Create Performance Tests –
Beyond "Open Interface"
Terry L. Grant
NASA, Ames Associate
tgrant@mail.arc.nasa.gov

Create Performance Tests - Beyond "Open Interface"

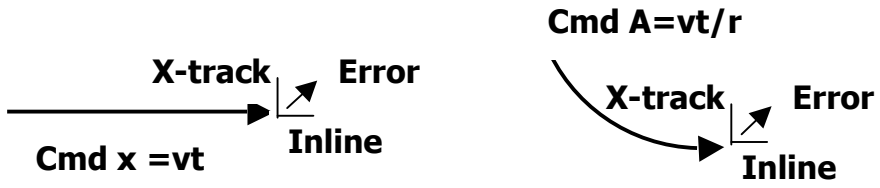
Introduction

During robot design and development, experienced team members could, until this year, start with an understanding of how the fast the robot would move using dead-reckoning with a certain accuracy. For example if you commanded a black motor to "move relative position" at a speed of 400, using a certain size wheel would move the robot 50 cm +/- 1cm, but if you pushed it to the maximum speed, the position error would increase because of slippage; likewise, turns required slower motion to repeat a rotation with good accuracy. However, now that we have the Create platform, the speed limits and accuracy conditions are unknown and some test experiments are needed in order to ease the development of robotics movement. These constraints are partially due to the larger mass of the Create, new motors and rotation sensors, but are also due to the delays associated with sending commands and sensor quantization associated with the Create processor. The delays are particularly large compared to XBC timing and are variable because the serial communications from the XBC-to-Create are asynchronous (each controller has an independent reference clock). This report focuses on experiments and code to define constraints on commanding rotation and linear motion with the Create.

Testing Motion Accuracy

Several factors conspire to cause robot motion errors: (1) The robot usually doesn't know exactly where it is on the playing field, so its position error after moving is the sum of the starting error and the error created while moving. (2) For straight motion, given two motor steering as with the Create, the movement error is due to measurement error on the wheel rotation, wheel radius error, and differential rotation error; together they result in an 'inline' and 'cross-track' error in the final relative position. (3) For turning motion all the errors in straight motion are multiplied by an additional error in the radius of the turn. (4) For both straight and turning motion there are quantization or resolution errors in the commanded motion. (5) Similarly, and especially for the Create controller, sample time resolution leads to errors (see below). (6) Finally, for both straight and turning motion, there are slippage effects associated with Newton's Laws and the friction between the wheels and the surface.

In the general environment slippage on uneven surfaces can dominate a vehicle or robot's position error, but on a Botball game board the surface is smooth flat and uniform, so we don't have large uncertainties due to slippage except in cases of high acceleration. One reason for testing the Create performance is to determine what speed is achievable without significant slippage or additional error in the resulting position.



The diagrams above illustrate the difference between the commanded move or rotate and the actual move in terms of a position error.

Test Objectives, Plans and Procedures

When we get down to testing the Create, as in most other development, it is best to start with a general test plan, then define a step-by-step procedure, and then write a little code to execute the procedure, or part of the procedure. At this point try it out, evaluate if the results are similar to those expected in the plan, and after some evaluation, revise the test and so on. We usually make changes in the test parameters, the number of test runs, the code, and hopefully lead ourselves to a better understanding of what is possible, how the results relate. For the Create tests we hope to find out what the resultant error will be when moving the Create according to particular sequences of commands. For each test development cycle we should ask why certain results occurred, and try to predict how they will change as a test parameter or sequence is changed. In order to do this successfully, we must be careful to change only one thing at a time, otherwise we create multiple possibilities as causes for the new results.

Test communication is important to assure all members of your team have a similar understanding of the test objective, the plan and the current procedure. This must be written down if more than one person is involved. Even if only one person is tasked with the test objective, having a written plan and procedure will be helpful to understanding new effects as they are uncovered. The test program code controls the test and therefore it must be changed and documented along with the results. Here are four coding rules which will help to make the tests easier to understand and to communicate to others:

1. Always add a comment line at the top describing the objective, and add comments at the top as code changes.
2. Always put in a 'printf' statement to identify the code and then a 'wait-for-button' function following, so the code is self identifying on the display screen.
3. As you change the code, do a 'Save As' to change the file name – usually just by adding a version #.
4. Append the same version # to the title display, so you will be able to read the code version actually loaded into the controller.

