ECE-CT580 Spring 19-20
Homework/Reading Week 3-4

**Please submit the filled in cover sheet as the first page of you HW submission.**

**Last Name:_Nguyen___________ First Name:__Tai____________**

Students must indicate the status of each problem by:

- **C:** completed,
- **P:** Partially completed,
- **N:** not attempted

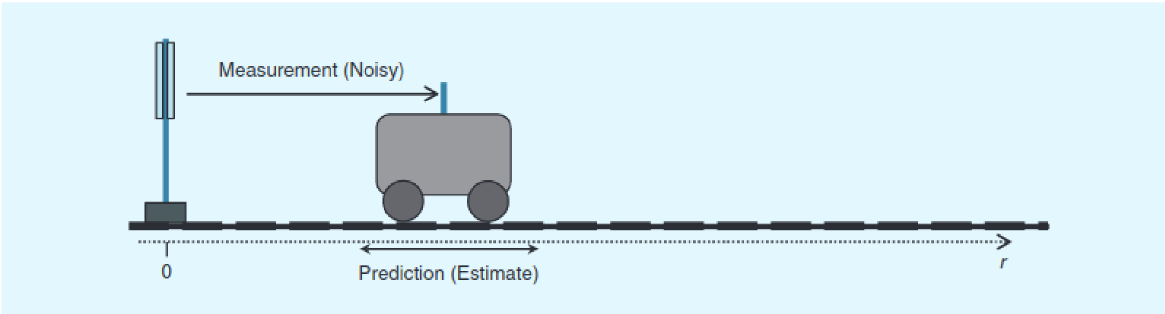
## Instructor Problem

| Problem | Status | Grade/Comments |
|---|---|---|
| HW 1<br>Briefing slides | **C** | |
| HW 2<br>Details of eq 10 thru 13 | **C** | |
| | | |
| | | |
| Coding 1<br>Interrupt via Timer | **C** | |
| Coding 2<br>Kalman filter<br>With table | **C** | |

Final Score:_(10 points)

# Figure Briefing for the article Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation by Ramsey Faragher

*R. Faragher, "Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation [Lecture Notes]," in IEEE Signal Processing Magazine, vol. 29, no. 5, pp. 128-132, Sept. 2012, doi: 10.1109/MSP.2012.2203621.*




[FIG1] This figure shows the one-dimensional system under consideration.

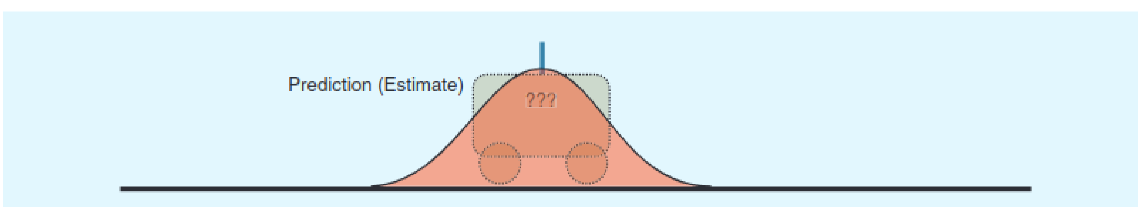
This figure shows the system where the measurements are noisy. The true state lie inside the distribution in which the measurement are the means and the measuring device's uncertainty is the variance.

$$Z_t = H_t \times X_t + v_t$$

Zt is the measurement, Ht is the transformation matrix, Xt is the ground truth, vt is the measurement noise

[FIG2] The initial knowledge of the system at time $t = 0$. The red Gaussian distribution represents the pdf providing the initial confidence in the estimate of the position of the train. The arrow pointing to the right represents the known initial velocity of the train.

The initial knowledge about the system is not exact. The true state of the system lies inside in the distribution.

