MathCode: A System for C++ or Fortran Code Generation from Mathematica

Peter Fritzson
Vadim Engelson
Krishnamurthy Sheshadri

***MathCode* is a package that translates a subset of *Mathematica* into a compiled language like Fortran or C++. The chief goal of the design of *MathCode* is to add extra performance and portability to the symbolic prototyping capabilities offered by *Mathematica*. This article discusses several important features of *MathCode*, such as adding type declarations, examples of functions that can be translated, ways to extend the compilable subset, and generating a stand-alone executable, and presents a few application examples.**

■ Introduction

MathCode is a *Mathematica* add-on that translates a *Mathematica* program into C++ or Fortran 90. The subset of *Mathematica* that *MathCode* is able to translate involves purely numerical operations, and no symbolic operations. In the following sections we provide a variety of examples that show precisely what we mean. The code that is generated can be called and run from within *Mathematica*, as if you were running a *Mathematica* function.

There are two important purposes that are served by *MathCode*. Firstly, the C++/Fortran 90 code runs faster, typically by a factor of about a few hundreds (or about 50 to 100) over interpreted (compiled) *Mathematica* code, resulting in considerable performance gains, while still requiring hardly any knowledge of C++/Fortran 90 on the part of the user. Secondly, the generated code can also be executed as a stand-alone program outside *Mathematica*, offering a portability otherwise not possible. You should note, however, that these advantages come at some loss of generality since integer and floating point overflow are not trapped and switched to arbitrary precision as in standard *Mathematica* code. Here the user is responsible for ensuring an appropriate choice of scaling and input data to

avoid such problems. The measurements in this article were made using *Mathematica* 6.

There are situations in which having a system such as *MathCode* can be particularly helpful and effective, like when a certain calculation involves a symbolic phase followed by a numerical one. In such a hybrid situation, *Mathematica* can be employed for the symbolic part to give a set of expressions involving only numerical operations that can be made part of a *Mathematica* function, which can then be translated into C++/Fortran 90 using *MathCode*.

In this article, we describe some of the more important features of *MathCode*. For a more detailed discussion the reader is referred to [1]. For brevity, we simply say C++ when we actually mean C++ or Fortran 90: *MathCode* can generate code in both C++ and Fortran, although we illustrate C++ code generation in this article.

In Section 2, we show how to quickly get started with *MathCode* using a simple example of a function to add integers.

Section 3 presents many useful features of *MathCode*. In Section 3.1, we discuss the way the system works, the various auxiliary files generated and what to make of them, and how to build C++ code and install the executable. We then compare the execution times of the interpreted *Mathematica* code and the compiled C++ code. This section also illustrates how *MathCode* works with packages.

Section 3.2 briefly makes a few points about types and type declarations in *MathCode*. There are two ways to declare argument types and return types of a function mentioned in this section.

In Section 3.3, we show how to generate a stand-alone C++ executable. This executable can be run outside of *Mathematica*. We illustrate how to design a suitable main program that the executable runs.

It should be emphasized that *MathCode* can generate C++ for only that subset of *Mathematica* functions referred to as the *compilable subset*. Section 3.4 gives a sample of this subset, while Section 3.5 presents three ways to extend it with the already-available features of *MathCode*: Sections 3.5.1 through 3.5.3 discuss, respectively, symbolic expansion of function bodies, callbacks to *Mathematica*, and handling external functions. Each of these extensions has its own strengths and limitations.

Section 3.6 discusses common subexpression elimination, a feature that is aimed at enhancing the efficiency of generated code.

Section 3.7 presents some shortcuts available in *MathCode* to extract and make assignments to elements of matrices and submatrices, while Section 3.8 is about array declarations.

In Section 4, we present several examples of effectively using *MathCode*. Section 4.1 provides a summary of the examples.

Section 4.2 discusses an essentially simple example, that of computing the function $\sin(x + y)$ over a grid in the x-y plane, but done in a somewhat roundabout manner so as to illustrate various features of *MathCode*.

Section 4.3 discusses an implementation of the Gaussian elimination algorithm [2] to solve matrix systems of the type $A.X = B$, where A is a square matrix of size n and X (the solution vector) and B are vectors of size n. In this section, we make a detailed performance study by computing the solution of a matrix

system by turning on a few compilation options available in *MathCode*, and also make comparisons with `LinearSolve`.

In Section 4.4, we show how to call external libraries and object files from a C++ program that is automatically generated by *MathCode*. We take the example of a well-known matrix library called SuperLU [3], and demonstrate how to solve, using one of its object modules, a sparse matrix system arising from a partial differential equation.

The *MathCode* User Guide that is available online discusses more advanced aspects, like a detailed account of types and declarations, the numerous options available in *MathCode* with the aid of which the user can control code generation and compilation, and other features. We refer interested readers to [1].

In Section 5, we summarize the salient aspects of *MathCode* and discuss the kinds of applications for which *MathCode* is particularly useful. We conclude the article with a brief summary of various points made. The first version of *MathCode*, released in 1998, was partly developed from the code generator in the ObjectMath environment [4, 5]. The current version is almost completely rewritten and very much improved.

■ 2. Getting Started with *MathCode*

□ 2.1. An Example Function

In this section we take the reader on a quick tour of *MathCode* using the simple example of a function to add integers.

The following command loads *MathCode*.

```
In[1]:= Needs["MathCode`"]
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

MathCode works by generating a set of files in the current directory (see Section 3.1). We can set the directory in the standard way as follows; here, `$MCRoot` is the *MathCode* root directory. The user can, however, use any other directory to store the files.

```
In[2]:= SetDirectory[$MCRoot <> "/Demos/SimplestExample"];
```

Let us now define a *Mathematica* function `sumint` to add the first n natural numbers.

```
In[3]:= sumint[n_] :=
          Module[{res = 0, i}, For[i = 1, i ≤ n, i++, res = res + i]; res]
```

Note that the body of this function has purely numerical operations, like incrementing the loop index i, adding two numbers, and assigning the result to a variable.

□ 2.2. Declaration of Types

We must now declare the data types of the parameter *n* and the local variables *res* and *i*; we must also specify the return type of the function. We do this using the function `Declare` that *MathCode* provides.

```
In[4]:= Declare[sumint[Integer n_] → Integer, {Integer, Integer}];
```

Note that `Integer n_` *does not* mean `Integer*n_`; the function `Declare` creates an environment in which this is interpreted as a type declaration, that is, an integer variable *n* is being declared in the example. The type `Integer` is translated to a native C int type, and the type `Real` to a native C double type.

□ 2.3. C++ Code

To generate and compile the C++ code, we execute the following command.

```
In[5]:= BuildCode["Global`"];
```

```
Successful compilation to C++: 1 function(s)
```

Since we have not specified the context of `sumint`, its default context is `Global`. We could, therefore, have simply executed the following command instead.

```
In[6]:= BuildCode[];
```

```
Successful compilation to C++: 1 function(s)
```

With the following command, we seamlessly integrate an external program with *Mathematica*.

```
In[7]:= InstallCode[];
```

```
Global is installed.
```

We can now run the external program in the same way that we would execute a *Mathematica* command.

```
In[8]:= sumint[1000]

Out[8]= 500 500
```

If we want to run the *Mathematica* code (and not the generated C++ code) for `sumint`, we must first uninstall the C++ executable.

```
In[9]:= UninstallCode[];
```

Now the *Mathematica* code for `sumint` will run.

```
In[10]:= sumint[1000]

Out[10]= 500 500
```

■ 3. A Tour of *MathCode*

□ 3.1. How the *MathCode* System Works

MathCode works by generating a set of files in the home directory. In the example of `sumint`, the default context is `Global` and the files generated by *MathCode* are: Global.cc (the C++ source file), Global.h and Global.mh (the header files), Globaltm.c, Global.tm and Globalif.cc (the *MathLink*®-related files that enable transparently calling C++ versions of the function `sumint` from *Mathematica*), and Globalmain.cc, which contains the function main( ) needed when building a stand-alone executable.

We can also create a package (let us call it `foo`) that defines its own context `foo` instead of the default context `Global`. See Figure 1 for a block diagram of the way the overall system works. The *MathCode* code generator translates the *Mathematica* package to a corresponding C++ source file foo.cc. Additional files are automatically generated: the header file foo.h, the *MathCode* header file foo.mh, the *MathLink*-related files footm.c, foo.tm, foo.icc, and fooif.cc, which enable calling the C++ versions from *Mathematica*, and foomain.cc, which contains the function main that is needed when building a stand-alone executable for `foo` (see Section 3.3). The generated file foo.cc created from the package `foo`, the header file foo.h, and additional files are compiled and linked into two executables. In the case of *MathCode* F90, Fortran 90 is generated and a file foo.f90 is created. No header file is generated in that case since Fortran 90 provides directives for the use of module. External numerical libraries may be included in the linking process by specifying their inclusion (Sections 3.5.3 and 4.5). The executable produced, foo.exe, can be used for stand-alone execution, whereas fooml.exe is used when calling on the compiled C++ functions from *Mathematica* via *MathLink*.

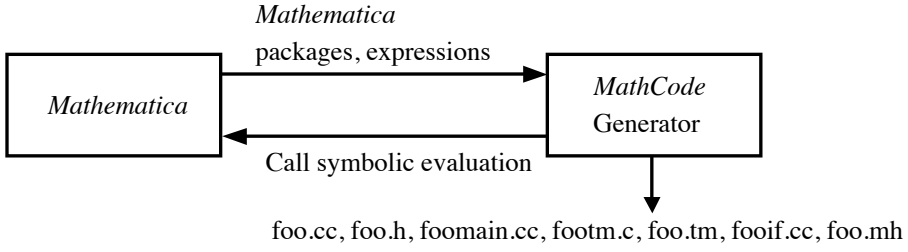


Figure 1. Generating C++ code with *MathCode* for a package called `foo`.

Let us see how to work with a package again using the same `sumint` example.

