Remotely Connected Electric Field Generator for Particle Separation in a Fluid

Team May1612

Dee, Timothy
timdee@iastate.edu

Long, Justin
jlong@iastate.edu

McDonnel, Brandon
bmcdonnel@iastate.edu

April 16, 2016

Contents

1 Introduction 3

2 Project Definition 3
- 2.1 Deliverables . . . 3
- 2.2 Constraints . . . 3
- 2.3 System Analysis . . . 3

3 Functional Decomposition 3
- 3.1 Web Interface . . . 3
- 3.2 Web Server . . . 5
- 3.3 Raspberry Pi . . . 5
- 3.4 Minigen . . . 5
- 3.5 Amplifier Circuit . . . 5
  - 3.5.1 Summing Amplifier . . . 5
  - 3.5.2 Relay . . . 6
  - 3.5.3 Programmable Gain Amplifier(PGA) . . . 6

4 Software Components 6
- 4.1 Modifying Frequency . . . 6
- 4.2 Controlling Voltage . . . 6
  - 4.2.1 Relay . . . 6
  - 4.2.2 Programmable Gain Amplifier(PGA) . . . 6

5 Cost Considerations 6

6 Redesigned Components 6
- 6.1 Frequency Control . . . 7
- 6.2 Voltage Control . . . 7
  - 6.2.1 Digital Potentiometer . . . 7
  - 6.2.2 Transistor switch . . . 7

7 Known Issues 7
- 7.1 B23 . . . 7

8 Concluding Remarks 7

A Operation Manual 8
- A.1 Setup . . . 8
- A.2 Demo . . . 8

B Initial Designs 9
- B.1 Frequency Control . . . 9
- B.2 Voltage Control . . . 9
- B.3 Web Interface . . . 9

C Other Considerations 10

D Code 11

Abstract

This document details the design and implementation of a remotely connected electric field generator. The goal of this design is to provide an easy interface for manipulating the output voltage and frequency of a circuit remotely in order to generate an electric field. This electric field, when applied to a fluid over a long period of time will cause particles in the fluid to separate. The hardware and software components used to accomplish the aforementioned goal are described in detail.

1 Introduction

New research has shown that certain particles may be separated from fluids through dielectrophoresis. This process involves applying an electric field to a fluid. The field may be manipulated in order to attract or repel certain particles. The particles the electric field will attract or repel depends on characteristics of the electric filed which may be controlled by varying the voltage and frequency of the electronics driving the field.

This technology has many useful applications in health care. The proposed end use of this equipment is for separating particles bodily fluids, such as spinal fluid. Such medial applications could be useful in testing or filtering these fluids.

Being capable of separating though the application of an electric field would represent an advancement over existing solutions. It introduces a new method of testing which could be more precise when compared to existing technologies.

2 Project Definition

In our implementation, an electric field is applied to two metal plates. Through varying the voltage and frequency applied to these plates, the properties of the electric field may be changed.

Our job is to construct a system containing an electronic circuit capable of providing the necessary voltages and frequencies required to drive a pair of metal plates. This system must enable the circuit to be controllable through the use of a web interface. In addition, a small form factor must be maintained.

The system must be able to generate up to a 60 V peak-peak sine wave with user-controlled variable frequency from 10 kHz to 1 MHz.

2.1 Deliverables

There are four items which must be constructed for this project:

For the analog circuit components, functionality of the circuit will be tested using an oscilloscope to verify the requirements have been satisfied. This method can also be used to ensure the output signal contains minimal amounts of noise and distortion.

The construction of this device is the first phase of the project. After the completion of this component, the device will be used to experiment with particle separation in various fluid types. These experiments constitute the remainder of the project. For these experiments our advisor at Minetronix, John Pritchard, will be the main source of guidance and testable material.

2.2 Constraints

Constraints on this project fall within the size, voltage, and portability domains.

The size requirements of this project are directly related to the portability of the final design. The design requirements specify this system must be easily and quickly moved around from one workstation to another. The maximal allowed size is approximately the size of a backpack with smaller sizes being more desirable but not explicitly required. With the electronics currently being used, these requirements will easily be met.

Another constraint arises from the power supply requirements. The power supply must deliver at least 60V DC in order to feed the amplifier circuit. Due to this, the final design requires a power brick similar to one which would be used to charge a laptop. Importantly, this would require the device to be plugged into a wall outlet. This is not seen as an issue. Every location this device will operate will most likely have other equipment with similar power requirements.

In order to use this system, there are other items which are required apart from the device itself. The first requirement is a network connection between the device and a computer. This connection is necessary to be able to interact with the web server hosted on the Raspberry Pi. Without a computer to interact with this system there is no practical means of utilizing the device's functionality. The next requirement, as mentioned above, is a network connection to the Raspberry Pi. The third system requirement is a standard wall outlet to accommodate the power needs of the system.

