Auto Code Generation for Onboard Space Object Detection and Other Flight Software Applications - A Feasibility Study

Amin Yahyaabadi [1], ***Paul Harrison*** [2] **and** ***Philip Ferguson*** [3]

[1] Mechanical Engineering, University of Manitoba, Winnipeg, Canada
[2] Magellan Aerospace, Ottawa, Canada,
[3] Mechanical Engineering, University of Manitoba, Winnipeg, Canada

E-mail: yahyaaba@myumanitoba.ca

Abstract

Fast code prototyping, doing experimental investigations in algorithms, and reusing the code for different hardware or missions has always been a challenge for embedded flight software. In this research, we proposed a method to make all of these possible using MATLAB Coder.

We started with a set of advanced algorithms using MATLAB's syntax, functions, and toolboxes that enable fast code prototyping as well as efficient post-run analysis and plotting. These algorithms are designed to detect resident space objects (RSOs) in images from commercial-off-the-shelf star trackers, using a mix of analytic and machine learning techniques.

We adapted these algorithms, considering specific guidelines that tailor the MATLAB code for automatic C/C++ code generation. Then, using MATLAB Coder and Embedded coder, we were able to generate C/C++ code optimized for multiple hardware platforms. The results show promise for certain coding applications when compared with the performance and labor associated with the hand-converted code. Following the practices and guidelines discovered in this research, we are able to improve the readability, reusability, and performance of the auto-generated code. This paper uses these results to explore the feasibility of auto-generated code for future space missions in more general flight software applications.

Keywords: code generation, MATLAB coder, Embedded coder, codegen, RSO, debris, image processing, flight software, machine learning, star tracker, object detection

Introduction

In this research, we proposed an embedded programming method using MATLAB coder [1] and Embedded Coder [2] to make fast code prototyping, doing experimental investigations in algorithms, and reusing the code for different hardware or missions for embedded flight software possible. The proposed embedded programming method is not limited to our case study and it has a good potential to be used for any general software applications.

Figure 1 [3] shows the procedure of the proposed embedded programming. The code prototyping is done in a high-level language and using MATLAB's syntax, functions, and toolboxes. Then the

code is tested using MATLAB's simulating, post-run analysis, and plotting features. When the programmer is satisfied with the result they will generate the C/C++ code.

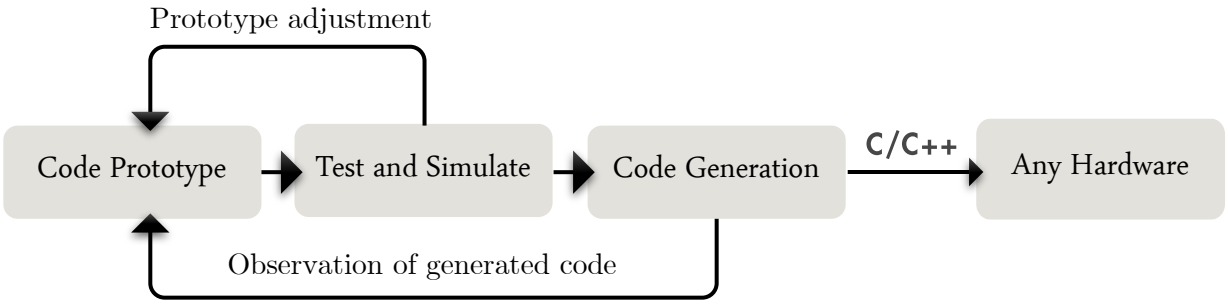

Figure 1 – Auto code generation procedure

Table 1 summarizes the differences between the traditional way of embedded programming versus the proposed method.

Traditional Embedded Programming	Proposed Embedded Programming
Program in a low-level language like C/C++	Program in a high-level language like MATLAB and generate C/C++ code automatically
Time-consuming, High cost of labor	Fast code prototyping (using MATLAB's syntax, functions, and toolboxes)
Hard to do experiments in algorithms	Doing experimental investigations in algorithms (using MATLAB's simulating, post-run analysis, and plotting features)
Hard to reuse the code	Reusing the code for different hardware or missions
Run and debug on the processor	Run and debug on a personal computer
Dependent on hardware	Independent of hardware
Hard to write parallelized and multithreading optimized code	Automatic code replacement to get multithreading and parallelized optimized code

