Introduction

This write up describes the working of estimation module (about 1000 lines of C code) in the main brain currently on the Ranger (March 2012). It exists as an interface between the sensors and control module and these three talk to each other by the global variables (data ID's / CAN ID's). Input to the estimator is the sensor data and output is an estimate of the state.

The sensors include 7 angle encoders: at the hip joint, outer-left ankle, outer-right ankle, right-inner ankle (left-inner ankle is rigidly connected to right one), one in the hip motor and remaining in the inner and outer ankle motors .There are contact sensors on the four feet which tell if that foot is touching the ground or not. Finally there is the Inertial Measurement Unit (IMU) mounted on outer leg (it provides absolute angles, angular velocities and if needed, the acceleration along its three axes. It also provides, if needed, the inclination with respect to the earth's magnetic field).

The robotic state in its 2D-bipedal model for motion consists of 3 joint angles (hip, inner and outer ankles), their derivatives, absolute angle of the outer leg with respect to gravity (which combined with relative joint angles gives the absolute angles of other parts) and its derivative. Besides these 'continuous' states there is the discrete state telling which leg is swinging and which is the stance, this state transitions when the foot hits the ground (heel strike event). It is 1 when the inner leg is swinging and 0 when the outer leg is swinging. The motor currents and voltages (as measured by respective sensors), stretch in the leg springs, slope of the ground, an estimation of wind or other external disturbances etc can be part of the state too but are not considered here.

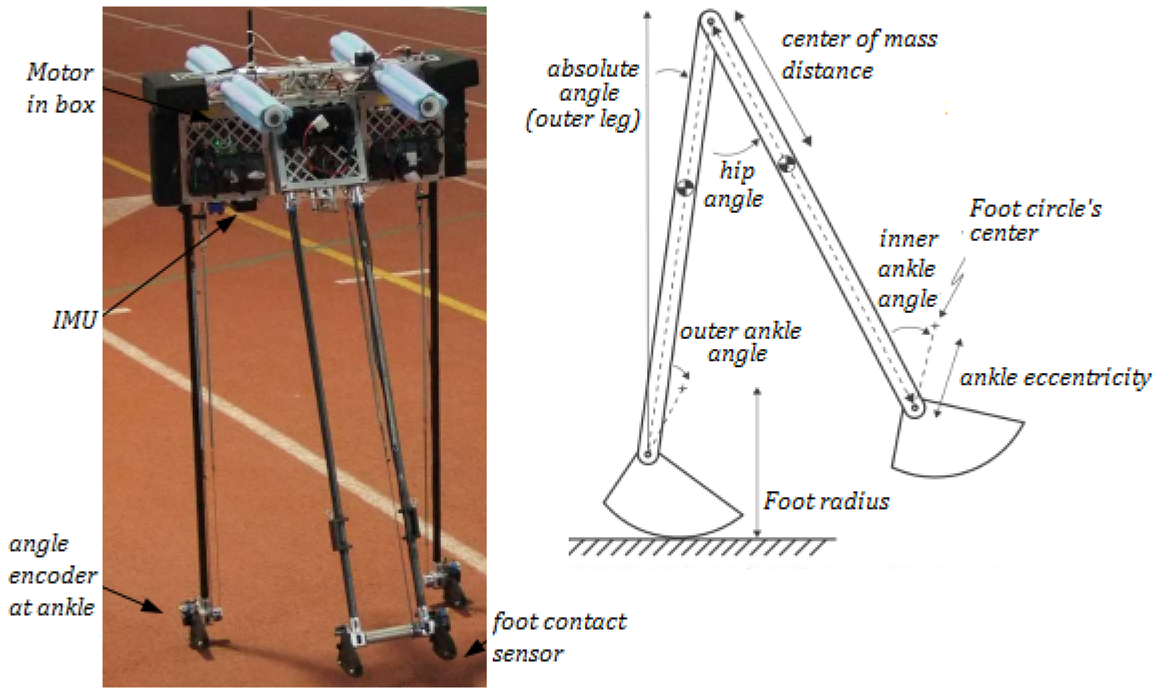

Figure 1: Ranger and some of its its sensors and state variables.

Angle sign conventions as used in the estimator are as follows:

1. Hip angle is angle between two legs and is positive when inner leg is in front (assuming robot is walking forward).

2. Ankle angle is angle between leg-line and the line joining ankle joint to the centre of the foot circle (feet are part of a circle). It is positive when ankle is infront of leg (e.g. when foot is flipped down). This angle is at about 96.2 degree offset from the angle given by encoder.

3. All absolute angles e.g. the legs, feet etc are measured from vertical and are positive if the line is slanting forward. So if the direction of forward motion is x, and y is opposite to gravity, then all absolute angles are positive if they are measured clockwise from y axis.

4. Angle rates are just derivatives of angles. Rate given by the IMU as it is currently mounted is negative according to this convention.

Moreover, the state estimator, in its current form is not doing 'model-based-estimation', by which I mean that the estimator doesn't use the information in the differential equations describing the motion of the robot to enhance its prediction ability as in for example a model-based observer or Kalman filter. Any vestiges of code implementing a simple model based observer are currently commented out.

<u>Estimation module</u>

Every 2ms the scheduler in the main brain calls the function mb_estimator_update( ), the main function in the file estimator.c, which further performs the operations as below by calling the appropriate functions.

1. **Heel strike detection and state machine**

   Various angle calculations and control actions need to know which leg (inner or outer) is on the ground and which is swinging (these are the 2 states of the simplistic state machine). Updating this state calls for detecting the instant when the foot hits the ground (heel strike). The contact sensors on the four feet give an output which roughly corresponds to the force applied to the base of that foot, as long as the output is not saturated (it saturates at 8000 units, corresponding to 5V analog output from the sensor). When the foot is in the air there is some non-zero output which remains relatively constant, and at the instant of heel strike this output surges up. Following psuedo-code shows the algorithm used to detect the heel strikes (here shown for the inner leg):

   (a) As the hip angle goes greater than a small number (0.005 rad), i.e. inner leg is in front, start looking at the inner-feet contact sensors to get an 'average' value for each. A proxy for average value is the output of a linear first order filter which is running for each foot sensor. The filter action is given by an expression:

   $$\text{filtered\_data} = 0.9(\text{previous\_filtered\_data}) + 0.1(\text{raw\_sensor\_data}).$$

$$TF(z) = \frac{0.1}{1 - 0.9z^{-1}}$$

The filter is initialized with the current raw_sensor_value.

(b) As the ankle angle sensor's reading goes greater than a particular threshold angle (currently 0.8 radian), this means the foot is positioning itself for the heel-strike and it is time to start looking for that event. The filtered_value at this instant is taken as 'average' value of contact sensor in the air (called baseline) and code starts checking if the (raw_sensor_value) > baseline + constant threshold. Threshold is 2000 units of sensor output and is chosen based on experiments.

(c) if the sensor value remains above the baseline+threshold for more than 2ms, for either of the inner feet, the heel-strike for inner leg is registered and outer-leg is labeled as the 'swinging leg'. The output of a particular feet is NOT considered in making this decision if its baseline is higher than a particular limit (5000 units), because in that case the sensor in that particular foot is considered 'stuck/unreliable'. Note that the double-stance-phase is not explicitly detected because the calculations running on ranger as of now, do not need it.

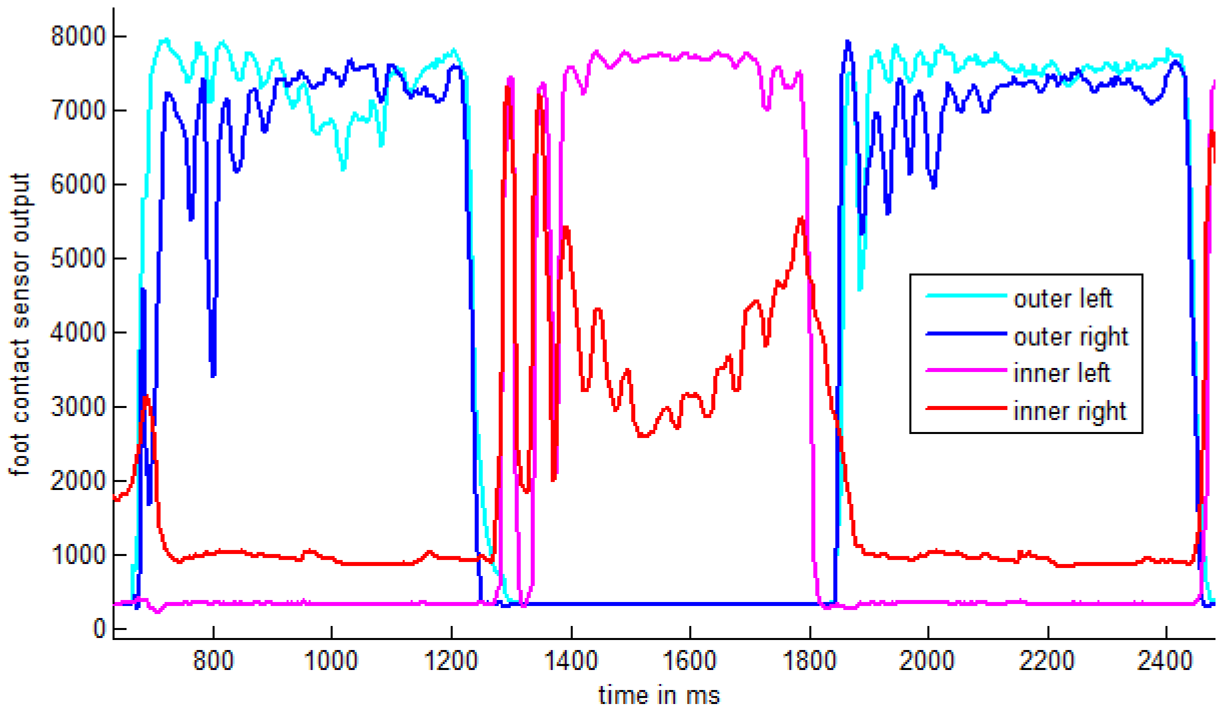

Figure 2: The four foot sensor outputs as the robot takes three steps. Notice how the sensor values shoot up as the heel strike happens, and remain high while the feet are on the ground. The initial fluctuations at the heel strike are due to bouncing/rocking of the feet upon collision . Also notice how the sensor corresponding to inner right foot is not behaving at its best: its baseline is higher and like other feet its output doesn't remain high even though the foot is in contact with ground.

The average heel strike detection delay (time difference between an actual heel-strike and its software detection) is 14 ± 3 ms. This delay could be reduced if the threshold is decreased (or the 2ms limit is decreased), but in that case the reliability of the code goes down, e.g. a spurious noise or a small bump can cause mis-detection.

2. **Filters**

   The sensor data from various joint angle sensors is fairly accurate (e.g. 0.0008 rad error for hip angle, and 0.001 for ankle angles) but the angular rate data which is computed numerically, is expectedly noisy. Estimation module filters the hip_angle_rate for both the hip joint angle and hip motor angle. It also filters the imu_rate which is the output from the inertial measurement unit (MicroStrain 3DM-GX3-25) and corresponds to the angular velocity of the outer leg with respect to the ground.

   A second order Butterworth filter with a cut off frequency of 10Hz is used for each of these. Butterworth filter was chosen because of their maximally flat characteristics in the pass and stop band, and cut-off of 10Hz was chosen by looking at the rough frequency of the prevalent noise (fluctuations) in the data. The walking frequency and hence the signal's base frequency is about 0.8 Hz (corresponding to a cycle of two steps of about 600ms each). The filter expression is as below:

$$\begin{aligned}\text{filtered\_data} = {} & b_0 * \text{raw\_data} \\ & + b_1 * \text{previous\_raw\_data} + b_2 * \text{previous\_to\_previous\_raw\_data} \\ & - a_1 * \text{previous\_filtered\_data} - a_2 * \text{previous\_to\_previous\_filtered\_data}\end{aligned}$$

$$TF(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

   Where:

$$\begin{aligned} a_1 &= -1.9112 \\ a_2 &= 0.9150 \\ b_0 &= 0.00094469 \\ b_1 &= 0.00188938 \\ b_2 &= 0.00094469 \end{aligned}$$

   SAMPLING ISSUE: The filter programs are called every 2ms in the main brain, but the sensor data received in main brain is not always at the sampling time of 2ms. Due to the workings of CAN transmission sometimes the data units can be 1ms apart and sometimes 3ms (or even more) as evidenced by their time stamps.

   To adjust to this issue the filter sampling time (i.e. time difference between 'data' and 'previous_data') is taken to be 1ms which is the least count for the time stamps. Filters are internally running at 1ms interval even though the code is called every 2ms. This is accomplished by running the filters N times whenever the filter is called. N is

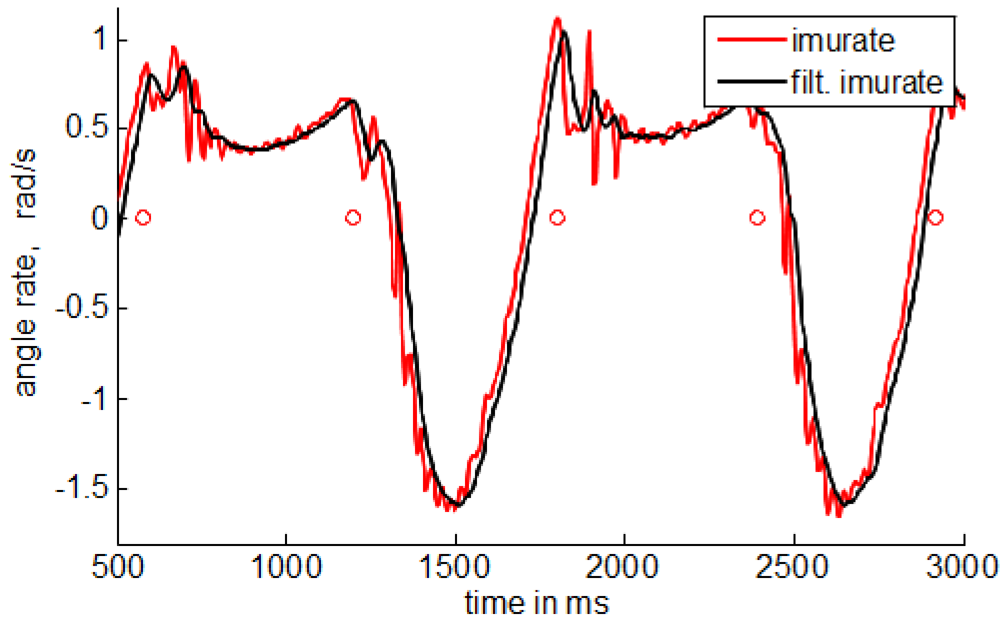

Figure 3: Filtering the imu-rate i.e. angular velocity of stance leg using the 2^{nd} order Butterworth filter.

difference in time-stamps of the data in the current run and the previous run. Missing data is assumed by the zero order hold on the data from the last function call.

The filtering process introduces a phase-lag (delay) between the raw and the filtered signal. Normally for a filter of given order the delay and 'smoothing-ability' work against each other. The delay for the current filters is about 20ms. The filters were designed in MATLAB by manually judging an optimal delay Vs. smoothing ability on a test sample of data. Also note that the hip-joint-angle-rate data received in the main-brain is already filtered in the satellite using an first order filter and hence there is some additional delay of about 5ms.

3. **Absolute angle and its rate**

One very important piece of information is the angle of robot's leg with respect to the ground frame (gravity vector). For example the state of robot at the instant when the robot is vertical (aligned with gravity) is an important indicator of speed and energy content of the robot which are useful in making control decisions.

Angular rate. The rate gyros of the IMU primarily measure the 3 components of angular velocity of the leg on which it is mounted. Because it is mounted with its X-axis along the hip joint's axis, IMU's x-rate-channel is directly the rate of change of absolute angle of the outer leg in its 2-D motion. Currently that is the only data channel from IMU which is being used in Ranger. IMU is set to give data rate at 500Hz, though it can go higher but that's not needed because 500Hz (2ms) is also the frequency at which estimator is being run.

Errors in IMU rate come primarily from the 1) bias which drifts with time 2) the random high frequency noise in angular velocity (deviation 0.0041 rad/s) 3) mounting errors and gyro nonlinearities 4) sensing delay (about 3-4ms). (See the specifications at `http://www.microstrain.com/inertial/3DM-GX3-25`).

Except for the first 2 other errors are neglected. Gyro bias was measured to be -0.0043 rad/s on an average in the static tests ranging over 2 hours. The bias is subtracted from the rate and then it is filtered using the previously described filters which smoothen out the high frequency noise and also to some extent the noise due to mechanical fluctuations.

