Ultrasonic distance meter with the MSP430

Alejandro Esquivel

February 2019

Contents

1 Objective **3**

2 Timer **4**
- 2.1 Capture Mode . . . 4
- 2.2 Compare Mode . . . 4

3 UART **5**
- 3.1 Transmitting/Receiving . . . 5

4 HC-SR04 Distance Sensor **6**
- 4.1 Operation . . . 6
- 4.2 Measurement . . . 7

5 Procedure **8**
- 5.1 Ultrasonic Distance Meter Firmware - ultrasonic.c . . . 9
- 5.2 Real-time distance plot - python-serial-plot.py . . . 14
- 5.3 Distance Measurements & Accuracy . . . 16
  - 5.3.1 Code for distance accuracy plot . . . 17

6 References **18**

1 Objective

The MSP430 (Texas Instruments) will be used to create an ultrasonic distance meter with the common ultrasonic sensor HC-SR04. Distance measurements will be taken by measuring the time difference between when ultrasonic pulses were fired and when they return to the receiver. Once measurements are taken, they are to be sent via the USB-Serial interface to a host computer.

2 Timer

The MSP430 comes with two Timers, which can be used to take time measurements. The special register TAR gives the Timer's current count. Moreover, there are two modes that the Timers can be in: Capture Mode, and Compare Mode.

2.1 Capture Mode

In capture mode the Timer can use an external signal ("Capture Input") to trigger a time measurement, and will store the timer value in register TACCRx.

The capture input can be used to trigger a measurement if the signal is either a rising edge, falling edge, or in both rising and falling edges, this can be configured in the CMx bits in the TACCTLx (Capture/Compare Control Register).

The capture input can be a select few of the GPIO pins on the MSP430 (P1.1, P1.2, P2.0-P2.5, ect.) depending on the version of the MSP430 (here we use MSP430g2553). We must enable the special function select bits (P1SEL/P2SEL) for the the pin we decide to use as the capture input.

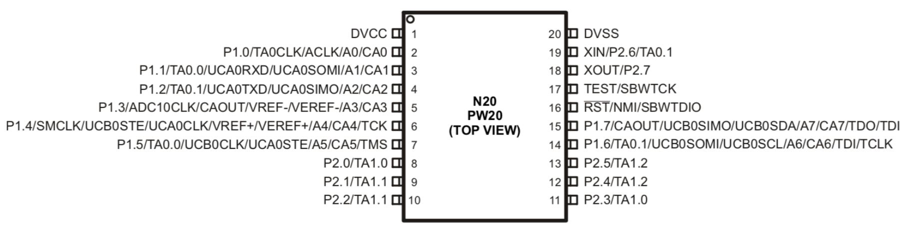

Figure 1: MSP430 Pinout

2.2 Compare Mode

In compare mode, the timer counts from 0 to a certain value, which can be specified (the value stored in TACCR0 [Up Mode], or 0xFFFF [Continous Mode], or TACCR0 then back to 0 [Up/Down Mode]. These modes can be configured in the MCx bits in the TACTL (Timer Control Register).

Every time the Timer counts up to it's given period (TACCRx value) an interrupt is fired (given interrupts are enabled with the TAIE bit within TACTL).

3 UART

The MSP430 comes with a USCI (Universal Serial Communications Interface) chip, which allows us to send data to some external source. The USCI can be configured to send data with asynchronous serial communication, which means that the transmitting and recieving clocks do not need to be synchronized, and the start/end of the data transmission is specified with start/stop signals, while a series of bits is sent sequentially in between. The hardware device to implement such asynchronous serial communication is UART, which the USCI implements if in UART mode.

There are a set of special registers that enable us to configure the USCI, particularly (UCAxCTL0 and UCAxCTL1) which are the "USCI Control Registers". UART mode is default, so it is optional to explicitly configure UCAxCTL0 to be in UART mode (UCMODEx = 00).

