# Lab #5 DSP Number Systems

## Objectives:

1. Explore how several data types commonly used in DSP (byte, signed integer, long integer, float) affect memory usage and computation speed..
2. Explore the effects of finite precision math on calculations.
3. Measure the execution speed for addition and multiplication, and compare how the speed changes using the various number data types.

## Reading:

Chapter 4 in Smith. http://Arduino.cc

## Intended Learning Outcomes:

|  |  |
| --- | --- |
| Demonstrate the use of an internal timer function to measure the execution time for a section of code. |  |
| Understand the impact of data types on execution speed and on memory usage. |  |

## Overview and Background Material:

In the previous lab on Analog to Digital Conversion (ADC), you were able to collect temperature reading values and average values to increase the effective resolution of the temperature measurement.

In embedded systems, the averaging must be done in real-time in a microprocessor or digital signal processing chip, not in a post-processing step such as MATLAB analysis. Consider the example of a medical instrument that is actively monitoring a patient. The averaging algorithm will have to execute quickly (e.g. < 1 millisecond) on a microprocessor chip with limited memory and speed rather than on a PC with 16GB RAM and a 3 GHz i7 processor.

How fast an algorithm runs depends on the number of instructions that have to be executed and how long each instruction takes to execute. Some instructions are faster than others e.g. addition in a microcontroller is faster than multiplication.

The datatype of the variable also affects how long it takes to execute an instruction. Typically, adding two integers is faster than adding two floating point numbers. However, it depends on how the microprocessor or DSP chip is built. If the chip has a hardware floating point processor then it may be able to quickly perform floating point calculations. Otherwise, floating point operations are typically slower than fixed point (integer) operations.

Knowing the execution speed of your particular microprocessor or DSP chip is critical because it affects the number of filter calculations that can be run in a given amount of time. If the filter takes longer to run, the bandwidth of the system will be lower. For example, you may not be able to perform a particular audio processing algorithm and have it run at the MP3 standard sample rate of 44.1kHz if there are too many calculations required for each sample.

Characterizing your system’s execution time can help you understand why it may misbehave when trying to run a particularly long filter.

**NOTE:** For this lab it will be easier to use the serial monitor than MATLAB to capture output data. You will make tables of results and it is easiest to use Excel for this rather than MATLAB. A set of blank excel tables has been provided for you in the file “Lab 05 Tables.xlsx” located in myCourses.

# Section 1 Basic Math Operations and Data Types

In this section, you will add two numbers and examine the results obtained when different data types are used.

## Description of the code:

This code adds two numbers and prints the results.

## Procedure:

1. Paste the program code from the following text box into the Arduino IDE. This code adds two numbers and prints the results. You will not use the temperature sensor or the dither circuit.
2. Run the code initially using byte variables.
3. Record the results in the **Table 3-1a.**
4. Next, add your own code to the main loop, as indicated by the TODO, to run through a series of calculations using an array rather than manually changing the numbers each time. You may see some unexpected results.
5. Repeat step 4 using int, long, and float datatypes. Record the results in **Table 3-1a**.
6. In the first row of **Table 3-1b**, determine what positive integer value for the ‘b’ number will result in a ‘yv’ that is “254” when the datatype is byte. You do not need to fill in other values for this row.
7. In the last row of **Table 3-1b**, determine what positive integer value for the ‘b’ number will result in a ‘yv’ that is “-10” when the datatype is long. You do not need to fill in other values for this row.

**Question 3-0:** Were any of the floating point calculations for “yv” incorrect (i.e. different than you would expect)? If so, explain why. If not, explain why not.

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Datatypes.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// modified by: Clark Hochgraf Sept 28, 2019 -- added number arrays

// purpose: Illustration of datatype effect on adding numbers

#define DATATYPE byte // change declaration to byte, int, long, float

DATATYPE av, bv, yv; // uses #define value to change datatype for each variable

// change declaration to byte, int, long, float

DATATYPE avl[16] = {100, 200, 20000, 100, 100, 100, 255, 255, 25500, 20000, 20000, 20000, 20000, 1000000000, 2000000000, 2000000000};

DATATYPE bvl[16] = {100, 200, 20000, 155, 156, 157, 100, 10, 25500, 12767, 12768, 12769, 12770, 1000000000, 147483649, 147483649};

void setup()

{

Serial.begin(9600);

Serial.println(F("Lab 3 datatypes190928"));

Serial.println(F("\nEnter 'g' to go ....."));

while (Serial.read() != 'g'); // spin until 'g' entry

Serial.println("Data type is byte"); // byte, int, long, float

av = 100;

bv = 100;

yv = av + bv;

Serial.println("Add two numbers (100+100)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

av = 200;

bv = 200;

yv = av + bv;

Serial.println("\nAdd two numbers (200+200)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

av = 20000;

bv = 20000;

yv = av + bv;

Serial.println("\nAdd two numbers (20,000+20,000)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

}

void loop() {

Serial.println("\nAdd numbers from array");

// TODO: Add a for-loop to run through all the 'a' and 'b' values in the

// arrays 'avl' and 'bvl'. Start at index 0 and run through all 16 values in the array.