Table 1 - Traditional vs. proposed embedded programming

Different works have been done to incorporate auto code generation into embedded programming workflow using different tools and for different situations, even for critical applications [4], [5], [6], [7], [8], [9]. Many NASA projects have used Mathworks Simulink, RealTime Workshop, and Embedded Coder, for their modeling, code development, and to generate actual C flight code from Simulink models, particularly in the Guidance, Navigation, and Control domain. NASA's Orion

spacecraft has made extensive use of code generators for the development of the flight software [10]. For the PROBA-2 spacecraft, model-based (MATRIXx/SYSTEMBUILD) technologies are used for a full automatic code generation of the onboard software [11]. In prototyping as C/C++ code chapter of [12], it is talked about how to generate C/C++ code for algorithms of latest mobile communications.

Some research has been done to improve the quality of the generated code itself. The official Mathworks documentation for MATLAB coder [13] and Embedded Coder [14] provide tutorial and tips on how to auto-generate code and how to improve the result. In [15], it is talked about techniques such as weak types, primitives and matrix views to allow the efficient generation of C code using MATISSE [16]. [17], which is a book about accelerating MATLAB's performance, has a chapter about a variety of ways by which we could use the speed associated with native code, among those mechanisms using MATLAB Coder Toolbox to convert m-code to compiled C/C++ code, in order to improve the performance of the MATLAB code. It also provides guidelines regarding ways to solve MATLAB Coder compatibility check errors. [18] talks about how to generate code for multicore technology for performance improvement by inserting user-defined S-Functions for Simulink applications. Polyspace sets of tools [19] allow formally proving the absence of critical run-time errors without executing code, checking coding rules, security standards, code metrics, and finding bugs, and proving the absence of run-time errors in the source code.

The objective of this mission, which is used as a case study for our proposed embedded programming method, is to detect resident space objects (RSOs) using commercial-off-the-shelf star trackers. For achieving this, a mix of analytic and machine learning techniques are used. Analytic image processing algorithm identifies an object that is moving from one frame to another frame and provides the centroid locations for the object in each frame. Machine learning algorithm passes an image sequence through a convolutional neural network and classifies the sequence as to whether it contains an RSO. The results were compared with those obtained from hand-converted code to identify the benefits and drawbacks of each approach and to gain knowledge for future software applications.

Programming for Code Generation

In the following sections, we talk about how to write MATLAB code such that we can generate a readable, efficient, and optimized auto-generated C/C++ code.

MATLAB Code structure

The best practice to make the full script to be converted to C/C++ code automatically is to create a `mainF` function without any outputs to wrap all the codes that we want to run on the processor. Inside `mainF`, we have a `dataLoader` function, which will be used for reading data (in this case images) from microSD card. Then after loading data needed, the `algorithm` is written, which gets the imageData as an input argument and returns some outputs which are used to

generate a report in `outputReportGenerator` function. Because MATLAB optimizes code generation, if you don't return the `outputArguments` or do not save or write any file, MATLAB won't generate complete code for the functions.

```
% main function to be run on the processor
function [outputArgument1,outputArgument2]=mainF(programInputSettings)

sdData=dataLoader(); %sdData for our case is imageData
[outputArgument1,outputArgument2]=algorithm(sdData,programInputSettings);
outputReportGenerator(outputArgument1,outputArgument2);
end
```

For keeping the code modular and more readable, inside each of these functions, add the `coder.inline` to prevent MATLAB from make the functions inline.

```
coder.inline('never')
```

Data Preparing

Data separation should be done to decide which part of data should be read from microSD card during running (images for our case), which part we want to be written in code as variables and to be stored in memory during compiling (all the other data), and which part we want to pass to program as input arguments when we run the executable (program input settings). This is the MATLAB script for separating microSD card data from compiled data:

```
clear;
load(' allData.mat');

save('imageData3DMatrix.mat','imageData');
clear('imageData');

save('otherData.mat');
clear;
```

Hard-Coded Data

The variables are hard-coded inside `algorithm()` function:

```
otherData=coder.load('otherData.mat'); % hard-coded data

var1=otherData.var1; % each variable in otherData
var2=otherData.var2;
```

To automatically generate the code for assigning each variable of the structure in otherData, we can use structvars [20]:

```
structvars(otherData)
```