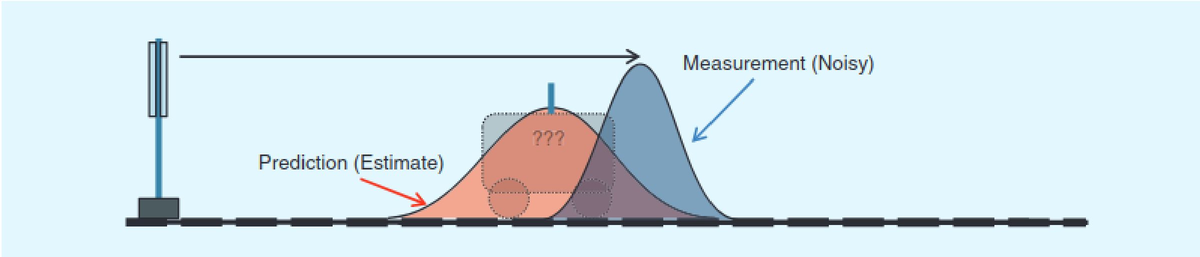



[FIG3] Here, the prediction of the location of the train at time $t = 1$ and the level of uncertainty in that prediction is shown. The confidence in the knowledge of the position of the train has decreased, as we are not certain if the train has undergone any accelerations or decelerations in the intervening period from $t = 0$ to $t = 1$.

Since the initial knowledge is only a guess, the prediction about the system's state will be less inaccurate the more time goes on because the train's acceleration/decelerations are unknown, which are represented by a process noise term in the following equation:

$$X_t = F_t \times X_{t-1} + B_t \times u_t + w_t$$

Xt is ground truth, Ft is the transition matrix, Bt is the control input matrix (applying on ut, representing speed up/down), wt is the process noise.

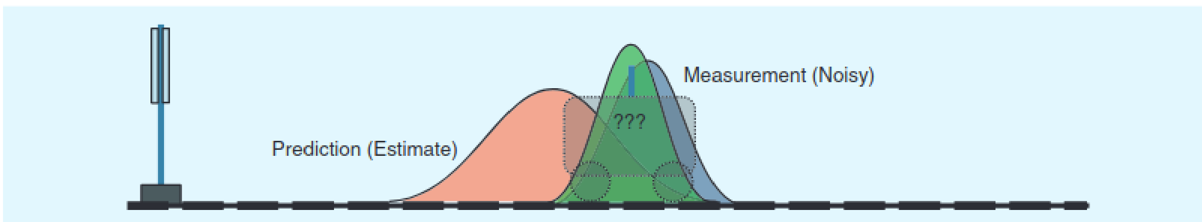



[FIG4] Shows the measurement of the location of the train at time *t* = 1 and the level of uncertainty in that noisy measurement, represented by the blue Gaussian pdf. The combined knowledge of this system is provided by multiplying these two pdfs together.

In order to reduce uncertainty in the predicted state at time t, a measurement of the system's state is made at time t. Together, they can be combined to improve the overall prediction.

$$\hat{X}_{t|t} = \hat{X}_{t|t-1} + K_t \times (Z_t - H_t \times \hat{X}_{t|t-1})$$

Kt is the Kalman gain.




[FIG5] Shows the new pdf (green) generated by multiplying the pdfs associated with the prediction and measurement of the train's location at time *t* = 1. This new pdf provides the best estimate of the location of the train, by fusing the data from the prediction and the measurement.

The Kalman gain is computed by fusing the 2 pdfs together. Multiplying the 2 gaussian distributions gives:

$$y_{fused}(r; \mu_1, \sigma_q, \mu_2, \sigma_2) = \frac{1}{2\pi\sqrt{\sigma_1^2 \sigma_2^2}} e^{-(\frac{(r-\mu_1)^2}{2\sigma_1^2} + \frac{(r-\mu_2)^2}{2\sigma_2^2})}$$

