Best Practices for a MATLAB to C Workflow Using Real-Time Workshop

By Houman Zarrinkoub and Grant Martin

Embedded software developers have long relied on MATLAB® for algorithm design and prototyping and on C code for implementation on embedded processors and DSPs. As a high-level language, MATLAB facilitates design exploration. In contrast, programming in C is well suited to optimizing DSPs for performance, memory, and processing power. The challenge is to transition a design from the flexible development environment of MATLAB to the constrained programming style of C. The solution is automatic translation of MATLAB to embeddable C.

Products Used

- MATLAB®
- Fixed-Point Toolbox™
- Real-Time Workshop™
- Signal Processing Toolbox™

Manually translating MATLAB to C involves incorporating into the code low-level details such as data-type assignments, memory allocations, and optimizations for computational load and memory. A great deal of effort is required to ensure that the MATLAB code and the C code remain equivalent.

When your MATLAB algorithm uses the Embedded MATLAB™ language subset, the translation to C becomes unambiguous, enabling you to focus on refining your design rather than producing and verifying hand-written C code.

This article outlines the challenges involved in the manual translation from MATLAB to C, demonstrates how to use the Embedded MATLAB subset for automatic translation, and provides best practices for coding your MATLAB algorithm to improve the generated C code.

Challenges of Translating MATLAB Concept Code into Implementation Code

MATLAB has several advantages for design exploration, such as polymorphism, matrix-based functions, and an interactive programming environment. During translation of an algorithm from MATLAB to C, however, software designers face some important constraints. For example:

MATLAB is a dynamically typed and C is a statically typed language. When writing a MATLAB program, you do not need to define data types and sizes for your variables. While this flexibility makes it easy to develop algorithms as proofs of concept, when it comes time for translation to C, the programmer must assign appropriate data types and sizes to all variables.

MATLAB is polymorphic. Functions in MATLAB can process different types of input parameters and can apply a different algorithm to each type of parameter. For example, the `abs` function computes the absolute value of real numbers and norm of complex numbers and can process scalars, vectors, or matrices.

```
>> abs(4-3i)
ans =
     5
>> abs([4 -3])
ans =
     4     3
```

This kind of flexibility is not supported in C, which assigns a single algorithm to each parameter type. To translate a polymorphic MATLAB function to C, the programmer must maintain separate function prototype for each possible parameter signature.

MATLAB is based on compact matrix notation. Most MATLAB expressions containing vectors and matrices are compact, single-line expressions similar to the corresponding mathematical formula. The equivalent C code requires iterators, such as `for` loops, to express the matrix operations as a sequence of scalar computations.

Automating the MATLAB to C Workflow

Automatic MATLAB to C conversion with Real-Time Workshop® addresses many of the challenges outlined in the previous section. For example, consider an algorithm depicted in the function `euclidean.m`. This algorithm minimizes the Euclidean distance between a column vector *x* and a collection of column vectors contained in the matrix *cb*. The function has two output variables: *y*, the vector in *cb* with the minimum distance to *x*, and *dist*, the value of the minimum distance (Figure 1).

In the body of the Euclidean function, we use the MATLAB function `norm` to compute the distance between *x* and each column vector of *cb*. To visualize the computed distances between any pair of points, we call the `plot_distances` function inside the loop. We use the `%#eml` directive to turn on the MATLAB M-Lint code analyzer and check the funtion code for errors and recommend corrections.

Figure 2 shows how the `plot_distances` function helps us visualize the process of computing all distances in a 2D space and finding the minimum value.

```
function plot_distances(x,cb,i)
% Plot a pair of points and
% a line segment connecting them
plot(x(1), x(2), 'or');
plot(cb(1,i), cb(2,i), 'xr');
line([x(1) cb(1,i)],[x(2) cb(2,i)]);
grid;
end
```

To generate C code from this algorithm, we must use only operators and functions that are part of the Embedded MATLAB subset. Visualization functions, such as plot, line, and grid, are not supported by the Embedded MATLAB subset. When you open the `euclidian.m` function in the MATLAB editor, the M-Lint code analyzer identifies and reports on these unsupported lines of code.

