Appendix

*__Revision 1__ refers to the program revisions that were implemented between the initial constant force scheme addressed in the report and the subsequent revised constant force scheme addressed in the report. The changes implemented in revision 1 only effect the constant force testing scheme. The program code listed in this appendix includes revision 1 and pre-revision 1 code, however, the pre-revision 1 code has been rendered inactive.

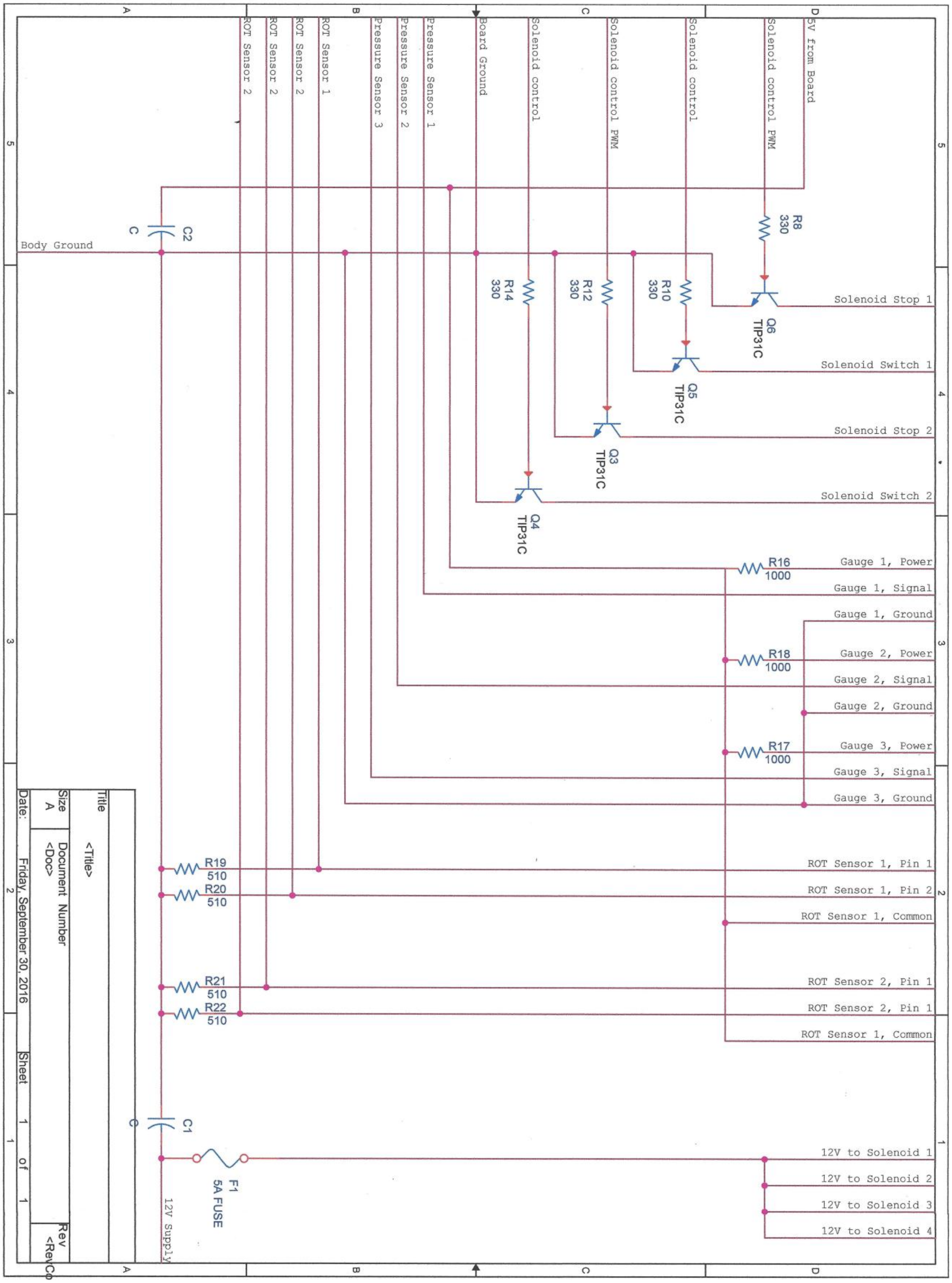

Figure 1: Test Apparatus Electrical Schematic

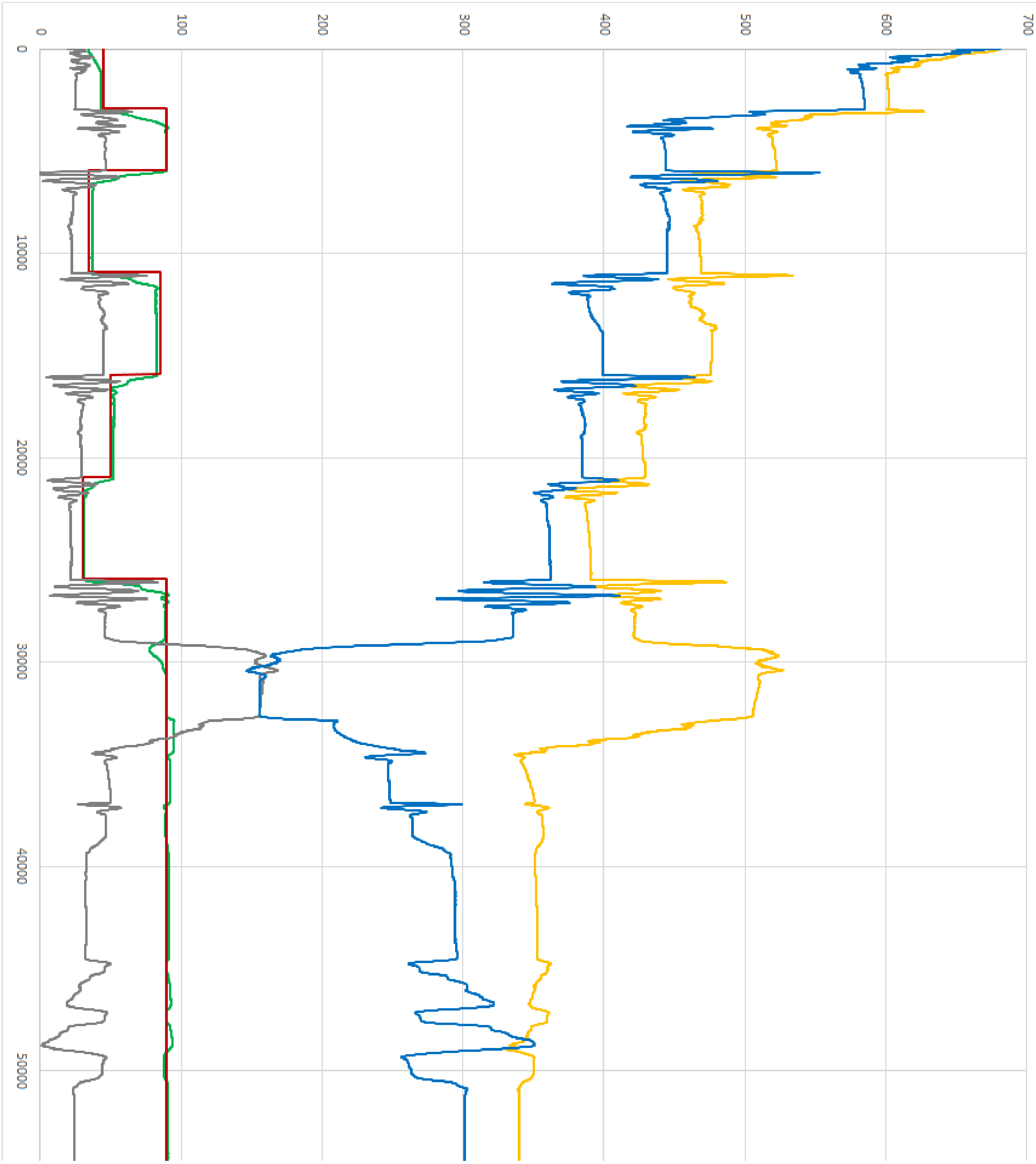

Figure 2: Final Test Data, Part A, Before Revision 1.