In order to normalize the measurements in both distribution, the means and the variances are divided by a constant c. Hence, the fused mean and the fused variance can be derived:

$$\mu_{fused} = \mu_1 + (\frac{\frac{\sigma_1^2}{c}}{\frac{\sigma_1}{c}^2 + \sigma_2^2}) \times (\mu_2 - \frac{\mu_1}{c})$$

$$\sigma_{fused}^2 = \sigma_1^2 - (\frac{\frac{\sigma_1^2}{c}}{\frac{\sigma_1}{c}^2 + \sigma_2^2}) \times \frac{\sigma_1^2}{c}$$

Hence, K can be derived as follow:

$$H = \frac{1}{c}$$

$$K = \frac{H\sigma_1^2}{H^2 \sigma_1^2 \sigma_2^2}$$

From the equation above, K represents the fused distribution, and can be calculated from the covariance matrixes of the 2 distributions.

$$K = P_{t|t-1} H_t^T (H_t P_{t|t-1} H_t^T + R_t)^{-1}$$

P is the covariance matrix of the process. R is the covariance of the measurement noise. P is updated using its previous value and K.

$$P_{t|t} = P_{t|t-1} - K_t H_t P_{t|t-1}$$

# Eq 10-13 from the same paper:

$$y_{\text{fused}}(r; \mu_1, \sigma_1, \mu_2, \sigma_2) = \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(r-\mu_1)^2}{2\sigma_1^2}} \times \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(r-\mu_2)^2}{2\sigma_2^2}} = \frac{1}{2\pi\sqrt{\sigma_1^2\sigma_2^2}} e^{-\left(\frac{(r-\mu_1)^2}{2\sigma_1^2} + \frac{(r-\mu_2)^2}{2\sigma_2^2}\right)}. \qquad (10)$$

$$y_{\text{fused}}(r; \mu_{\text{fused}}, \sigma_{\text{fused}}) = \frac{1}{\sqrt{2\pi\sigma_{\text{fused}}^2}} e^{-\frac{(r-\mu_{\text{fused}})^2}{2\sigma_{\text{fused}}^2}}, \qquad (11)$$

where

$$\mu_{\text{fused}} = \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2} = \mu_1 + \frac{\sigma_1^2(\mu_2 - \mu_1)}{\sigma_1^2 + \sigma_2^2} \qquad (12)$$

and

$$\sigma_{\text{fused}}^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2} = \sigma_1^2 - \frac{\sigma_1^4}{\sigma_1^2 + \sigma_2^2}. \qquad (13)$$

- Eq 10 describes the multiplication of 2 pdfs.
- Eq 11 shows that the multiplication of 2 pdfs results in a new pdf with a new mean and new stddev.
- Eq 12 shows how to get the new mean using the 2 means from the 2 pdfs.
- Eq 13 shows how to get the new stddev from those in the previous 2 pdfs.

# Problem 1: Function generator using Interrupts

A sine wave with variable frequency [0.1, 30] Hz is generated using interrupts on the arduino. The frequency is controlled by a potentiometer. The sampling rate of the voltage at the potentiometer is ~9.6 kHz. The output rate of the sine wave is 8 kHz. The sampling rate of the serial output is > 2x the output rate of the sine wave (19.2 kHz). The Arduino code and result are shown below. *Note: the code also include a SCPI interface so that data collection can be done using other softwares such as MATLAB.*

