7-9-2019

Mathematics and Programming Exercises for Educational Robot Navigation

Ronald I. Greenberg
Loyola University Chicago, Rgreen@luc.edu

Mathematics and Programming Exercises for Educational Robot Navigation
Ronald I. Greenberg
rig@cs.luc.edu, Loyola University Chicago

Mathematics and Programming Exercises for Educational Robot Navigation

Abstract

This paper points students towards ideas they can use towards developing a convenient library for robot navigation, with examples based on Botball primitives, and points educators towards mathematics and programming exercises they can suggest to students, especially advanced high school students.

1 Introduction

This paper will develop mathematical and programming exercises for navigating with a typical educational robot, such as those used in the Botball® educational robotics program [7]. We consider a robot with two wheels that may be driven independently (forwards or backwards up to some maximum speed) and a third balance point such as a caster wheel or track ball (drawn as in Figure 1). This type of robot is generally referred to as a differential-drive robot [2, p. 11–12], [3, 1] or a two-driving wheel robot [8, p. 4]. We do not consider Swedish wheels [2, p. 89] that can roll in all directions (aka omni wheels as in a Botball kit experiment a few years ago) or even wheels that turn as in a typical modern automobile.

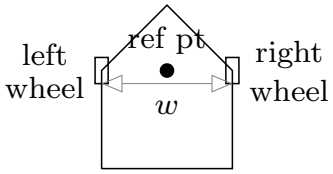

Figure 1: We will sketch our robot as shown, with a pointed front and a reference point centered between the two independently-driven wheels; there is also an assumed third balance point, for example a caster wheel in the back.

This model is a good fit for most educational robots, including the two types generally used in the Botball program: the iRobot Create® robot based on the Roomba® vacuum cleaning platform, and robots built from LEGO pieces and other provided parts in style used for many LEGO demobots over the years.

An important parameter for this type of robot is the distance w between the wheels, most often referred to as *track width*. In the Create robot this distance is fixed; in LEGO robots, there is some flexibility in the design. Intuitively, a small w allows for more nimble movement, but it can increase the likelihood of rollover problems.

In the limit, with $w = 0$, we would be essentially dealing with moving a single point around a two-dimensional field, but this paper assumes a positive track width, which is enough to put the options for robot motion into the relatively complicated category referred to as non-holonomic [2, p. 88–91].

While there exist deeper technical treatments of motion planning using inertial navigation (elementary overview in [2, p. 77–80] for example) and/or attending to complicated obstacle avoidance scenarios, this paper assumes use of dead reckoning [1] given a known terrain and manually preplanned paths.

It is true that there can be great value in using sensors such as a range finder to judge distance to a landmark, a reflectance sensor to detect colored tape boundaries on the driving surface, or even a camera, but dead reckoning is still a valuable technique. The sensors may be complicated to use or prone to failure (e.g., internal glitches, loosening of a mount, or loosening of an electrical connection); participants in Botball tournaments may especially notice how rarely the camera is used successfully. Thus teams may eschew some sensors entirely, or advanced programmers may wish to use them in conjunction with code that detects failures and can fall back on fail-safe alternative code. In any case, dead reckoning is likely to be helpful, even if only for initial rough positioning before using sensors, because this may be achievable at higher speed and lower power consumption.

One simple bit of sensor use that may be particularly helpful is to use built-in Botball functionality for testing motor rotation amounts, or Create travel distances or rotation angles (`get_motor_position_counter`, `get_create_distance`, or `get_create_total_angle`). Results may then be more robust under varying levels of battery charge, but good results may also be achievable using only timing delays as long as the battery is frequently recharged.

Section 2 discusses the more general mathematical context for motion planning, and then Section 3 returns to discussion of concrete programming implementations. Most of Section 3 remains generally applicable to any differential-drive robot, but we use Botball-provided functions at the lowest implementation level.

2 Mathematical Exercises

A preliminary analysis of the time required to complete a typical robot motion under different schemes of operation varying in programming difficulty is given in [4]. The set of schemes considered was later expanded [6] and formed into a worksheet suitable for high school students [5]. For convenient reference, Figure 2 reproduces a compact illustration of the six types of paths considered. The paths vary primarily according to whether motion is mostly rectilinear or more general and whether turns are based on rotations (wheels moving in opposite directions) or swings (one wheel stationary).