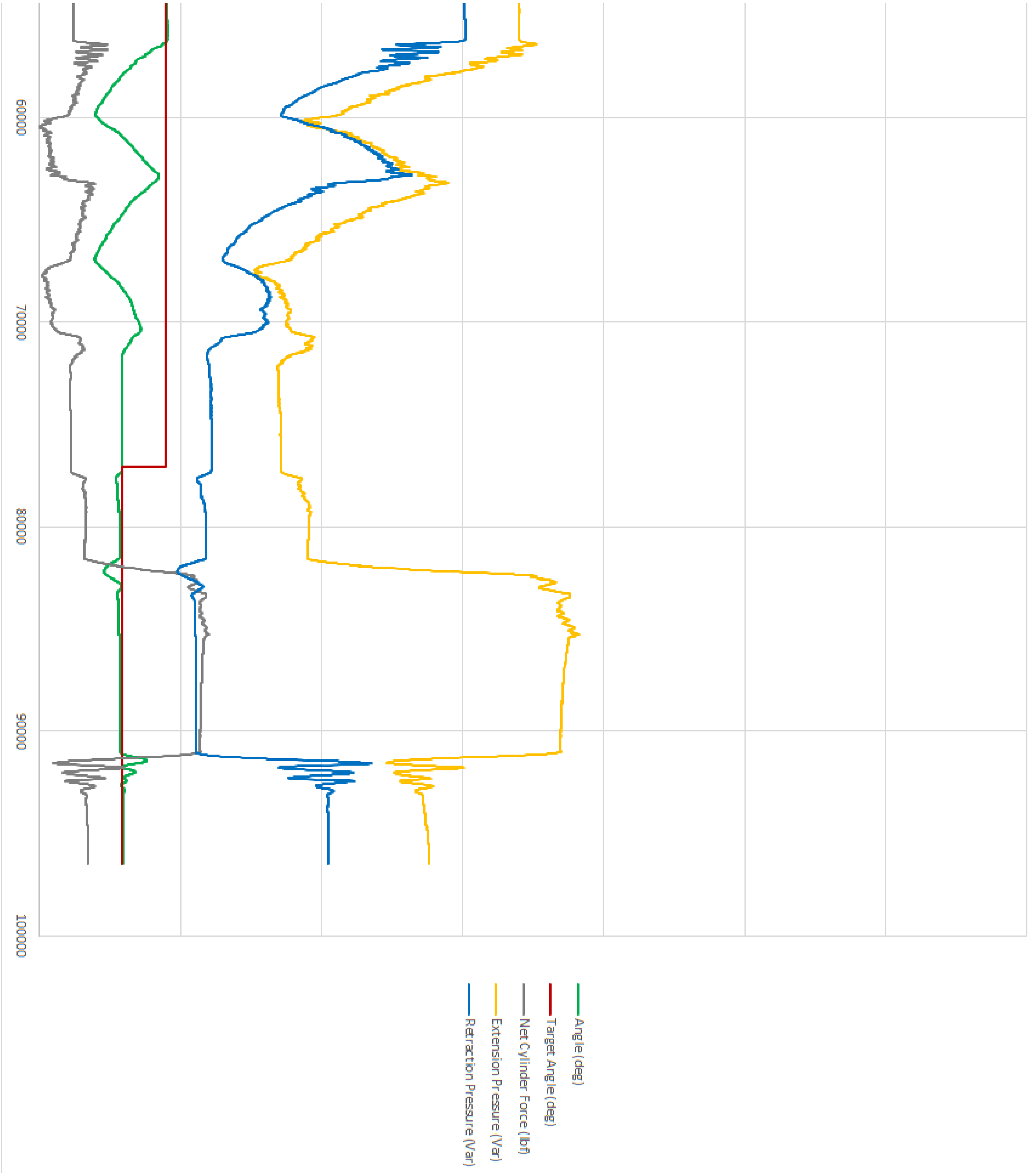

Figure 3: Final Test Data, Part B, Before Revision 1.

Figure 4: Final Test Data, Part A, Revision 1.

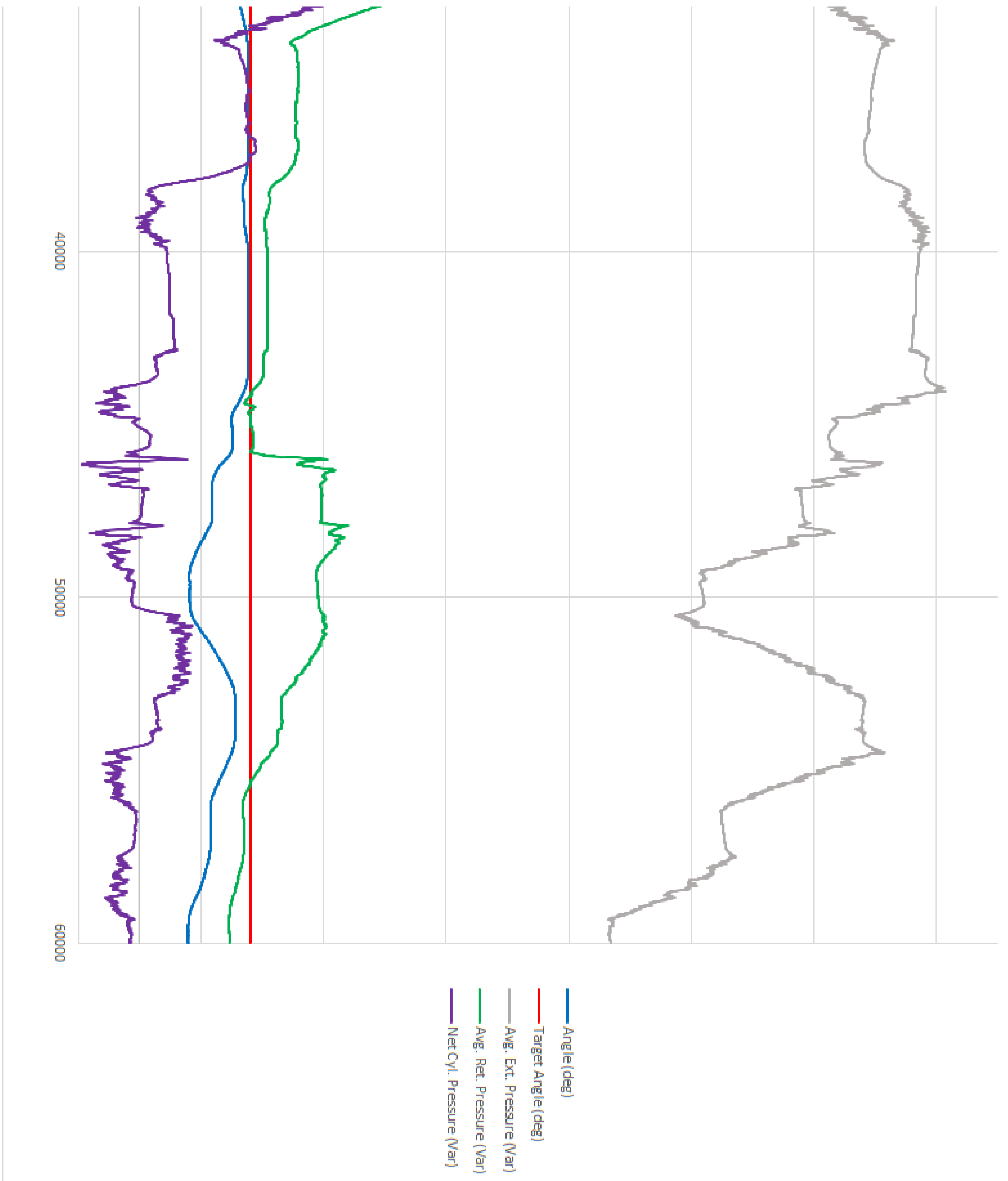

Figure 5: Final Test Data, Part B, Revision 1.

Complete Program:

```
//Subroutine List
  //Solenoid: The current solenoid control subroutine now encompasses both of the
          //following subroutines: SolenoidExt, SolenoidRet.
  //SolenoidExt: Joins all functions for the extend solenoids. Offering full control
                  //over the extend side of the cylinder. PWM modulated for
                  //Intermediate power values. PWM enabled.
  //SolenoidRet: Joins all functions for the extend solenoids. Offering full control
                  //over the extend side of the cylinder. PWM modulated for
                  //Intermediate power values. PWM enabled.
  //InitializeSystem: Defunct. Moves the cylinder through its full range of motion,
taking
        //readings at the extremes to calculate the increment/degree for each rotation
        //sensor as well as the offset and max pressure for each. Also functions to
        //reset the cylinder to zero position before starting the main program.
        //This subroutine is mostly Defunct, but it is still used to reset position
        //and record max pressures
  //UpdAngle: This subroutine queries the anguar sensors, calculates angular positions
        //(4 values), averages them and posts the values to global variables for later
        //access.
  //PresUpdate: Calculate running average of pressures on each side of the cylinder.
        //Updates the global variables PEAvg and PRAvg when called.
  //StaticForce: This subroutine calculates the static load when the arm is static.
      //Calling this subroutine after load has been calculated will return the static
      //pressure at the current angle. The static pressure is used to determine the
      //pressure required to hold the arm steady.
  //PressurePID: This subroutine contains the PID scheme utilized in constant pressure
      //operations.
  //Trigger: This subroutine contains the algorithm that detects a constant upward
      //pressure for a defined period. Returns a value when true, thereby allowing
      //the program to switch modes
  //PressureMeasurment: This subroutine contains algoritm that averages the high and
      //low side pressures over a number of periods. Acts to smooth PWM induced noise
      //for further operations.
  //PWM Constant: This subroutine contains the PID for the constant positioning scheme.