MicroSD Card Data Pre-processing

The data that is read during running should be converted to a compatible format. The following MATLAB script will convert "imageData.mat" file into separate ".bin" files for each image. Based on the type of data the same `precision` type should be used (here "unit" for unsigned integers). If the images are in 8-bit JPEG format, you don't need to convert them to ".bin" file.

```
data=load('imageData3DMatrix.mat')
% Determining the proper format
imageData=data.imageData;

imageDataProperClass=realClasser(imageData)
% use the properClass to adjust imageLoader code as well.

% Start conversion
numImage=4;

for i=1:numImage

    name=['imageData',int2str(i),'.bin'];
    fileID = fopen(name, 'w');

    fwrite(fileID, imageData(:,:,i),imageDataProperClass); % our images are stored in
3D matrix.

    fclose(fileID);

end
```

We used `imageDataProperClasser,` which is a custom function that detects the proper class of the data (e.g. `'uint64', 'double',` etc.). However, because MATLAB uses double as its default class for numbers, after the processor reads the data, if they are numeric, it will convert them to double numbers.

MicroSD card Data Loader

The dataLoader function which will be run on the processor is used to read the data from microSD card during running.

```
function imageData=dataLoader()

coder.inline('never');
numImage=4;
pixelCount=256;

% Read image data from bin/jpg file
imageData=zeros(pixelCount,pixelCount,numImage);
```

```
for i=1:numImage

    name=['imageData',int2str(i),'.bin'];
    % name=['imageData',int2str(i),'.jpg']; % for 8-bit jpg file

    fileID=fopen(name);

       imageData(:,:,i)=fread(fileID,[pixelCount,pixelCount],'uint64');
    % imageData(:,:,i)=imread(name); % for 8-bit jpg file

    fclose(fileID);
end

end
```

There are multiple things to pay attention to. First, the same precision that we used previously in `fwrite` is used here to read the data. Second, for creating the `name` of the file we used array concatenation and `int2str` instead of using `sprintf` because the generated code is better. Third, using `coder.varsize` for defining the `name` variable and specifying the upper bound of `numImages` gives an error that left-hand side is fixed-sized but the right-hand side is varying:

```
coder.varsize('name',[1,14],[0,0])
```

Forth, if you are sure that the number of files is less than 10, you can use the following instead of `int2str` to get a simpler generated code. You can use if conditions to include two or more characters for a greater number of files.

```
name = ['Data', char(i+'0'), '.bin']
```

or

```
name = 'DataX.bin';
name(5) = char(i+48);
```

Data Definition

Using the correct data type is very important for generating efficient and readable code. The big difference between MATLAB and C/C++ code is that in MATLAB in contrast to C/C++, we don't need to define variables class and size, we can change the type of a variable, and we can expand the size of the variables very easily. To make the data definition in MATLAB code optimized in terms of code generation we need to consider some guidelines.

As a general rule define as many as variables you can during writing MATLAB code like the way you code in C/C++, especially, if the variable is going to be assigned inside a loop, or if the variable type is structure.

If a variable's value is calculated or assigned from another defined variable, we don't need redefining it.

```
h=3;
i=h+5;                  % initialize i with h+5 values
```

Among all the data types that are available, use arrays, matrices (multidimensional arrays), and structures.

Static Arrays

To define statically allocated arrays, use the following syntax:

```
% Arrays
a=0;                    % 0-dimensional double
b=zeros(2);             % 1-dimensional array double
c=zeros(2,4);           % 2-dimensional array double
d=zeros(2,4,3);         % 3-dimensional array double
h=[5,4,3;2,6,7];        % 2-dimensional double with known initial value

e=int32(0);             % 0-dimensional int32
f= zeros(2,'uint32');   % 1-dimensional unint32
g=5*ones(5,3,'int8');   % 2-dimensional int8 initialized to 5
```

The double data type has better compatibility with MATLAB because it is the type which it uses most of the time. So, usually during prototyping the code, double data is used. Unless only integer numbers exist in your program, we recommend sticking to double type to avoid round-off errors, and also because sometimes double data type generates more readable code, especially if those variables are used in defining other variables sizes.

