P545 Lab Manual

Fall 2011

Steven D. Johnson, Bryce Himebaugh, Caleb Hess and Scott Dial

September 1, 2011

NOTE: This Laboratory Manual is under constant revision. Check frequently whether you have the most up-to-date version.

I

Introduction and Overview

This chapter briefly describes tools and methods used in the P545 Lab Projects.

I.1 ERTS

ERTS is an electic golf car outfitted for computer control. The logical view of ERTS's architecture is typical of systems of its kind (Fig. 1). Its five logical levels are:

1. *Mechanical*. The vehicle and its moving parts, wheels, axels, gear boxes, steering mechanism, pedals, motor, and so forth. Mechanical control behaves like a standard vehicle. The driver activates a power switch, steers with the steering wheel and column, accelerates with the throttle pedal, deccelerates with the brake pedal, sets the parking brake with a friction lever.

2. *Electrical*. Under computer control, electro-mechanical *actuators* perform the driver's actions. Actuator position is driven by varying the ampherage supplied to it. With each actuator there is a *sensor* measuring the physical postion of the actuators. Under *closed-loop control* a controller repeatedly reads a voltage present in the sensor, computes the difference between the actual and desired position of the actuator and emits a *command* to correct this *error*. The frequency of this loop depends on the mechanical properties of the actuator.

3. *Digital*. The next layer of the logical architecture digitizes the sensor readings and actuator commands, so that they can be dealt with by a program or dedicated digital controller. Hardware ADC/DAC devices convert from/to

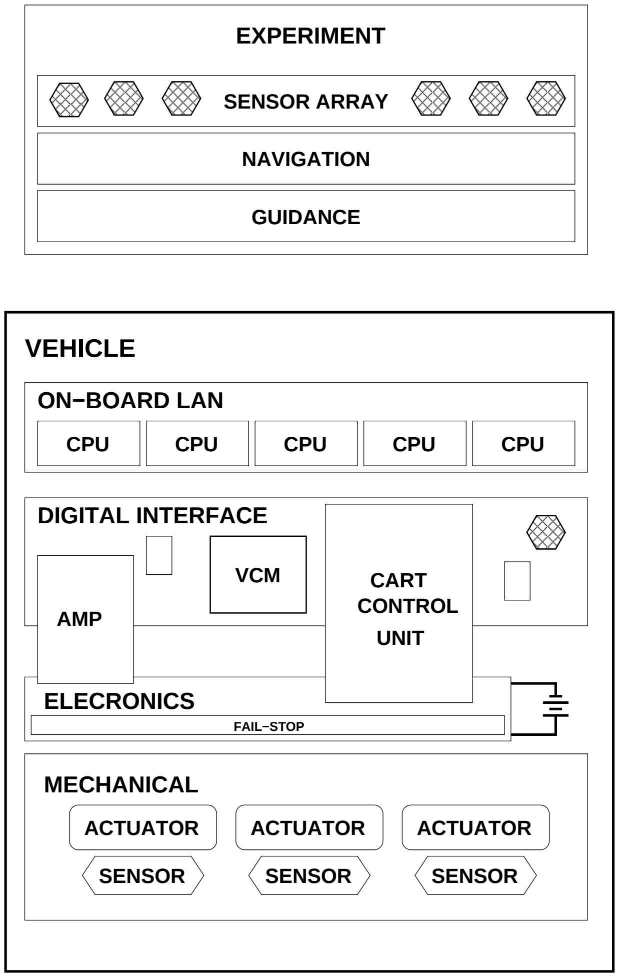


Figure 1: ERTS logical architecture

continuous analog electronics to/from discrete digital values that conventional processors can manipulate.

4. *Computational.* On ERTS, digital control calculations are done with software. There is an on-board network of *compute nodes* running this software.

5. *Experimental.* ERTS mission is to serve as a platform for experimental research in autonomous robotics. A system deployed on ERTS for performing research is referred to as "the experiment." The aim in ERTS development is to provide an environment for researchers who are not experts in robotic control. Such an environment is itself a research problem.

I.2 ERTS Design Model

The design model for ERTS development is a system of synchronized, communicating *components*. The operating system provides a global synchronizing event called clock. Each component is a cyclic process with the following behavior:

```
$\mathcal{C}$:  cycle [
           synchronize;
           accept all inputs, $I$;
           $S$ := next($I, S$);
           emit all outputs from $S$;
         ]
```

The model is similar to that of a clocked-sequential finite-state machine and is the dominant model for digital hardware design. Variable S represents component $\mathcal{C}$'s internal state. In each cycle, $\mathcal{C}$ first synchronizes with all components in the system. At that point, it captures and holds the values present on each of its input channels. $\mathcal{C}$ then updates its local state as a function of its current inputs and state. Finally, $\mathcal{C}$ presents and holds values on all its output channels, until the next system cycle begins.

There is a potential race condition if two or more components write to the same output channel. It is the *designer's* responsibility to see that this condition never arises. In any system cycle, the design must assure that exactly one[1] component presents output to a given channel.

I.3 Python

Python is a *scripting language* used for prototyping, software system control, and applications that involve interaction with the underlying file and operating systems. A Python program is similar in character to a "shell script" written in tcsh or bash, in that user-level commands are part of the language. Python is much more powerful, however, in that it provides for object-oriented style, symbolic data manipulation, and other features.

[1]As discussed later, one may in some cases relax this condition to "at most one"

As we shall see however, *any* language can be used to develop prototypes in ERTS. Later in the course, you may wish to transform Python models into higher performance languages. Conversely, Python my be too detailed a level of programming for your purposes. With some preparation, You can use much higher level languages, such as Matlab to develop and explore experimental models.

I.4 Communication

A novel aspect of the ERTS software environment is its treatment of communication. All transactions among components and with the vehicle are done in a uniform way through the file system. ERTS is mounted as a directory in the global file name space, and components are subdirectories. These file-like entities are memory mapped, so they act as global variables, but they are accessed by ordinary file I/O system calls.

In accordance with the design model these communications are synchronized. The synchronization mechanism is provided by the file system. Uniformity is supported by a collection of Python classes.

I.4.1 SySeFS or "SyncFS"

SySeFS stands for Synchronous-Sequential File System.[2] It is a modified version of the 9p network file system protocols that originated at Bell Labs. A *memory mapped* file-space region file is mounted by each computer in the ERTS network. Consequently, all file transactions take place in some node's primary memory and not on any hard disk. Communication is relatively fast, but some caution is needed to assure that dynamic memory allocation is bounded. Briefly, SySeFS works as follows:

- If a transaction involves a target file on another node, SySeFS uses TCP/IP to transfer data to or from the target.
- All file reads are performed asynchronously, as in a normal file system.
- All file writes are performed synchronously, that is, write calls are *deferred* to a *synchronization event* at which point all pending writes are performed (in an arbitrary order).
- SySeFS provides a sync call that blocks the caller until a global synchronization event occurs. This event is triggered after all pending writes are completed.

I.4.2 CartFS

CartFS.py is a framework for writing ERTS components in Python. Its signature is shown in Figure 2.

[2]It used to be called SyncFS until we learned that the name has been taken.

CartFSFile

encapsulates special properties of the files through which ERTS components communicate. The content of a CartFSFile is presented to *Python* as an association list, or "dictionary." Values are accessed by *key*. For instance the compass component's output file is specified as

Compass			
clock	*tick*	integer	cycle count
enable	*boolean*	boolean	compass enabled?
heading	*degrees*	real	direction relative to magnetic North

It has three values accessed by keys clock, enable and heading.

JSON. Were you to print the `compass` file the display would look like

```
{"enable": true, "heading": 137.306060932, "clock": 9979}
```

This *transport string* is expressed in JSON format [`www.json.org`] format, a simple, human readable, data description language. JSON is supported for many programming languages, including Python, C, Java, and many others. Using it as a transport language makes it easy to create systems using components written in different languages.

Sensor

is is class encapsulating ERTS components. This class manages all the I/O performed by its instances, which need only provide file names through calls to add_reader and add_writer. In essence, these routines connect the components together. Methods read_files and write_files perform file operations on all files that have been "registered" with add_reader and add_writer

The run method, below, fixes the cyclic behavior of a Sensor:

```
while 1:
  self.wait_for_clock()
  self.read_files()
  if not self.has_requirements(): continue
  self.process()
  self.write_files()
```

Compare this with the component design model in Section I.2. The only difference is that Sensor.run() checks whether all input files (channels) are present in the current cycle. Inputs may not be present while the system is starting up, or may disappear if a component fails.

CartFS Sensors execute synchronously through the mechanism `wait_for_clock()`. Once a component is released to perform a cycle, it reads all of its inputs at once. Then it performs its `process` function, and finally, writes all its results.

The default `process` method in `class Sensor` is empty:

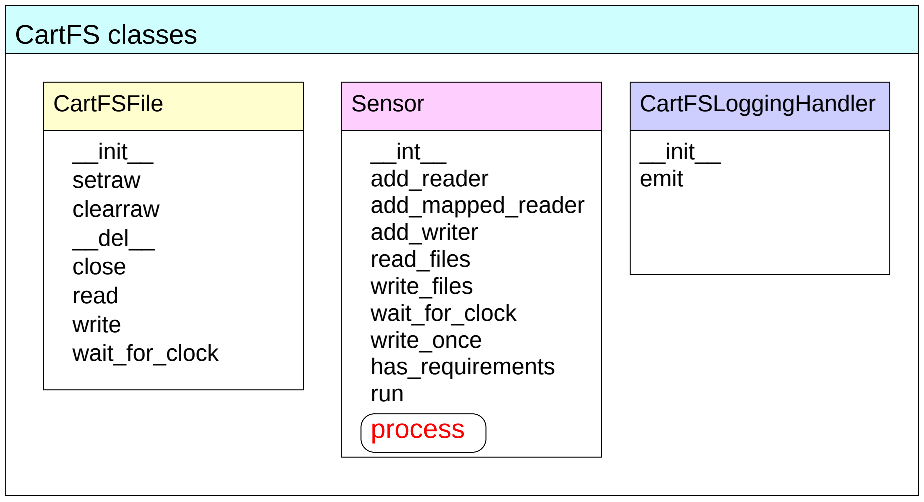


Figure 2: CartFS Signature

```
def process(self):
"""Process the current data and update state."""
    pass
```

Every Sensor instance must provide its own a definition for `process()`. In all early assignments, the connection specification is included in a skeleton program.