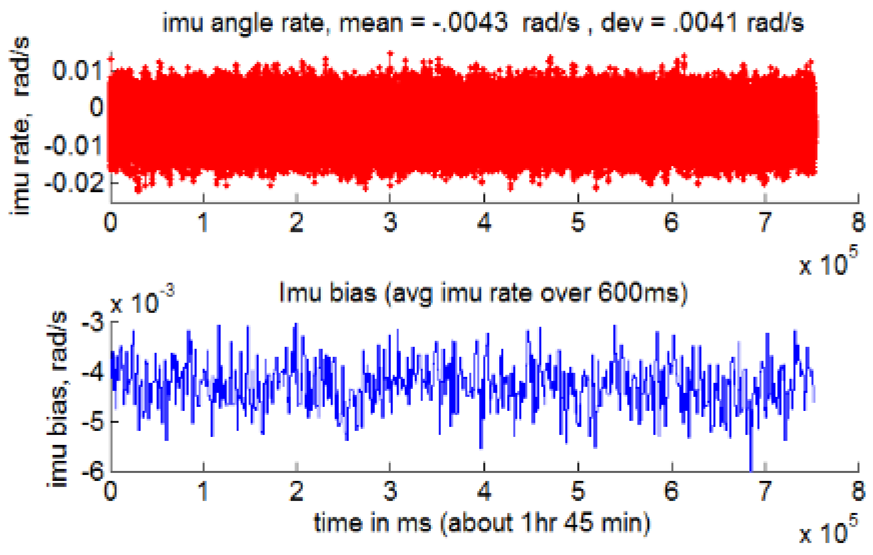

Figure 4: Measurement of the gyro-bias and random signal intensity by observing the imu's angular rate output when imu is sitting still.

Angle of the outer leg. IMU provides an estimate of the absolute angle of the outer leg, with respect to gravity, using a complementary filter which uses the rate data as well as the accelerometer data. This quantity is not very accurate however, with an accuracy of 2 degrees in dynamic (cyclic) test conditions. Moreover the general algorithm and the filters used to produce this angle may not be optimized for the specific motion which ranger has. If we obtained the angle by integrating the gyro rate in the estimator code, it will suffer from accumulation of errors typical due to integration in this type of dead reckoning.

IMU however is not the only source of absolute angles. When both inner and outer feet of the robot are on ground, the relative joint angles alone in this situation can be used to find the absolute angles, with respect to ground. (NOT with respect to gravity. The two are different if the ground is having a slope or is rough/wavy).

SENSOR FUSION: Currently in the estimator module, these sources are fused together to get a better update of the imu bias. The gyro rate is integrated using the mid-point rule (basically a running-trapezoid rule for numerical integration), after a fixed bias is subtracted from it. Since integration is a low pass filter, it takes care of the zero-mean high frequency noise in the rate data. At the instant of heel strike the relative angles (inner & outer ankle angles and hip angles) and the robot dimensions (leg lengths, inner and outer foot radii & distance between ankle joint and foot center) are used to get the absolute angle of the outer leg assuming the ground has zero slope. The calculation is a bit involved and uses various functions like cosine, sine, sine inverse, tangent inverse and square root. These are not called from the standard C library, rather their code, based on quadratic interpolation, is included in the estimator module. This code (courtesy of Gregg Steisberg) is optimized for speed and computational load. Finally once every heel strike the outer-leg's absolute angle is updated as follows:

$$\text{Absolute\_angle} = 0.9*\text{Angle\_by\_integration} + (1 - 0.9)*\text{Angle\_by\_geometry}$$

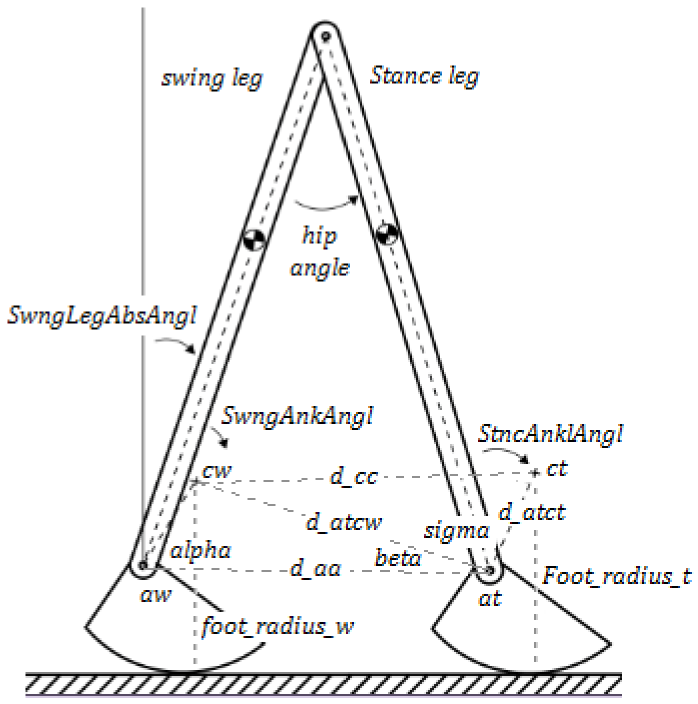

Figure 5: Absolute angles wrt vertical can be calculated from the relative joint angles when the four feet are on the ground, e.g. at the instant of heelstrike. The figure shows various variables used in the code to calculate these angles. The basic convention is a for ankle, c for center, t for stance, w for swing and d for distance. Intermediate angles 'alpha', 'beta' and 'sigma' are shown too.

The coefficient 0.9 was decided by doing a few test runs with ranger walking many steps on a roughly flat ground with no compensation by geometrical calculation. The geometric calculations are still running once per step and are read using a data-id through labView program. Finally in MATLAB it was checked which coefficient for combining the integration and geometry will produce the least deviation (not necessarily the least error) in the swing foot height just before the instant of heel strike. (Foot height is proxy-ed by the vertical distance between the two ankles). The idea is that when the foot is about to hit the ground the swing foot's height above the ground is supposed to go to zero and any discrepancy is due to various errors, primarily due to absolute angle not being reset.

The main point of doing this periodic reset/update is to provide partial compensation for integration errors and errors in gyro rate (most prominently the errors due to bias fluctuations). This method has presumed zero slope of the ground and will not work on a ground with non-zero average slope.

INITIALIZATION: When the robot starts the absolute angle is initialized to zero. This will work fine if according to the angle convention the robot is started with outer leg in the vertical position. This however is not very reliable, hence at the instant of the very first heel strike the absolute angle is set entirely to the value calculated by the joint angles. From next heel strike onwards the angle update happens as mentioned before.

4. **Heel strike prediction** Ranger's walk is powered by two main mechanisms: hip motor swinging the leg and ankle motor making the foot push the ground. It can be shown that for a simple model of the robot, it is energetically more efficient if the foot is pushing the ground just before the heelstrike. (e.g. section 7.2 in `http://ruina.tam.cornell.edu/research/topics/locomotion_and_robotics/why_walk_and_run/collisional_model.pdf`). Doing this 'prepush' requires an ability to predict when the heel strike is about to happen. One way to predict that is to monitor the height of the swinging foot and see when it approaches zero, because ideally when a foot hits the flat ground the height of it lowest point is zero (though in reality there may be offsets due to sensor errors and delays). Ankle motor on the stance leg can be triggered to prepush when the swing foot height goes below a particular threshold.

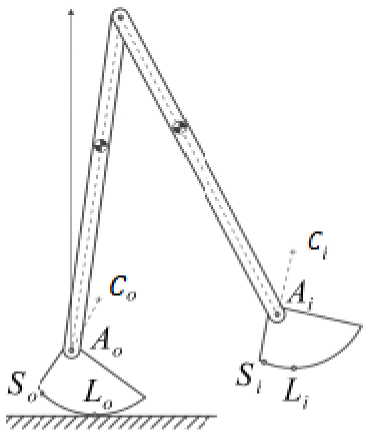

Figure 6: Various candidates for the swing foot height. L is the lowest point on the foot, considering it as part of a circle with center C, A is the ankle joint and S is the 'sweet spot', the point directly below the ankle joint, which is touching the ground when the robot is standing vertically balanced in an unstable equilibrium. SAC is a straight line.

The swing foot height can be calculated geometrically using the dimensions of the robot, the joint angles and the absolute angle of the stance leg. There are many candidates for what to chose as the swing foot height as shown in the figure 6. Ranger currently calculates the following types.

(a) Lowest point's height. (vertical component of L_oL_i) This makes the most sense. But there are some problems with it while using. Because the robot flips its foot up while swinging the leg, the lowest point on the imaginary foot circle passes through the ground and sometimes barely comes above the ground or just grazes it before it (and hence the foot) hits the ground. This makes the triggering based on a threshold difficult.

(b) Sweet spot's height. (vertical component of L_oA_i) As explained in the figure 6 caption, robot has the tendency to settle down to the sweet spot. During the stance phase the foot is almost in the 'sweet spot position' because it takes minimal ankle torque to maintain that position. When the swinging foot is about to hit the ground the controller tries to position it so that it hits the ground at the sweet spot, because it will require minimal torque from ankle motor to hold that position at the collision. With this anticipation, one can monitor the height of sweet spot point. Because it is a material point on the foot, it never goes below the ground and the height approaches zero from the top as the heel strike is about to happen. Figure 7 shows the behavior of this measure of foot's height and triggers based on that for both the inner and outer feet. Controller needed some adjustments and in one of the test runs of 143 steps (in Barton Hall) the average time to heel strike (prepush time) was 31 ± 16 ms (ideally desired to be about 20ms, based on simulations by Pranav.)

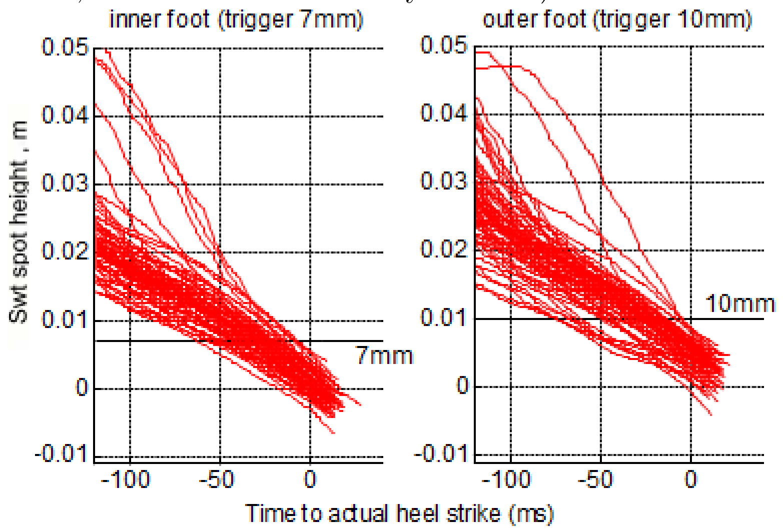

Figure 7: Figure showing the sweet spot's height as a function of time. 0ms is instant of actual heel strike. Each of the curve in the red bundle represents one step, and the curve stops at the point when the heel strike is 'detected' in the software. Average distance of the curve's endpoint from 0ms shows the delay in heel strike detection, and the fact that the curves still are decreasing after 0ms is explainable possibly by the delay in sensors and prominently the delay after which the effect of heel strike is felt on the IMU (and hence the stance-leg's absolute angle) due to flexural plasticity of the leg. (Absolute angle is important in computing the foot height). The trigger lines are shown at 7mm and 10mm. The width of curve-bundle along the trigger lines represents the spread in pre-push time, which ideally should be minimum.

(c) Virtual sweet spot's height. (vertical component of L_oA_i) However relying on that fact that controller behaves perfectly and foot will always be positioned in the sweet spot position is not very reasonable. Next possible change is to 'assume' that the foot is in sweet spot position, even though it might not be, and the track the height of a virtual sweet spot that way. This is basically height of the ankle L_oA_i minus S_iA_i (which is fixed number 5.4cm).

(d) Angle to ankle height. (vertical component of A_oA_i). Closely related to the above measure is the ankle to ankle height. Given that stance foot is closely in the sweet spot position, hence the height $L_oA_o \simeq S_oA_o = 5.4$cm. Hence the ankle to ankle height becomes similar to the virtual sweet spot's height, with a difference that it is much simpler to calculate. This measure was finally used in the Ranger's 1.5 marathon walk and is currently being used in the ranger.

To make the tuning of triggers efficient, the triggers for inner and outer feet are defined as parameters in the controller which can be input through the LabView. Currently the trigger for inner foot is 10mm and outer 12mm, the prepush times were between 30 to 50ms. This measure has proven to be reliable in that during the marathon walk, there were only few cases of late/early prepush per 200m lap in Barton Hall, usually localized to some particular spots on the track.

5. **Miscellaneous**

Besdies the above main functions there are smaller functions which are called to do various operations like

- counting the number of steps.
- finding the step time (i.e. time between consecutive heel strikes in milliseconds) and time since last heel strike.
- calculating the distance traveled by the robot by dead reckoning i.e. summing the distance traveled during each step. Distance for each step is calculated at the instant of heel strike by adding the distance between ankles (obtained from joint angles and dimensions) plus the distance traveled by rolling (assuming no slipping) of the stance foot since the last heel strike. The distance measurements had an error of about 1 percent as measured on the lab floor.
- Various variables like those pertaining to the leg-state and absolute angle calculations are reset every time reset button is pressed on the UI-board.
- As mentioned before there are in-situ functions: cosine, sine, sine inverse, tangent inverse and square root. These are called during various calculations and are provided through courtesy of Gregg Stiesberg.