Data is transmitted physically bit by bit in serial communication, and information is represented by HIGH(1) or LOW(0) digital signals. These signals transition between these states (HIGH/LOW) over time to convey data. The "Baud Rate" in our case, represents how many bits per second are sent via serial communication. The baud rate can be configured with the special registers UCA0BR0, and UCA0BR0.

3.1 Transmitting/Receiving

There are various interrupts and interrupt flags associated with transmitting or recieving data over UART. Particularly, interrupts are fired when data is ready to be sent from the host computer (or other external device), and when data has been transferred from the MSP430. The UCAxRXBUF & UCAxTXBUF are special registers to read (Rx) and write (Tx) bytes.

When data is to be transferred, a byte can be copied to UCAxTXBUF, at which point the MSP430 begins transmitting the data over UART. The interrupt flag UCAxTXIFG indicates whether the byte has been successfully transferred, and is cleared during transmission. After the transmission is complete the UCAxTXIFG flag is truthy and an interrupt is fired.

The receiving process is similar, the special register UCAxRXBUF is read when data is available from a host computer(or external device). The interrupt flag UCA1RXIFG indicates whether there is data waiting to be consumed, and is automatically cleared when the UCAxRXBUF data is read. An interrupt fires when UCA1RXIFG is truthy or equivalently, when Rx byte transfer is ready. If data is not read and the external device transfers a new byte, the data in the Rx buffer will be overwritten and the UCOE (data overwritten) flag in the UCAxSTAT register will be truthy.

4 HC-SR04 Distance Sensor

The HC-SR04 is a common ultrasonic distance sensor which uses a piezo transducer to generate ultrasonic pulses and a receiver which detects ultrasonic pulses. Applying a voltage to a piezoelectric material causes is to deform which in turn causes a pressure wave or "sound wave" upon deformation. By deforming the piezoelectric material in the piezoelectric transducer at a certain frequency, an ultrasonic sound wave can be generated (any frequency over 20kHz). In theory the receiver can act in a similar manner but in reverse, receiving an ultrasonic pulse and generating minute voltages upon deformation of the piezoelectric material which would need to be amplified.

4.1 Operation

By applying at least a 10 miscrosecond HIGH signal on the Trigger pin, eight ultrasonic 40kHz pulses are sent from the piezo transducer, at which point the Echo pin emits a HIGH signal until the reflected pulses are detected (As can be seen in the timing diagram).