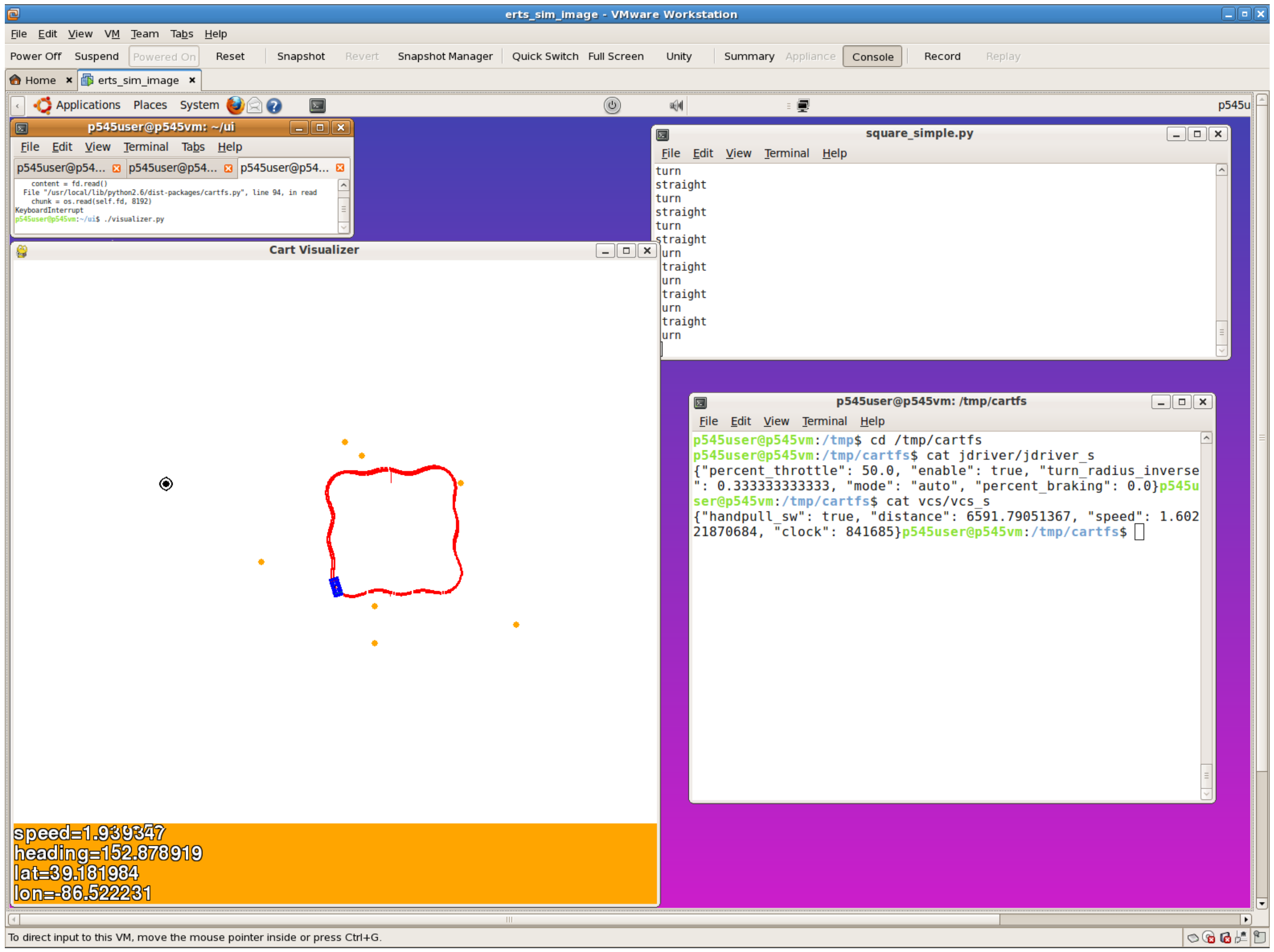
I.5 ERTS Simulator

ERTS has a simulator, CartSim that is accessed through CartFS in exactly the same way as on the ERTS vehicle. Figure 3 shows a snapshot of `cartsim` running a simulation. The `term` window to the right is running a navigation component, and the 3-tab `term` window to the upper left is running the simulator. The graphical window at the lower-left is a component called visualizer that displays the GPS position of the vehicle. Visualizer may be used either in the simulator or on the vehicle without modification.

Figure 3 illustrates that multiple-component systems run concurrently as ordinary linux processes, each often in its own terminal window. The `term` window at the lower-right shows that one may observe the communication behavior of the system using basic Linux commands like `cat`.

I.5.1 VMware

VMware (http://www.vmware.com/ is a virtual machine environment in which one computer may be used to emulate others. Emulation may be at the level of the operating system, or the underlying computer hardware, or both. In


erts_sim_image - VMware Workstation
File Edit View VM Team Tabs Help
Power Off Suspend Powered On Reset Snapshot Revert Snapshot Manager Quick Switch Full Screen Unity Summary Appliance Console Record Replay
Home erts_sim_image
Applications Places System
p545u
p545user@p545vm: ~/ui
Cart Visualizer
square_simple.py
p545user@p545vm: /tmp/cartfs
speed=1.939347
heading=152.878919
lat=39.181984
lon=-86.522231
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
    content = fd.read()
  File "/usr/local/lib/python2.6/dist-packages/cartfs.py", line 94, in read
    chunk = os.read(self.fd, 8192)
KeyboardInterrupt
p545user@p545vm:~/ui$ ./visualizer.py
```

```
turn
straight
turn
straight
turn
straight
urn
traight
urn
traight
urn
traight
urn
```