One of most important limitations of integer numbers is their arithmetic operations with double numbers. For example, the following code generates `uint64 11` instead of the correct result:

```
uint64(5)*2.3
```

For using double variables as integers in sprint or fprintf, you need to convert those using int convertors:

```
fprintf('previous image: %d & current image: %d\n', int8(iImage-1), int8(iImage));
```

Static Structures

To define statically allocated structures, use the following syntax:

```
% Structures
struct1=struct('field1',0,'field2',[0,0],'field3',zeros(3,3),'field4','string');    %
0 dimensional structures

dim1=5;
struct2=repmat(struct1,dim1)        % 1 dimensional structure

dim2=3;
struct3=repmat(struct1,dim1,dim2);  % 2 dimensional structure
```

Then after initializing, you can access structure fields and change their values. If you don't initialize structure first, you will receive an error if you want to use other structure fields during changing values.

```
struct1.field1=54; % changing field value

struct1.field2=[5*struct1.field1,3];
% if you initialize structure, this does not give an error

struct2(3).field1=31;          % accessing 1-dimensional structure fields value
struct3(2,2).field4='hello!'   % accessing 2-dimensional structure fields value
```

There is a limitation in using multidimensional structures that the size of values in the same fields in different structure elements should be the same. For example, you will get an error if you define the following structure:

```
% different sizes of val1 in different structure elements
struct2(1).field1=0;
struct2(2).field1=[0;0];
```

In this case, you should either make all values the same size using dummy values or use cells data types instead of multidimensional structure, although not recommended because there is no native equivalent type for cells in C/C++, and cell elements indexing in MATLAB has limitations compared to arrays.

Static Cells

For statically allocated cell definition, all cell array elements on all execution paths should be assigned before using or passing to/from a function:

```
struct2=struct('field1',0,'field2',zeros(3,3)); % initialize structure
% cell definition
k=cell(2,1);
k{1,1}=struct2;
k{2,1}=struct2;
k{1,1}.field1=0; % accessing inner structure fields
```

Dynamic Variables

As a goal, always try to program your code such that all the variables have defined sizes that do not change during running. However sometimes when you try to generate code you will receive errors about the size mismatch between left- and right-hand side of an assignment, which is caused since the variable changes size. In these cases, defining a variable as coder.varsize helps.

Dynamic Arrays

To define dynamically allocated arrays, use the following syntax:

```
% Varsize array
coder.varsize('a',[5],[1])        % 1st dim is variable with 5 as upper bound
coder.varsize('b',[5,10],[0,1])   % 2nd dim is variable size with 10 as upper bound
coder.varsize('c',[5,10],[1,1])   % both dims are variable size with 5 and 10 as upper
bounds
coder.varsize('d',[5,10],[0,0])   % both dims are fixed size with 5 and 10 as size
```

Dynamic Structures

To define dynamically allocated structure field, use the following syntax:

```
myStruct=struct('field1',0,'field2',zeros(3,3)); % an initialized structure
coder.varsize('myStruct.field2',[3,5],[0,1]);
% 2nd dimension of the field2 is variable with 5 as upper bound
```

Generating Executable

After generating the source code, if you want to make an executable from the generated code, you need to provide a custom main.c/cpp (along with the main.h header file). MATLAB by default generates an example main source code and header files that call our mainF function. These examples are the place for adding custom code around the whole mainF code. If you follow the structure of the MATLAB code that we suggested, examples can be used with minimal adjustment. To provide custom main files, you need to include the main.c/cpp source code in MATLAB coder app and include the external directory if it is not in the workspace directory. Also, you should choose the option of "generate example main code, but do not compile" to exclude example main files from making and instead use the custom main code provided.

For our research, to test the performance of the code, we added some Linux commands that measure the time it takes to execute the code, and also, the amount of memory which the code takes during the run-time.