```
In[1]:= Needs["MathCode`"];

        MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode

In[2]:= SetDirectory[$MCRoot <> "/Demos/SimplestExample"];
```

If we are compiling the package `foo` using *MathCode*, we also need to mention `MathCodeContexts` within the path of the package.

```
In[3]:= BeginPackage["foo`", {MathCodeContexts}];
```

We define the function `sumint`

```
In[4]:= sumint[n_] :=
          Module[{res = 0, i}, For[i = 1, i ≤ n, i++, res = res + i]; res];
```

and close the context `foo`.

```
In[5]:= EndPackage[];
```

We next declare the types, and then build and install as before.

```
In[6]:= Declare[sumint[Integer x_] → Integer, {Integer, Integer}];
```

```
In[7]:= BuildCode["foo`"];
```

```
Successful compilation to C++: 1 function(s)
```

Again, since the package `foo` has been defined, it is the default context, and so we could simply have executed the following command.

```
In[8]:= BuildCode[];
```

```
Successful compilation to C++: 1 function(s)
```

To run the executable from the notebook, we must install it.

```
In[9]:= InstallCode[];
```

```
foo is installed.
```

Now the following command runs the C++ executable fooml.exe. The call to `sumint` via *MathLink* is executed 1000 times. The timing measurement includes *MathLink* overhead, which typically for small functions is much more than the execution time for the compiled function. This can be avoided if the loop is executed within the external function itself, as in the example in Section 4.2.5.

```
In[10]:= Timing[Do[res = sumint[1000], {1000}]; res]
Out[10]= {1.392, 500 500}
```

Here is the C++ code that was generated.

```
In[11]:= FilePrint["foo.cc"]
```

```
#include "foo.h"

#include "foo.icc"

#include <math.h>
void foo_TfooInit ()
{
;
}

int foo_Tsumint ( const int &n)
{
  int res = 0;
  int i;
  i = 1;
  while (i <= n)
  {
    res = res+i;
    i = i+1;
  }
  return res;
}
```

Note that the function `sumint` appears as foo_Tsumint in the generated code. This is because the full name of the function is in fact `foo`sumint`, and *MathCode* replaces the backquote "`" by "_T" in the C++ code.

To run the *Mathematica* function (and not its C++ equivalent) `sumint`, we must use the following command to uninstall the C++ code.

```
In[12]:= UninstallCode[];
```

Now it is the *Mathematica* code that runs when you execute `sumint`.

```
In[13]:= Timing[Do[res = sumint[1000], {1000}]; res]

Out[13]= {22.161, 500 500}
```

You can see that the C++ executable together with the *MathLink* overhead runs about 15 times faster than the *Mathematica* code. The factor by which the performance is enhanced is problem dependent, however. The performance of the *Mathematica* code could also have been improved by using the built-in `Compile` function. In Section 4 we will see many more examples, some quite involved, where we get a range of performance enhancements, also including usage of the `Compile` function.

We clean up the current directory by removing the files automatically generated by *MathCode*.

```
In[14]:= CleanMathCodeFiles[Confirm → False, CleanAllBut → {}];
```

□ 3.2. Types and Declarations

To be able to generate efficient code, the types of function arguments and return values must be specified, as we have seen in the preceding examples. The basic types used by *MathCode* are

```
{Real, Integer, Null}
```

Arrays (vectors and matrices) of these types can also be declared.

```
{Real[5], Real[3, 4], Real[_], Integer[m, n], Integer[2, 3, n_]}
```

Type declarations can be given in two different ways:

- Directly in the function definition

```
f[Real x_] → Real := x^2
```

- In a separate command

```
g[x_] := Sin [x]

Declare[g[Real x_] → Real]
```

The latter construction can be useful if you want to separate already existing *Mathematica* code with the type information needed to be able to generate C++ code using *MathCode*.

□ 3.3. Generating a Stand-Alone Program

So far we have only seen examples in which the installed C++ code can be run within *Mathematica*. However, we can also produce a stand-alone executable. This offers a degree of portability that can be useful in practice.

To illustrate, we take the same example function `sumint` that we discussed in the previous sections. The sequence of commands is very much as in the previous section, except for the option `StandAloneExecutable→True` for the *MathCode* function `MakeBinary`, and an appropriate option `MainFileAndFunction` for the function `SetCompilationOptions` immediately after `BeginPackage`. Figure 2 illustrates the process of building the two kinds of executable, namely fooml.exe and foo.exe (on some systems foomain.exe) from a package called `foo`.

```
In[15]:= Needs["MathCode`"];

In[16]:= SetDirectory[$MCRoot <> "/Demos/SimplestExample"];

In[17]:= BeginPackage["foo`", {MathCodeContexts}];
```

The option `MainFileAndFunction` is used to specify the main file. The functions defined in *Mathematica* must have the prefix Global_T (*packagename*_T in general) to be recognized in the main file.

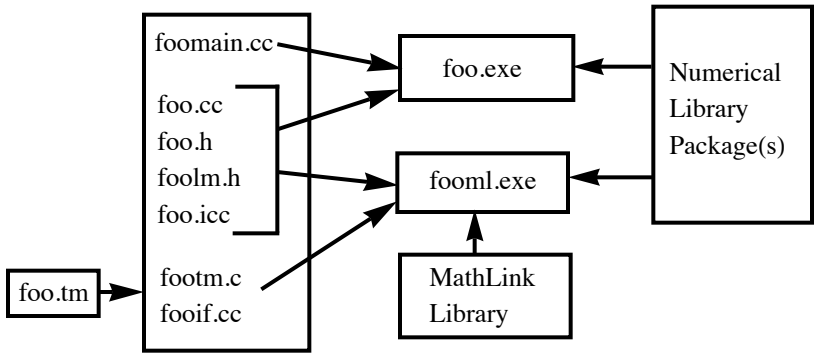


Figure 2. Building two executables from the package `foo`, possibly including numerical libraries.

```
In[18]:= SetCompilationOptions[
           MainFileAndFunction → "#include <stdio.h>\n int main()
         {int n;printf(\"give an integer:\");scanf(\"%d\",&n);printf(\"the
              sum is %d.\n\",foo_Tsumint(n));return 0;}
         "];

In[19]:= sumint[n_] :=
           Module[{res = 0, i}, For[i = 1, i ≤ n, i++, res = res + i]; res];

In[20]:= EndPackage[];

In[21]:= Declare[sumint[Integer x_] → Integer, {Integer, Integer}];
```

Now we are ready to generate and compile the C++ code for the package `foo`. We can do this in two ways: we can either employ the *MathCode* function `BuildCode`, as in the previous examples, or first execute `CompilePackage` (which generates the C++ source and header files) and then the function `MakeBinary` (which creates the executable).

```
In[22]:= CompilePackage["foo`"];

         Successful compilation to C++: 1 function(s)

In[23]:= MakeBinary["foo`", StandAloneExecutable → True];
```

The last command generates the stand-alone executable foo.exe that can be executed from a command line, or, alternatively, by using the *Mathematica* function `Run`.

```
In[24]:= Run["foo.exe"]

Out[24]= 0
```

If you desire, you can, in addition to the stand-alone executable foo.exe, also generate foom l.exe that can be run from within *Mathematica*, just like before.

```
In[25]:= MakeBinary["foo`"];
```

```
In[26]:= InstallCode[];

         foo is installed.
```

Now the following command runs the C++ program foo.cc.

```
In[27]:= sumint[1000]

Out[27]= 500 500
```

3.3.1. Generating a DLL

Here we briefly mention the possibility of generating a DLL, without giving a full example. To generate a DLL from a package, you have to write a file containing one simple wrapper function in order to make a generated function visible outside the DLL. You write a wrapper function for each generated function. The flags used are as follows:

```
CompilePackage[NeedsExternalObjectModule → "ext"];
MakeBinary[StandAloneExecutable → True, LinkerOptions → "/DLL"]
```

Here "ext.cpp" is a C++ file with wrapper functions, and "`/DLL`" is a flag for the Visual C++ linker. For other C++ compilers this procedure is not automatic and requires several operating system commands, but the wrapper functions are not needed.

□ 3.4. The Compilable Subset

MathCode generates C++ code for a subset of *Mathematica* functions, called the *compilable subset*. The following items give a sample of the compilable subset. For a complete list of *Mathematica* functions in the compilable subset, see [1].

- Statically typed functions, where the types of function arguments and return values are given by the types discussed in Section 3.2
- Scoping constructs: `BeginPackage[ ]`, `EndPackage[ ]`, `Module[ ]`, `Block[ ]`, `With[ ]`
- Procedural constructs: `For[ ]`, `While[ ]`, `If[ ]`, `Which[ ]`, `Do[ ]`
- Lists and tables: `List[ ]`, `Table[ ]`, `Array[ ]`, `Range[ ]`, `Identity Matrix[ ]`
- Size functions: `Dimensions[ ]`, `Length[ ]`
- Arithmetic and logical expressions, for example: `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `!`, `&&`, `||`, and so forth
- Elementary functions and some others, for example: `Sin[ ]`, `Exp[ ]`, `ArcSin[ ]`, `Sqrt[ ]`, `Round[ ]`, `Max[ ]`, `Cross[ ]`, `Transpose[ ]`, `Dot[ ]`
- Constants: `True`, `False`, `E`, `Pi`
- Assignments: `:=`, `=`

- Functional commands: `Map[ ]`, `Apply[ ]`
- Some special commands: `Sum[ ]`, `Product[ ]`

Functions not in the compilable subset can be used in external code by callbacks to *Mathematica* (see Section 3.5.2 for an example).

Examples of functions that are not a part of the compilable subset include: `Integrate[ ]`, `Solve[ ]`, `FindRoot[ ]`, `LinearSolve[ ]`, `Expand[ ]`, `Factor[ ]`.

These functions can be used if *Mathematica* can evaluate them at compile time to expressions that belong to the compilable subset. In general, *Mathematica* functions that perform symbolic operations are not in the compilable subset. Also, many functions in the subset are implemented with limitations, that is, more difficult cases are not always supported. However, *MathCode* currently provides several ways to extend the compilable subset, as we discuss in the next section.

□ 3.5 Ways to Extend the Compilable Subset

3.5.1. Symbolic Expansion of Function Bodies

Functions not entirely written using *Mathematica* code in the compilable subset, but whose definitions can be evaluated symbolically to expressions that belong to the compilable subset, can be handled by *MathCode*.