C- Code: mb_estimator_update.c

```
/* To READ code BETTER in Kyle: click on all '-' signs, on left, hence
    shrinking all blocks of code */

#include <mb_includes.h>
// a file that includes all the other .h files from all other codes.
// This is a convenient way to manage the large number of headers, which might
     be changing all the time.

/* ////////////////////////////////////////////////////
// //////// sign conventions and notations  //////////
// //////////////////////////////////////////////////// */

/* /1) hip angle is the angle between two legs and is positive when inner leg
    is in front (assuming robot is walking forwards)
//2) ankle angle is angle between leg-line and the line joining the ankle
    joint to the centre of foot-circle (the foot is part of a circle).
//    It is positive when ankle is infront of leg (i.e. eg when foot isflipped
    down)
//    The ankle angle can-ids measure the physical angle of encoder which is
    zero when foot is (almost fully) flipped up, so we have
//    ankle_angle = measured from can id - ankle_angle_offset (defined as
    parameter = 96.2 deg)
//3) all absolute angles (imu or legs, feet) are measured from vertical and
    are positive if the line making the angle is slanting forwards.
//    So if the direction of the robot's forward motion is x, and y is
    oppositve to gravity, then all abolute angles are measured from y with
    clockwise as positive.
//4) angles rate are just angle dots. imu-rate as given by the imu in its
    current state of mounting on outer leg is negative in my convention.
//    Hence I put a negative sign whenever I read that CAN-ID.
//5) In the code below: L = leg, F = foot, I = inner, O = outer, HS = heel
    strike = hs,  ang = angle, abs = absolute (or magnitude),
//    prev = previous (variable from last time the function was called), curr =
     current (variable updated during the current function call)
//    stnc = t = stance,  swng = w = swing, L = length, d = distance, a = ankle
     joint,  c = center (of the foot circle)
//    E = estimation,  H = hip


// THE MAIN FUNCTION of this file which calls all other functions here is
    mb_estimator_update. It inturn is called by the scheduler.
// This code assumes at many places that the main brain runs the code every 2
    ms (eg in the counters in leg state machine)
// This code assumes that the robot walks in the forward motion.
// !!!!!!!!!!!!! PLEASE REGULARLY CHECK and update the gyro-bias variable, it
    changes with time !!!!!!!!!!!!!!!!!!!

/* ////////////////////////////////////////////////////
//////////global variables and parameters ////////////
//////////////////////////////////////////////////// */
```

```
static int prev_ui_fsm_state = 0; // the can ID ID_FSM_RESET changes to 2 when
     the robot goes to walking state, after for example the reset button was
    pressed.
static int curr_ui_fsm_state = 0; // I need to reset some variables when the
    robot starts a new segment of its walk. Resetting happens in
    mb_estimator_update

static float current_LO_absang_by_integration; // LO absang estimated by
    integrating the imu-rate 'manually'.
static float prev_LO_absang_by_integration = 0; // angle from last time the
    code was run
static float prev_StncFootAbsAngl=0;  // stance foot's previous abs angle (
    used to compute how much distance the robot moved due to rolling of its
    foot which is on the ground) (no sliding assumed)

//These static variable for:  mb_leg_state_machine_old(), but many are still
    in use.
const unsigned static int  above_ground = 0 , on_ground = 1 ;
const unsigned static int  inner_leg_swings =1 , outer_leg_swings =3; //
    double_stance= 2;// state for the robot leg
static volatile unsigned int est_leg_state_old_version = inner_leg_swings ,
    est_prev_leg_state = inner_leg_swings; // state for the robot leg.

//used by leg machine new
static int fi_left_state = above_ground, fi_right_state= above_ground,
    fo_left_state= above_ground, fo_right_state= above_ground; //individual
    foot state
static int inner_sense_reset = 0,      outer_sense_reset = 0;      //flags which
     tell that it's time to reset the filter which computes the average sensor
     values from the foot contact sensors
static int inner_take_baseline = 0, outer_take_baseline = 0; //flags which
    tell that it's time to store the average value of above filters. When the
    foot hits the ground that instant is detected as the sensor value going a
    particular threshhold above this average.
static unsigned int counter1=0, counter2=0; // counter for counting how many
    units (unit = 2ms = main brain freq) has the contact sensor's value been
    above the threshold.

//Some flags and counters
static volatile unsigned int reset_integration_flag = 0,
    reset_time_after_HS_flag = 0 ; // flags telling that HW has happened and
    its time to reset the imu angle based on joint angle, and its time to
    reset the counter which tells the time since the last HS
static int step_counter = 0;            // this is updated when the time since
     last HS is set.
static float  distance_travelled = 0;    // it is updated where the integration
     is reset.

//various geometry parametes
static const float ankle_flip_down_threshold = 0.8 ;  // if can id for ankle
    sesor (not the ankle angle, in my convention) gets larger than this it
    means foot has flipped down
static const float foot_fo_absang_calibration_constant = 1.75; //By Pranav:
    approx gives ankle_angle +/- hip/2, + for outer and - for inner
```

```
static const float foot_fi_absang_calibration_constant = 1.75; //By Pranav:
    approx gives ankle_angle +/- hip/2, + for outer and - for inner

static const float L_leg = 0.960; //leg length (hip joint to ankle joint)

static const float d_ankle2sweetspot = .054; // distance between ankle joint
    and the sweet spot in the foot, this distance I trust within +- 0.0005
    meters
static const float foot_radius_outer  = 0.168; // foot radius 20 // this is
    dubious because it's hard to measure its from 15-22cm,
static const float d_ac_outer = 0.168-.054; //d_ac=ankle length = distance
    from ankle_joint to foot_circle's center.  //its being written as
    FootRadius-d_ankle2sweetspot, I TRUST the ankle2sweetspot (0.054 m) MORE
static const float foot_radius_inner  = 0.23; // foot radius 20 // this is
    dubious because it's hard to measure its from 15-22cm,
static const float d_ac_inner = 0.23-.054; // d_ac average for the two inner
    feet.

static const float ankle_angle_offset = 1.68;// 96.2 deg  // ankle_angle =
    value measured from can id - offset (ankle angle is between leg and the
    ankle-joint-2-foot-centre line, foot being thought as a big circle). add
    pi to this ankle_angle and you get the angle to sweet spot's angle

// for imu
static float gyro_bias =-.004; // new imu  //earlier imu 0.0165 ;
//  // this is the average imu rate read from can id when the imu is supposed
    to be at rest.
// hence this value will be subtracted from the can-id to correct for thsi
    offset.
// THIS NUMBER SHOUKD BE REGULARLY CHECKED AND UPDATED. it changes with time.

/* ////////////////////////////////////////////////////
/////////////// various functions ///////////////////
/////////////////////////////////////////////////// */

void mb_imu_rate_filter (void)  {
/*/ second order butter worth filter
// Read the documentation for the working of the filters. They are basically
    operating at the 1ms sampling time even though main brain
// runs every 2ms, because the least count of time stamp is 1, the missing
    data is assumed to same as the previous data (this is called zero order
    hold).
// input to the filter is data stream = filter_in , output is the data stream
    called filter_out
// filter_n = value at n-th time step, filter_n_1 = value at previous (n-1)
    time step ( 1 ms before) , filter_n_2 = value 2ms before. */

float currently_read_data;
unsigned long time_of_curr_read_data;

static float prev_read_data = 0;
static unsigned long time_of_prev_read_data = 0;

float filter_in_n =0, filter_out_n = 0;
```

```
static float filter_in_n_1 = 0, filter_in_n_2 = 0, filter_out_n_1 = 0 ,
    filter_out_n_2 =0 ;

static float  a1= -1.911197067426073, a2 = 0.914975834801434  , b0 =
    0.000944691843840, b1 = 0.001889383687680, b2 = 0.000944691843840 ;
long  i   ,delta_t;

// READ DATA
currently_read_data = -(mb_io_get_float(ID_UI_ANG_RATE_X)- gyro_bias); //
    negative sign because of sign convention difference
time_of_curr_read_data = mb_io_get_time(ID_UI_ANG_RATE_X);

// DO FILTER
delta_t = time_of_curr_read_data - time_of_prev_read_data ;
/*/ considering that the filter runs every 1ms, this variable tells me
// how many data points spaced at 1ms are missing, these data points will be
    assumed to same as the last read data and filter will be run delta_t
    number of times.
// to make up for the fact that it was supposed to run that many times but the
     main brain only calls it every 2ms or so.
// delta_t is not always 2, it is sometimes 1, sometimes 3 sometimes even more
    . This is why the sampling time of filters is chosen to be 1ms, which is
    the least count of delta_t. */
if (delta_t < 0){
    ///////////////////////////////////ERROR//////////////////////////////////
    }
else if (delta_t == 0) {
    if (currently_read_data != prev_read_data){
        /////////////////////////////////ERROR////////////////////////////
      prev_read_data  = currently_read_data;
        }
     }
else
    {
     i = 1;
    while(i < delta_t) // this is loop accounting for the missed filter runs.
      { filter_in_n = prev_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;

        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        i++;
      }
        filter_in_n = currently_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;
        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        mb_io_set_float(ID_E_OUTER_ANG_RATE, filter_out_n);  // this sets the
           estimated data to the estimation id.
        mb_io_set_time(ID_E_OUTER_ANG_RATE, time_of_curr_read_data); // this
           updated the time stamp to be same as the unfiltered data.
        mb_io_set_float(ID_E_INNER_ANG_RATE, filter_out_n - mb_io_get_float(
           ID_E_H_RATE));
        mb_io_set_time(ID_E_INNER_ANG_RATE, time_of_curr_read_data);
    }

prev_read_data = currently_read_data;
time_of_prev_read_data = time_of_curr_read_data;



}




void mb_hip_rate_filter (void)  {
/*/ second order butter worth filter
// Read the documentation for the working of the filters. They are basically
    operating at the 1ms sampling time even though main brain
// runs every 2ms, because the least count of time stamp is 1, the missing
    data is assumed to same as the previous data (this is called zero order
    hold).
// input to the filter is data stream = filter_in , output is the data stream
    called filter_out
// filter_n = value at n-th time step, filter_n_1 = value at previous (n-1)
    time step ( 1 ms before) , filter_n_2 = value 2ms before. */

float currently_read_data;
unsigned long time_of_curr_read_data;

static float prev_read_data = 0;
static unsigned long time_of_prev_read_data = 0;

float filter_in_n =0, filter_out_n = 0;
static float filter_in_n_1 = 0, filter_in_n_2 = 0, filter_out_n_1 = 0 ,
    filter_out_n_2 =0 ;

static float  a1= -1.911197067426073, a2 = 0.914975834801434  , b0 =
    0.000944691843840, b1 = 0.001889383687680, b2 = 0.000944691843840  ;
long  i   ,delta_t;

// READ DATA
currently_read_data = mb_io_get_float(ID_MCH_ANG_RATE);
time_of_curr_read_data = mb_io_get_time(ID_MCH_ANG_RATE);

// DO FILTER
```

```
delta_t = time_of_curr_read_data - time_of_prev_read_data ; /*/ considering
    that the filter runs every 1ms, this variable tells me
// how many data points spaced at 1ms are missing, these data points will be
    assumed to same as the last read data and filter will be run delta_t
    number of times.
// to make up for the fact that it was supposed to run that many times but the
     main brain only calls it every 2ms or so.
// delta_t is not always 2, it is sometimes 1, sometimes 3 sometimes even more
    . This is why the sampling time of filters is chosen to be 1ms, which is
    the least count of delta_t. */

if (delta_t < 0){
    //////////////////////////////////ERROR//////////
    }

else if (delta_t == 0) {
    if (currently_read_data != prev_read_data){
        ////////////////////////////////ERROR////
        prev_read_data  = currently_read_data;
        }
     }
else
    {
     i = 1;
    while(i < delta_t) // this is loop accounting for the missed filter runs.
         { filter_in_n = prev_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;

        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        i++;
      }
        filter_in_n = currently_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;

        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        mb_io_set_float(ID_E_H_RATE, filter_out_n);  // this sets the
           estimated data to the estimation id... time  stamp is current
        mb_io_set_time(ID_E_H_RATE, time_of_curr_read_data); // this updated
           the time stamp to be same as the unfiltered data.
    }

prev_read_data = currently_read_data;
```

```
time_of_prev_read_data = time_of_curr_read_data;



}






void mb_hip_motor_rate_filter (void)  {
/*/ second order butter worth filter
// Read the documentation for the working of the filters. They are basically
    operating at the 1ms sampling time even though main brain
// runs every 2ms, because the least count of time stamp is 1, the missing
    data is assumed to same as the previous data (this is called zero order
    hold).
// input to the filter is data stream = filter_in , output is the data stream
    called filter_out
// filter_n = value at n-th time step, filter_n_1 = value at previous (n-1)
    time step ( 1 ms before) , filter_n_2 = value 2ms before. */


float currently_read_data;
unsigned long time_of_curr_read_data;

static float prev_read_data = 0;
static unsigned long time_of_prev_read_data = 0;

float filter_in_n =0, filter_out_n = 0;
static float filter_in_n_1 = 0, filter_in_n_2 = 0, filter_out_n_1 = 0 ,
    filter_out_n_2 =0 ;

static float  a1= -1.911197067426073, a2 = 0.914975834801434  , b0 =
    0.000944691843840, b1 = 0.001889383687680, b2 = 0.000944691843840  ;
long  i   ,delta_t;

// READ DATA
currently_read_data = mb_io_get_float(ID_MCH_MOTOR_VELOCITY);
time_of_curr_read_data = mb_io_get_time(ID_MCH_MOTOR_VELOCITY);

// DO FILTER
delta_t = time_of_curr_read_data - time_of_prev_read_data ; /*/ considering
    that the filter runs every 1ms, this variable tells me
// how many data points spaced at 1ms are missing, these data points will be
    assumed to same as the last read data and filter will be run delta_t
    number of times.
// to make up for the fact that it was supposed to run that many times but the
     main brain only calls it every 2ms or so.
// delta_t is not always 2, it is sometimes 1, sometimes 3 sometimes even more
    . This is why the sampling time of filters is chosen to be 1ms, which is
    the least count of delta_t. */
```

```
if (delta_t < 0){
    /////////////////////////////////ERROR//////////////
    }

else if (delta_t == 0) {
    if (currently_read_data != prev_read_data){
        /////////////////////////////////ERROR/////////
        prev_read_data  = currently_read_data;
        }
     }
else
    {
     i = 1;
    while(i < delta_t) // this is loop accounting for the missed filter runs.
        { filter_in_n = prev_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;

        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        i++;
      }
        filter_in_n = currently_read_data;