2.3 System Analysis

A user will interface with this system though the web interface. This web interface may be accessed by typing the IP address of the device into a standard web browser. The interface will allow the user to choose the values for Voltage and Frequency. Once these values have been entered, update scripts on the Raspberry Pi will set the voltage and frequency output of the circuit according to the values entered.

3 Functional Decomposition

This system has four fundamental functional blocks. These include the Web Interface, Raspberry Pi, Minigen Signal Generator, and Amplifier Circuit. The project will be described in terms of these components and their interactions.

3.1 Web Interface

The web interface is hosted on the Raspberry Pi using an Apache web server. This web server displays an interface which allows the user to set a voltage and frequency output by the system. The interface is simple and interactive, implemented using cgi-scripts on the Apache web server.

Our implementation provides several functionalities. Among these are the ability to set: voltage and frequency, sine or triangle or square waveforms, and the ability to set a voltage and frequency for an amount of time. The table displayed in the

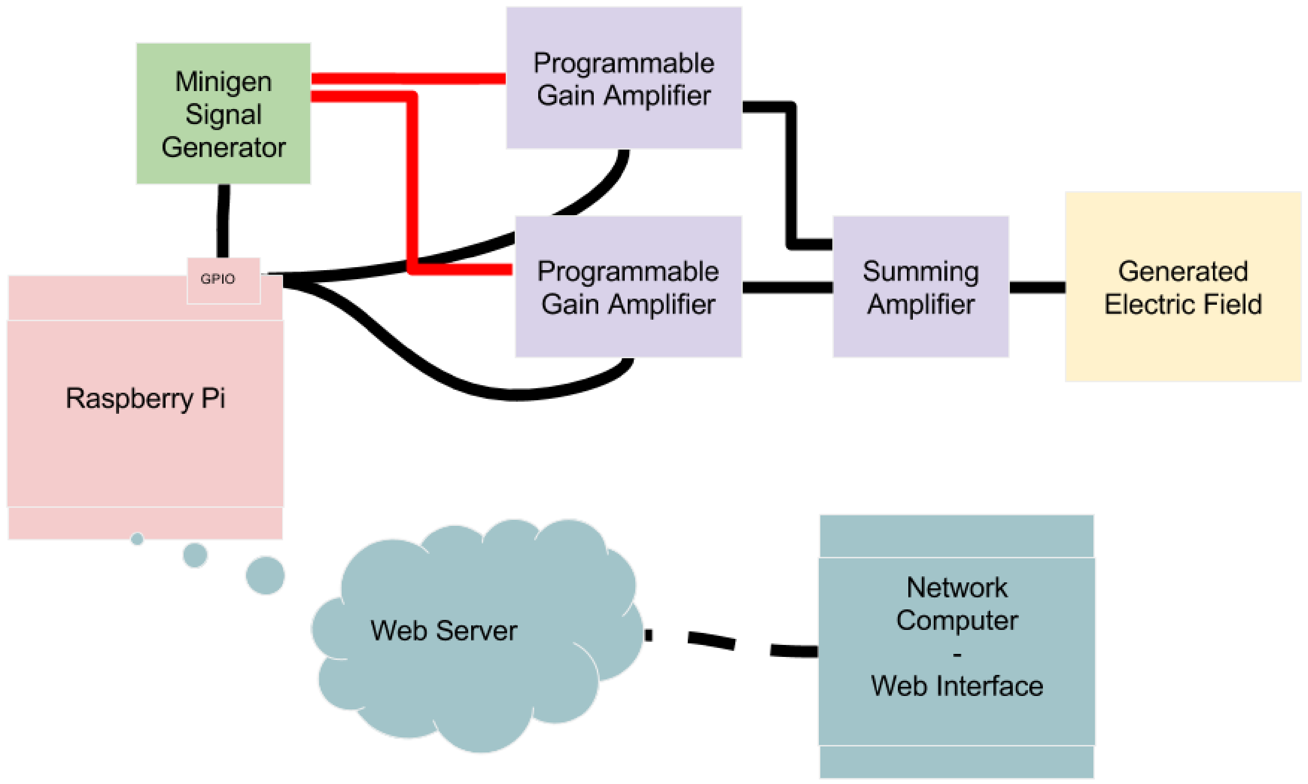
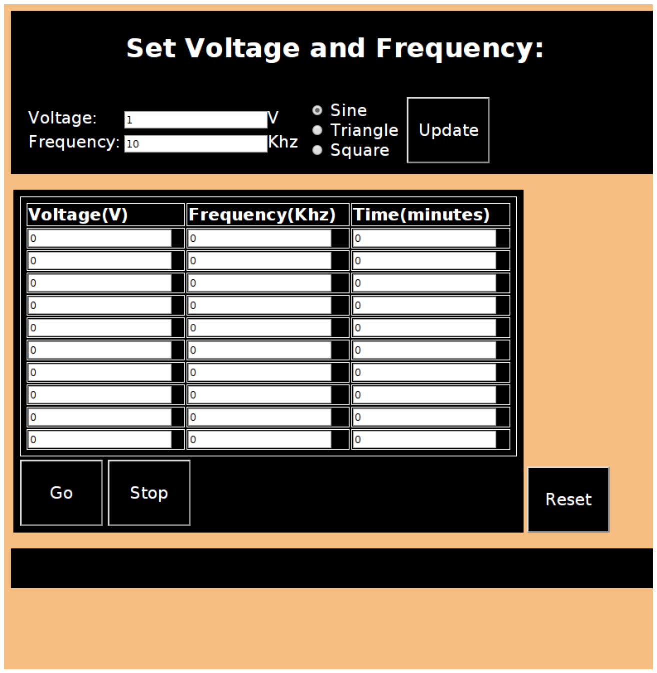