      //This subroutine also contains an earlier model that is now defunct, but
      //certain variables are interlaced, so it cannot be commented out entirely.
  //PressureTest: This subroutine, when active, moves the leg throughout a defined
      //angle, taking measurements at defined intervals, allowing the formation of a
      //pressure vs angle plot.
  //BodePlot: This subroutine, when active, collects, calculates and exports the data
      //required to form magnitude and phase plots.
  //TapTap: This was an early attempt to develop an algorithm to detect user input
      //to switch modes. Abandoned due to the excess noise induced in the pressure
      //lines by PWM operations. It would always false trigger because the noise
      //is almost identical to the signature pressure variations of a double tap.
```

```
//User Control
int Main_User_Mode = 0; //0 for defined angles, 1 for constant force.
int Total_angle = 94; //Defunct
float AngleTolerance = 5;  //Angles that the left and right sensors must be within
                            //of each other or program will be stopped to prevent
                            //malfunction. +/- Angle Tolerance
float SetAngle = 45.00;
float CylinderForceLimit = 10;
float Main_Secondary_Pressure_Tolerance = 5;  //Pressure tolerance for constant
                                                //pressure loop.

//Solenoid Control
    //Solenoid Pins
int SolenoidDir1 = 5; //Solenoid Extending directional pin number
int SolenoidTrig1 = 10; //Solenoid Extending trigger pin number
int SolenoidTrig2 = 9; //Solenoid Retract trigger pin number
int SolenoidDir2 = 2; //Solenoid Retract directional pin number

int SwitchPin = A8; //Pin for manual switch

//Solenoid control setting overrides
float SolExtPWMOffset = 37;  //Min length of solenoid pulsing, experimental
float SolRetPWMOffset = 37;  //Min length of solenoid pulsing, experimental

//Solenoid register reset
int myPrescaler = 7;          // 7 corresponds to 30 hz.
int myEraser = 7;              // this is 111 in binary and is used as an eraser

//Global Variables
unsigned long CurrentTime = 0;
unsigned long StartTime = 0;
int Main_Setting = 0; //0 is standard setting (defined angles), 1 is con pres setting
int Main_Setting_RUNONCE = 0;  //This variable is 0 for the first run in the main
                                //loop only. Afterwards it equals 1 for all cycles.
unsigned long Main_prevtime = 0;  //Time of previous cycle through main loop.
float AngleHigh = 0;  //Current high format angle of leg. Call UpdAngle() before use.
                        //Value is highest at zero position.
float AngleLow = 0;   //Current low format angle of leg. Call UpdAngle() before use.
                        //Value is lowest at zero position.
float AngleAvg = 0;   //Average of all four angle inputs, starting from Zero position.
float CylinderStaticForce = 0;  //Net pressure of cylinder when under static load at
                                  //designated angle.
int Main_switch = 0;  //Switch value used in operation of main loop.
float Power_Factor = 0; //Solenoid PWM power factor
float PPIDoffset = 0; //Offset for pressure PID. Acts in an additive fashion.
                        //PWM power = PPID + PPIDoffset.
int SEC_Run_State = 0;
//End Global Variables

//Pressure variables (for subroutine PresUpdate)
int PavgExt[11] = {0,0,0,0,0,0,0,0,0,0,0};
int PavgRet[11] = {0,0,0,0,0,0,0,0,0,0,0};
float PEAvg = 0;
float PRAvg = 0;
float PEAvg_prev = 0;
float PRAvg_prev = 0;
int Pinit = 0;  //counter to initialize arrays (prevents subroute from functioning
```

```
                  //until array has been first built with current values
  float NetCylinderForce = 0; //instantaneous net cylinder force
  float NetCylinderForceAvg = 0;  //Average net cylinder force (averaged over 10 cycles).
  float NCFext = 0; //Net cylinder force during extending operation (avoids using the
                    //retract side pressure sensor to avoid PWM fluctuation error)
  float NCFret = 0; //Net cylinder force during retraction operation (avoids using the
                      //extension side pressure sensor to avoid PWM fluctuation error)
  //End pressure Variables

  //Trigger Subroutine Variables
  int Trigger_State = 0;
  int Trigger_Output = 0;
  unsigned long Trigger_Start = 0;
  //End trigger subroutine variables

  //TapTap Variables
//  float Pavg[2][101]; //hash with Pavg data stored, 100 entries at a time. Reference
                        //as Pavg[0,1][0-100] where in first [], 0 denotes ext
                        //& 1 denotes ret
//  int TTfirstrun = 0;

  //PWM_Constant subroutine variables
  float PWM_Kp = 3;           //Kp constant for current PID, 3.5, 6
  float PWM_Ki = 2.7;       //Ki constant for current PID, 0.00175, 2.7
  float PWM_Kd = 0;             //Kd constant for current PID
  unsigned long PWM_LastTime = 0;
  float PWM_DPrev = 0;
  float PWM_ITot = 0;
  //End PWM_Constant variables

  //Pressure PID controller
  //Note: PKp, PKi, PKd are global variables
  float PKp = 2.5;
  float PKi = 0;
  float PKd = 0;
  unsigned long PIDP_LastTime=0;
  float PIDP_DPrev = 0;
  float PIDP_ITot = 0;
  //End Pressure PID controller

  //Static Force variables
    float SF_Load = 0;  //Calculated Load by subroutine
    float StaticForceAtAngle = 0; //Calculated static force at current angle.
  //End Static Force variables

  //Temporary Variables
  int temp = 0;
  int x = 0;
  int row = 0;
  //End temp variables

  //BODE subroutine variables
//  float BODE_STATE = 0; //state variable: 0 is initial state, 1 is testing state,
                          //2 is transition to next state (resets to 0 state with
                          //target angle)
//  float BODE_n = 0.00;  //step variable for the Bode subroutine. Increments by 1 for
                          //each testing cycle to max steps.
```

```
//  unsigned long BODE_current_time = 0;
//  unsigned long BODE_previous_time = 0;
//  unsigned long BODE_end_time = 0;
//  float BODE_max_value = 0;
//  float BODE_min_value = 0;
//  unsigned long BODE_Phase_Applied_Start = 0;   //Time of applied change in set point
                                                    //for Phase Shift Measurement
//  unsigned long BODE_Phase_Measured_Trigger = 0;  //Time of measured maximum amplitude
                                                      //for Phase Shift Measurement
//  float BODE_Phase_min = 0;
//  int BODE_Phase_trigger = 0;
//  int BODE_Phase_counter = 0;
  //END BODE subroutine variables

  //Calibration Variables (These are set by the InitializeSystem() Subroutine)
//  double CAL_RA1offset = 0;  //offset from zero position
//  double CAL_RA2offset = 0;  //offset from zero position
//  double CAL_LA1offset = 0;  //offset from zero position
//  double CAL_LA2offset = 0;  //offset from zero position
//  float CAL_RA1 = 0;
//  float CAL_RA2 = 0;
//  float CAL_LA1 = 0;
//  float CAL_LA2 = 0;
//  float CAL_Pin = 0;
//  float CAL_Pext = 0;
//  float CAL_Pret = 0;
  //End Calibration Variables

  //Pressure Test variables
  unsigned long PT_lastrun = 0; //time record of last run
  float PT_tolerance = 0.5; //tolerance for test deadband
  float PT_increment = 0;   //calculated increment size
  //End Pressure Test variables



void setup() {
  //Reset Registers to 30 hertz base frequency
  TCCR2B &= ~myEraser; //this op. (AND plus NOT), set the three bits in TCCR2B to 0
  TCCR2B |= myPrescaler; //this op. (OR), replaces the last three bits in TCCR2B with
                          //our new value 011

  //Initialize Solenoids
  pinMode(SolenoidDir1, OUTPUT);
  pinMode(SolenoidTrig1, OUTPUT);
  pinMode(SolenoidTrig2, OUTPUT);
  pinMode(SolenoidDir2, OUTPUT);
  //End Initialize Solenoids

  //Pressurize cylinder, both sides for subsequent actions.
  Solenoid(255,-255);
  delay(3000);

  //Set Times
  CurrentTime = millis();
  StartTime = millis();
  PWM_LastTime = millis();
```

```
  PIDP_LastTime = millis();

  //General Data collection code
//  Serial.begin(128000); // opens serial port for data logger
//  Serial.println("CLEARDATA");
//  Serial.println("LABEL,Time,MilliSec,Angle (deg),Target Angle (deg),Ext Pavg (Var),
//      Ret Pavg (Var),Inlet Pressure,Extend Pressure,Retract Pressure");


  //BODE Data collection
//  Serial.begin(128000);
//  Serial.println("CLEARDATA");
//  Serial.println("LABEL,Time,Millis,Angle,Set Angle,Cyl. Force,Static Force @ angle,
//  Ext. Pressure,Ret. Pressure");

//  Serial.begin(128000);
//  Serial.println("CLEARDATA");
//  Serial.println("LABEL,Time,Millis,Angle,PEAvg,PRAvg");

  //Serial interface
//  Serial.begin(250000);  //open serial port for serial interface

//  InitializeSystem();  //Reset position and record maximum pressures. Defunct
}



void loop() {
  CurrentTime = millis();

  UpdAngle();  //Update global angle variables  (read, calculate, and update)
  PresUpdate();  //Update global pressure variables (read, calculate, and update)

  PID_Pressure();

//Bode plot subroutine. This subroutine is disabled except while forming bode plot.
