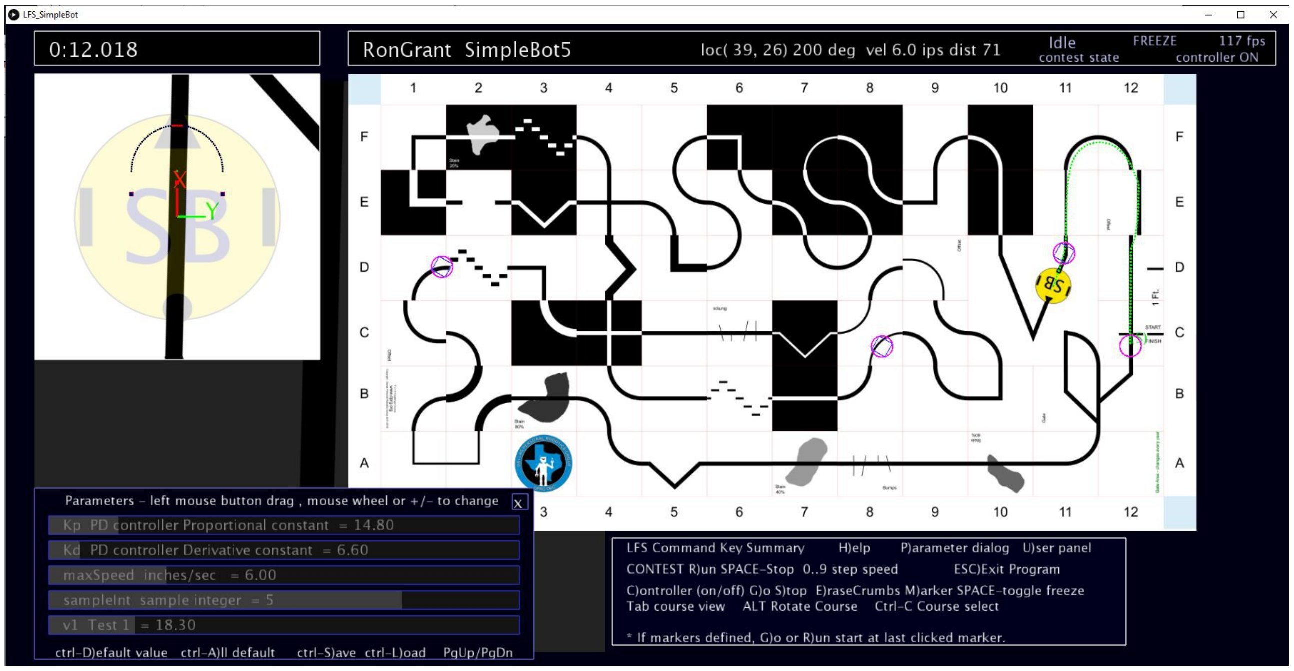
Line Follower Simulator (LFS) version 1.5
by Ron Grant

Sample Tricycle Controller
by Will Kuhnle

USER'S GUIDE
Nov 1, 2020

LFS is a Processing library and application program framework that uses your commands to drive a virtual robot on an image of a line following course. LFS renders a views of robot surrounding area area for sensor data acquisition and also renders a course view showing the progress of your robot. You define spot and line sensors and are presented the data as input to your controller program which then outputs, robot speed and turn-rate commands to the simulator. (http://github.com/ron-grant/LFS)

Simple line following control is achieved with a few lines of code, more complex line following courses such as Dallas Personal Robotics Group's Challenge Line Following Course, shown here, require more coding effort www.dprg.org.



```
0:12.018
RonGrant  SimpleBot5     loc( 39, 26) 200 deg  vel 6.0 ips dist 71     Idle           FREEZE      117 fps
                                                                        contest state          controller ON
```

```
Parameters - left mouse button drag , mouse wheel or +/- to change
Kp  PD controller Proportional constant  = 14.80
Kd  PD controller Derivative constant  = 6.60
maxSpeed  inches/sec   = 6.00
sampleInt  sample integer  = 5
v1  Test 1  = 18.30
ctrl-D)efault value   ctrl-A)ll default     ctrl-S)ave  ctrl-L)oad     PgUp/PgDn
```

```
LFS Command Key Summary        H)elp      P)arameter dialog   U)ser panel
CONTEST R)un SPACE-Stop  0..9 step speed             ESC)Exit Program
C)ontroller (on/off) G)o S)top  E)raseCrumbs M)arker SPACE-toggle freeze
Tab course view    ALT Rotate Course    Ctrl-C Course select

* If markers defined, G)o or R)un start at last clicked marker.
```

Note from Author

I have had an avid interest in computer graphics and simulation since the early days of personal computing (Apple II) when even low precision integer multiplication was a time-wise expensive proposition, and writing optimized machine language programs was the only practical way of getting real-time performance for the simplest simulation / animation. Such a stark contrast to today with billions of times the compute power available at fraction of the cost. Just as fun now as it was 40 years ago, but definitely living the dream.

Hope you find this tool useful in creating or experimenting with line following controllers.

– Ron Grant

Acknowledgments

Will Kuhnle for trying to impart mathematics, mechanical engineering concepts and control systems knowledge to me – to name a few. Will has also made a number of suggestions for LFS features including state save/restore. Also, thanks to Will for developing his DemoTrike controller, which uses a state space controller to track lines. The current rendition of his DemoTrike program recognizes some of the Challenge course features, but does not solve the entire course.

Doug Paradis for design of DPRG Novice,Advanced and Challenge Line Following Contest Courses https://github.com/dprg/Contests and also involvement in promoting the idea of virtual line following contests for DPRG.

Carl Ott for developing Robo-Realm based simulator https://www.dprg.org/simulation-success/ which uses advanced image processing algorithms to solve the Challenge Course. Carl's simulation inspired me to employ the simplest of image processing techniques to produce LFS with higher frame rate and a very simple interface for writing line follower controller programs.

Table of Contents

LFS Features

- Overall course view shows robot location and progress.
- Robot view shows location of user sensors.
- Cookie crumb trail shows how well robot is tracking the line or path.
- Interactive markers define starting location and heading.
- Run State markers store robot state and location for continuing from a defined location.
- Parameter Editor serves to allow real-time adjustment of select controller variables.
- Automated stopwatch and lap timer for lap based courses.
- Ability to throttle and freeze simulation step rate.
- Contest run mode that does not allow interference with robot, also does not permit inquiry of robot location - you must implement your own odometry if location and heading estimates are needed.
- Automated contest report with run time, distance driven, and a screenshot of the contest run.

User Controller Features

- Define your name and robot name for contest report.
- Define a list of available course images, and select an initial course.
- Custom acceleration/deceleration rates for drive and turning.
- Simple interface for user controller program, read your sensor values and inform LFS of target speed and turn rate.
- For lap timed courses, specify a number of laps to run.