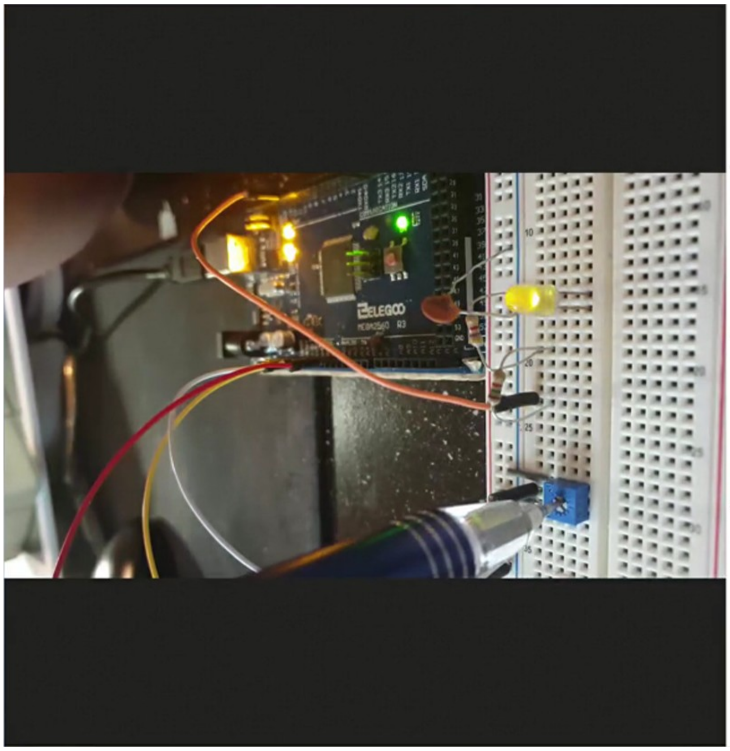
```
#include <scpiparser.h>
#include <Arduino.h>

#define TAB_LEN 1024

volatile uint16_t a0;   // ADC ch 0 result
const float F0_MIN = 0.1;
const float F0_MAX = 30;
uint16_t freqtable[TAB_LEN];
uint16_t wavetable[TAB_LEN];

volatile uint16_t phase = 0;
volatile uint16_t freqADC = 0;
volatile uint16_t freqSCPI = 0;
volatile boolean isSCPI = false;
volatile uint16_t sine_sample = 0;

struct scpi_parser_context ctx;

scpi_error_t identify(struct scpi_parser_context* context, struct scpi_token* command);
scpi_error_t identify(struct scpi_parser_context* context, struct scpi_token* command);
scpi_error_t set_freq(struct scpi_parser_context* context, struct scpi_token* command);


void wavetable_init();
void freqtable_init(float f_min, float f_max);
void adc_init_pins();
void adc_init_free_running();
void timer2_init_pwm();
void timer0_init_ctc();

void setup() {
    Serial.begin(19200);
    adc_init_pins();
    adc_init_free_running();
    timer2_init_pwm();
    timer0_init_ctc();
    freqtable_init(F0_MIN, F0_MAX);
    wavetable_init();

    scpi_init(&ctx);

    scpi_register_command(ctx.command_tree, SCPI_CL_SAMELEVEL, "*IDN?", 5, "*IDN?", 5, identify);

    struct scpi_command* set_freq_cmd;
    set_freq_cmd = scpi_register_command(ctx.command_tree, SCPI_CL_CHILD, "SOURCE", 6, "SOUR", 4, NULL);
    scpi_register_command(set_freq_cmd, SCPI_CL_CHILD, "FREQUENCY", 9, "FREQ", 4, set_freq);
}

void loop() {
    char line_buffer[256];
    unsigned char read_length;

//    Serial.println(sine_sample);
//    Serial.print(",");
//    Serial.println(a0);
    while(1)
```