        filter_out_n = b0*filter_in_n + b1*filter_in_n_1  + b2*filter_in_n_2 -
            a1*filter_out_n_1 - a2*filter_out_n_2 ;

        filter_in_n_2 = filter_in_n_1;
        filter_in_n_1 = filter_in_n;

        filter_out_n_2 = filter_out_n_1;
        filter_out_n_1 = filter_out_n;

        mb_io_set_float(ID_E_H_MOTOR_RATE, filter_out_n);  // this sets the
           estimated data to the estimation id... time  stamp is current
        mb_io_set_time(ID_E_H_MOTOR_RATE, time_of_curr_read_data); // this
           updated the time stamp to be same as the unfiltered data.
    }

prev_read_data = currently_read_data;
time_of_prev_read_data = time_of_curr_read_data;



}
```

```
void mb_leg_state_machine_new (void){
/*/ this function registers when the heel strike happens and updates which leg
     is swinging and which is stance,
// and sets various flags.

// the main idea to detect the HS is, as soon as the swing legs crosses the
    other leg to go in front start averaging the value of foot contact sensor,
// this averaging is to find the mean-foot sensor vlues when foot is in air,
    and is accomplished by a simple first order filter. For coding simplicity
    the
// filter is running continuously, but when the swing leg crosses it is reset
    so that it gives the current average by the time the foot is about to hit.
// when the foot is about to hit (based on when the ankle angle is sufficient)
     we store the mean value so far, and we call it the baseline.
// compare the foot sensor value to baseline, if its above by a particular
    threshold for counter_max number of times for either of the two feet. call
     it a HS. */

const unsigned int counter1_max = 2, counter2_max = 2; // number of times to
    wait on the signal to make decision about foot status
float filter_factor = 0.9;
float threshhold_high = 2000.0;
//float threshhold_low  = 1500.0;
float max_baseline_possible = 5000.0; // if baseline is above this then the
    sensor is considered bad or stuck.

static float mean_fo_right = 0, mean_fi_right = 0, mean_fi_left = 0,
    mean_fo_left = 0;  //mean foot-sensor value
static float fo_right_baseline = 0;
static float fi_right_baseline = 0;
static float fi_left_baseline  = 0;
static float fo_left_baseline  = 0;

float hip_angle = mb_io_get_float(ID_MCH_ANGLE);
float outer_ankle_angle = mb_io_get_float(ID_MCFO_RIGHT_ANKLE_ANGLE); // just
    the can id value not the ankleangle of my convention.
float inner_ankle_angle = mb_io_get_float(ID_MCFI_MID_ANKLE_ANGLE);


// update the mean foot sensor data

mean_fo_right = filter_factor*mean_fo_right +  (1-filter_factor)*
    mb_io_get_float(ID_MCFO_RIGHT_HEEL_SENSE);
//mb_io_set_float( ID_E_TEST8 , mean_fo_right); // output the data to a canid
    , for testing.

mean_fo_left =   filter_factor*mean_fo_left    + (1-filter_factor)*
    mb_io_get_float(ID_MCFO_LEFT_HEEL_SENSE);
//mb_io_set_float( ID_E_TEST9 , mean_fo_left);

mean_fi_right = filter_factor*mean_fi_right +  (1-filter_factor)*
    mb_io_get_float(ID_MCFI_RIGHT_HEEL_SENSE);
//mb_io_set_float( ID_E_TEST11 , mean_fi_right);
```

```
mean_fi_left =  filter_factor*mean_fi_left   + (1-filter_factor)*
    mb_io_get_float(ID_MCFI_LEFT_HEEL_SENSE);
//mb_io_set_float( ID_E_TEST12 , mean_fi_left);


// state stuff
 if (est_leg_state_old_version == inner_leg_swings) // inner swing
 {  // reset inner senser filter which finds the mean sensor value when the
     foot is in air.
         if ((hip_angle>0.005)&&(inner_sense_reset == 0)) {
              inner_sense_reset = 1;
              mean_fi_right = mb_io_get_float(ID_MCFI_RIGHT_HEEL_SENSE); // this
                   resets the filter to whatever the current sensor value is.
              mean_fi_left  = mb_io_get_float(ID_MCFI_LEFT_HEEL_SENSE);
         }
         else if((inner_ankle_angle > ankle_flip_down_threshold)&&(
             inner_take_baseline == 0)&&(inner_sense_reset == 1)){
              inner_take_baseline = 1;
              fi_right_baseline = mean_fi_right; // store the mean value of foot
                   in air, so far.
              fi_left_baseline  = mean_fi_left ;
         }
         else if(inner_take_baseline == 1) {
              if ( (mb_io_get_float(ID_MCFI_RIGHT_HEEL_SENSE))>=(threshhold_high
                 +fi_right_baseline) )
                   {
                   counter1 = counter1 + 1;
                   }
              else
                   {
                   counter1 = 0 ;
                   fi_right_state = above_ground;
                   mb_io_set_float( ID_E_TEST4 , (float) fi_right_state); // this
                        line can be commented out
                   }

              if (counter1 >= counter1_max)
                   {
                   counter1 =  0;
                   fi_right_state = on_ground;
                   mb_io_set_float( ID_E_TEST4 , (float) fi_right_state);  //
                       this line can be commented out
                   }

              /////////// //////////////////////////////////
              if ( (mb_io_get_float(ID_MCFI_LEFT_HEEL_SENSE))>=(threshhold_high+
                 fi_left_baseline) )
                   {
                   counter2 = counter2 + 1;
                   }
              else
                   {
                   counter2 = 0 ;
                   fi_left_state = above_ground;
```

```
                mb_io_set_float( ID_E_TEST5 , (float) fi_left_state ); // this
                    line can be commented out
                }

            if (counter2 >= counter2_max)
                {
                counter2 =  0;
                fi_left_state = on_ground;
                mb_io_set_float( ID_E_TEST5 , (float) fi_left_state);  // this
                    line can be commented out
                }

            ////////////////////////////////////////////////
            if (((fi_left_state == on_ground)&&(fi_left_baseline<
               max_baseline_possible)) || ((fi_right_state == on_ground)&&(
               fi_right_baseline<max_baseline_possible)))
                {

                est_leg_state_old_version = outer_leg_swings ;
                est_prev_leg_state = inner_leg_swings;
                reset_integration_flag = 1;  // this tell that its time to
                   current the absolute angle being integrated from the imu
                   rate.
                reset_time_after_HS_flag = 1; // this tells that its time
                   reset the counter which tells how much time has passed
                   since last HS.
                mb_io_set_float( ID_E_TEST3 , est_leg_state_old_version);
                mb_io_set_float( ID_E_SWING_LEG , 0.0);  // 0 means outer, 1
                   means inner

                // reset stuff for next run
                inner_sense_reset = 0;
                inner_take_baseline = 0;
                counter1 =0;
                counter2 =0;
                }


          }

    } else if ((est_leg_state_old_version == outer_leg_swings))
    {  // reset outer sense
          if ((hip_angle<0)&&(outer_sense_reset == 0)) {
              outer_sense_reset = 1;
              mean_fo_right = mb_io_get_float(ID_MCFO_RIGHT_HEEL_SENSE);
              mean_fo_left =  mb_io_get_float(ID_MCFO_LEFT_HEEL_SENSE);
          }
          else if((outer_ankle_angle > ankle_flip_down_threshold)&&(
             outer_take_baseline == 0)&&(outer_sense_reset == 1)){
              outer_take_baseline = 1;
              fo_right_baseline = mean_fo_right;
              fo_left_baseline  = mean_fo_left;
          }
          else if(outer_take_baseline == 1) {
```

```
if ( (mb_io_get_float(ID_MCFO_RIGHT_HEEL_SENSE))>=(threshhold_high
   +fo_right_baseline) )
    {
    counter1 = counter1 + 1;
    }
else
    {
    counter1 = 0 ;
    fo_right_state = above_ground;
    mb_io_set_float( ID_E_TEST6 , (float) fo_right_state);  //
        this line can be commented out
    }

if (counter1 == counter1_max)
    {
    counter1 =  0;
    fo_right_state = on_ground;
    mb_io_set_float( ID_E_TEST6 , (float) fo_right_state);  //
        this line can be commented out
    }

/////////// //////////////////////////////////
if ( (mb_io_get_float(ID_MCFO_LEFT_HEEL_SENSE))>=(threshhold_high+
    fo_left_baseline) )
    {
    counter2 = counter2 + 1;
    }
else
    {
    counter2 = 0 ;
    fo_left_state = above_ground;
    mb_io_set_float( ID_E_TEST7 , (float) fo_left_state);  // this
         line can be commented out
    }

if (counter2 == counter2_max)
    {
    counter2 =  0;
    fo_left_state = on_ground;
    mb_io_set_float( ID_E_TEST7 , (float) fo_left_state);  // this
         line can be commented out
    }

//////////////////////////////////////////////
if (((fo_left_state == on_ground)&&(fo_left_baseline<
    max_baseline_possible)) || ((fo_right_state == on_ground)&&(
    fo_right_baseline<max_baseline_possible) ))
    {

    est_leg_state_old_version = inner_leg_swings ;
    est_prev_leg_state = outer_leg_swings;
    reset_integration_flag = 1;
    reset_time_after_HS_flag = 1;
    mb_io_set_float( ID_E_TEST3 , est_leg_state_old_version);
```

```
                mb_io_set_float( ID_E_SWING_LEG  , 1.0);  // 0 means outer, 1
                    means inner

                // reset stuff for next run
                outer_sense_reset = 0;
                outer_take_baseline = 0;
                counter1 =0;
                counter2 =0;
                }


        }

 }



}


void mb_time_since_last_HS_and_step_counter (void){
static unsigned long int time_at_latest_HS = 0;
unsigned long int time_since_HS = 0;

if ( reset_time_after_HS_flag == 1 ) // when the swing foot hits the gorund.
    This flag is set in leg_state_machine
    { //Set the total time taken for a step before resetting
    float step_time = mb_io_get_time(ID_TIMESTAMP) - time_at_latest_HS;
    mb_io_set_float(ID_E_STEP_TIME, step_time);

    // mark the instant of collision
    time_at_latest_HS = mb_io_get_time(ID_TIMESTAMP);
    reset_time_after_HS_flag = 0 ; // extinguish the flag
    }

time_since_HS = mb_io_get_time(ID_TIMESTAMP) - time_at_latest_HS; // current
    time - instant of collision
mb_io_set_float(ID_E_T_AFTER_HS, (float) time_since_HS);
}


void mb_abs_leg_and_ankle_angles_and_swingfootheight(void){

//variables
static float          int_angle_error_at_hs = 0; // error in angle integrated
    from the imu rate.
static float prev_LO_absang_rate = 0;
float       current_LO_absang_rate = -(mb_io_get_float(ID_UI_ANG_RATE_X)-
    gyro_bias); // negative sign because of sign convention difference
static unsigned long prev_time = 0;
unsigned long       current_time;
float           hip_angle = mb_io_get_float(ID_MCH_ANGLE);
float outer_ankle_angle = mb_io_get_float(ID_MCFO_RIGHT_ANKLE_ANGLE)-
    ankle_angle_offset; // the offset is defined in parameters at the top is
    this file.
float inner_ankle_angle = mb_io_get_float(ID_MCFI_MID_ANKLE_ANGLE)-
    ankle_angle_offset;

float StncAnklAngl, SwngAnklAngl,abs_HipAngl, abs_HipAngl_by2, d_aa, d_atcw,
    d_cc, d_cc_inv, beta, d_atct, d_awcw, foot_radius_t, foot_radius_w;
float complement_alpha, CosAlpha, SinAlpha; // complement_angle = 90 - angle
float complement_sigma, CosSigma, SinSigma;
float StncLegAbsAngl, SwngLegAbsAngl, StncFootAbsAngl, SwngFootAbsAngl,
    angle_d_cc;
float FI_absang, FO_absang, LI_absang , LO_absang,  swing_foot_height,
    ank_height, ank_2_ank_height, virtual_swt_spt_height,
    material_swt_spt_height;

/*/float d_stance_foot_roll;
// static float  prev_swing_foot_height, swing_foot_velocity=0; time_to_hs ;
    // I reset the swing foot velocity to 0 at hs, just after the reset
    integration flag is used.
// float current_slope, abs_swng_ft_heit_just_b4_hs;
//static int time_2_hs_counter;
// const int counter_limit = 5;
// static const float filter_coeff1 = .1;
// static const float filter_coeff2 = (1-filter_coeff1)*500/counter_limit; //
    * 500 is 1/ main brain tick time;
// float time_to_hs_adjustment; */

// integrate the imu rate to get the LO abs ang. Note that integration is by
    the mid point rule and it assumes that time stamps are measured in units
    of ms, hence the factor 0.0005;
 current_time = mb_io_get_time(ID_UI_ANG_RATE_X);
 current_LO_absang_by_integration = prev_LO_absang_by_integration + .0005*(
     prev_LO_absang_rate + current_LO_absang_rate)*(current_time-prev_time);
     // integration done just usingrate gyro, .0005 = .5 (for averaging) *
     .001 (converting millisec to sec, because current_prev_time are in ms)


// reset integration every hs by geometrically calculating absangs using only
    the joint angles, when all four feet are on ground (i.e when the heel
    strike occurs)
 if ( reset_integration_flag == 1 ) // when the swing foot hits the gorund
  {      reset_integration_flag = 0;  // extinguish the flag. This flag is set
       in the state machine again at next HS
         // swing_foot_velocity=0 ;

         step_counter++; // increment the step counter. at the HS.
         mb_io_set_float(ID_E_STEP_NO, (float)step_counter); // number of
             steps so far.

         LO_absang = mb_io_get_float(ID_E_LO_ABSANG);  // absolute angles so
             far
         LI_absang = LO_absang - hip_angle;
         FO_absang = LO_absang + outer_ankle_angle;
         FI_absang = LI_absang + inner_ankle_angle;
```

```
/*///////////////////////////////////////////////////////
// FROM HERE ON: abbreviation: stnc and enclitic: t stand for stance,
     and swng = w = swing.
// see the diagram in documentation for clarity. its basically two
    lines representing legs forming an inverted v. the two feet are
    liek two circles resting on ground.
// The end points of the leg lines (called ankle joints) are at some
    distance away from the foot centers.
// hip angle is anglebetween legs postive when inner leg is in front.
     ankle_angle is angle between leg line and the line joining
// ankle joint to the foot center, postive when this line is in front
     of the leg line.
/////////////////////////////////////////////////////// */

if (hip_angle >= 0)
      {
        StncAnklAngl = inner_ankle_angle;     // stance leg = new
            stance leg = leg which just hit and was swinging till now,
             its the leg in front for forward walking.
        SwngAnklAngl = outer_ankle_angle;
        abs_HipAngl  = hip_angle;
        d_atct = d_ac_inner; // distance from stance leg's angle joint
             to the foot's center = ankle eccentricity.
        d_awcw = d_ac_outer; // same as above but for swing leg.
        foot_radius_t = foot_radius_inner;
        foot_radius_w = foot_radius_outer;
      }
else {
        abs_HipAngl  = -hip_angle;
        StncAnklAngl = outer_ankle_angle;
        SwngAnklAngl = inner_ankle_angle;
        d_atct = d_ac_outer;
        d_awcw = d_ac_inner;
        foot_radius_t = foot_radius_outer;
        foot_radius_w = foot_radius_inner;
      }

abs_HipAngl_by2 = abs_HipAngl/2; // just to save division by 2 so
    many times..

d_aa = 2*L_leg*greg_sin(abs_HipAngl_by2);            // d_aa= distance
     between inner and outer ankle joints
complement_alpha = abs_HipAngl_by2 + SwngAnklAngl;  // alpha = angle
    between d_aa and d_awcw,  complement = pi/2 - angle
CosAlpha = greg_sin(complement_alpha);
SinAlpha = greg_cos(complement_alpha);

d_atcw = greg_sqrt(d_awcw*d_awcw + d_aa*d_aa - 2*d_awcw*d_aa*CosAlpha
    ) ;  // d_atcw = distance between stance leg's ankle joint to
    swinging leg's foot-circle's centre
beta = greg_atan( d_awcw*SinAlpha / (d_aa - d_awcw*CosAlpha) );
              // beta= angle between d_atcw and d_aa
complement_sigma = abs_HipAngl_by2 - StncAnklAngl + beta;
    // sigma is the angle between d_atcw and d_atct
```

```
CosSigma = greg_sin(complement_sigma);
SinSigma = greg_cos(complement_sigma);

StncFootAbsAngl = greg_atan((d_atct - d_atcw*CosSigma)/(d_atcw*
    SinSigma ));