Figure 1: Block Diagram provides an overview of the system components.

figure below provides the ability to set voltage and frequency for the number of minutes specified. The ”Go” button will cause the first voltage and frequency to be set for the corresponding amount of time. After the time has expired, the next voltage and frequency will be set for the corresponding amount of time. This process continues until the table entries are completed or the user presses the ”Stop” button.



Set Voltage and Frequency:

Voltage: 1 V
Frequency: 10 Khz

○ Sine
● Triangle
● Square

Voltage(V)	Frequency(Khz)	Time(minutes)
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

3.2 Web Server

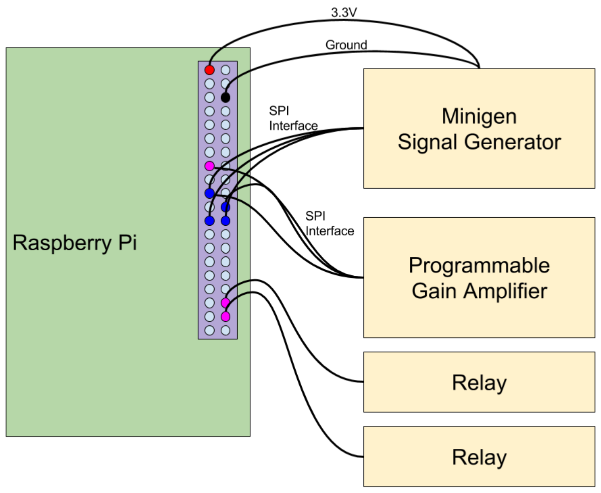
The primary function of the web server is to communicate with the Raspberry Pi. This is the primary method of control afforded to the user by the system. The web pages displayed by the server have the ability to control the voltage and frequency output by the circuit.

Displaying this interface is accomplished by running an Apache web server on the Raspberry Pi. When the user clicks update, the server could executes a cgi-script performing the update functionality.

3.3 Raspberry Pi

The Raspberry Pi will act as the bridge between the user and the circuit. The Raspberry Pi will host a web server allowing the user to interact with the system. Based on the results of this user interaction, the Raspberry Pi will update the state of the GPIO pins. The GPIO pins connect to a circuit causing the output to change based on their state.

In addition to hosting the web server the Raspberry pi is used to communicate with the Minigen Signal Generator and amplifier circuit. This communication is accomplished via the Raspberry Pi's SPI interface and GPIO pins respectively.



3.4 Minigen

The Minigen Function Generator device controls the frequency output by the circuit. Varying the frequency is accomplished by writing to registers present on the Minigen. This communication is completed over SPI between the Raspberry Pi and the Minigen. The frequency produced is a function of the values contained in the Minigen's frequency registers.

The Minigen outputs a waveform from -0.5V to 0.5V. This waveform may be a triangle, square, or sine wave. The voltage output by the Minigen is not variable. Given that the design specification requires a variable voltage, the voltage needs to be adjusted separately. Accordingly, the output of the Minigen is supplied to the input of the amplifier circuit.

The Minigen is controlled by setting five registers, two registers for frequency, two for phase shift and one as a control. There exists no need for phase shifting to meet the design requirements, however the frequency and control registers are needed. By having two frequency registers, data can be sent to one register while it is not in use, followed by a write to the control register to use this register. This allows for a nicer gradient, because the frequency will not change until the entire frequency register is written. The control register also allows for changing between sine, square and triangle waveforms. In the event that the frequency needs to be finely adjusted, this system utilizes the functionality of the control register to modify the way in which writes to the frequency registers are received. The way writes are received by the frequency registers can be varied between two modes. In one mode, two consecutive 14-bit writes to a frequency register are used. In the other mode, one write to the lower 14-bits of the 28-bit frequency register is used. This functionality affords the ability to accurately dial in small changes to the register values quickly.

Until this point, several functional benefits of the Minigen Signal Generator have been discussed. An additional benefit which increases the practicality of this solution is the Minigen's small form factor. The small chip size allows the Minigen to fit easily into a small case with the Raspberry Pi. This is consistent with the system's requisite small footprint.

3.5 Amplifier Circuit

As mentioned in the previous section, the output of the Minigen Function Generator is applied to the amplifier circuit as input. The amplifier also receives input from the GPIO pins of the Raspberry Pi. These GPIO pins act as switches which help to control the output voltage. Based on these inputs the amplifier circuit manages the overall voltage and frequency output.