Optional User Controller Features

- Define variable list, available for Parameter Editor, variable description, default value , and min/max ranges are specified.
- Set a custom time step (default 0.01667 sec) range 0.01 to 0.1 seconds.
- Custom Icon bitmap for robot can be specified , or select a LFS generated icon with ability to include your initials.
- Optionally user program can color sensors every controller update to show how data is being interpreted.

Introduction

LFS is a Processing library that provides a framework for developing a line following robot controller through real-time simulation.

An image of a line following course is used as the simulated robots world which is imaged with a down looking camera attached to the robot.

Robot camera images are not obscured by the robot allowing for placement of user sensors without any constraints. Of course sensor placement might need to be a consideration when if developing a real world robot.

The first model developed for LFS is that of a differential drive robot, e.g. , a two wheeled robot with caster. This model accepts target turn rate and forward speed as inputs to move the robot around in the world which is a 2D surface defined by the course image.

Will Kuhnle's included tricycle robot sketch is a variation that uses a steering wheel angle and wheel steering "speed" to move his simulated robot.

Currently DPRG line following courses use a 12 inch square tile as a common size to contain a course feature, e.g. line segment, 90 degree turn, etc.

LFS is informed of the course scale in "dots per inch" (DPI). A suggested scale is 64 DPI. At this scale a each 12" x 12" tile is 768 pixels. The current Challenge course including 6" borders is 7x13 tiles or 5376x9984 pixels. This is a very large bit map.

Sample course images are provided including advanced level features. Images of DPRG Novice, Advanced and Challenge level courses are provided with each example sketch, found in their data sub-folder.

Depending on the sensor view size (default 800x800 pixels), the area imaged by the simulators camera is about 1 square foot (12.5" x 12.5") where the center of the robot (midpoint between the drive wheels) is placed at the center of the robot viewport. Sensors including spot and line sensors can be defined which sample the screen image and report their data to the controller program which you write.

Note: Other image processing techniques could be applied to the “camera” image. Processing is capable of specifying a 3D perspective camera if you are interested in simulating a video camera, but for this implementation I avoided adding in this complexity.

LFS presents sensor data at a fixed stepTime (default value 0.01667 seconds, near 1/60th second) to the user's controller program (java method) which then provides turn rate and forward velocity values which are used by the simulator to update robots position and heading.

It is important to note this simulation step time is independent of real time. The simulation can be frozen or stepped as slow as about 1 step per second. Also, if CPU & GPU are fast enough, faster than real-time simulation is achievable and quite helpful.

Again, the results of the simulation are the same regardless of how fast it is able or allowed to run. Program code runs on the CPU on a single thread. Most of the image display portions are run using graphics processing unit (GPU) on board your video/graphics card.

LFS provides simple keyboard and mouse interface, with single key commands.

The robot can be driven manually by keyboard commands or by user controller which can be enabled / disabled by key command.

The robot can also be interactively positioned and its heading adjusted.

Newer and optional features (v1.3 – v1.5) include:

A parameter editor facility allows for real-time adjustment of floating point or integer variables. For example if your robot uses a fixed drive speed, you might make that variable available to the parameter editor (requires one line of code) to see the behavior of your robot as it runs, varying the speed.

Robot state can be saved and restored in the middle of a non-contest run, allowing for a continuation of a run from a point just before a failure to save time when working on a controller design. This facility does require effort to maintain robot controller state variables using a special class definition called RobotClass.

Support of display of a user robot icon file is provided. Also LFS provides a facility to generate an icon for you with a 2 or 3 character identifier, e.g. your initials.

LFS Environment - Processing (Java based IDE)

LFS uses the Processing programming environment which is available for free (or donation!) at www.processing.org.

Processing is Java based, but only a basic understanding of java is needed for a user to write a controller of reasonable complexity. The Arduino IDE is modeled after the Processing IDE, so if you have a little experience with Arduino, the Processing environment should present no difficulty.

A processing sketch (program) is composed of methods (similar to C functions) you write including a setup() method that is called one time then draw() which is called at video frame rate, e.g. typically at 60 times per second. In the case of LFS, a core sketch program is provided where you (the user) will write a few key methods, relevant to your controller, which can be derived from example sketches bundled with the LFS library.

The example sketches separate different logical groups of program code into separate notebook tabs. Each tab is stored in a separate (.pde) file, which are combined into a single file at run (compile and run) time by Processing. Also, you can add more program tabs if you need them to help organize your program.

For very complex programs, beyond say your line follower controller, other options exist when using processing, for example you can create pure Java code that is independent of processing or can write classes that are passed a reference to the class called PApplet which all your normal processing code including classes you may write exist. Other programming environments such as Eclipse can be used to create Java programs that are able to use the processing run time functionality (importing processing.core library) and also then in this case import the LFS library, LineFollowerSim.

Processing Help allows easy access to information about the Language including their Reference page (https://processing.org/reference/) which is quite handy.

The included debugger can be quite helpful allowing you to set break points and monitor variables.

Tweak mode allows adjusting program constant values defined typically in methods that are called repetitively (not global constants). I have found the tweak functionality useful, but sometimes not available depending possibly on the complexity of the sketch, hence the Parameter Editor facility is provided.

The User Panel provides reserved regions on-screen for display of robot status information and also there is a console display available shown in the Processing IDE window. for displaying data from print/println method calls. It should be noted that if you are not particular about formatting, a comma separated list can be used for multiple variable display. Also, Processing printArray() is handy for simple way to print out the contents of arrays.

Right clicking on variables or method names provides some very handy functions including most notably ShowUsage, Jump To Declaration and Rename.

Few Notes On Java, If You Have C, C++ Background

If you have a C or C++ background, you will note that the low-level program constructs are identical to C, but do use care integer types, they are all signed in java. Class instances and arrays are passed to methods by reference.

Certainly Google searches can be a great ally, e.g. “how do I declare an array in Java”

Of course, first course of action, try out example sketches. LFS_SimpleBot is the simplest. In processing IDE click File>Examples, scroll down to “Line Follower Simulator” and expand the branch.

System Requirements

LFS does have some reasonably heavy demands on GPU memory. The DPRG challenge course requires the graphics card be able to handle a large texture map, i.e. approximately a 54 mega pixel image at 64 dpi.

Any machine equipped for light gaming should not have a problem.
For example my machine is i7-8700 equipped with a Nvidia GTX-1060 works well with an average frame rate (simulation step rate) of 150 frames (steps) per second which is about 2X real time.

Low-power laptops with minimal GPU hardware might have to struggle.
Smaller course size would help, the full Challenge course image might be a problem.

Simulation Loop

The following 5 steps are executed per one simulation time step.

1. Draw view of course visible in local area of robot as would be seen from above a transparent robot where front of robot is oriented toward top of screen-window and robot center is center of screen-window.
2. Acquire sensor input from user defined sensors from view drawn in step 1.
3. If in course view mode (Tab key toggles on/off), draw course view over top of robot view.
4. Call user controller (java method) which analyzes sensor data and ultimately calculates robot turn rate and forward velocity (or in tricycle mode, wheel angle and velocity). Also user code can draw its own overlay graphics and/or color the sensors to indicate / verify controller understanding of sensor data.
5. Update robot location (in world coordinates) including x,y location and heading based on fixed time step.

