MathCode C++ Release Notes

Version 1.3

1 November 2005

Troubleshooting and support

Always use the latest MathCode version available.

If **Demos/SimpleststExample/SimplestExample.nb** fails then all other compilation attempts will fail as well. Execute the notebook **Demos/Troubleshoot/MathCodeSetupVerify.nb** . It covers typical troubleshooting situations.

Send the whole notebook to support@mathcore.com .

In case of compilation errors that appear in your example please send us whole notebook.

List of supported *Mathematica* functions is available at
http://www.mathcore.com/products/mathcode/mathcodec++_subset.pdf

New in 1.3

This version includes possibility to work with **Complex** numbers and lists of **Complex** numbers. Complex numbers and lists of them can be passed as arguments to the most of numerical functions. The full list of these functions can be found in Appendix A of the User Manual (section A.2.9). Note that updated version of the manual in pdf-format can be found in MathCode installation.

This version includes several extra Demo examples which illustrate numerical integration and interpolation.

This versions includes a configurator in order to switch easily between several C++ compilers available on Windows.

Fortran code generation is available as a separate product, MathCode F90.

Configuring environment variables for C++ compiler.

Visual C++ should be configured for "command line compilation" for MathCode to be able to call it. Normally at the end of Visual C++ compiler installation you will be presented with a dialog "Setup Environment Variables". Check the box "Register Environment Variables" and finish the installation.

Starting *Mathematica* using a BAT file.

If for some reason you do not configure your C++ complier this way, you have to modify the way you start *Mathematica*. Create a BAT file with two lines: In the first line you call the BAT file which sets environment variables of your C++ compiler (e.g. **vcvars32.bat**). In the second line you start *Mathematica*.exe.

New in 1.2

This version includes the following important capabilities: Fortran90 code generation and compatibility with *Mathematica* 5.0

Fortran90 code generation

Fortran90 code generation is a new feature of MathCode. In order to use it you first install Digital Visual Fortran 6.0 and MicroSoft Visual C++ 6.0. They both must be installed such way that necessary environment variables are set in the System properties / Environment.

It means that **df** and **cl** are availale in the **PATH**, as well as **LIB** and **INCLUDE** are set correctly

The dialog box allowing to do this automatically appears at the very end of setup of Digital Visual Fortran 6.0 and MicroSoft Visual C++ 6.0.

There exist a directory DemosF90 with several demo examples of functions compiled to Fortran90.

The *Mathematica* language implemented in the *Mathematica* to Fortran90 compiler is identical to the subset implemented in *Mathematica* to C++ compiler, except of External Functions. These are not implemented yet.

By default C++ code is generated. In order to activate Fortran90 code generation add the rule in SetCompilationOptions:

```
SetCompilationOptions[Language → "Fortran90"];
```

Availablility of Fortran90 code generator depends on your MathCode license.

Mathematica 5.0 compatibility

MathCode 1.2 is compatible with Mathematica 5.0, 5.1, 5.2. Several routines had to be changed in order to provide this compatibility.

If you use XML package symbol **Declare** you should note that it is made non-available in MathCode due to names conflict. Do not load MathCode and XML simultaneously.

Solaris/UltraSparc vertsion

- The Solaris/UltraSPARC version contains the LightMat run-time library for sevearl different compliers. Please check that your compiler corresponds to specifications in MathCodeConfig.m

Bugfixes

- These major bugs (as well as several minor bugs) were fixed:
- ArcTan bug fixed
- Fortran stub for 3D ,4D arrays (for external Fortran functions)fixed
- Spell (octal digits) bug fixed
- Missing compiler messages (when using mingw32) bug fixed

MathCode Compiler messages

Messages from the compiler are often pretty long and get shortened by *Mathematica* using **Short[]**. This shortening can be prevented by doing this:

```
$MessagePrePrint = Identity;
```

This will make **Message[]** always print the full text of messages.

New in 1.1

The big addition to 1.1 is the inclusion of the Free C++ compiler EGCS in the Windows version. Other items are some bugfixes, a new license verification method, the users guide as *Mathematica* notebooks, and a runtime configuration file to facilitate switching between different compilers.

Online manual in notebook format

Besides PostScript and PDF formats, MathCode now also comes with the users guide in notebook format. After installation, open *Mathematica* and select the Help/Rebuild Help Index menu item. Now you can use *Mathematica*'s help browser to browse the MathCode users guide.

Runtime configuration

There is now a file `MathCodeConfig.m` in the main MathCode directory. This file is really a *Mathematica* package that contains some configuration directives; currently **`DefineCompiler[]`** and **`DefaultCompiler[]`**.