```
In[1]:= Needs["MathCode`"];
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= SetDirectory[$MCRoot <> "/Demos/SimplestExample"];

In[3]:= f[Real a_, Real b_] → Real := Integrate[x Sin[x], {x, a, b}]

In[4]:= f[1., 2.]

Out[4]= 1.44042

In[5]:= ? f
```

```
Global`f

f[a_, b_] := ∫_a^b x Sin[x] dx
```

Generate C++ code and compile it to an executable file.

```
In[6]:= BuildCode[EvaluateFunctions → {f}]
```

```
Successful compilation to C++: 1 function(s)
```

The option `EvaluateFunctions` tells *MathCode* to let *Mathematica* expand the function body as much as possible. Everything works fine because the result belongs to the compilable subset.

```
In[7]:= Integrate[x Sin[x], {x, a, b}]

Out[7]= a Cos[a] - b Cos[b] - Sin[a] + Sin[b]
```

The generated executable is connected to *Mathematica*:

```
In[8]:= InstallCode[];

        Global is installed.

In[9]:= f[1., 2.]

Out[9]= 1.44042
```

3.5.2. Callbacks to Mathematica

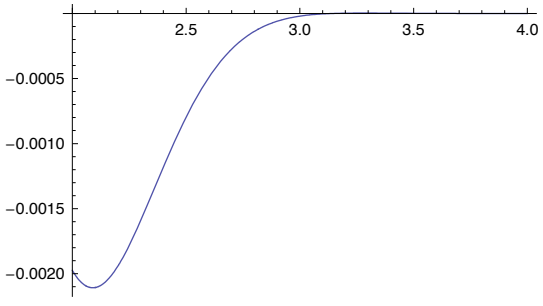
Consider the following function whose definition includes the `Zeta` function, which does not belong to the compilable subset.

```
In[1]:= Needs["MathCode`"]

        MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode

In[2]:= SetDirectory[$MCRoot <> "/Demos/SimplestExample"]

Out[2]= C:\MathCode\Demos\SimplestExample

In[3]:= f[x_] := Sin[x] Cos[x] / (1 + Tan[x]^2) E^(-x^2) Zeta[x]
```

Let us plot the function:

```
In[4]:= Plot[f[x], {x, 2, 4}, PlotRange → All]
```

Out[4]=



We now make the declarations:

```
In[5]:= Declare[f[Real x_] → Real]

In[6]:= Declare[Zeta[Real x_] → Real]
```

These declare statements do not change the way *Mathematica* computes the function.

```
In[7]:= {f[2.5], f[5/2]}

Out[7]= {-0.000796932, Cos[5/2] Sin[5/2] Zeta[5/2] / (E^(25/4) (1 + Tan[5/2]^2))}
```

Let us now generate C++ code and compile it to an executable file. The option `CallBackFunctions` tells *MathCode* which functions have to be evaluated by *Mathematica*. As a result, although the function `Zeta` is not in the compilable subset, an executable is still generated and communicates with the kernel to evaluate `Zeta`.

```
In[8]:= BuildCode[CallBackFunctions → {Zeta}]
```

```
Successful compilation to C++: 2 function(s)
```

The generated executable is connected to *Mathematica*:

```
In[9]:= InstallCode[];
```

```
Global is installed.
```

Now it is the external code that is used to compute the function:

```
In[10]:= {f[2.5], f[5/2]}
```

```
Out[10]= {-0.000796932, f[5/2]}
```

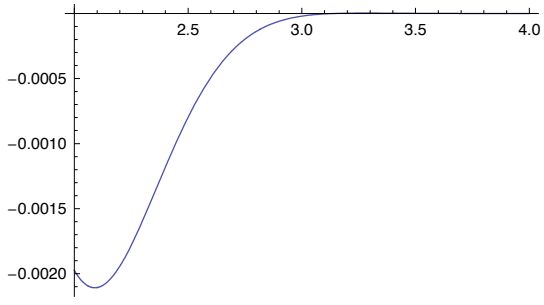
In this case the external code calls *Mathematica* when the `Zeta` function has to be evaluated. After the evaluation the computation proceeds in the external code.

Note that it is the installed code for the function `f` that is executed above, and not the original *Mathematica* function. In the installed code, the argument of `f` must be real, according to our declaration. As a result, `f[5/2]`, in which we pass a rational number as an argument, is left unevaluated.

We again plot the function, but this time using the external code to evaluate it:

```
In[11]:= Plot[f[x], {x, 2, 4}, PlotRange → All]
```

Out[11]=



3.5.3. External Functions

We can have references to external objects in C++ code generated by *MathCode*. Let us consider three very simple external functions that compute x^2, e^x, and $\sin(x)$ to illustrate the idea. These must be defined as follows in an external source file that must be in the working directory.

```
In[1]:= Needs["MathCode`"]
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= SetDirectory[$MCRoot <> "/Demos/Overview"];
```

```
In[3]:= FilePrint["external1.cc"]
```

```
#include <math.h>

extern double extsqr(const double &x) {
 return x*x;
}

extern double extexp(const double &x) {
 return exp(x);
}

extern double extoscillation(const double &x)
{
 return sin(x);
}
```

Observe here that each function definition, which is in C language syntax, is followed by a "wrapper" that enables *MathCode* to recognize the object as external. We can then create an object file corresponding to these functions and link the object as follows.

```
In[4]:= extsqr[Real x_] → Real := ExternalFunction[];
        extexp[Real x_] → Real := ExternalFunction[];
        extoscillation[Real x_] → Real := ExternalFunction[];
```

We define a function to create a list of numbers using the external functions.

```
In[7]:= Makeplot[Integer n_] → Real[n] := Module[{Integer i, Real[_] arr},
          arr = Table[extsqr[extoscillation[0.1 i]]
              extexp[-extsqr[0.03 i]], {i, n}]; arr]
```

We now compile the package. Since this is a very small example, we do not bother to create a special package for the code.

```
In[8]:= CompilePackage[]
```

```
Successful compilation to C++: 4 function(s)
```

Let us now create the *MathLink* binary; to do this when there are external functions, we must specify the option `NeedsExternalObjectModule` as follows.

```
In[9]:= MakeBinary[NeedsExternalObjectModule → {"external1"}]
```

Here, as we noted above, `external1` and `external2` represent the external object modules external1.o and external2.o. Install the *MathCode*-compiled code so it is called using *MathLink*.

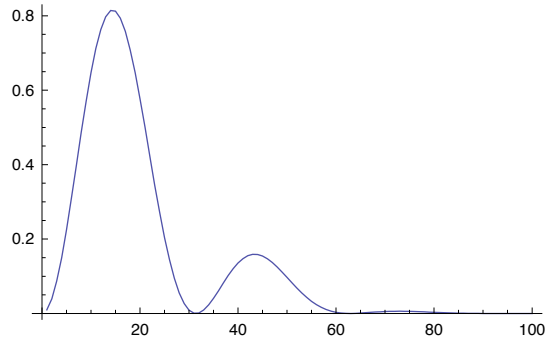
```
In[10]:= InstallCode[];
```

```
Global is installed.
```

When we make the following plot, it is the external code for `extsqr`, `extexp`, and `extoscillation` that is used.

```
In[11]:= ListPlot[Makeplot[100], Joined → True]
```

Out[11]=



□ 3.6. Common Subexpression Elimination

Consider the following function whose definition contains a number of common subexpressions (e.g., $1 + x^2$ and $\sqrt{1 + x^2}$).

```
In[1]:= Needs["MathCode`"];
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= g[Real x_] → Real :=
```

$$\frac{x \cos[x] \cos\left[\sqrt{1+x^2}\right]}{(1+x^2)^{3/2}\left(1+\cos[x]^2\right)} - \frac{2x \cos[x] \sin\left[\sqrt{1+x^2}\right]}{(1+x^2)^2\left(1+\cos[x]^2\right)} +$$

$$\frac{2\cos[x]^2 \sin[x] \sin\left[\sqrt{1+x^2}\right]}{(1+x^2)\left(1+\cos[x]^2\right)^2} - \frac{\sin[x] \sin\left[\sqrt{1+x^2}\right]}{(1+x^2)\left(1+\cos[x]^2\right)}$$

There are very efficient algorithms to evaluate functions containing common subexpressions. The basic idea is to evaluate common subexpressions only once and put the results in temporary variables.

Now we generate C++ code using *MathCode* and run it.