Currently two robot views appear in the simulation. The largest is the sensor view drawn for sensor data acquisition. It is drawn at 1:1 scale where typical course is 64 dots per inch. In this case the default 800x800 pixel area covers 12.5” x 12.5” area. A second scaled down robot view is drawn over top of the sensor view. There is an option to cover the original sensor view or gray it out. This may change in the future, but early attempts to render this view off screen slowed frame rate.

Details on Robot Position and Heading Calculations

User robot drive input is now specified as targetSpeed and targetTurnRate. where the robot speed and turn rate change are governed by user defined acceleration and deceleration rates subject to simulator maximum values.

Note: Turn deceleration is made equal to turn acceleration rate, so only turn acceleration rate is specified.

All acceleration/deceleration rates are constant values, thus change in speed or turn rate occurs at a constant rate over time creating a linear (straight-line) ramp up or down from an old rate to a new one.

For Example: Given simulation dt = 0.01 (to keep math simple) default is 0.0166 sec and given accRate = 20 inches/sec^2 (inches/sec/sec)

In one time interval speed would increase by accRate * dt (20 x 0.01 = 0.2 inches/sec)

speed = speed + accRate * dt

If speed was initially 0 it would ramp up at a 0.2 inches/sec for each simulation step. After 10 steps speed would be 2.0 inches/sec.

Thus a target speed could be specified and if speed is less than target speed, the acceleration process continues. If the speed exceeds the target speed, the speed is set to that target speed and the acceleration stops.

If for example target speed was 2.1, on the 11th step the 2.2 value would be limited to 2.1 (the target speed) and would remain at that value until a new target speed is set.

Similarly if a target speed is less than the current speed the above process would be applied but using set deceleration rate and a subtraction of decelRate*dt from speed until the new lower target speed is reached.

Likewise turn rate is handled the same way, but independently in each time step.

With a current speed (and turn rate) in hand, a second numerical integration action is performed for each simulation time step:

The distance d traveled in a given time step is the product of speed (inches/sec) and time in seconds (dt)

```
  d = speed * dt
```

Thus for a current speed of 2.2 inches per sec with dt = 0.01 sec
total distance traveled would be 0.022 inches.

The actual change in X and Y location on the course depends on robot heading.
At default heading of 0 degrees the robot moves in -X direction so calculating position is very simple. Y remains constant and 0.022 would be subtracted from X every time step.

Relying on some simple trigonometry we can make use of sine and cosine functions to resolve distance traveled into changes in X and Y as a function of heading.

Given heading in degrees = h, current x,y location, update x,y and heading

```
  h = h + turnRate * dt          new heading based on current turn rate

  if h>=360.0) h = h – 360       restrict to 0..360 degrees,  causing wrap around
 if (h<0) h = h + 360            for example heading  -1 becomes 359, 366 becomes 6

  hr = h * PI/180                convert heading in degrees  to radians, for sine,cosine
functions

  x = x – cos(hr) * d            updated robot x,y coordinates
  y = y – sin(hr) * d
```

Sensors

Sensors are defined and placed relative to robot using code defined in userInit() method in the userInit tab. Sensors are placed relative to the robot center with offset distances in inches. Positive X is distance in front of center point, positive Y is distance to right of robot center.

The sensors are used to sample the visible region around the robot where the simulator renders the view as would be seen from a camera placed above the robot, looking down, without the robot obscuring the view. (Invisible robot).

Two types of sensors can be defined.

1. Spot Sensor samples a rectangular region of the course and returns a single intensity value ranging from 0.0 ,e.g., black line, to 1.0 white background.

2. Line Sensor is linear array of spot sensors and returns an array of intensity values.

Line sensors now have two available modifiers. setRadius(radiusInches)
turns line into 1/2 circle with center at offset specified for sensor center. Also setRotation(degrees) rotates line or 1/2 circle about center. Negative values clockwise, positive counter-clockwise.

Line sensor data is read using readArray() method which returns array of floats ranged from 0.0 (black) to 1.0 (pure white). Spot sensor data is returned as a single float value using read() method.

See Sample controller code for example of how to access sensor data.

Getting Started With LFS

1. Download Processing from www.processing.org website.
2. Play with the processing IDE, there are many good tutorials
3. Get the LFS library code from git hub.

The library code and bundled demo sketches are included on Git Hub at https://github.com/ron-grant/LineFollowerSimLib.

Download the LineFollowerSim-XXX.zip file to your computer.
Extract the LineFollowerSim folder from zip file and copy to the libraries sub-folder of your Sketchbook. To locate your Sketchbook folder, start up Processing and use menu command File->Preferences. The Sketchbook location appears at the top of the Preferences dialog box.

After copying the LineFollowerSim folder to the libraries folder, close down and restart Processing. You should now see the LineFollowerSimulator library after issuing File>Examples menu command. The example sketches will appear when you click the [-] on the LineFollowerSimulator tree entry. Click on any of the examples (two at the moment) to load the sketch and press run button (Circle with triangle).

Note this rendition of the simulator & examples use simple keypress. You may need to click on the window outside the course view or robot view to give the application focus.

For now the command pattern on a contest run is (Press R) to run, space bar to stop, "F" to finish the contest run , including adding or appending to contest.cdf file in sketch data sub-folder.

Also, in sketch folder a screen shot is saved for the "Finished" run named with user and current time of day FirstnameLastnameRobotname-hr-mn-se.png

The demo sketches should be quite useful in seeing how to use LFS.

The LFS library application interface is documented in reference sub-folder of library as html pages generated by javadoc utility.

At present every LFS sketch includes a fair amount of code not relevant to your controller code. Check out the section that details what all the notebook tabs are, and which are relevant to your controller program.

Key Command Summary

LFS accepts keyboard commands when the program has focus. When the program does not have focus, it displays a message: “Click in application window to give focus for key command response.”

At present you may want to click outside robot and course viewport when clicking to give focus to avoid moving the robot.

Horz. Cursor Arrows Change robot turn rate when controller OFF.
Vertical Cursor Arrows Change speed of robot when controller OFF.

User code may override the arrow key functionality. See information on UKey notebook tab.

