E85: Digital Design and Computer Engineering
Lab 6: Linear Algebra in C on a Microcontroller

Objective

The purpose of this lab is to familiarize yourself with programming and debugging in C on an embedded microcontroller. Specifically, you will write some linear algebra routines that will help you become comfortable with loops, arrays, and pointers.

1. Welcome to the STM32 Nucleo

The STM32 Nucleo board has a complete 32-bit ARM Cortex-M0 microcontroller with a wide assortment of peripherals packed into a circuit board smaller than 2 x 5 cm at a cost of $11. With this board, you could add a microprocessor to a system for comparable cost to a piece of lumber or pipe. The Nucleo works with the industry-standard Keil Microcontroller Development Kit (MDK). You can use the tools in the E85 lab, or can purchase your own STM32F042 and download the free unlicensed version of the MDK-ARM software to use on your own Windows, Linux, or Mac computer (microUSB cable required). If you install yourself, make sure you install the STM32F0 device pack during the installation process.

2. Keil Tutorial

In this section, you will learn to write, compile, and debug programs with the Kiel MDK.

Launch Keil uVision from the start menu. Choose **Project -> New uVision Project** and create a project named tutorial in your Charlie directory. Select the STM32F042K6 device target. In the **Manage Run-Time Environment**, expand **Device** and check the **Startup** box. Then click on the **Resolve** button to resolve dependencies and you'll see that **CMSIS** (the Cortex Microcontroller Software Interface System) is selected too. Choose OK.

In the project pane, click to expand **Target 1** and **Source Group 1**. Click on the **Target1** item, then choose **Project -> Options for Group 'Target 1...'**. In the **Debug** tab, choose **Use: ST-Link Debugger**. Click on **Settings** and go to the **Flash Download** tab and choose **Erase Full Chip**. Close the dialog boxes by clicking **OK** and **OK**.

Choose **File -> New...** and create a file named tutorial.c. Enter the following program, which is supposed to compute the dot product of two vectors but has some bugs. Save your program. Predict what the dot product of [3 4 5] and [1 2 3] should be.

```
// tutorial.c
// Your Name, date, email
// Dot product code to learn the Keil tools
#define DIM 3
double dotproduct(int n, double a[], double b[]) {
     volatile int i;
     double sum;
     for (i=0; i<n; i++) {
          if (i=0) sum = 0;
          sum += a[i]*a[i];
     }
return sum; }
int main(void) {
     double x[DIM] = {3, 4, 5};
     double y[DIM] = {1, 2, 3};
     double dot;
     dot = dotproduct(DIM, x, y);
     return dot;
}
```

Right-click on the **Source Group 1** and "**Add Existing Files to Group 'Source Group 1'…**" and add tutorial.c.

Choose **Project -> Build Target** to compile. Scroll through the **Build Output** panel at the bottom to look for warnings and errors. You will see a warning that the `(i=0)` comparison should have been `(i==0)`. Change your code to correct this and rebuild and ensure there are no errors or warnings.

Make sure a Nucleo board is plugged into a USB port. Choose **Debug -> Start/Stop Debug Session** to invoke the debugger. You should see a green arrow pointing to the beginning of the main function.

Use the **Step** command (on the icon bar, or press F11) to step through your code. You can watch the variable addresses and values in the **Call Stack + Locals** pane in the lower right. Make it bigger so you can see everything. Expand the x and y arrays so you can see their values. As you step through the first two lines, you should see arrays getting initialized. The compiler aggressively optimizes the code, so sometimes you will see weird things in the debugger or variables not changing when you expect them to change. i is declared as volatile in this code to force it to be preserved by the optimizer and appear in the debugger.

Find the bug that causes the dot product to be incorrect. Fix your code. Choose **Debug -> Start/Stop Debug** session, then rebuild the code, download it again, and start a new debug session.

3. Linear Algebra

The goal in this section is to write a library of linear algebra routines in C. This will help you get accustomed to loops, arrays, and pointers in C, and it is good to

understand these routines because they are fundamental building blocks of signal processing code.

The following functions operate on m (rows) × n (columns) matrices. The result in a matrix which should already have been allocated prior to the function call. Transpose produces an n × m result.

```
void add(int m, int n, double *A, double *B, double *Y); // Y = A + B

void linearcomb(int m, int n, double sa, double sb, double *A, double
*B, double *Y); // Y = sa*A + sb*B

void transpose(int m, int n, double *A, double *At); // At =
transpose(A). int equal(int m, int n, double *A, double *B); // returns
1 if equal, 0 if not
```

The following function multiplies a m1 (rows) × n1m2 (columns matrix) A by a n1m2 × n2 matrix B to produce a m1 × n2 matrix Y. Y should already be allocated and the contents will be overwritten.

```
void mult(int m1, int n1m2, int n2, double *A, double *B, double *Y);
// Y = A * B
```

Test your program with the following code, using the `newMatrix` and `newIdentityMatrix` code from lecture. You will need to `#include <stdlib.h>` to use the `malloc` function.

```
int main(void) {
  double v1[3] = {4, 2, 1};                      // 1 x 3 vector
  double v2[3] = {1, -2, 3};                     // 1 x 3 vector
  double dp = dotproduct(3, v1, v2);
  double m1[9] = {0, 0, 2, 0, 0, 0, 2, 0, 0};    // 3 x 3 matrix
  double *m2 = newIdentityMatrix(3);        // 3 x 3 identity matrix
  double *m3 = newMatrix(3, 3);             // 3x3matrix
  double m4[6] = {2, 3, 4, 5, 6, 7};        // 3x2matrix
  double *m5 = newMatrix(3, 2);             // 3x2matrix
  double m6[6] = {6, 2, 5, 8, 2, 7};        // 2x3matrix
  double *m7 = newMatrix(3, 2);             // 3x2matrix
  double *m8 = newMatrix(3, 2);             // 3x2matrix
  double expected[6] = {2, 1, 0, 1, 0, -1};
  int eq;
  add(3, 3, m1, m2, m3);
  mult(3, 3, 2, m3, m4, m5);                     // 3x2result
  transpose(2, 3, m6, m7);
  linearcomb(3, 2, 1, 1-dp, m5, m7, m8);
  eq = equal(3, 2, m8, expected);
  return eq;
}
```

Predict what each of the matrices should be and particularly check that m8 matches your expectations.

4. Extra Credit: Solving Systems of Linear Equations

Write a C function to invert an n × n matrix. If the matrix is singular, the function should return 0 and Y is a don't care; otherwise, the function should return 1 and Y is A^{-1}.

```
int invert(int n, double *A, double *Y);
```

Then write a function to solve Ax=b for a system of n variables.

```
int solve(int n, double *A, double *b, double *x);
```

The function should again return 0 if A is singular. Use your function to solve the following system of linear equations:

```
3a  + b  + c  + d = 10
2a  + 3b + 4c - d = 16
-4a +           d = 0
      3b - 2c     = 0
```

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Code for `add`, `linearcomb`, `transpose`, `equal`, and `mult`.
3. What does your code produce for m8? Does it match your expectations?
4. Extra credit, if applicable. Give your code and a, b, c, and d.

If you have suggestions for further improvements of this lab, you're welcome to include them at the end of your lab.