```
In[3]:= BuildCode[]
```

```
Successful compilation to C++: 1 function(s)
```

```
In[4]:= InstallCode[];
```

```
Global is installed.
```

```
In[5]:= Timing[Do[g[3.0], {100}]]

Out[5]= {0.15, Null}
```

MathCode does common subexpression elimination (CSE) when the option `EvaluateFunctions` is given to `CompileCode[ ]` or `BuildCode[ ]`. This basic strategy could be further improved for special cases in future versions of *MathCode*. Moreover, since mathematical expressions are intrinsically free of side effects and do not have a specific evaluation order, the CSE optimization may change the order of computing subexpressions if this improves performance. Changing the order can sometimes have a small influence on the result when floating-point arithmetic is used.

```
In[6]:= UninstallCode[];

In[7]:= CleanMathCodeFiles[Confirm → False, CleanAllBut → {}];

In[8]:= BuildCode[EvaluateFunctions → {g}]
```

```
Successful compilation to C++: 1 function(s)
```

```
In[9]:= InstallCode[];
```

```
Global is installed.
```

```
In[10]:= Timing[Do[g[3.0], {100}]]

Out[10]= {0.21, Null}
```

We take a look at the generated C++ file.

```
In[11]:= FilePrint["Global.cc"]
```

```
#include "Global.h"

#include "Global.icc"

#include <math.h>
void Global_TGlobalInit ()
{
;
}

double Global_Tg ( const double &x)
{
  double mc_T1;
  double mc_T2;
  double mc_T3;
  double mc_T4;
  double mc_T5;
  double mc_T6;
  double mc_T7;
  double mc_T8;
  double mc_T9;
  double mc_T10;
  double mc_T11;
```

```
  double mc_T12;
  double mc_T13;
  double mc_T14;
  double mc_T15;
  double mc_T16;
  double mc_T17;
  double mc_T18;
  double mc_T19;
  double mc_T20;
  double mc_T21;
  double mc_T22;
  double mc_T23;
  double mc_T24;
  double mc_T25;
  mc_T1 = (x*x);
  mc_T2 = 1+mc_T1;
  mc_T3 = cos(x);
  mc_T4 = (mc_T3*mc_T3);
  mc_T5 = 1+mc_T4;
  mc_T6 = mc_T5*mc_T2;
  mc_T7 =             0.5;
  mc_T8 = pow(mc_T2, mc_T7);
  mc_T9 = sin(mc_T8);
  mc_T10 = sin(x);
  mc_T11 = mc_T10*mc_T9;
  mc_T12 = mc_T11/mc_T6;
  mc_T13 = -mc_T12;
  mc_T14 = pow(mc_T5, -2);
  mc_T15 = 2*mc_T4*mc_T14*mc_T10*mc_T9;
  mc_T16 = mc_T15/mc_T2;
  mc_T17 = pow(mc_T2, -2);
  mc_T18 = -2*x*mc_T17*mc_T3*mc_T9;
  mc_T19 = mc_T18/mc_T5;
  mc_T20 = cos(mc_T8);
  mc_T21 =            -1.5;
  mc_T22 = pow(mc_T2, mc_T21);
  mc_T23 = x*mc_T22*mc_T3*mc_T20;
  mc_T24 = mc_T23/mc_T5;
  mc_T25 = mc_T24+mc_T19+mc_T16+mc_T13;
  return mc_T25;
}
```

Note how the computation of the function has been divided into small subexpressions that are evaluated only once and then stored in temporary variables for future use. This gives very efficient code for large functions. The speed enhancement of roughly 150% brought about by CSE in this example is not appreciable because the example itself is rather small.

□ 3.7. Extended Matrix Operations

When dealing with matrices, it is very convenient to have a short notation for part extraction. *MathCode* extends the functionality of `Part[ ]` or `[[ ]]` to achieve this.

Consider the following 4×5 matrix:

```
In[12]:= A = Table[a[i, j], {i, 4}, {j, 5}]; A // MatrixForm
```

```
Out[12]//MatrixForm=
⎛a[1, 1]  a[1, 2]  a[1, 3]  a[1, 4]  a[1, 5]⎞
⎜a[2, 1]  a[2, 2]  a[2, 3]  a[2, 4]  a[2, 5]⎟
⎜a[3, 1]  a[3, 2]  a[3, 3]  a[3, 4]  a[3, 5]⎟
⎝a[4, 1]  a[4, 2]  a[4, 3]  a[4, 4]  a[4, 5]⎠
```

We can extract rows 2 to 4 as follows, with the shorthand available in *MathCode*.

```
In[13]:= A[[2 | 4]] // MatrixForm
```

```
Out[13]//MatrixForm=
⎛a[2, 1]  a[2, 2]  a[2, 3]  a[2, 4]  a[2, 5]⎞
⎜a[3, 1]  a[3, 2]  a[3, 3]  a[3, 4]  a[3, 5]⎟
⎝a[4, 1]  a[4, 2]  a[4, 3]  a[4, 4]  a[4, 5]⎠
```

We can extract the elements in all rows that belong to column 3 and higher:

```
In[14]:= A[[_, 3 | _]] // MatrixForm
```

```
Out[14]//MatrixForm=
⎛a[1, 3]  a[1, 4]  a[1, 5]⎞
⎜a[2, 3]  a[2, 4]  a[2, 5]⎟
⎜a[3, 3]  a[3, 4]  a[3, 5]⎟
⎝a[4, 3]  a[4, 4]  a[4, 5]⎠
```

We can assign values to a submatrix of `A`.

```
In[15]:= A[[2 | 3, 2 | 3]] = {{1, 2}, {3, 4}};
```

```
In[16]:= A // MatrixForm
```

```
Out[16]//MatrixForm=
⎛a[1, 1]  a[1, 2]  a[1, 3]  a[1, 4]  a[1, 5]⎞
⎜a[2, 1]  1        2        a[2, 4]  a[2, 5]⎟
⎜a[3, 1]  3        4        a[3, 4]  a[3, 5]⎟
⎝a[4, 1]  a[4, 2]  a[4, 3]  a[4, 4]  a[4, 5]⎠
```

All these operations belong to the compilable subset and can result in compact code. Note: `A[[2|4]]]` denotes the same *Mathematica* computation as `Take[A,{2,4},All]`, and `A[[_, 3|_]]` is equivalent to `Take[A,All,{3,-1}]`.

□ 3.8. Array Declaration and Dimension

In this subsection, we give a few examples of array declarations. There are two main cases to consider.

- Arrays that are passed as function parameters or returned as function values, where the actual array size has been previously allocated
- Declaration of array variables, usually specifying both the type and the allocation of the declared array

There are five allowed ways to specify array dimension sizes in array types for function arguments and results.

- Integer constant dimension sizes, for example: `Real[3,4]`
- Symbolic-constant dimension sizes, for example: `Real[three, four]`
- Unknown dimension sizes with unnamed placeholders, for example: `Real[_, _]`
- Unknown dimension sizes with named placeholders, for example: `Real[n_, m_]`
- Unknown dimension sizes with variables as dimension sizes, for example: `Real[n, m]`

The dimension sizes can be constant, in which case the size information is part of the type. Alternatively, the sizes are unknown and thus fixed later at runtime when the array is allocated. Such unknown dimension sizes are specified through named (e.g., `n_`) or unnamed (`_`) placeholders.

All arrays that are passed as arguments to functions have already been allocated at runtime. Thus, their sizes are already determined. These sizes might, however, be different for different calls. Therefore it is not allowed to specify conflicting dimension sizes through integer variables (e.g., `Real[n, m]`) in array types of function parameters or results, as can be done for ordinary declared variables. Only constants and named, or unnamed, placeholders are allowed.

We now give examples of the five different ways of specifying array dimension information in variable declarations. The examples show a global variable declaration using `Declare`, but the same kinds of declarations can also be used for local declarations in functions.

The fifth case is where sizes are specified through integer variables. This is needed to handle declaration and allocation of arrays for which the sizes are not determined until runtime.

- Integer constant dimension sizes using the array `arr`:

```
Declare[Real[3, 4] arr];
```

- Symbolic constant dimension sizes:

```
Declare[Real[three, four] arr];
```

- Unknown dimension sizes with unnamed placeholders:

```
Declare[Real[_, _] arr];
```

- Unknown dimension sizes with named placeholders:

```
Declare[Real[k_, m_] arr];
```

- Unknown dimension sizes that are specified and fixed to the values of integer variables, for example, `n`, `m` (e.g., function parameters, local or global variables that are visible from the declaration):

