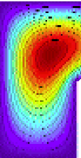


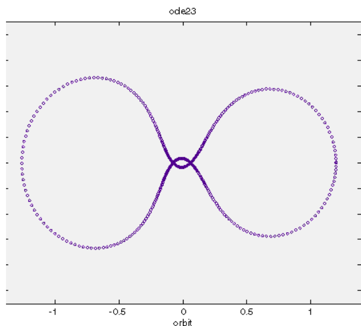
Performance of the MATLAB Compiler

Speedups by two orders of magnitude are possible

by Cleve Moler

The primary objective of MCC, the new MATLAB compiler, is to make MATLAB programs run faster. Without the compiler, MATLAB is an interpreted computing environment with dynamic storage allocation. Compiling programs eliminates the interpretive overhead and, more importantly, provides faster storage management.

MCC translates function M-files into equivalent functions in the C language. These functions are then processed by the C compiler for a particular machine to produce either MEX-files for use in MATLAB or external functions for use in other programs outside of MATLAB.



Three-body orbit.

The distinguishing feature of MATLAB as a programming language is its extensive collection of operations and functions involving matrices. The matrix, a two-dimensional rectangular array of real or complex numbers, is the language's only data type. There are no declarations, dimension statements, or separate storage allocation statements. Sizes are determined automatically and dynamically from context. It is impossible to get an error message from MATLAB saying that a particular use of a variable is inconsistent with its declaration. The C language, on the other hand, has types and declarations. It is important to distinguish between integers, floating point values, and characters, and between scalars and arrays.

MCC's most significant feature, one which took us two years to perfect, is its type inference capability. This involves the detection of various instances of general matrices that can be represented in less space and processed in less time. For example, it is easy to translate the MATLAB `for` statement

```
for i = 1:n
```

into the equivalent C. The tricky part is recognizing that the C program can declare `i` to be an integer.

```
int i;
for (i = 1; i <= n; i++)
```

The declaration is possible in this particular situation because both the starting value and the increment of the `for` loop are one, so the `i` takes on integer values. (Note that it is not possible to infer from the `for` statement by itself that `n` is an integer.)

```
function x = tridi(a,b,c,d)
% TRIDI(a,b,c,d) solves Tx = d where
% T = diag(a,-1) + diag(b,0) + diag(c,1)
n = length(b);
x = zeros(n,1);
for k = 1:n-1
    p = a(k)/b(k);
    b(k+1) = b(k+1) - p*c(k);
    d(k+1) = d(k+1) - p*d(k);
end
x(n) = d(n)/b(n);
for k = n-1:-1:1
    x(k) = (d(k) - c(k)*x(k+1))/b(k);
end
```

It is important to realize that compilation will not substantially speed up any function that spends most of its time in the built-in indexing, math, and graphics functions of MATLAB. A useful rule-of-thumb is that the execution time of a MATLAB function is proportional to the number of statements executed, no matter what those statements actually do. This is clearly a very rough approximation, but it serves to emphasize the point that the only reason to compile many functions is to encapsulate the code, not to speed them up.

The functions that profit most from compilation involve nested `for` loops driving a body of substantially scalar code. Our first example involves a function which solves a tridiagonal system of linear equations

$$\begin{aligned} b_1x_1 + c_1x_2 &= d_1 \\ a_1x_1 + b_2x_2 + c_2x_3 &= d_2 \\ a_2x_2 + b_3x_3 + c_3x_4 &= d_3 \\ &\dots \\ a_{n-1}x_{n-1} + b_nx_n &= d_n \end{aligned}$$

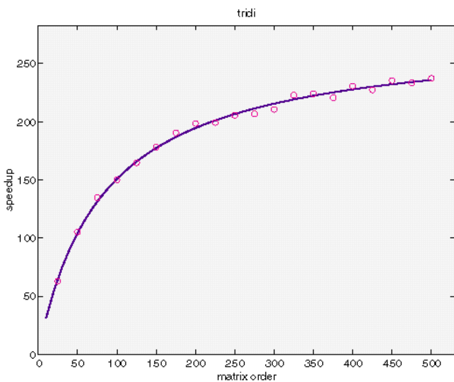
The input consists of four vectors, `a`, `b`, `c` and `d`, which specify the diagonal, superdiagonal, and subdiagonal of a tridiagonal matrix, together with the right hand side of the linear system. The output is another vector, `x`, the solution to the system. Both the execution time and the storage requirements should be proportional to the number of unknowns, n. It is very inefficient to use a full matrix representation because the storage requirements would be proportional to n^2 and the execution time proportional to n^3.

