# ECSE426 Microprocessor Systems

# Fall 2015

## Lab 1: Zero-velocity Detection and the Viterbi Algorithm

## 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, such 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, specifically purely assembly code projects. The tutorial will show you how to let the tool insert 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.** But since the reference material is huge (hundreds of pages over the course of the semester), make sure to attend the tutorials so that the TA can guide you where to look.

## Background on pedestrian navigation and zero-velocity detection

When GPS signals are unavailable, one method for determining the position of a pedestrian is to use an inertial navigation algorithm based on the measurements from accelerometers and gyroscopes (and possibly magnetometers). One of the problems with such algorithms is accumulated error in the position and velocity estimates. In order to address this problem, localization techniques try to take advantage of the known characteristics of the human gait. In particular, there are times when the foot is at rest when the velocity is zero. By detecting these times and setting the velocity back to zero, the accumulation of error can be reduced. This technique is referred to as zero velocity updating.

One way of detecting the times when the foot is at rest is to define several states corresponding to (1) foot at rest; (2) foot lifting; (3) foot moving; (4) foot being placed back on the ground. When a person is walking, there should be a cycle through these states 1-2-3-4-1- etc.

The states are characterized by different y-axis gyroscope values (positive for the foot leaving or returning to ground, negative for the swinging motion and close to zero when the foot is stationary). The paper [1] presents the details of the technique. The first step is to segment the gyroscope data by comparing each value to a set of thresholds. The k-th segment has an associated value $Y_k$ that is determined by its relationship to the thresholds.

Once the algorithm has identified the $Y_k$ values, the next step is to identify the associated states $X_k$. This can be achieved by applying a Viterbi algorithm, as described in the following section.

The final step is to use the identified states to identify the periods of time when the velocity should be set to zero.

NOTE: The paper [1] first describes the procedure for processing the raw readings to derive observations $Y_k$ and then proposes the use of a Viterbi algorithm to estimate the $X_k$. In this lab we will go in the opposite direction. In Parts I and II of the lab, we will assume that we have a single observation Y, and we will develop the estimation code to process it. In Part III we will return to the task of processing the raw measurements to derive a sequence of observations.

## Background on the Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm that can be used to determine the most probable sequence of hidden states in a hidden Markov model (HMM). In the pedestrian tracking example, the hidden states are the $X_k$, and the observations are the $Y_k$. The goal is to infer the most probable sequence of $X_k$, k=1,…,T.

If all possible sequences were explored, the computational cost would be exorbitant (with four different states, there are $4^T$ paths to evaluate). The Viterbi algorithm takes advantage of the Markov property of the HMM to avoid evaluating the probability of all sequences. It executes with a computational cost that is linear in the number of observations and quadratic in the number of states.

Python code for the Viterbi algorithm is provided below. The lectures will discuss hidden Markov models and the equations underpinning the Viterbi algorithm in more detail. The code below assumes that self (an hmm) contains an array of prior probabilities for the individual states at time 0 (self.priors); a matrix specifying the probability of transitioning from one state to another (self.transition); and a matrix specifying the conditional probability of an observation given a particular state (self.emission).

While the code in Fig. 1 is functional, and it can be directly used within a larger (Python) program, it is used here as a compact high-level specification. You can use it as a basis for development in C, but you will need to be careful with data types and function prototypes.

```
def viterbi (self,observations):
    """Return the best path, given an HMM model and a sequence of observations"""
    # A - initialisation
    nSamples = len(observations)
    nStates = self.transition.shape[0] # number of states
    c = np.zeros(nSamples) # scale factors (necessary to prevent underflow)
    vit = np.zeros((nStates,nSamples)) # initialise viterbi table
    psi = np.zeros((nStates,nSamples)) # initialise the best path table
    vit_path = np.zeros(nSamples); # initialize best sequence

    # B - insert initial values into viterbi and best path (bp) tables
    vit[:,0] = self.priors.T * self.emission[:,observations(0)]
    c[0] = 1.0/np.sum(vit[:,0])
    vit[:,0] = c[0] * vit[:,0] # apply the scaling factor
    psi[0] = 0;

    # C- Viterbi iterations for time>0 until T
    for t in range(1,nSamples): # loop through time
        for s in range (0,nStates): # loop through the states
            trans_p = vit[:,t-1] * self.transition[:,s]
            psi[s,t], vit[s,t] = max(enumerate(trans_p), key=operator.itemgetter(1))
            vit[s,t] = vit[s,t]*self.emission[s,observations(t)]

        c[t] = 1.0/np.sum(vit[:,t]) # scaling factor
        vit[:,t] = c[t] * vit[:,t]

    # D - Back-tracking
    vit_path[nSamples-1] =  vit[:,nSamples-1].argmax() # last state
    for t in range(nSamples-1,0,-1): # states of (last-1)th to 0th time step
        vit_path[t-1] = psi[vit_path[t],t]

   return vit_path
```

**Fig. 1** Python code for the (based on an implementation provided at stackoverflow (Zhubarb): http://stackoverflow.com/questions/9729968/python-implementation-of-viterbi-algorithm )

## The Experiment

### Part I – Pure Assembly Programming

You are required to write an assembly subroutine "`ViterbiUpdate_asm`" in ARM Cortex M4 assembly language that processes one observation to update the Viterbi variables (psi and vit). This corresponds to the python lines:

```
for s in range (0,nStates): # loop through the states
    trans_p = vit[:,t-1] * self.transition[:,s]
   psi[s,t], vit[s,t] = max(enumerate(trans_p), key=operator.itemgetter(1))
   vit[s,t] = vit[s,t]*self.emission[s,observations(t)]

c[t] = 1.0/np.sum(vit[:,t]) # scaling factor
vit[:,t] = c[t] * vit[:,t]
```

Let S denote the number of states and V the number of observation-types. The inputs are thus the S-entry vector vit[:,t-1], the SxS transition probability matrix self.transition[:,:], and the SxV observation probability matrix self.emission[:,:]. We will combine vit and psi into a 2S-entry vector vitpsi.