```
Declare[Real[n, m] arr];
```

Integer variables, such as `n` and `m`, are assumed to be assigned once; that is, their values are not changed after the initial assignment, so that the declared sizes of allocated arrays are kept consistent with the values of those variables. This single-assignment property is not checked by the current version of the system, however. Thus, the user is responsible for maintaining such consistency.

■ 4. Application Examples

□ 4.1. Summary of Examples

In the following we present a few complete application examples using *MathCode*. The first example application is a small *Mathematica* program called `SinSurface` (Section 4.2), which has been designed to illustrate two basic modes of the code generator: compiling without symbolic evaluation (the default mode, in which the function body is translated into C++ as it is), and compilation preceded by symbolic expansion, which is indicated by setting the option `EvaluateFunctions→True` (the function body is expanded using symbolic operations, simplified, and then translated).

The second example, presented in Section 4.3, is an implementation of the Gaussian elimination procedure to solve a linear algebraic system of equations (see any standard text on numerical techniques for a discussion of the procedure, e.g., [2]). Here we compile generated C++ code with various options and do a detailed performance analysis.

In Section 4.4, we discuss the example of SuperLU, an external library [3] that performs efficient sparse matrix operations. We give an example of a program useful in solving partial differential equations that calls the SuperLU library and some of its object modules to solve a matrix equation of the type $A.X = B$, where A is a very sparse square matrix.

□ 4.2. The SinSurface Application Example

Here we describe the `SinSurface` program example. The actual computation is performed by the functions `calcPlot`, `sinFun2`, and their helper functions. The two functions `calcPlot` and `sinFun2` in the `SinSurface` package will be translated into C++ and are declared together with a global array `xyMatrix`.

The array `xyMatrix` represents a 21×21 grid on which the numerical function `sinFun2` will be computed. The function `calcPlot` accepts five arguments: four of these are coordinates describing a square in the x-y plane and one is a counter (`iter`) to make the function repeat the computation as many times as necessary in order to measure execution time. For each point on a 21×21 grid, the numeric function `sinFun2` is called to compute a value that is stored as an element in the matrix representing the grid.

4.2.1. Introduction

The `SinSurface` example application computes a function (here `sinFun2`) over a two-dimensional grid. The function values are stored in the matrix `xyMatrix`. The execution of compiled C++ code for the function `sinFun2` is over 500 times faster than evaluating the same function interpretively within *Mathematica*.

The function sinFun2 computes essentially the same values as sin($x + y$), but in a more complicated way, using a rather large expression obtained through converting the arguments into polar coordinates (through ArcTan) and then using a series expansion of both Sin and Cos, up to 10 terms. The resulting large symbolic expression (more than a page) becomes the body of sinFun2, and is then used as input to CompileEvaluateFunction to generate efficient C++ code. The symbolic expression and the call to CompileEvaluateFunction is initiated by using the EvaluateFunctions option.

4.2.2. Initialization

We first set the directory in which *MathCode* will store the auxiliary files, the C++ code, and executable, and then load *MathCode*.

```
In[1]:= Needs["MathCode`"]
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= SetDirectory[$MCRoot <> "/test"];
```

The SinSurface package starts in the usual way with a BeginPackage declaration that references other packages. MathCodeContexts is needed in order to call the code generation related functions.

```
In[3]:= BeginPackage["SinSurface`", {MathCodeContexts}];
        Clear["SinSurface`*"];
```

Next we define possibly exported symbols. Even though it is not necessary here, we enclose these names within Begin["SinSurface`"] … End[ ] as a kind of context bracket, since this can be put into a cell, which can be conveniently re-evaluated by itself if new names are added to the list.

```
In[5]:= Begin["SinSurface`"]
        xyMatrix;
        calcPlot;
        sinFun1;
        sinFun2;
        arcTan;
        sin;
        cos;
        plot;
        cplus;
        plotfile;
        End[]
Out[5]= SinSurface`
```

Now we set compilation options as follows. This defines how the functions and variables in the package should be compiled to C++. By default, all typed variables and functions are compiled. However, the compilation process can be controlled in a more detailed way by giving compilation options to Compile`Package or via SetCompilationOptions. For example, in this package the function sinFun2 should be symbolically evaluated before being translated to code, because it contains symbolic operations; the functions sin, cos, and arcTan

should not be compiled at all, because they are expanded within the body of `sinFun2`. The remaining typed function, `calcPlot`, will be compiled in the normal way.

```
In[6]:= SetCompilationOptions[EvaluateFunctions → {sinFun2},
          UnCompiledFunctions → {sin, cos, arcTan},
          MainFileAndFunction → "int main(){return 0;}"];
```

4.2.3. The Body of the SinSurface Package

We begin the implementation section of the `SinSurface` package, where functions are defined. This is usually private, to avoid accidental name shadowing due to identical local variables in several packages.

```
In[7]:= Begin["SinSurface`Private`"];
```

Declare public global variables and private package-global variables:

```
In[8]:= Declare[Real[21, 21] xyMatrix];
```

Taylor-expanded `sin` and `cos` functions called by `sinFun2` are now defined, just for the sake of the example, even though such a series gives lower relative accuracy close to zero. A substitution of the symbol `z` for the actual parameter `x` is necessary to force the series expansion before replacing with the actual parameter.

```
In[9]:= sin[Real[x_]] → Real := Normal[Series[Sin[z], {z, 0, 10}]] /. z → x;
        cos[Real[x_]] → Real := Normal[Series[Cos[z], {z, 0, 10}]] /. z → x;
```

Define `arcTan`, which converts a grid point to an angle, called by `sinFun2`:

```
In[11]:= arcTan[Real[x_], Real[y_]] → Real :=
           If[x < 0, π, 0] + If[x == 0, 1/2 Sign[y] π, ArcTan[y/x]];
```

`sinFun2` is the function to be computed and plotted, called by `calcPlot`. It provides a computationally heavy (series expansion) and complicated way of calculating an approximation to $\sin(x + y)$. This gives an example of a combination of symbolic and numeric operations as well as a rather standard mix of arithmetic operations. The expanded symbolic expression, which comprises the body of `sinFun2`, is about two pages long when printed.

Note that the types of local variables to `sinFun2` need not be declared, since setting the `EvaluateFunctions` option will make the whole function body be symbolically expanded before translation.

Note also that a function should be without side effects in order to be symbolically expanded before final code generation. For example, there should be no assignments to global variables or input/output, since the relative order of these actions when executing the code often changes when the symbolic expression is created and later rearranged and optimized by the code generator.

```
In[12]:= sinFun2[Real x_, Real y_] → Real :=
          Block[{r, xx, yy},
           r = √(x^2 + y^2) ;
           xx = r cos[arcTan[x, y]];
           yy = r sin[arcTan[x, y]];
           sin[xx + yy]]
```

The function `calcPlot` calculates data for a plot of `sinFun2` over a 21×21 grid, which is returned as a 21×21 array.

```
In[13]:= calcPlot[Real xmin_, Real xmax_, Real ymin_,
            Real ymax_, Integer iter_] → Real[21, 21] :=
           Module[{Integer n = 20, Real {x, y}, Integer {i, j, count}},
            For[count = 1, count ≤ iter, count = count + 1,
             For[i = 1, i ≤ n + 1, i = i + 1,
              For[j = 1, j ≤ n + 1, j = j + 1,
               x = xmin + (xmax - xmin) (i - 1) / n; y = ymin + (ymax - ymin) (j - 1) / n;
               xyMatrix[[i, j]] = sinFun2[x, y]]]];
            xyMatrix]
```

```
In[14]:= End[]
         EndPackage[];
```

```
Out[14]= SinSurface`Private`
```

4.2.4. Execution

We first execute the application interpretively within *Mathematica*, and then use `Compile` on the key function and execute the application again. Then we compile the application to C++, build an executable, and call the same functions from *Mathematica* via *MathLink*.

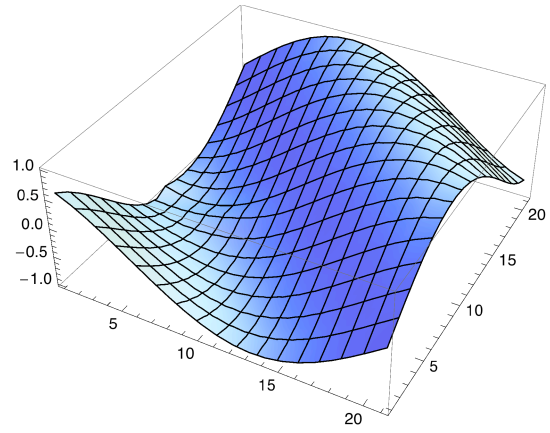
Let us first do the *Mathematica* evaluation and plot.

```
In[16]:= meval = Timing[plot = calcPlot[-2., 2., -2., 2., 20]][[1]] / 20
```

```
Out[16]= 0.1305
```

```
In[17]:= ListPlot3D[plot]
```

Out[17]=



Next, we redefine `sinFun2` to become a compiled version, using *Mathematica*'s standard `Compile`.

```
In[18]:= sinFun2 = Compile[{x, y}, Evaluate[sinFun2[x, y]]];

In[19]:= compeval = Timing[plot = calcPlot[-2., 2., -2., 2., 100];]

Out[19]= {7.109, Null}

In[20]:= compeval = compeval[[1]] / 100

Out[20]= 0.07109

In[21]:= sinFun2 = .
```

4.2.5. Using the MathCode Code Generator

Compile the `SinSurface` package.

```
In[22]:= CompilePackage["SinSurface"]
```

```
MathCodeConv`defConv::untypedlocalvars: Warning: Untyped local variable(s):
    {SinSurface`Private`r, SinSurface`Private`xx, SinSurface`Private`yy} in function with
    head sinFun2[SinSurface`Private`x_, SinSurface`Private`y_]. Real type(s) assumed
```

```
Successful compilation to C++: 2 function(s)
```

The warnings concern local variables in `sinFun2` that have no type information. This is not important because those variables disappear upon symbolic expansion.

The command `MakeBinary` compiles the generated code using a compiler (g++ in the present case). The object code is by default linked into the executable SinSurfaceml.exe for calling the compiled code via *MathLink*.

```
In[23]:= MakeBinary[];
```

If any problems are encountered during code compilation, then warning and error messages are shown. Otherwise no messages are shown. When `MakeBinary` is called without arguments, the call applies to the current package.

The command `InstallCode` installs and connects the external process containing the compiled and linked `SinSurface` code.

```
In[24]:= InstallCode["SinSurface"]
```

```
SinSurface is installed.
```

```
Out[24]= LinkObject[".\SinSurfaceml.exe", 14, 7]
```

Execute the generated C++ code for `calcPlot`.

