Autonomous Mobile Robots

Lecture 07: Algorithmic Control

Lecture is based on material from Robotic Explorations: A Hands-on Introduction to Engineering, Fred Martin, Prentice Hall, 2001.

Outline

- Algorithmic vs. Reactive Control
- Approaches to Robot Control
- Robo-Pong Contest
  - Groucho (code, mechanics and strategy)
- Strengths and Weaknesses of Algorithmic Control
- Exercise: Groucho's Program
- Exit Conditions

Homework #7

- **Robot Contests:** Read Appendix F of Robotic Explorations (textbook)

Algorithmic vs. Reactive Control

- How to design an overall control strategy for operating a robot?
- Feedback control is appropriate for one piece of a robot's behavior, e.g., tracking a wall
- How should a robot coordinate a bunch of these lower-level feedback behaviors?
- We will examine two differing approaches to this problem:
  - **Algorithmic control** refers to a procedural series of steps or phases that a robot's program moves through in service of accomplishing some task
  - **Reactive control** refers to a collection of stimulus-response behaviors that dynamically trigger and retire as the robot moves through the physical environment

Approaches to Robot Control

- Robot control refers to the way in which the sensing and action of a robot are coordinated. There are infinitely many possible robot programs, but they all fall along a well-defined spectrum of control. Along this spectrum, there are four basic practical approaches being used today:
  - **Deliberative Control:** Think hard, then act.
  - **Reactive Control:** Don't think, (re)act.
  - **Hybrid Control:** Think and act independently, in parallel.
  - **Behavior-Based Control:** Think the way you act.
- References:
  - Maja J. Mataric, "Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior," *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2–3), 1997, 323–336.
  - *Behavior-Based Robotics* by R. Arkin, MIT Press, 1998.
  - *MIT Encyclopedia of Cognitive Sciences*, Robert A. Wilson and Frank C. Keil, eds., MIT Press, 1999.

Approaches to Robot Control

- No single approach is "the best" for all robots; each has its strengths and weaknesses. Control requires some unavoidable **trade-offs:**
  - Thinking is slow
  - Reaction must be fast
  - Thinking allows looking ahead (planning) to avoid bad actions
  - Thinking too long can be dangerous (e.g., falling off a cliff, being run over)
  - To think, the robot needs (a lot of) accurate information
  - The world keeps changing as the robot is thinking, so the slower it thinks, the more inaccurate its solutions
- As a result of these trade-offs, some robots don't think at all, while others mostly think and act very little. It all depends on the robot's task and its environment.
  - If the task and environment require the robot to move and react very quickly, there is usually no time for thinking, such as in automated fast-moving cars, or in soccer playing robots
  - If the environment does not change much and the robot has enough time, it can plan far ahead to find the best action, such as in playing chess, monitoring a warehouse at night, or assembling a complicated object

Approaches to Robot Control

Four different approaches to robot control that address these trade-offs:

- **"Think hard, then act."** In deliberative control, the robot takes all of the available sensory information, and all of the internally stored knowledge it has, and it creates a plan of action. The robot searches through potentially all possible plans until it finds one that will do the job. This requires the robot to look ahead and think in terms of: "If I do this next, and then this happens, then what if I do this next, then this happens, : : :" and so on. This can take a long time, which is why if the robot must react quickly, it may not be practical. However, if there is time, this allows the robot to act strategically.
- **"Don't think, react!"** Reactive control is a technique for tightly coupling sensory inputs and effector outputs, to allow the robot to respond very quickly to changing and unstructured environments. Think of reactive control as a "stimulus-response." This is a powerful control method; many animals are largely reactive. Limitations to this approach are that such robots, because they only look up actions for any sensory input, do not usually keep much information around, have no memory, no internal representations of the world around them, and no ability to learn over time.