// Inside the for-loop:

// 1) set av and bv to values picked from 'avl' and 'bvl'

// 2) perform the addition

// 3) print the a and b values and the result yv

// INSERT YOUR CODE HERE

// END INSERTION OF CODE

while (true) {}; // spin forever

}

**Table 3-1a**.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| input  a | input  b | expected result a+b | actual result byte | actual result int | actual result long | actual  result float |
| 100 | 100 | 200 |  |  |  |  |
| 200 | 200 | 400 |  |  |  |  |
| 20,000 | 20,000 | 40,000 |  |  |  |  |
| 100 | 155 | ? |  |  |  |  |
| 100 | 156 | ? |  |  |  |  |
| 100 | 157 | ? |  |  |  |  |
| 255 | 100 | ? |  |  |  |  |
| 255 | 10 | ? |  |  |  |  |
| 25,500 | 25,500 | ? |  |  |  |  |
| 20,000 | 12,767 | ? |  |  |  |  |
| 20,000 | 12,768 | ? |  |  |  |  |
| 20,000 | 12,769 | ? |  |  |  |  |
| 20,000 | 12,770 | ? |  |  |  |  |
| 1,000,000, 000 | 1,000,000, 000 | ? |  |  |  |  |
| 2,000,000, 000 | 147,483, 649 | ? |  |  |  |  |
| 2,000,000,000 | 147483649 | ? |  |  |  |  |

Table 3-1b.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| input  a | Find Positive number b | expected result a+b | actual result byte | actual result int | actual result long | actual  result float |
| 255 | (?) | //skip// | 254 | //skip// | //skip// | //skip// |
| 2,000,000,000 | (?) | //skip// | //skip// | //skip// | -10 | //skip// |

# Section 2 Illustration of Roundoff Error With Floats

The roundoff error associated with floating point data types can be illustrated by adding a random number to a float value and then subtracting the same number off later. In this case we add the random numbers A and B to a constant value of 1.0 and then later subtract them both off from the original number. By comparing the ideal value to the actual value, you can see the roundoff error.

## Procedure:

1. Read the code from the textbox below to understand how it works.
2. Paste the code from the textbox into the Arduino IDE, load and run the file.
3. Compare the ideal output value and the actual value and record the results
4. Modify the code so that the scale factor for the random numbers A and B is 10.0 instead of 1.0. Compile, upload and run the code and note how the error changes.
5. Compare the ideal output value and the actual value and record the results for the different random number scale factors listed in the table.

**Table 3-2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scale factor for random numbers | Starting value of x | Ending value of x | The total error in percent after 1000 calculations | Average error in percent for one calculation |
| 1.0 | 1.0000000000 |  |  |  |
| 10.0 | 1.0000000000 |  |  |  |
| 100.0 | 1.0000000000 |  |  |  |
| 1000.0 | 1.0000000000 |  |  |  |
| 10000.0 | 1.0000000000 |  |  |  |
| 100000.0 | 1.0000000000 |  |  |  |

**Q3-1 If the signal is the original value of x and the final value of x contains the signal plus noise, what is the resulting signal to noise ratio (in dB) of the final result for when the random scale factor is 10,000.0?**

**Q3-2 Explain why the difference between the starting value of X (1.0) and the ending value of X was larger when the random numbers (A and B) were larger? In other words, why does the size of A and B affect the result?**

**Q3-3 Your lab partner says that if you use floating point numbers, that the calculations will always be accurate enough. Write a paragraph illustrating a case where this is not true and write a paragraph explaining what feature or characteristic of floating point numbers causes the problem.**

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Roundoff\_1.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// purpose: Illustration of roundoff error when using float

const int NUM\_CALC = 1000;

float xv = 0.0, A, B, error, scale\_factor;

void setup()

{

Serial.begin(9600);

Serial.println(F("Lab 3 P2 datatypes 190928"));

Serial.println(F("\nEnter 'g' to go ....."));

while (Serial.read() != 'g'); // spin until 'g' entry

randomSeed(425);

scale\_factor = 100000.0; // start with 1.0

xv = 1.0;