```
In[25]:= AbsoluteTiming[plot = calcPlot[-2.0, 2.0, -2.0, 2.0, 3000];]
```

```
Out[25]= {3.6408347, Null}
```

Since the external computation was performed 3000 times, the time needed for one external computation is

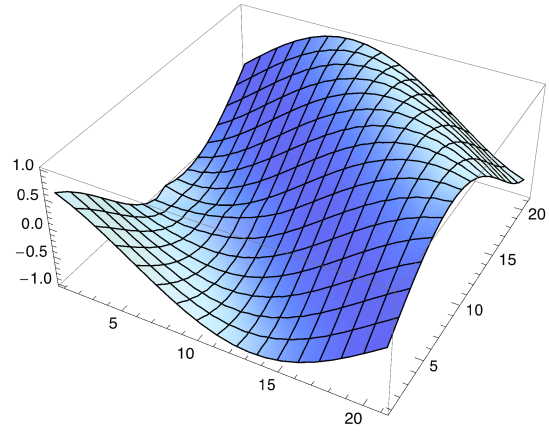
```
In[26]:= externaleval = %[[1]] / 3000
```

```
Out[26]= 0.0012136116
```

Check that the result appears graphically the same.

```
In[27]:= ListPlot3D[plot]
```

Out[27]=



4.2.6. Performance Comparison

Let us now compare the running times for the three cases, the standard *Mathematica*, compiled *Mathematica*, and the generated C++ code.

```
In[28]:= {meval / externaleval, compeval / externaleval}
```

```
Out[28]= {107.53, 58.5772}
```

The performance between the three forms of execution are compared in Table 1. The generated C++ code for this example is roughly 100 times faster than standard interpreted *Mathematica* code, and 50 times faster than code compiled by the internal *Mathematica* `Compile` command. This is on a Toshiba Satellite-2100, 400 Mhz AMD-K6, running Windows XP Pro SP2 and *Mathematica* 6, without inline and norange optimization. If the inline is specified,

the inline directive is passed to the C++ compiler for all functions to be compiled. If norange is specified, array element index range checking is turned off in the code generated by the C++ compiler, resulting in faster but less safe code.

We should emphasize that the comparisons in Table 1 are rather crude for several reasons. From a separate measurement, the loop part of `calcPlot` excluding the call to `sinFun2` comprises 25% of the total `calcPlot` time executed in interpreted *Mathematica*. The `calcPlot` function itself cannot be compiled using `Compile`, since it contains an assignment to a global matrix variable that cannot currently be handled by `Compile`. This might be regarded as unfair to `Compile`. On the other hand, a *MathLink* overhead (divided by 500) in returning the 21×21 matrix is embedded in the figure for *MathCode*, which can be regarded as unfair to *MathCode*. A better comparison for another small application example is available in Section 4.3.6.

```
In[29]:= TableForm[{
           {"Execution Form", "Time consumed", "Relative"},
           {"Standard Mathematica", meval, meval / externaleval},
           {"Compile[]", compeval, compeval / externaleval},
           {"External C++ via MathLink", externaleval, 1}}]
```

Out[29]//TableForm=

Execution Form	Time consumed	Relative
Standard *Mathematica*	0.1305	107.53
Compile[]	0.07109	58.5772
External C++ via *MathLink*	0.0012136116	1

Table 1. Approximate performance comparison for the `calcPlot` example.

□ 4.3. Gauss Application Example

4.3.1. Introduction

In this section, we present a textbook algorithm, Gaussian elimination (e.g., [2]), to solve a linear equation system. The given linear system, represented by a matrix equation of the type $A.X = B$, is subjected to a sequence of transformations involving a pivot, resulting in the solution to the system, contained in the matrix X.

The following subsections illustrate the various aspects of the application.

4.3.2. Initialization

```
In[1]:= Needs["MathCode`"]
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= SetDirectory[$MCRoot <> $PathnameSeparator <>
          "Demos" <> $PathnameSeparator <> "Gauss"];

In[3]:= BeginPackage["Gauss`", {MathCodeContexts}];
```

Define exported symbols:

```
In[4]:= Begin["Gauss`"];
        GaussSolveArraySlice;
        End[];
```

4.3.3. Body of the Package

We now define the function `GaussSolveArraySlice`, based on the Gaussian elimination algorithm.

```
In[7]:= Begin["`Private`"];
        GaussSolveArraySlice[Real[n_, n_] ain_, Real[n_, m_] bin_,
           Integer iterations_] → Real[n, m] :=
         Module[{Real[n] dumc, Real[n, n] a, Real[n, m] b,
           Integer[n] {ipiv, indxr, indxc}, Integer {i, k, l, irow, icol},
           Real {pivinv, amax, tmp}, Integer {beficol, afticol, count}},
          For[count = 1, count ≤ iterations, count = count + 1, (a = ain;
            b = bin;
            For[k = 1, k ≤ n, k = k + 1, ipiv[[k]] = 0];
            For[i = 1, i ≤ n, i = i + 1,
             (*find the matrix element with largest absolute value*)
             amax = 0.0;
             For[k = 1, k ≤ n, k = k + 1,
              If[ipiv[[k]] == 0,
               For[l = 1, l ≤ n, l = l + 1, If[ipiv[[l]] == 0,
                 If[Abs[a[[k, l]]] > amax, amax = Abs[a[[k, l]]];
                  irow = k;
                  icol = l]]
               ]
              ]
             ];
             ipiv[[icol]] = ipiv[[icol]] + 1;
             If[ipiv[[icol]] > 1, "*** Gauss2 input data error ***" >> "";
              Break];
             (*if irow ≠ icol,
             then interchange rows irow and icol in both a and b*)
             If[irow ≠ icol, For[k = 1, k ≤ n, k = k + 1, tmp = a[[irow, k]];
               a[[irow, k]] = a[[icol, k]];
               a[[icol, k]] = tmp];
              For[k = 1, k ≤ m, k = k + 1, tmp = b[[irow, k]];
               b[[irow, k]] = b[[icol, k]];
               b[[icol, k]] = tmp]];
             indxr[[i]] = irow;
             indxc[[i]] = icol;
             If[a[[icol, icol]] == 0,
              Print["*** Gauss2 input data error 2 ***"];
              Break];
             (*prepare to divide by the
              pivot and subsequent row transformations*)
             pivinv = 1.0 / a[[icol, icol]];
             a[[icol, icol]] = 1.0;
```

```
      a[[icol, _]] = a[[icol, _]] * pivinv;
      b[[icol, _]] = b[[icol, _]] * pivinv;
      dumc = a[[_, icol]];
      For[k = 1, k ≤ n, k = k + 1, a[[k, icol]] = 0];
      a[[icol, icol]] = pivinv;
      For[k = 1, k ≤ n, k = k + 1,
       If[k ≠ icol, a[[k, _]] = a[[k, _]] - dumc[[k]] * a[[icol, _]];
        b[[k, _]] = b[[k, _]] - dumc[[k]] * b[[icol, _]]]]
     ];
     For[l = n, l ≥ 1, l = l - 1,
      For[k = 1, k ≤ n, k = k + 1, tmp = a[[k, indxr[[l]]]];
       a[[k, indxr[[l]]]] = a[[k, indxc[[l]]]];
       a[[k, indxc[[l]]]] = tmp]]
   ];
   b];
End[];
EndPackage[];
```

This function accepts three arguments in an attempt to solve a matrix equation of the form $A.X = B$. The first two arguments are essentially the matrices A and B. The third argument specifies the number of times the body of the function must run; this is useful for an accurate measurement of the running time. The function output (X) has the same shape as the second argument (B).

4.3.4. Mathematica Execution

Let us create two random matrices.

```
In[10]:= a = RandomReal[{0, 1}, {10, 10}];
         b = RandomReal[{0, 1}, {10, 2}];
```

In the following, `loops=1 factor` specifies the number of times the body of `GaussSolveArraySlice` runs. The appropriate value of `loops` for reliable estimates of running time is system dependent. A reasonable value of `factor` for a 1.5 GHz computer is about 10. The output checks that the solution obtained is correct.

```
In[12]:= factor = 30; loops = 2 factor;
         s = Timing[c = GaussSolveArraySlice[a, b, loops];];
         meval = s[[1]] / loops;
         Print["TIMING FOR NON-COMPILED VERSION= ", meval];
         MatrixForm[a.c - b]
```

```
TIMING FOR NON-COMPILED VERSION= 0.1401
```

Out[16]//MatrixForm=

$$\begin{pmatrix} 0. & 0. \\ -1.33227\times10^{-15} & 0. \\ 2.22045\times10^{-16} & -1.11022\times10^{-16} \\ 6.66134\times10^{-16} & -2.22045\times10^{-16} \\ 2.22045\times10^{-15} & -3.33067\times10^{-16} \\ 2.22045\times10^{-16} & -1.249\times10^{-16} \\ -2.22045\times10^{-15} & 5.55112\times10^{-17} \\ 1.77636\times10^{-15} & -1.11022\times10^{-16} \\ -2.66454\times10^{-15} & 0. \\ -1.77636\times10^{-15} & -1.66533\times10^{-16} \end{pmatrix}$$

4.3.5. Generating and Running the C++ Code

The command `BuildCode` translates the package and produces an executable.

```
In[17]:= BuildCode["Gauss"]
```

```
Successful compilation to C++: 1 function(s)
```

Interpreted versions are removed, and compiled ones are used instead.