```
p545user@p545vm:/tmp$ cd /tmp/cartfs
p545user@p545vm:/tmp/cartfs$ cat jdriver/jdriver_s
{"percent_throttle": 50.0, "enable": true, "turn_radius_inverse
": 0.333333333333, "mode": "auto", "percent_braking": 0.0}p545u
ser@p545vm:/tmp/cartfs$ cat vcs/vcs_s
{"handpull_sw": true, "distance": 6591.79051367, "speed": 1.602
21870684, "clock": 841685}p545user@p545vm:/tmp/cartfs$ 
```

Figure 3: The ERTS simulator

P545, the lab desktops use VMware run a configuration Ubuntu^{TM} identical to that running on the ERTS vehicle. The simulator and its graphical display are installed and interact with software components in (almost) exactly the same way as the physical vehicle.

I.6 Other Tools

I.6.1 Telemetry Visualization

It is important to render test data in a visual form, for both analysis and presentation. In all test-runs, you will record sensor readings, including the GPS position of the vehicle. In this way you can plot other readings, such as steering error, against vehicle position (Fig. **??**). You may use any graphical tools you like to visualize your tests. A popular choice is GnuPlot [`http://www.gnuplot.info/`] because it is widely available.

I.6.2 Subversion

Subversion [`http://subversion.tigris.org/`] (SVN) is a version control system used as a repository for ERTS software. I also contains homework directories for each participant. You will receive instructions for using SVN in class.

I.6.3 Doxygen

Doxygen [`http://www.stack.nl/ dimitri/doxygen/index.html`] is a source-code documentation system. For projects and other reports, Doxygen generated documentation is required. You will receive instructions on its use later in the course.

II

Project Requirements

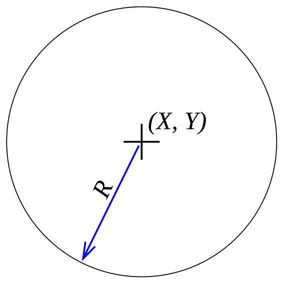
The P545 Lab Project is to design, implement, test and demonstrate a *navigation component* (NAV) or subsystem that steer the ERTS robotic golf cart along a sequence of lattitude-longitude points on the ERTS test field (Fig. 2.

The vehicle may encounter *obstacles* along its path and must steer as to avoid them while staying within the boundaries of the course.

II.1 Design Elements

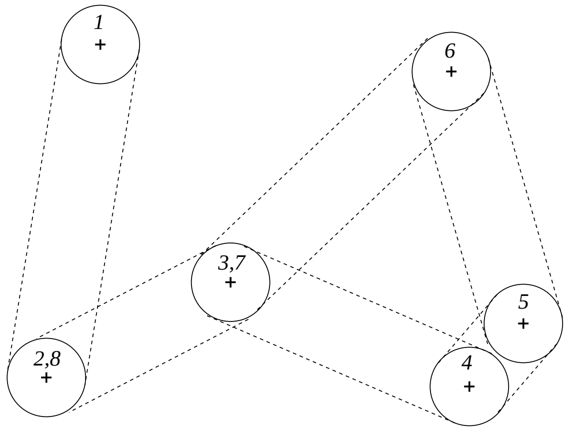
II.1.1 Waypoints

A *waypoint* is a positional reference used for navigation. It specifies the coordinates (X, Y) of point in degrees of lattitude and longitude (a "lat-lon" for short) and a radius R indicating how close to (X, Y) is "close enough."



II.1.2 Courses and Corridors

A list of waypoints defines a two-dimensional *course*, consisting of a sequence of segments, or *corridors*. A corridor is the area that results by "sweeping" a waypoint in a straight line to the next waypoint in the course.



II.1.3 Obstacles

Obstacles may appear at any point withing the boundaries of a course. For the basic Lab Project, these obstacles placed before a course run and are stationary throughout that run. In order to simplify detection and avoidance, obstacles are uniform: each is an orange traffic cone placed atop a five-gallon bucket.

II.2 Path Planning

NAV components should do more than simply react to instantaneous conditions. Examples of *path planning* include:

- Efficient navigation through tight turns. For example, a driver approaching a sharp left turn will "swing" to the right before reaching the turning point, in order to maximize the turning radius.
- When driving multiple laps through a course, particularly one that includes fixed obstacles, the vehicle should be able to traverse successive laps faster and more efficiently.

ERTS has both simulated and actual sensors that detect obstacles at a range of five to ten meters. They return the obstacle *bearing* (distance and angle relative the vehicle *heading*, or direction of travel) from the sensor, and an estimate of the obstacle's width. NAV must determine how to bypass this "negative waypoint" while remaining inside the current corridor (if possible), and take action to avoid hitting the obstacle.

Specification R1 (GPS Navigation)

> *The vehicle driver shall navigate any course within the confines of the test field, passing through each waypoint in sequence, and staying within the the course boundaries defined by the waypoint corridors.*

Specification R2 (Obstacle Avoidance)

> *While traversing a course, the vehicle shall avoid any obstacles it encounters, while remaining inside the boundaries of the course. In cases that it is impossible to avoid an obstacle and remain inside the course, the vehicle should stop.*

Specifcation R3 (Path Planning)

> *On a given course with stationary obstacles, higher evaluations are given to NAV components that show evidence of Path Planning.*

II.3 Project Report

To receive full credit for the P545 Lab Project,

- The software must be demonstrated and evaluated in the field for full credit. Simulated evidence is not considered.
- The design, implementation, and evidence of functionality must be presented in a written report, which may be accompanied by an oral presentation.

It is a waste of effort to put off writing a final report until the course is almost over. This final P545 Lab Project Report should contain a cumulative set of reports for each of the laboratory assignments. If individual lab reports are written carefully, it will take less effort to assemble them into a final report.

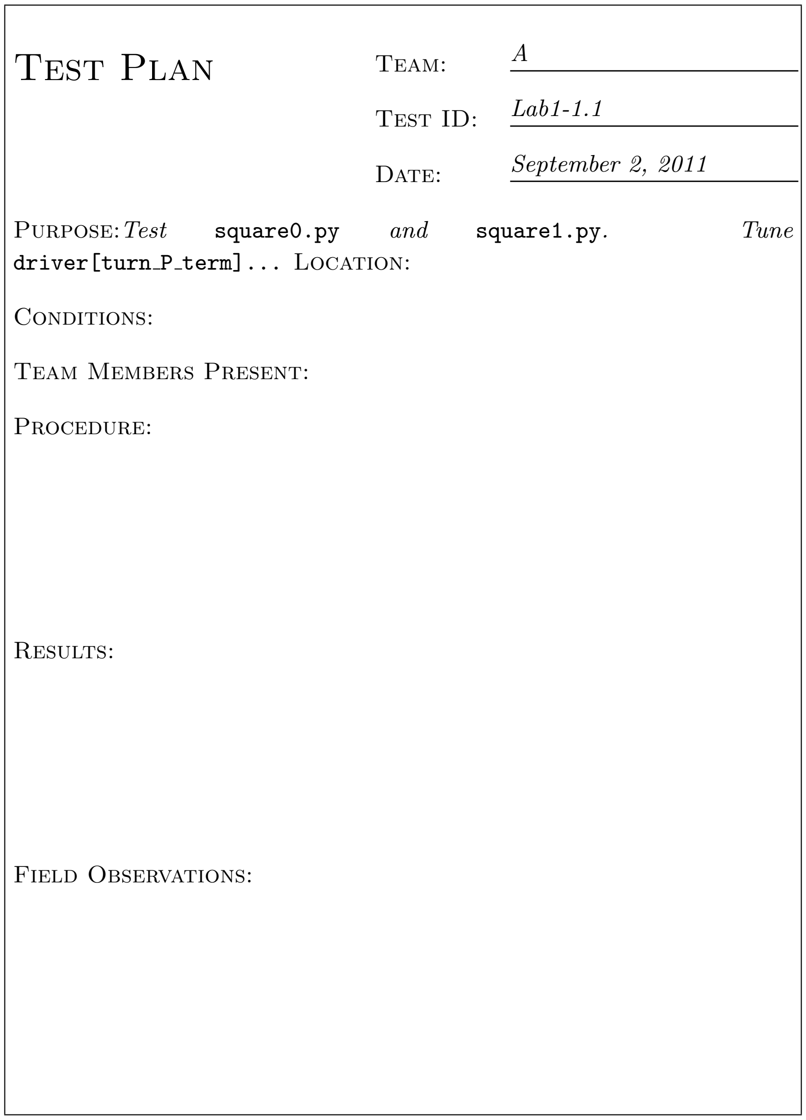
The primary purpose of the report is explaining the design and implementation of your solution to requirements listed in this section. It should also include a record of testing along the way, but, in the end, your report is concerned with your software as a whole.

II.3.1 Test Plans

Lab assignments provide instructions about what tests to perform, but it is good practice to write a *test plan* prior to each field test. Figure 1 shows an example format for such a test plan.

II.3.2 Test Reports

II.3.3 Final Report



TEST PLAN	
TEAM:	*A*
TEST ID:	*Lab1-1.1*
DATE:	*September 2, 2011*

PURPOSE: *Test* `square0.py` *and* `square1.py`*. Tune* `driver[turn_P_term]`... LOCATION:

CONDITIONS:

TEAM MEMBERS PRESENT:

PROCEDURE:

RESULTS:

FIELD OBSERVATIONS:

Figure 1: A Test Plan

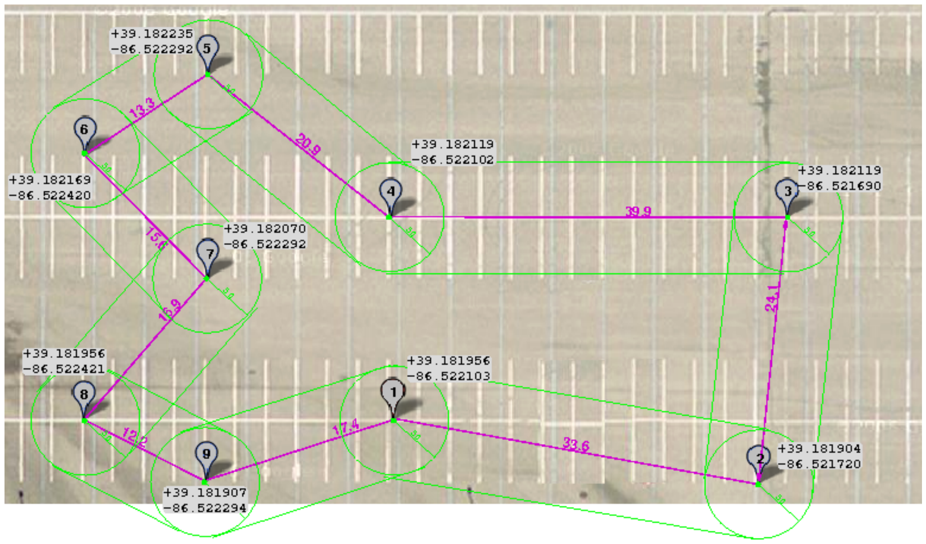


Figure 2: P545 Test Field with GPS course

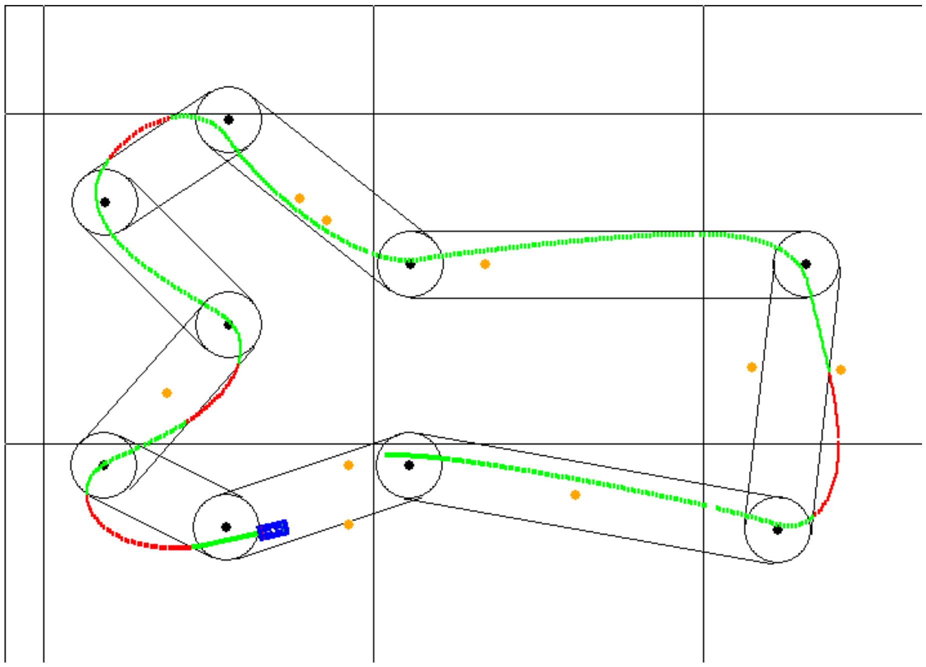


Figure 3: *Visualizer* trace of a simulated GPS follower with obstacle avoidance

Lab 1

Meet ERTS

Your first laboratory assignment introduces you to the robotic vehicle ERTS, the laboratory platform for this course. ERTS is a standard electric golf car, modified for computer control. The primary tools used to develop ERTS navigation components are used in this first assignment.

1.1 Objectives

You are to tune a `Python` `driver` component that navigates a square course repeatedly (Fig. 1.1).

1. Use the ERTS simulator to tune the value of a constant that contols how aggressively to correct steering.
2. Perform a field test on ERTS to fine-tune that constant.
3. Record vehicle telemetry, including GPS position and steering error.
4. Plot the vehicle path and compare it to the simulated path.
5. Write a report explaining any discrepancies.

1.2 Design

A standard design model for controllers is a finite state automaton, depicted in Figure 1.2(a) as a control-flow diagram rendered in *algorithmic state machine* (ASM) syntax. Boxes represent control states; diamonds represent state-transition decisions; and ovals represent transitional actions. The design for the `square` driver has two control states.

- In its `turn` state, `driver` is negotiating a corner of the square. It continues to turn until ERTS's direction of travel, or *heading*, reaches the desired compass point, North, East, South or West.

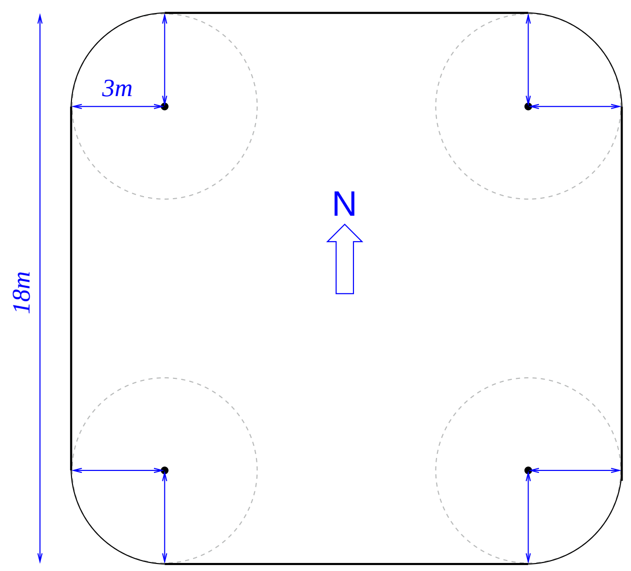


Figure 1.1: The *square* course

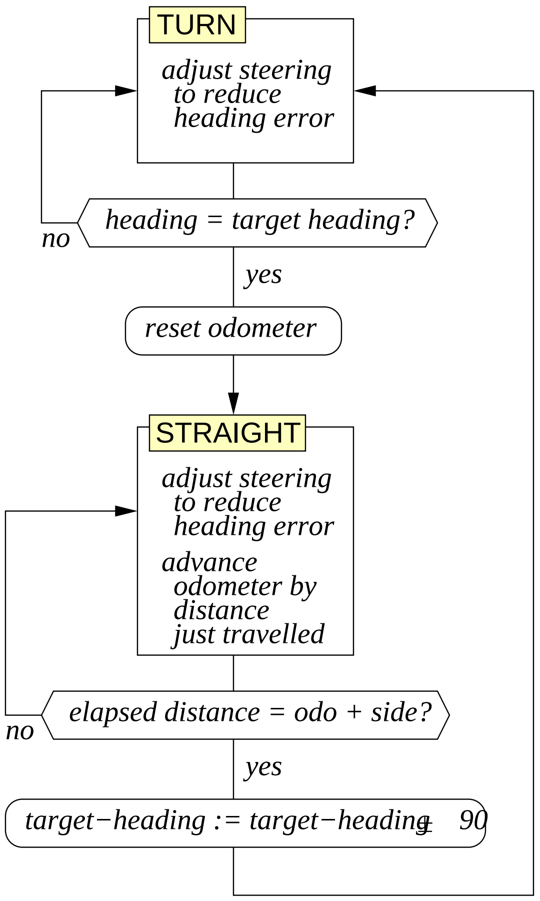
- In its straight state, ERTS is traveling along a side of the square. Driver remains in this state until ERTS has reached the next turning point.

While in either state, driver steers to correct its true heading relative to the target heading. The ASM refers to two components of the driver's local *data* state: target-heading is the desired direction of travel; odometer is the cumulative distance traveled. When driver from turn to straight state, it resets *odometer*, and when it goes from straight to turn state it changes it's *target-heading*, from North to East, East to South, etc.

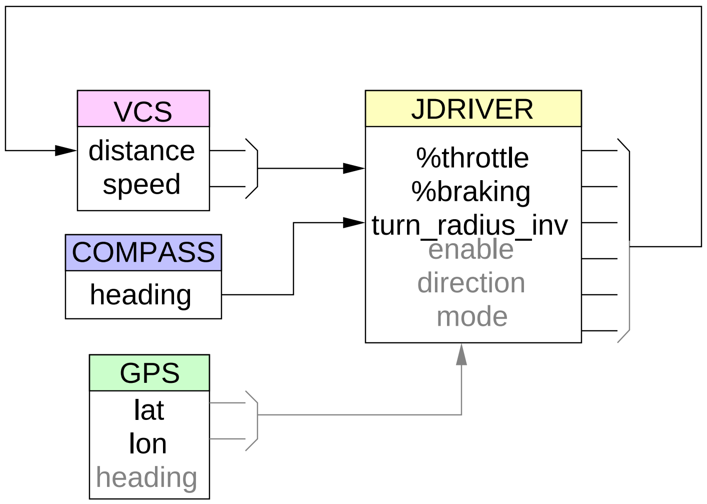
Driver runs concurrently in a system of other components, including compass that reads a compass sensor, gps that reads a GPS sensor, and vcs the *vehicle control module*. Figure 1.2(b) shows the system architecture, that is, how the components are connected. In every system step each of these components reads inputs, computes and emits outputs, updates its data state, and decides what its next control state should be. Conceptually, the components are running in parallel, but in reality the host operating system is choosing a (possibly) linear order of exectution.

1.3 Vehicle Status (Sensors)

The components mentioned in the previous section are describe in more detail below.


TURN
adjust steering to reduce heading error
heading = target heading?
no
yes
reset odometer
STRAIGHT
adjust steering to reduce heading error
advance odometer by distance just travelled
elapsed distance = odo + side?
no
yes
target−heading := target−heading ± 90

(a) control



(b) logical system architecture

Figure 1.2: Square-driver design

1.3.1 Compass

The compass component interfaces to a PNI V2XE compass device that measures the vehicle's *heading*, or direction of travel. The `heading` value is returned in decimal degrees with 0=north, 90=east, 180=south, and 270=west.

COMPASS			
clock	*tick*	`int`	cycle count
enable	*boolean*	`bool`	compass enabled?
heading	*degree*	`float`	direction relative to magnetic North

The SySeFS file interface to COMPASS looks like