Serial.print("Adding random floating point numbers with scale factor ");

Serial.println(scale\_factor);

Serial.print("x value starts at x = ");

Serial.println(xv,10); // print to 10 decimal places

for (int i = 1; i < NUM\_CALC; i++)

{

A = random\_float() \* scale\_factor;

B = random\_float() \* scale\_factor;

xv = xv+A; // add A to x

xv = xv+B; // add B to x

xv = xv-A; // subtract A from x

xv = xv-B; // subtract B from x

} // for

error=(1.0-xv);

// end result should just be the original value of x (i.e. 1.0)

Serial.print("x value finishes at x = "); Serial.println(xv,10);

Serial.print("Total percent error after 1000 calculations = ");

Serial.print(error\*100.0,10); Serial.println(" %");

Serial.print("Average percent error per addition or subtraction = ");

Serial.print(error\*100.0/(4.0\*NUM\_CALC),10); Serial.println(" %");

Serial.println();

} // setup()

void loop(){} // spin forever

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

float random\_float() { return (random(2147483648)/2147483648.0); }

# Section 3 Unexpected Behavior Due to Finite Precision Effects With Floats

## Overview

The finite precision associated with floating point data types can cause unexpected results in many ways. Examine the code in the textbox below. The difference between two floating point numbers A=18000002.0 and B=18000001.0 is exactly 1. The two numbers are first printed and then the difference between the two numbers is calculated and printed.

Next in the code, the value of A is set to a counter variable, and B is set to the counter variable minus 1. The counter value is continuously incremented until the value of A-B does not equal exactly 1, which should never happen resulting in an infinite loop. Let the code run a while and observe what happens.

**Procedure:**

1. Read the code from the textbox below to understand how it works.
2. Paste the code from the textbox into the Arduino IDE, load and run the file.
3. Examine the results. Answer questions Q3-4 and Q3-5
4. Modify the code so that variables A and B have long datatypes instead of float datatypes and change the definition of the initial values of A and B to be

A=18000002;

B=18000001;

Not the original:

A=18000002.0;

B=18000001.0;

1. Upload the code and observe the results. Answer questions Q3-6, Q3-7
2. Modify the code so that variables A and B have long datatypes instead of float but change the definition of the initial values of A and B to be

A=18000002.0;

B=18000001.0;

Be sure to include the decimal points. Upload and run the code. How are the results for the first section different than when no decimal point was included? Answer question Q3-8

**Q3-4 In the first section of the code, why doesn’t the value of B print out as expected?**

**Q3-5 In the second section of the code, at what value of the counter does the program produce an unexpected result? Why does the error occur at this particular value of the counter? (hint: What is 2^24?)**

**Q3-6 When the data type were set to** long**, did the computer calculate the correct value of A-B when A= 18000002, and B=18000001?**

**Q3-7 When the datatypes were set to** long**, did the computer break out of the while loop at the same value of counter as when the datatype was** float**? What did happen?**

**Q3-8 What happened when the datatypes were set to** long**, but the initial values of A and B included a decimal point? Why did this happen?**

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Roundoff\_2.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// modified by: Mark Thompson December 31, 2019

// purpose: Illustration of unexpected behavior due to roundoff error

float A = 18000002.0;

float B = 18000001.0;

//long A = 18000002;

//long B = 18000001;

//long A = 18000002.0;

//long B = 18000001.0;

void setup()

{

Serial.begin(9600);

Serial.print("A = float(18000002.0) value printed out is "); Serial.println(A);

Serial.print("B = float(18000001.0) value printed out is "); Serial.println(B);

Serial.print("A - B is "); Serial.println(A-B);

//-------------------------------------------------------------------

Serial.println("\nAnother illustration: Counting up by 1 ");

Serial.println("A is the counter value");

Serial.println("B is the counter value minus 1");

Serial.println();

long counter=16000000;

A = counter;

B = A-1.0;

Serial.print("Starting at a counter value of = "); Serial.println(counter);

Serial.print("A value is "); Serial.println(A);

Serial.print("B value is "); Serial.println(B);

Serial.print("A - B is "); Serial.println(A-B);

Serial.println();

Serial.println("Continue to count up by 1. Print the counter every 50,000 counts");

Serial.println("Check to see that the differnce between A and B is always 1, error otherwise");

Serial.println();

while ((A-B) == 1)

{

if (counter%50000 == 0) Serial.println(counter);

counter = counter+1;

A=counter;

B=counter-1;

}

Serial.print("\nUnexpected result: Counter = ");

Serial.println(counter);

Serial.print("A value is "); Serial.println(A);

Serial.print("B value is "); Serial.println(B);

Serial.print("A - B is "); Serial.println(A-B);

} // setup

void loop() {} // spin forever

# Section 4 Finite Precision Effects on Sinewave Calculations

## Overview

Finite precision effects of the floating point number system can be seen when calculating sine and cosine functions. The value of a sine wave is zero when the angle is zero radians and when the angle is a multiple of radians. In this section, the Arduino is used to calculate the value of a sine wave when the angle is, etc. The result should always be zero, but due to finite precision effects, the resulting sine value is not zero.