While it makes sense to use visualization functions to debug and verify the algorithm, when we implement the algorithm as C code, we must separate the offline analysis portions of the design from the online portions involved in embedded C code generation. In our example, we can make the algorithm compliant with the Embedded MATLAB subset simply by identifying and commenting out the `plot_ distances` function.

Now we use the `emlc` command in Real-Time Workshop to generate C code for the Embedded MATLAB compliant function `euclidean.m`.

Typical syntax for translation is

```
>> emlc -eg {x,cb} -report euclidean.m
```

The example option (following the `-eg` delimiter) sets the data types and dimensions of function variables by specifying an example at the function interface. The `-report` option opens the Embedded MATLAB compilation report with

```
function [y,dist] = euclidean(x,cb) %#eml
% Initialize minimum distance as first element of cb
idx=1;
dist=norm(x-cb(:,1));
% Find the vector in cb with minimum distance to x
for i=2:size(cb,2)
    d=norm(x-cb(:,i));
    if d < dist
        dist=d;
        idx=i;
    end
    plot_distances(x,cb,i);
end
% Output the minimum distance vector
y=cb(:,idx);
end
```

Figure 1. Source code for the `euclidean.m` *function.*

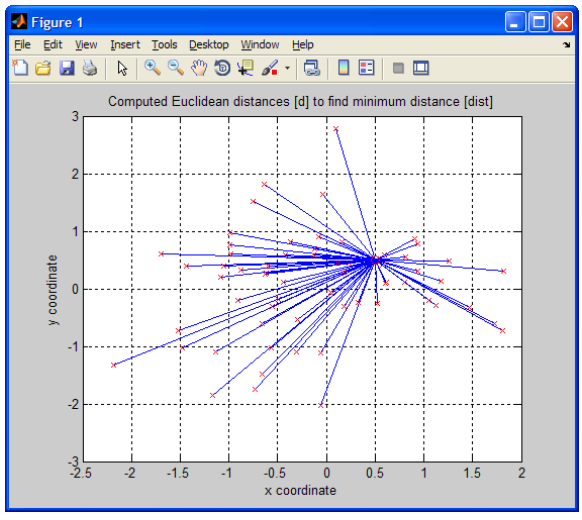

Figure 2. Visualizing the computation of Euclidean distances by the `plot_distances` *function.*