//  Bode(55,40,6,0.05,100);

  //User switch.
  if(analogRead(SwitchPin) >= 750){
    if(Main_User_Mode == 0){
      Main_User_Mode = 1;
    }else if(Main_User_Mode == 1){
      Main_User_Mode = 0;
    }
    delay(1000); //Delay 1 second to prevent double trigger
    StaticForce(1);  //Update static force at changeover
  }

  if(Main_User_Mode == 0){  //Define const angle angles
//Testing: Change angles for each time period.
    if( (CurrentTime - StartTime) > 26000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 90;
    }else if( (CurrentTime - StartTime) > 21000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 30;
    }else if( (CurrentTime - StartTime) > 16000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 50;
```

```
    }else if( (CurrentTime - StartTime) > 11000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 85;
    }else if( (CurrentTime - StartTime) > 6000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 35;
    }else if( (CurrentTime - StartTime) > 3000 && (CurrentTime - StartTime) < 30000){
      SetAngle = 90;
    }
  }

  //Current movement routine.
  Power_Factor = PWM_Constant();
  float tolerance = 2; //Degrees about target to cut movement to limit solenoid cycles.
  float tolerance_compare = abs(SetAngle - AngleAvg);

  if(Main_User_Mode == 0){  //User mode
    if(Main_Setting == 0){  //Defined angles mode, pressure not utilized for positioning
      if(tolerance_compare < tolerance){    //Deadband
        //if first run and stable for 2 seconds.
        if(Main_Setting_RUNONCE == 0 && (CurrentTime - Main_prevtime) > 2000){
          Main_Setting_RUNONCE = 1;
          Main_Setting = 0; //Main loop override
        }
        if((CurrentTime - Main_prevtime) < 1000){ //User trigger
          //Update once a second
          PRAvg_prev = PRAvg;
          PEAvg_prev = PEAvg;
        }
        Solenoid(0,0);
        StaticForce(1); //Update Applied load force while stopped. For constant pressure.
        PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.
        UserTrigger(0); //Check for user trigger
        if(Trigger_Output == 1){ //Watch for pressure inflection. Trigger if if true.
          UserTrigger(1);
          Main_Setting = 1;
          Main_prevtime = CurrentTime;
        }
      }else{
        Solenoid(Power_Factor,Power_Factor);
        Main_prevtime = CurrentTime;
        UserTrigger(1); //Reset pressure inflection subroutine.
      }
    }else if(Main_Setting == 1){
      StaticForce(0); //Calculate static force at current angle for constant pressure.
      //Constant pressure mode. Maintain a constant angle, but give at a constant
pressure
        //rate. Main_Secondary_Setting
      if((NetCylinderForceAvg - StaticForceAtAngle) < Main_Secondary_Pressure_Tolerance
&&
          (StaticForceAtAngle - NetCylinderForceAvg) <
Main_Secondary_Pressure_Tolerance){
        Solenoid(0,0);
        PIDP_ITot = 0;
        if((CurrentTime - Main_prevtime) > 5000){
          Main_Setting = 0;  //Change to location positioning once pressures have
equalized.
          SetAngle = AngleAvg;  //Update target angle for location positioning.
        }
```

```
      }else{
        PIDP_ITot = 0;  //Prevent pressure integral windup.
        float PPID = PID_Pressure();
        Main_prevtime = CurrentTime;
        Solenoid(PPID+PPIDoffset,PPID+PPIDoffset);
      }
    }else if(Main_Setting == 3){
      //Diagnositic loop. Cancels all other main loops.
      //PressureTest(5,95,150);  //min target: 0; max target: 95 degrees; increments: 95

    }

  }else if(Main_User_Mode == 1){  //Secondary defined user mode
    //Main_Secondary_Pressure_Tolerance

    //First state is initialization. Move to 90 degrees, wait for settle, calc. SForce
    if(SEC_Run_State == 0){
      //Set to default settings, 90 degrees
      SetAngle = 90;
      tolerance_compare = abs(SetAngle - AngleAvg);

      //Movement
      if(tolerance_compare < tolerance){    //Deadband
        Solenoid(0,0);
        PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.

        //if stable for 3 seconds, configure static press, then leave loop.
        if((CurrentTime - Main_prevtime) > 3000){
          //Change leave this state, move to next
          SEC_Run_State = 1;
        }
      }else{
        Solenoid(Power_Factor,Power_Factor);
        Main_prevtime = CurrentTime;
      }

    //This state is running state.
    }else if(SEC_Run_State == 1){

      //Auxillary method constant force. Controller is in here.
      float Aux_P_Constant = 1.5;   //Kp constant for aux controller

      //From best fit line in excel on pressure vs angle plot for 35 lb weight.
      float Aux_Eq_Pres = -0.0000000074*pow(AngleAvg,6) + 0.0000024882*pow(AngleAvg,5)
                          - 0.0003340320*pow(AngleAvg,4) + 0.0231939949*pow(AngleAvg,3)
                          - 0.8858717096*pow(AngleAvg,2) + 22.4203933939*AngleAvg
                          - 28.1536734799;

      float Aux_diff = PEAvg - PRAvg;
      float Aux_error = Aux_Eq_Pres - Aux_diff; //Calculate controller error
      float Aux_Power = Aux_P_Constant * Aux_error; //calculate power value for control

      if(Aux_error < 15 && Aux_error > -5){
        //Stop all motion in deadband.
        Solenoid(0,0);
      }else{
        //Move accordingly
```

```
        Solenoid(Aux_Power,Aux_Power);  //move
      }
    }
  }


/*
  //Data collection, transmit data via serial
  Serial.print("DATA,TIME,");  Serial.print(millis());
  Serial.print(","); Serial.print(AngleAvg); Serial.print(","); Serial.print(SetAngle);
  Serial.print(","); Serial.print(PEAvg); Serial.print(",");
  Serial.println(PRAvg);

  //Data collection reset.
  row++;
  x++;
  if( (CurrentTime - StartTime) > 60000){
     if (row > 3000) //Collect 3000 lines of data before resetting
     {
       row=0;
       Serial.println("ROW,SET,2");
       delay(20000000); //Stop run after a complete set of data has been collected.
     }
  }
  //end data collection
*/

}
```

```
//////////////SUBROUTINES//////////////

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////Static Force////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
/

float StaticForce(int SF_choice){
  //Using equation developed in Excel, either calculate the static load or static
cylinder
    //force at the current angle.
    //Global variables: float SF_Load = 0; float StaticForceAtAngle = 0;
    //calculated applied load when static.

  //Variables
  float SF_rate = 0;
  float SF_Angle = AngleAvg;  //Read in from global
  float SF_Cylinder_Force = NetCylinderForceAvg;  //Read in from global
  float SF_Static_Force_output = 0;

  //if choice is 0, calculate static force.
  if(SF_choice == 0){
    SF_rate = SF_Load*0.05922 + 0.2172;
    SF_Static_Force_output = SF_rate * SF_Angle + 5;
    StaticForceAtAngle = SF_Static_Force_output;
    return(StaticForceAtAngle);
  }else if(SF_choice == 1){
    //if choice is 1, calculate static load. (this function should only be called while
      //the arm is static, otherwise error will result).
      //note: choice 1 must be called before choice 2, so system can calculate and write
      //the static load to global variables.
    float SF_denum = 0.05922*SF_Angle;
    float SF_num = SF_Cylinder_Force - 5;
    SF_Load = (SF_num / SF_denum) - 3.667;
    return(0);
  }

}


////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////Pressure PID Controller/////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
//PID controller that works to hold pressure constant during constant pressure
operations.
//NetCylinderForceAvg - CylinderStaticForce) > CylinderForceLimit  Retract
//CylinderStaticForce - NetCylinderForceAvg) > CylinderForceLimit

//Global Variables:
  //unsigned long PIDP_LastTime=0
  //float PIDP_DPrev = 0;
  //float PIDP_ITot = 0;

float PID_Pressure(){
  //Note: PKp, PKi, PKd are global variables

  //Get Time values
```

```
  unsigned long PIDP_CurrentTime = CurrentTime;

  //Calculate error value
  float PIDP_Error = NetCylinderForceAvg - StaticForceAtAngle;
  //Positive if leg needs to extend, negative if leg needs to retract to reach target.