// new change done. because when stance foot is near vertical,
    cmplmnt_StncFootAbsAngl = near pi/2 and its tangent is huge and
    when huge number isfed to greg_atan there is large error in that
    range. Now instead I feed (stancefootangl) to greg tan, which is
    close to 0 and accurate
// plus the sign is taken care of, I dont have to do an if loop

// now correction for the inclination of line d_cc  (this line is
    inclined with respect to horizontal due to differences in foot
    radius)
d_cc =  greg_sqrt(d_atct*d_atct + d_atcw*d_atcw - 2*d_atct*d_atcw*
    CosSigma) ;  // centre to centre distance of foot circles. used
    in calculating distance travelled.
d_cc_inv = 1/d_cc; // one of the few float divisions in this code,
    trying to minimize the instances of this division.
angle_d_cc = anoop_asin((foot_radius_t - foot_radius_w)* d_cc_inv);
StncFootAbsAngl = StncFootAbsAngl - angle_d_cc; // correct the stance
     angles because the foot radii are different, hence d_cc is not
    horizontal. and so far the ansolute angles geometry assumed that
    the foot radii are the same.

// other abs angs
StncLegAbsAngl = StncFootAbsAngl - StncAnklAngl;
SwngLegAbsAngl = StncLegAbsAngl + abs_HipAngl;
SwngFootAbsAngl = SwngLegAbsAngl+ SwngAnklAngl;

// step length

/*/ d_stance_foot_roll = foot_radius_w*(SwngFootAbsAngl-
    prev_StncFootAbsAngl) ; // distance moved by the previous stance
    foot's (which is now considered swing after the hs) center due to
     foot rolling during its stance phase till now
// = foot radius* (absolute angle change of the stance foot between
    two collisions).
//distance_travelled = distance_travelled + d_cc + d_stance_foot_roll
    ;
// this formula gives a better distance experimentally. maybe because
     the d_ankle2sweetspot is more accurate than foot_radius_w. */
distance_travelled = distance_travelled + d_aa + d_ankle2sweetspot*(
    SwngFootAbsAngl-prev_StncFootAbsAngl);
mb_io_set_float(ID_E_TOTAL_DISTANCE , distance_travelled );
prev_StncFootAbsAngl = StncFootAbsAngl;  // as of now I want to put
    the static variable prev_StncFootAbsAngl = 0; ie the foot hits
    the ground in sweet spot. Which is not necessarily true. But
    since after hitting the foot rocks a lot
                         // its hard to find what is the final stance
                             foot abs ang. And taking the
                             prev_stancefootabsangl as the stance foot
```

```
                                  abs angle at the collision (HS) is not a
                                  good idea.
                               // This needs some testing though, so maybe
                                  the next user can make these changes.



        // reset integration
        if (hip_angle < 0)
            {
              int_angle_error_at_hs = current_LO_absang_by_integration -
                 StncLegAbsAngl ;    // Outer Leg is the front leg (New
                 Stance Leg)
            }
        else {
              int_angle_error_at_hs = current_LO_absang_by_integration -
                 SwngLegAbsAngl ;    // Outer leg is the back leg (Next
                 Swinging Leg)
            }

        mb_io_set_float(ID_E_INT_LO_ABSANG_RESET , int_angle_error_at_hs );
           /// difference between abs ang by integration and by the geometry
           .


        //*********** HERE I RESET !!! *************//
       if (step_counter <= 1){ // first time counter hits
        current_LO_absang_by_integration =   current_LO_absang_by_integration
             - int_angle_error_at_hs;
        // first step fully reset based on geometry, because I don't know if
           the initial conditionvalue from which the integration started
           happening is correct or not.
        // initial condition is right now 0 for integration but this is not
           correct if the robot is not turned on with the outerleg in
           vertical position.
        } else {
        current_LO_absang_by_integration =   current_LO_absang_by_integration
             - .1*int_angle_error_at_hs;
        // 0.1 represent that the final angle = 90% of imu integrtion says +
           10% of what the joint angle angles an geometry say.
        // This number was found rounded off from experiments and may not be
           the most optimal.
        }
    }

 LO_absang = current_LO_absang_by_integration; // USING JUST INTEGRATION
    RESETTED every HS as above
// other abs angs
 LI_absang = LO_absang - hip_angle;
 FO_absang = LO_absang + outer_ankle_angle;
 FI_absang = LI_absang + inner_ankle_angle;

// fill the absang canids
 mb_io_set_float(ID_E_LO_ABSANG  ,LO_absang);
```

```
  mb_io_set_float(ID_E_LI_ABSANG  ,LI_absang);
  mb_io_set_float(ID_E_FI_ABSANG  ,FI_absang+foot_fi_absang_calibration_constant
      ); //mb_io_set_float(ID_E_TEST4 , get_io_float(ID_MCFI_MID_ANKLE_ANGLE) -
        get_io_float(ID_MCH_ANGLE) * 0.5 ); //Pranav testing
  mb_io_set_float(ID_E_FO_ABSANG  ,FO_absang+foot_fo_absang_calibration_constant
      ); //mb_io_set_float(ID_E_TEST6 , (get_io_float(ID_MCFO_RIGHT_ANKLE_ANGLE
      ) + get_io_float(ID_MCSO_LEFT_ANKLE_ANGLE))*0.5 + get_io_float(
      ID_MCH_ANGLE) * 0.5);

// recycle the previous stuff (for use in the integration in the beginning of
    this function)
  prev_LO_absang_rate  = current_LO_absang_rate;
  prev_time = current_time;
  prev_LO_absang_by_integration = current_LO_absang_by_integration;

/*///*********************
// FROM NOW ON FOR ALL SWH CALCULATION ABSOLUTE ANGLES ARE MADE RELATIVE TO
    SLOPE OF THE GROUND ////////////
////**********************
// LO_absang = LO_absang - snsr_based_slope;
// LI_absang = LI_absang - snsr_based_slope;
// FO_absang = FO_absang - snsr_based_slope;
// FI_absang = FI_absang - snsr_based_slope; */

// SWING FOOT HEIGHT (based solely on IMU and RESETTED every step)
if((int)mb_io_get_float(ID_E_SWING_LEG) == 1 )
        {// inner leg swings
          ank_2_ank_height         =                          L_leg*(greg_cos(
              LO_absang) -greg_cos(LI_absang))  ;  // vertical distance between
              two ankles (this goes close to 0 at HS because both the feet are
              in the sweet spot position approximately,and the foot radii are
              approximatley equal.
          ank_height               = ank_2_ank_height   + foot_radius_outer -
              d_ac_outer*greg_cos(FO_absang) ; // height of the ankle
          swing_foot_height        = ank_height         - foot_radius_inner +
              d_ac_inner*greg_cos(FI_absang); // height of lowest point of foot
              circle
          material_swt_spt_height = ank_height          - d_ankle2sweetspot*
              greg_cos(FI_absang) ; // height of the sweet spot on the swingin
              foot.
          virtual_swt_spt_height  = ank_height          - d_ankle2sweetspot;  //
              height of the sweet spot of the foot assuming that the foot will
              hit in the sweet spot position, which it might not.

/*/           if(hip_angle>.09){
//              time_2_hs_counter++;
//              if (time_2_hs_counter == counter_limit){
//                  time_2_hs_counter = 0;
//                  swing_foot_velocity = filter_coeff1*swing_foot_velocity -
    filter_coeff2*(swing_foot_height-prev_swing_foot_height);// - because
    velocity is negative, filtering because the velocity is so noisy due to
    numerical differentiation.
//                  time_to_hs = swing_foot_height/swing_foot_velocity;
//                  prev_swing_foot_height = swing_foot_height;
//                  }
//              mb_io_set_float(ID_E_INNER_ANGLE   , time_to_hs -
    time_to_hs_adjustment );
//               }
//          else {
//              time_2_hs_counter = 0;
//              mb_io_set_float(ID_E_INNER_ANGLE   , -.5   );     // putting
    somerandom number there
//               } */
        }


  else { // outerlegswings
        ank_2_ank_height          =                              L_leg*(greg_cos(
            LI_absang) -greg_cos(LO_absang)) ;
        ank_height                = ank_2_ank_height         + foot_radius_inner
            -d_ac_inner*greg_cos(FI_absang) ;
        swing_foot_height         = ank_height               - foot_radius_outer
            +d_ac_outer*greg_cos(FO_absang) ; // height of lowest point of
            foot circle
        material_swt_spt_height = ank_height                 - d_ankle2sweetspot*
            greg_cos(FO_absang) ;
        virtual_swt_spt_height  = ank_height                 - d_ankle2sweetspot;


/*/           if(hip_angle<.09){
//              time_2_hs_counter++;
//              if (time_2_hs_counter == counter_limit){
//                  time_2_hs_counter = 0;
//                  swing_foot_velocity = filter_coeff1*swing_foot_velocity -
    filter_coeff2*(swing_foot_height-prev_swing_foot_height);// - because
    velocity is negative
//                  time_to_hs = swing_foot_height/swing_foot_velocity;
//                  prev_swing_foot_height = swing_foot_height;
//                  }
//              mb_io_set_float(ID_E_INNER_ANGLE   , time_to_hs -
    time_to_hs_adjustment );
//               }
//          else {
//              time_2_hs_counter = 0;
//              mb_io_set_float(ID_E_INNER_ANGLE   , -.5 );    //putting some
    random number there.
//               } */
        }

  mb_io_set_float( ID_E_SWING_F_HEIGHT, swing_foot_height);
  mb_io_set_float( ID_E_ANK2ANK_HT, ank_2_ank_height);  // pranav' is currently
       using this to do the trigger for pre-pushing the stance foot before HS.
      This is the simplest of all the height calculations and hence more
      reliable.
  mb_io_set_float( ID_E_TEST10, virtual_swt_spt_height);
  mb_io_set_float( ID_E_OUTER_ANGLE, material_swt_spt_height);

}
```

```
void mb_model_based_estimator(void){
/*  // THIS IS AN EXPERIMENTAL FUNCTION, trying to implement model based state
     estimation, using an simple observer.
// NOTE d_ac and foot_radis are made different for inner and outer
// smoothes the hip-rate, gyro-rate by estimating a 4 dimentional state vector
float hip_torque, sensor_hip_ang, sensor_hip_angrate, sensor_imu_ang,
    sensor_imu_angrate, sensor_FI_ang, sensor_FO_ang, swing_foot_height;

static float stance_ang=0.0, stance_angrate=0.0, swing_ang=0.0 , swing_angrate
    =0.0 ;  // state vector
float stance_angrate_dot, swing_angrate_dot, temp_ang;
static int phase = 0; // 1 mean inner is stance, -1 means outer is stance

 hip_torque          = mb_io_get_float(ID_MCH_MOTOR_CURRENT)*1.32;   // G*K*Ihip
      // does not include hip spring
 sensor_hip_ang      = mb_io_get_float(ID_MCH_ANGLE);
 sensor_hip_angrate  = mb_io_get_float(ID_MCH_ANG_RATE); //
 sensor_imu_ang      = mb_io_get_float(ID_E_LO_ABSANG); // corrected imu
     actually, by once per step big geometry calcualtions
 sensor_imu_angrate =-mb_io_get_float(ID_UI_ANG_RATE_X)+ gyro_bias; //
     measurments/sensors to be filtered
 sensor_FO_ang       = mb_io_get_float(ID_MCFO_RIGHT_ANKLE_ANGLE) -
     ankle_angle_offset;
 sensor_FI_ang       = mb_io_get_float(ID_MCFI_MID_ANKLE_ANGLE)-
     ankle_angle_offset;


// set the initial conditions by reading sensors every time heelup happens.
//if((est_leg_state_old_version == inner_leg_swings) && (est_prev_leg_state ==
     double_stance)){  // this event happens every heel-up.
if(innerheelup == 1) {
  innerheelup = 0;
  phase = -1;
  stance_ang = sensor_imu_ang;
  stance_angrate = sensor_imu_angrate ;
  swing_ang =  sensor_imu_ang-sensor_hip_ang ;
  swing_angrate = sensor_imu_angrate-sensor_hip_angrate;
//}else if ((est_leg_state_old_version == outer_leg_swings) && (
    est_prev_leg_state == double_stance)) {
} else if (outerheelup == 1) {
  outerheelup = 0;
  phase = 1;
  swing_ang = sensor_imu_ang;
  swing_angrate = sensor_imu_angrate ;
  stance_ang = sensor_imu_ang-sensor_hip_ang ;
  stance_angrate = sensor_imu_angrate-sensor_hip_angrate ;
}

// now integrate
if (phase == 1)
{ hip_torque =-hip_torque; }  // inner leg is stance

temp_ang = stance_ang-swing_ang;
```

```
stance_angrate_dot = 10.8555*greg_sin(stance_ang); //
swing_angrate_dot  = 2*(0.6912*greg_cos(temp_ang)*stance_angrate_dot -
    hip_torque + 7.5*temp_ang - 7.056*greg_sin(swing_ang) - 0.6912*greg_sin(
    temp_ang)*stance_angrate*stance_angrate);

//stance_angrate_dot = 10.8555*stance_ang;
//swing_angrate_dot  = 0;

stance_ang = stance_ang + stance_angrate*.002 ; // assuming my function is
    called every 2 millisecond
swing_ang  = swing_ang  + swing_angrate*.002;
stance_angrate = stance_angrate + stance_angrate_dot*.002;
swing_angrate  =  swing_angrate +  swing_angrate_dot*.002;

// now filter and put the values into required canid's
if (phase == -1){// outerleg is stance
stance_ang = stance_ang + .9*(sensor_imu_ang                   -stance_ang);
swing_ang  = stance_ang + .9*(sensor_imu_ang-sensor_hip_ang    -swing_ang);
stance_angrate = stance_angrate + .02*(sensor_imu_angrate                  -
    stance_angrate);
swing_angrate  = swing_angrate  + .02*(sensor_imu_angrate-sensor_hip_angrate -
    swing_angrate);

mb_io_set_float(ID_E_TEST3, swing_foot_height); // estimated hip rate
mb_io_set_float(ID_E_TEST4, stance_angrate-swing_angrate); // estimated hip
    rate
mb_io_set_float(ID_E_TEST5, stance_angrate);  // estimated gyro rate  // test
    five is used somewhere else (in anoop_asin)

} else if (phase == 1){ //inner leg is stance
stance_ang = stance_ang + .9*(sensor_imu_ang-sensor_hip_ang - stance_ang);
swing_ang  = stance_ang + .9*(sensor_imu_ang                - swing_ang);
swing_angrate   = swing_angrate   + .02*(sensor_imu_angrate
                      -swing_angrate);
stance_angrate  = stance_angrate  + .02*(sensor_imu_angrate-sensor_hip_angrate
     -stance_angrate);


mb_io_set_float(ID_E_TEST4,swing_angrate-stance_angrate); // estimated hip
    rate
mb_io_set_float(ID_E_TEST5,swing_angrate);  // estimated gyro rate


}
*/
}


//The MAIN function
void mb_estimator_update(void) { // main function called by SCHEDULER every
    row (approx 2ms as of 1 Jan 2011)

 // reset the estimation variables whenever the robot transitions to walk
     state from some other state.
```

```
// and hence is about to begin a new segment of its walk.

 curr_ui_fsm_state = (int) (mb_io_get_float(ID_FSM_RESET)+.1);
 if ((curr_ui_fsm_state==2)&& (curr_ui_fsm_state != prev_ui_fsm_state)){ //
     initialize things here.
     step_counter = 0;
     distance_travelled = 0;

     mb_io_set_float( ID_E_SWING_LEG , 1.0);
     est_leg_state_old_version = inner_leg_swings ,
     est_prev_leg_state = inner_leg_swings;


     fi_left_state = above_ground;
     fi_right_state= above_ground;
     fo_left_state = above_ground;
     fo_right_state= above_ground; //individual foot state
     inner_sense_reset = 0;
     outer_sense_reset = 0;      //flags
     inner_take_baseline = 0;
     outer_take_baseline = 0; //flags
     counter1=0;
     counter2=0;


     prev_LO_absang_by_integration = 0;
     current_LO_absang_by_integration = 0;
     prev_StncFootAbsAngl=0;

     reset_integration_flag = 0;
     reset_time_after_HS_flag = 0;
     }
 prev_ui_fsm_state = curr_ui_fsm_state;


 //call filters (2nd order butterworth)
 mb_hip_rate_filter();
 mb_hip_motor_rate_filter(); // motor rate is obtained from the encoder in
     the motor.
 mb_imu_rate_filter();

 //call the state machine
mb_leg_state_machine_new (); // do heelstrike detection better using the
    direct data from heel strike sensors.

 // angles,distance,swingfoot height
 mb_time_since_last_HS_and_step_counter(); // time since last HS and how many
      steps the robot has moved.
 mb_abs_leg_and_ankle_angles_and_swingfootheight();
 // the most important function which calculates the distance travelled by
     the robot.
 // calculates the swing foot heights used to predict the HS in adavance