```
In[18]:= InstallCode["Gauss"]
```

```
Gauss is installed.
```

```
Out[18]= LinkObject[".\Gaussml.exe", 14, 7]
```

```
In[19]:= c = GaussSolveArraySlice[a, b, 1];
```

We now make two runs of the C++ code for the package `Gauss`. The first run evaluates the body of `GaussSolveArraySlice` `loops` times, and returns the solution only once. The second run evaluates the body of `GaussSolveArraySlice` only once, but does this inside a `Do`-loop for `loops` times, returning the solution `loops` times as a result. Clearly, there is overhead in the second run, and the time taken is expected to be higher, as can be seen from the following.

```
In[20]:= loops = 800 * factor;
         externaleval =
          ((c = GaussSolveArraySlice[a, b, loops];) // AbsoluteTiming)[[1]] /
           loops

Out[21]= 0.00029885107

In[22]:= loops = 500 * factor;
         externalevalPass =
          ((Do[c = GaussSolveArraySlice[a, b, 1], {loops}];) //
              AbsoluteTiming)[[1]] / loops

Out[23]= 0.00110737671
```

We now also solve the same system using `LinearSolve`.

```
In[24]:= loops = 10 000 * factor;
         internalEval =
          ((Do[c = LinearSolve[a, b], {loops}];) // AbsoluteTiming)[[1]] /
           loops

Out[25]= 0.000031044051

In[26]:= UninstallCode["Gauss"];

In[27]:= Map[DeleteFile, FileNames["*.o"]];
         Map[DeleteFile, FileNames["*.obj"]];
         Map[DeleteFile, FileNames["*.exe"]];
```

4.3.6. Performance Comparison

We present the performance analysis in Table 2. As we observe from the table, a performance enhancement by a factor of approximately 500 can be obtained for the compiled C++ code over interpreted *Mathematica*. More importantly, we are able to get a performance close to `LinearSolve`, although we have implemented a simple version of a Gaussian elimination algorithm directly from a textbook as straight-line code without any attempts at tuning or optimization. Also note that `LinearSolve` is more general in that it can also handle sparse arrays efficiently using the *SuperLU* package linked into the *Mathematica* kernel. To achieve similar generality with the *MathCode* package, the example would need to be extended and a *SuperLU* routine, for example, called as an external function from the generated code. In general, it is better to use already implemented robust and reliable routines from packages like *LAPACK* and *SuperLU*, which also can be called as external functions from *MathCode*-generated code. The Gauss example in this article is not intended to replace such routines but to be a simple example of using *MathCode*.

```
In[29]:= TableForm[{{"", "Time(seconds)", "Relative"},
    {"Standard interpreted Mathematica", meval, meval / externaleval},
    {"C++ with call overhead",
     externalevalPass, externalevalPass / externaleval},
    {"C++ without call overhead", externaleval, 1},
    {"LinearSolve", internalEval, internalEval / externaleval}}]
```

Out[29]//TableForm=

	Time(seconds)	Relative
Standard interpreted *Mathematica*	0.1401	468.795
C++ with call overhead	0.00110737671	3.7054466
C++ without call overhead	0.00029885107	1
LinearSolve	0.000031044051	0.103877996

Table 2. Performance comparison for the Gauss example.

□ 4.4. External Libraries and Functions

We now demonstrate how to call external functions and libraries using *MathCode*. We have already presented an example of how to do this for three very simple functions, x, $\exp(x)$, and $\sin(x)$, in Section 3.5.3. In this section we present a more realistic application example that illustrates how to employ an external library for handling sparse matrix systems that arise in the solution of partial differential equations [6].

We take as our example the problem of solving the one-dimensional diffusion equation using the method of finite differences.

$$\frac{\partial u(x, t)}{\partial t} == \frac{\partial^2 u(x, t)}{\partial x^2}$$

In this method, the continuous x domain is approximated by a set of discrete points called a *grid*, and each derivative is replaced by a certain linear function of values of the dependent variables, called a *finite difference*. For the previous equation, a variant of this method gives

$$\frac{u(x, t+1) - u(x, t)}{k} == \frac{u(x-1, t) - 2\,u(x, t) + u(x+1, t)}{h^2},$$

where now x and t are assumed to take integer values, and k and h are step sizes along x and t directions, respectively. The algebraic equation must be solved at each grid point, thus resulting in a simultaneous system of equations, which is essentially a matrix system of the form $A.X = B$. Since the matrix system in this case is very sparse, we solve it using the sparse matrix library called SuperLU [3].

The rest of this section assumes that the SuperLU library has been compiled. We now explain how to call the external objects based on this library using *MathCode*.

```
In[1]:= Needs["MathCode`"];
```

```
MathCode C++ 1.4.0 for mingw32 loaded from C:\MathCode
```

```
In[2]:= SetDirectory[$MCRoot <> "\Demos\ExternalFunction"];

In[3]:= BeginPackage["foo`", {MathCodeContexts}];
```

Here is the *Mathematica* code to solve the one-dimensional diffusion equation.

```
In[4]:= SolveDiffusion1D[Nx_, dt_, nnz_, xasize_, U_] :=
 Module[{k, x, dx, kt, rhsmat,
   colmat, rowmat, valmat, amat, asubmat, xamat},
  (*initialize variables and arrays*)
  kt = 0; dx = 1 / (Nx - 1); rhsmat = Table[0., {Nx}];
  colmat = Table[0, {nnz}];
  rowmat = Table[0, {nnz}]; valmat = Table[1., {nnz}];
  amat = Table[1., {nnz}]; asubmat = Table[0, {nnz}];
  xamat = Table[0, {xasize}];
  (*define the matrices*)
  For[x = 1, x < 2, x = 1 + x, rhsmat[[x]] = 0.; (++kt; colmat[[kt]] = x;
    rowmat[[kt]] = x; valmat[[kt]] = 1)]; For[x = 2, x < Nx, x = 1 + x,
   rhsmat[[x]] = U[[x]] / dt + (U[[-1 + x]] - 2 U[[x]] + U[[1 + x]]) / dx^2;
   (++kt; colmat[[kt]] = x; rowmat[[kt]] = x; valmat[[kt]] = 1 / dt)];
  For[x = Nx, x < 1 + Nx, x = 1 + x, rhsmat[[x]] = 0.;
   (++kt; colmat[[kt]] = x; rowmat[[kt]] = x; valmat[[kt]] = 1)];
  (*transform the matrices into SuperLU format*)
  kt = 0; Do[Do[If[colmat[[k1]] == k, ++kt; amat[[kt]] = valmat[[k1]];
     asubmat[[kt]] = -1 + rowmat[[k1]]], {k1, 1, nnz}], {k, 1, Nx}];
  kt = 0; Do[Do[If[colmat[[k1]] == k, ++kt], {k1, 1, nnz}];
   xamat[[1 + k]] = kt, {k, 1, Nx}];
  (*call SuperLU-based function to solve the matrix system A.x=B*)
  linsolvepp[Nx, Nx, nnz, 1, amat, asubmat, xamat, rhsmat]
 ]
```

In[5]:= **FilePrint[$MCRoot <> "/Demos/ExternalFunction/linsolvepp.cc"]**

```
#include <math.h>
#define LM_NNNN
#include "lightmat.h"

extern "C" void linsolve(int m, int n, int nnz, int nrhs, double *a,
int *asub, int *xa, double *rhs );

void linsolvepp(const int &nx, const int &nx1, const int &nx2,
const int &one, const doubleN &expamat, const intN &expasubmat,
const intN &expxamat, doubleN &exprhsmat)
{

   double * expamat_c = new double [expamat.dimension(1)];
   int * expasubmat_c = new int [expasubmat.dimension(1)];
   int * expxamat_c = new int [expxamat.dimension(1)];
   double * exprhsmat_c = new double [exprhsmat.dimension(1)];

   expamat.Get(expamat_c);
   expasubmat.Get(expasubmat_c);
   expxamat.Get(expxamat_c);
   exprhsmat.Get(exprhsmat_c);

   linsolve(nx, nx1, nx2, one, expamat_c, expasubmat_c, expxamat_c,
exprhsmat_c);

    exprhsmat.Set(exprhsmat_c);

};
```

Note that this source file is somewhat different from the one in Section 3.5.3, mainly because arrays are involved here. This wrapper function makes a reference to a C function linsolve( ) that is defined in the following source file.

In[6]:= **FilePrint[$MCRoot <> "/Demos/ExternalFunction/linsolve.c"]**

```
/*
 * -- SuperLU routine (version 2.0) --
 * Univ. of California Berkeley, Xerox Palo Alto Research Center,
 * and Lawrence Berkeley National Lab.
 * November 15, 1997
 *
 */
#include "dsp_defs.h"

/*************************************************/
/* a function to solve AX = B using SuperLU library */
/*    (based on SuperLU_3.0\EXAMPLE\superlu.c)  */
/*************************************************/

void linsolve(int m, int n, int nnz, int nrhs, double *a, int *asub,
int *xa, double *rhs )
{
  SuperMatrix A, L, U, B;
  int    info, permc_spec;
  int    *perm_r; /* row permutations from partial pivoting */
  int    *perm_c; /* column permutation vector */
  superlu_options_t options;
  SuperLUStat_t stat;

  /* Create matrices A and B in the format expected by SuperLU. */

  dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, SLU_NC, SLU_D,
SLU_GE);

  dCreate_Dense_Matrix(&B, m, nrhs, rhs, m, SLU_DN, SLU_D, SLU_GE);

  if ( !(perm_r = intMalloc(m)) ) ABORT("Malloc fails for
perm_r[].");
  if ( !(perm_c = intMalloc(n)) ) ABORT("Malloc fails for
perm_c[].");

  /* Set the default input options. */
  set_default_options(&options);
  options.ColPerm = NATURAL;

  /* Initialize the statistics variables. */
  StatInit(&stat);

  dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);

  /* De-allocate storage */
  SUPERLU_FREE (rhs);
  SUPERLU_FREE (perm_r);
  SUPERLU_FREE (perm_c);
  Destroy_CompCol_Matrix(&A);
  Destroy_SuperMatrix_Store(&B);
  Destroy_SuperNode_Matrix(&L);
  Destroy_CompCol_Matrix(&U);
  StatFree(&stat);
}
```

It is the function linsolve( ) that solves the matrix equation $A.X = B$ by calling other object modules of the SuperLU library; from these two C/C++ source codes, object files must be generated using suitable makefiles.

The matrices are expected to be in a special format called "column-compressed storage format," so as to minimize storage space. Thus, the $Nx * Nx$ matrix elements of A need not all be specified, since only a small number, *nnz*, of them are nonzero; here Nx is the number of spatial grid points. The matrix A is specified through three row matrices `amat` and `asubmat` (that have a length *nn*), and `xamat` (that has a length $xasize = Nx + 1$). Our function takes these integers Nx, *nnz*, and *xasize* as parameters; in addition, we must pass as parameters the time step dt and the solution vector of the PDE at time t; the function then returns the solution vector at time $t + d\,t$.

The function `linsolvepp` must now be defined as an external procedure using the following command.