SPACE	Toggle Controller FREEZE (stop taking steps) ON/OFF -- OR If Contest Running STOP Contest.
0..9	Throttle controller step rate from 0=FREEZE to 9 full speed
C	Toggle user controller On/Off. When off arrow keys can be used to drive robot.
E	Erase cookie crumbs.
G	Go – similar to run contest, except you have freedom to modify robot location, grab with mouse and move, and also freedom to set markers, recording robot state if your robot code uses RobotState class.
H	Help – Multi-Page help press H repetitvely to page through key command details, simulator imposed maximums, and user help text file.
M	Marker placed at current robot location. If running, a state save marker is placed (circle with rotating box). If robot is stopped controller OFF, then a simple marker is placed (circle). If marker is already present at the location it is erased. See also Mouse Commands
P	Parameter Editor Display on/off. (See Section – Parameter Editor)

R	Run Contest. Robot is moved to starting position, stopwatch cleared, cookie crumbs are cleared and user controller is enabled. All keyboard commands are offline at this point except for SPACE bar to stop contest run. At that point, a modal dialog overrides the key summary window, giving you two choices: F – Finish Contest and append run data to report file (contest.cdf) located in sketch data sub-folder. X – Cancel Report.
S	Stop Robot (turns controller OFF, if ON)
TAB	Toggle between course visible with small robot view, and just larger sensor view with large User Panel 2 visible.
ALT	Toggle rotation of course view 0 and 90 degrees

Mouse Commands

When simulation is not in run contest mode (where R was pressed):

Hold down left mouse button in either robot view or course view and drag to move the robot. Hold down right mouse button with mouse positioned in either view and move horizontally to rotate the robot to a new heading.

If you click on a marker (magenta circle), the robot will move to that location and turn to the heading recorded for the marker. The robot will be stopped and controller OFF, ready for contest run state (R key) or a non-contest run start (G key).

If you click on a state save marker (magenta circle with rotating square), the robot will move to that location with controller turned on and run state recovered, but in FREEZE mode where simulation steps are not being taken (your controller code is not being called, and robot position update is not occurring). Press SPACE bar to un freeze or number 1..9 to resume robot run. See Robot State Save Recover.

Hovering over sensors in robot view or larger sensor view (accessed when course not visible toggled with TAB key) will identify sensor by name and show value at the bottom of the viewport. Spot index number is identified when hovering over line sensor spots. If moving the robot around, if might be helpful to enable controller (C key) and FREEZE robot with space bar.

Parameter Editor

The Parameter Editor is an optional facility that allows for editing parameter values that are typically set as constants within your controller program. The parameter editor supports int and float variables defined in global variable pool (not defined inside any methods).

One line of code is required for each parameter (variable) you want to include in the variables accessible by the parameter editor. This line of code specifies:

The variable you are modifying.

A friendly string name for the variable. Typically same name as variable.

Description string, which can be empty.

Default Value (also used as Initial Value) that the variable is set to.

Minimum Value allowed

Maximum Value allowed

Delta Value (for float parameters specifies minimum increment e.g. 0.1), int parameters have delta of 1.

Adding Your Own Parameters (UPar NoteBook Page)

For example if you have a float maxSpeedOnCurve constant defined in you program as 3.0 (inches/sec). This variable would be a good candidate for the Parameter Editor.

Again, make sure the variable is global, defined outside of your controller or other methods, then

in UPar tab, you would add a line of code to parEditorUpdate method after p.beginList() and before end of p.endList().

Shown here with comment reminder on arguments

maxSpeedOnCurve = p.ParF(maxSpeedOnCurve,”maxSpeedOnCurve”,”speed in/sec”,

3.0,1.0,20.0,0.1); // var,name,description,default value, min value, max value, delta

Note: there are examples already present which you might want to comment out. Also, there are comments in UPar to remind/inform you about parameter details.

Parameter Editor Usage / Commands

When not running a contest, pressing P toggles the display (on/off) of the Parameter Editor window in the lower-left corner of LFS window.

Use mouse to hover over Parameter Editor items, and they will "light up".

While hovering over an item (variable):

Use the + or – keys (or roll of mouse wheel) to increment or decrement variables.

Also, you can press and hold left mouse button and move mouse horizontally to adjust value.

Additional Parameter Editor Keys

(Control Keys e.g. Ctrl-A : Press and hold Ctrl then press A key and release, both keys)

Ctrl-D	Set a parameter to its default, while hovering over it.
Ctrl-A	Set all parameters to their defaults.
Ctrl-S	Save parameters to data folder param.cdf
Ctrl-L	Load parameters from data folder param.cdf
PgUp PgDn	If more than 5 parameters defined, use PgDn/PgUp to scroll though pages.

Clicking on [X] in upper right of window is an alternate means of hiding the Parameter Editor window. When hidden, parameters are not available for edit or processing of any key press data.

Robot State Save and Recover (Optional – New Feature)

Robot state save and recover is provided as a means of being able to instantly return a robot with all its controller state variables in tact, to a location just before navigation failure on a complex course, saving time waiting for the robot to navigate to that point.

This facility is implemented as an extension to the marker system where placement of a marker while the robot is running in non-contest mode (started with G key), pressing M will generate a marker with state information saved. The state save marker appears as a marker circle, but with added rotating box inside.

The basic marker system records only a position and and heading for the robot to start driving from. Basic markers are defined when robot is not running, e.g. after issuing stop command via S key or after contest run stop, and finish or cancel.

The state save marker includes location,heading, current speed, turn rate, time elapsed, and information you optionally provide via variables defined inside a class called RobotState which appears in UserReset tab.

If your controller carries very little state information then this facility may not be important to you, where a marker is "good enough". Also, with care in placement of simple markers that again might eliminate the need to invest effort in placing variables or copies of variables into RobotState class.

Your userControllerUpdate must insure all state variables are updated before its completion or return to the LFS code calling it.

The simplest and safest way to handle this task is to define state variables within the RobotState class.

The more complex method of copying from currentRobotState instance to the controller at the start of your userControllerUpdate, then back to currentRobotState just before return from your userControllerUpdate is another option.

After creating one or more markers with saved state. Click on the marker and the robot will move to the marker with controller enabled , but will be be frozen, press SPACE to unfreeze or a digit 1..9 to select a run step rate.

If your controller is very state critical, you may have problems. As of (lib 1.5), I am aware LFS state is not fully captured, e.g. missing target speed, target turn rate..

Important: If you add or remove state variables you might have to erase old save state information to eliminate program crash. The state data is saved as a file with same name as course image, but with a ".srs" (saved robot state) file extension. This is a human readable ASCII file in JSON format. Key,value data does not appear in any defined order.

You can always try a few iterations of saving a state in different , then testing if your robot will start and repeat desired failure behavior some time later. When that is established verify the same behavior is obtained after closing and restarting the sketch. At that point, go to work on your controller.

Implementing Robot State in User Controller Code

In the UserReset notebook tab, there is class definition called RobotState. Within the class definition which contains only variables declarations you can add your own robot state variables in the section commented USER ROBOT STATE VARIABLES BELOW.

The supported data types for RobotState class are float,int,boolean,String.

Also supported (by untested at release of this document) are int[] and float[], integer and float arrays.

An instance of this class is defined as currentRobotState. This instance should be used by your initialization and by your controller code. A terse shorthand reference to this variable is demonstrated in the LFS_SimpleBot and LFS_TrikeDemo applications.

That is, RobotState cs = currentRobotState; is coded which allows access to the data filelds, e.g. , cs.sampleCounter + = 1; // increment sampleCounter

Example Robot Sketches Included

LFS_SimpleBot

Simplebot is a simple controller more suitable for the advanced course. It is not designed to handle reverse images, acute angles...