In each of the six cases illustrated, we are seeking to move the reference point on the robot from a starting position at coordinates $(0, 0)$ to a target position (x, y) with the robot pointing in the same direction at the end of the motion as at the beginning.

Educators may wish to access a free-standing PDF worksheet on comparing the time required to traverse the different types of paths, and LaTeX [10] source (using the tikz package [9]) that could be used to produce modified versions of the worksheet, both of which will be stored in the Loyola eCommons (`https://ecommons.luc.edu/cs_facpubs/231`) in association with [5]. A seventh variation that might be assigned is to consider moving the reference point along a single circular arc (instead of the two arcs in Figure 2f) without

[1] calculating one's current position by using a previously determined position and advancing that position based upon known or estimated speeds over elapsed time and course

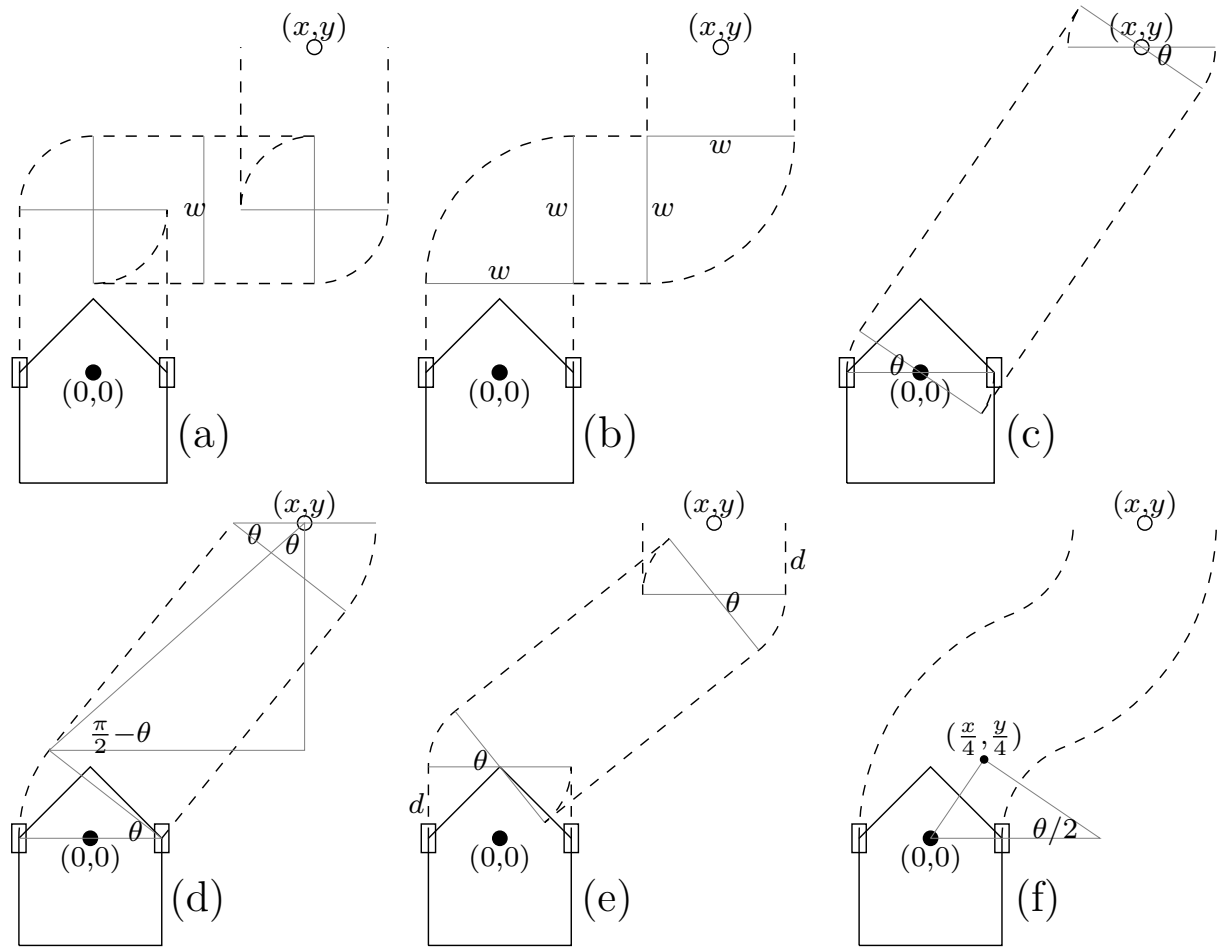

Figure 2: The paths of the robot wheels for the routing methods considered in [6, 5]. (a) and (b) for horizontal and vertical navigation, using rotations and swings, respectively. (c) and (d) for general navigation, using rotations and swings, respectively, and (e) and (f) for two other path types designed to avoid hitting a competition field boundary as may occur in (c). In (e), straight segments are added at the beginning and end of the path; these lengths are exaggerated for visual effect. In (f), we follow two mirror-image circular arcs.

worrying about the final robot orientation (direction). This seventh type of path would look as in Figure 3. (This could be a sufficient type of motion to contemplate in a case where the robot has a rotating effector anchored above the reference point.)

The proof is left to the reader, but the angle θ and the distance traveled by the reference point are actually the same in Figures 3 and 2f. Specifically, $\theta = 2\arctan(x/y)$, and the radius of the arc traversed by the reference point is $(x^2 + y^2)/(2x)$ in Figure 3 and $(x^2 + y^2)/(4x)$ for each of the two arcs in Figure 2f. The time to traverse the single-arc path is still a little less than for the two-arc path, since it is determined by arc lengths of the outer wheel, which is the one traveling at full speed, and this part of the analysis is also left to the reader.