```
In[7]:= linsolvepp[ nx_, nx1_, nx2_, one_,
          expamat_, expasubmat_, expxamat_, exprhsmat_] :=
        ExternalProcedure[nx, nx1, nx2, one, expamat,
         expasubmat, expxamat, InOut exprhsmat];
```

Note the keyword `InOut` preceding the last argument of `ExternalProcedure`: in the calling function `SolveDiffusion1D`, the array `rhsmat` is passed to `linsolvepp` as input, but `linsolvepp` also returns the solution vector by destroying `rhsmat` and using it to store the solution vector. As a result, the array `rhsmat` is both an input and an output. The way to declare this is by using the keyword `InOut`.

```
In[8]:= EndPackage[];
```

We next declare the types, and then build and install.

```
In[9]:= Declare[SolveDiffusion1D[Integer Nx_, Real dt_, Integer nnz_,
          Integer xasize_, Real[_] U_] → Real [Nx], {Integer, Integer,
         Real, Integer, Real[Nx], Integer[nnz], Integer[nnz],
         Real[nnz], Real[nnz], Integer[nnz], Integer[xasize]}];

In[10]:= Declare[
          linsolvepp[Integer nx_, Integer nx1_, Integer nx2_,
            Integer one_, Real[_] expamat_, Integer[_] expasubmat_,
            Integer[_] expxamat_, Real[_] exprhsmat_] → Real[nx];
         ];

In[11]:= CompilePackage["foo`"];
```

```
Successful compilation to C++: 2 function(s)
```

We now create the executable. Note that an additional option `NeedsExternalLibrary` must also be specified in this example, since the external objects depend on other objects of the SuperLU library.

```
In[12]:= MakeBinary["foo'", NeedsExternalObjectModule →
           {$MCRoot <> "\Demos\ExternalFunction\linsolve", $MCRoot <>
             "\Demos\ExternalFunction\linsolvepp"}, NeedsExternalLibrary →
           {$MCRoot <> "/PDESOLVER/MathPDE2/SuperLU_3.0/superlu_cygwin.a",
            $MCRoot <> "/PDESOLVER/MathPDE2/SuperLU_3.0/blas_cygwin.a"}];

In[13]:= InstallCode[];
```

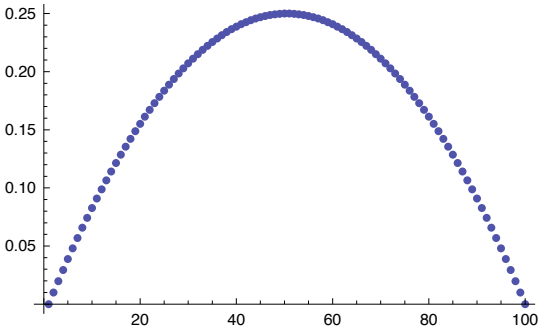
```
foo is installed.
```

We take the following initial conditions.

```
In[14]:= soln = Table[(x - 1) * (-x + 100) / (99.0 * 99.0), {x, 1, 100}];
```

Now the following command runs the C++ executable fooml.exe. We evolve from $t = 0$ to $t = 1000\,dt$ with $dt = 0.00001$.

```
In[15]:= Timing[
          Do[soln = SolveDiffusion1D[100, 0.00001, 100, 101, soln], {1000}];]

Out[15]= {6.8 Second, Null}

In[16]:= soln // ListPlot
```



■ 5. Summary and Conclusions

MathCode is an application package that generates optimized Fortran/C++ code for numerical computations. The code can be either compiled and run from within a notebook environment, or ported, and typically runs several hundred times faster than original *Mathematica* code.

MathCode is easy to use, since only the following three simple steps are involved for most applications:

- Add type declarations.
- Execute `BuildCode[ ]` to generate C++ code and an executable program.
- Execute `InstallCode[ ]` to connect the executable program to *Mathematica*.

It must be remembered that only a subset of *Mathematica* functions and operations are translated into C++ by *MathCode*. However, *MathCode* also provides these ways to extend the subset:

- Symbolic evaluation
- Callbacks to *Mathematica*
- Use of external code

To conclude, we remark that *MathCode* can turn *Mathematica* into a powerful environment for prototyping advanced numerical algorithms and production code development. Since it can generate stand-alone code, applications that use *Mathematica* as an environment for development and need to automatically generate efficient C++ code as embedded code in large software systems can greatly benefit.

MathCode is a product available both for purchase and free trial (see the website of MathCore Engineering, [1]). Currently, both the C++ and Fortran 90 versions of the code generator are available.

■ References

[1] P. Fritzson, *MathCode C++*, Linköping, Sweden: MathCore Engineering AB, 1998 www.mathcore.com.

[2] C. F. Gerald and P. O. Wheatley, *Applied Numerical Analysis,* 5th ed., Reading, MA: Addison-Wesley, 1994.

[3] The *SuperLU* package is available for download at crd.lbl.gov/~xiaoye/SuperLU.

[4] L. Viklund, J. Herber, and P. Fritzson. "The implementation of ObjectMath—A High-Level Programming Environment for Scientific Computing," in *Compiler Construction, Proceedings of the Fourth International Workshop on Compiler Construction (CC 1992)*, Paderborn, Germany (U. Kastens and P. Pfahler, eds.), *Lecture Notes in Computer Science*, **641**, London: Springer-Verlag, 1992 pp. 312-318. Also see the ObjectMath home page, www.ida.liu.se/~pelab/omath.

[5] P. Fritzson, V. Engelson, and L. Viklund, "Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment," in *Proceedings of the International Symposium on Design and Implementation of Symbolic Computaton Systems (DISCO 1993)*, Gmunden, Austria (A. Miola, ed.), *Lecture Notes in Computer Science*, **722**, London: Springer-Verlag, 1993 pp. 145-160. Also see the ObjectMath home page, www.ida.liu.se/~pelab/omath.

[6] K. Sheshadri and P. Fritzson, "*MathPDE*: A Package to Solve PDEs," submitted to *The Mathematica Journal*, 2005.

P. Fritzson, V. Engelson, and K. Sheshadri, "MathCode: A System for C++ or Fortran Code Generation from Mathematica," *The Mathematica Journal*, 2011.
dx.doi.org/doi:10.3888/tmj.10.4–7.

About the Authors

Peter Fritzson, Ph.D., is Director of research at MathCore Engineering AB. He is also a full professor at Linköping University and Director of the Programming Environment Laboratory (PELAB) at the Department of Computer and Information Science, Linköping University, Sweden. He initiated the development of *MathCode* and the ObjectMath environment, and developed parts of the first version of *MathCode*. Professor Fritzson is regarded as one of the leading authorities on the subject of object-oriented mathematical modeling languages, and has recently published a book, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press. He currently holds positions as Chairman of the Scandinavian Simulation Society, Secretary of EuroSim, and Vice Chairman of the Modelica Association, an organization he helped to establish. Professor Fritzson has published 10 books/proceedings and approximately 170 scientific papers.

Vadim Engelson, Ph.D., is a development engineer at MathCore Engineering AB. He was previously an assistant professor at Linköping University. Dr. Engelson received his Ph.D. in computer science at Linköping University in 2000 for his research in tools for design, interactive simulation, and visualization of object-oriented models in scientific computing. He has participated in several Swedish and European projects. Dr. Engelson is Technical Coordinator of the Scandinavian Simulation Society. He developed most of the *MathCode Mathematica* to C++ translator and several parts of *MathModelica*. He is the author of 21 publications.

Krishnamurthy Sheshadri, Ph.D., is a development engineer at *Connexios Life Sciences Private Limited*. He was previously a post-doctoral student at PELAB, Linköping University, where he used the *MathCode* system for advanced applications. Work on the project was done when Sheshadri was associated with Modeling and Simulation Research, Bangalore, India and Linköping University.

Peter Fritzson
Vadim Engelson
Mathcore Engineering AB
Teknikringen 1B
SE 583 30 Linköping, Sweden
peter.fritzson@mathcore.com
vadim.engelson@mathcore.com

Krishnamurthy Sheshadri
Connexios Life Sciences Private Limited
49, Shilpa Vidya
First Main Road, J P Nagar 3rd Phase
Bangalor-560 078, India
kshesh@gmail.com