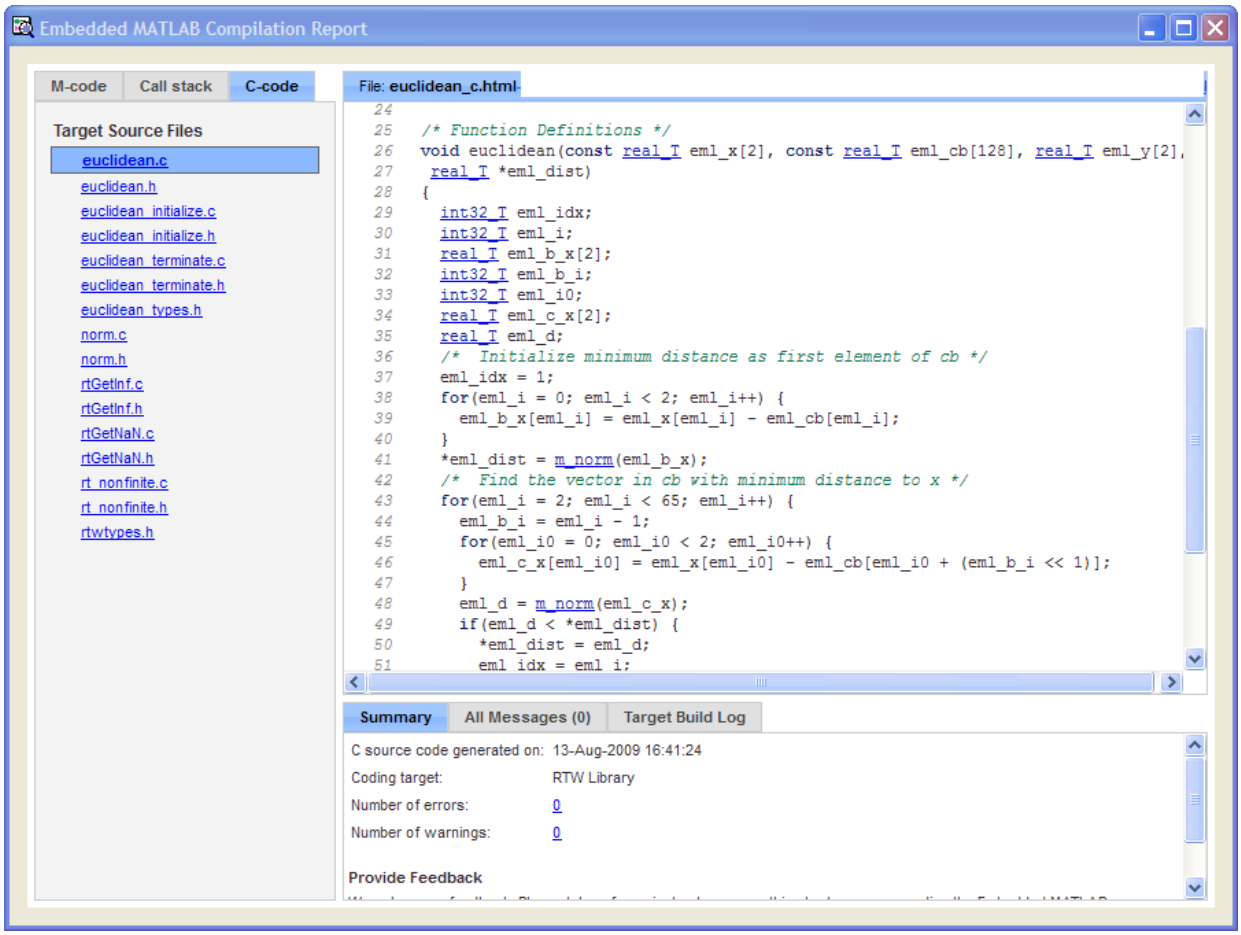

Figure 3. C code generated automatically from the `euclidean.m` *function displayed inside the Embedded MATLAB compilation report.*

hyperlinks to C source files and header files generated from the MATLAB function. The generated C code is created in a file named `euclidean.c` (Figure 3).

By using the example option, we declare the data type, size, and complexity of the variables *x* and *cb* in the function interface, enabling the Embedded MATLAB engine to assign data type and sizes automatically to all the local variable in the generated C program. The generated C code correctly maps to zero-based indexing for accessing array elements, and the vector operations are automatically mapped to scalar computations with `for` loops. As a result, many difficulties encountered in manual MATLAB to C conversion are eliminated through automatic translation.

Design Patterns for a MATLAB to C Workflow

In an embedded system, the size and data type of each variable must be set before implementation. In addition, if the performance requirements are not met, the algorithm's memory and computational footprint must be optimized. The following sections examine design patterns that use supported Embedded MATLAB features to ensure that the generated C code adheres to these requirements.

Accommodating Changes in Variable Dimensions

In the MATLAB language, all data can vary in size. Embedded MATLAB supports variable-sized arrays and matrices with known upper bounds. This means you can accommodate the use of variable-sized data for embedded implementations by using buffers of a constant maximum size and by addressing subportions of the constant-size buffers. Within your Embedded MATLAB functions you can define variable-size inputs, outputs, and local variables, with known upper bounds. For inputs and outputs, you must specify the upper bounds explicitly at the function interface. For local data,

Embedded MATLAB uses in-depth analysis to calculate the upper bounds at compile time. However, when the analysis fails to detect an accurate upper bound, you must specify them explicitly for local variables.

We update the `euclidean.m` function to accommodate changes in dimensions over which we compute the distances. We want to compute only the distance between first N elements of a given vector x with the first N elements of every column vector contained in the matrix cb. The resulting function, `euclidean_varsize.m`, will have a third input argument, *N* (Figure 4).

The compilation of this function will result in errors because we have not yet specified an upper bound for the value of N. As a result, the local variable *1:N* will have no specified upper bound. To impose the upper bound we can constrain the value of the parameter N in the first line of the function by using the `assert` function with relational operators (Figure 5).