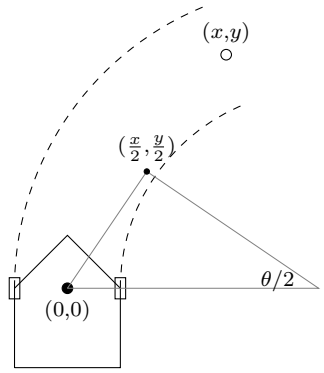

Figure 3: The path using one arc and not worrying about final orientation

```
void gostraight(float dist){ /* dist in millimeters */
  create_drive_direct(speed,speed); msleep(straightfit(dist)); create_stop();}
void rotate(float theta){ /* theta in positive/negative radians for right/left rotations */
  float dist=theta*w/2; /* computers arc length given rotation radius w/2 */
  if (dist>=0) {create_drive_direct(speed,-speed); msleep(rrotfit(dist)); create_stop();}
  else {create_drive_direct(-speed,speed); msleep(lrotfit(dist)); create_stop();}}
void swing(float theta){ /* theta in positive radians for right swings; negative for left */
  float dist=theta*w; /* computes arc length given swing radius w */
  if (dist>=0) {create_drive_direct(speed,0); msleep(rswingfit(dist)); create_stop();}
  else {create_drive_direct(0,speed); msleep(lswingfit(dist)); create_stop();}}
```

Figure 4: Example code for straight, rotation, and swing movements. `straightfit`, `rrotfit`, `lrotfit` `rswingfit`, and `lswingfit` incorporate linear regression results as suggested in a similar LEGO scenario in [6, Section 5].

3 Programming Exercises

Students may also find it to be an interesting exercise to create a library of program routines that can be used to send the robot along the paths discussed in Section 2. Most of this section is of general applicability, but we use the following bottom-level Botball functions (as well as alluding to a few others below):

- `create_drive_direct(left_speed,right_speed); /* drive Create wheels; speeds in mm/s */`
- `create_stop(); /* stop the Create movement */`
- `msleep(milliseconds); /* Wait before executing next command; does not stop movement. */`

Routines for the basic primitives could be programmed for a LEGO robot as described in [6, Figure 1 and Section 5] using `cmpc` and `gmpc` to measure motor rotations. Similarly, basic navigation primitives may be programmed for the Create as in Figure 4. (The variable `w` is assumed to be globally defined to represent the track width.)