The project requirements state that the system must generate signals which range from $1V_{pp}$ to $60V_{pp}$. To accomplish this, various circuit components were used to accomplish the amplification. The overall scheme is to split the output voltages into three different ranges. One component is used to adjust the voltage within each of the ranges while other comp are used to change the range the voltage adjuster is acting within.

3.5.1 Summing Amplifier

The summing amplifier is the last component in the amplifier circuit. The overall voltage range which needs to be produced is divided into three segments. Each of these segments have their output connected as input to the summing amplifier. One of the segments utilizes a programmable gain amplifier(PGA) to vary the voltage within one third of the range. While the relays are used to increase the voltage by one third of the overall voltage range when they are turned on.

The advantage of this design is that it allows for three times the number of voltage steps allowed for by a single PGA. This could also be accomplished by using multiple PGA's. Utilizing the relays in increase the voltage instead of additional PGA's allows for reduced software and hardware complexity.

3.5.2 Relay

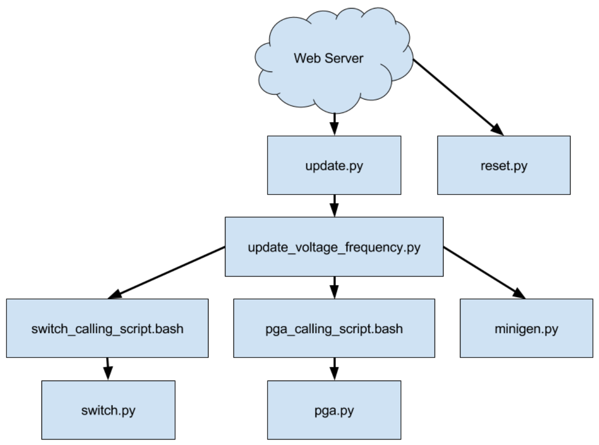
The overall voltage range, $1V_{pp}$ to $60V_{pp}$, is divided into three $20V_{pp}$ ranges. The relay's have two states, on and off, which allow for moment between the ranges. When on, the input the the Amplifier circuit is amplified to $20V_{pp}$. This $20V_{pp}$ signal is input to the summing amplifier. There are two of these relay circuits used. Turning them on or off allows the voltage range to be chosen. These relay circuits are controlled by GPIO pins on the Raspberry Pi.

3.5.3 Programmable Gain Amplifier(PGA)

There are three $20V_{pp}$ voltage ranges. Which range being acted in is chosen by the relay circuits. Within a given range, however, a PGA is used to control the voltage output. In our implementation, a PGA has control over a $20V_{pp}$ range. The PGA has 8 steps within this range. This Device is controlled by the SPI interface on the Raspberry Pi.

4 Software Components

The web interface displayed by the web server hosted on the Raspberry Pi is the user's window into the system. Through this interface, the user can seamlessly control various components of the system cause the desired voltage and frequency to be produced. Previously covered are the hardware components used to accomplish this. This section describes the software counterparts used to control the hardware.



4.1 Modifying Frequency

The frequency output by the circuit is controlled by the Minigen Function Generator. Thus the software components used to modify the output frequency are in essence a method of communication with the Minigen device.

The process begins when the user enters a new frequency value into the web interface and clicks the "update" button. The *update.py* script parses the parameters contained in the URL resulting from the user's update request. These parameters are than passed on to *update_voltage_frequency.py* which determines the appropriate update procedure with which to call *minigen.py*.

4.2 Controlling Voltage

4.2.1 Relay

4.2.2 Programmable Gain Amplifier(PGA)

5 Cost Considerations

The monetary cost of this project is fairly low. The precise costs of components in the future are unknown, but a table of current prices has been provided along with the necessary quantity of each component. The projected cost of op-amps and other electronic components is minimal. The largest expenditure of the project is the purchase of the Raspberry Pi 2 and Minigen Function Generator. The Raspberry Pi 2 package is currently priced at \$99.95 and the Minigen cost at \$29.95. Thinking conservatively the cost of the major hardware will be \$129.90. In addition, the cost of a resistor kit, a capacitor kit, and a handful of op-amps must be included.

The Minigen Function Generator may be acquired from *http://www.sparkfun.com*. This website also provides a resistor kit for \$7.95 which includes all necessary resistors. From this same website, individual capacitors may be purchased at a rate of \$0.25 per capacitor. Operational amplifiers may also be purchased at a rate of \$0.95 per amplifier. Given this, the total cost is approximated at \$152.35. This is far below the maximum allotted funds of \$1,000 specified in the project description.