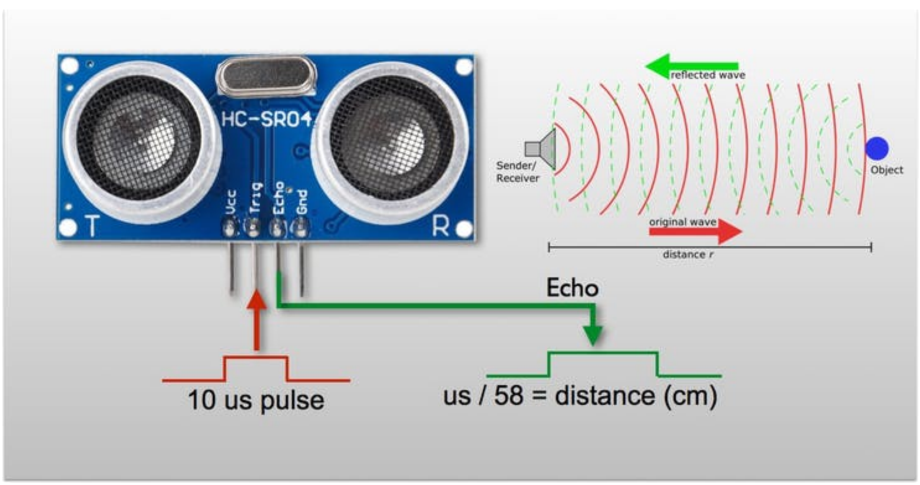

Figure 2: [2] HC-SR04 Pins and generated pulses

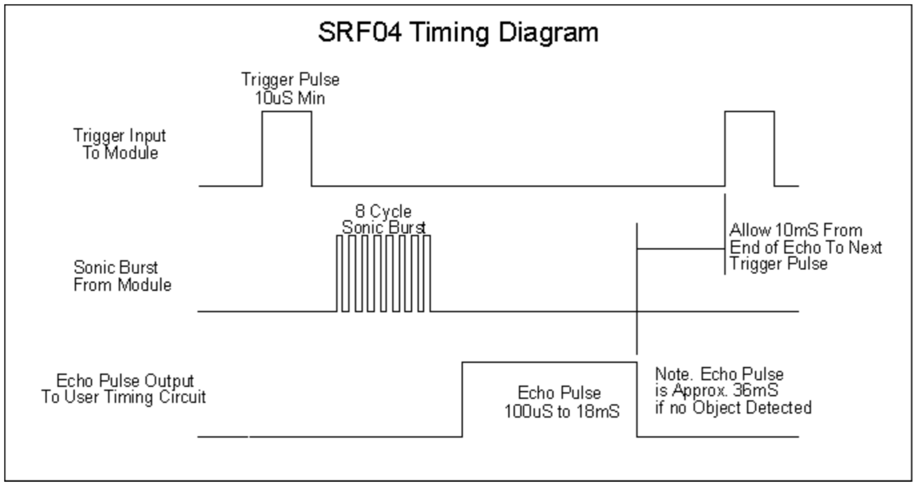

Figure 3: [3] HC-SR04 Timing Diagram

4.2 Measurement

The pulses travel a distance d in total, one trip towards the object, and another back to the receiver, therefore the object is $d/2$ meters away from the sensor. The distance can be computed in practice by measuring the time the Echo signal is HIGH. As sound waves travel at 343m/s (at 20 degrees Celsius), we can calculate the displacement (s) [meters] from the sensor to an object in it's vicinity as (Δt is in seconds)

$$s = \frac{d}{2} = \frac{343\Delta t}{2} = 171.5\Delta t$$

where Δt is the duration of the Echo signal. To calculate the displacement s in centimeters we can use the equivalent formula (Δt is in microseconds)

$$s = \frac{\Delta t}{58} \tag{1}$$

For an even higher precision measurement we can express s in micrometers as (Δt is in microseconds)

$$s = \frac{\Delta t}{0.00583090379} \tag{2}$$

which is what we will use in our implementation of the ultrasonic distance meter.

5 Procedure

In order to make a distance measurement, we must measure the time that the HC-SR04's Echo signal is high. There are many different approaches, here we will discuss an approach using the MSP430's Timer Capture mode.

The Timer's capture mode is to used in order to detect the rising and falling edges of the Echo signal and record the corresponding times of those events. In order to do this we will use MSP430's GPIO pin P1.1 to be the "Capture input" and to receive the Echo signal from the HC-SR04 as in input. The P1.1 pin is connected in series with a $1k\Omega$ resistor (to protect the MSP430 from the 5V powered HC-SR04) and to the HC-SR04's Echo pin. Furthermore, the P2.1 pin is connected directly to the HC-SR04's Trigger pin so that we can programmatically begin a measurement.