This suggested code uses timing delays rather than the functions `get_create_distance` and `get_create_total_angle`, since other experience with Create programming seems to show that if the Create is moving too slowly or testing too frequently, changes in the return values of these functions may not be detected as needed to bring the motion to a stop. Successful testing was done with a LEGO robot in [5], where it was also suggested to teach students to use a spreadsheet for linear regression to determine how to convert distances to

```
int gotorectswing(float x, float y){
  float dx, dy, midy;
  dx=x-curx; dy=y-cury; /* find x & y changes from start & target */
  midy=(dy-w)/2; /* find y midpoint */
  gostraight(midy); swing(M_PI/2); gostraight(dx-w); swing(-M_PI/2); gostraight(midy);
  curx=x; cury=y; return(0);}
```

Figure 5: A routine to implement rectilinear paths with swings as in Figure 2b.

appropriate numbers of motor ticks. Here the conversion is from distances to appropriate timing delays. It is possible that when working with the Create robot, it would not be necessary to use so many different fitting functions as sketched here, since the Create robot might have more consistent behavior; in contrast, LEGO robots are likely to be not completely symmetrical in their construction, so that regression fitting for left rotation might be different from right rotation, for example. For the LEGO robot, one may also typically need to run the wheels at slightly different speeds to actually go straight; similar adjustments might be needed for the Create but would perhaps be less necessary.

One other detail before using the navigation primitives to construct paths as in Figure 2 is that mathematically simple analyses assume that the robot can instantaneously switch from, for example, going straight to rotating. To get reasonably reliable behaviors from the navigation primitives, it is probably sensible to terminate each one with a pause that allows movement to settle; this could be achieved by inserting a short `msleep` after each `create_stop` invocation in Figure 4. (In tests performed in conjunction with a LEGO robot in [6], it was sufficient to insert a single `msleep` of 100ms at the end of the `doticks` procedure there [5, Fig. 1].)

Once the primitives are in place for the LEGO and/or Create robot, a good programming exercise is to write routines to traverse each of the paths in Figure 2. For example, a routine to follow a path as in Figure 2b and update global variables indicating current position (`curx`,`cury`) could be written as in Figure 5 (after doing `#include <math.h>` to define `M_PI` for the value of π):

This solution assumes $x \geq w$, as well as $y \geq w$; it would not be difficult to also handle $x \leq -w$. It also would be good practice to follow the standard C programming practice of having the function return a non-zero value when the motion cannot be completed. Similarly implementing paths as in Figures 2a, c, and e is reasonably straightforward, and these are left as exercises for the reader.

A more involved case is implementing paths as in Figure 2d, where there is some complexity in determining the value of the angle θ. As shown in [6], the correct θ is where the following function evaluates to zero:

$$y \sin\theta - (x - w)\cos\theta - w \ .$$

An analytical solution is not evident, but the equation can be solved numerically, e.g., by the bisection method. This is a good opportunity to teach students how to program numerical root finding. The root finding could be done just for this particular function, but one can even teach advanced students how to use the interesting programming technique of passing a function to a general root-finding function. A solution is provided in Figure 6 (for y large

```
float genswingtheta(float theta){ return(dy*sinf(theta)-(dx-w)*cosf(theta)-w); }

int gotogenswing(float x, float y){
  int findroot(float low, float high, float (*fun)(float), float maxerr, float *result);
  float thetasoln, sx, sy; /* thetasoln, straightx, and straighty */
  dx=x-curx; dy=y-cury;
  if (findroot(-M_PI/2,M_PI/2,&genswingtheta,.001,&thetasoln))
    {printf("Couldn't find theta for generalized navigation with swings.\n"); return(1);}
  sx = dx-w+w*cosf(thetasoln); sy = dy-w*sinf(thetasoln);
  swing(thetasoln); gostraight(sqrt((sx)*(sx)+(sy)*(sy))); swing(-thetasoln);
  curx=x; cury=y; return(0);}

int findroot(float low, float high, float (*fun)(float), float maxerr, float *result){
  /* Assumes fun well-defined and non-zero at each of high and low */
  float highval, lowval, midval; highval = fun(high); lowval = fun(low);
  if (highval*lowval > 0) return(1); /* Maybe no zero in range */
  while (high-low > maxerr) {
    *result = (high+low)/2; midval = fun(*result);
    if (midval==0) return(0);
    if (highval*midval>0) {high=*result;highval=midval;} else {low=*result;lowval=midval;}}
  *result=high; if (fabsf(highval)>fabsf(lowval)) *result=low; return(0);
}
```