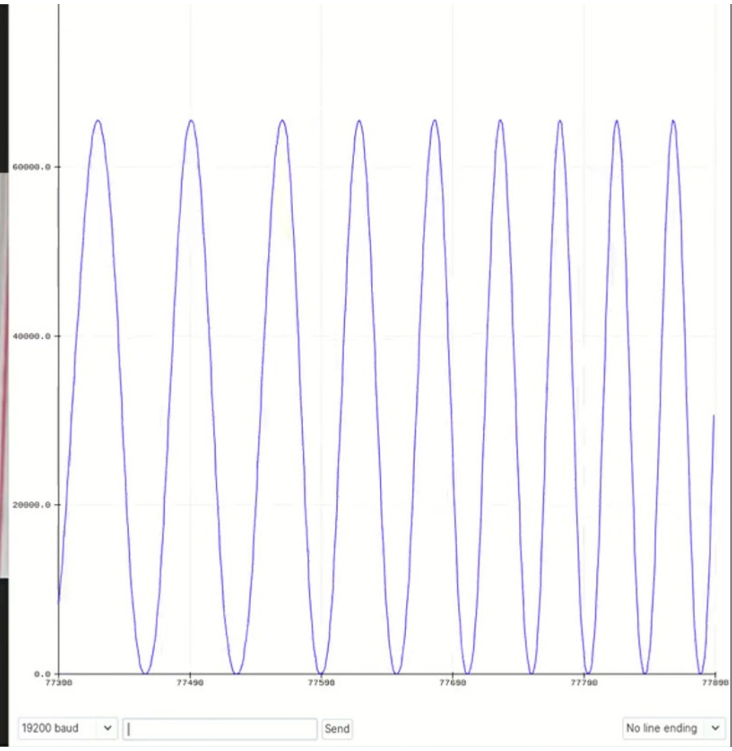
```
    while(1)
    {
        /* Read in a line and execute it. */
        read_length = Serial.readBytesUntil('\n', line_buffer, 256);
        if(read_length > 0)
        {
            scpi_execute_command(&ctx, line_buffer, read_length);
        }
    }

}

/*
 * Respond to *IDN?
 */
scpi_error_t identify(struct scpi_parser_context* context, struct scpi_token* command)
{
  scpi_free_tokens(command);
  Serial.println("ECE-303,SimpleDMM,1,10");
  return SCPI_SUCCESS;
}

scpi_error_t set_freq(struct scpi_parser_context* context, struct scpi_token* command)
{
    struct scpi_token* args;
    struct scpi_numeric output_numeric;
    unsigned char output_value;

    args = command;

    while (args != NULL && args->type == 0)
    {
        args = args->next;
    }

    output_numeric = scpi_parse_numeric(args->value, args->length, F0_MIN, F0_MIN, F0_MAX);
    if (output_numeric.unit[0] != 'H')
    {
        Serial.println("Command error;Invalid unit");
        scpi_error error;
        error.id = -200;
        error.description = "Command error;Invalid unit";
        error.length = 26;
        Serial.print(output_numeric.length);

        scpi_queue_error(&ctx, error);
        scpi_free_tokens(command);
        return SCPI_SUCCESS;
    }
    else
    {
        if (output_numeric.value > F0_MAX || output_numeric.value < F0_MIN)
        {
            Serial.println("Command error;Out of range");
            scpi_error error;
            error.id = -201;
            error.description = "Command error;Out of range";
```

```
            error.description = "Command error;Out of range";
            error.length = 34;
            Serial.print(output_numeric.length);

            scpi_queue_error(&ctx, error);
            scpi_free_tokens(command);
            return SCPI_SUCCESS;
        }
        else
        {
            freqSCPI = (uint16_t)output_numeric.value;
            Serial.print("Setting freq to ");
            Serial.print(freqSCPI);
            Serial.println("Hz");

            freqSCPI = freqSCPI*(8000/1024);
            isSCPI = true;
        }
    }

    scpi_free_tokens(command);

    return SCPI_SUCCESS;
}

void wavetable_init() {
    for (int i = 0; i < TAB_LEN; i++) {
        wavetable[i] = (1 + sinf(2*M_PI*i / (TAB_LEN-1))) * 32767.5;
    }
}


// Exponential mapping between freq and pot ADC reading
void freqtable_init(float f_min, float f_max) {
   float coeff = powf(f_max/f_min, 1.0/(TAB_LEN-1));
   float val = f_min;
   for (int i = 0; i < TAB_LEN; i++) {
       freqtable[i] = roundf(val * 65535.5);
       val *= coeff;
   }
}


// Linear mapping between freq and pot ADC reading
void freqtable_init(float f_min, float f_max) {
    for (int i = 0; i < TAB_LEN; i++) {
        freqtable[i] = roundf(i*(f_max-f_min)/1024);
    }
}

/*
 * More or less equivalent to pinMode(A0, INPUT). Also disables digital input
 * buffer on A0 to save power
 */
void adc_init_pins() {
    DDRF &= ~(1 << PF0);      // Configure first pin of PORTF as input
    DIDR0 |= (1 << ADC0D);    // Disable digital input buffer
```