```
{"enable": true, "heading": 137.306060932, "clock": 9979}
```

1.3.2 Odometer

The *Vehicle Control Module* (VCS) is the digital interface to the stock sensors and actuators in the ERTS vehicle. The VCS component has the (partial) signature shown below. One of its functions counts revolutions of the cart's motor to measure distance travelled as a running total of elapsed meters since the cart was last powered on.

VCS			
clock	*tick*	`int`	cycle count
distance	*meters*	`real`	distance traveled since start-up
speed	*km/hr*	`real`	vehicle speed
handpull_sw	{on, off}	`bool`	status of the handpull switch

The SySeFS file interface to VCS looks like

```
{"handpull_sw": true, "distance": 100.85311581, "speed":
            2.44885968334, "clock": 3457}
```

1.3.3 GPS

ERTS's GPS sensor is a Garmin GPS18 5Hz. It measures absolute global position along with speed and heading when the vehicle is in motion. For this assignment, you will be using the position measurement to quantify the performance of your driver.

gps			
clock	*tick*	`int`	cycle count
lat	*degrees*	`float`	current lattitude
lon	*degrees*	`float`	current longitude
heading	*degrees*	`float`	direction relative to true North

The SySeFS file interface to VCS looks like

```
{"lat": 39.1823418929, "speed": 2.26470957754, "lon":
-86.5219751882, "heading": 81.0308328403, "clock": 188657}
```

1.4 Vehicle Command

Square is an instance of the jdriver component, which indirectly controls the vehicle's mechanical actuators by writing commands to a predetermined location at `jdriver_s`. These commands are read by other components, including the VCS, which translate symbolic commands to voltages governing the actuators. The JDRIVER signature is

JDRIVER			
clock	*tick*	`int`	cycle count
enable	*boolean*	`bool`	computer control enabled
percent_throttle	*percentage*	`float`	throttle control
percent_braking	*percentage*	`float`	brake control
turn_radius_inverse	*meters*	`float`	steering control
direction	{forward, backward}	`string`	direction of travel relative to the vehicle
mode	{auto, manual}	`string`	vehicle control mode handshake

The SySeFS file interface to JDRIVER looks like