Approaches to Robot Control

- **"Think and act independently, in parallel."** In hybrid control, the goal is to combine the best of both reactive and deliberative control. In it, one part of the robot's "brain" plans, while another deals with immediate reaction, such as avoiding obstacles and staying on the road. The challenge of this approach is bringing the two parts of the brain together, and allowing them to talk to each other, and resolve conflicts between the two. This requires a third part of the robot brain, and as a result these systems are often called "three-layer systems."
- **"Think the way you act."** Behavior-Based control is inspired from biology, and tries to model how animal brains may deal with hard problems of both thinking and acting. Behavior-based systems, like hybrid systems, also have different parts or layers, but unlike hybrid systems, they are not as different from each other. All of them are encoded as behaviors, processes that take inputs and send outputs to each other, quite quickly. So if a robot needs to plan ahead, it does so in a network of behaviors which talk to each other and send information around, rather than a single planner, as with hybrid systems. Behavior-based systems are an alternative to hybrid systems; these days, they are equally powerful and equally popular. However, they are not the same and thus are used for different types of robot applications.

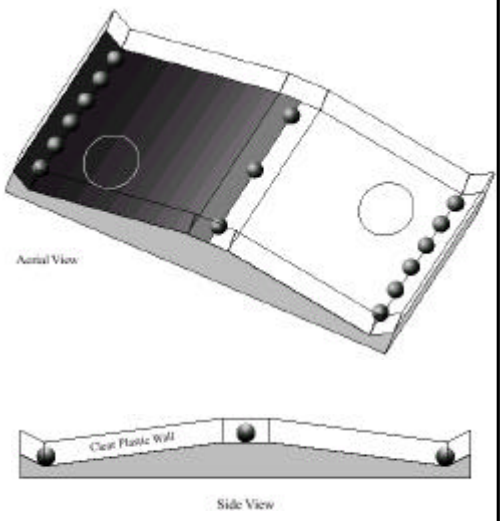
Approaches to Robot Control

- How time, or the lack of it, is handled distinguishes different approaches to robot control
  - **Deliberative systems** look into the future (plan) before deciding how to act.
  - **Reactive systems** respond to the immediate requirements of the environment, and do not look into the past or the future.
  - **Hybrid systems** respond to some of the urgent requirements, while taking time to think about some others. This requires waiting for the thinking to finish, or interrupting the reaction based on new information.
  - **Behavior-based systems** also think and act at the same time, but spread out the thinking over multiple distributed computation modules (behaviors). Thus they think the way they act, as quickly as possible.
- In many cases, just by observing a robot's behavior, it is impossible to tell what control approach it is using. This is similar to not knowing in what language some program is implemented, what somebody is thinking, or what exactly makes an animal do what it does. New robot control "architectures" for structuring robot control are being invented all the time, as novel applications for robotics are found. Sometimes, they become specialized robot programming languages. However novel they are, they can still fit into the fundamental control spectrum just described. Thus, once you can program a robot in all of the four approaches above, you know all possibilities at your disposal.

Robo-Pong Contest

Rules

- Run at MIT January 1991
- Contest involved 2 robots and 15 plastic practice golf balls
- Goal: have your robot transport balls from its side of table to opponent's in 60 minutes
- Robot with fewer balls on its side is the winner
- Table 4x6 feet, inclined surfaces, small plateau area in center
- Robots start in circles, balls placed as shown
- Table top painted two colors so that robots could use reflectance sensors to determine which side they were on
- Plan encouraged diversity in robot strategies