The function `varsize_example.m` shows another common pattern, one where an array such as Y is first initialized and then grows in size based on a condition related to the value of an input or local variable:

```
function Y = varsize_example(u) %#eml
Y = [1 2 3 4 5];
if (u > 0)
    Y = [Y u];
else
    Y = [Y u -u];
end
```

The compilation of this function will again result in errors since we have not specified an upper bound for the variable Y. To accommodate this type of size change for the local variable Y, we can specify the upper bound using the `eml.varsize` function for all instances of that local variable. In this example we constrain a maximum

```
function [y,dist] = euclidean_optimized(x,cb) %#eml
% Initialize minimum distance as first element of cb
idx=1;
dist=sum((x-cb(:,1)).^2);
% Find the vector in cb with minimum distance to x
for i=2:size(cb,2)
    d=sum((x-cb(:,i)).^2);
    if d < dist
        dist=d;
        idx=i;
    end
    %plot_distances(x,cb,i);
end
% Output the minimum distance vector
y=cb(:,idx);
end
```

Figure 4. Input argument for `euclidean_varsize.m`

```
function [y,dist] = euclidean_varsize(x,cb,N) %#eml
% Constrain the dimension to be less than 5
assert(N<=5);

% Initialize minimum distance as first element of cb
idx=1;
dist=norm(x(1:N)-cb(1:N,1));
% Find the vector in cb with minimum distance to x
for i=2:size(cb,2)
    d=norm(x(1:N)-cb(1:N,i));
    if d < dist
        dist=d;
        idx=i;
    end
end
% Output the minimum distance vector
y=cb(1:N,idx);
end
```

Figure 5. Using the `assert` *function with relational operators.*

dimension of 1-by-8 for the variable *Y*. In the upper branch of the `if` statement, the variable Y has a dimension of 1-by-6 and in the lower branch, a dimension of 1-by-7. The resulting function will be

```
function Y = varsize_example(u) %#eml
eml.varsize('Y', [1 8]);
Y = [1 2 3 4 5];
if (u > 0)
    Y = [Y u];
else
    Y = [Y u -u];
end
```

Optimizing for Memory or Computational Complexity

Another common refinement is to optimize the generated C code for memory and complexity. In our `euclidean.m` algorithm, to compute the distance between two points, `norm` takes the square root of the squared values of each element of a given vector. Because computing the square root is computationally expensive, we can update our function by computing only the sum of the squared elements, without any loss in the intended behavior. The resulting function, which has a much lower computational load, uses the `sum` function supported by the Embedded MATLAB language subset (Figure 6).

It may be desirable to reduce memory footprint of the generated C code. In some cases, the initialization of new variables in your Embedded MATLAB function may produce redundant copies in the generated C code. Although Embedded MATLAB technology eliminates many copies automatically, you can eliminate data copies that are not automatically handled by declaring uninitialized variables using the `eml.nullcopy` function. In Figure 7, the variable *Y* is initialized with such a construction.

```
function [y,dist] = euclidean_optimized(x,cb) %#eml
% Initialize minimum distance as first element of cb
idx=1;
dist=sum((x-cb(:,1)).^2);
% Find the vector in cb with minimum distance to x
for i=2:size(cb,2)
    d=sum((x-cb(:,i)).^2);
    if d < dist
        dist=d;
        idx=i;
    end
    %plot_distances(x,cb,i);
end
% Output the minimum distance vector
y=cb(:,idx);
end
```

Figure 6. Optimized `euclidean.m` *function.*

```
function Y =  nullcopy_example(alpha)%#eml
N = 5;
Y = eml.nullcopy(zeros(1,N));
for i = 1:N
      Y(i) = alpha^i;
end
```

Figure 7. Initializing the variable Y using the `eml.nullcopy` *function.*

Using Fixed-Point and Native Integer Data Types

By default, MATLAB uses 64-bit double-precision numerical representation for variables created in the workspace. As convenient as this choice is for design exploration, it is not memory-efficient for real-time processing of many common signals, such as image or audio signals represented natively with word lengths of 8 or 16 bits. To handle these types of signals and to implement your MATLAB algorithm on target processors with limited word lengths, you must convert the design to a fixed-point or integer-based representation. You can use Fixed-Point Toolbox™ to create fixed-point variables and perform fixed-point computations. Since the Embedded MATLAB language subset supports the fixed-point data object (`fi`), by using Real-Time Workshop you can generate pure integer C code from your Embedded MATLAB code. This usually involves modifying your original MATLAB function to declare variables based on integer or fixed-point representations.