```
{"clock": 0, "enable": false, "percent_throttle": 0.0,
"percent_braking": 0.0, "turn_radius_inverse": 0.0, "direction":
"forward", "mode": "manual"}
```

NOTE: Your `driver` component must produce and write a string of this form on every cycle

1.4.1 Throttle

For this lab, the throttle control is held at a a fixed value throughout a run of the course. A value around 60% is a reasonable starting point, but may need to be adjusted at the field. Ask experimentors that have gone before you what setting they would recommend.

A fixed throttle value does *not* result in a constant vehicle speed. Speed fluctuates as the load on the drive motor changes. For instance when driving up hill the vehicle slows down if the throttle is held in one position.

The VCS component reads a desired throttle setting from the percent_throttle value in `jdriver/jdriver_s`. In turn, VCS manipulates the voltage supplied to the vehicle's drive motor, which determines how much mechanical power is appied to the wheels.

1.4.2 Steering

You will be using the steering actuator to control the heading of the cart. Steering is specified as the *inverse of turn radius* in meters. For example, if you want to turn an arc that is 20 m in radius, you set `jdriver_s[turn_radius_inverse]` to $\frac{1}{20} = 0.050$.

The turn radius is negative for left turns and positive for right turns. Writing 0 commands an effectively infinite turn radius for straight driving.

1.5 Implementation

The *square* driver is written in *Python*. For this assignment you are given two, functionally identical verstions.

`square_raw.py` Is a version that has been stripped down to a minimum so you can most easily see what is going on.

`square_sensor.py` Is an instance of the CartFS Sensor class, which provides a general framework for component develop. Sensor manages communications with other components in the system, including set-up and cycle-by-cycle interaction.

All components perform their cycles ten times per second. The process routine in `square` is this cycle.

NOTE: `process` is the only portion of code that requires modification. The two required modifications are:

1. Tune the value of constant `steering_sensitivity` to experimentally improve driving performance.

2. On each cycle, record for later analysis the current vehicle speed, postion, actual heading and *heading error*—the difference between the desired and actual direction of travel.

You will ride in ERTS as you test your driver is running. Write down your observations. From these, decide how to change the steering constant to improve performance. Retest and record write down your observations.

1.6 ERTS Simulator

You are encouraged to use the simulator to obtain a reasonable initial estimate for `turn_P_term`, and to make sure your coding changes execute properly. Interfacing with the simulator is exactly[1] like interfacing with ERTS. To use the simulator,

1. Sign on to any computer in LH035. Open a `term` window.

2. Run `vmware` and select the Linux virtual machine.

[1] Almost exactly. There may be formatting discrepancies. These will be corrected fixed as they are found.

3. Log in to the virtual machine with user ID and password `p415user`.
4. Create a top-level directory with your CS network user ID as its name. Keep copies of *all* your work in this directory so as not to interfere with other students using the same virtual machine.
5. Check out a working copy of your SVN directory, including your versions of `square_raw` and `square_sensor.py`.
6. In a separate `term` window, start `cartsim`. The simulator has a graphic display. You will find it in `~/ui/visualizer.py`.
7. Run one of your `driver` components.

NOTE: The ERTS simulator is a development tool. Importantly, it can be used to determine whether your programs will run properly on the real ERTS vehicle. The simulator *does not* contain an accurate model of vehicle dynamics, nor does it even consider the terrain of the test field. Thus, simulation can usually "get you close" to a solution—usually—but do not expect it to predict behavior in the real world. Solving an assignment in the simulator carries no credit. Credit is given only for demonstration on the ERTS vehicle itself.

1.7 Lab Report

See the Test Plan on the following page. Your report assumes (does not have to repeat) the content of design and implementation descriptions given in this assignment. Later, when you are responsible for design and implementation documentation should be part of the report.

It should focus on field observations and subsequent telemetry analysis. An crucial element day-to-day reports are the observations you make in real time during testing. It is very important that you write these down as they occur, because you will forget or dismiss them later. It is from these observations, often, that hypotheses for future testing arise. In both research and development, maintaining a chronological record of your activities is good practice.

Don't hesitate to take pictures and video clips, if you have the means. These can save the effort of later trying to write descriptions, and they help document the work. However, recordings cannot capture what you are thinking, so write that down.

1.7.1 Visualization and Analysis of Telemetry

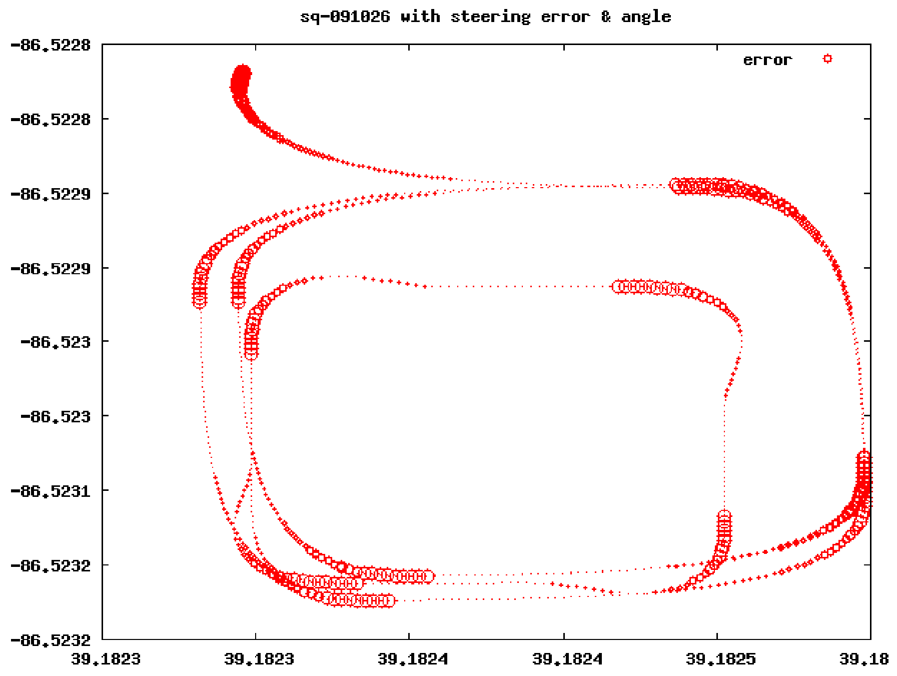
It is important to render test data in a visually meaningful form, for both analysis and presentation. In all test-runs, record salient sensor readings, including the GPS position of the vehicle. In this way you can plot other readings against vehicle position. You may use any visualization tool you like for these purposes, but a popular choice is *GnuPlot*[2]

A fragment of the telemetry recorder for an ERTS field test is shown below.

```
#TICK  LAT                    LON   DIST   RAD                ANG     ERR
26392  39.182295536 -86.522839268  284.4  1.17 1021.66  82. -0.287  -75.62 0.004
26393  39.182295832 -86.522840568  286.4  1.20 1021.78  82. -0.280  -73.65 0.004
26394  39.182296169 -86.522841884  288.3  1.22 1021.90  82. -0.273  -71.68 0.004
26395  39.182296550 -86.522843214  290.3  1.25 1022.02  82. -0.265  -69.72 0.004
26396  39.182296976 -86.522844555  292.2  1.28 1022.14  82. -0.258  -67.77 0.004
26397  39.182297446 -86.522845906  294.2  1.30 1022.27  82. -0.251  -65.82 0.004
26398  39.182297961 -86.522847263  296.1  1.33 1022.40  82. -0.244  -63.90 0.004
26399  39.182298522 -86.522848625  298.0  1.36 1022.53  82. -0.237  -61.98 0.004
26400  39.182299130 -86.522849988  299.9  1.38 1022.67  82. -0.230  -60.09 0.004
```

Here is a *GnuPlot* script that plots steering error (red circles) against position. Its output is just below.

```
set title "sq-091026.dat"
set term png
set output 'sq-091026a.png'
set title 'sq-091026 with steering error & angle'
plot 'sq-091026.dat' using 2:3:(abs($9)/50) with points lt 1 pt 6 ps variable lc rgb "red" title 'error',\
```



As you can see, the plot is a clear representation of steering sensitivity. When the target heading changes at the corners, the error immediately gets large and then smoothly diminishes with successive steering adjustments. There is not much over-steer or under-steer.

Observation. The telemetry shows substantial precession in the overall vehicle path. This is likely due to . . . *What?*

[2]http://www.gnuplot.info/

Test Plan

Team: ____________________

Test ID: ____________________

Date: ____________________

Purpose: *Field Test* `square_raw.py` *and* `square_sensor.py`. *Tune* `nav['steering_sensitivity']`
Location: *ERTS Test Field*

Conditions:

Team Members Present:

Procedure:

1. *Copy the* `square_raw.py` *and* `square_sensor.py` *programs onto a portable flash drive.*
2. *On the ERTS host computer, create a directory at top level named with your login ID or team ID. Load and run the* `square_*.py` *drivers on ERTS at the P545 test field.*
3. *Record telemetry data, including GPS position, speed, and* steering error.
4. *Tune value of* `nav['steering_sensitivity']` *to improve steering feel and qualitative performance.*

Results:

- *Number of test runs*
- *Location and names of telemetry files*
- *Final value of* `nav['steering_sensitivity']`
- *. . .*

Field Observations:

Lab 2

Basic GPS Following

Let's get started on implementing the Lab Project (Ch. II) In the `square` driver, you designed a heading controller that adjusts the turn radius of the steering subsystem on each clock cycle to minimize the difference between the current compass heading and the desired heading. In this lab you will use the same controller to follow a course defined by list of points in the Earth coordinate system, lattitude and longitude.

Where in Lab 1, steering corrections are based on four fixed headings, 0deg, 90deg, 180deg, and 270deg, for this assignment your component is given a list of GPS *positions* $P = (P_{\text{lat}}, P_{\text{lon}})$ called *lat-lons* for brevity. Once the heading is correctly oriented to a point, the steering controller holds this course until ERTS is "close enough" to P. Once it is close enough, ERTS is steered toward the next lat-lon in the list.

2.1 Assignment Overview

Design a GPS driver that follows GPS course as depicted in Figure (3.2). Your driver is given a list of waypoint specifications (See Sec. 2.3, below).