 // and most importantly calculated the absolute angles, by integrating the
     imu rate and
```

```
  // partially resetting it at every heel strike to account for gyro/
      integration errors
  // We are able to reset because at HS, all the feet are on ground and hence
      the absolute angles
  // can be calculated by just geometry and the relative joint angles (joint
      angle sensors are pretty accurate)


  // modelbased estimation.
// mb_model_based_estimator();

}


//Utility functions  (called during other functions, optimized for speed, no
    division of floats)
float greg_sin(float xin) {  //// Greg Steisberg's sine function using
    quadratic interpolation)
     // int k,klo,khi;
      int klo,khi;
     float xdiff0 , xdiff1;
     float x=xin;
static float xlo = -3.141592653589793;
static float xhi = 3.141592653589793;
static float ya[91] = {-1.224606353822377e-16, -0.06975647374412552,
    -0.1391731009600657, -0.2079116908177593, -0.2756373558169992,
    -0.3420201433256689, -0.4067366430758004, -0.4694715627858907,
    -0.5299192642332049, -0.5877852522924732, -0.6427876096865395,
    -0.6946583704589975, -0.7431448254773942, -0.7880107536067220,
    -0.8290375725550417, -0.8660254037844387, -0.8987940462991669,
    -0.9271838545667874, -0.9510565162951536, -0.9702957262759965,
    -0.9848077530122080, -0.9945218953682734, -0.9993908270190958,
    -0.9993908270190958, -0.9945218953682733, -0.9848077530122080,
    -0.9702957262759965, -0.9510565162951535, -0.9271838545667873,
    -0.8987940462991670, -0.8660254037844386, -0.8290375725550417,
    -0.7880107536067219, -0.7431448254773941, -0.6946583704589973,
    -0.6427876096865393, -0.5877852522924731, -0.5299192642332049,
    -0.4694715627858908, -0.4067366430758002, -0.3420201433256687,
    -0.2756373558169992, -0.2079116908177593, -0.1391731009600654,
    -0.06975647374412530, 0.000000000000000, 0.06975647374412530,
    0.1391731009600654, 0.2079116908177593, 0.2756373558169992,
    0.3420201433256687, 0.4067366430758002, 0.4694715627858908,
    0.5299192642332049, 0.5877852522924731, 0.6427876096865393,
    0.6946583704589973, 0.7431448254773941, 0.7880107536067219,
    0.8290375725550417, 0.8660254037844386, 0.8987940462991670,
    0.9271838545667873, 0.9510565162951535, 0.9702957262759965,
    0.9848077530122080, 0.9945218953682733, 0.9993908270190958,
    0.9993908270190958, 0.9945218953682734, 0.9848077530122080,
    0.9702957262759965, 0.9510565162951536, 0.9271838545667874,
    0.8987940462991669, 0.8660254037844387, 0.8290375725550417,
    0.7880107536067220, 0.7431448254773942, 0.6946583704589975,
    0.6427876096865395, 0.5877852522924732, 0.5299192642332049,
    0.4694715627858907, 0.4067366430758004, 0.3420201433256689,
    0.2756373558169992, 0.2079116908177593, 0.1391731009600657,
    0.06975647374412552, 1.224606353822377e-16};
static float y2a[91] = {0.000000000000000, 0.03486407314861756,
    0.06955829203737018, 0.1039136299193089, 0.1377627110416193,
    0.1709406260836165, 0.2032857355782110, 0.2346404574032933,
    0.2648520345058517, 0.2937732791191829, 0.3212632898465866,
    0.3471881381191405, 0.3714215206820309, 0.3938453749312992,
    0.4143504541030941, 0.4328368595128685, 0.4492145272517128,
    0.4634036669686713, 0.4753351506013358, 0.4849508491607664,
    0.4922039159301915, 0.4970590146973936, 0.4994924919091513,
    0.4994924919091627, 0.4970590146973708, 0.4922039159302029,
    0.4849508491607779, 0.4753351506013244, 0.4634036669686585,
    0.4492145272517271, 0.4328368595128671, 0.4143504541031054,
    0.3938453749312955, 0.3714215206820106, 0.3471881381191607,
    0.3212632898465763, 0.2937732791191838, 0.2648520345058525,
    0.2346404574032984, 0.2032857355782058, 0.1709406260836170,
    0.1377627110416227, 0.1039136299193038, 0.06955829203737375,
    0.03486407314861598, 0.000000000000000, -0.03486407314861598,
    -0.06955829203737375, -0.1039136299193038, -0.1377627110416227,
    -0.1709406260836170, -0.2032857355782058, -0.2346404574032984,
    -0.2648520345058525, -0.2937732791191838, -0.3212632898465763,
    -0.3471881381191607, -0.3714215206820106, -0.3938453749312955,
    -0.4143504541031054, -0.4328368595128671, -0.4492145272517271,
    -0.4634036669686585, -0.4753351506013244, -0.4849508491607779,
    -0.4922039159302029, -0.4970590146973708, -0.4994924919091627,
    -0.4994924919091513, -0.4970590146973936, -0.4922039159301915,
    -0.4849508491607664, -0.4753351506013358, -0.4634036669686713,
    -0.4492145272517128, -0.4328368595128685, -0.4143504541030941,
    -0.3938453749312992, -0.3714215206820309, -0.3471881381191405,
    -0.3212632898465866, -0.2937732791191829, -0.2648520345058517,
    -0.2346404574032933, -0.2032857355782110, -0.1709406260836165,
    -0.1377627110416193, -0.1039136299193089, -0.06955829203737018,
    -0.03486407314861756, -0.03486407314861756};
// static int numpoints = 91;
static float h = 0.06981317007977318;
static float hinv = 14.32394487827058;
static float pi2=6.283185307179587;
static float pi2inv=0.1591549430918953;
if(xin<xlo){
    x=xin+((int)((xhi-xin)*pi2inv))*pi2;
}else if(xin>xhi){
    x=xin-((int)((xin-xlo)*pi2inv))*pi2;
}
    klo=(int)((x-xlo)*hinv);
    khi=klo+1;
    xdiff0=(x-(xlo+h*klo));
    xdiff1=xdiff0-h;
    return( ya[klo] + (ya[khi]-ya[klo])*hinv*xdiff0 + y2a[khi]*xdiff0*xdiff1 )
        ;
}



float greg_cos(float xin)  { //// Greg Steisberg's cosine function using
```

```
    quadratic interpolation)
    //int k,klo,khi;
     int klo,khi;
    float xdiff0, xdiff1;
    float x=xin;
static float xlo = -3.141592653589793;
static float xhi = 3.141592653589793;
static float ya[91] = {-1.000000000000000, -0.9975640502598242,
    -0.9902680687415703, -0.9781476007338057, -0.9612616959383189,
    -0.9396926207859083, -0.9135454576426008, -0.8829475928589270,
    -0.8480480961564260, -0.8090169943749473, -0.7660444431189779,
    -0.7193398003386510, -0.6691306063588582, -0.6156614753256583,
    -0.5591929034707467, -0.4999999999999998, -0.4383711467890775,
    -0.3746065934159121, -0.3090169943749473, -0.2419218955996676,
    -0.1736481776669303, -0.1045284632676533, -0.03489949670250096,
    0.03489949670250108, 0.1045284632676535, 0.1736481776669304,
    0.2419218955996677, 0.3090169943749475, 0.3746065934159122,
    0.4383711467890775, 0.5000000000000001, 0.5591929034707468,
    0.6156614753256583, 0.6691306063588582, 0.7193398003386512,
    0.7660444431189780, 0.8090169943749475, 0.8480480961564260,
    0.8829475928589270, 0.9135454576426009, 0.9396926207859084,
    0.9612616959383189, 0.9781476007338057, 0.9902680687415704,
    0.9975640502598242, 1.000000000000000, 0.9975640502598242,
    0.9902680687415704, 0.9781476007338057, 0.9612616959383189,
    0.9396926207859084, 0.9135454576426009, 0.8829475928589270,
    0.8480480961564260, 0.8090169943749475, 0.7660444431189780,
    0.7193398003386512, 0.6691306063588582, 0.6156614753256583,
    0.5591929034707468, 0.5000000000000001, 0.4383711467890775,
    0.3746065934159122, 0.3090169943749475, 0.2419218955996677,
    0.1736481776669304, 0.1045284632676535, 0.03489949670250108,
    -0.03489949670250096, -0.1045284632676533, -0.1736481776669303,
    -0.2419218955996676, -0.3090169943749473, -0.3746065934159121,
    -0.4383711467890775, -0.4999999999999998, -0.5591929034707467,
    -0.6156614753256583, -0.6691306063588582, -0.7193398003386510,
    -0.7660444431189779, -0.8090169943749473, -0.8480480961564260,
    -0.8829475928589270, -0.9135454576426008, -0.9396926207859083,
    -0.9612616959383189, -0.9781476007338057, -0.9902680687415703,
    -0.9975640502598242, -1.000000000000000};
static float y2a[91] = {0.000000000000000, 0.4985794744479298,
    0.4949329651034867, 0.4888751921035583, 0.4804356683092536,
    0.4696555102321589, 0.4565872377188284, 0.4412945180793348,
    0.4238518559065110, 0.4043442300971526, 0.3828666798433050,
    0.3595238416108955, 0.3344294393613353, 0.3077057304999832,
    0.2794829102500575, 0.2498984773549504, 0.2190965641978670,
    0.1872272346035282, 0.1544457527422005, 0.1209118266983633,
    0.08678883038885456, 0.05224300762167937, 0.01744266217281756,
    -0.01744266217281756, -0.05224300762168079, -0.08678883038885313,
    -0.1209118266983633, -0.1544457527422005, -0.1872272346035248,
    -0.2190965641978707, -0.2498984773549532, -0.2794829102500658,
    -0.3077057304999634, -0.3344294393613453, -0.3595238416108968,
    -0.3828666798432948, -0.4043442300971652, -0.4238518559065009,
    -0.4412945180793457, -0.4565872377188172, -0.4696555102321718,
    -0.4804356683092443, -0.4888751921035540, -0.4949329651035017,
    -0.4985794744479198, -0.4997969547099114, -0.4985794744479198,
    -0.4949329651035017, -0.4888751921035540, -0.4804356683092443,
    -0.4696555102321718, -0.4565872377188172, -0.4412945180793457,
    -0.4238518559065009, -0.4043442300971652, -0.3828666798432948,
    -0.3595238416108968, -0.3344294393613453, -0.3077057304999634,
    -0.2794829102500658, -0.2498984773549532, -0.2190965641978707,
    -0.1872272346035248, -0.1544457527422005, -0.1209118266983633,
    -0.08678883038885313, -0.05224300762168079, -0.01744266217281756,
    0.01744266217281756, 0.05224300762167937, 0.08678883038885456,
    0.1209118266983633, 0.1544457527422005, 0.1872272346035282,
    0.2190965641978670, 0.2498984773549504, 0.2794829102500575,
    0.3077057304999832, 0.3344294393613353, 0.3595238416108955,
    0.3828666798433050, 0.4043442300971526, 0.4238518559065110,
    0.4412945180793348, 0.4565872377188284, 0.4696555102321589,
    0.4804356683092536, 0.4888751921035583, 0.4949329651034867,
    0.4985794744479298, 0.4985794744479298};
// static int numpoints = 91;
static float h = 0.06981317007977318;
static float hinv = 14.32394487827058;
static float pi2=6.283185307179587;
static float pi2inv=0.1591549430918953;
if(xin<xlo){
    x=xin+((int)((xhi-xin)*pi2inv))*pi2;
}else if(xin>xhi){
    x=xin-((int)((xin-xlo)*pi2inv))*pi2;
}
    klo=(int)((x-xlo)*hinv);
    khi=klo+1;
    xdiff0=(x-(xlo+h*klo));
    xdiff1=xdiff0-h;
    return( ya[klo] + (ya[khi]-ya[klo])*hinv*xdiff0 + y2a[khi]*xdiff0*xdiff1 )
        ;
}



float greg_atan(float xin)    { // Greg Steisberg's tangent inverse function
    using quadratic interpolation)
    //int k,klo,khi;
     int klo,khi;
    float xdiff0, xdiff1;
    float x=xin;
static float xlo = -40.00000000000000;
static float xhi = 40.00000000000001;
static float ya[401] = {-1.545801533175976, -1.545675983898545,
    -1.545549167250735, -1.545421063948932, -1.545291654316402,
    -1.545160918273219, -1.545028835325896, -1.544895384556677,
    -1.544760544612509, -1.544624293693659, -1.544486609541974,
    -1.544347469428778, -1.544206850142363, -1.544064727975104,
    -1.543921078710140, -1.543775877607632, -1.543629099390573,
    -1.543480718230135, -1.543330707730525, -1.543179040913354,
    -1.543025690201476, -1.542870627402284, -1.542713823690457,
    -1.542555249590109, -1.542394874956336, -1.542232668956137,
    -1.542068600048667, -1.541902635964819, -1.541734743686084,
    -1.541564889422676, -1.541393038590892, -1.541219155789653,
    −1.541043204776231, −1.540865148441088, −1.540684948781814,
    −1.540502566876121, −1.540317962853851, −1.540131095867948,
    −1.539941924064371, −1.539750404550873, −1.539556493364628,
    −1.539360145438622, −1.539161314566784, −1.538959953367782,
    −1.538756013247430, −1.538549444359643, −1.538340195565877,
    −1.538128214392981, −1.537913446989385, −1.537695838079558,
    −1.537475330916649, −1.537251867233218, −1.537025387189983,
    −1.536795829322475, −1.536563130485516, −1.536327225795389,
    −1.536088048569622, −1.535845530264244, −1.535599600408398,
    −1.535350186536182, −1.535097214115573, −1.534840606474297,
    −1.534580284722485, −1.534316167671947, −1.534048171751901,
    −1.533776210920966, −1.533500196575230, −1.533220037452173,
    −1.532935639530258, −1.532646905923914, −1.532353736773709,
    −1.532056029131417, −1.531753676839730, −1.531446570406291,
    −1.531134596871769, −1.530817639671607, −1.530495578491115,
    −1.530168289113524, −1.529835643260581, −1.529497508425290,
    −1.529153747696308, −1.528804219573527, −1.528448777774321,
    −1.528087271029893, −1.527719542871135, −1.527345431403366,
    −1.526964769069259, −1.526577382399244, −1.526183091748594,
    −1.525781711020370, −1.525373047373320, −1.524956900913780,
    −1.524533064370540, −1.524101322751566, −1.523661452981396,
    −1.523213223517913, −1.522756393947134, −1.522290714554513,
    −1.521815925871169, −1.521331758193305, −1.520837931072954,
−1.520334152778041,
  −1.519820119719583, −1.519295515843671, −1.518760011985680,
     −1.518213265183955, −1.517654917949961, −1.517084597491664,
     −1.516501914886591, −1.515906464200743, −1.515297821549180,
     −1.514675544093720, −1.514039168972803, −1.513388212158097,
     −1.512722167231943, −1.512040504079174, −1.511342667486240,
     −1.510628075639887, −1.509896118516909, −1.509146156155638,
     −1.508377516798939, −1.507589494897436, −1.506781348960553,
     −1.505952299241690, −1.505101525242432, −1.504228163019073,
     −1.503331302272992, −1.502409983204393, −1.501463193106688,
     −1.500489862676272, −1.499488862009606, −1.498458996256302,
     −1.497399000893285, −1.496307536581015, −1.495183183558067,
     −1.494024435525119, −1.492829692963354, −1.491597255825445,
     −1.490325315529436, −1.489011946176901, −1.487655094906455,
     −1.486252571281899, −1.484802035600656, −1.483300985992505,
     −1.481746744160399, −1.480136439594151, −1.478466992063298,
     −1.476735092166910, −1.474937179684883, −1.473069419436178,
     −1.471127674303735, −1.469107475031820, −1.467003986337854,
     −1.464811968805297, −1.462525735934441, −1.460139105621001,
     −1.457645345204412, −1.455037109074086, −1.452306367636759,
     −1.449444326224133, −1.446441332248135, −1.443286768579658,
     −1.439968930720840, −1.436474884841928, −1.432790303137377,
     −1.428899272190733, −1.424784069083621, −1.420424898787762,
     −1.415799584870956, −1.410883203636677, −1.405647649380270,
     −1.400061115319614, −1.394087470724860, −1.387685509532413,
     −1.380808038876181, −1.373400766945016, −1.365400937605129,
     −1.356735643231075, −1.347319725654264, −1.337053145925995,
     −1.325817663668033, −1.313472611823808, −1.299849476456476,
     −1.284744885077578, −1.267911458419925, −1.249045772398254,
     −1.227772386374193, −1.203622492976677, −1.176005207095135,
     −1.144168833668020, −1.107148717794090, −1.063697822402560,
    −1.012197011451334, −0.9505468408120751, −0.8760580505981934,
    −0.7853981633974483, −0.6747409422235527, −0.5404195002705842,
    −0.3805063771123649, −0.1973955598498808, 0.000000000000000,
    0.1973955598498808, 0.3805063771123649,
 0.5404195002705842,
 0.6747409422235527, 0.7853981633974483, 0.8760580505981934,
    0.9505468408120751, 1.012197011451334, 1.063697822402560,
    1.107148717794090, 1.144168833668020, 1.176005207095135,
    1.203622492976677, 1.227772386374193, 1.249045772398254,
    1.267911458419925, 1.284744885077578, 1.299849476456476,
    1.313472611823808, 1.325817663668033, 1.337053145925995,
    1.347319725654264, 1.356735643231075, 1.365400937605129,
    1.373400766945016, 1.380808038876181, 1.387685509532413,
    1.394087470724860, 1.400061115319614, 1.405647649380270,
    1.410883203636677, 1.415799584870956, 1.420424898787762,
    1.424784069083621, 1.428899272190733, 1.432790303137377,
    1.436474884841928, 1.439968930720840, 1.443286768579658,
    1.446441332248135, 1.449444326224133, 1.452306367636759,
    1.455037109074086, 1.457645345204412, 1.460139105621001,
    1.462525735934441, 1.464811968805297, 1.467003986337854,
    1.469107475031820, 1.471127674303735, 1.473069419436178,
    1.474937179684883, 1.476735092166910, 1.478466992063298,
    1.480136439594151, 1.481746744160399, 1.483300985992505,