Our `euclidean_optimized.m` function can process integer data types or fixed-point data types as its input variables. To generate C code we only need to compile the same function with integer or fixed-point

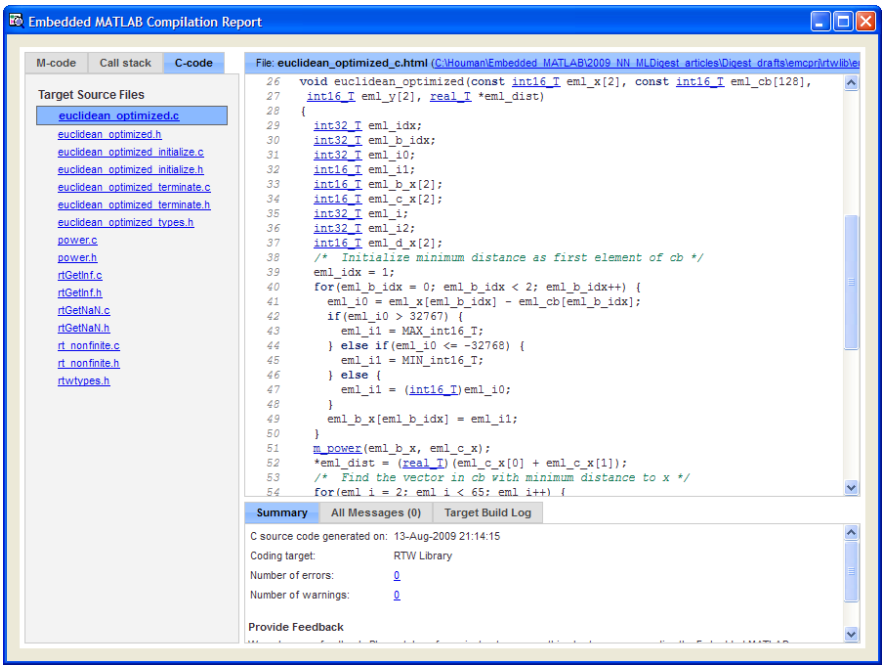

Figure 8. C code for `euclidean _ optimized.m` *compiled with integer input variables. Computations are purely integer-based.*

variables in the example option of the `emlc` command. The command syntax for input variables of 16-bit signed integer type, for example, will be

```
>> emlc -eg {int16(x),int16(cb)} -report euclidean _ optimized.m
```

The resulting generated C code contains only integer C data types, and can be readily compiled into fixed-point processors (Figure 8).

A Common Language and Development Environment

Automatic translation of MATLAB to C with the Embedded MATLAB subset eliminates the need to produce, maintain, and verify hand-written C code. Design iterations become easier, as you stay within the MATLAB environment and take advantage of its interactive debugging and visualization capabilities. Many desirable features of MATLAB programs, such as matrix-based operations, polymorphism, variable-size data and fixed-point numerical representations, are automatically translated to C code, enabling you to focus on improving your design rather than maintaining multiple copies of the source code written in different languages. ■

Resources

VISIT
www.mathworks.com

TECHNICAL SUPPORT
www.mathworks.com/support

ONLINE USER COMMUNITY
www.mathworks.com/matlabcentral

DEMOS
www.mathworks.com/demos

TRAINING SERVICES
www.mathworks.com/training

THIRD-PARTY PRODUCTS AND SERVICES
www.mathworks.com/connections

Worldwide CONTACTS
www.mathworks.com/contact

E-MAIL
info@mathworks.com

91768v00 10/09

For More Information

- Convert MATLAB Code to Embedded C Using Embedded MATLAB
  www.mathworks.com/embeddedc
- From MATLAB to Embedded C.
  DSP Design Line
  www.dspdesignline.com/howto/207800773