```
course = [[1,39.181917,-86.5221208333,1.5,3.0],\
          [2,39.1818975,-86.521724,1.5,3.0],\
          [3,39.182143,-86.5217033333,1.5,3.0],\
          [4,39.182199,-86.5220985,1.5,3.0],\
          [5,39.1819156667,-86.522309,1.5,3.0],\
          [6,39.1819645,-86.522398,1.5,3.0],\
          [7,39.1820415,-86.5223095,1.5,3.0],\
          [8,39.1821313333,-86.5223926667,1.5,3.0],\
          [9,39.1822116667,-86.522302,1.5,3.0]]
```

The objective is to traverse the course determined by this list and analyze the path ERTS takes. When the cart reaches the end of the list, start again at the beginning and continue as before until the safety driver overrides computer control and stops the vehicle.

2.2 What to turn in

Feel free to work in groups. In fact, it is encouraged. Feel free to share your observations with others on the test field and ask them to share theirs' with you. We're all doing this together.

- Devise a way to plot the waypoints and corridors along with telemetry. The vehicle path is meaningless without this frame of reference.

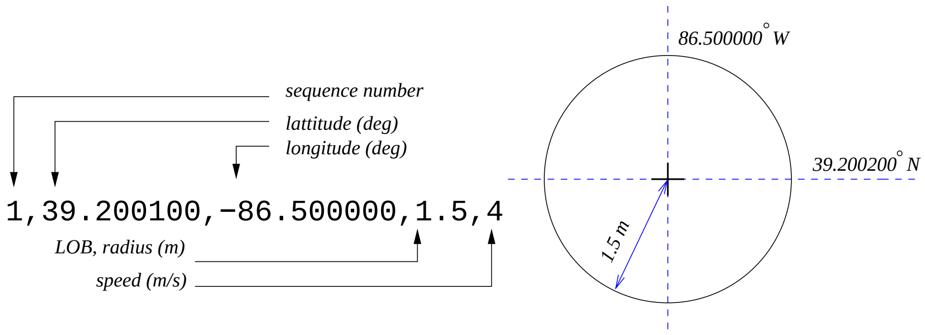
- Design, develop and test a *GPS-follower* component that navigates through `course`. You may (and should) use the same steering-control function that was given in Lab 1.
- Write a an implementation document describing your solution.
- Perform a field test of your GPS follower, that traverses `course` three times. Record and plot positional data. With a pencil, add to the plot your estimate of an optimal course.
- Write a concise summary of testing that was performed including field observations and results. Discuss why your driver's path differs from the optimal path.
- Post the results to the SVN under `class/fall10/your-id/lab/2/`.
-

Suggestions.

- As you begin testing your driver, shorten the course to two or three waypoints, in order to conserve time for other teams. Use the full course only when you are confident in the correctness of the code.
- As in Lab 1, use a fixed throttle setting that makes the average speed of around 3 m/s. A throttle percentage of around 60% is a good starting point. Once you have completed the assignment, feel free to explore manipulating the throttle, say, to maintain a constant speed.
- Clip the turning radius at 2.5 meters (*inverse_turning_radius* ≤ 0.4). Once you have completed the assignment, feel free to manipulate (within reason) the turning range.

2.3 Waypoints

A *waypoint* is a positional reference used for navigation. The waypoint format and information is based on the specification for the 2005 DARPA Grand Challenge "Race Across the Desert" [1].



There are five information components.

1. *Sequence number*, a positive integer that uniquely identifies the waypoint. Multiple waypoints with the same positional information will have different sequence numbers.
2. *Latitude*, a numeral with 7 decimal places in the range $[-90, 90]$ representing degrees of latitude.
3. *Longitude*, a numeral with 7 decimal places in the range $[-180, 180]$ representing degrees of longitude.
4. *Lateral Boundary Offset* (LBO), a number with representing the radius in meters of a circle centered at the given lat/lon. The LOB specifies how near the vehicle must be in order to be "close enough."

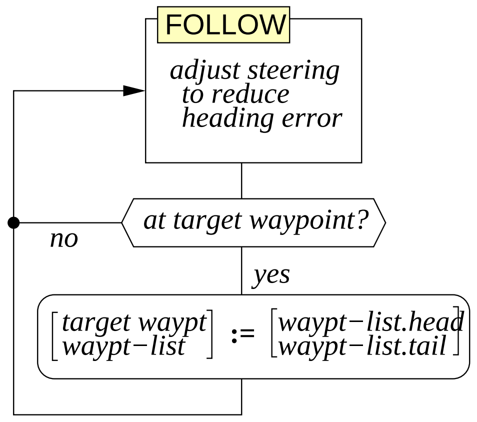

FOLLOW
adjust steering to reduce heading error
at target waypoint?
no
yes
[target waypt / waypt−list] := [waypt−list.head / waypt−list.tail]

Figure 2.1: Simple GPS follower control

5. *Speed limit*, a number specifying the maximum speed in meters per second allowed between this waypoint and the next.
6. A *lateral boundary offset*, or *LOB*, defined to a circle centered on the lat/lon whose radius,

2.4 Design

??

The *square* driver was designed as a two-state control automaton (Fig. **??**). At any time ERTS was either turning or driving straight, and the criteria for changing state was different. The GPS follower has only one state (Fig. 2.1). At all times it is trying to reach the target waypoint. When it does reach the target waypoint, it simply selects a new target.

2.4.1 Heading

The architecture of the GPS-following system is the same as that of the *square* driver (Fig. **??**). However, both the GPS component and the Compass component provide a heading value. One important difference is that the GPS heading is only accurate when the vehicle is in motion, while the Compass is always accurate. Even though the GPS is wrong at the beginning of a run, it is worthwhile to use it instead of the Compass.

2.5 Angle and Distance to a Waypoint

In the design of your GPS driver, you will need to take into account the *bearing* (angle, or azimuth) and distance to the target waypoint. The test field is small, compared to the circumference of the earth, and one may perform this computation using plane geometry. However, there are a couple of potential problems in performing the calculation.

1. Latitude and longitude are measured in *degrees*, but the vehicle provides elapsed distance in terms of meters.
2. The conversion from degrees of latitude to meters is fixed. There are approximately 111,122 meters per degree. However, the conversion from degrees of longitude to meters *depends on the latitude*. Our test field is located at approximately 86 degrees West,

39 degrees North. At 39 degrees North each degree of longitude corresponds to about 86,358 meters.

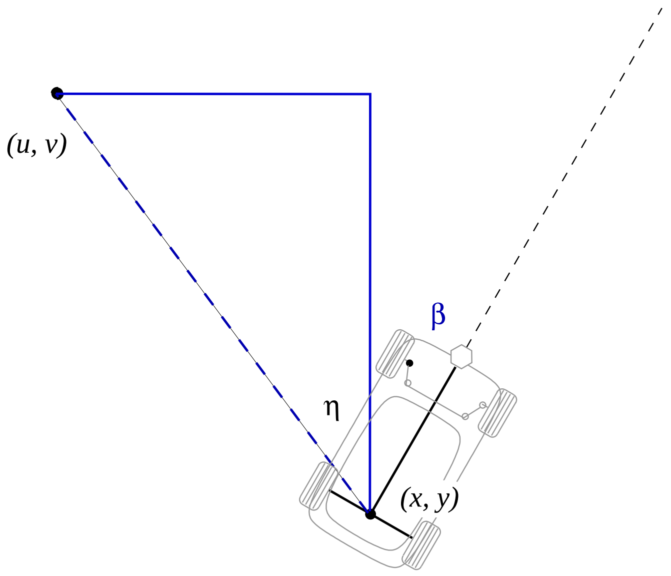
3. Of course, the surface of the Earth is not flat (Right?). For large distances, planar geometry is inaccurate.

The distance-azumith calculation is developed according to the current vehicle position, (x, y), and the next-waypoint position, (u, v). Planar distance is

$$\sqrt{(u-x)^2 + (v-y)^2}$$

and the bearing is

$$\eta = \arctan\left(\frac{u-x}{v-y}\right)$$



Fortunately, *Python* provides a module, geopy [2] that performs sufficiently accurate calculations for "geocoding." You may make use geopy to estimate distance and bearing, or you may challenge yourself to design the calculations yourself (Sec. **??**.Starting with geopy is recommended for this assignment:

```
from geopy import distance

current_latlon = (39.181903,-86.522041)
waypoint_latlon = (39.182169,-86.522007)

meters_to_target = distance.distance(current_latlon,waypoint_latlon).kilometers * 1000
heading_to_target = distance.distance(current_latlon,waypoint_latlon).forward_azimuth
```

2.6 References

[1] DARPA, *DARPA Grand Challenge 2005. Route Data Definition File.* Aug Report August 3, 2005.

[2] Goggle. *geopy: A Geocoding Toolbox for Python.* Getting Started guide. Updated July 2, 2010.

Test Plan

Team:	____________
Test ID:	____________
Date:	____________

Purpose: *Field test a basic GPS follower.* Location: *ERTS Test Field*

Conditions:

Team Members Present:

Procedure:

1. Load and execute the driver on ERTS.
2. Record and save telemetry, including steering error.
3. Tune parameters for steering sensitivity, turn starting point, steering radius, etc.

Results:

Field Observations:

Lab 3

Path Planning

In the second assignment you developed a driver component capable of passing through an orderd list of GPS waypoints. The goal was simply to “touch” each waypoint in sequence. In this assignment, you are to refine your driver to *keep with in the course boundaries*, as discussed below. This is an open-ended assignment. There are many ways to approach it and there is no perfect solution. As you progress in the lab project you will think of many ways to improve what you have.

3.1 Courses and Corridors

A given list of waypoints defines a two-dimensional *course*, consisting of a sequence of segments, or *corridors*. A corridor is the area that results by “sweeping” a waypoint in a straight line to the next waypoint in the course. In other words, it is a rectangle whose length is the distance between two waypoints and whose width is twice the LOB of the first waypoint. A semi-circle with radius equal to the first waypoint’s LOB is added at both ends of the rectangle.

Figure 3.1 illustrates the corridors in a course of four waypoints having different LOBs. The vehicle is traveling in a counter-clockwise direction. A clockwise traversal would result in different corridors.

NOTE. In the default test course used in this and the previous assignments, all LOBs are initially equal

3.2 Staying In The Lines

Figure 3.2 is a screen shot of the ERTS visualizer tracing the path of a point-to-point GPS follower. It renders positions within the course boundaries in green and postions outside the boundaries in red. The image shows that the vehicle has difficulties negotiating turns. The sharper the turn, the more it strays outside corridor boundaries.

To solve this problem, your driver will have to “think ahead.” There are many ways to do this, so think carefully. Do not expect to solve this problem, or even understand it, in a few weeks. On the other hand, it is important to *improve* your driver as a means of better understanding all aspects of the problem.

For instance, if you have an algorithm for the geometry of path planning, how are its results conveyed to the vehicle control system? In some approaches, the simple point-to-point GPS follower isn’t changed at all. Instead, the planning procedure might modify the course! Figure 3.3 illustrates a new course (in green) charting a more valid path through the original.

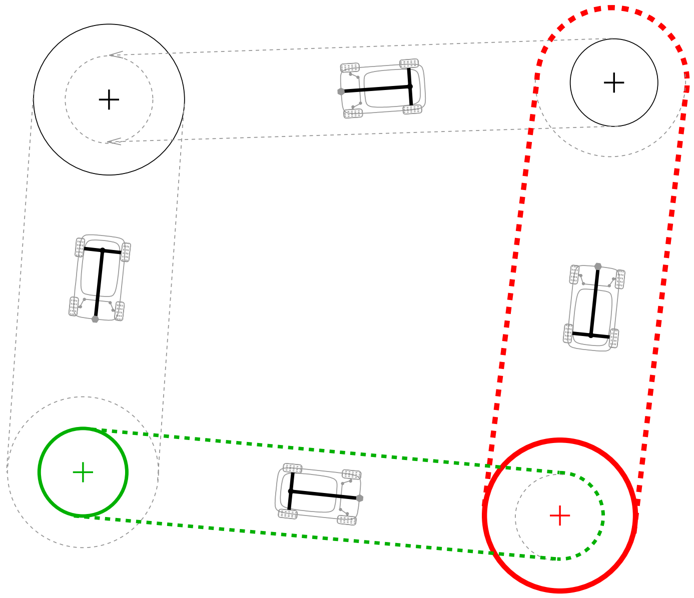


Figure 3.1: A *course* determined by four waypoints

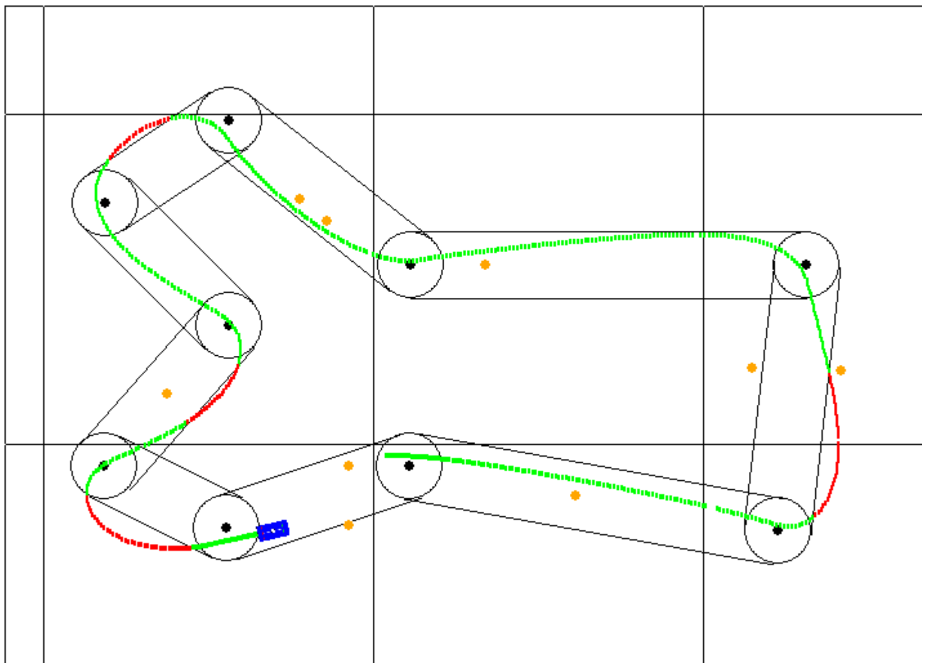


Figure 3.2: *Visualizer* trace of a simple GPS follower

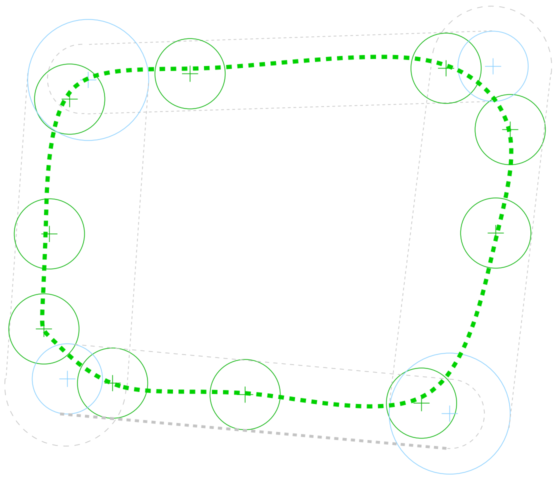


Figure 3.3: A planned path through the original course.

3.3 What to turn in

Use the same course as was given in the previous lab