Robo-Pong Contest

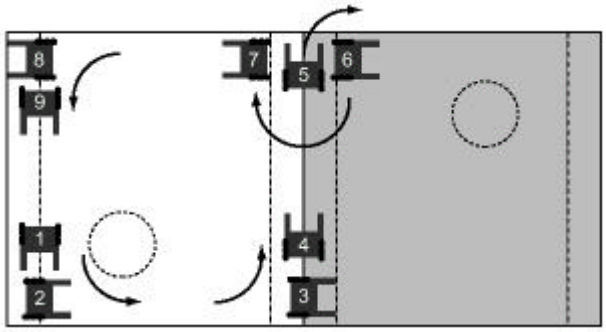
Designs

- Successful robots needed to be able uphill and downhill, maneuver in the trough area, and coordinate activities of collecting and delivering balls
- Overall, ball-collecting robots were the most popular design choice. This was not surprising in that they were also the mechanically least complex design
  - **Harvester Robots:** (18 teams) scooped the balls into some sort of open arms and then pushed them onto the opponent's side of the playing field
  - **Eater Robots:** (11 teams) similar to harvester approach with the exception that the eaters collected balls "inside their bodies" before driving over to the opponent's side. Students believed the eater robots to be a safer design than the harvesters, since balls couldn't be returned to the robot's own side by the opponent. On the other hand, the eaters were more mechanically complex and hence more prone to failure.
- **Shooter robots:** (4 teams) catapulted balls onto the opponent's side of the table (if a ball went over the playing field wall on the opponent's territory, the ball would be permanently scored against the opponent)
  - Building a shooter design required a fair bit of mechanical ingenuity, both in the shooting mechanism and a device to load balls into the shooter. The four shooters that were finally fielded were sophisticated and pleasing to watch, but ultimately lost against aggressor designs, which could trap them easily, and effective collector designs, which delivered balls back after they had been shot across the table.

Robo-Pong Contest

Groucho

Strategy pattern of **Groucho,** an algorithmic ball-harvester



From start circle, the robot drives to position 1. It drives forward until striking the wall, and then rotates to position 2. From there, it climbs up the ramp until crossing the black/white divider (position 3)—at which point any balls it has collected are dumped onto the opponent's side. Then it rotates to position 4, and drives across the center plateau until striking the opposing wall (position 5). Next it turns to face down the opponent's slope (position 6), dumping any balls it may have collected along the center plateau. Then it performs a 180 degree turn, lining itself up to drive down its own slope (position 7), and it drives downward until striking the back wall (position 8). Finally it rotates to position 9, and scoops up balls in the trough, continuing the loop at position 1.

Robo-Pong Contest

Groucho's Code

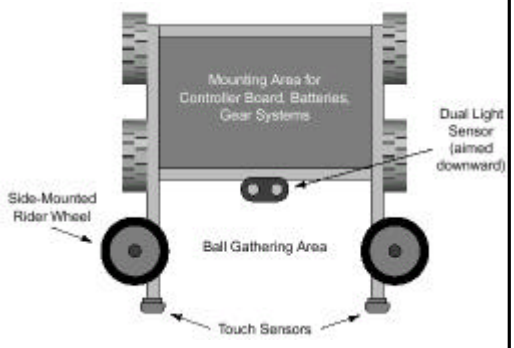
- *Groucho* perfectly exemplifies the algorithmic approach: a series of actions that are performed in order to accomplish the desired task—as a mathematical algorithm prescribes a series of operations to perform on data
- Algorithmic approach is based on a linear series of actions, which are performed in a repetitive loop
- Sensing may be used in the service of these actions, but it does not change the order in which they will be performed.
- Robots rarely go precisely straight or turn a precise number of degrees so it's safe to assume that some kind of feedback based on the surrounding environment will be necessary

```
void groucho() {
while (1) { /* loop indefinitely */
/* for simplicity, assume robot starts at position 1 */
    forward();
    waituntil_hit_wall();
    rotate_left_ninety(); /* now at position 2 */
    forward();
    waituntil_see_black(); /* position 3 */
    rotate_left_ninety(); /* position 4 */
    forward();
    waituntil_hit_wall(); /* position 5 */
    rotate_left_ninety(); /* position 6 */
    rotate_onehundred_eighty(); /* position 7 */
    forward();
    waituntil_hit_wall(); /* position 8 */
    rotate_left_ninety(); /* position 9 */
}
}
```


Robo-Pong Contest

Groucho's Mechanics

- Basic turtle drive system with pair of driven wheels one each side.
- Pair of free-spinning rider wheels mounted parallel to the floor, jutting out from the front "arms," for driving along a wall with no sensing or feedback required
- Two kinds of sensors: a touch sensor at the end each of of its "arms," and a pair of light sensors facing downward located near its geometric center. Touch sensors used for cornering, and light sensors determined when it had crossed onto the other side of the table and tracked the light/dark boundary across the center plateau.