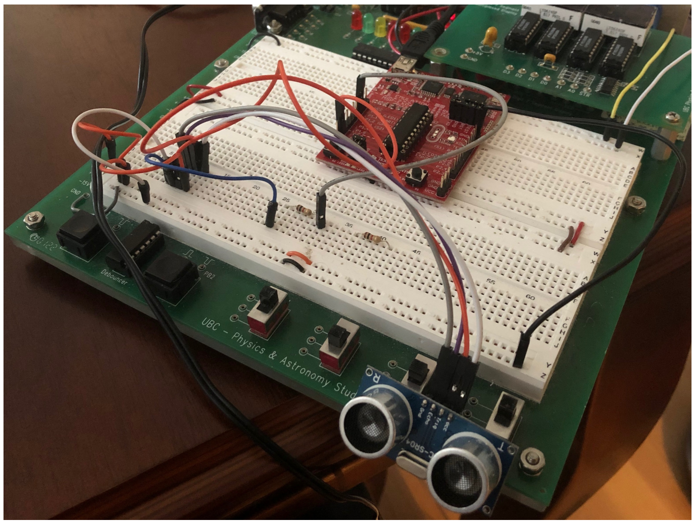

Figure 4: MSP430, HC-SR04 and Breadboard Connections

The MSP430's P1.1 pin corresponds to TA0.0 (Timer0_A0) from Figure 1. In the datasheet, we see this pin can be a capture input, particularly a CCI0A input. This implies a few things. The corresponding Interrupt vector address is Timer0_A0/Timer0_A3 (0xFFF2) which can be referenced in code by

```
Timer0_A0 // as defined in the header files
```

Additionally, we will use TACCTL0 capture/compare control register (TACCTLx where x = 0, as CCIxA is CCI0A due to the pin we selected). The TACCTL0 can be configured as follows:

```
TACCTL0 = CM_3 + SCS + CCIS_0 + CAP + CCIE;
```

Which configures the Timer to capture the rising and falling edge of the Echo signal (CM_3), set the capture input signal to CCI0A/P1.1 (CCIS_0), enables capture mode (CAP), and enables interrupts (CCIE).

Now, when an Echo signal is received via pin P1.1, two interrupts will be fired, once when there is a rising edge and another when there is a falling edge. Each time the interrupts are fired, the internal time counter TAR is copied to the TACCR0 register, which can be stored then subtracted to give us Δt (the time it took for wave to travel to and back from some object). During each interrupt the special register TAIV indicates the source of the interrupt was (capture input, timer overflow [timer counted to 0]), furthermore the CCI bit in the TACCTL0 gives us the capture input value (P1.1 value) which can help us distinguish between a rising/falling edge.

As the master clock of the MSP430 runs at $\approx 1MHz$, we can use the command

```
__delay_cycles(10);
```

to wait for $\approx 10\mu s$, and can be used to trigger a measurement by outputting HIGH to P2.1 (trigger pin) for $\approx 10\mu s$.

UART can only only transmit data byte by byte, so in order to transmit higher precision measurements, we must break data that is to be transmitted into byte chunks. One way of doing this is by converting a numerical data type (long in this case) to a string, as a string is just an array of characters (char) which are one byte each, we can just send a series of characters over UART.

```
while (!(IFG2 & UCA0TXIFG)); UCA0TXBUF = char_value;
```

The above code snippet waits until the interrupt flag UCA0TXIFG is 1 (indicating a new byte is ready to be transferred), and sends a (char) byte value through UART. A string can simply be sent by iterating the character array (the string) until the null-character ("\0") is encountered (indicating the end of the string).