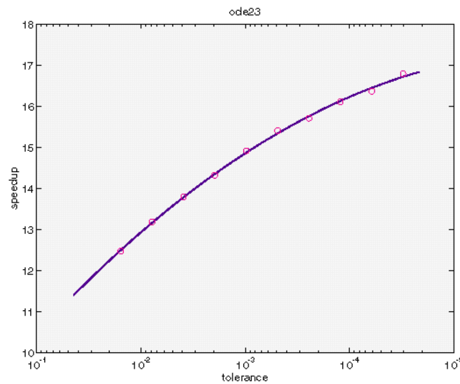
A sparse matrix representation has the proper storage and time complexity, but is not tuned to the tridiagonal special case.

The function shown in the box handles the most frequent situation, where it is known *a priori* that pivoting is not required because the matrix is diagonally dominant or positive definite. Even though this is a computation involving vectors, it cannot be "vectorized." The individual components must be modified or computed one at a time, using components computed in the previous step.

In our timing experiments, we used a 75 MHz Pentium laptop and varied the problem size, *n*. We measured the execution time of both `tridi.m` and the MEX-file produced by MCC. Theoretically, these times should be linear functions of *n*, because each operation is done either once or *n* times. Least squares fits to the measured values show this linear model is satisfactory.

Compilation of `tridi` speeds it up considerably. The dots on the top graph are the ratios of the actual execution times. The curve is the ratio of the two linear fits. We see a speedup of over two orders of magnitude for systems with more than about 50 unknowns. In fact, on this particular computer, the *asymptomatic speedup*, which is the limiting value of the quotient as *n* approaches infinity, is over 250. Comparable results are obtained on other machines.





The `tridi` example clearly shows MCC performance at its best. It is not a toy example, and speedups by two orders of magnitude can be expected from other functions that involve long loops around scalar operations.

Our second example involves a different kind of computation. A *function function* is a MATLAB function whose primary argument is the name of another function. Zero finders, minimizers, quadrature routines, and ordinary differential equation solvers all fall into this category. Our simplest differential equation solver, `ode23`, is already available as a hand-written MEX-file on some machines, but this experiment involves comparison with the original M-file.

The differential equation is a pair of nonlinear, second-order equations describing the *restricted three-body problem*, which models the orbit of a body, say, a satellite, under gravitational attraction from two much heavier bodies, say, the earth and the moon. With properly chosen initial conditions, the orbit is periodic. The satellite starts out on the far side of the moon, passes near the earth, continues in a big loop on the opposite side of the earth from the moon, passes near the earth again, and returns to its starting position and velocity. The graph of the orbit at the beginning of the column shows the steps required to obtain a certain accuracy. You can see that it is very important to have a variable step size, automatically chosen, for this problem.

One of the optional parameters for `ode23` is the tolerance or required accuracy. Since `ode23` employs a third-order formula, the number of steps taken and resulting execution time is roughly proportional to the cube root of the tolerance.

The differential equation is defined in a separate M-file, `orbit.m`. Its name is passed as a parameter to `ode23`.

```
[t,y] = ode23('orbit',0,tfinal,y0,tol);
```

One of the most time-consuming portions of the interpreted computation involves the processing of the string `'orbit'` and the call to the corresponding M-file. MCC provides a mechanism for compiling the two M-files, `ode23` and `orbit`, into a single MEX-file with a very efficient link between the two functions. The lower graph shows the resulting speedup as a function of varying tolerance. The compiled program runs between 12 and 18 times faster than the interpreted program. This is not as spectacular as our first example, but is certainly worthwhile.

How much speedup can be expected by compiling a "typical" M-file? That's impossible to answer—there is no typical M-file in this context. As we have seen, some functions speed up by factors of 18 or 250. Other functions, including those that spend most of their time in built-in subfunctions, will not speed up at all. Your own mileage will certainly vary—possibly by a couple of orders of magnitude. ■

Cleve Moler is chairman and co-founder of The MathWorks. His e-mail address is moler@mathworks.com