## Procedure:

1. Read the code from the textbox below to understand how it works.
2. Paste the code from the textbox into the Arduino IDE, load and run the file.
3. Examine the results. Answer questions Q3-9, Q3-10.
4. Modify the code so the for loop starts at i=0 instead of i=1.Upload and run the code. Answer question Q3-11.

**Q3-9 What was the size of the error in the sin() calculation when the angle was ? What was the size of the error in the sin() calculation when the angle was ? Where the errors the same size?**

**Q3-10 Consider the case where you are generating a digital sinewave in your code. If the angle argument to the sine function is increasing to value that is several hundred times Pi, is there a concern about the accuracy of the resulting sine value?**

**Q3-11 What is the value of the sine function as generated by the Arduino when the angle is zero (i.e.)? Is this more accurate than the value of the sine function generated by the Arduino when the angle is? For the best accuracy, to what range of angle values the sine function be limited? Note that for any angle greater than, the value of could be subtracted from the angle and the result is theoretically the same.**

**P3-1** **Plot your Arduino generated data for sin(** ) versus n = 0…1000 without modifying the argument. Upgrade your sketch with argument modifications to maximize accuracy and plot your new results on the same axes.

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Sinewave\_zero.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// purpose: Illustration of roundoff error affecting sine

#include "Math.h" // for accessing sine and PI

float result;

void setup()

{

Serial.begin(9600);

Serial.println("Prints values of sin(2\*pi\*i) where i is integer");

Serial.println("Theoretically, the value should always be zero");

Serial.print("\ni\tsine(n2pi)\n");

for (int i = 1; i < 1000; i+=10)

{

result=sin(TWO\_PI\*i); // TWO\_PI is Arduino constant

Serial.print(i); Serial.print('\t');

Serial.println(result,20); // print 20 decimal places

}

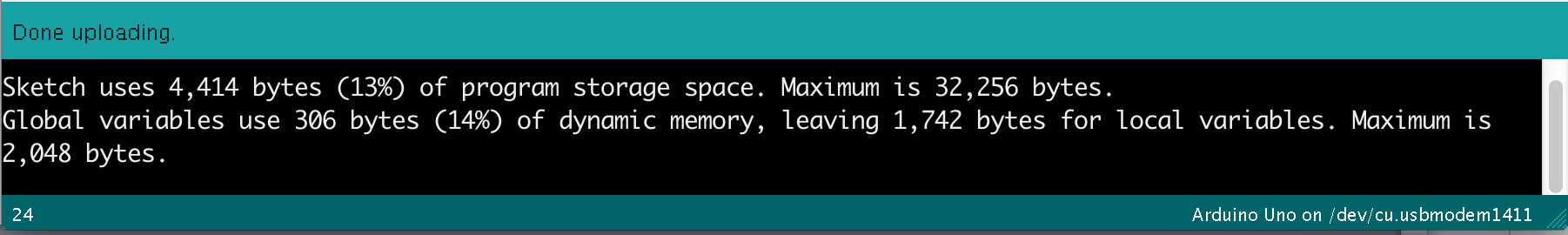
} // setup

void loop() {} // spin forever

# Section 5: Measuring Execution Time and Memory Usage versus Datatype

## Overview

The Arduino Uno microprocessor has a RAM memory space of 2048 bytes. While this may seem like a lot, it is very easy to use it all up. If the free RAM memory falls below 300 bytes, the microprocessor may produce irregular results or completely stop working. The amount of RAM usage is estimated and displayed each time you compile (Arduino IDE 1.57 or higher). An example is shown in the textbox below



Note that there are two types of memory usage displayed by the Arduino IDE. The first memory usage refers to FLASH memory where the program code is stored. The UNO has 32kB of FLASH for storing the program. The second is the RAM memory for storing variables. The UNO has 2kB of RAM. You need to make sure your program does not use up all the RAM as the processor needs about 300 bytes of RAM for stack operations, etc.