```
course = [[1,39.181917,-86.5221208333,1.5,3.0],\
          [2,39.1818975,-86.521724,1.5,3.0],\
          [3,39.182143,-86.5217033333,1.5,3.0],\
          [4,39.182199,-86.5220985,1.5,3.0],\
          [5,39.1819156667,-86.522309,1.5,3.0],\
          [6,39.1819645,-86.522398,1.5,3.0],\
          [7,39.1820415,-86.5223095,1.5,3.0],\
          [8,39.1821313333,-86.5223926667,1.5,3.0],\
          [9,39.1822116667,-86.522302,1.5,3.0]]
```

Feel free to work in groups. An easy way to visualize your tests is using screen-shots capturing the visualizer.py window. As is seen in Figure 3.2, it correctly draws corridors and colors the cart path trace.

- Design, develop and test a *course follower* component that navigates through a `course` and, perhaps imperfectly, stays inside corridor boundaries.
- Write a design document describing your approach and analyzing test results.
- Perform a field test, recording positional data.
- Post the results to the SVN under `class/fall10/your-id/lab/3/`.

3.4 Suggestions

3.4.1 Plotting Courses

For extra credit, devise and share with the class a *GnuPlot* script that plots the boudaries of a `course`. This would really be helpful for those of us that use *GnuPlot*.

3.4.2 RDDF files

The *Python* fragment below shows how to read a route definition data file (RDDF) into a list. This code is extracted from the visualizer source in `~/ui/rddf.py` on ERTS, or the VMware image. You should consider adding this capability to your driver.

```
def read_rddf(rddf_file='RDDF'):
    rddf_data = None
    try:
        rddf_fd = open(rddf_file, 'r')
        rddf_data = rddf_fd.readlines()
    except IOError:
        print "Sorry, could not open", rddf_file
        return(None)

    # course is a waypoint list
    course = []
    if rddf_data != None:
        for line in rddf_data:
            line = line.rstrip()
            (index, lat, lon, lbo, speed) = line.split(',')
            course.append((int(index), float(lat), float(lon), float(lbo), float(speed)))
        return course
```

3.5 Obstacles

In the next assignment you will begin to incorporate obstacle avoidance in your driver. ERTS's primary obstacle sensor, is a SICK LMS100 laser range-finding device. It's graphical user display is shown in Figure 3.4. One the test field, place three traffic cones in front of the vehicle at different distances and bearings. Print the *raw* readings from the laser and save them on your memory device.

Figure 3.4 shows an example of the readout. The list of values associated with key `"raw_laser"` represents the range readings from a bearing of $-150°$ to $+150°$, relative to the current heading. The value 64000 denotes *no reading.* Remaining values are in units of millimeters.

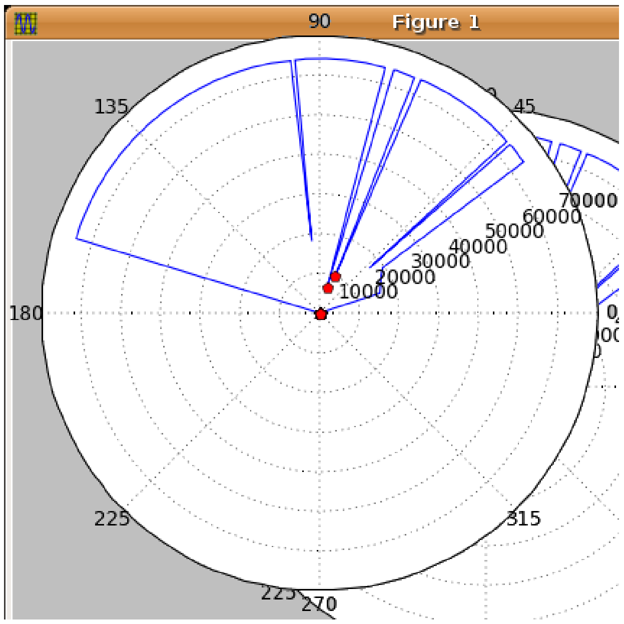
k this

Study these readings and think about how you would derive obstacle ranges and bearings from them. The lazer component attempts to do this, returning of list of possible obstacles with the key `"obstacle_list"`. It does this using a simple scan of the raw readings, assuming that the only obstacles are traffic cones. Obstacle positions are given in GPS coordinates.

3.6 References

[1] DARPA, *DARPA Grand Challenge 2005. Route Data Definition File.* Aug Report August 3, 2005.

[2] Goggle. *geopy: A Geocoding Toolbox for Python.* Getting Started guide. Updated July 2, 2010.



```
{"raw_laser":
  [ 0.0,
    15982, 16055, 16084, 16174, 16227, 16259, 16367, 16473, 16522, 16610, 16681, 16759,
    16823, 16918, 16962, 17027, 17143, 17239, 17346, 17451, 17542, 17626, 17752, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000,  9911,  9884, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000,  6679,  6675,  6688, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    18980, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000, 64000,
    64000, 64000, 64000,
    0.0],
"obstacle_list":
  [["obs", [39.1819068009, -86.5220214353], 0.25],
   ["obs", [39.1819277773, -86.5220490907], 0.25]],
"clock":
  30705}
```

Figure 3.4: A reading from the SICK LMS100 laser range-finder. The graphical display (of a different reading) is shown above.

Lab 4

Obstacle Avoidance

In this assignment you are to steer ERTS to avoid hitting detected *obstacles*, remaining within the course boundaries if possible.

In the last assignment you were asked to record readings from the `laser/laser_s` file to obtain information about objects in the vehicle path. For this stage of development, the only kind of object is a traffic cone, which has a simple "signature" in the laser scan. In fact, the laser handler scans the raw bearing-distance readings and produces a list of obstacles with the key obstacle_list.