The controller calculates the centroid of a dark area on a line censor with respect to the center of the sensor returning a signed value which indicates the sensor centers offset from the center of the line, which I will call the error.

The the error is in pixel units which at 64DPI, equates to 64ths of an inch per unit.
For Example a value of 4 = 4/64 = 1/16 of an inch.

The error and its first derivative (change since last time step) is used to control robot turn rate implementing a PD controller.

The robot velocity is controlled, optionally by a line of code which sets the robot drive speed to a value related to the inverse of the error. If commented out, you can manually set robot speed with up and down arrows when the controller is enabled, but not in contest "Run" mode.

LFS_TrikeX (where X = revision date)

This sketch implements Will Kuhnle's tricycle robot controller.

His claim at the moment is "The code is a work in progress – not refined, messy..
Will's trike (tricycle) operates like the tricycle you had as a kid, except in reverse.
The steered and driven wheel is at the back and the inputs are wheel angle and wheel velocity. You could replace his controller code with your own while leaving trikeMode = true to provide the trike functionality vs differential drive robot functionality.

An interesting note from Will, and see his code, a given trike wheel angle and speed is easily transformed to a turn rate and speed of a differential drive robot.

Writing A Controller

Your userControllerUpdate method (UserCon tab) is called every simulation time step.

A time step is defined when the simulation starts as a tiny amount of time with a default value of   0.01667 seconds (about 1/60th of second). It can be set within the range 0.1 down to 0.01 seconds.

LFS can be instructed to throttle the rate at which time steps are taken or even stop them, i.e. SPACE bar during non-contest run will freeze the simulation and 1..9 will control how fast the steps are taken. Also, LFS generates time steps at faster than realtime if possible, e.g. for a timestep of 1/60th second, if LFS can generated 120 steps per second your robot will run 2X real-time if step rate has not been throttled by 1..9 keys. It is very handy to run the robot faster or slower than real-time depending on the circumstances. Single step can be very handy.

With all this said, your userControllerUpdate method should expect one time step has happened since last invocation and that sensor data has been updated to reflect the distance the robot has traveled (if not stopped) and the heading change it may have experienced.

Also, the stop watch is advanced one tick per time step which is why the simulation can be stepped at any speed an the simulation will produce the same result.

Your method may call other methods in the same tab, or methods in other tabs you create. State variables declared outside your method are global to program and are active for entire sketch. For complex line following courses you may have to implement complex logic to recognize various course features.

Your choice of sensors may simplify or complicate the task.

Finally after analyzing the sensor data and updating internal state, your controller should calculate a targe speed and target turn rate which are both subject to acceleration and deceleration values defined in your userInit() method which are subject to simulator maximums available on 2nd Help page (H – key).

Load and play with the LFS_SimpleBot sketch and refer to the next section for a discussion on the different code notebook tabs present in the application.

The LFS_Simple bot has a very simple line detector not suitable for handling reverse polarity images (white line on black background).

You might want to look at Will Kunhle's LFS_DemoTrike example which included sensor code that looks at transitions spanning a line sensor to help handle reverse polarity (white on black tiles) or write your own.

The complexity of the code will depend on what problem you are trying to solve. The challenge course is likely to require a significant coding effort unless you are very clever with sensor placement, then your code complexity might be reduced.

Description Of LFS Files (IDE Notebook Tabs)

All files located within your sketch folder that contain .pde extension are included in the sketch notebook tabs that appear at the top of the screen when running the Processing IDE.

All tabs prefixed with LFS_ are part of the LFS application which you should not need to modify or be aware of any details except they often do have useful header comments. A future goal might be to push more of this code out-of-sight into the LFS library.

Tabs prefixed with “User” (UserCon,UserDraw,UserReset) contain pre-defined methods that you will modify from the original LFS_SimpleBot sketch.

Tabs prefixed with “U” (UKey,UMisc,UPanel,UPar) can be optionally modified to add features that are not mission critical to writing your line follower controller.

An exception to the above is modifying UserDraw is also optional, and the optional feature of including your controller state variables into the state saving RobotState class is included in UserReset.

It is suggested when you load a demo sketch and decide to modify it, execute a Save As to a meaningful name with LFS_prefix recommended, e.g. LFS_SpeedyBot5.
This sketch will be a clone of the demo sketch except saved to your sketchbook folder using the name you specified for the folder and the name of the main notebook tab.

Just like the Arduino IDE (developed by the Processing folks), the main tab is always given the name of the sketch and that name is the same as the sketch folder.

Every tab defined within LFS application does have a header comment that will help clue you into its function

LFS Files you should not need to modify or know details on, except reading header comments might be useful. These files are subject to change with subsequent library/demo program release, with the goal being. replace these files and leave your code alone.

NoteBook Tab (.pde file)	**Description of Code**
LFS_SimpleBot	Very high-level processing sketch skeleton should not change with code versions, renamed to your sketch name when you SaveAs to your sketchbook folder.
LFS_M	LFS High level code.
LFS_Key	LFS Key command decode, with calls to user custom key decode.
LFS_Panel	LFS Command Summary, Help Screens with calls to user Panel code.
LFS_Par	LFS Parameter Editor, handles editing variables you specify in a list in UPar tab.
LFS_RS	LFS RobotState save/restore, works with RobotState class defined in UserReset tab.

User Notebook Tabs (.pde files) Methods that are shown in bold type are essential methods that you will interact with, others, while they need to be present are used with optional program features. In almost all cases there are examples present that help illustrate how to use the methods. Also, all User tabs have header comments to help explain the purpose of the tab / methods contained.

Tab	Methods Called by LFS	Description of Code
UserCon	**userControllerUpdate()**	Your controller method called every simulation time step. The purpose of this method is to read sensor data and make a decision about speed and turn rate needed to keep the robot on track following the line/path. Your method may call other methods in the same tab, or methods in other tabs you create.
UserInit	**userInit()**	UserInit tab contains userInit() method which is called by LFS at startup and typically when userControllerResetAndRun is called when a “R-Run” command is executed. This method defines contestant name, robot name, list of courses to be used, and default course#. Also, acceleration deceleration parameters, sensor definitions including their type (spot or line) and placement. Also, modifiers can be used to turn line sensors into 1/2 circle arc sensors and rotate both line and 1/2 circle sensors.
UserReset	**userControllerResetAndRun()**	Called when robot commanded to start running contest (R-Run contest command key pressed). This method typically should call userInit() method followed by resetting any state variables you have defined in your controller.

Tab	Methods Called by LFS	Description of Code
UserDraw	userDraw**()**	Called by LFS when it is prepared to allow optional drawing overlay on robot or sensor view (larger robot view) using robot coordinates.

.UKey	userKeypress()
	userArrowKeyDecode()
		Called by LFS when key is pressed and not decoded . Used for optional user key commands.
		userArrowKeyDecode, allows repurposing arrow keys as is performed in LFS_DemoTrike application.