You should use the built-in floating-point unit by using the existing floating-point assembler instructions. Your assembly viterbi update subroutine will take four parameters:

1. A pointer to the vitpsi[:,t-1] vector
2. A pointer to the vitpsi[:,t] vector (for output)
3. The observation
4. A pointer to the hmm variables (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 hmm structure consists of two integers (the number of states and the number of observation types) and two matrices of single-precision floating-point numbers (the SxS matrix transition, and the SxV matrix emission). It is convenient to keep these state variables in a single C language `struct`, consisting of:

```
integer S; //number of states
integer V; //number of observation types
float[][] transition; // SxS transition probability matrix
float[][] emission; // SxV emission probability matrix
float[] prior; // Sx1 prior probability matrix
```

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

The deliverables of this part are:

1. An assembly based Viterbi algorithm update subroutine "`ViterbiUpdate_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.

## Part II – Performance Evaluation against C

You are required to write the same subroutine described in the previous part in the C language. As for the assembly subroutine in Part I, this subroutine should process ONE observation and execute the associated Viterbi update step. Then you are to **call both the C and assembly subroutines from main**. You should compare the performance between the two implementations and present analysis on execution time differences. (Use the time measurement features in Keil). Needless to say, the output of both implementations should match. Similar to the assembly part, the prototype of your C-function should look as follows:

```
int ViterbiUpdate_C(float* InputArray, float* OutputArray,
hmm_desc* hmm, int Observation)
```

Here:

- ***InputArray*** is the array (vector) vitpsi[:,t-1]
- ***OutputArray*** is the array (vector) vitpsi[:,t] to be updated
- ***The hmm struct*** contains the parameters of the hidden Markov model (dimensions, transition probabilities, and emission probabilities)
- ***Observation*** contains the observation

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 Viterbi update functions
2. A **C-based test workbench** to call both functions and test for correctness and speed

## Part III– Viterbi Code and Pedestrian Navigation

In this part of the experiment, you will perform two tasks. First, you will develop a complete Viterbi decoder in C that processes a sequence of observations and a specified hidden Markov model to identify the most probable sequence of states. The function should have a prototype of the form:

```
int Viterbi_C(int* Observations, int Nobs, int* EstimatedStates,
hmm_desc* hmm)
```

In this case, `Nobs` specifies the number of observations, and `EstimatedStates` is a pointer to where the output should be written (an array of `Nobs` state estimates).

In the second task, you will develop code that can process a stream of accelerometer readings to identify segments (following the approach described in Section 3 of [1]). Be careful because [1] uses the transposes of the matrices we work with in this lab. By applying thresholds you will associate a $Y_k$ with each identified segment. You will be provided with example data for development and testing purposes and we will specify the thresholds you should use.

You may wish to consider the application of smoothing filters to address the case when the data is noisy. If so, there should be an option in your code to turn the smoothing on and off. Your function should calculate and store the start and end of each segment. It should return the number of segments or an error code if there is a problem during execution. It should generate and store a sequence of integer-valued observation labels.

**Bonus**: For bonus marks, you should rewrite the code to make use of the vector functions in the CMSIS DSP library whenever possible (and sensible). You will need your original code for demonstration purposes.

## 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 those that we have specified.

## 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.

## Demonstration and Documentation

On demonstration day, you will be provided with a header file ***test.h.*** This will contain the test data.

**Test 1**: You will be provided with a test array ***vitTestArray[]*** of a known length **T.** This will contain a set of observations $Y_k$, $k = 1,\ldots,T$, where each $Y_k$ takes a value from the set {1,2,3}. You will also be provided with three hidden Markov model specifications (`hmm_desc* hmm1, hmm2, hmm3)` with different priors, transition and emission matrices. In each case there will be four possible states. Your Viterbi code should generate the correct output (sequence of state estimates $X_k$) for each hmm.

**Test 2**: You will be provided with accelerometer data **accObs[]** of known length **accT**, an associated hmm, and thresholds. There will be three observation states {1,2,3} and the thresholds will be as in [1], alpha1, alpha2, N1, N2, etc. You should be able to call a function to generate an observation sequence $Y_k$, and then call your Viterbi decoder to determine the most probable state sequence.

The demonstration involves showing your source code and demonstrating a working program. You should be able to call your subroutine (and other functions) 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 also refer to any skeleton code that you are provided with; ask if you do not completely understand how it works! You should also be prepared to show how you evaluated the execution times of your assembly and C code.

It is essential that you have complete documentation for your assembly code subroutine when you demonstrate your work to the TA. Future labs will require careful and thorough documentation. The Doxygen documentation standard should be followed throughout the course.

1. **Demos will be held on Friday, Oct. 2nd, 2015. This lab has a weight of 6 marks. There is NO report associated with this lab. The assembly code <u>should</u> be submitted for review.**

2. Demo slots will be announced and students will reserve their own preferred slot on Friday. Students should show up and be ready for demo 10 minutes prior to their reserved slot. **T.A.s will not wait for you** if you are late and will move on to the next ready group. You will demo on Monday with a substantial penalty (35%) so make sure you are ready on time.

3. Early demos are allowed (whenever possible) upon appointment. Contact the T.A. whose schedule falls on your preferred demo day to check if a demo is possible. There is a cap on the max number of early demos per day.

## References

[1] S. K. Park and Y. S. Soo, "A zero velocity detection algorithm using inertial sensors for pedestrian navigation systems", Sensors, vol. 10, pp. 9163-9178, 2010.