Item	**Part Number**	**Quantity**	**Price($)**
Raspberry Pi 2 Kit	**-**	**1**	**99.95**
Minigen Function Generator	**AD9837**	**1**	**29.95**
Resistor Pack	**-**	**1**	**7.95**
Capacitor Pack	**-**	**1**	**5.00**
Op Amps	**-**	**8**	**9.50**
Total	**-**	**-**	**152.35**

6 Redesigned Components

In an introductory economics class, it is common that students are taught about sunk costs. Sunk costs refer to those costs which have already been incurred and cannot be recovered. These classes teach that suck costs should not be considered in making future investments. In other words, one should not ask themselves the extent of current investment down a particular path. Rather, one should ask, given what is known now, which investment is more intelligent.

While economics classes teach about sunk costs in the context of monetary investment, the same logic may be applied to other domains. In the case of this project, the investment capital is time and energy. Regardless of how much energy has been invested in a particular design, it is necessary to evaluate alternatives to that design. If these alternatives than look better given what is known about both designs than the design should be modified accordingly to fit the alternative design.

Throughout the course of this project, several components needed to be redesigned. Below is some mention of these designs and the issues encountered which motivated deviation from them.

6.1 Frequency Control

The Raspberry pi is capable of producing square waves by turning the GPIO pins on and off rapidly. We can use this functionality to produce a wave of the frequency indicated by the user. The GPIO pins can also be used to set the voltage by communicating with the circuit how much the output waveform should be amplified. The downside to this approach is the analog circuit component will need to be more complex. The analog circuit needs to output a sine wave. With this approach we would need to integrate the square wave produced by the GPIO pin.

There exist alternatives to using the GPIO pins to generate a signal with a given frequency. We could instead use the GPIO pins on the raspberry pi to communicate with a small signal generator, such as *sparkfun.com*s Minigen Function Generator. This would make programming the Raspberry pi more complex, but could lead to higher quality waveforms. Producing a sine wave using the Minigen signal generator is likely to to produce fewer distortions compared to integrating a square wave produced by the RPIs GPIO pin twice.

6.2 Voltage Control

6.2.1 Digital Potentiometer

The output of the digital potentiometer has 128 steps. This Translates into our ability to set 128 different gains on our amplifier. We will need multiple stages of amplifier to go between 1 and 60 Vpp. The most prominent reason for this is due to the gain bandwidth of the op-amps. We will not be able to have a large gain while still producing a frequency of 1Mhz.

One problem we foresee with the digital potentiometer is that it cannot handle a large amount of power. This may force us to come up with different amplifier configurations, or use the digital potentiometer in a different way. Another way we could possibly use this device is as an attenuator at the input to the amplifier.

Another problem which might arise with the digital potentiometer is the capacitance of the wiper. We dont have any context for understanding how much this will affect the output signal. According to some preliminary calculations, we have determined that the capacitance will not present a large problem. This design was replaced by a programmable gain amplifier.

6.2.2 Transistor switch

The overall voltage output by the amplifier circuit varies between $1V_{pp}$ to $60V_{pp}$. In our implementation, this range is segmented into three parts of $20V_{pp}$ each. Each of the three segments are input to a summing amp to create up to $60V_{pp}$ overall.

A PGA is used for fine adjustment of one $20V_{pp}$ range while the other two ranges are either $20V_{pp}$ or $0V_{pp}$. In other words, there ranges are turned on or off.

The original design for turning on and off the ranges required that a BJT transistor switch circuit be used. The intent was for this circuit to have either $20V_{pp}$ or $0V_{pp}$ depending on a GPIO pin from the Raspberry Pi. This design did not work as intended. Even when the transistors were off there was still current leakage. This current leakage caused problems when input to the summing amplifier. For this reason, the BJT transistor switch ciruits were replaced with solid state relays.

7 Known Issues

The current solution has some noteworthy issues which need to be addressed. Issues are defined as unintended issues of unknown origin. Possible reasons for each of the issues are discussed as well as the accommodations made in this implementation to lessen their impact. Potential solutions are also mentioned.

7.1 B23

In the current prototype, there is an ongoing issue with the Minigen Function Generator. This device works as expected in most cases. The exception occurs when bit 23 of either frequency register is set high. This case will cause the Minigen device to produce a $4Mhz$ sine wave. There is potential that the specific Minigen device used is responsible for this bug. Further testing is required.

The solution currently implemented is to detect conditions which will cause the error and avert these situations problematically. Whenever bit 23 is set to high in a frequency register write, the current implementation sets bit 23 to low and sets all less significant bits to high.

8 Concluding Remarks

A Operation Manual

A.1 Setup

A.2 Demo

B Initial Designs

This section covers the designs that have been tried throughout the course of the project. These were problems with each of these designs which motivated a change from that design to a more resent design. Designs are presented in roughly chronological order in each section to perhaps make it more clear what motivated a particular design choice. Each component underwent various adaptations over the duration of the project.