    1.484802035600656, 1.486252571281899, 1.487655094906455,
    1.489011946176901, 1.490325315529436, 1.491597255825445,
    1.492829692963354, 1.494024435525119, 1.495183183558067,
    1.496307536581015, 1.497399000893285, 1.498458996256302,
    1.499488862009606, 1.500489862676272, 1.501463193106688,
    1.502409983204393, 1.503331302272992, 1.504228163019073,
    1.505101525242432, 1.505952299241690, 1.506781348960553,
    1.507589494897436, 1.508377516798939, 1.509146156155638,
    1.509896118516909, 1.510628075639887, 1.511342667486240,
    1.512040504079174, 1.512722167231943, 1.513388212158097,
    1.514039168972803, 1.514675544093720, 1.515297821549180,
    1.515906464200743, 1.516501914886591, 1.517084597491664,
    1.517654917949961, 1.518213265183955, 1.518760011985680,
    1.519295515843671, 1.519820119719583, 1.520334152778041,
    1.520837931072954, 1.521331758193305, 1.521815925871169,
    1.522290714554513, 1.522756393947134, 1.523213223517913,
    1.523661452981396, 1.524101322751566, 1.524533064370540,
    1.524956900913780,
 1.525373047373320,
1.525781711020370, 1.526183091748594, 1.526577382399244, 1.526964769069259,
   1.527345431403366, 1.527719542871135, 1.528087271029893,
   1.528448777774321, 1.528804219573527, 1.529153747696308,
   1.529497508425290, 1.529835643260581, 1.530168289113524,
   1.530495578491115, 1.530817639671607, 1.531134596871769,
   1.531446570406291, 1.531753676839730, 1.532056029131417,
   1.532353736773709, 1.532646905923914, 1.532935639530258,
   1.533220037452173, 1.533500196575230, 1.533776210920966,
   1.534048171751901, 1.534316167671947, 1.534580284722485,
   1.534840606474297, 1.535097214115573, 1.535350186536182,
   1.535599600408398, 1.535845530264244, 1.536088048569622,
   1.536327225795389, 1.536563130485516, 1.536795829322475,
    1.537025387189983, 1.537251867233218, 1.537475330916649,
    1.537695838079558, 1.537913446989385, 1.538128214392981,
    1.538340195565877, 1.538549444359643, 1.538756013247430,
    1.538959953367782, 1.539161314566784, 1.539360145438622,
    1.539556493364628, 1.539750404550873, 1.539941924064371,
    1.540131095867948, 1.540317962853851, 1.540502566876121,
    1.540684948781814, 1.540865148441088, 1.541043204776231,
    1.541219155789653, 1.541393038590892, 1.541564889422676,
    1.541734743686084, 1.541902635964819, 1.542068600048667,
    1.542232668956137, 1.542394874956336, 1.542555249590109,
    1.542713823690457, 1.542870627402284, 1.543025690201476,
    1.543179040913354, 1.543330707730525, 1.543480718230135,
    1.543629099390573, 1.543775877607632, 1.543921078710140,
    1.544064727975104, 1.544206850142363, 1.544347469428778,
    1.544486609541974, 1.544624293693659, 1.544760544612509,
    1.544895384556677, 1.545028835325896, 1.545160918273219,
    1.545291654316402, 1.545421063948932, 1.545549167250735,
    1.545675983898545, 1.545801533175976};

static float y2a[401] = {0.000000000000000, 1.584212972477525e-05,
    1.608317490570012e-05, 1.632913410326743e-05, 1.658013314969961e-05,
    1.683630175830461e-05, 1.709777369413381e-05, 1.736468686613530e-05,
    1.763718353037717e-05, 1.791541041900638e-05, 1.819951891135433e-05,
    1.848966522117309e-05, 1.878601055543114e-05, 1.908872131201864e-05,
    1.939796930021926e-05, 1.971393187860604e-05, 2.003679225599426e-05,
    2.036673963639096e-05, 2.070396951110494e-05, 2.104868385137585e-05,
    2.140109140735752e-05, 2.176140794514504e-05, 2.212985651943396e-05,
    2.250666780452707e-05, 2.289208033405335e-05, 2.328634087500548e-05,
    2.368970473128469e-05, 2.410243609133709e-05, 2.452480839806060e-05,
    2.495710471887216e-05, 2.539961817260500e-05, 2.585265228846071e-05,
    2.631652152021920e-05, 2.679155163562077e-05, 2.727808023342566e-05,
    2.777645722318233e-05, 2.828704540333513e-05, 2.881022093106919e-05,
    2.934637399762555e-05, 2.989590934801346e-05, 3.045924701354645e-05,
    3.103682290062104e-05, 3.162908955018882e-05, 3.223651687815934e-05,
    3.285959292997932e-05, 3.349882473358014e-05, 3.415473913692704e-05,
    3.482788374637667e-05, 3.551882787616078e-05, 3.622816353576778e-05,
    3.695650652446618e-05, 3.770449755746724e-05, 3.847280339303486e-05,
    3.926211815668640e-05, 4.007316459497840e-05, 4.090669549483590e-05,
    4.176349513718177e-05, 4.264438084872285e-05, 4.355020463425631e-05,
    4.448185491057220e-05, 4.544025832831121e-05, 4.642638170571123e-05,
    4.744123407446022e-05, 4.848586885048161e-05, 4.956138610623419e-05,
    5.066893503102890e-05, 5.180971648921543e-05, 5.298498574329073e-05,
    5.419605535190363e-05, 5.544429827197867e-05, 5.673115107129907e-05,
    5.805811745529562e-05, 5.942677188390642e-05, 6.083876354929943e-05,
    6.229582050210114e-05, 6.379975411308126e-05, 6.535246375172267e-05,
    6.695594188525318e-05, 6.861227935258276e-05, 7.032367114467914e-05,
    7.209242248451593e-05, 7.392095531799700e-05, 7.581181527263900e-05,
    7.776767911590563e-05, 7.979136264491039e-05, 8.188582922286139e-05,
    8.405419885393399e-05, 8.629975793141954e-05,
8.862596968259718e-05,
 9.103648531960727e-05, 9.353515612320406e-05, 9.612604625342037e-05,
     9.881344666663848e-05, 0.0001016018899597639, 0.0001044961664087304,
     0.0001075013412038536, 0.0001106227730224293, 0.0001138661340326417,
     0.0001172374315073718, 0.0001207430310901101, 0.0001243896820213317,
     0.0001281845443040784, 0.0001321352181836653, 0.0001362497759839262,
     0.0001405367966833558, 0.0001450054033513847, 0.0001496653037946782,
     0.0001545268347029430, 0.0001596010096806170, 0.0001648995714392821,
     0.0001704350487075080, 0.0001762208182154458, 0.0001822711723663212,
     0.0001886013931030937, 0.0001952278326777815, 0.0002021680020697373,
     0.0002094406677338749, 0.0002170659578164017, 0.0002250654786596873,
     0.0002334624428418380, 0.0002422818100561459, 0.0002515504422622381,
     0.0002612972747368730, 0.0002715535049406976, 0.0002823528012619711,
     0.0002937315340234580, 0.0003057290314744205, 0.0003183878638291980,
     0.0003317541588830946, 0.0003458779531257929, 0.0003608135829835682,
     0.0003766201214032649, 0.0003933618656638626, 0.0004111088834800476,
     0.0004299376249962278, 0.0004499316102052374, 0.0004711822018081711,
     0.0004937894762525867, 0.0005178632065726218, 0.0005435239738783933,
     0.0005709044263918535, 0.0006001507085738393, 0.0006314240863504701,
     0.0006649027994463610, 0.0007007841767632884, 0.0007392870575800899,
     0.0007806545691683472, 0.0008251573204944576, 0.0008730970834811728,
     0.0009248110467238046, 0.0009806767433978639, 0.001041117775632308,
     0.001106610482393370, 0.001177691728737356, 0.001254968032294003,
     0.001339126289365287, 0.001430946421715199, 0.001531316337511095,
     0.001641249691236266, 0.001761907042150169, 0.001894621155984322,
     0.002040927379276521, 0.002202600251158455, 0.002381697820494162,
     0.002580615526168168, 0.002802152005837439, 0.003049589859346885,
     0.003326795261840690, 0.003638341468397999, 0.003989662776615603,
     0.004387247553105925, 0.004838881676222364, 0.005353957471171757,
     0.005943868297300294, 0.006622515936668948, 0.007406967609019364,
     0.008318312927094158, 0.009382790034465599,
 0.01063327689321495,
 0.01211128162117434, 0.01386961982827448, 0.01597604403884326,
     0.01851820014457067, 0.02161044098444632, 0.02540324205021711,
     0.03009625002988176, 0.03595634216818142, 0.04334240605033331,
     0.05273859431965207, 0.06479678058519307, 0.08038474397000833,
     0.1006239444961865, 0.1268669961004170, 0.1604827446827836,
     0.2021387123357935, 0.2499666746643801, 0.2958027597384121,
     0.3198960150656345, 0.2899711763033103, 0.1785592823424581,
     0.000000000000000, −0.1785592823424581, −0.2899711763033103,
     −0.3198960150656345, −0.2958027597384121, −0.2499666746643801,
     −0.2021387123357935, −0.1604827446827836, −0.1268669961004170,
     −0.1006239444961865, −0.08038474397000833, −0.06479678058519307,
     −0.05273859431965207, −0.04334240605033331, −0.03595634216818142,
     −0.03009625002988176, −0.02540324205021711, −0.02161044098444632,
     −0.01851820014457067, −0.01597604403884326, −0.01386961982827448,
     −0.01211128162117434, −0.01063327689321495, −0.009382790034465599,
     −0.008318312927094158, −0.007406967609019364, −0.006622515936668948,
     −0.005943868297300294, −0.005353957471171757, −0.004838881676222364,
     −0.004387247553105925, −0.003989662776615603, −0.003638341468397999,
     −0.003326795261840690, −0.003049589859346885, −0.002802152005837439,
     −0.002580615526168168, −0.002381697820494162, −0.002202600251158455,
     −0.002040927379276521, −0.001894621155984322, −0.001761907042150169,
     −0.001641249691236266, −0.001531316337511095, −0.001430946421715199,
     −0.001339126289365287, −0.001254968032294003, −0.001177691728737356,
     −0.001106610482393370, −0.001041117775632308, −0.0009806767433978639,
     −0.0009248110467238046, −0.0008730970834811728, −0.0008251573204944576,
     −0.0007806545691683472, −0.0007392870575800899, −0.0007007841767632884,
     −0.0006649027994463610, −0.0006314240863504701, −0.0006001507085738393,
     -0.0005709044263918535, -0.0005435239738783933, -0.0005178632065726218,
     -0.0004937894762525867, -0.0004711822018081711, -0.0004499316102052374,
     -0.0004299376249962278, -0.0004111088834800476, -0.0003933618656638626,
     -0.0003766201214032649,
   -0.0003608135829835682,
   -0.0003458779531257929, -0.0003317541588830946, -0.0003183878638291980,
     -0.0003057290314744205, -0.0002937315340234580, -0.0002823528012619711,
     -0.0002715535049406976, -0.0002612972747368730, -0.0002515504422622381,
     -0.0002422818100561459, -0.0002334624428418380, -0.0002250654786596873,
     -0.0002170659578164017, -0.0002094406677338749, -0.0002021680020697373,
     -0.0001952278326777815, -0.0001886013931030937, -0.0001822711723663212,
     -0.0001762208182154458, -0.0001704350487075080, -0.0001648995714392821,
     -0.0001596010096806170, -0.0001545268347029430, -0.0001496653037946782,
     -0.0001450054033513847, -0.0001405367966833558, -0.0001362497759839262,
     -0.0001321352181836653, -0.0001281845443040784, -0.0001243896820213317,
     -0.0001207430310901101, -0.0001172374315073718, -0.0001138661340326417,
     -0.0001106227730224293, -0.0001075013412038536, -0.0001044961664087304,
     -0.0001016018899597639, -9.881344666663848e-05, -9.612604625342037e-05,
     -9.353515612320406e-05, -9.103648531960727e-05, -8.862596968259718e-05,
     -8.629975793141954e-05, -8.405419885393399e-05, -8.188582922286139e-05,
     -7.979136264491039e-05, -7.776767911590563e-05, -7.581181527263900e-05,
     -7.392095531799700e-05, -7.209242248451593e-05, -7.032367114467914e-05,
     -6.861227935258276e-05, -6.695594188525318e-05, -6.535246375172267e-05,
     -6.379975411308126e-05, -6.229582050210114e-05, -6.083876354929943e-05,
     -5.942677188390642e-05, -5.805811745529562e-05, -5.673115107129907e-05,
     -5.544429827197867e-05, -5.419605535190363e-05, -5.298498574329073e-05,
     -5.180971648921543e-05, -5.066893503102890e-05, -4.956138610623419e-05,
     -4.848586885048161e-05, -4.744123407446022e-05, -4.642638170571123e-05,
     -4.544025832831121e-05, -4.448185491057220e-05, -4.355020463425631e-05,
     -4.264438084872285e-05, -4.176349513718177e-05, -4.090669549483590e-05,
     -4.007316459497840e-05, -3.926211815668640e-05, -3.847280339303486e-05,
     -3.770449755746724e-05, -3.695650652446618e-05, -3.622816353576778e-05,
     -3.551882787616078e-05, -3.482788374637667e-05, -3.415473913692704e-05,
   -3.349882473358014e-05, -3.285959292997932e-05, -3.223651687815934e-05,
     -3.162908955018882e-05, -3.103682290062104e-05, -3.045924701354645e-05,
     -2.989590934801346e-05, -2.934637399762555e-05, -2.881022093106919e-05,
     -2.828704540333513e-05, -2.777645722318233e-05, -2.727808023342566e-05,
     -2.679155163562077e-05, -2.631652152021920e-05, -2.585265228846071e-05,
     -2.539961817260500e-05, -2.495710471887216e-05, -2.452480839806060e-05,
     -2.410243609133709e-05, -2.368970473128469e-05, -2.328634087500548e-05,
     -2.289208033405335e-05, -2.250666780452707e-05, -2.212985651943396e-05,
     -2.176140794514504e-05, -2.140109140735752e-05, -2.104868385137585e-05,
     -2.070396951110494e-05, -2.036673963639096e-05, -2.003679225599426e-05,
     -1.971393187860604e-05, -1.939796930021926e-05, -1.908872131201864e-05,
     -1.878601055543114e-05, -1.848966522117309e-05, -1.819951891135433e-05,
     -1.791541041900638e-05, -1.763718353037717e-05, -1.736468686613530e-05,
     -1.709777369413381e-05, -1.683630175830461e-05, -1.658013314969961e-05,
     -1.632913410326743e-05, -1.608317490570012e-05, -1.584212972477525e-05,
     -1.584212972477525e-05};
static int numpoints = 401;
static float h = 0.2000000000000000;
static float hinv = 5.000000000000000;
     klo=(int)((x-xlo)*hinv);
     khi=klo+1;
```

```
if(x<xlo){
    return(ya[0]);
}else if(x>xhi){
    return(ya[numpoints-1]);
}
    xdiff0=(x-(xlo+h*klo));
    xdiff1=xdiff0-h;

    return( ya[klo] + (ya[khi]-ya[klo])*hinv*xdiff0 + y2a[khi]*xdiff0*xdiff1 )
        ;

}


float greg_sqrt(float xin) {
//Greg Steisberg's square function using quadratic interpolation)
// INPUT should only be BETWEEN 0 and 2. Doesn't work well when less than .01

   //int k,klo,khi;
     int klo,khi;
    float xdiff0, xdiff1;
    float x=xin;
static float xlo = 0.000000000000000;
//static float xhi = 1.000000000000000;

static double ya[401] = {0.000000000000000, 0.07071067811865475,
   0.1000000000000000, 0.1224744871391589, 0.1414213562373095,
   0.1581138830084190, 0.1732050807568877, 0.1870828693386971,
   0.2000000000000000, 0.2121320343559643, 0.2236067977499790,
   0.2345207879911715, 0.2449489742783178, 0.2549509756796393,
   0.2645751311064591, 0.2738612787525830, 0.2828427124746190,
   0.2915475947422650, 0.3000000000000000, 0.3082207001484488,
   0.3162277660168379, 0.3240370349203930, 0.3316624790355400,
   0.3391164991562634, 0.3464101615137755, 0.3535533905932738,
   0.3605551275463990, 0.3674234614174767, 0.3741657386773942,
   0.3807886552931954, 0.3872983346207417, 0.3937003937005906,
   0.4000000000000000, 0.4062019202317980, 0.4123105625617661,
   0.4183300132670378, 0.4242640687119285, 0.4301162633521313,
   0.4358898943540673, 0.4415880433163923, 0.4472135954999579,
   0.4527692569068708, 0.4582575694955840, 0.4636809247747852,
   0.4690415759823430, 0.4743416490252569, 0.4795831523312720,
   0.4847679857416329, 0.4898979485566356, 0.4949747468305833,
   0.5000000000000000, 0.5049752469181039, 0.5099019513592785,
   0.5147815070493500, 0.5196152422706632, 0.5244044240850758,
   0.5291502622129182, 0.5338539126015656, 0.5385164807134504,
   0.5431390245600107, 0.5477225575051661, 0.5522680508593630,
   0.5567764362830022, 0.5612486080160912, 0.5656854249492380,
   0.5700877125495690, 0.5744562646538028, 0.5787918451395113,
   0.5830951894845301, 0.5873670062235365, 0.5916079783099616,
   0.5958187643906492, 0.6000000000000000, 0.6041522986797286,
   0.6082762530298219, 0.6123724356957945, 0.6164414002968976,
   0.6204836822995429, 0.6244997998398398, 0.6284902544988268,
   0.6324555320336759, 0.6363961030678927, 0.6403124237432849,
   0.6442049363362563, 0.6480740698407860, 0.6519202405202649,
    0.6557438524302001, 0.6595452979136459, 0.6633249580710799,
    0.6670832032063166, 0.6708203932499369, 0.6745368781616021,
    0.6782329983125268, 0.6819090848492928, 0.6855654600401044,
    0.6892024376045111, 0.6928203230275509, 0.6964194138592059,
    0.7000000000000000, 0.7035623639735145, 0.7071067811865476,
0.7106335201775947,
0.7141428428542850, 0.7176350047203662, 0.7211102550927979,
    0.7245688373094720, 0.7280109889280518, 0.7314369419163897,
    0.7348469228349535, 0.7382411530116700, 0.7416198487095663,
    0.7449832212875670, 0.7483314773547883, 0.7516648189186453,
