ECSE426 Microprocessor Systems

Fall 2016

Lab 1: Kalman Filter

Objective

The objective of this experiment is to familiarize you with assembly language programming concepts, ARM's Cortex instruction set (ISA) and assembly addressing modes. The ARM calling convention will need to be respected, so that the assembly code can later be used with the C programming language. In the follow-up experiments, the code developed here will be used in larger programs written in C. We will introduce the Cortex Microprocessor Software Interface Standard (CMSIS) application programming interface that incorporates a large set of routines optimized for different ARM Cortex processors. The tutorial associated with this lab will introduce you to the new version of Keil IDE (5.x) and the ARM compiler, simulator and associated tools.

Preparation for the Lab

To prepare for Lab 1, you will need to attend Tutorial 1, where you will learn how to create and define projects. One of the examples will be pur assembly code projects. The tutorial will show you how to ensure that the tool inserts the proper startup code for a given processor, write and compile the code, as well as provide the basics of program debugging. Elaborate details about debugging in Keil are provided in the document entitled ***"Debugging with Keil"*** which will be uploaded on my courses.

Other documents that will be of importance include the Cortex M4 programming manual, and Quick reference cards for ARM ISA, all available within the course online documentation. **More useful references are found in the tutorial slides.** Since there is a vast amount of reference material (hundreds of pages over the course of the semester), make sure to attend the tutorials so that the TA will guide you where to look.

Background on the Kalman filter

The Kalman filter is a sequential estimation algorithm that produces estimates of an unobserved state variable based on a sequence of noisy observations. It minimizes the estimation error when the noise is Gaussian and the system (describing the evolution of the state and its relationship with the observations) is linear.

The system is modeled as follows:

$$x_k = Fx_{k-1} + w_k \qquad w_k \sim N(0, Q)$$
$$z_k = Hx_k + v_k \qquad v_k \sim N(0, R)$$

Here x_k is the unobserved state variable, z_k is the observed measurement, F is the state transition model, w_k is the process noise, assumed drawn from a zero-mean Gaussian distribution with covariance matrix Q, H is the observation model mapping the state to the noiseless measurement, and v_k is the observation noise, assumed drawn from a zero-mean Gaussian distribution with covariance matrix R.

The Kalman filter maintains an internal state estimate $\hat{x}_k$, updated at each time step k, and an estimated error covariance matrix $\hat{P}_{k|k}$. The Kalman filter consists of two steps: (i) a predict step and (ii) an update step.

Predict:

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1}; \quad \hat{P}_{k|k-1} = F\hat{P}_{k-1|k-1}F^T + Q$$

Update:

$$K = \hat{P}_{k|k-1}H^T(H\hat{P}_{k|k-1}H^T + R)^{-1}$$
$$\hat{P}_{k|k} = (I - KH)\hat{P}_{k|k-1}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K(z_k - H\hat{x}_{k|k-1})$$

The Experiment

Part I – Pure Assembly Programming: 1-d Kalman filter

You are required to write an assembly subroutine "`Kalmanfilter_asm`" in ARM Cortex M4 assembly language that implements a one-dimensional Kalman filter to process one measurement input to update the state estimate. You should naturally use the built-in floating-point unit by using the existing floating-point assembler instructions.

Your assembly filter will take four parameters:

1. A pointer to the input data array
2. A pointer to the filtered data array
3. The arrays' length
4. A pointer to the Kalman filter state (struct)

Your subroutine should follow the ARM compiler parameter passing convention. Recall that up to four integers and 4 floating-point parameters can be passed by integer (R0 – R3) and floating-point registers (S0 – S3), respectively. For instance, R0 and R1 might contain the values of the first two integers and S0 will contain the value of a floating-point parameter. If the datatype is more complex (e.g, struct or a matrix), then a pointer to it is passed instead in R0 or R1. For the function return value, the register R0/R1 or S0/S1 are used for integer and floating-point results of the subroutine, respectively.

The filter will hold its state as (f, h, q, r, x, p, k) that are five single-precision floating-point numbers. It is convenient to keep these state variables in a single C language `struct`, holding Kalman filter state in each time step of operation, consisting of:

```
float q; //process noise covariance
float r; //measurement noise covariance
float x; //estimated value
float p; //estimation error covariance
float k; // adaptive Kalman filter gain.
```

You have to ensure that the operation of the filter is correct for all operations of inputs and state variables, including when there are arithmetic conditions such as overflow.

The deliverables of this part are:

1. An assembly based 1D-Kalman filter subroutine "`Kalmanfilter_asm`"
2. An **assembly-based test workbench** to call the function and pass parameters.

ARM CALLING CONVENTION

In assembly, parameters for a subroutine are passed on the memory stack and via internal registers. In ARM processors, the scratch registers are R_0:R_3 for integer variables and S_0:S_3 for floating point variables. Up to four parameters are placed in these registers, and the result is placed in R0 - R1 (S0 – S1). If any parameter requires more than 32 bits, then multiple registers are used. If there are no free scratch registers, or the parameter requires more registers than remain, then the parameter is pushed onto the stack. In addition to the class notes, please refer to the document ***"Procedure Call Standard for the ARM Architecture"***, especially its sections **5.3-5.5**. This particular order of passing parameters is eventually a convention applied by specific compilers. Please be aware that the several different procedure calling and ordering conventions exist beyond the one used here, but this procedure call convention is standardized by ARM.