B.1 Frequency Control

B.2 Voltage Control

B.3 Web Interface

C Other Considerations

D Code

Listing 1: Bash script used to evoke GPIO library

```
#! /bin/bash

# open python thing
cd /home/pi/cpre494/cpre492/cgi-bin

echo -e "import pga\np_amp=pga.pga()\np_amp.setGain($1)" | sudo python
#sudo python import pga
#sudo python p_amp = pga.pga()
#sudo python p_amp.setGain(-1)
```

Listing 2: Update Website Script

```
#! /usr/bin/python2.7

import cgi
import cgitb
import update_voltage_frequency

# create instance of field storage to get values
parameters = cgi.FieldStorage()

# get the data from the fields
#update_voltage = parameters.getvalue('update_voltage')
#update_frequency = parameters.getvalue('update_frequency')

voltage_value = parameters.getvalue('voltage')
frequency_value = parameters.getvalue('frequency')

# Update the frequency and voltage
update_voltage_frequency.update_voltage(voltage_value)
update_voltage_frequency.update_frequency(frequency_value)

# This script is designed to count as a test
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>CGI-Update-Procedure</title>"
print "<head>"
print "<body>"
print "<h1>Current Values:</h1>"
print "<h3>Voltage: %s V</h3>" % (voltage_value)
print "<h3>Frequency: %s Khz</h3>" % (frequency_value)
print "</body>"
print "</html>"

# Reprint index.html so that the user may change the voltage again
with open('/home/pi/cpre494/cpre492/www/index.html', 'r') as file:
    line = file.readline()

    while line:
        #print "\""+line+"\""
        print line
        line = file.readline()
```

Listing 3: Update Voltage and Frequency Script

```
#! /usr/bin/python2.7

import spidev
import minigen
#import cPickle as pickle
#import pga
import subprocess

#Designed to communicate with a Minigen connected to the GPIO pins
spi = spidev.SpiDev()

# Test function
def main():
  print 'running in test mode'

  # Test setting the frequency using spi
  #update_frequency(100)

  # Test setting the voltage using I2c
  update_voltage("22")

# define variables
#minigen_pickle_file = "/tmp/mini_pickle"
#digital_pot_pickle_file = "/tmp/pot_pickle"

# Update the voltage level to the specified value.
# This is not the voltage output by the minigen,
# Instead it is the voltage output by the circuit as a whole.
def update_voltage(voltage):
  # range of each step fed into the summer
  # pga will have many steps between 0 -> step_range
  # ex: if max value of the pga is 10vpp, then step_range is 10
  # ADJUST STEP RANGE TO COSNTANT AMPLIFIER OUTPUT
  step_range = 10
  pga_step_size = step_range / 7.0

  # compute voltage values
  pga_voltage = int(voltage)

  if ( pga_voltage > step_range ):
    state_0 = "1"
    pga_voltage -= step_range
  else:
    state_0 = "0"

  if ( pga_voltage > step_range ):
    state_1 = "1"
    pga_voltage -= step_range
  else:
    state_1 = "0"

  # convert the pga voltage into a pga gain
  pga_gain = pga_voltage / pga_step_size

  # debug: print out pga_voltage and states of pins
#  print str(pga_voltage)
 # print str(state_0)
  #print str(state_1)
#  print str(int(round(pga_gain)))

  # call a script that will set the pga values
  subprocess.call("./pga_calling_script.bash " + str(int(round(-1*pga_gain))), shell=True)

  # set switch 0
  subprocess.call("./switch_calling_script.bash 0 " + state_0, shell=True)

  # set switch 1
  subprocess.call("./switch_calling_script.bash 1 " + state_1, shell=True)

  #print 'voltage updated'

  # make an instance of the voltage_regulator class to handle the connection
  #vr = voltage_regulator.voltage_regulator()
  #vr = get_pickle_digital_pot()

  # ask vr to set the voltage to the given value
#  vr.set_voltage(voltage)

  # update pickled information
  ###set_pickle_digital_pot(vr)

  # preform cleanup actions
  #vr.close_regulator()

# Update the frequency to the specified value. Values are given in Khz.
def update_frequency(frequency):
  #print 'frequency updated'

  # make an instance of the minigen class to handle the connection
  m = minigen.minigen()
 # m = get_pickle_minigen()

  # ask the minigen to set the new frequency
  m.setFrequency(float(frequency)*1000)

  # update pickled information
  ###set_pickle_minigen(m)

  #close the conection
  m.close()

# attempt to grab pickeled information about minigen
# if no pickle is found, create new minigen object
#def get_pickle_minigen():
 # try:
   # m = pickle.load( open( minigen_pickle_file , "rb" ) )
    ##print "pickle loaded successfully"
  #except:
   # m = minigen.minigen()
    #print "new object created"

 # return m

# attempt to grab pickeled information about ditital pot
# if no pickle is found, create new minigen object
#def get_pickle_digital_pot():
#  try:
#    vr = pickle.load( open( digital_pot_pickle_file ,  "rb" ) )
#  except:
#    vr = voltage_regulator.voltage_regulator()

#  return vr

# set pickeled information about minigen
#def set_pickle_minigen(m):
#  pickle.dump( m, open(minigen_pickle_file  , "wb" ) )

# set pickeled information about digital pot
#def set_pickle_digital_pot(vr):
#  pickle.dump( vr, open(digital_pot_pickle_file , "wb" ) )

if(__name__ == "__main__"):
  main()
```