5.1 Ultrasonic Distance Meter Firmware - ultrasonic.c

The full implementation of the Ultrasonic distance meter firmware for the MSP430. The code can also be viewed or cloned from `https://github.com/AlejandroEsquivel/msp430-ultrasonic-distance`.

```c
#include "msp430.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define TRIG_PIN BIT1 // Corresponds to P2.1
#define ECHO_PIN BIT1 // Corresponds to P1.1

#define TXD BIT2 // TXD on P1.2

volatile unsigned long start_time;
volatile unsigned long end_time;
volatile unsigned long delta_time;
volatile unsigned long distance;

void wait_ms(unsigned int ms)
{
  unsigned int i;
  for (i = 0; i <= ms; i++)
  {
    // Clock is ~1MHz so 1E3/1E6 = 1E-3 (1ms) seconds
    __delay_cycles(1000);
  }
}

/* Write byte to USB-Serial interface */
void write_uart_byte(char value)
{
  while (!(IFG2 & UCA0TXIFG))
    ;
  // wait for TX buffer to be ready for new data
  // UCA0TXIFG register will be truthy when available to recieve new data to computer.
  UCA0TXBUF = value;
}

void write_uart_string(char *str)
{
  unsigned int i = 0;
  while (str[i] != '\0')
  {
    //write string byte by byte
    write_uart_byte(str[i++]);
  }
```

```c
}

void write_uart_long(unsigned long l)
{
  //allocate memory for string representation of long
  char buf[sizeof(l) * 8 + 1];
  //convert long to string
  sprintf(buf, "%ld\n", l);
  //write string over UART
  write_uart_string(buf);
}

#if defined(__TI_COMPILER_VERSION__)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void ta1_isr(void)
#else
void __attribute__((interrupt(TIMER0_A0_VECTOR))) ta1_isr(void)
#endif
{
  switch (TAIV)
  {
  //Timer overflow
  case 10:
    break;
  //Otherwise Capture Interrupt
  default:
    // Read the CCI bit (ECHO signal) in CCTL0
    // If ECHO is HIGH then start counting (rising edge)
    if (CCTL0 & CCI)
    {
      start_time = CCR0;
    } // If ECHO is LOW then stop counting (falling edge)
    else
    {
      end_time = CCR0;
      delta_time = end_time - start_time;
      distance = (unsigned long)(delta_time / 0.00583090379);

      //only accept values within HC-SR04 acceptible measure ranges
      if (distance / 10000 >= 2.0 && distance / 10000 <= 400)
      {
        write_uart_long(distance);
      }
    }
```

```
    break;
  }
  TACTL &= ~CCIFG; // reset the interrupt flag
}

/* Setup TRIGGER and ECHO pins */
void init_ultrasonic_pins(void)
{
  // Set ECHO (P1.1) pin as INPUT
  P1DIR &= ~ECHO_PIN;
  // Set P1.1 as CCI0A (Capture Input signal).
  P1SEL |= ECHO_PIN;
  // Set TRIGGER (P2.1) pin as OUTPUT
  P2DIR |= TRIG_PIN;
  // Set TRIGGER (P2.1) pin to LOW
  P2OUT &= ~TRIG_PIN;
}

/* Setup UART */
void init_uart(void)
{

  // Set P1.2 as TXD
  P1DIR |= TXD;
  P1OUT |= TXD;
  P1SEL |= TXD;
  P1SEL2 |= TXD;

  // Use SMCLK - 1MHz clock
  UCA0CTL1 |= UCSSEL_2;
  // Set baud rate to 9600 with 1MHz clock (Data Sheet 15.3.13)
  UCA0BR0 = 104;
  // Set baud rate to 9600 with 1MHz clock
  UCA0BR1 = 0;
  // Modulation UCBRSx = 1
  UCA0MCTL = UCBRS0;
  // Initialize USCI state machine - enable
  UCA0CTL1 &= ~UCSWRST;
}

void init_timer(void)
{
  /* Use internal calibrated 1MHz clock: */
  BCSCTL1 = CALBC1_1MHZ; // Set range
  DCOCTL = CALDCO_1MHZ;
```

```c
  BCSCTL2 &= ~(DIVS_3); // SMCLK = DCO = 1MHz

  // Stop timer before modifiying configuration
  TACTL = MC_0;

  /*
  1. Capture Rising & Falling edge of ECHO signal
  2. Sync with clock
  3. Set Capture Input signal to CCI0A
  4. Enable capture mode
  5. Enable interrupt
  */
  CCTL0 |= CM_3 + SCS + CCIS_0 + CAP + CCIE;
  // Select SMCLK with no divisions, continous mode.
  TACTL |= TASSEL_2 + MC_2 + ID_0;
}

void reset_timer(void)
{
  //Clear timer
  TACTL |= TACLR;
}

void main(void)
{
  // Stop Watch Dog Timer
  WDTCTL = WDTPW + WDTHOLD;

  init_ultrasonic_pins();
  init_uart();
  init_timer();

  // Global Interrupt Enable
  __enable_interrupt();

  while (1)
  {
    // send ultrasonic pulse
    reset_timer();
    // Enable TRIGGER
    P2OUT |= TRIG_PIN;
    // Send pulse for 10us
    __delay_cycles(10);
    // Disable TRIGGER
    P2OUT &= ~TRIG_PIN;
```

```
    // wait 500ms until next measurement
    wait_ms(500);
  }
}
```