  //Calculate PID components
    //Proportional
  float PIDP_Ep = PKp * PIDP_Error;

    //Derivative
  float PIDP_dT = PIDP_CurrentTime - PIDP_LastTime;
  float PIDP_dE = PIDP_Error - PIDP_DPrev;
  float PIDP_dEdT = PIDP_dE / PIDP_dT;
  float PIDP_Ed = PKd * PIDP_dEdT * 1000;

    //Integral
  float PIDP_Itime = (PIDP_CurrentTime - PIDP_LastTime);
  float PIDP_ItimeSec = PIDP_Itime / 1000;
  float PIDP_Eavg = ( PIDP_Error + PIDP_DPrev ) / 2;
  PIDP_ITot = (PIDP_Eavg * PIDP_ItimeSec) + PIDP_ITot;  //Update global total.
  float PIDP_Ei = PKi * PIDP_ITot;

    //Calculate PWM constant
  float PIDP_PID = (PIDP_Ep + PIDP_Ed + PIDP_Ei) * (-1);
//  Serial.print("PID: "); Serial.println(PIDP_PID);

    //update previous error value for next cycle.
  PIDP_DPrev = PIDP_Error;

  //Return Factor
  return(PIDP_PID);

}


////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////Trigger////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
//Detect inflection of pressures during deadband. Time inflection. If inflection time
  //greater than a specified amount, trigger result.
//Global variables required: Trigger_Start, Trigger_State, CurrentTime.

int UserTrigger(int Trigger_Choice){
  float UT_Duration_req = 3000; //Duration required for trigger.

  if(Trigger_Choice == 0){
    //Start Timer
    if(Trigger_State == 0 && (PRAvg - PRAvg_prev) > 4 && (PEAvg_prev - PEAvg) > 4){
      Trigger_Start = CurrentTime;
      Trigger_State = 1;
    }else if(Trigger_State == 1){
      if((CurrentTime - Trigger_Start) > UT_Duration_req && (PRAvg - PRAvg_prev) > 4 &&
          (PEAvg_prev - PEAvg) > 4){
        //Report True
        Trigger_State = 0;
        Trigger_Output = 1;
```

```
      }else if((PRAvg - PRAvg_prev) < 4 && (PEAvg_prev - PEAvg) < 4){
        Trigger_State = 0;
      }else{
        Trigger_Output = 0;
      }
    }
  }else if(Trigger_Choice == 1){
    //Reset Timer
    Trigger_Output = 0;
    Trigger_Start = CurrentTime;
  }
}


/////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////Update Angle v2/////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
/*Subroutine: Update Angle
 * Calling this subroutine pulls the values for the angular position sensors and using
 * the values calculated in InitializeSystem() calculates the current angle of the leg.
 * AngleHigh output is at Total_angle degrees in Zero position. AngleLow output is at 0
 * at Zero position. AngleAvg is an average of all four inputs, which is 0 at Zero
 * position, moving to Total_angle at full extension.
 *
 * This subroutine needs no data passed to it as it pulls all required data from the
 * global variables and then writes to the global variables in turn, So no data is
 * passed from the subroutine.
*/


void UpdAngle(){
  float R1 = analogRead(A0);
  float R2 = analogRead(A1);
  float L1 = analogRead(A2);
  float L2 = analogRead(A3);

  //These equations were calculated by plotting data points and calculating a best fit,
    //4th order polynomial equation in Excel. Angle position vs resistance is not a
linear
    //relationship. It is best characterized by a 4th order polynomial.
  double angleR1 = -0.000000001576468*pow(R1,4) + 0.000002144625211*pow(R1,3) -
              0.000671936010561*pow(R1,2) - 0.220259350171163*R1 + 122.894320795906000;
  double angleR2 = 0.000000000266411*pow(R2,4) - 0.000000193817164*pow(R2,3) -
              0.000331713174445*pow(R2,2) + 0.431925844770142*R2 - 46.648588452906300;
  double angleL1 = -0.000000001215649*pow(L1,4) + 0.000001736402512*pow(L1,3) -
              0.000547735672999*pow(L1,2) - 0.233726690234637*L1 + 127.786188368232000;
  double angleL2 = 0.000000000214559*pow(L2,4) - 0.000000130522079*pow(L2,3) -
              0.000349690681517*pow(L2,2) + 0.435326785484359*L2 - 47.381785062013900;

  //Check that angles are within two tolerance distances, if so, average both sets.
  if(angleR2 <= (angleL2+AngleTolerance) && angleR2 >= (angleL2-AngleTolerance)){
    AngleLow = (angleR2 + angleL2) / 2;
    if(angleR1 <= (angleL1+AngleTolerance) && angleR1 >= (angleL1-AngleTolerance)){
      AngleHigh = (angleR1 + angleL1) / 2;
    }else{
      exit;
    }
```

```
  }else{
    exit;
  }

  AngleAvg = (AngleHigh + AngleLow)/2;
}


/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////Pressure measurement////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
//Pin A4, Pext A5, Pret A6
//Array size: 10
//Calculate and keep running averages of the pressure change, hopefully will compensate
  //for the pwm's wild pressure fluctuations.

int PresUpdate(){

  //temporary variables
  float totalExt = 0;
  float totalRet = 0;

  //Read pressures
  int tempPinlet = analogRead(A4);
  int tempPext = analogRead(A5);
  int tempPret = analogRead(A6);

  if(Pinit <= 9){  //if arrays are not initialize, do so.
    PavgExt[Pinit] = tempPext;
    PavgRet[Pinit] = tempPret;
    Pinit++;
  }else{
    //update array with current values and average
    for (int x=0; x <= 9; x++){
      if(x == 9){
        PavgExt[x] = tempPext;
        totalExt = totalExt + tempPext;
        PavgRet[x] = tempPret;
        totalRet = totalRet + tempPret;
      }else{
        totalExt = totalExt + PavgExt[x];
        totalRet = totalRet + PavgRet[x];
        PavgExt[x] = PavgExt[x+1];
        PavgRet[x] = PavgRet[x+1];
      }
    }
    //Calculate average, write to global variables
    PEAvg = totalExt / 10;
    PRAvg = totalRet / 10;
  }

  //////////////////////////////////////////////////////
  //Force Calculation subsection
  float Ext_Area = 2.4052819; //1.75 bore
  float Ret_Area = 2.2089323; //1.75 bore, 0.5 rod
  float PExtPSI = 0.18310547*tempPext - 18.75; //Voltage to pressure conversion from spec
  float PRetPSI = 0.18310547*tempPret - 18.75; //Voltage to pressure conversion from spec
```

```
  float FnetCyl = (Ext_Area * PExtPSI) - (Ret_Area * PRetPSI);  //Positive force causes
                                    //the leg to extend. Negative causes it to retract.

  //Average Force Calculation subsection
  float PExtPSIavg = 0.18310547*PEAvg - 18.75; //Voltage to pressure conversion from spec
  float PRetPSIavg = 0.18310547*PRAvg - 18.75; //Voltage to pressure conversion from spec
  float FnetCylAvg = (Ext_Area * PExtPSIavg) - (Ret_Area * PRetPSIavg);  //Positive force
                              //causes the leg to extend. Negative causes it to
retract.

  //Write to globals
  NetCylinderForce = FnetCyl;
  NetCylinderForceAvg = FnetCylAvg;

  return(0);
}


////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////PWM Constant////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
//Calculates the PWM constant for use during transition operations. Incorporates pressure
//measurement and angular displacement. Note: this subroutine contains two distinct
//sections of code. The second section is in use, but variables are called from the first
//section. Memory usage should not be an issue at this point, so this can stay like it is
//for the time being. Some of the more intensive, unused function have been commented
out.

float PWM_Constant(){
  //Constant values:
  float PWM_Pagrf = 1;
  float PWM_Poffset = 0;    //Pressure offset, applied to calculate F pressure.
                              //For PRet = PExt, Fpressure = Poffset / Pinlet
  float PWM_AgrF = 2;       //Angle agression factor. Multiplier for angle difference.
                              //Higher factor, the constant will be more aggressive for
                              //smaller angle differences.
  float PWM_Aoffset = 0;    //Angle offset, applied to calculate F angle. Independent of
                              //angle measurements, has the effect of increasing Fangle
                              //by a factor of Aoffset / Total angle.
  float PWM_TAngle = 95.00; //Total angle (read from global)

  //Variables (do not declare other than 0)
  float PWM_PRet = 0; //analog input (A6)
  float PWM_PExt = 0; //analog input (A5)
  float PWM_Pin = 0;  //analog input (A4)
  float PWM_CAngle = 0; //Current angle (read from global)
  float PWM_SAngle = 0; //Set angle (read from global)
  float PWM_FPres = 0; //Pressure factor output
  float PWM_FAng = 0; //Angle factor output
  float PWM_F = 0;    //Overall Factor output

  //Read in values (from global or analog inputs)
//  PWM_PRet = analogRead(A6);
//  PWM_PExt = analogRead(A5);
//  PWM_Pin = analogRead(A4);
  PWM_CAngle = AngleAvg;  //Global
  PWM_SAngle = SetAngle;  //Global
```

```
  //Calculate FPressure (Pressure Factor)
  float PWM_Pdiff = PWM_PRet - PWM_PExt;
  float PWM_PdiffABS = abs(PWM_Pdiff);
//  PWM_FPres = ( PWM_Pagrf * ( PWM_PdiffABS + PWM_Poffset ) / PWM_Pin ) + 1;
//The +1 sets it such that the factor is 1 for diffP = 0, so then the pressures have no
effect.