Listing 4: Minigen Control Script

```
#! /usr/bin/python2.7

import spidev
import time
import sys
# Designed to provide some of the functionality from SparkFun_MiniGen.cpp
# designed so that you only need to use set_frequency to set the frequency
class minigen:
  # initialize the connection with the minigen
  def __init__(self):
    self.spi = spidev.SpiDev()

    # open(bus, device)
    self.spi.open(0, 0)

    # minigen is driven at 40Mhz
    #self.spi.max_speed_hz = 15000000

    self.controlReg = [False]* 16
    self.controlReg[16-13] = True

    self.freqReg0 = [False]*32
    self.freqReg1 = [False]*32

    self.freqReg0[31-30] = True
    self.freqReg0[31-14] = True

    self.freqReg1[31-31] = True
    self.freqReg1[31-15] = True

    self.fudgeFactor = 1

##########      Control Register 16Bits
#    Bit Number     Name     Function
#    D15           Addr1      Always 0         D15 and D14 is the address of the control register
#    D14           Addr0      Always 0
#    D13           B28        When 1: allows a complete word to be loaded into a freq reg with two consecutive write. First contains 14 LSB, second contains
#                             14 MSB. (First two bits is freq reg addr)  Consecutive writes to the same freq register is not allowed, you must alternate.
#                             When 0: Configures the 28bit freq reg is act as two 14 bit regs. One contains 14 LSB, the other 14MSB. This allows for coarse, or fine
#                             grain tuning. HLB defines which to change.
#
#
#
#    D12           HLB        This allows the user to continiously load the MSB or LSB of a freq reg. Ignoring ther other 14 bits. When B28 = 1, this is ignored.
#                             When 1: Allows write to 14 MSB
#                             When 0: Allows write to 14 LSB
#
#
#
#    D11           FSEL       Selects either freq0 or freq1
#    D10           PSEL       Selects either phase0 or phase1
#    D09        reserved       0
#    D08        RESET      When 1: resets internal regs to 0. When 0: disables the reset function.
#    D07        Sleep1       Enables or disables MCLK
#    D06        Sleep12      Powers down on chip DAC
#    D05        OPBITEN      0
#    D04                     0
#    D03
#    D02           TODO
#    D01
#    D00

  def chooseFreq0(self):
    self.controlReg[15-11] = False
  def chooseFreq1(self):
    self.controlReg[15-11] = True
  def enableB28(self):
    self.controlReg[15-13] = True
  def disableB28(self):
    self.controlReg[15-13] = False
  def enableHLB(self):
    self.controlReg[15-12] = True
  def disableHLB(self):
    self.controlReg[15-12] = False

  def sendControlReg(self):
    controlRegNum = self.boolListToInteger(self.controlReg)
    self.spi.xfer([controlRegNum >> 8, controlRegNum & 0xFF])

  def getControlReg(self):
    return (self.boolListToInteger(self.controlReg))


  #Frequency Functions
  #
  #Frequency Registers are set up as one 1 32bit register with [31-30] and [15-14] defined to be the address
  #

  def setFreqRegister(self,freqReg, isMSB, num):
    if(num >  0x3FFF):
      return -1
    bitString = bin(num)[2:][::-1]
    if(isMSB == 1):
      x = 15
    else:
      x = 31
    for i in bitString:
      if int(i) == 1:
        if(freqReg == 0):
          self.freqReg0[x] = True
        else:
          self.freqReg1[x] = True
      else:
        if(freqReg == 0):
          self.freqReg0[x] = False
        else:
          self.freqReg1[x] = False
      x= x - 1
    return 1


  def setFreq0MSB(self,num):
    return self.setFreqRegister(0,1,num)

  def setFreq0LSB(self,num):
    return self.setFreqRegister(0,0,num)

  def setFreq1MSB(self,num):
    return self.setFreqRegister(1,1,num)

  def setFreq1LSB(self,num):
    return self.setFreqRegister(1,0,num)

  def setEntireFreqReg0(self,num):
    actualValue = self.calculateFrequency(num)
    if(actualValue > 0x3FFFFFFF):
      return -1
    self.setFreq0LSB(actualValue & 0x3FFF)
    self.setFreq0MSB(actualValue >> 14)
    return 1

  def setEntireFreqReg1(self,num):
    actualValue = self.calculateFrequency(num)
    if(actualValue > 0x3FFFFFFF):
      return -1
    self.setFreq1LSB(actualValue & 0x3FFF)
    self.setFreq1MSB(actualValue >> 14)
    return 1


  def getFreqReg0(self):
    return (self.boolListToInteger(self.freqReg0))
  def getFreqReg1(self):
    return (self.boolListToInteger(self.freqReg1))

  def sendFreqReg0MSB(self):
    sendFreqRegNum = self.boolListToInteger(self.freqReg0)
    self.spi.xfer([sendFreqRegNum >> 24, (sendFreqRegNum >> 16) & 0xFF])

  def sendFreqReg0LSB(self):
    sendFreqRegNum = self.boolListToInteger(self.freqReg0)
    self.spi.xfer([(sendFreqRegNum >> 8) & 0xFF, (sendFreqRegNum) & 0xFF])

  def sendFreqReg1MSB(self):
    sendFreqRegNum = self.boolListToInteger(self.freqReg1)
    self.spi.xfer([sendFreqRegNum >> 24, (sendFreqRegNum >> 16) & 0xFF])

  def sendFreqReg1LSB(self):
    sendFreqRegNum = self.boolListToInteger(self.freqReg1)
    self.spi.xfer([(sendFreqRegNum >> 8) & 0xFF, (sendFreqRegNum) & 0xFF])

  def setFrequency(self, freq):
    if(freq < 10000):
      return -1
    self.enableB28()
    self.chooseFreq1()
    self.sendControlReg()
    self.setEntireFreqReg0(freq)
    self.sendFreqReg0MSB()
    self.sendFreqReg0LSB()
    self.chooseFreq0()
    self.sendControlReg()
    return 1

  def setFrequency1(self, freq):
    self.disableB28()
    self.enableHLB()
    self.chooseFreq1()
    self.sendControlReg()

    calculatedValue = self.calculateFrequency(freq)
    if(calculatedValue > 0x3FFF):
      return -1

    MSB = (calculatedValue >> 14) & 0x3FFF
    self.setFreqRegister(0, 1, MSB)
    self.sendFreqReg0MSB()
    self.disableHLB()
    self.sendControlReg()
    LSB = calculatedValue & 0x3FFF
    self.setFreqRegister(0,0,LSB)
    self.sendFreqReg0LSB()

    self.chooseFreq0()
    self.sendControlReg()

  def calculateFrequency(self, num):
    #print "Calculated Value: " + str(int(num/(.0596)))*self.fudgeFactor
    return int(num/(.0596))*self.fudgeFactor

  def close(self):
    self.spi.close()

  #Converts a boolean array to a number
  def boolListToInteger(self,lst):
    return int( ''.join(['1' if x else '0' for x in lst]),2)

# Test function
def main():
  print 'running in test mode'
  m = minigen()
  m.setFrequency(float(sys.argv[1]))
  print "Frequency Register: " + bin(m.getFreqReg0())
  m.close()


if(__name__ == "__main__"):
  main()
```