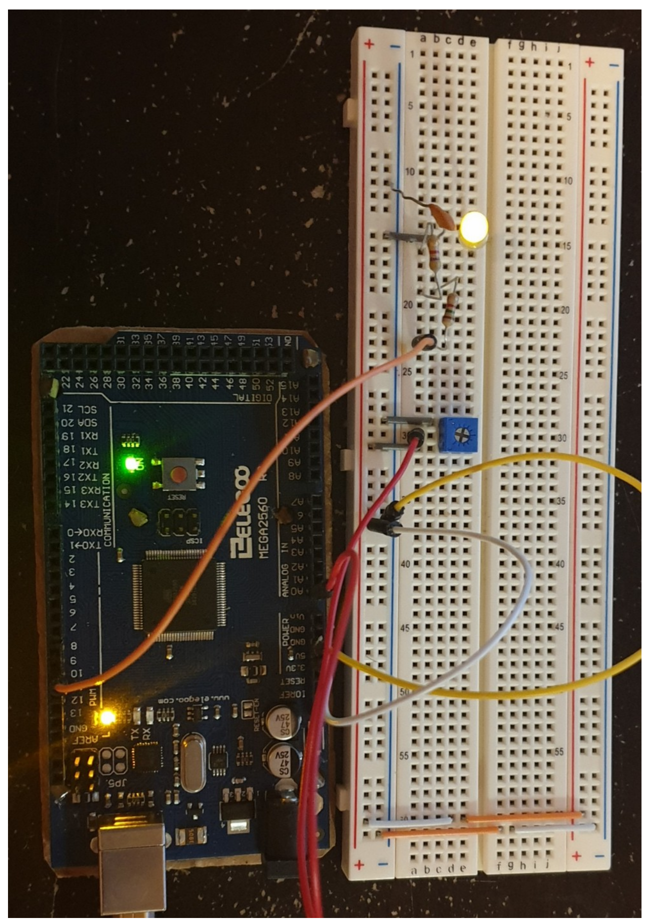
```
    DIDR0 |= (1 << ADC0D);    // Disable digital input buffer
}

/*
 * Initialize the ADC for free running mode with prescaler = 128, yielding
 * ADC clock freq 16e6/128 = 125kHz (need clock between 50kHz and 200kHz for
 * 10-bit resolution), and sample rate fs = 16e6/128/13=9.615kHz
 */
void adc_init_free_running() {
    ADMUX |= (1 << REFS0);    // Use AVCC as the reference
    ADMUX &= ~(1 << ADLAR);   // Right-adjust conversion results
    ADCSRA |= (1 << ADEN);    // Enable ADC
    ADCSRA |= (1 << ADIE);    // Call ISR(ADC_vect) when conversion is complete
    ADCSRA |= (1 << ADATE);                                // Enabble auto-trigger
    ADCSRB &= ~((1 << ADTS2) | (1 << ADTS1) | (1 << ADTS0));  // Free running mode
    ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);     // Set prescaler
    ADCSRA |= (1 << ADSC);    // Start first conversion
    sei();                    // Enable interrupts
}

/*
 * ADC interrupt service routine. Enabled by ADIE bit in ADCSRA register.
 *
 * Note: if we needed to read multiple ADC channels, we would read one per ISR
 * by reading the MUX bits (MUX4:0 in ADMUX and the MUX5 bit in ADCSRB) to get the
 * channel of the most recent conversion, store the result accordingly, and
 * set the MUX bits to read from the next channel we want.
 */
ISR(ADC_vect) {
    uint8_t lsb = ADCL;
    uint8_t msb = ADCH;
    a0 = (msb << 8) | lsb;
    freqADC = freqtable[a0]*(8000/1024);
}


/*
 *  Initialize Timer2 for fast PWM mode, at a frequency of 16e6/1/256 = 62.5kHz
 */
void timer2_init_pwm() {
    DDRB |= (1 << PB4); // Enable output on channel A (PB4, Mega pin 10)
    DDRH |= (1 << PH6); // Enable output on channel B (PH6, Mega pin 9)
    TCCR2A = 0;         // Clear control register A
    TCCR2B = 0;         // Clear control register B
    TCCR2A |= (1 << WGM21) | (1 << WGM20);  // Fast PWM (mode 3)
    TCCR2A |= (1 << COM2A1);  // Non-inverting mode (channel A)
    TCCR2A |= (1 << COM2B1);  // Non-inverting mode (channel B)
    TCCR2B |= (1 << CS20);    // Prescaler = 1
}

/*
 * Initialize Timer0 to control audio sample timing at fs = 8kHz
 */
void timer0_init_ctc() {
    TCCR0A = 0;               // Clear control register A
    TCCR0B = 0;               // Clear control register B
    TCCR0A |= (1 << WGM01);   // CTC (mode 2)
    TIMSK0 |= (1 << OCIE2A);  // Interrupt on OCR0A
    TCCR0B |= (1 << CS01);    // Prescaler = 8
    cli();                    // Disable interrupts
    TCNT0 = 0;                // Initialize counter
    OCR0A = 125;              // Set counter match value
    sei();                    // Enble interrupts
}

/*
 * Timer0's compare match (channel A) interrupt
 */
ISR(TIMER0_COMPA_vect) {
    phase += isSCPI ? freqSCPI : freqADC;
    sine_sample = wavetable[phase >> 6];
    OCR2A = OCR2B = (sine_sample/256);
}
```