UMisc	userMiscSetup()
	userMiscUpdate()
	userLapDetected()
	userStartedRun()
	userStop()
	userInBounds()
	userOutOfBounds()
	userFinishLineDetected()
	userNewMarkerPlaced(placed);
	userMarkerClicked();
		A bunch of methods you may never need. See tab header comments for description of methods.

UPanel	userDrawPanel1()
	userDrawPanel2()
		Your methods that can optionally draw text and graphics in User panels1 located screen lower-left when parameter editor is not visible, and Panel2 located screen right when course hidden (Tab Key toggles view). Note: as a demo feature, Panel2 displays robot state information stored in currentRobotState.

UPar	parEditorUpdate()	You specify a list of global variables as s parF and parI method calls to allow edit of float or int variables. See Parameter Editor

LFS API Documentation

Most all LFS functionality is accessed through a single instance lfs of LFS class (predefined in the LFS application) and the instances of the SpotSensor and the LineSensor class you define in userInit tab.

The LFS Application Interface (API) documentation might come in handy. Do note that there are many methods which don't have relevance to the writing of a line following controller. The reference is more there to serve as a reference when reading demo code that makes reference to the API.

The API documentation is located in the reference folder of LineFollowerSim (which is located in your sketch book libraries sub-folder). Click on index.html which will display the document in your browser. You might want to create a shortcut to this documentation on your desktop.

The following section lists most of the LFS methods and classes you will encounter. Also there are some methods which are defined within the application and not the library. More explanation is given in that case.

Example LFS Methods Used In Controller

Listed here are LFS methods and variables you will find in a typical user controller program. Study LFS_SimpleBot to see complete program. You will find API documentation on the bolded items. The essential notebook tabs (UserInit,UserReset and UserCon) are presented first.

UserInit Notebook Tab

SpotSensor sensorL,sensorM,sensorR;   // declarations for 3 spot sensors

LineSensor sensor1;   // declaration for line sensor

void userInit()   // LFS calls this method at start up. All code is contained within except
                        // global sensor declarations.

// Define Contestant Name, Robot

// Course List and Initial Choice, also number of laps if lap timed course

lfs.setFirstNameLastNameRobotName("Ron","Grant","SimpleBot5");

lfs.defineLapCourse(2,"Advanced_LF_course_Fall-2018_64DPI.jpg");
lfs.defineCourse (3,"DPRG_Challenge_2011_64DPI.jpg",71.7,120,0);
lfs.chooseCourseOneTime(3);
lfs.lapTimer.lapCountMax = 3;

lfs.setTimeStep(0.0166); // set timestep (optional) default is 0.01667 range 0.01 to 0.1
// seconds

lfs.setAccRate(16); // acceleration rate (inches/sec^2)
lfs.setDecelRate(32); // deceleration rate (inches/sec^2)
lfs.setTurnAcc(720); // turn acceleration and deceleration rate (degrees/sec^2)

lfs.**setMaxSpeed**(16); // optional - inform LFS of your max speed *
lfs.**setMaxTurnRate**(720); // optional – inform LFS of your max turn rate (degrees/sec)

sensorL = lfs.**createSpotSensor**(1,-2,12,12); // example instantiated spot sensor
sensor1 = lfs.**createLineSensor(**2.0f, 0, 5,5, 65) // example instantiated line sensor
sensor1.**setArcRadius**(2);
sensor1.**setRotation** (0);
sensor1.**setPosition**(2.5,0); // example position modification

nameSensorUsingVariableNames is application method used for getting sensor variable names when hovering over sensor in robot or sensor view. This method call should be included.

nameSensorsUsingVariableNames(); // defined in local app.

genRobotIcon is application method that generates a robot icon image allowing user specified initials (2 or 3 characters), robot color and text color. This method is used if you would like a robot icon, but don't want to create your own custom image. bigIcon is a PImage reference defined in UserDraw tab. PImage is a Processing Image class defined in the Processing reference manual, e.g. right click on PImage in code and Find in Reference

```
bigIcon = genRobotIcon (initialsString,robotColor,textColor);

bigIcon = loadImage (“myIconImage.png”);  // example of loading image file from data
                                          // folder using Processing loadImage
```

// with either of the of the above you have a PImage, bigIcon that can be displayed in userDraw method of UserDraw tab.

// next setRobotIcon or setRobotIconImage are used for icon to be displayed on the robot course. This can be the same or different than the image displayed in the robot or sensor views.

```
lfs.setRobotIcon("SimpleBotIcon.png",255);  // example load icon from data folder
                                            // specify alpha transparency 0=clear to
                                            // 255 = opaque

lfs.setRobotIconImage(bigIcon,255);         // example using bigIcon bitmap for
                                            // course displayed icon versus loading
                                            // from file.
lfs.setRobotIconScale(0.1);                 // example scale robot shown on course
                                            // = 1/10 scale

 lfs.setCrumbThresholdDist(0.5);            // Distance between cookie crumbs in
                                            // inches. Increase  to reduce frame rate
                                            // slow-down ing on long course runs.
```

UserReset NotebookTab

```
void userControllerResetAndRun()   // method called by LFS when R or G key pressed
```

```
currentRobotState;    // instance of RobotState used for optional storage of robot state
```

```
class RobotState  { // class definition that holds some LFS variables and also
                  // your variables if you want to use Robot State Save / Recover

 int sampleCounter;      // sample counter variable – part of user Robot state

}
```

lfs.**moveToStartLocationAndHeading**(); // should always be present
// needed for Marker support, go to last
// clicked marker when R or G pressed

```
  RobotState cs = currentRobotState;       // example of short hand access robot state
  cs.sampleCounter = 1;                    //
  currentRobotState.sampleCounter = 1;  // same thing without short hand
```

UserCon Notebook Tab – User Robot Controller

void userControllerUpdate ()   // LFS calls every time step.

float[] sensor = **sensor1.readArray**();   // example getting a reference to line sensor
// data. floating point array 0.0 to 1.0 values
sensorL.**read**(); // example read of spot sensor intensity value 0.0 black to 1.0 white

```
// optional color sensors to illustrate controller interpretation of data,
// default is green to show location of sensor.
// controller must be enabled for the updating of color information
// at least when updated within userControllerUpdate(). There might be other
// options to consider.  Coloring sensors has no impact on data values.
```

int[] colorTable = sensor1.**getColorArray**();   // example reference to line sensor colors

// example setting all spots within line sensor to black.

for (int i=0; i<sensor1**.getSensorCellCount()**; i++) colorTable[i] = color(0,0,0);

color c = color (255,0,0);   // example Processing – set color, bright red
sensorL.setColor(c);   // example setting spot sensor color

// output target Speed / Turn Rate to LFS

```
lfs.setTargetTurnRate(t);              // set turn rate subject to defined turn
                                       // acceleration
lfs.setTargetSpeed(s);                 // set target speed subject to accel./deceleration
lfs.setTargetSidewaysSpeed (ss) ;      // set sideways target speed subject to
                                       //  acceleration /deceleration rates.
                                       //  For Example, If using Mecanum  wheels or
                                       //  floating / flying robot with sideways
                                       // motion possible
```

UserDraw Notebook Tab

LFS_SimpleBot app. contains examples for optional drawing over robot or sensor view.