The following is our custom main.c code, which is edited from example Matlab generated main.c:

```
/* Include Files */
#include "mainF.h"
#include "main.h"
#include "mainF_terminate.h"
#include "mainF_initialize.h"

/* time libraries */
#include <time.h>

#include <sys/time.h>
#include <sys/resource.h>


/* time variables */
double micros;
```

```
float seconds;

clock_t start, end;

struct rusage before;
struct rusage after;
float a_cputime, b_cputime, e_cputime;
float a_systime, b_systime, e_systime;
float maxMemory;

/* Function Declarations */
static void main_mainF(void);

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
static void main_mainF(void)
{
  // similar to what Matlab generates
}


/*
 * Arguments    : int argc
 *                const char * const argv[]
 * Return Type  : int
 */
// int main(int argc, const char * const argv[])
int main()

{
//   (void)argc;
//   (void)argv;

    getrusage(RUSAGE_SELF, &before);
    start = clock();

  // The following initialize and run section is automatically generated
  /* Initialize the application.
     You do not need to do this more than one time. */
  mainF_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_mainF();

  /* Terminate the application.
     You do not need to do this more than one time. */
```

```
  mainF_terminate();


  /* time calculation */
  end = clock();
  getrusage(RUSAGE_SELF, &after);

  micros = end - start;
  seconds = micros / 1000000;

  a_cputime = after.ru_utime.tv_sec + after.ru_utime.tv_usec / 1000000.0;
  b_cputime = before.ru_utime.tv_sec + before.ru_utime.tv_usec / 1000000.0;
  e_cputime = a_cputime - b_cputime;

  a_systime = after.ru_stime.tv_sec + after.ru_stime.tv_usec / 1000000.0;
  b_systime = before.ru_stime.tv_sec + before.ru_stime.tv_usec / 1000000.0;
  e_systime = a_systime - b_systime;
  printf("CPU time (secs): user=%.4f; system=%.4f; real=%.4f\n",e_cputime, e_systime,
seconds);

  maxMemory=after.ru_maxrss-before.ru_maxrss;

  printf(" max Memory:%.4f\n",maxMemory);

  return 0;
}
```

And here is the differences of our custom main.cpp with main.c code:

```
#include<iostream>
std::cout<<"CPU time (secs):<<
user="<<e_cputime<<"system="<<e_systime<<"real="<<seconds<<endl;
std::cout<<"max Memory"<<maxMemory<<endl;
```

The main.h is similar to what MATLAB generates, and the only difference is the function prototype that needs to be consistent with main definition in corresponding main source file.

After debugging the algorithm, itself for MATLAB run-time, we also need to debug the generated code in an embedded run-time. The best workflow for debugging is to generate debug executable for the Intel processor of the Linux desktop computer. To do this we need to check "generate run time error checks", show "verbose compiler output", and "show differences from MATLAB". By running this debug executable on the desktop computer, we can find the MATLAB lines of the code that generate run-time errors. In addition, we can use Polyspace bug finder that automates the debugging procedure by analyzing and running the code.

For generating the main embedded processor, uncheck the debug options, to get faster and more readable code.

Custom C/C++ Code

Sometimes for some parts of the code, there are better ways to write the C/C++ code than what MATLAB generates automatically because of differences between language. For example, because MATLAB supports variable-sized arrays, it can extend a variable size easily during running the code, while, in C/C++ this size checking can generate an inefficient unnecessary logic.

In these cases, make a wrapper MATLAB function for that part of the code using MATLAB refractor or make it manually. Then generate C/C++ code only for this function. You can use your MATLAB script to automatically detect input arguments types, or you can define them manually. After generation, edit the C/C++ code the way which is more appropriate, or you may want to write the function from scratch. The C/C++ header file for the function should include the function prototype. Also, check the generated main file since MATLAB sometimes defines input arguments there.

If you do not want to use variable-sized variables, remove `customFuncStackData *SD` from input arguments of function prototype in .h file. Also, remove `#include "rtwtypes.h"` and `#include "customFunc_emxutil.h"` from .h and .c file respectively.

In MATLAB code use the following syntax to include and evaluate your custom C/C++ function:

```
% including custom function header file in MATLAB
coder.cinclude('customFunc.h');

% if the custom function has only one output
output = coder.ceval('customFunc', input1, input2);

% if the custom function has multiple outputs
coder.ceval('customFunc',input1,input2,coder.ref(output1),coder.ref(output2));
```

Extrinsic Functions

Some of MATLAB's functions are not supported for code generation. MATLAB considers some of them such as plot, disp, figure as extrinsic functions by default. For cases that the function is not necessary to your embedded code, you can make it extrinsic using the following syntax:

```
coder.extrinsic('warning'); % warning function is not supported for code generation.
```

Auto Code Replacement for Optimized Libraries and Intrinsics[1]

In MATLAB Embedded coder, we can replace a function with its optimized equivalent in a library or for a processor. This can have a significant effect on the performance of the software, while there is no need to learn different intrinsic and libraries, which is especially hard for manually using these intrinsic functions in programming. The replacement is done automatically for the

[1] Instrinisc functions of the processor

operations that are done inside BLAS libraries or for the processor's manufacture provided intrinsic multithreading (SIMD[2]) instructions (e.g., Neon intrinsics on ARM Cortex-A processors [21] or Helium MVE intrinsics on ARM Cortex-M processors [22]). Also, there is a possibility for code replacement based on custom rules that the user defines for the functions of other external libraries.