Figure 6: A routine to implement a straight path with swings at the end as in Figure 2d, using a general root finding procedure.

```
void arc(radius,theta){ float speedratio = (radius-w/2)/(radius+w/2);
  /* ref pt arcs right/left; radius in mm & theta in positive/negative radians */
  outerdist=theta*radius; /* computes arc length for outer wheel */
  if (theta>0) create_drive_direct(speed,speed*speedratio);
  else create_drive_direct(speed*speedratio,speed);
  msleep(arcfit(outerdist)); create_stop();
}
```

Figure 7: Example code to traverse a circular arc.

enough); this solution partially implements the practice of having functions return a non-zero value when they cannot complete successfully. (We also assume here, for simplicity, that `dx` and `dy` are declared globally.)

Finally, programming a path as in Figure 2f, like for Figure 2d, requires relatively substantial mathematics; in addition, we need another navigation primitive beyond those of Figure 4. This slightly more advanced primitive to follow a circular arc of specified radius through a specified angle could be programmed as in Figure 7. This time, we will assume that one linear regression function `arcfit` suffices for both rightward and leftward arcs.

The simplest way to use the circular arc primitive is as suggested at the end of Section 2 in a path of just one arc in which we do not worry about the robot's final orientation. Using the results stated at the end of Section 2, the core part of a procedure implementing this path would just be the call

```
arc((x*x+y*y)/(2x),atan2f(x/y));
```

An exercise left for the reader is to write a procedure to follow the path of Figure 2f, using two calls to the `arc` procedure.

4 Conclusion

This paper has presented a number of mathematical and programming exercises that students can carry out in an exploration of robot navigation. It has also sketched solutions to some of these exercises and has pointed to other sources with additional details and/or source files for generating student worksheets. Example solutions to programming exercises have extended as far down as showing how to complete implementations using pre-defined functions provided in the Botball program for control of the Create robot. These implementations have not been tested on a Create robot, but successful testing has been completed using analogs of the primitives in Figure 4 for a LEGO robot to implement paths as in Figures 2a–d.

Acknowledgments

The author is supported in part by National Science Foundation grants CNS-1738691, CNS-1543217, and CNS-1542971.

References

[1] Devin J. Balkcom and Matthew T. Mason. Time optimal trajectories for bounded velocity differential drive vehicles. *International Journal of Robotics Research*, 21(3):199–217, 2002.

[2] Mordechai Ben-Ari and Francesco Mondada. *Elements of Robotics*. Springer-Verlag, 2018.

[3] Hamidreza Chitsaz, Steven M. La Valle, Devin J. Balkcom, and Matthew T. Mason. Minimum wheel-rotation paths for differential-drive mobile robots. *The International Journal of Robotics Research*, 28(1):66–80, 2009. `https://doi.org/10.1177/0278364908096750`.

[4] Ronald I. Greenberg and Jeffery M. Karp. Motion planning for simple two-wheeled robots. In *Proceedings of the 2017 Global Conference on Educational Robotics (GCER)*, July 2017. `http://ecommons.luc.edu/cs_facpubs/182`.

[5] Ronald I. Greenberg and George K. Thiruvathukal. Exercises integrating high school mathematics with robot motion planning. In *Proceedings of 2019 IEEE Frontiers in Education Conference (FIE)*, October 2019. To appear.

[6] Ronald I. Greenberg, George K. Thiruvathukal, and Sara T. Greenberg. Integrating mathematics and educational robotics: Simple motion planning. In *Proceedings of the 10th International Conference on Robotics in Education, RiE 2019*, Advances in Intelligent Systems and Computing. To be published by Springer-Verlag.

[7] KISS Institute for Practical Robotics. Botball. `http://www.botball.org`, 2019. Accessed Jan. 22, 2019.

[8] J. P. Laumond, S. Sekhavat, and F. Lamiraux. Guidelines in nonholomic motion planning for mobile robots. In Jean-Paul Laumond, editor, *Robot Motion Planning and Control*, Lecture Notes in Control and Information Sciences 229, chapter 1, pages 1–53. Springer-Verlag, 1998.

[9] Till Tantau. CTAN: package pgf. `https://ctan.org/pkg/pgf?lang=en` accessed 6/19/19.

[10] The LaTeX Project. Latex - A document preparation system. `http://www.latex-project.org` accessed 6/19/19.