Assembly Code: Initial values for Test Case

For the purposes of this lab for the assembly code, we will initialize the values of ***q***, ***r*** and ***p*** to 0.1 while the gain ***k*** to 0. The initial estimate of ***x*** can be 0. We will use ***f***=0.8 and ***h***=1.2.

Part II – Performance Evaluation against C

You are required to write a more general multi-dimensional Kalman filter in the C language. Then you are to **call both the C and assembly subroutines from main**. You should compare the performance between the two implementations and present an analysis of execution time differences. (Use the time measurement features in Keil). Needless to say, the output of both implementations should match for the 1 dimensional case. Similar to the assembly part, the prototype of your C-function should look as follows:

```
int Kalmanfilter_C(float* InputArray, float* OutputArray,
kalman_state* kstate, int Length, int State_dimension, int
Measurement_dimension)
```

Here:

- ***InputArray*** is the array of measurements (these may be multi-dimensional, so be careful)
- ***OutputArray*** is the array of values x obtained by Kalman fiter calculations over the input field (these may be multi-dimensional, so be careful)
- The ***kstate*** struct contains the Kalman filter state
- Integer ***Length*** specifies the length of the data array to process
- Integer **State_Dimension** specifies the dimension of the state
- Integer **Measurement_Dimension** specifies the dimension of the measurement vector

The function return value encodes whether the function executed properly (by returning 0) or the result is not a correct arithmetic value (e.g., it has run into numerical conditions leading to NaN)

The deliverables of this part are:

1. Assembly and C based Kalman filter functions
2. A **C-based test workbench** to call both functions and test for correctness and speed

Part III– CMSIS-DSP

The residuals $z_k - H\hat{x}_{k|k}$ should not be correlated with the observed data z_k (i.e., the correlation should be close to zero). The residuals should have a mean close to zero and should have a Gaussian distribution. In this part of the lab you will assess whether your implementation of the Kalman filter achieves this.

Calculate the correlation between the residuals and the observed data, the standard deviation of the residuals, and the mean of the residuals. Also calculate the autocorrelation of the residuals (with a lag of one time step and a lag of two time steps).

For each of the above operations, you will implement the required function twice, once in your own C implementation and once using the ***CMSIS-DSP*** library functions. You are required to profile the code and measure the execution speed of your own subroutines against the library subroutines.

How can you test if the residuals are normally distributed?

Part IV– BONUS

Implement an assembly-code version of the multi-dimensional Kalman filter.

Function Requirements for all parts

1. Your code should not use registers/variables unnecessarily whenever you can overwrite registers/variables you do not use anymore.
2. Your code should have minimum code density; that is; it should consume minimum memory footprint.
3. You may use the stack for passing parameters if needed (Assembly part)
4. Your code should run as fast as possible. You should optimize beyond your initial crude implementation.
5. Your code should make use of modular design and function reuse whenever possible
6. Codes will be compared against each other for the above criteria during demo time. Those who achieve best results will ***get the highest demo grades.***
7. The subroutine should be robust and correct for all cases. Grading of the experiment will depend on how efficiently you solve the problem, with all the corner cases (if any) being correct.
8. All registers specified by ARM calling convention are to be preserved, as well as the stack position. It should be unaffected by the subroutine call upon returning.
9. The calling convention will obey that of the compiler.
10. The subroutine should not use any global variables besides any that we have specified (if any).

Linking

1. When creating a new project, it is best to include the startup code, for which the tool will ask you whether to include it. Then, modify that code to branch to the assembly subroutine instead of __main for testing assembly code alone, prior to embedding into C main program. Please note that you will need to declare as exported the subroutine name in your assembly code.
2. If the linker complains about some other missing variable to be imported in startup code, you can either declare it as "dummy" in your assembly code, or comment its mention in the startup code.
3. For linking with C code that has main, none of the above two measures are needed.

Reference Material and Required Reading

- Doc_01 - Cortex-M4 Devices - Generic User Guide
- Doc_02 - Cortex-M4 Programming Manual
- Doc_04 - The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition (Optional)
- Doc_07 - Procedure Call Standard for ARM Architecture
- Doc_08 - ARM® and Thumb®-2 Instruction Set
- Doc_09 - Vector Floating Point Instruction Set
- Doc_16 - Debugging with Keil
- Doc_17 - Introduction to Keil 5.xx
- Doc_24 - Doxygen Manual 1.8.2 (Optional)
- Doc_25 - ProGit 2nd Edition (Optional)

Demonstration and Documentation

The demonstration is performed in two parts. Before midnight September 23, you must demonstrate correct operation of the assembly code (Part 1) to a TA. You can do this during a lab session when a TA is present.

Before midnight September 30, you must demonstrate correct operation of the code from the other parts of the lab (Parts II-III and IV if you choose to do the bonus).

Detailed instructions for the demonstration will be provided in a separate document.

The demonstration involves showing your source code and demonstrating a working program. You should be able to call your subroutine several times consecutively. You should be able to explain what every line in your code does – there will be questions in the demo dealing with all the aspects of your code and its performance. The questions might regard the skeleton code to initiate and start the assembly code that we gave you in Tutorial 1; ask if you do not know!

We expect you to have the full documentation and commenting of your assembly code subroutine completed before demonstrating. Future labs will require substantial documentation following the Doxygen documentation standard.

1. **This lab has a weight of 8 marks. There is NO report associated with this lab. The assembly code should be submitted for review.**