    0.7549834435270749, 0.7582875444051550, 0.7615773105863908,
    0.7648529270389177, 0.7681145747868608, 0.7713624310270756,
    0.7745966692414834, 0.7778174593052023, 0.7810249675906654,
    0.7842193570679061, 0.7874007874011811, 0.7905694150420949,
    0.7937253933193772, 0.7968688725254613, 0.8000000000000000,
    0.8031189202104505, 0.8062257748298550, 0.8093207028119322,
    0.8124038404635960, 0.8154753215150046, 0.8185352771872451,
    0.8215838362577492, 0.8246211251235321, 0.8276472678623424,
    0.8306623862918074, 0.8336666000266533, 0.8366600265340756,
    0.8396427811873333, 0.8426149773176358, 0.8455767262643882,
    0.8485281374238570, 0.8514693182963200, 0.8544003745317531,
    0.8573214099741123, 0.8602325267042626, 0.8631338250816034,
    0.8660254037844386, 0.8689073598491384, 0.8717797887081347,
    0.8746427842267951, 0.8774964387392122, 0.8803408430829505,
    0.8831760866327847, 0.8860022573334675, 0.8888194417315589,
    0.8916277250063505, 0.8944271909999159, 0.8972179222463180,
    0.9000000000000000, 0.9027735042633894, 0.9055385138137416,
    0.9082951062292475, 0.9110433579144299, 0.9137833441248533,
    0.9165151389911680, 0.9192388155425117, 0.9219544457292888,
    0.9246621004453465, 0.9273618495495703, 0.9300537618869137,
    0.9327379053088815, 0.9354143466934853, 0.9380831519646859,
    0.9407443861113389, 0.9433981132056604, 0.9460443964212251,
    0.9486832980505138, 0.9513148795220224, 0.9539392014169457,
    0.9565563234854496, 0.9591663046625439, 0.9617692030835673,
    0.9643650760992956, 0.9669539802906858, 0.9695359714832658,
    0.9721111047611790, 0.9746794344808963, 0.9772410142846032,
    0.9797958971132712, 0.9823441352194250, 0.9848857801796105,
    0.9874208829065749, 0.9899494936611666, 0.9924716620639604,
    0.9949874371066200, 0.9974968671630001, 1.000000000000000,
    1.002496882788171, 1.004987562112089,
1.007472083980494,
1.009950493836208, 1.012422836565829, 1.014889156509222, 1.017349497468790,
    1.019803902718557, 1.022252415013044, 1.024695076595960,
    1.027131929208707, 1.029563014098700, 1.031988372027515,
    1.034408043278860, 1.036822067666386, 1.039230484541326,
    1.041633332799983, 1.044030650891055, 1.046422476822817,
    1.048808848170152, 1.051189802081432, 1.053565375285274,
    1.055935604097144, 1.058300524425836, 1.060660171779821,
    1.063014581273465, 1.065363787633126, 1.067707825203131,
    1.070046727951635, 1.072380529476361, 1.074709263010234,
    1.077032961426901, 1.079351657246145, 1.081665382639197,
    1.083974169433940, 1.086278049120021, 1.088577052853862,
    1.090871211463571, 1.093160555453772, 1.095445115010332,
    1.097724920005007, 1.100000000000000, 1.102270384252430,
    1.104536101718726, 1.106797181058933, 1.109053650640942,
    1.111305538544644, 1.113552872566004, 1.115795680221070,
    1.118033988749895, 1.120267825120404, 1.122497216032182,
    1.124722187920199, 1.126942766958464, 1.129158979063621,
    1.131370849898476, 1.133578404875463, 1.135781669160055,
    1.137980667674104, 1.140175425099138, 1.142365965879586,
    1.144552314225960, 1.146734494117972, 1.148912529307606,
    1.151086443322134, 1.153256259467080, 1.155422000829134,
    1.157583690279023, 1.159741350474320, 1.161895003862225,
    1.164044672682282, 1.166190378969060, 1.168332144554792,
    1.170469991071962, 1.172603939955857, 1.174734012447073,
    1.176860229593982, 1.178982612255160, 1.181101181101772,
    1.183215956619923, 1.185326959112970, 1.187434208703792,
    1.189537725337032, 1.191637528781298, 1.193733638631332,
    1.195826074310140, 1.197914855071094, 1.200000000000000,
    1.202081528017131, 1.204159457879230, 1.206233808181482,
    1.208304597359457, 1.210371843691020, 1.212435565298214,
    1.214495780149112, 1.216552506059644, 1.218605760695394,
    1.220655561573370, 1.222701926063748, 1.224744871391589,
    1.226784414638530, 1.228820572744451, 1.230853362509117,
    1.232882800593795, 1.234908903522847, 1.236931687685298,
    1.238951169336387, 1.240967364599086, 1.242980289465605,
    1.244989959798873,
1.246996391333993, 1.248999599679680, 1.250999600319680, 1.252996408614167,
    1.254990039801113, 1.256980508997654, 1.258967831201417,
    1.260952021291849, 1.262933094031509, 1.264911064067352,
    1.266885945931993, 1.268857754044952, 1.270826502713883,
    1.272792206135785, 1.274754878398196, 1.276714533480370,
    1.278671185254442, 1.280624847486570, 1.282575533838066,
    1.284523257866513, 1.286468033026861, 1.288409872672513,
    1.290348790056394, 1.292284798332008, 1.294217910554478,
    1.296148139681572, 1.298075498574717, 1.300000000000000,
    1.301921656629154, 1.303840481040530, 1.305756485720060,
    1.307669683062202, 1.309580085370879, 1.311487704860400,
    1.313392553656370, 1.315294643796590, 1.317193987231949,
    1.319090595827292, 1.320984481362291, 1.322875655532295,
    1.324764129949177, 1.326649916142160, 1.328533025558642,
    1.330413469565007, 1.332291259447423, 1.334166406412633,
    1.336038921588739, 1.337908816025965, 1.339776100697426,
    1.341640786499874, 1.343502884254440, 1.345362404707371,
    1.347219358530748, 1.349073756323204, 1.350925608610630,
    1.352774925846868, 1.354621718414407, 1.356465996625054,
    1.358307770720613, 1.360147050873544, 1.361983847187624,
    1.363818169698586, 1.365650028374766, 1.367479433117734,
    1.369306393762915, 1.371130920080209, 1.372953021774598,
    1.374772708486752, 1.376589989793620, 1.378404875209022,
    1.380217374184226, 1.382027496108525, 1.383835250309805,
    1.385640646055102, 1.387443692551161, 1.389244398944981,
    1.391042774324356, 1.392838827718412, 1.394632568098135,
    1.396424004376894, 1.398213145410956, 1.400000000000000,
    1.401784576887619, 1.403566884761820, 1.405346932255520,
    1.407124727947029, 1.408900280360537, 1.410673597966589,
    1.412444689182554, 1.414213562373095};
```

```
static double y2a[401] = {0.000000000000000, -828.4271247461900,
    -136.2966948437271, -70.55236082016594, -45.08684654082272,
    -32.02658045281394, -24.26818333318854, -19.21315841012812,
    -15.70192610677369, -13.14541923899103, -11.21546305644340,
    -9.716079080924210, -8.523697716497361, -7.556919490032663,
    -6.760155613916889, -6.094278481760210, -5.531029087798557,
    -5.049540198221543, -4.634102185722600, -4.272685601194307,
    -3.955939296681041, -3.676495768161635, -3.428479888470942,
    -3.207155264227251, -3.008665560274974, -2.829842527463095,
    -2.668061640948011, -2.521132223206597, -2.387212882323619,
    -2.264745765098902, -2.152404953948968, -2.049055608788205,
    -1.953721352229151, -1.865558036598932, -1.783832493926107,
    -1.707905207620323, -1.637216093758328, -1.571272765336082,
    -1.509640792219932, -1.451935575188521, -1.397815533053454,
    -1.346976363994564, -1.299146190240072, -1.254081432867646,
    -1.211563292877746, -1.171394737976559, -1.133397913082848,
    -1.097411907164413, -1.063290821100569, -1.030902090619621,
    -1.000125026255903, -0.9708495385862808, -0.9429750220624183,
    -0.9164093751645713, -0.8910681380136494, -0.8668737314043702,
    -0.8437547838980454, -0.8216455352527058, -0.8004853064891279,
    -0.7802180281002422, -0.7607918191676597, -0.7421586111555620,
    -0.7242738110035938, -0.7070959988442193, -0.6905866563156048,
    -0.6747099219439600, -0.6594323705066950, -0.6447228137940175,
    -0.6305521202455953, -0.6168930516291043, -0.6037201147490723,
    -0.5910094267358436, -0.5787385924427281, -0.5668865927077594,
    -0.5554336824142848, -0.5443612973876623, -0.5336519691589982,
    -0.5232892469653185, -0.5132576262001356, -0.5035424827570791,
    -0.4941300126448658, -0.4850071764930446, -0.4761616484158903,
    -0.4675817688348785, -0.4592565010153030, -0.4511753908742470,
    -0.4433285297866788, -0.4357065202364999, -0.4283004439464825,
    -0.4211018323285735, -0.4141026391013540, -0.4072952148104207,
    -0.4006722831739744, -0.3942269190862713, -0.3879525281011238,
    -0.3818428273349987, -0.3758918276974825, -0.3700938172190853,
-0.3644433455907141,
-0.3589352096278191, -0.3535644397190112, -0.3483262871384334,
    -0.3432162121796972, -0.3382298729914795, -0.3333631151525290,
    -0.3286119618839394, -0.3239726048387398, -0.3194413954843428,
    -0.3150148369424061, -0.3106895764060086, -0.3064623979120989,
    -0.3023302155868984, -0.2982900672850665, -0.2943391085500744,
    -0.2904746069898318, -0.2866939368861838, -0.2829945741766866,
    -0.2793740916762206, -0.2758301545680015, -0.2723605161381032,
    -0.2689630137786553, -0.2656355651153847, -0.2623761644482410,
    -0.2591828793141455, -0.2560538472251568, -0.2529872726286793,
    -0.2499814239631521, -0.2470346309091371, -0.2441452817647511,
    -0.2413118209187988, -0.2385327465459493, -0.2358066082686068,
    -0.2331320051052184, -0.2305075833608501, -0.2279320347264853,
    -0.2254040944249521, -0.2229225394523300, -0.2204861869046180,
    -0.2180938923848162, -0.2157445484707306, -0.2134370832918005,
    -0.2111704591034598, -0.2089436710028810, -0.2067557456708609,
    -0.2046057401172693, -0.2024927405974704, -0.2004158614776760,
    -0.1983742441780119, -0.1963670561910821, -0.1943934901116325,
    -0.1924527627084057, -0.1905441140692684, -0.1886668067174743,
    -0.1868201248655940, -0.1850033735784070, -0.1832158780801226,
    -0.1814569830282938, -0.1797260518276996, -0.1780224659975179,
    −0.1763456245229552, −0.1746949432646080, −0.1730698544033512,
    −0.1714698058506903, −0.1698942607454410, −0.1683426969267817,
    −0.1668146064681683, −0.1653094951814535, −0.1638268821735342,
    −0.1623662994187922, −0.1609272913349888, −0.1595094143858053,
    −0.1581122366767215, −0.1567353376108468, −0.1553783075114443,
    −0.1540407472777613, −0.1527222680652859, −0.1514224909526793,
    −0.1501410466309136, −0.1488775751345980, −0.1476317255200141,
    −0.1464031556008824, −0.1451915317085548, −0.1439965283855924,
    −0.1428178281925871, −0.1416551214195036, −0.1405081059013822,
    −0.1393764867607669, −0.1382599762045005, −0.1371582933351977,
    −0.1360711639186716, −0.1349983202092450, −0.1339395007771136,
    −0.1328944502843041, −0.1318629193658436, −0.1308446644210368,
    −0.1298394474558149,
−0.1288470359561698,
−0.1278672026860936, −0.1268997255876589, −0.1259443876056030,
    −0.1250009765751984, −0.1240692850656920, −0.1231491102493651,
    −0.1222402538369860, −0.1213425218416159, −0.1204557245748364,
    −0.1195796764874903, −0.1187141960312285, −0.1178591056017349,
    −0.1170142314088144, −0.1161794033844288, −0.1153544550790941,
    −0.1145392235658096, −0.1137335493894141, −0.1129372763864670,
    −0.1121502517097737, −0.1113723256796765, −0.1106033516883541,
    −0.1098431861935413, −0.1090916885647787, −0.1083487210774480,
    −0.1076141487675902, −0.1068878394395750, −0.1061696635472552,
    −0.1054594941530929, −0.1047572068246838, −0.1040626796532177,
    −0.1033757931147267, −0.1026964300331035, −0.1020244755502228,
    −0.1013598170616801, −0.1007023441236697, −0.1000519484482618,
    −0.09940852385941987, −0.09877196616364531, −0.09814217323506599,
    −0.09751904481918428, −0.09690248262268783, −0.09629239017020869,
    −0.09568867282335779, −0.09509123769049842, −0.09449999365049633,
    −0.09391485125333836, −0.09333572268198133, −0.09276252178592800,
    −0.09219516395452580, −0.09163356614139495, −0.09107764681792770,
    −0.09052732590863578, −0.08998252481152426, −0.08944316631787456,
    −0.08890917461122377, −0.08838047523409344, −0.08785699503219621,
    −0.08733866216273384, −0.08682540604906375, −0.08631715734423127,
    −0.08581384792290446, −0.08531541083183099, −0.08482178031705408,
    −0.08433289171244168, −0.08384868149330441, −0.08336908723177988,
    −0.08289404755695168, −0.08242350211935433, −0.08195739164411338,
    −0.08149565782463247, −0.08103824332664859, −0.08058509181537957,
    −0.08013614785276557, −0.07969135696925649, −0.07925066555638297,
    −0.07881402092989030, −0.07838137123984593, −0.07795266550291298,
    −0.07752785358583811, −0.07710688613571647, −0.07668971462571110,
    −0.07627629131167908, −0.07586656921166506, −0.07546050209771303,
    −0.07505804449338413, −0.07465915163616298, −0.07426377947074964,
    −0.07387188465566102, −0.07348342452120844, −0.07309835707588738,
    −0.07271664095568377, −0.07233823550574953, −0.07196310064109934,
    −0.07159119693251981,
−0.07122248553408575
−0.07085692824665990, −0.07049448738788155, −0.07013512592207667,
    −0.06977880731268495, −0.06942549563420539, −0.06907515548255490,
    −0.06872775196247000, −0.06838325074905356, −0.06804161799767462,
    −0.06770282039173534, −0.06736682509430859, −0.06703359976614730,
    −0.06670311252940672, −0.06637533200584073, −0.06605022725912123,
    −0.06572776778581053, −0.06540792359109676, −0.06509066503302845,
    −0.06477596296949158, −0.06446378864666674, −0.06415411372628756,
    −0.06384691028937642, −0.06354215079263228, −0.06323980812971534,
    −0.06293985553008506, −0.06264226663342642, −0.06234701544639986,
    −0.06205407633695206, −0.06176342403565323, −0.06147503363427129,
    −0.06118888056064051, −0.06090494058594152, −0.06062318982639042,
    −0.06034360472640090, −0.06006616205485751, −0.05979083887823287,
    −0.05951761263118987, −0.05924646098436408, −0.05897736198252280,
    −0.05871029391806866, −0.05844523540860203, −0.05818216534164693,
    −0.05792106288987359, −0.05766190752704522, −0.05740467897202055,
    −0.05714935723595178, −0.05689592258832740, −0.05664435555548377,
    −0.05639463692201235, −0.05614674774719689, −0.05590066930408683,
    −0.05565638313162224, −0.05541387102239714, −0.05517311497724402,
    −0.05493409724168839, −0.05469680031842287, −0.05446120687925950,
    −0.05422729987990081, −0.05399506246117855, −0.05376447797801289,
    −0.05353553000741180, −0.05330820235183757, −0.05308247898405296,
    −0.05285834410340261, −0.05263578210013092, −0.05241477758210698,
    −0.05219531530376580, −0.05197738027261945, −0.05176095762061594,
    −0.05154603271494258, −0.05133259107838276, −0.05112061841115396,
    −0.05091010061799286, −0.05070102373214745, −0.05049337400922113,
    −0.05028713782881538, −0.05008230175743200, −0.04987885254767160,
    −0.04967677704584782, −0.04947606234647447, −0.04927669562452658,
    −0.04907866425425891, −0.04888195574374851, −0.04868655774537197,
    −0.04849245809064159, −0.04829964469992643, −0.04810810571404502,
    −0.04791782933394002, −0.04772880395515031, −0.04754101809395603,
    −0.04735446039455202, −0.04716911964858757,
−0.04698498476223645, −0.04680204478475328, −0.04662028889317278,
    −0.04643970637659106, −0.04626028665625875, −0.04608201929219941,
    −0.04590489393211623, −0.04572890037143288, −0.04555402849762802,
    −0.04538026835040695, −0.04520761003037873, −0.04503604380820735,
    −0.04486556002427785, −0.04469614913136286, −0.04452780173182812,
    −0.04436050846499556, −0.04436050846499556};


static int numpoints = 401;
static float h = 0.005000000000000000;
static float hinv = 200.0000000000000;
    klo=(int)((x−xlo)*hinv);
    khi=klo+1;
if(khi>=numpoints){
    khi=numpoints−1;
    klo=khi−1;
}else if(klo<0){
    klo=0;
    khi=klo+1;
}
    xdiff0=(x−(xlo+h*klo));
    xdiff1=xdiff0−h;
    return( ya[klo] + (ya[khi]−ya[klo])*hinv*xdiff0 + y2a[khi]*xdiff0*xdiff1 )
        ;

}

float anoop_asin(float xin)  {
 static float poly[5] = {0.000004763208132,  0.999652198116539,
    0.005859820836040, 0.129800591883359, 0.093780757121448};
 float x[5]={1.0,1.0,1.0,1.0,1.0};
 float asin;// time;
 int i;

 asin = poly[0];
// time = mb_clock_get_execution_time();

 for (i=1; i<5 ; i++){
 x[i] = x[(i-1)]*xin;  //can avoid the first multiplication here (its
    justmultiply by 1)
 asin = asin + x[i]*poly[i];
 }

// mb_io_set_float(ID_E_TEST5,(mb_clock_get_execution_time()-time));
 return(asin);

}
```