# Demo of the system






Constructed Circuit



# Problem 2: Kalman Filter

The MATLAB code for the Kalman filter on page 21 & 22 is shown below

```
%% Tai Duc Nguyen - ECEC T580 - HW3

clear all; close all;

%% Kalman Filter Target Tracker

rng(0);

%% Create ground truth and measurement

max_time = 2000;
dt = 1;

a_gt = zeros(1,max_time);
a_gt(600:800) = 32.2;
a_gt(1000:1200) = -32.2;

v_gt = cumsum(a_gt)*dt;
x_gt = cumsum(v_gt)*dt;

sigma_x = 100; mu_x = 0;
noise_measure = normrnd(mu_x,sigma_x, [1,max_time]);
x_measure = x_gt + noise_measure;

%% Create Kalman Filter tracking position and velocity

order = 2;
A = [1 dt; 0 1];
B = [(dt^2/2); dt];
H = [1; 0];

PC_init = 1e6*eye(order);
xp_init = [0;0];
R = sigma_x^2;
QA = B*B'* (32.2^2);

P_corrected = PC_init;
P_pred = A*P_corrected*A' + QA;
x_pred = xp_init;

for i = 1:max_time
    K = P_pred*H*inv(H'*P_pred*H + R);
    x_corrected = x_pred + K*(x_measure(i) - H'*x_pred);
    P_corrected = (eye(order) - K*H')*P_pred;
    x_pred = A*x_corrected;
    P_pred = A*P_corrected*A' + QA;
end

disp(P_corrected);
disp(P_pred);
disp(K);
```

Table showing calculated K, P_predicted and P_corrected values after convergence of the Kalman filter algorithm for different measurement noise's sigmas (50, 100, 200).

| Sigma | K | P_predicted | P_corrected |
|---|---|---|---|
| 50 | 0.6738<br>0.3678 | 5163.7 2818.9<br>2818.9 2417.7 | 1684.5 919.6<br>919.6 1380.9 |
| 100 | 0.5494<br>0.2161 | 12193 4797<br>4797 3154 | 5494.1 2161.4<br>2161.4 2117.1 |
| 200 | 0.4320<br>0.1213 | 30418 8545<br>8545 4209 | 17278 4854<br>4854 3173 |