```
{"raw_laser": [ 0.0, 15982, 16055, ...  0.0],
  "obstacle_list":
    [["obs", [39.1819068009, -86.5220214353], 0.25],
     ["obs", [39.1819277773, -86.5220490907], 0.25]],
"clock":
  30705}
```

You may use the raw readings if you wish, but it will be easier to use the obstacle list. At least it will be easier at the start. Each member of obstacle_list contains an identifier, lat-lon coordinates, and an LOB. The LOB is always `0.25`.

You already know how to use geopy to find the distance and bearing to a target GPS position. Now, instead of steering toward the target, you need to steer away from it.

You can modify your path planning solution to account for obstacles. A good (i.e. recommended) approach is to insert a waypoint next to the obstacle that causes the vehicle to bypass it. This is depicted in Figure fig:bypass.

Figure 4.2 is the layout of the test course. LOBs have been expanded from previous tests to all more maneuvering room.

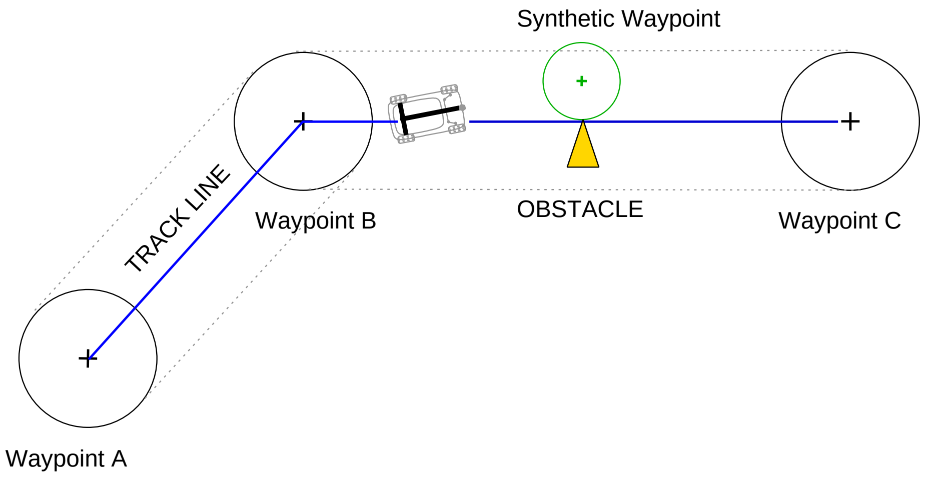


Figure 4.1: Adding a waypoint to bypass an obstacle

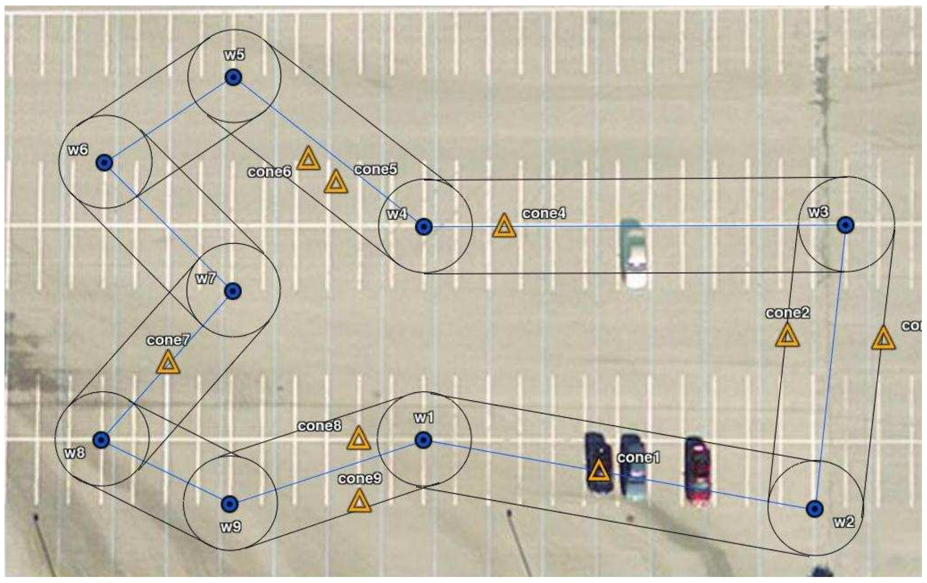


Figure 4.2: Test course

Lab 5

Midterm Field Trials

This week you are to schedule a field trial of your GPS follower on the ERTS test field. Submit the results of your trial(s) and a report describing your /sf GPS driver's design. Trials are run as follows:

1. Inform the Test Driver (Caleb) that you would like to run a field trial for evaluation.
2. The Test Driver may give you a new course. Be prepared to load that course into your driver. Your driver should also be configured to log positional data.
3. Once the vehicle is in position start the visualizer.
4. Load your driver and start execution. Then leave the vehicle.
5. When it is safe to do so, the Test Driver will give control to your driver.
6. the driver it should navigate three laps of the course, then set the *throttle* to 0% and the turning radius to 0 and terminate.
7. Capture a screen shot of the visualizer.

Evaluation of a trial run is based on the following:

1. Three laps must be completed, after which the vehicle rolls to a stop.
2. Missing a waypoint is penalized.
3. Traveling outside the course boundary is penalized.
4. Smooth steering is rewarded.
5. Faster lap times are rewarded.
6. Smooth and appropriate speed control is rewarded.

Appendix A

System Documentation

A.1 Organization of System Documentation

Section level organization of system documentation is given in Figure **??**. The remainder of this section discusses the purpose and content of each section. Generally, each section develops a successively more concrete view of the target realization. In general, specifics are deferred to later sections insofar as it makes sense. For instance, the *requirements* should not prescribe design decisions; and the *design* should not prescribe representation details. However, there are no precise "boundaries" for what is specified where. It depends on the nature of the component being described. these sections, and they are all describing the same thing at differing abstraction levels.

A.1.1 Requirements

The Requirements section specifies *what* the component under design does in terms of its *externally observable* behavior. Requirement specifications may include such properties as:

- *Functionality,* the input-output relations.
- *Preconditions* or "assumptions," are conditions for correct use.

1. *Requirements.* What the system does.
2. *Design.* How the requirements are satisfied.
3. *Implementation.* Key representations, algorithms, etc.
4. *Coding.* Indexed presentation of source code.
5. *Testing.* Purpose and procedures.

A. *Developer Instructions.* `make` procedures, etc.

B. *User Instructions.* End-user procedures.

Figure A.1: Organization of System Documentation

- *Postconditions* or "guarantees," including not only output values but also such things as effects on call-by-reference argements, file space, etc.
- *Invariants*, such as safety and liveness conditions that are preserved by the executing component.
- *Constraints* on resources such as time, space, etc.
- *Validationvalidation.* The requirements may include a collection of specific observable (i.e. input/output) behaviors to which the delivered realization must comply. These may be thought of as being provided by the End User (or customer) for determining minimal satisfactory functionality.

Formal Requirements Specification. In applications where software is subject to certification, it is necessary to identify and subsequently track critical requirement properties through all levels of description. The *formal requirements statement* consists of a numbered sequence of individual *requirement specifications*, for instance

SPEC. R-1.1 *System documentation follows the outline given in Figure A.1.*

A.1.2 Design

The *Design* section explains *how* the requirements are satisfied. For this reason, it is usually organized according to component functionality (rather than architecture, as is the case in the *Implementation* section).

The design is presented abstractly, and routine representation details are deferred. It typically includes, for example, graph depictions of data structures or control-flow. Key algorithms may be presented in abbreviated form (e.g. pseudo-code) the main goal being to show mathematically how requirements are addressed.

Ideally, the design description gives just enough information for someone with sufficient programming expertise to develop an equivalent implementation on their own.

Design *verification*verification is a comparison of two levels of *description* (as opposed to *testing* the actual realization, see Section A.1.5). In critical applications, it is necessary to explain how each formal requirement property is satisfied by the design. The means of verification ranges from demonstration by model simulation to machine-checked mathematical proof, although in commone practice may be merely a careful, more-or-less rigorous English explanation.

SPEC. D-1.1 *A formal* design specification *statement explains how the system design satisfies the requirements specification with the same sequence number.*

A.1.3 Implementation

The *Implementation* section presents "key" representation details, including the overall architecture of the component. It should not include incidental coding details that can be readily understood by inspection of the source code. Instead, this section gives the "lay of the land," that a competent programmer would need to know before delving into the low-level coding details.

Specific design specification statements should be addressed. This is another level at which the term "*verification*" is applicable.

SPEC. I-1.1 *A formal* implementation specification *statement explains how the program or system realizes the properties of the requirements and design specifications having the same sequence number.*

A.1.4 Code

Source code is processed by a *code documentation* tool. *Doxygen*[1] that generates navigation indices, such as call graphs. These tools often have comment formatting provisions as well,

[1]`http://www.stack.nl/~dimitri/doxygen/index.html` is used in P545 unless it is superceded by an equivalent tool provided by the project development environment.

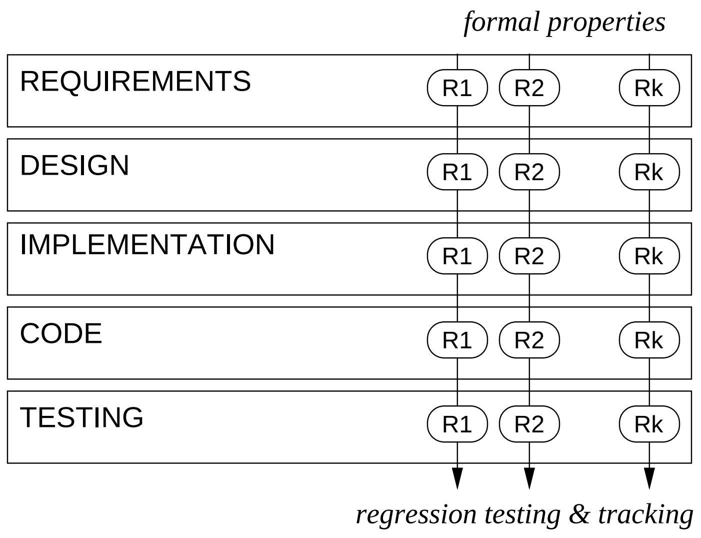


Figure A.2: Formal specification threads

allowing source-comments to be integrated logically in the higher-level system documentation.

However, the primary purpose of source-comments to providing *local* guidance in the immediate code context. Hence, these comments are generally insufficient for the higher purpose of the *Implementation* section.

> SPEC. C-1.1 *In applications subject to certification, there must be references in the source code to all formal specification statements, including their sequence numbers.*

A.1.5 Test

In contrast to *validation* (Sec. A.1.1) and *verification* (Sec. A.1.2), *testing*testing refers to execution of the component realization (hardware device, object code, etc.) against a selected sample of inputs and expected outputs.

The test of interest in this section do not include routine tracing for the purpose of programming, but rather, a cumulative suite of fixed tests whose purposes include final validation against end-user requirements, and *regression* testing against revisions, diagnoses of failures in the field, and so forth.

These tests should be automated for repeatability, and anomales in testing must be tracked and resolved prior to release of a component. This section includes *both* test specifications *and* documentation of repeatable test procedures.

> SPEC. T-1.1 *For each formal specification thread, there must be a battery of tests to validate that the specification is met. In critical applications, a reporting system is used to notify and track the resolution of the problem.*