  //Calculate FAngle (Angle Factor)
  float PWM_Adiff = PWM_CAngle - PWM_SAngle;
//  float PWM_AdiffABS = abs(PWM_Adiff);
//  PWM_FAng = ( ( PWM_AgrF * PWM_AdiffABS ) + PWM_Aoffset ) / PWM_TAngle;

  //Computer overall factor
//  PWM_F = PWM_FAng;  //PWM_FPres *

////////////////Alternate: PID attempt 2  (This is the current
controller///////////////
  //Note: Kp, Ki, Kd are global variables

  //Get Time values
  unsigned long PWM_CurrentTime = CurrentTime;

  //Calculate error value
  float PWM_Error = PWM_SAngle - PWM_CAngle;
  //Positive if leg needs to extend, negative if leg needs to retract to reach target.

  //Calculate PID components
    //Proportional
  float PWM_Ep = PWM_Kp * PWM_Error;

    //Derivative
  float PWM_dT = PWM_CurrentTime - PWM_LastTime;
  float PWM_dE = PWM_Error - PWM_DPrev;
  float PWM_dEdT = PWM_dE / PWM_dT;
  float PWM_Ed = PWM_Kd * PWM_dEdT * 1000;

    //Integral
  float PWM_Itime = (PWM_CurrentTime - PWM_LastTime);
  float PWM_ItimeSec = PWM_Itime / 1000;
  float PWM_Eavg = ( PWM_Error + PWM_DPrev ) / 2;
  PWM_ITot = (PWM_Eavg * PWM_ItimeSec) + PWM_ITot;  //Update global total.
  float PWM_Ei = PWM_Ki * PWM_ITot;

    //Calculate PWM constant
  float PWM_PID = PWM_Ep + PWM_Ed + PWM_Ei;

    //update previous error value for next cycle.
  PWM_LastTime = PWM_CurrentTime;
  PWM_DPrev = PWM_Error;
  PWM_F = PWM_PID;

  //Return Factor
  return(PWM_F);
}
```

```
/////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////Solenoid Control v3 (builds off of v2)//////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
//Unified solenoid control subroutine. Input values from -255 to 255 with 0 being stop.
//Positive values cause the side of the solenoid they reference to expand in a positive
//manner. So a positive value on the extend side will cause the solenoid to elongate.
//Negative values vent gasses to the atmosphere.

int Solenoid(float Extend_PWR, float Retract_PWR){

  //Conditioning
  if(Extend_PWR > 255){Extend_PWR = 255;}else if(Extend_PWR < -255){Extend_PWR = -255;}
  if(Retract_PWR > 255){Retract_PWR = 255;}else if(Extend_PWR < -255){Retract_PWR = -
255;}

  //Modify Power levels
  float Ext_PWR = SolExtPWMOffset + (255 - SolExtPWMOffset)*(abs(Extend_PWR) / 255);
  float Ret_PWR = SolRetPWMOffset + (255 - SolRetPWMOffset)*(abs(Retract_PWR) / 255);

  if(Extend_PWR == 0){
    //Turn off solenoids
    digitalWrite(SolenoidDir1,LOW);
    digitalWrite(SolenoidTrig1,LOW);
  }else if(Extend_PWR > 0){
    //Open Solenoid in positive direction
    digitalWrite(SolenoidDir1, HIGH);
    if(Extend_PWR == 255){
      digitalWrite(SolenoidTrig1, HIGH);
    }else{
      analogWrite(SolenoidTrig1, Ext_PWR);
    }
  }else if(Extend_PWR < 0){
    //Open Solenoid in negative direction
    digitalWrite(SolenoidDir1, LOW);
    if(Extend_PWR == -255){
      digitalWrite(SolenoidTrig1,HIGH);
    }else{
      analogWrite(SolenoidTrig1,Ext_PWR);
    }
  }

  if(Retract_PWR == 0){
    //Turn off Solenoids
    digitalWrite(SolenoidDir2,LOW);
    digitalWrite(SolenoidTrig2,LOW);
  }else if(Retract_PWR > 0){
    //Open Solenoid in positive direction
    digitalWrite(SolenoidDir2, LOW);
    if(Extend_PWR == 255){
      digitalWrite(SolenoidTrig2, HIGH);
    }else{
      analogWrite(SolenoidTrig2, Ret_PWR);
    }
  }else if(Retract_PWR < 0){
    //Open Solenoid in negative direction
    digitalWrite(SolenoidDir2, HIGH);
    if(Retract_PWR == -255){
```

```
      digitalWrite(SolenoidTrig2,HIGH);
    }else{
      analogWrite(SolenoidTrig2,Ret_PWR);
    }
  }
  return(0);
}


///////////////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////Pressure Test//////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////////////////
//Increment through angle values, recording data for each value to establish a plot of
//static pressure vs angle.

/*

float PressureTest(float PT_Min,float PT_Max,float PT_n){

  //variables
  float PT_req_time = 5000;
  unsigned long PT_currentrun = CurrentTime;  //Global time input.
  float PT_Angle = AngleAvg;  //Global angle input.
  float PT_cyl_force = PEAvg - PRAvg; //NetCylinderForceAvg;
  float PT_current_cyl_force;

  //Calculate step size
  float PT_range = PT_Max - PT_Min;
  float PT_step_size = PT_range / PT_n;
  float PT_current_target = (PT_step_size * PT_increment) + PT_Min;
  float PT_error = abs(PT_Angle - PT_current_target);

  //Write target to global
  SetAngle = PT_current_target;

  //Move to position and hold.
  if(PT_error < PT_tolerance){    //Deadband
//    Serial.println("DEADBAND!");
    Solenoid(0,0);
    PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.

    //If angle is within tolerance for specified time, record value and increment.
    if((PT_currentrun - PT_lastrun) > PT_req_time){
      //write pressure value
      PT_current_cyl_force = PT_cyl_force;

      //Data collection, transmit data via serial
      Serial.print("DATA,TIME,");  Serial.print(millis()); Serial.print(",");
      //Serial.print(PT_Angle); Serial.print(","); Serial.println(PT_current_cyl_force);
      Serial.print(AngleAvg); Serial.print(","); Serial.print(PEAvg); Serial.print(",");
      Serial.println(PRAvg);

      //Data collection end.
      row++;
      x++;
      if(PT_current_target >= PT_Max){
        delay(2000000); //Stop run after a complete set of data has been collected.
```

```
        return(0);
      }
      //end data collection

      //increment
      PT_lastrun = PT_currentrun;
      PT_increment++;
    }
  }else{
    Solenoid(Power_Factor,Power_Factor);
    PT_lastrun = PT_currentrun;
  }

}
*/


/////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////Bode plot//////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
//form and format bode plot over specified frequency. Calculate max and min values at
//each frequency and report to serial PLX-DAQ for plotting.
//format Bode(center_angle,maximum_amplitude,maximum_frequency,minimum_frequency,steps);
//returns an integer = 1 when finished.
//Bode(60,40,10,1,10);

/*

int Bode(float BODE_Target,float BODE_max_amplitude,float BODE_freq_max,
          float BODE_freq_min, int BODE_steps){
  //BODE_Target: Angle about which to cycle.
  //BODE_max_amplitude: Cyclic amplitude to aim for. If this value is 10, destination
    //angles will be BODE_target +/- 10/2.
  //BODE_max_frequency: Maximum frequency to cycle through. Note: higher frequncy
    //corresponds to shorter period.
  //BODE_min_frequency: Minimum frequency to cycle through.
  //BODE_steps: Number of testing points to utilize. Causes each testing requency to be
    //delta(frequncy) / steps higher than the last.