5.2 Real-time distance plot - python-serial-plot.py

The supplied python code[1] was modified to take in a micrometer measurement, and convert it to centimeters. Furthermore, it was also modified to write measurements to a file (distance_vs_time.txt) when the key 'm' was pressed during runtime.

```python
# University of British Columbia
# Department of Physics & Astronomy

#!/usr/bin/python2.7
import serial # for serial port
import numpy as np # for arrays, numerical processing
from time import sleep,time
import gtk #the gui toolkit we'll use:
# graph plotting library:
from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas

#define the serial port. Pick one:
port = "/dev/ttyACM0"  #for Linux
#port = "COM5" #For Windows?
#port = "/dev/tty.uart-XXXX" #For Mac?

outFile = open("distance_vs_time.txt","a")

#function that gets called when a key is pressed:
# press 'm' to make a measurement
def press(event):
    print('press', event.key)
    if event.key == 'q':
        print ('got q!')
        quit_app(None)
    if event.key == 'm':
        outFile.write(str(current_time)+" "+str(current_distance)+"\n") #write to file
```

```python
    return True

def quit_app(event):
    outFile.close()
    ser.close()
    quit()

#start our program proper:
#open the serial port
try:
    ser = serial.Serial(port,2400,timeout = 0.050)
    ser.baudrate=9600
# with timeout=0, read returns immediately, even if no data
except:
    print ("Opening serial port",port,"failed")
    print ("Edit program to point to the correct port.")
    print ("Hit enter to exit")
    raw_input()
    quit()

#create a window to put the plot in
win = gtk.Window()
#connect the destroy signal (clicking the x in the corner)
win.connect("destroy", quit_app)
win.set_default_size(800,800)

yvals = np.zeros(50) #array to hold last 50 measurements
times=np.arange(0,50,1.0) # 50 from 0 to 49.

#create a plot:
fig = Figure()
ax = fig.add_subplot(111,xlabel='Time Step',ylabel='Distance [cm]')
ax.set_ylim(2,400) # set limits of y axis.

canvas = FigureCanvas(fig) #put the plot onto a canvas
win.add(canvas) #put the canvas in the window

# define a callback for when a key is pressed
fig.canvas.mpl_connect('key_press_event',press)

#show the window
win.show_all()
win.set_title("ready to receive data");

```

```python
line, = ax.plot(times,yvals)
start_time = time()
ser.flushInput()

while(1): #loop forever
    data = ser.readline() # look for a character from serial port, will wait up to timeout above.
    if (len(data) > 0): #was there a byte to read? should always be true.
        current_distance = float(data)/10000;
        print(current_distance);
        yvals = np.roll(yvals,-1) # shift the values in the array
        yvals[49] = current_distance # take the value of the byte
        current_time = time() - start_time;
        line.set_ydata(yvals) # draw the line
        fig.canvas.draw() # update the canvas
        win.set_title("Distance: "+str(current_distance)+"cm")
    while gtk.events_pending():        #makes sure the GUI updates
        gtk.main_iteration()
#    sleep(.05) # don't eat the cpu. This delay limits the data rate to ~ 200 samples/s
```

5.3 Distance Measurements & Accuracy

Once compiling the C code into a binary and writing it into the MSP430's program memory, and upon having the connections as stated in Section 5, we have our very own ultrasonic distance meter. The last step is to connect the MSP430 to a host PC via USB in order to receive the serial data from the MSP430. The python code "python-serial-plot.py" can be executed in order to see a real-time plot of the distances measured by the ultrasonic distance meter.