## Execution Time

In this section you will also measure the time it takes for a section of code to execute. The Arduino’s internal timer 0 can be used to count the number of microseconds that have elapsed since the microprocessor was reset. The timer value can be accessed by calling micros().

To measure how long a section of code takes to execute, first call micros() and record its value. Then after the code runs, call micros() again to record the clock at the end of the routine. The difference between these gives you the elapsed time in microseconds.

Using this approach, you will measure how long it takes to add two numbers, to multiply two numbers, and to multiply and add in one combined operation (called multiply and accumulate or MAC). The amount of time each operation takes will depend on the datatype (byte, integer, long, float) of the numbers you put into operation.

To change the datatype, we will use the **cast** function in C. For example, you can temporarily change an integer datatype to a float data type using the C statement

xv[i]=(float)i\*(float)i;

This takes the integer variable i and makes it a float datatype for the calculation.

If the x variable is already a float, then the memory usage and execution speed will reflect the float datatype. To make the calculation with long datatype, change the code to

xv[i]=(long)i\*(long)i;

and declare the xv[] array as a long datatype.

long xv[DATA\_LEN]={0}; // array of values initialized to zero

**For this lab, the code is set up so you only need to change the datatype of result and xv[]. You should not need the cast function.**

Alternative Algorithm Timing Measurement Method

Sometimes you may not be able to use the micros() command to measure execution time. For example, if your code modifies the operation of Timer 0 in the Arduino chip. Another way to measure algorithm execution time is to use a spare digital I/O pin and set it HIGH when the algorithm starts and then set it LOW when the algorithm ends. Use an oscilloscope to measure the pin pulse width to ascertain run time.

startTime = micros(); becomes digitalWrite(timingPin, HIGH);

end Time = micros(); becomes digitalWrite(timingPin, LOW);

## Program Description

Unpack the code in the text box below to a new Arduino sketch. Variables are declared to store the time values as measured in microseconds since the microprocessor was reset. The number of variables stored is changed by changing the value of

const int DATA\_LEN = 200;

To modify the input array and result variable’s datatype, change the datatype in line

float xv[DATA\_LEN] = {0}, res = 1.0;

The execution time of adds, multiplies and multiply-accumulates is calculated using three different methods.

1. First, a set of 20 adds is computed without using a for loop. The timing result is calculated and displayed. The process is repeated for multiplications and multiply accumulates.
2. Second, a for loop is setup to perform the operations and the timing results are displayed.
3. Third, a for loop is used where the variables are arrays, indexed by the for loop iterator variable.

When using the for-loop, the calculations take longer because of the loop instruction’s overhead. To get the execution time per operation, total time is divided by the total number of operations.

// OPEN A NEW SKETCH WINDOW IN ARDUINO

// CLICK IN THIS BOX, CTL-A, CTL-C (Copy code)

// CLICK IN SKETCH, CTL-A, CTL-V (Paste code into sketch)

// file: Timing.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// purpose: Measuring execution time and memory usage as datatype is

// changed from byte to integer, long, float

const int DATA\_LEN = 200; // number of data array values

unsigned long startUsec, endUsec, execUsec;

byte xv[DATA\_LEN]={0}, seed = 1.01, res; // array initialized to zero

void setup()

{

Serial.begin(9600);

Serial.print("Data array length = "); Serial.println(DATA\_LEN);

res = seed;

startUsec=micros();

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

endUsec = micros();

execUsec = endUsec-startUsec;

Serial.print("\nuSec per individual addition = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

execUsec = micros()-startUsec;

Serial.print("uSec per individual multiplication = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

execUsec = micros()-startUsec;

Serial.print("uSec per individual multiply and accumulate = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++)

{

res = res+res;

// Serial.print(i); Serial.print('\t');

// Serial.println(res);

}

execUsec = micros()-startUsec;

Serial.print("uSec per loop addition = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ res = res\*res; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop multiplication = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values and add to original value (accumulate)

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ res = res+res\*res; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop multiply and accumulate = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// Add two values using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]+xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array addition = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]\*xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array multiplication = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values and add to original value (accumulate) using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]+xv[i]\*xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array multiply and accumulate = ");

Serial.println((float)execUsec/DATA\_LEN,2);

}

void loop(){ } // spin forever

## Procedure

1. Paste the file from the textbox into the Arduino IDE. Make sure to select all the text in the textbox. (Use CTRL-A, then CTRL-C.)
2. Change the datatype from float to integer, long and byte. Remember to check that the right values are being printed and divided by the number of additions, etc.
3. **Record the data on the microprocessor performance in table similar to Table 3-3 below.** For the speed multipliers**, calculate them in Excel** **after you have collected all the data**. The speed multiplier indicates how much faster an operation is relative to the corresponding floating point operation. For example, addition of two bytes is about 10.6 times faster than the addition of two floats. The values in the column labeled FLOAT are example values. Your results may vary from what is currently in the table.