Here's an early version of a test code that ran to measure the average time to send commands to the Create to measure distance.

```
// Test to measure the time to sample a Create parameter
/* Read a 'gc_distance' sample 10 times in a loop and measure the time to record the
samples. Display samples and the time for the 10 samples
after gathered ( so as to not slow-down the sampling with a 'printf') */
int sampl[11]; // an array to hold samples

void main(){
  int i;
  long st;
  display_clear();
  printf("test sample period \n press Down to exc \n");
  create_connect();
  while(!down_button()){} //wait for 'down' button
  create_distance();// set ref. distance
  create_drive_straight(100);
  st=mseconds(); //start time reference
```

```
  for(i=0;i<10;i++){
     create_distance();//msleep(15L);
     sampl[i]=gc_distance;
   }
   sampl[10]=(int)(mseconds()-st);
   printf("\n i, dist\n");
   for(i=0;i<10;i++){
     printf(" %d  , %d  \n",i,sampl[i]);
   }
   printf("time = %d , ms" ,sampl[10]);
   create_disconnect();
   printf("\n END");
}
```

Data Evaluation

When this test code is run the display shows ten distance samples and a time of ~150 milliseconds (ms). A total change of only 5 mm is indicated, which is less than expected. At 100 mm/sec, 0.150 sec should show a change of 15 mm, with at least an increment of 1 mm/sample. However, we know from basic physics that the robot must accelerate from stopped to the commanded speed, so perhaps it just takes more than 150 ms to get up to speed. Thus the next test is to add 10 ms to the sample loop. Multiple runs of this code raises new questions: (1) Why does adding 10 ms per sample loop, which had an average sample time of 15 ms, cause sets of 10 samples to range from 289 to 268 ms? (2) Why does the distance read over the last 5 samples create changes of 6, 7 or 8 mm, when samples spanning more than 0.1 sec should show at least 10 mm change?

One could assume that the calibration is wrong, but other tests show that the XBC clock has a timing uncertainty of ~ 1 ms, and a sample loop of 10 analog values on the XBC takes about 15 ms. The answer to both observations involves understanding the quantization of the sample times, sync delays between the XBC and the Create controllers, and quantization of the measurement (distance). The 10 ms delay usually causes the readings to skip a sample adding 15 ms, but not always, so the average of the ten samples is more than 250 ms, but less than 300 ms. Furthermore, from the Create interface description, the individual readings measure the *change* in distance since the last data sample. Lets assume that the wheels are accurately commanded to move at 100 mm/s or 1.5 mm/0.015sec, the data sample period. If the period between samples is small, such as the minimum data sample period of 15 ms, and on average only 1 or 2 mm increments occur between samples, the measurement will be quite erratic, and less than or equal to the maximum. The value gc_distance recorded by the XBC is just the accumulation of these small, truncated distance measures (see the function description of 'create_distance()' in the library file "createlib.ic". For a fixed robot speed, adding more samples results in reading fewer increments of change for each sample, and can provide a more biased estimate on the low side of the actual distance moved.

Extended Tests

To better understand this effect an expanded version of the original test code was written to gather five sets of data, stopping after each set to display the last ten samples and the time lapse for gathering 20 samples. Then a few different delays were added to the sample loop to see the effect of reading more distance increments between samples.

```
//Test sample times and distance accuracy
// -1 w. drive straight during velocity samples
```

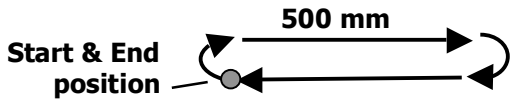
```
// -2 w. drive straight during distance samples
// -2-d add 10 ms delay to loop, clear distance to start & add sim
// -2-da-1 increase runs to 20 samples, display the last 10 per run
// -2-da-2 increase loop delay to 25 ms
// -2-da-3 increase loop delay to 28 ms

int sampl[21][5];//2-da 2-dim. array for upload

void main(){
   int i,j;
   long st;// start time ref
   //iROBOinit(BIGEMPTYWORLD); // initialize w/world - out for array
   display_clear();
   printf("test sample period 2-da-3\n press Down to exc");
   create_connect();
   for(j=0;j<5;j++){
      while(down_button()){}// debounce
      while(!down_button()){}
      create_distance();msleep(28L);
      gc_distance=0;
      create_drive_straight(100);
      st=mseconds();
      for(i=0;i<20;i++){
         create_distance();msleep(28L);
         sampl[i][j]=gc_distance;
      }
      sampl[20][j]=(int)(mseconds()-st);
      create_stop();
      // display recorded samples for each set here
      display_clear();
      printf("i, dist\n");
      for(i=10;i<20;i++){
         printf(" %d  , %d  \n",i,sampl[i][j]);
      }
      printf("time = %d ms set # %d" ,sampl[20][j],j+1);
   }
   create_disconnect();
   printf("\n END");
}
```

The data show about 400 ms is required to get up to the commanded speed, and that 28 ms gives the best control of the sample periods – at ~45 ms per sample. If one wants apply this understanding to move forward or backward a fixed distance, a while loop can be used similar to that shown in the Botball tutorial (slide115), and adjusting the loop delay for best accuracy. For example at 100 mm/s with a delay of 28 ms, a distance can be commanded with an inline accuracy of ~ 10mm or better. For moves at higher or lower speeds the delay should be tailored to provide about 4 mm or more increments per sample if possible. In tests that the author performed, the cross track error seemed to be dominated by the angular orientation of the robot as it was positioned. For rotation orientation, the round Create structure poses new challenges. Adding a long straight edge or two markings on the Create body are needed to align the robot to the edges or points on the field.