Listing 5: PGA Control Script

```
#! /usr/bin/python2.7

import RPi.GPIO as GPIO
import time
import os
import sys

class pga:

  def __init__(self):
    #Switch user
    #sudoPassword = 'root'
    #command = 'sudo␣-i'
    #p = os.system('echo␣%s|sudo␣-S␣%s' % (sudoPassword, command))

    #Default Gx pins
    self.pinGx = [4, 17, 27]

    #6910-3
    self.gainList = [0, -1, -2, -3, -4, -5, -6, -7]

    #6910-2
    #self.gaintList = [0, -1, -2, -4, -8, -16, -32, -64]

    #6910-3
    #self.gainList = [0, -1, -2, -5, -10, -20, -50, -100]


    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    self.updatePins()


  def updatePins(self):
    GPIO.cleanup()
    for pos in range(0,3):
      GPIO.setup(self.pinGx[pos], GPIO.OUT)

  def setPinG0(self, pin):
    self.pinGx[0] = pin
    self.updatePins()

  def setPinG1(self, pin):
    self.pinGx[1]= pin
    self.updatePins()

  def setPinG2(self, pin):
    self.pinGx[2] = pin
    self.updatePins()

  def setGainList(self, list):
    if(len(list) == 8):
      self.gainList = list
      return True
    return False

  def getGainList(self):
    return self.gainList

  def printGainList(self):
    print self.gainList

  # Only Allows for gains set in the given list
  def setGain(self, gain):
    try:
      binNum = '{:03b}'.format(self.gainList.index(gain))
```

```
            #print binNum
            binNum = binNum[::-1]
            for pos in range(0, 3):
                if int(binNum[pos]) == 1:
                    GPIO.output(self.pinGx[pos], 1)
                else:
                    GPIO.output(self.pinGx[pos], 0)
        except ValueError:
            print "Gain not Found"


def main():
    print "Running PGA test"
    p = pga()
    p.setGain(int(sys.argv[1]))


 #self.gainList = [0, -1, -2, -5, -10, -20, -50, -100]
if(__name__ == "__main__"):
    main()
```

Listing 6: Reset Script

```
#! /usr/bin/python2.7

# TODO the purpose of this script is to reset the minigen to a default state


# This script is designed to count as a test
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Reset-Procedure</title>"
print "<head>"
print "<body>"
print "</body>"
print "</html>"

# reprint index.html
with open('/home/pi/senior_design/www/index.html', 'r') as file:
    line = file.readline()

    while line:
        #print "\""+line+"\""
        print line
        line = file.readline()
```