Table 3-3 Memory and execution times as a function of input variable datatypes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Summary Table |  |  |  |  |
| Datatype | byte | int | long | float |
| Free Memory (bytes) |  |  |  |  |
| bytes per variable for given datatype |  |  |  |  |
| SPEED USING INDIVIDUAL OPERATIONS |  |  |  |  |
| microseconds per addition |  |  |  |  |
| microseconds per multiply |  |  |  |  |
| microseconds per multiply and accumulate |  |  |  |  |
| Speed multiplier for addition |  |  |  |  |
| Speed multiplier for multiply and accumulate |  |  |  |  |
| SPEED USING FOR LOOP  (200 iterations, DATA\_LEN = 200) |  |  |  |  |
| microseconds per addition |  |  |  |  |
| microseconds per multiply |  |  |  |  |
| microseconds per multiply and accumulate |  |  |  |  |
| Speed multiplier for addition |  |  |  |  |
| Speed multiplier for multiply and accumulate |  | | | |
| SPEED USING FOR LOOP and ARRAY Storage  (200 iterations, DATA\_LEN = 200) |  |  |  |  |
| microseconds per addition |  |  |  |  |
| microseconds per multiply |  |  |  |  |
| microseconds per multiply and accumulate |  |  |  |  |
| Speed multiplier for addition |  |  |  |  |
| Speed multiplier for multiply and accumulate |  |  |  |  |

**Observations**

Generally, floating point operations are the slowest to execute. While byte, integer and longs are faster, you must use them carefully to ensure they produce valid results. One approach is using integer datatypes is to pre-scale numerical values coming into the algorithm so that the results stay within the dynamic range of the datatype (e.g. -32768 to 32767 for integers).

**Tips**

1. The bytes per variable is a number that you can look up in the Arduino documentation online or find on the Arduino user support forum.
2. You can also calculate the bytes per variable by running the program with DATA\_LEN =1 and recording the number of bytes of free memory. Then increase DATA\_LEN to 101. Now look at the change in free memory. Divide the number of additional bytes used by 100 (the number of additional variables) and you can get the bytes per variable.

# Section 6 Convolution Implementation – Measuring Execution Time

## Overview

Convolution is one of the key operations using in signal processing. For example, it is used in the implementation of Finite Impulse Response (FIR) digital filters.

In this section, you will measure the execution time of a convolution algorithm and its memory usage as a function of variable datatype.

The convolution sum equation is calculated as a sum of products, very similar to the multiply and accumulate instruction previously analyzed. Points in the impulse response, stored in h, are multiplied by the points in the input signal x, and summed together to create the output y.

The simplest form of a convolution is the moving average filter, discussed in Smith chapter 15. The moving average filter simply provides an average of the input signal over M samples. All the impulse response coefficients, h, are identical and are equal to 1/M. For example, a 10 point moving average filter has h coefficients that are all equal to 0.1. For this section, you will implement a moving average filter. If the input to the filter is a constant, then the expected output of the filter is just the average value of the input signal, i.e. the same constant value.

## Program Description

Unpack the code in the following text box to a new Arduino sketch.

// OPEN A NEW SKETCH WINDOW IN ARDUINO

// CLICK IN THIS BOX, CTL-A, CTL-C (Copy code)

// CLICK IN SKETCH, CTL-A, CTL-V (Paste code into sketch)

// file: Convolution.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// purpose: Convolution implementation

// x data stored in float,

// h impulse response stored in float

// y result stored in float

unsigned long startTime, endTime, execTime; // micros() casting

const int IMP\_RESP\_LEN = 30;

const int DATA\_LEN = 100;

// initialize all arrays to zero

float xv[DATA\_LEN]={0}; // convolution input: x variable

float yv[DATA\_LEN]={0}; // convolution output: y variable

float h[IMP\_RESP\_LEN]={0}; // filter impulse response

void setup()

{

Serial.begin(9600);

Serial.println("x variable\nn\txv[n]");

for (int i = 0; i < DATA\_LEN; i++)

{

xv[i] = 5.0;

Serial.print(i); Serial.print('\t');

Serial.println(xv[i]);