Testing Distance and Angle Accuracy

Similar issues apply to making accurate turns, and since the parameter measured is *degrees* of

rotation, let us look at how many degrees are covered in a minimum sample period. If the robot is spinning (turning about its center) we need to find the radius (r) which is half the distance between the wheels, or 129 mm. From the geometry of a circle we know that circumferential distance around a half circle (180 degrees) is pi*r =3.14*129
Thus 180 deg = 405 mm, and 1 deg = 2.25 mm
If the speed is 100 mm/s, each 15 ms gives a change of only 1.5 mm so that a minimal sample period reads only 0.67 degrees change, and because the angle is quantized in degrees, we'll need at least 4 periods, or 60 ms, to establish a good direct angular measurement and therefore the angular resolution by direct sensor reading is limited to ~ 3 deg. Somewhat higher angular resolution can be achieved using the distance measure and converting. The game objective in commanding accurate turns is usually to arrive at a target position with minimal error after following a path requiring both turns and straight distances. Therefore a test has been devised which eases the measurement of the error for such a path with two two 180 deg turns and two straight segments to return to the starting position.



Since the total angle change is 360 degrees the initial orientation of the robot doesn't affect the error measured, and with a total distance of 1000 mm, the resulting cross-track error can be directly converted to angular error: For small angles, A, the sinA = A, in radians, therefore the cross-track error, XT, in mm, divided by 1000 and multiplied by 57.3 deg/rad is the angular error, in degrees. A test code was written which makes the robot follow this path, and allows interactive adjustment of the turns to minimize XT. Multiple runs (5 in the example code) with measurement of XT will provide data on the angular error while the inline error gives distance accuracy. The measured angle and distance data, along with the battery voltage can then be uploaded to a spreadsheet for analysis with the physical measurements of the XT values.

```
//Test two wheel turn and distance for accuracy & angle calibration 6/5/08 TLG
// -2 use functions, setup initial angle via buttons
// -3 left turn only
// -4 use timed turns & 'create_full' & battery capacity output
// -1L-2 use 'create_angle' and total angle instead of timed rotation
// -1L-3-2 use XBC3 [w. USB} print set # (j+1)
// -1L-4a use 2-dim array to record data
int j, ar[6][5]; //array for recording data

void main(){
  int ang=172, dist=500, sp=100; //nominal total_angle for 180 deg, distance, & speed
  // iROBOinit(BIGEMPTYWORLD); // initialize w/world – take out for robot tests
  display_clear();
  printf("Angle meas-2wheel\n -left turn CCW v1L-4a\n");
  // turn 180 deg = 12.9 cm radius * pi= 40.5 cm
  create_connect();
  msleep(100L);
  for(j=0;j<5;j++){
    while(!down_button()){ //wait for down button
      display_set_xy(0,5);
      if(left_button())ang-=1;
      if(right_button())ang+=1;
      printf("deg = %d vel = %d\n",ang,sp);
      msleep(100L);
    }
```

```
      create_full();//move outside test loop?
      turn_left(ang,sp);ar[0][j]=gc_total_angle;
      go_straight(dist,2*sp);ar[2][j]=gc_distance; // dist speed set to 200 mm/s
      turn_left(ang,sp);ar[1][j]=gc_total_angle;
      go_straight(dist,2*sp);ar[3][j]=gc_distance;
      create_battery_charge();msleep(50L);
      ar[5][j]=gc_batt_voltage;
      printf("\nBattery voltage / set # =\n %d / %d ",ar[5][j],j+1);
      printf("\n ang   - dist  - ang  - dist \n");
      printf(" %d   - %d   - %d   - %d", ar[0][j],ar[2][j],ar[1][j],ar[3][j]);
      sleep(4.);
   }
   create_disconnect();
   printf("\n END");
} //A similar program can test and calibrate right-hand turns.

void turn_left(int deg,int vel){ // ang ~ deg
   create_angle();msleep(30L);//set ref angle
   gc_total_angle =0;
   create_spin_CCW(vel);//turn Counterclockwise, two wheels,
   while(gc_total_angle<deg){create_angle();msleep(30L);}
   create_stop();msleep(30L);
   create_angle();msleep(30L);
}

void go_straight(int dist,int vel){// vel in mm/s
   create_distance();msleep(30L);//set ref dist
   gc_distance=0;
   create_drive_straight(vel);
   while(gc_distance<dist){create_distance();}
   create_stop();msleep(30L);
}
```

Test data show that ~40 mm of cross-track accuracy, or 2.3 deg, can be achieved, along with 10-20 mm of inline accuracy, and it is unaffected by battery voltage, if above 13v. Developers should be warned that other parallel processing in the XBC, such as the vision system will add a variable delay in sampling Create parameters, thus will add error (try adding it to the test code). Also note that although test code can be tried on the simulator, it does not match the response of an actual Create very well, and thus can't be used to calibrate angle or measure errors.

Conclusions

We have shown that the Create motion must be tested to understand how well it can be driven from straight or spin commands. Because of slow data updates, measurement quantization, and sync delays between the XBC and the Create controllers, poor accuracy of motion can result. Care must be taken to reset references for distance or angle by sending read commands, and to allow for truncated measures if the parameters are sampled before much change can occur in the sensor values. The program code listed should help to record test data to assure that the accuracy is good enough for the robot move objectives.