The number of code replacement occurrences depends on the type (e.g., integers vs. float) and size (16x8 and 32x4 arrays) of the data, and the operation which is done (e.g., multiplying vectors or adding matrices). So, the MATLAB code can be written by considering these beforehand. Based on our experiments, choosing "column major arrays" option and also disabling the "preserve the array dimensions" option will increase the number of catches.

Neural Networks Code Generation

We can use MATLAB deep learning toolbox (also known as neural network toolbox) to create an artificial neural network. The training for the network should be done inside MATLAB. Then after training, using MATLAB coder, a code can be generated for the trained network for ARM and Intel processors and GPUs. The generated code for processors uses ARM Compute and Intel MKL-DNN libraries, which are optimized libraries for running code on ARM and Intel processors respectively.

The following is the syntax for generating the code:

```
net=load('Network.mat'); % Network.mat contains the trained network

targetlib='arm-compute'; % on arm hardware
% targetlib='mkldnn'; % on Intel hardware

cnncodegen(net.rsoConvnet,'targetlib',targetlib)
```

However, if you want to use MATLAB's neural network objects (e.g., batchNormal) in combination with custom code to make a custom neural network, you cannot generate C/C++ code automatically. The possible option is to generate the C/C++ code for a trained network and then rewrite the MATLAB code to use the results of the functions in the generated code using MATLAB external code option (`coder.ceval`).

MATLAB Coder Limitations

In addition to what has been discussed throughout the paper, there are some points to consider.

[2] Single input multiple data

Having a constant variable appear by name

Having a constant variable appear by name (rather than the value) in the sizes of other variables is not supported in MATLAB Coder as of MATLAB R2019a. In C/C++, you can define a constant using one of the following syntaxes:

```
const int arraySize=5
#define arraysize 5
```

Later, you can define an array using the following syntax:

```
int array[arraySize];
```

However, in MATLAB when you write the following, Coder just replaces `arraySize` with the actual number which is 5:

```
arraySize=int8(5);
array=zeros(1,arraySize); % zeros is just used for specifying size
```

The generated code is

```
void coder(double A[5])
{
  memset(&A[0], 0, sizeof(double) << 16);
}
```

Even if you define a constant using the following syntax, you are not allowed to use `arraySize` in MATLAB calculations:

```
arraySize=coder.opaque('const int16','5');
A=zeros(1,arraySize);
```

Subtract vector from matrix

Although MATLAB supports subtracting a vector from a matrix when you try to generate code using MATLAB coder for subtracting a vector from matrix gives me size mismatch error.

```
A=3*ones(4,5);
B=ones(1,5);
C=A-B;
```

You should use the following syntax instead:

```
C = bsxfun(@minus, A, B);
```

Generated Code Performance and Quality

Different benchmarks and tests are done to examine the performance of the generated code. The result of the benchmarks is compared with those obtained from hand-converted code.

As the main benchmark test for this case study, the generated C code performance is tested on Zybo Z7-20 that is a development board based on the Xilinx Zynq Z-7020 system on the chip (SoC) architecture. It has a dual-core ARM Cortex-A9 processor, and an FPGA equivalent to Artix-7 FPGA [23], [24]. For the operating system of the ARM cortex processor, Linux Linaro is used as the real-time operating system.

The average performance of the auto-generated code for the whole project is as follows:

```
CPU time (secs): user=0.6514; system=0.0404; real=0.6917
max Memory:19096
```

The result of the auto-generated code shows promise, and it passes our requirement with a 40% margin.