}

Serial.println("\nh impulse response\nn\th[n]");

for (int i = 0; i < IMP\_RESP\_LEN; i++)

{

h[i] = 1.0/IMP\_RESP\_LEN;

Serial.print(i); Serial.print('\t');

Serial.println(h[i],4);

}

// perform sum of products

// start convolution only where data is valid

// first IMP\_RESP\_LEN-1 datapoints are not valid

startTime = micros();

for (int k = IMP\_RESP\_LEN-1; k < DATA\_LEN; k++){

for (int i = 0; i < IMP\_RESP\_LEN; i++){

yv[k] = yv[k]+h[i]\*xv[k-i];

}

}

execTime = micros()-startTime;

Serial.println("\nN\txv\tyv");

for (int i = 0; i < DATA\_LEN; i++)

{

Serial.print(i); Serial.print('\t');

Serial.print(xv[i],4); Serial.print('\t');

Serial.println(yv[i],4);

}

Serial.println("\nFor given data type:");

Serial.println("------------------------");

Serial.print("Data length = ");

Serial.println(DATA\_LEN);

Serial.print("final value of yv = ");

Serial.println(yv[DATA\_LEN-1]);

Serial.print("microseconds for each new datapoint = ");

Serial.println(execTime/(DATA\_LEN-(IMP\_RESP\_LEN-1)));

Serial.print("Max update rate (Hz) = ");

Serial.println(1000000/(execTime/(DATA\_LEN-(IMP\_RESP\_LEN-1))));

}

void loop(){ } // spin forever

The file declares variable arrays xv[]to store input data, h[] to store the impulse response, and yv[] to store the result of the convolution sum. Input xv[] is initialized to a constant value of 5.0 in a for loop(). The input values could come from the ADC, but to make it easier to see what is going on, we’ll use a constant value for now.

A second FOR loop creates the impulse response of the filter. In this case, it is a moving average filter where each point in the impulse response has the same value, equal to the inverse of the number of points in the impulse response: h[i]=1.0/DATA\_LEN. Changing the impulse response changes filter characteristics, e.g. low pass, high pass, bandpass. For example, a windowed sinc low pass filter (chp 16 Smith) may be created by changing the formula for calculating the impulse response values.

Both the input data and impulse response values are printed to the terminal window so they can be checked in Excel. The impulse response length is set to 30 and the DATA\_LEN is set to 100. This leaves memory space free to allow increases in the length of the impulse response.

Execution time for the main convolution loop is measured.

The convolution process consists of two for loops. The outer loop index k goes through all the valid output points. The inner for loop index i goes through all the impulse response points. The first IMP\_RESP\_LEN -1 points of the output are not a valid part of the convolution because there are not input data points yet to fully cover the impulse response. You can think of this as the impulse response not fully immersed in the input data. Because of this, the for loop starts at k = IMP\_RESP\_LEN -1 and runs to the end of the data at DATA\_LEN. Inside the inner for loop, there is a convolution sum where points in the input data are multiplied by corresponding points in the impulse response and summed together over all point in the impulse response.

startTime = micros();

for (int k = IMP\_RESP\_LEN-1; k < DATA\_LEN; k++){

for (int i = 0; i < IMP\_RESP\_LEN; i++){

yv[k] = yv[k]+h[i]\*xv[k-i];

}

}

execTime = micros()-startTime;

After the convolution, the execution time is calculated and the output values printed. Note that when calculating per-output-point-performance, you must be careful to normalize total execution time by the number of output points   
created (i.e. DATA\_LEN-IMP\_RESP\_LEN).

**Procedure**

1. Paste the file from the textbox into the Arduino IDE.
2. Run the program and observe results.
3. Change the datatype from byte to integer, long, and float in the variable declarations only. Make no other changes. **Complete Table 3-4 shown below.**

float xv[DATA\_LEN]={0}; // convolution input: x variable

float yv[DATA\_LEN]={0}; // convolution output: y variable

float h[IMP\_RESP\_LEN]={0}; // filter impulse response

Table 3-4 Convolution memory usage and execution time versus datatype

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Convolution Performance impulse response length = 30  data length = 100** |  |  |  |  |
|  | byte | int | long | float |
| FreeMemory |  |  |  |  |
| microseconds for each new data point |  |  |  |  |
| Max update rate (Hz) |  |  |  |  |
| final value of output |  |  |  |  |