`DefineCompiler[]` is used to associate a symbolic compiler name (a string) with a make file, a command template, and a build command. You don't normally need to bother with these details.

`DefaultCompiler[]` is used to select the default compiler definition for a language. Currently the only language supported for code generation is C++. In `MathCodeConfig.m` you will find a line

```
DefaultCompiler["C++" -> "mingw32"];
```

This tells MathCode to use the included "mingw32" compiler definition when generating C++ code. If you wish to use Visual C++ instead, you should change this to read:

```
DefaultCompiler["C++" -> "vc60"];
```

Using a different compiler can be easier than that, with the new options to **`CompilePackage[]`**, **`MakeBinary[]`** and **`BuildCode[]`**.

`CompilePackage[]` now takes a **`Language`** option (currently only C++ is supported). MathCode will then use the default compiler for the specified language. Example:

```
CompilePackage[Language -> "C++"];
```

MakeBinary[] now takes a Compiler option; the option value should be one of the symbolic names (strings) defined using DefineCompiler[]. The Compiler option to MakeBinary overrides the default compiler specified for the selected language. Example:

```
MakeBinary[Compiler -> "g++"];
```

As usual, **`BuildCode[]`** can be given both **`CompilePackage[]`** and **`MakeBinary[]`** options. The following example will generate C++ code and use the "CC" compiler to compile it, overriding any default specification:

```
BuildCode[Language -> "C++", Compiler -> "CC"];
```

The included compiler(s) (Windows only)

The Windows version of MathCode C++ now ships with the free EGCS compiler. EGCS stands for Experimental GNU Compiler System and is by far the most popular of the free compilers. EGCS contains a C compiler (gcc), a C++ compiler (g++) and a FORTRAN 77 compiler (g77). The included Windows port is the mingw32 port from Mehmet Khan's pages.

This table shows what is installed in the `MathCode\mingw32` directory during the MathCode installation process:

Files	Source (URL)
contents of `egcs-1.1.2-mingw32.zip`	`ftp://ftp.xraylith.wisc.edu/pub/khan/gnu-win32`
contents of `i386-mingw32-make.zip`	`ftp://ftp.xraylith.wisc.edu/pub/khan/gnu-wi`
`libml32i1.a`	Produced by MathCore from `ml32i1m.lib` of the *Mathe*

The original files can also be found in the GNU subdirectory of the MathCode CD.

More information about EGCS can be found in the following locations:

- http://www.xraylith.wisc.edu/~khan/software/gnu-win32/
- http://sourceware.cygnus.com/cygwin

More information about free software in general can be found at http://www.gnu.org/fsf/

About license verification

License information is distributed by MathCore in the form of key files, containing encoded license data. To create a key file, MathCore needs the MathID (the value of the *Mathematica* variable **`$MachineID`**) for the machine the license is intended for. The normal procedure is to distribute key files as e-mail attachments. To install a new license, simply save the attached file to the `Licensing` subdirectory of your MathCode installation. Any number of key files can be put in this directory; MathCode automatically updates an index file (`index.m`) keeping track of machine identities and key files. On a site/network installation, this file and the `Licensing` subdirectory needs to be world writable if normal users should be allowed to install license files.

You also have the opportunity to select a key file during the installation process. Obtaining a key file for your machine before installing MathCode thus makes the process a bit smoother.

Bugfixes

MathCode used to produce strange errors when the **`*^`** notation for very large or very small numbers were used. This has been fixed in version 1.1.

Some people were confused when MathCode reported two generated functions when they had only defined one. The extra function is of course the *packagename*_init function. This function is now excluded from the function count.

`CompilePackage[]` without an argument and with no active package compiles the `Global`` context.

Other changes

InterpolatingFunctionObjects has changed name to **InterpolatingFunctionNames**. This is more appropriate as it is the names of the objects that are given, not the objects themselves. We feel that the MathCode support for interpolating function objects hasn't been around long enough for this change to be a big compatibility issue.

New in 1.03

Most of the improvements in 1.03 are in the Windows 95/98/NT installation procedure, which has undergone a major overhaul. The only added functionality in 1.03 is the definition of **Erf** in System.nb.

New in 1.02

The functional addition to version 1.02 is the ability to compile interpolating function objects. The use of interpolating function objects is described in a new demo notebook.

There are also minor updates and bugfixes in serveral areas

- Some combinations of input/output/inout arguments to external functions did not work properly.
- Some combinations of arguments to callback functions did not work properly.
- Paths with spaces were not correctly handled in Windows (they are not allowed in MathCode for UNIX).
- Visual C++ 6.0 doesn't allow extra WinMain() functions; MathCode has been updated accordingly.