  //BODE variables
  //unsigned long BODE_Phase_Applied_Start = 0;   //Time of applied change in set point
    //for Phase Shift Measurement
  //unsigned long BODE_Phase_Measured_Trigger = 0;  //Time of measured maximum amplitude
    //for Phase Shift Measurement


  //BODE Time
  BODE_current_time = millis();

  //BODE setup
  float BODE_Bandwidth = BODE_freq_max - BODE_freq_min;
  float BODE_step_frequency = BODE_Bandwidth / BODE_steps;
  float BODE_current_frequency = BODE_freq_min + BODE_step_frequency * BODE_n;
  float BODE_current_period = 1000 / BODE_current_frequency;
  float BODE_test_duration = 1.1*BODE_current_period; //2.5 periods for test length
  //float BODE_step_bandwidth = BODE_Bandwidth / BODE_steps;
  //float BODE_step_period = 1000 / BODE_step_bandwidth;  //1000 converts seconds to ms
```

```
//Initial period
//float BODE_min_period = 1000 / BODE_freq_max; //1000 converts seconds to ms
//float BODE_current_period = BODE_min_period + BODE_step_period * BODE_n;
float BODE_current_half_period = BODE_current_period / 2;

//Target angles
float BODE_half_amplitude = BODE_max_amplitude / 2.00;
float BODE_Target_high = BODE_Target + BODE_half_amplitude;
float BODE_Target_low = BODE_Target - BODE_half_amplitude;

//Motion variables
float Power_Factor = PWM_Constant();
float tolerance = 0.25; //Degrees about target to cutoff to limit solenoid cycles.
float tolerance_compare = abs(SetAngle - AngleAvg);

//Initial Move to target angle:
if(BODE_STATE == 0){
  SetAngle = BODE_Target;
  if(tolerance_compare < tolerance && (BODE_current_time - BODE_previous_time) > 5000){
      //Deadband & delay
    Solenoid(0,0);
    PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.
    BODE_STATE = 1;
    //Once this comment is reached, leg is at target position, ready to begin testing.

    //Update clocks for test start
    BODE_current_time = millis();
    BODE_end_time = BODE_current_time + BODE_test_duration;
    BODE_previous_time = BODE_current_time;

    //set first test angle target
    SetAngle = BODE_Target_high;

    //Set max and min variables to target angle so they have a common starting point.
    BODE_max_value = BODE_Target;
    BODE_min_value = BODE_Target;

    //Reset phase shift detect
    BODE_Phase_min = BODE_Target_low;  //Write in a value that will always be lower
                                      //than current angle for time less than 1/2 period.
    BODE_Phase_trigger = 0; //variable follows the angle values, looks for inflection
    BODE_Phase_counter = 0; //Counter for phase. Causes time to be recorded after 10
                   //successive greater values. Guarantees inflection and not a hickup.

    //Start phase detection timer
    BODE_Phase_Applied_Start = BODE_current_time; //Record applied impulse start time
                                                  //for Phase Shift.

  }else if(tolerance_compare < tolerance){  //Deadband
    Solenoid(0,0);
    PWM_ITot = 0;

  }else{
    Solenoid(Power_Factor,Power_Factor);
  }
}else if(BODE_STATE == 1){  //testing section.
  //Begin testing
```

```
    BODE_current_time = millis();

    //While time is less than end time, test...test like crazy.

    //Phase shift detection.
    //Detect when minimum has been reached. Record time at inflection.
    if(BODE_Phase_trigger == 0){
      if(BODE_Phase_counter == 15){ //if 15 successive read angles have been greater.
        BODE_Phase_trigger = 1; //terminate this string of if statements
      }
      if(AngleAvg > BODE_Phase_min){  //if arm is still increasing do this
        BODE_Phase_min = AngleAvg;
        BODE_Phase_Measured_Trigger = BODE_current_time;
      }else{  //if arm is static or decreasing, do this
        BODE_Phase_counter++;
      }
    }

    //Bode positioning sequence
    if((BODE_current_time - BODE_previous_time) > BODE_current_period){
    //if elapsed time greater than 1 period, go high
      //Go high
      //Note: for period less than 2 max, this should only trigger once.
      SetAngle = BODE_Target_high;
      PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.
      if(BODE_min_value == BODE_Target){  //record first value
        BODE_min_value = AngleAvg;
      }
      BODE_previous_time = BODE_current_time; //Update previous time for next cycle
    }else if((BODE_current_time - BODE_previous_time) > BODE_current_half_period){
    //if elapsed time greater than 0.5 period, move low
      //Go low
      //Note: for period less than 1.5 max, this should only trigger once.
      SetAngle = BODE_Target_low;
      PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.
      if(BODE_max_value == BODE_Target){  //record first value
        BODE_max_value = AngleAvg;
      }

    }

    //Move to SetAngle
    tolerance_compare = abs(SetAngle - AngleAvg);
    if(tolerance_compare < tolerance){    //Deadband
      Solenoid(0,0);
      PWM_ITot = 0;  //Reset integral total to prevent PID integral windup.
    }else{
      Serial.print("SetAngle:"); Serial.println(SetAngle);
      Solenoid(Power_Factor,Power_Factor);
    }

    //Calculate max and min values over entire cycle
//    if(AngleAvg > BODE_max_value){
//      BODE_max_value = AngleAvg;
//    }
//    if(AngleAvg < BODE_min_value){
//      BODE_min_value = AngleAvg;
```

```
//    }

    if(BODE_current_time > BODE_end_time){
      BODE_STATE = 2; //cycle complete, leave loop this section
    }

//    Serial.println("Stage 1 Cycle working");

  }else if(BODE_STATE == 2){  //Testing complete, transition to next frequency.
    //calculate experimental amplitude and current frequency
    float BODE_exp_amplitude = BODE_max_value - BODE_min_value;
    float BODE_current_frequency = 1000 / BODE_current_period;

    //Calculate Phase shift
    float BODE_Delta_Phase = BODE_Phase_Measured_Trigger-BODE_Phase_Applied_Start;
    float BODE_Phase_Shift = BODE_Delta_Phase / BODE_current_period;
    float BODE_Phase_Shift_Rad = BODE_Phase_Shift * 2 * PI;


      //Post the following variables
        //BODE_exp_amplitude
        //BODE_max_amplitude
        //BODE_current_frequency
    Serial.print("DATA,TIME,");
    Serial.print(BODE_current_frequency); Serial.print(",");
    Serial.print(BODE_max_amplitude); Serial.print(",");
    Serial.print(BODE_exp_amplitude); Serial.print(",");
    Serial.print(BODE_Delta_Phase); Serial.print(",");
    Serial.println(BODE_current_period);

    //if all steps have been completed, terminate loop, otherwise increment and repeat.
    if(BODE_n > BODE_steps){
      BODE_STATE = 3;  //terminate testing by denying entry to primary loop once all
                       //steps have been processed.
      Solenoid(0,0);
      return(1);
    }else{
      BODE_STATE = 0;
      BODE_n++;
    }

//    Serial.println("Stage 2 (final) cycle complete");

  }
*/
```

```
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////Solenoid Control v2////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////


/*    Solenoid extend
Description:  Joins all functions for the extend solenoids. Offering full control
                over the extend side of the cylinder. PWM modulated for
                Intermediate power values.

input format: SolenoidExt(State,Trig,Power);

Variables:
State: Set to 1 for intake, set to 0 for exhaust
Trig: Set to 1 to activate trigger solenoid, set to 0 to disable solenoid
Power1: Power output to solenoid. 0: off, 255: fully on, in between is pulsed from
          minpulse to 0.1 second pulse


Global settings required:
SolenoidDir1 = 2; //Solenoid Extending directional pin number
SolenoidTrig1 = 10; //Solenoid Extending trigger pin number

Global variables required:
unsigned long PrevSolCycle1 = 0; DEFUNCT

Required in main loop:
CurrentTime = millis();

Required in setup:
Clock frequency change to 30 hertz for trigger pins

Returns a value of 1 upon completing execution.

*/

/*
int SolenoidExt(int State1,int Trig1, float Power1) {
  float SolPower1Divisor = 255;  //Power 1 input range accepted from 0 to
                                  //SolPower1Divisor (0 - 255)
  if(Power1 > SolPower1Divisor){ //Cap power level at stated maximum
    Power1 = SolPower1Divisor;
  }
  float SolPower1 = SolExtPWMOffset + (255 - SolExtPWMOffset)*(Power1 /
SolPower1Divisor);

  //if direction state has changed update
  if (State1 == 1) { //if state is set to intake
    digitalWrite(SolenoidDir1, HIGH);
  }else if (State1 == 0) { //if state is set to exhaust
    digitalWrite(SolenoidDir1,LOW);
  }

  if(Trig1 == 0){
    digitalWrite(SolenoidTrig1, LOW);
  }else if(Trig1 == 1){
    if(Power1 == SolPower1Divisor){
```

```
      digitalWrite(SolenoidTrig1, HIGH);
    }else if(Power1 == 0){
      digitalWrite(SolenoidTrig1, LOW);
    }else if(Power1 > SolPower1Divisor){
      digitalWrite(SolenoidTrig1, HIGH);
    }else{
      analogWrite(SolenoidTrig1, SolPower1);
    }
  }
  return(1);
}
*/

/*    Solenoid Retract
Description:  Joins all functions for the extend solenoids. Offering full control
                over the extend side of the cylinder. PWM modulated for
                Intermediate power values.