Because rewriting the whole project by hand is a time-consuming task, by the help of MATLAB profiler for MEX code, we detected the functions and parts of the code that take most of the run-time, and then we wrote handwritten code snippets for those parts. By doing benchmark for these parts, we got this average result:

```
CPU time (secs): user=0.3456; system=0.0000; real=0.3455
max Memory:304
```

The difference in memory and run-time is because the handwritten code only contained minimal code snippets to test the most CPU-intensive tasks, and so the code was not functional in a real manner and lacked many features of the software (e.g., many image-processing functions, reading data from microSD card, etc.).

To further assess the performance of generated code, a mathematical algorithm was used to test the performance of auto-generated code.

The structure of MATLAB code is similar to our proposed structure:

```
function [ x,invA ] = mainF()

[A,B,C] = dataLoader();
[ x,invA ] = algorithm(A,B,C);

end
```

`dataLoader` function is

```
function [A,B,C] = dataLoader()
coder.inline('never');

A=rand(200,200);
B=rand(200,1);
C=rand(100,100);

end
```

In handwritten code, simple for loops and `rand()` is used for generating random numbers.

`algorithm` function is:

```
function [ x,invA,mulAb ] = algorithm( A, b,C)
coder.inline('never');

x=A\b;
invA=inv(C);

end
```

In handwritten code, Gaussian elimination for solving the linear system, Gauss Jordan algorithm is used for calculation of inverse. Different algorithms result in different benchmarks.

The average performance for auto-generated code is

```
CPU time (secs): user=0.2645; system=0.0000; real=0.2644
max Memory:0.0000
```

The average performance for hand-written code is

```
CPU time (secs): user=0.0704; system=0.0000; real=0.0703
max Memory:0.0000
```

The auto-generated code is readable and so it is possible to edit code. As an example, the algorithm generated for linear system solving with preserving dimension disabled is as follows:

```
#include <string.h>
#include "mainF.h"
#include "mldivide.h"
#include "xtrsm.h"
#include "xgetrf.h"

void mldivide(const double A[40000], double B[200])
{
  static double b_A[40000];
  int ipiv[200];
  int info;
  double temp;
  memcpy(&b_A[0], &A[0], 40000U * sizeof(double));
  xgetrf(b_A, ipiv, &info);
  for (info = 0; info < 199; info++) {
    if (ipiv[info] != info + 1) {
      temp = B[info];
      B[info] = B[ipiv[info] - 1];
      B[ipiv[info] - 1] = temp;
    }
  }
  xtrsm(b_A, B);
  b_xtrsm(b_A, B);
}
```

Future Steps

As future plans built on the experience and results of this research, we plan to use auto code generation for the flight software of ManitobaSat-1 mission, which is a student cube satellite mission, for attitude and determination control subsystem's algorithms, payload image processing, and communication algorithms. The main processor of ManitobaSat-1 onboard processor is Microsemi Smart Fusion 2 system on a chip, which has an ARM Cortex-M3 and FPGA fabric [25]. As an extension to this research, we want to test the performance and quality of HDL coder which generates HDL code for FPGA fabric of this pressor. This can offload some standalone operations for image processing and communication algorithms or any other CPU intensive algorithm from the ARM Cortex processor. Also, Fixed Point Designer will be examined to evaluate the effect of use fixed-point numbers compared to floating-point numbers on the performance.

In addition, as part of another ongoing research of STARLab in the University of Manitoba, the proposed embedded programming method is used to design and implement different control and machine learning algorithms for onboard flight controllers of unmanned aerial vehicles.

Conclusion

The proposed embedded programming method uses high-level MATLAB language in a structured and organized way to automatically generate C/C++ code. By showing promise, and saving a huge amount of time, energy, cost this method can be considered as a suitable method for implementing new algorithms and can have a huge effect on the making new technologies practical and accessible.

Acknowledgment

This project was done as a partnership between Magellan Aerospace and the University of Manitoba and was also supported with the funding of NSERC, CSA, and DND. This paper presented the University of Manitoba's proposed embedded programming method which undertook the research on code conversion of algorithms provided by Magellan Aerospace.

References:

[1] Mathworks, "MATLAB Coder." [Online]. Available: https://www.mathworks.com/products/matlab-coder.html. [Accessed: 16-Jul-2019].

[2] Mathworks, "Embedded Coder." [Online]. Available: https://www.mathworks.com/products/embedded-coder.html. [Accessed: 16-Jul-2019].