1. **Q3-12 Your lab partner collected the data in the table 3-4 but doesn’t understand what it means. Write a paragraph to your lab partner describing what the data means and what observations you have made about how datatype affects execution speed, memory usage and accuracy? Are all the results accurate? Include your paragraph in your report.**
2. Change all the variable datatypes back to floats.
3. Set the IMP\_RESP\_LEN = 20, 30, 40, 50, 60 and **complete Table 3-5 shown below.**
4. **P3-2** **Plot the execution time versus impulse response length.**
5. **Q3-13 Write a paragraph to your lab partner that discusses how the length of the impulse response affects the execution speed of the algorithm. In your paragraph, give an example of the effect, but express the effect in terms of the algorithm’s bandwidth in Hz.**
6. Restore IMP\_RESP\_LEN = 30. Change the datatype to long for all three variables and modify the code as follows so that the impulse response h[ i ] is multiplied by 100,000. This normalizes the impulse response to be an integer value, thus allowing it to be stored as a long integer. **Note**: the Serial.print() precision modifier only makes sense for floats. Eliminate it for integer type outputs.

for (int i=0; i <IMP\_RESP\_LEN; i++)

{

h[i]=100000.0/IMP\_RESP\_LEN;

Serial.print(i); Serial.print('\t');

Serial.println(h[i],3);

}

1. Next modify the printing of the output yv[], so that the output value is de-normalized by dividing yv[] by 100,000. Now the value of yv[] is displayed as expected.

Serial.print("final value of yv = ");

Serial.println(yv[DATA\_LEN-1]/100000.0);

1. Run the code again and note the final value of output.
2. For an impulse response length of 30, try changing the datatype of xv[] to integer, while keeping yv[] and hv[] as long datatypes. Use a normalization of 100,000. Does the algorithm still produce the correct result? How many microseconds does the algorithm take to execute? Is this faster or slower than with all long datatypes? What about memory usage? Try again with h[] as an integer and xv[] as an integer, but yv[] as a long. What is the fastest bandwidth and with what combination of datatypes resulting in correct output?
3. **Q3-14** **Write one paragraph to your lab partner that describes how much faster your algorithm executes when using long data types versus floats. Then tell your lab partner what the fastest bandwidth was that you were able to achieve and with what datatype. Make sure to tell to your lab partner that the resulting value was correct, or if not, why you would use this combination of data types even though resulting calculated value is wrong.**
4. The moving average filter used in this lab was implemented by convolution. It is also possible to implement it by recursion, which is even faster. Sample code for a recursive moving average filter can be found in Smith chapter 15.

What update rate (Hz) can you achieve for a 30 point moving average filter using recursion and using long datatypes? If floating point datatypes are used, the recursion algorithm can suffer from accumulating roundoff error.

Table 3-5 Convolution update rate versus impulse response length for float datatype

|  |  |  |
| --- | --- | --- |
| Impulse response length | Update rate (Hz) of FIR algorithm implemented with floats | Execution Time for each data point (microseconds) |
| 20 |  |  |
| 30 |  |  |
| 40 |  |  |
| 50 |  |  |
| 60 |  |  |

## Write Up:

Refer to the rubric below. Include the requested plots, tables and answers to questions in a narrative format discussing the results and their implications.

Be sure to fully label the axes and title the graphs. Be sure to include descriptive captions for each figure, table and graph.

Use the IEEE Format to complete your report. Maintain the report title and Author line. Use the 2-column format for the remaining portions of the report. Maintain the correct font and margins.

Team Members:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lab Grade \_\_\_

**Grading Rubric – Digital Signal Processing LAB 05: DSP Number Systems**

|  |  |  |  |
| --- | --- | --- | --- |
| Lab Section | Result Description | Points Available | Points Obtained |
| Abstract | Abstract stating in your own words the objective of the lab and the approach to be taken | 10 |  |
| Section 1 | Correct and properly labeled tables 3-1a and 3-1b. Narrative answer to question Q3-0 | 10 |  |
| Section 2 | Correct and properly labeled tables 3-2. Narrative answers to question Q3-1 through Q3-3 | 10 |  |
| Section 3 | Narrative answers to question Q3-4 through Q3-8 | 15 |  |
| Section 4 | Narrative answers to question Q3-9 through Q3-11.  Correct plot P3-1 including descriptive titles, axes labels and caption | 15 |  |
| Section 5 | Complete and correct Table 3-3.  Narrative description of the results of this section. | 15 |  |
| Section 6 | Complete and correct Table 3-4 and 3-5. Narrative answers to questions Q3-12 to Q3-14.  Correct plot P3-2 including descriptive titles, axes labels and caption | 15 |  |
| Conclusion and Formatting | Concluding paragraph stating in your own words what you learned from this lab. Format for the overall report. | 10 |  |

**Instructor comments:**