize the statements with `codegen[optimize]` using the `tryhard` option was abandoned once memory usage climbed well above the physical memory on the computer (2 Gb). The performance is described in Table 6.

We note that the statement count actually dropped for `codegen[optimize]`, but this is because some statements were removed by the optimization cleanup step. It is a mandate of the new optimizer not to remove existing statements (as they may still be relevant), so this is not done,

	Unoptimized	codegen 1	codegen 2	new optimizer
Additions	8211865	71473	N/A	65137
Multiplications	20154711	73577	N/A	65620
Negations	4833318	62155	N/A	58858
Divisions	3894	104	N/A	98
Exponentiations	2234	10	N/A	10
Function Calls	24519413	156	N/A	156
Assignments	52856	16606	N/A	59890
Time (cpu sec.)	N/A	415.31	> 8600	113.13
Memory (Mb)	N/A	101.17	> 2500	62.12

Table 6: optimization comparison: large statement sequence

but the authors *are* considering the implementation of this feature as an option. Other than that difference, we note that the new optimizer completed in almost a quarter of the time using a little more than half of the memory when compared with `codegen[optimize]`, and at the same time obtained a better optimization.

5.3 Compile Example #1: The 2-D N-dulum DAE system

The *N-dulum* is just a short name for the N-weight pendulum problem, where the N weights are attached in a chain by strings of equal length. The dynamic equations for this model in Cartesian coordinates are small in size, but the problem is a DAE problem, as the constraints for the length of the strings are algebraic, and the DAE formulation is stiff. This means we need to construct a Jacobian evaluation procedure, along with several other procedures for computation of the residual of the algebraic constraints and their Jacobians as well.

Table 7 contains timings for the construction of the DAE solution procedure in Maple (which includes compile time for the compiler), and the run time for integration of the problem with initial conditions placing all weights on a horizontal line to the right of the attachment point, and integrating t from 0 to 10.

Number of	Without compile		With compile	
weights	Construction	Integration	Construction	Integration
5	0.232	11.232	0.384	0.140
10	0.584	40.806	0.820	0.664
15	1.084	91.341	1.364	1.760
20	1.748	114.455	2.056	2.804
30	3.796	325.708	3.952	10.832
40	6.344	595.141	6.396	26.206
50	11.764	621.015	11.496	35.238

Table 7: N-dulum DAE timing comparison

So we see from the results that the expense for compiling the procedure is minimal, maxing out at about 0.27s for the hardest problem, but the benefits for the run time are significant, providing an improvement factor of $17 - 80$ times over the native procedures. It should be noted that the procedures that are not compiled are running under Maple's `evalhf` interpreter, which runs at least an order of magnitude faster than standard interpreted code, so the performance of the generated

procedures is quite significant.

5.4 Compile Example #2: Singular perturbation BVP

This problem is a simple singular perturbation boundary value problem with boundary layers at both boundaries. The width of the boundary layer is controlled by the parameter ϵ in the problem, and is proportional to $\sqrt{\epsilon}$.

The statement of the problem is:

$$\epsilon y'' - y = 0, y(0) = 1, y(1) = 1$$

Table 8 measures the time taken for different values of ϵ, smaller values corresponding to more difficult problems.

epsilon	Without compile	With compile
1e-3	0.056	0.080
1e-4	0.060	0.084
1e-5	0.068	0.112
1e-6	0.136	0.132
1e-7	0.588	0.480
1e-8	1.796	1.396
1e-9	14.904	13.480
1e-10	73.408	67.940

Table 8: singular perturbation BVP timing comparison

We see that for the easier problems, the run-time with the compiled procedure is less than stellar when compared with the run-time of the native procedure. In fact, the time difference between the two timings is the time required to generate and compile the procedure. Once the difficulty increases past a certain point, the performance of the compiled procedure becomes apparent, and for ϵ values $1e-6$ or smaller, the compiled procedure out-performs the native one. This is fairly easily explained, as for the easier problem, the majority of the time is spent performing linear algebraic operations on a linearized discretization of the ODE over the integration region. When the problem becomes more difficult, a significant number of additional evaluations are required, and the expense of evaluating the discretized ODE becomes more significant.

6 Conclusion

In summary, we have seen that it is possible to implement effective optimization and compiled-speed numerical ODE solution within Maple. Also that development within an interpreted PSE has a few pitfalls, but several benefits as well. In the future we hope to extend these facilities to direct use by more components of Maple, such as 3-d plotting, and numerical integration.

Acknowledgment

The author wishes to thank the referees for their careful reading of the manuscript and their helpful comments and suggestions.

References

[1] E.V. Zima, *Numeric Code Optimization in Computer Algebra Systems and Recurrent Relations Technique*. Proc. ISSAC 1993, pp. 42-46.

[2] K.O. Geddes, G.J. Fee, *Hybrid symbolic-numeric integration in MAPLE*. Proc. ISSAC 1992, pp. 36-41.

[3] M.B. Monagan, K.O. Geddes., K.M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, P. DeMarco, *Maple Advanced Programming Guide (Maple 11)*. Maplesoft, 2007.