Here you are using primarily Processing drawing methods after setting up coordinate transforms via :

lfs.**setupUserDraw**()   // called if course visible and robot view is visible

lfs.**setupUserDrawSensorViewport**() // called if course not visible and sensor view is
                                                    // visible

After setting of transforms all drawing methods use robot coordinates (inches).

lfs.**drawRobotCoordAxes**(); // draw robot/sensor view axes

UKey Notebook Tab

User key decode, see LFS_SimpleBot app. for details. This tab is only used if you want to add some of your own key commands.

userKeypress(char key)   // method called by LFS if key press not decoded.

userArrowKeyDecode is method called by LFS allowing user to override arrow key decoding. Simply returning true informs LFS not to use keycodes. This is done in LFS_DemoTrike, which uses arrows to control its wheel speed and turn angle, which are transformed into speed and rate of turn values.

```
boolean userArrowKeyDecode(int keyCode)   // method called by LFS allowing take
                                          // arrow key override.
```

UMisc Notebook Tab

Miscellaneous user methods. See LFS_SimpleBot app. for details.

UPar Notebook Tab

Parameter Editor, see Parameter Editor section for details.

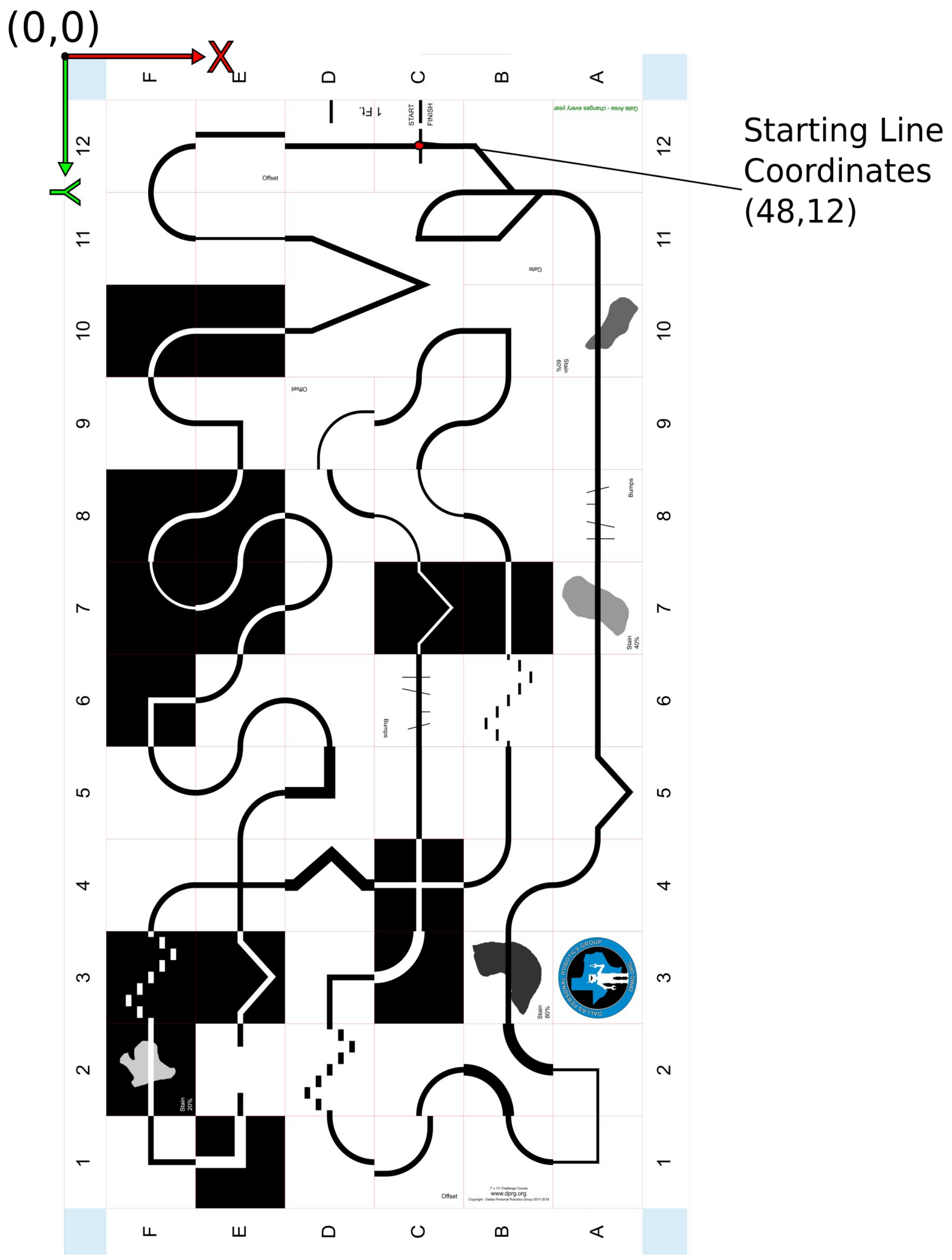
Here you will define parameters for integer or float parameters used by your controller. One line of code for each variable. For example

```
 maxSpeed = p.parF(maxSpeed,"maxSpeed","inches/sec ",6.0,0.0,24.0,1.0);
  sampleInt= p.parI(sampleInt,"sampleInt","sample integer",0,-10,10);
```

maxSpeed and sampleInt are global variables defined outside of any method.