[3] Mathworks, "Modified diagram from The Joy of Generating C Code from

MATLAB." .

[4] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from Matlab/Simulink for critical applications," Can. Conf. Electr. Comput. Eng., pp. 1–6, 2014.

[5] S. Gang, C. Dacheng, H. Jingfeng, and H. Jun-wei., "Research on three-axis six-DOF shaking table based on rapid prototyping of DSP algorithms using SIMULINK," 2008 2nd Int. Symp. Syst. Control Aerosp. Astronaut. ISSCAA 2008, pp. 1–6, 2008.

[6] "IDNEO Develops Embedded Computer Vision and Machine Learning Algorithms for Interpreting Blood Type Results - MATLAB & Simulink." [Online]. Available: https://www.mathworks.com/company/user_stories/idneo-develops-embedded-computer-vision-and-machine-learning-algorithms-for-interpreting-blood-type-results.html. [Accessed: 16-Jul-2019].

[7] "Respiri Develops Mobile App for Wheeze Detection and Asthma Management." [Online]. Available: https://www.mathworks.com/company/user_stories/respiri-develops-mobile-app-for-wheeze-detection-and-asthma-management.html. [Accessed: 16-Jul-2019].

[8] Mathworks, "VivaQuant Accelerates Development and Validation of Embedded Device for Ambulatory ECG Sensing." [Online]. Available: https://www.mathworks.com/tagteam/78014_92143v00_VivaQuant_UserStory_final.pdf.

[9] Mathworks, "Delphi Develops Radar Sensor Alignment Algorithm for Automotive Active Safety System." [Online]. Available: https://www.mathworks.com/tagteam/81084_92225v00_DelphiAutomotive_UserStory_final.pdf.

[10] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," IEEE Aerosp. Conf. Proc., 2008.

[11] J. Côté and J. De Lafontaine, "Magnetic-Only Orbit and Attitude Estimation Using the Square-Root Unscented Kalman Filter Application to the.pdf," no. August, pp. 1–24, 2008.

[12] D. H. Zarrinkoub, Understanding LTE with MATLAB®. Chichester, UK: John Wiley & Sons, Ltd, 2014.

[13] Mathworks, "Matlab Coder Documentation," Mathworks. [Online]. Available: https://www.mathworks.com/help/coder/index.html. [Accessed: 16-Jul-2019].

[14] Mathworks, "Embedded Coder Documentation," Mathworks. [Online]. Available: https://www.mathworks.com/help/ecoder/index.html. [Accessed: 16-Jul-2019].

[15] J. Bispo, L. Reis, and J. M. P. Cardoso, "Techniques for efficient MATLAB-to-C

compilation," pp. 7–12, 2015.

[16] P. Pinto, R. Nobre, T. Carvalho, and P. C. Diniz, "The MATISSE MATLAB Compiler * A MATrix (MATLAB) -aware compiler InfraStructure for embedded computing SystEms," pp. 602–608, 2013.

[17] Y. M. Altman, Accelerating MATLAB Performance. 2018.

[18] M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim, "Deriving high-performance real-time multicore systems based on simulink applications," Proc. - IEEE 9th Int. Conf. Dependable, Auton. Secur. Comput. DASC 2011, pp. 267–274, 2011.

[19] Mathworks, "Polyspace." [Online]. Available: https://www.mathworks.com/products/polyspace.html?s_tid=srchtitle. [Accessed: 16-Jul-2019].

[20] Matt Jacobson, "Structure fields to variables." File Exchange - MATLAB Central, 2009.

[21] ARM, "SIMD ISAs | Neon Intrinsics Reference – Arm Developer." [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics. [Accessed: 22-Jul-2019].

[22] ARM, "SIMD ISAs | Helium MVE Intrinsics Reference – Arm Developer." [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/helium/mve-intrinsics. [Accessed: 22-Jul-2019].

[23] Digilent, "Zybo Z7 Reference Manual [Reference.Digilentinc]." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual. [Accessed: 17-Jul-2019].

[24] Xilinx, "Zynq-7000 SoC Data Sheet: Overview." pp. 1–25.

[25] Microsemi, "SmartFusion2 SoC FPGAs | Microsemi." [Online]. Available: https://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2. [Accessed: 22-Jul-2019].