input format: SolenoidRet(State,Trig,Power);

Variables:
State: Set to 1 for intake, set to 0 for exhaust
Trig: Set to 1 to activate trigger solenoid, set to 0 to disable solenoid
Power1: Power output to solenoid. 0: off, 255: fully on, in between is pulsed from
          minpulse to 0.1 second pulse

Global settings required:
SolenoidDir2 = 5; //Solenoid Retract directional pin number
SolenoidTrig2 = 9; //Solenoid Retract trigger pin number

Global variables required:
PrevSolCycle2 = 0;  DEFUNCT

Required in main loop:
CurrentTime = millis();

Returns a value of 1 upon completing execution.

*/

/*
int SolenoidRet(int State2,int Trig2, float Power2) {
  float SolPower2Divisor = 255;  //Power 1 input range accepted from 0 to
                                   //SolPower1Divisor (0 - 255)
  if(Power2 > SolPower2Divisor){ //Cap power level at 2
    Power2 = SolPower2Divisor;
  }
  float SolPower2 = SolRetPWMOffset + (255 - SolRetPWMOffset)*(Power2 /
SolPower2Divisor);

  //if direction state has changed update
  if (State2 == 1) { //if state is set to intake
    digitalWrite(SolenoidDir2, HIGH);
  }else if (State2 == 0) { //if state is set to exhaust
    digitalWrite(SolenoidDir2,LOW);
  }
```

```
  if(Trig2 == 0){
    digitalWrite(SolenoidTrig2, LOW);
  }else if(Trig2 == 1){
    if(Power2 == SolPower2Divisor){
      digitalWrite(SolenoidTrig2, HIGH);
    }else if(Power2 == 0){
      digitalWrite(SolenoidTrig2, LOW);
    }else if(Power2 > SolPower2Divisor){
      digitalWrite(SolenoidTrig2, HIGH);
    }else{
      analogWrite(SolenoidTrig2, SolPower2);
    }
  }
  return(1);
}
*/


/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////Initialize Sensors /////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

//This Code is for the most part defunct, except for the maximum pressure measurements
//and resetting the leg to initial positions. It executes at the start of the program,
//so it should have little effect on the speed of the main program.

/*
float InitializeSystem(){
  //Charge the cylinder.
  SolenoidExt(1,1,255);
  SolenoidRet(1,1,255);
  delay(1500);  //Pause for fill
  SolenoidExt(1,1,255);
  SolenoidRet(0,1,255);
  delay(4000);  //Pause for leg to extend

  //Grab extended sensor values
  float ExtRA1 = analogRead(A0);
  float ExtRA2 = analogRead(A1);
  float ExtLA1 = analogRead(A2);
  float ExtLA2 = analogRead(A3);
  float PmaxExt = analogRead(A5);
  float Pinlet = analogRead(A4);

  //Charge the cylinder
  SolenoidExt(1,1,255);
  SolenoidRet(1,1,255);
  delay(200);  //Pause for fill
  SolenoidExt(0,1,255);
  SolenoidRet(1,1,255);
  delay(3500);  //Pause for leg to retract
  SolenoidExt(0,1,255);
  SolenoidRet(0,1,255);
  delay(2000);  //Drain

  //Grab retracted values
  float RetRA1 = analogRead(A0);
```

```
  float RetRA2 = analogRead(A1);
  float RetLA1 = analogRead(A2);
  float RetLA2 = analogRead(A3);
  float PmaxRet = analogRead(A6);

  //Calculate calibration constants
  float RA1diff = RetRA1 - ExtRA1;
  float RA2diff = ExtRA2 - RetRA2;
  float LA1diff = RetLA1 - ExtLA1;
  float LA2diff = ExtLA2 - RetLA2;

  //Write to Globals
  CAL_RA1offset = ExtRA1;
  CAL_RA2offset = RetRA2;
  CAL_LA1offset = ExtLA1;
  CAL_LA2offset = RetLA2;
  CAL_RA1 = Total_angle / RA1diff;
  CAL_RA2 = Total_angle / RA2diff;
  CAL_LA1 = Total_angle / LA1diff;
  CAL_LA2 = Total_angle / LA2diff;
  CAL_Pin = Pinlet;
  CAL_Pext = PmaxExt;
  CAL_Pret = PmaxRet;

  //Reset Solenoids
  SolenoidExt(0,0,255);
  SolenoidRet(0,0,255);

  return(1);
}
*/


/////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////Tap detect  (unfinished)//////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
//This routine requires a more advanced filtration algorithm to filter
//the effects of PWM modulation on the air stream. Otherwise, this algorithm
//generates too many false triggers to be useful.
//Searching for the proverbial needle in a haystack.
/*

int TapTap(int TAP_status){
  //Look for a double tap on the arm within a time period with an amplitude between
  //the specified limits. Result: Too sensitive. Has too many false triggers due to
  //transient pressure variations.

  //Reset during movement operations
  if(TAP_status == 1){
    //rewrite arrays with last value. This prevents multiple detections from a single
    //event.
    //It will also delay another detection for atleast 30 cycles.
    for (int y=0; y <= 100; y++){
      Pavg[0][y] = PEAvg;
    }
    for (int z=0; z <= 100; z++){
      Pavg[1][z] = PRAvg;
```

```
    }
    return(true);
  }
  //else continue

  //User variables
  float TAP_Ethreshold = 5;   //Spike offset on Extend line that must occur to trigger.
  float TAP_Rthreshold = 5;   //Spike offset on Retract line that must occur to trigger.
  float TAP_Emax = 8;
  float TAP_Rmax = 8;
  float TAP_Zthreshold = 0.5;   //Distance that a value must be within average for it
                                  //to be considered equal to average. +/- value.
  float TAP_time_separation = 1;  //Time between high points to trigger double tap.

  //Variables
  float TAP_PEAvg_total = 0;
  float TAP_PRAvg_total = 0;
  int TAP_first_tap = 0;
  int TAP_rebound = 0;
  int TAP_second_tap = 0;

  if(TTfirstrun == 0 && (CurrentTime - StartTime) > 2000){ //if first run true and 2
                              //seconds has elapsed (to allow transients to subside)
    //initialize arrays with first value
    for (int y=0; y <= 100; y++){
      Pavg[0][y] = PEAvg;
    }
    for (int z=0; z <= 100; z++){
      Pavg[1][z] = PRAvg;
    }
    TTfirstrun = 1;
  }else if(TTfirstrun == 1){
    //Handle matrix as normal, cycling entries downwards. Placing newest entry at the
    //end. Calculate average for each array. This will be used as a baseline to
    //compare odd entries.
    for (int y=0; y <= 100; y++){
      if(y == 100){
        Pavg[0][100] = PEAvg;
        TAP_PEAvg_total = TAP_PEAvg_total + PEAvg;
      }else{
        Pavg[0][y] = Pavg[0][y+1];
        TAP_PEAvg_total = TAP_PEAvg_total + Pavg[0][y+1];
      }
    }
    for (int z=0; z <= 100; z++){
      if(z == 100){
        Pavg[1][100] = PRAvg;
        TAP_PRAvg_total = TAP_PRAvg_total + PRAvg;
      }else{
        Pavg[1][z] = Pavg[1][z+1];
        TAP_PRAvg_total = TAP_PRAvg_total + Pavg[1][z+1];
      }
    }

    //Calculate Averages
    TAP_PEAvg_total = TAP_PEAvg_total / 101;
    TAP_PRAvg_total = TAP_PRAvg_total / 101;
```

```
    //Tap Detect
    for (int x=0; x <= 100; x++){
      float TAP_PEdiffabs = abs(Pavg[0][x] - TAP_PEAvg_total);
      float TAP_PRdiffabs = abs(Pavg[1][x] - TAP_PRAvg_total);
      if(TAP_PEdiffabs >= TAP_Ethreshold && TAP_PRdiffabs >= TAP_Rthreshold &&
          TAP_PEdiffabs < TAP_Emax && TAP_PRdiffabs < TAP_Rmax && TAP_first_tap == 0)
      { //if both pressure differences exceed threshold on same position in hash.
        TAP_first_tap = 1;  //first tap detected
      }else if(TAP_first_tap == 1){
      //if first tap has happened, seach for rebound between taps
        if((TAP_PEdiffabs - TAP_PEAvg_total) <= TAP_Zthreshold  &&
            (TAP_PRdiffabs - TAP_PRAvg_total) <= TAP_Zthreshold &&
            TAP_PEdiffabs < TAP_Emax && TAP_PRdiffabs < TAP_Rmax)
        {
          //Rebound has occured. Search for second tap.
          if(TAP_PEdiffabs >= TAP_Ethreshold && TAP_PRdiffabs >= TAP_Rthreshold){
            TAP_second_tap = 1;   //Second tap detected.
          }
        }

      }
    }

    if(TAP_second_tap == 1){
      //rewrite arrays with last value. This prevents multiple detections from a
      //single event.
      //It will also delay another detection for atleast 30 cycles.
      for (int y=0; y <= 100; y++){
        Pavg[0][y] = PEAvg;
      }
      for (int z=0; z <= 100; z++){
        Pavg[1][z] = PRAvg;
      }

      //return boolean true
      return(true);
    }else{
      return(false);
    }
  }
}
*/
```