LFS World Coordinates

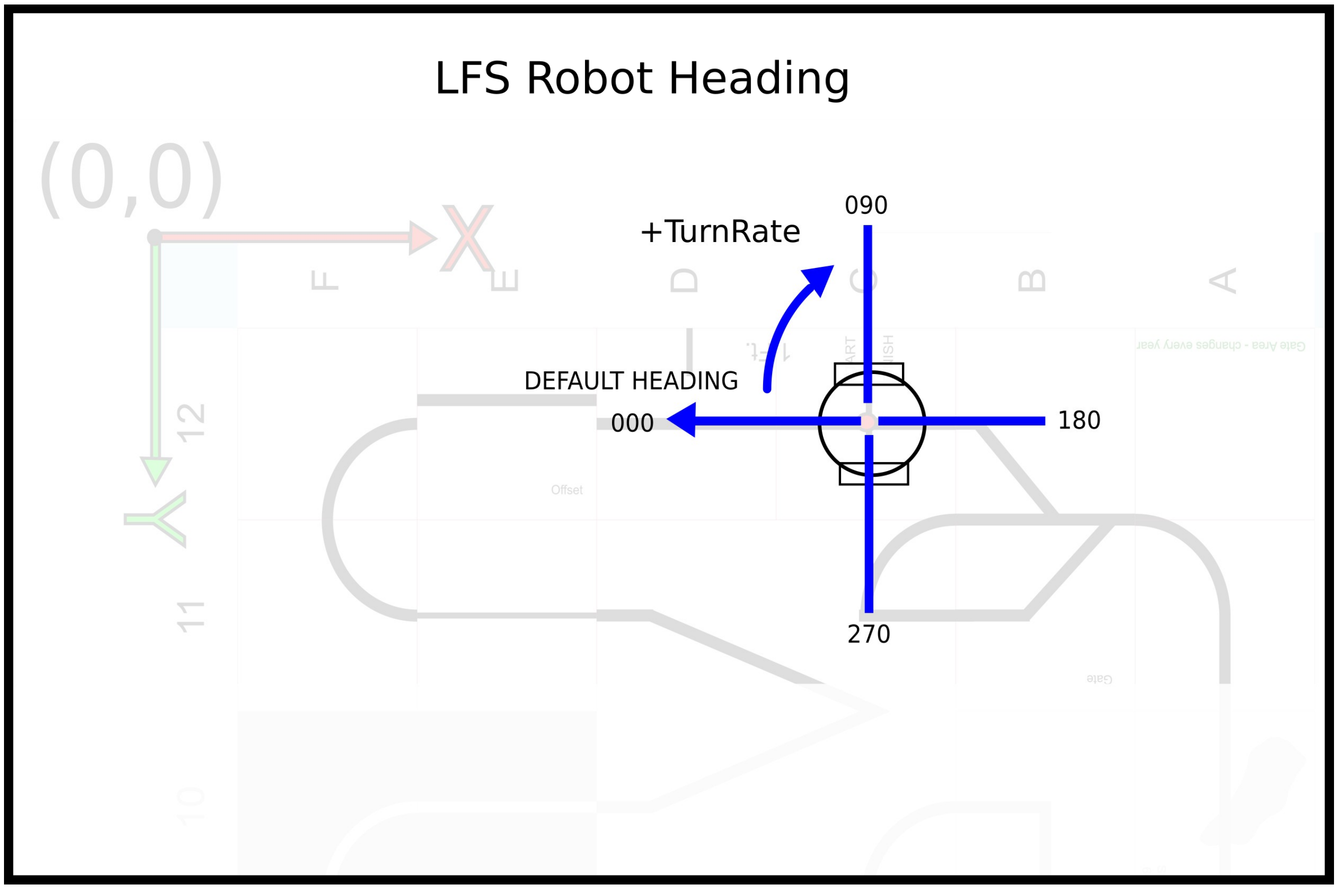
Origin is Course Image Top-Left Corner (DPRG Challenge Course shown here)
Tiles are 12x12 (inches) at 64 dots/inch (DPI) = 768x768 pixels per tile
Borders are 6 inches. Robot is located in world coordinates.



Robot Heading

Robot default heading 0 degrees.
Positive turnRate will turn robot in direction of arc (right turn) with increasing heading, while negative turnRate will turn robot the other direction (left turn) with decreasing heading.

Heading values are confined to the interval 0..359.99,
e.g. decreasing heading crossing 0 (and going negative) has 360 added to its values.
Likewise, increasing heading exceeding 360 (359.99) will have 360 subtracted from its value.

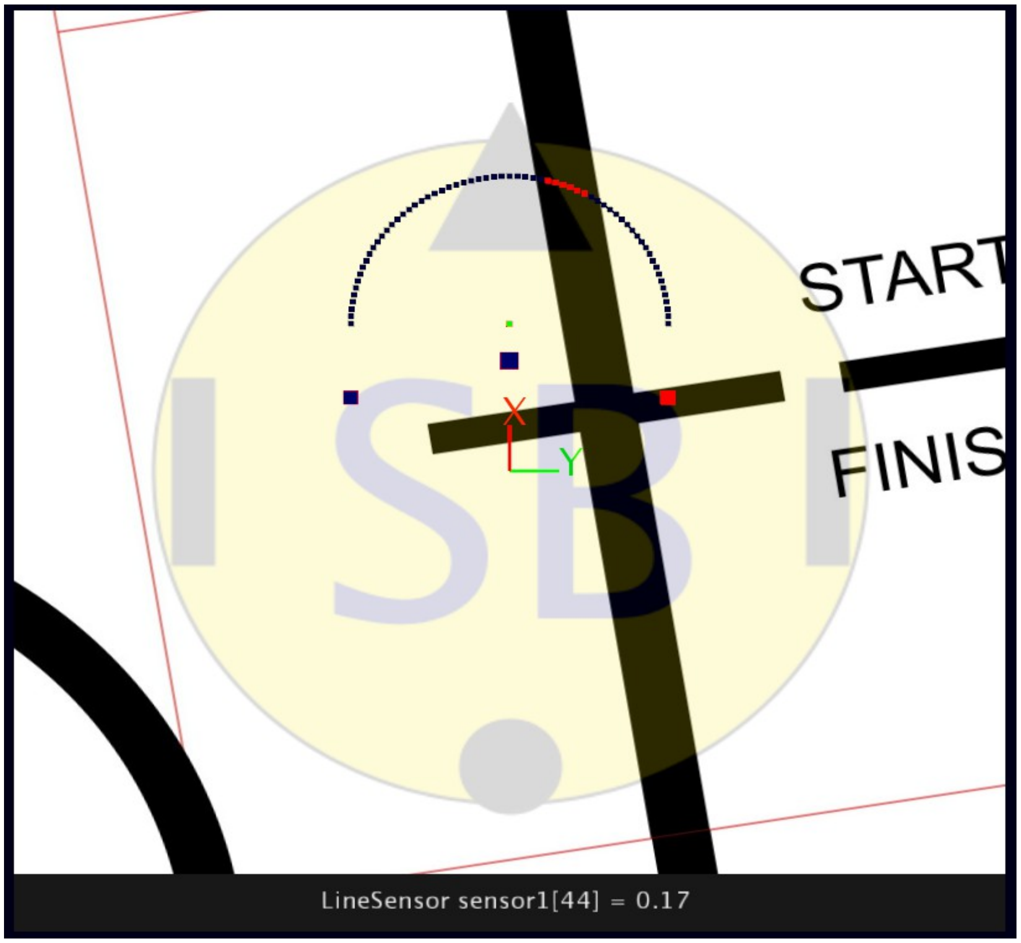


Robot Coordinates / Robot View

Robot origin (0,0) is midpoint between wheels, center of screen in robot view and larger sensor view. Robot position in world is such that robot origin is placed at current world (x,y) location.

The Sensor viewport image below shows robot overhead view with optional robot icon superimposed. Robot Coordinate axes are show (center of robot). Sensors are located in robot coordinates (inch units). Line sensor with 1/2 circle modifier location is at (2,0) with 2 inch radius. Spot sensors are located as (1,-2), center at (1.5,0) and right at (1,2)

Sensor data shown at the bottom of this screen image appeared due to mouse being hovered over a given sensor. In this case a spot at the edge of the line was chosen where the reading reported was mostly “on the line” (the line color, black = 0.0, background white = 1.0).



LineSensor sensor1[44] = 0.17