Caveats

Configuring Visual C++

Visual C++ must be configured for "command line compilation" for MathCode to be able to call it. Here's how to configure it.

■ Windows NT

Re-run the Visual C++ installation (only missing or corrupt files will be updated). At the end you will be presented with a dialog "Setup Environment Variables". Check the box "Register Environment Variables" and finish the installation.

■ Windows 95/98

Under Windows 95/98, the environment variables necessary for command line compilation are installed using a batch file. The trick is to make the command prompt execute the batch file whenever it is opened.

1. Locate the file "`command.pif`" e.g. using "Find/Files or Folders..." on the Start menu.

2. right-click on the file and select "properties" from the menu.

3. Under the "Program" tab, in the "Batch file" box, enter the path to your `VCVARS32.BAT` file, in quotes. Example:

```
"C:\Program Files\Microsoft Visual Studio\Vc98\Bin\VCVARS32.BAT"
```

The quotes are necessary if the path contains any spaces.

4. Under the "Memory" tab, select "1024" from the drop-down box named "Initial Environment".

5. Save these changes to `command.pif`.

C++ Compiler warnings

We try to keep MathCode compatible with many C++ compilers, but sometimes an error or two creeps in. In Windows you might see the following message during **`MakeBinary`** or **`BuildCode`**:

```
Ccompiler::warning :
 C/C++ compiler:LINK : warning LNK4089: all references to
    "GDI32.dll" discarded by /OPT:REF
```

This message can be safely ignored. If you find it annoying, it can be suppressed by editing the file `MathCode\System\compwin.mak`. Look for this line:

```
LINKFLAGS = $(LINKOPT) /LIBPATH:"$(MATHLINK)\lib"
```

Comment out this line by inserting a dash (#) as the first character. Below this line you can se a commented out line that also defines the `LINKFLAGS` variable with some additions:

```
#LINKFLAGS =
 $(LINKOPT) /LIBPATH:"$(MATHLINK)\lib" /include:__imp__GetStockObject@4
```

Remove the dash in front to use this definition. This will prevent the message on our computers at MathCore; probably for you too.

MathCode Compiler messages

Messages from the compiler are often pretty long and get shortened by *Mathematica* using **`Short[]`**. This shortening can be prevented by doing this:

```
$MessagePrePrint = Identity;
```

This will make **`Message[]`** always print the full text of messages.

Recommendations

The interactive environment in *Mathematica* supports very flexible and powerful ways to develop, test, and with MathCode produce executable code. However, it is a good rule to take some precaution of how to evaluate and redefine the declarations and functions that are intended for the code generation.

The normal rules of how to define and evaluate *Mathematica* packages should be used. Encapsulate the code into BeginPackage and EndPackage. Let all "exported" symbol be created outside the private section (marked by Begin["`Private`"] and End[]), and let the function definitions be inside the private section. NOTE! When redefining a function in a package it is a good rule to always evaluate the complete package to prevent symbols from being created in the wrong context. To make a fresh definition of the a package symbol or function, use Remove to delete the created symbol and definition from *Mathematica*.

If the *Mathematica* notebook is used to write and test the package code, problems with contexts may occur. The reason for this is that the notebook parser sets the context for all symbols in an input cell before evaluating if the input cell contains a single expression. This means that if

```
Begin["foo`"]; fee
```

is evaluated then the context for symbol fee will not be changed. The solution to this is to separate all Begin, End, BeginPackage, and EndPackage in separate cells.

There is one exception from the concept of packages, for functions defined "directly" into the default context "Global`". In this case all symbols used will be put into the same context. All definitions made in the "Global´" context can be compiled to the "Global" package by using CompilePackage["Global`"].

Installing the compiled executables by using InstallCode, will store the *Mathematica* version of the functions into the container SourceDownValues. This container will not change if the functions are redefined in *Mathematica* while the corresponding executable is installed. When using UninstallCode to uninstall the executable, all the functions in the package will be restored to the definitions they had before InstallCode was used. All changes in these definitions during the period when the executable was installed, are deleted from *Mathematica*. However, the internal MathCode tables are not affected, which leads to discrepancies between the *Mathematica* definitions and definitions stored by MathCode. The following rule is therefore recommended:

Do UnistallCode[] before redefining the source.

Release Information (updated for 1.1)

Variable names

`dim` is not an allowed variable name in MathCode, as this creates conflicts with internal MathCode identifiers.

Increments, Decrement, PreIncrement, PreDecrement

The occurrences of i++, ++i, i--, and --i are translated to set statements as i=i+1, and i=i-1. Note that this transformation is only valid when Increments, Decrement, PreIncrement, and PreDecrement are used as statements since the behavior of return values from this operations are not preserved as in *Mathematica*.