A meter stick was used to test the accuracy of the ultrasonic distance meter (as in plot below).

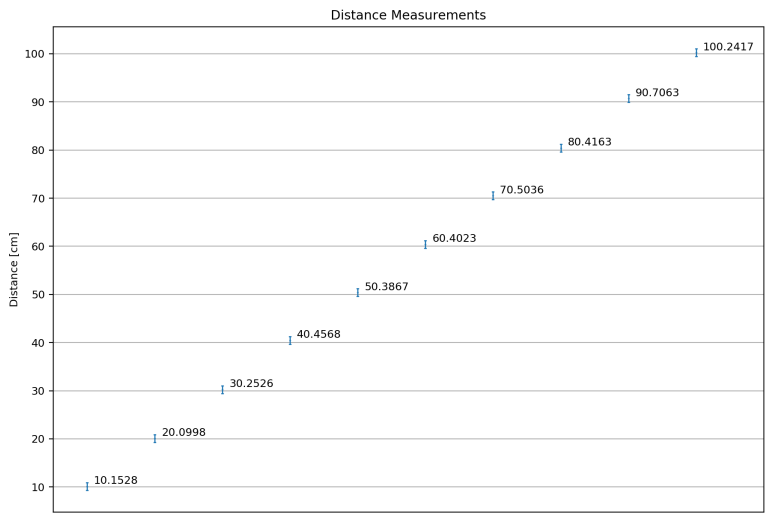

Figure 5: Ultrasonic Distance Meter measurements

To collect this data, the ”python-serial-plot.py” program was used to gather data (by pressing ’m’ key to record values). Furthermore, a meter stick was used to place the ultrasonic distance meter at a varying distance from a wall. As the meter stick used had uncertainty 0.5cm and the HC-SR04 has a resolution of 0.3cm[4], the total uncertainty associated with each measurement was 0.8cm (as meter stick determined distance meter position). By inspecting the plot above, our collected data seems to agree with the true distances within the bounds of the respective uncertainty.

5.3.1 Code for distance accuracy plot

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 9})

distance_meter_error = 0.3;
meter_stick_error = 0.5;

x = np.arange(1, 11, 1);
# Measured distances from ultrasonic distance meter
```

```
d = [
    10.1528,
    20.0998,
    30.2526,
    40.4568,
    50.3867,
    60.4023,
    70.5036,
    80.4163,
    90.7063,
    100.2417
];

# Meter stick measurements
m = [];
for i in range(1,11):
    m.append(i*10);

# First illustrate basic pyplot interface, using defaults where possible.
plt.figure()

plt.errorbar(x, d, yerr=(distance_meter_error + meter_stick_error), linewidth=1, marker='.', markersize=1, barsabove=True,linestyle = 'None',capsize=1);

for i,j in zip(x,d):
    plt.annotate(str(j),xy=(i+0.1,j+0.5))

plt.xlim(0.5,11);
plt.xticks([]);
plt.yticks(m)
plt.ylabel('Distance [cm]');
plt.title("Distance Measurements")
plt.grid();
plt.show();
```

6 References

[1] Lab manual provided by the University of British Columbia in the URL: `https://www.phas.ubc.ca/~kotlicki/Physics_319/Lab_Manual-2019\%20AK.pdf`

[2] Figure 2 HC-SR04 Pins and generated pulses figure https://www.hackster.io/powerberry/like-a-bat-with-hc-sr04-829486

[3] HC-SR04 Timing diagram http://www.robot-electronics.co.uk/htm/srf04tech.htm

[4] HC-SR04 User Guide https://web.archive.org/web/20171215050436/http://www.micropik.com/PDF/HCSR04.pdf