Strings

Only 7-bit ascii characters are supported in strings. This implies that international characters and other special symbols like Å are cannot be converted to C++.

Blocks

The blocks Module, With, and Block can be nested and variables can be declared at different levels and with different scope. However, declarations of local variables in a nested block must be independent of prior execution of code in the block enclosing the block. For example

```
Module[{x = a, y = c},
        x = 3;
     Module[{y = d}, x = y]
]
```

is converted to

```
Module[{x = a, y = c, y$xx = d},
        x = 3;
        x = y$$xx
    ]
```

This will work fine, but for the following

```
Module[{x = a, y = c},
        x = 3;
    d = x;
     Module[{y = d}, x = y]
]
```

the conversion will fail since the resulting code is then

```
Module[{x = a, y = c, y$xx = d},
        x = 3;
        d = x;
        x = y$$xx
    ]
```

Note! The same rule is true for variable used as dimension specifiers for declarations of local arrays, e.g.

```
Module[{x = a, n = c},
        n = 3;
     Module[{Integer[n]@ y = d}, x = y]
]
```

Round

Converting reals to integers can be done by using Round. Round will be converted to the standard C function irint. Unfortunately this is not always the same conversion as Mathematica Round for half way numbers, where Round is converting to the nearest even number. How the conversion is done is dependent on the C-compiler used.

The following function can be used to test the floating point conversion.

```
irinttest[Real x_] -> Integer := Round[x];
```

ArcTan

The two input argument version of ArcTan[x_,y_] is not supported for matrix input arguments.

Inline and Range Check flags

By setting the option

```
MakeBinary[InlineFlag->True]
```

the C++ compiler will make a inline recompilation of the matrix library LightMat located at lib/lightmat. This recompilation is necessary to make the options RangeCheckFlag significant.

If

```
MakeBinary[InlineFlag->False, RangeCheckFlag->False
```

is executed the RangeCheckFlag will be obsolete since the precompiled version of LightMat is used in this case.

Iterators, Do, Table, Sum, Product, Array

■ Zero Step Size

Using a zero step size in an iterator specification like

```
Table[i, {i, start, stop, 0}]
```

will in *Mathematica* generate an error.

For the generated code this will give an infinite loop, since this case is not checked for at runtime.

■ Table

The code generation for Table is only possible when the Table is placed directly to an assignment of a declared variable, like

```
var = Table[i, {i, 1, 3}]
```

This makes it possible to translate as a loop that assigns each element of the variable. For the case above the corresponding *Mathematica* code is

```
setshape[var, 3];
Do[
        var[[i]] = i, {i, 1, 3}
]
```

The function `setshape` is an internal MathCode function that sets the size or dimensions of the data structure corresponding to `var` before the loop is executed.

Remark

Using iterators that is not iterated at all will in *Mathematica* produce empty lists at some specific level. If the resulting table has 2 or more levels and one or more of its dimensions are zero, this will not be treated properly by *MathLink*.

By externally executing the following:

```
Table[i, {3}, {i, 1, 3, -1}]
```

will return

```
$Aborted
```

by *MathLink*. See section "Zero Dimensions and *MathLink*".

MatrixExport

Code with embedded type specifications as e.g

```
foo[Real@ x_]->Real
```

needs to load "System/MathCodeLanguage" to work as expected, since the type syntax is not a valid syntax for *Mathematica* alone.

An alternative is to use Declare as

```
Declare[foo[Real x_]->Real]
```

to provide MathCode with the necessary type information. The advantage with this methods is that the code will also work in *Mathematica* without any preloaded packages. The Declare statement will evaluated to Null and ignored.

To distribute files written with MathCode embedded type syntax, please contact MathCore AB to download a free version of the package "MathCodeLanguage`".

MathLink

■ Zero Dimensions and MathLink

When using list and tensor constructors like Table and Array it is possible to produce tensors with one or more zero dimensions.

```
zten = Table[i + j, {i, 1, 3}, {j, 1, 3, -1}]
```

```
{{}, {}, {}}
```

```
Dimensions[zten]
```

```
{3, 0}
```

For the moment variables from Table and Array are declared as real or integer lists, matrices, or tensors where the maximum number of dimensions supported by MathCode is 4. Tensors with dimensions from 2 - 4 are pasted through *MathLink* using the functions `MLPutIntegerArray` and `MLPutRealArray`. These function cannot handle zero dimensions which implies that if a zero dimension tensor is sent, the symbol `$Aborted` will appear as output in *Mathematica*.

For one dimensional lists the *MathLink* functions MLPutRealList and MLPutIntegerList are used instead which can handle lists of zero length. Therefore Table and Array constructors returning one dimensional empty lists will work correctly.