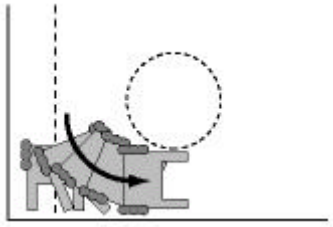


Schematic of Groucho Robot

Robo-Pong Contest

Groucho's Strategy

- *Groucho* did not make a ninety degree turn from **position 1 to position 2**, strictly speaking.
- *Groucho* instead was designed to take corners with a repetitive series of little turns and collisions back into the wall.
- Each time *Groucho* hit the wall, it would back up, turn a little, and then start driving forward again.
- After about four or five of these iterations, *Groucho* would have turned the required amount, and it would be able to continue along its path
- This turning method was fairly reliable, because even if the wheels slipped a little on one turn, *Groucho* kept turning until the touch sensor no longer struck the wall—in which case, it would have completed its turn.
- Note that this method works for a wide range of corners—ones less than and greater than the right angles on the *Robo-Pong* playing field.



Groucho's Cornering Algorithm

Robo-Pong Contest

Groucho's Strategy

- In performing the ninety degree turn from **position 3 to position 4**, *Groucho* did things differently—it simply executed a single timed turn movement. The students who built *Groucho* attempted to use feedback to ensure that the robot would turn the proper amount, but found it difficult to get reliable results (lacking the wall as a reference point, as they had done with the inside-corner turns). Instead, a timed turn (with the duration determined experimentally) performed satisfactorily to them.
- The students did not expect *Groucho* to be able to **cross the center plateau** without feedback sensing, however. This is where the dual light sensor, aimed downward at the playing surface, was used. As *Groucho* drove across the table, it made sure that one sensor was kept on the dark side of the table and the other on the light side. The sensor and a feedback program thus compensated for any variances in the mechanical performance of *Groucho*'s geartrain, and *Groucho* would reliably reach the opposite side of the plateau, triggering a touch sensor.
- **Summary:** algorithmic strategy method is relatively simple and can be effective when a straight-forward algorithm can be devised. The *Groucho* robot for *Robo-Pong* is a good example of such an approach.

Strengths and Weaknesses of Algorithmic Control

- **Strengths** of the algorithmic approach: simplicity, directness, and predictability when things go according to plan
- **Weaknesses:** inability to detect or correct for problems or unexpected circumstances, and the chained-dependencies required for proper functioning
  - If any one step fails, the whole solution typically fails. Each link-step of an algorithmic solution has a chance of failing, and this chance *multiplies* throughout the set of steps, e.g., suppose each step has a 90% chance of functioning properly on any given trial, and there are six such steps in the solution. Then the likelihood of overall program working is the likelihood that each steps functions properly: ~53% chance
- How to bolster algorithmic process: have separable steps along the way performed by feedback loops
  - Groucho's handling of inside corners: rather than just turning by a timed amount, Groucho used a series of little turns and bumps to negotiate the corners. This method implicitly acknowledged that Groucho would not reliably hit the wall at a perfect perpendicular angle, from which a timed turn could be based. The feedback used at the corners thereby compensated for variances in the playing field, the performance of the robot, and real-world unpredictability.
  - On the other hand, Groucho did use a timed turn at the crest of the hill, turning in preparation for crossing the plateau. Perhaps because of the rolling rider wheels, or because the right-angle turn was immediately followed by a feedback program that tracked the light/dark edge, Groucho's builders determined that an open-loop, timed turn was appropriate in that situation.
- By **embedding feedback controls within the algorithmic framework**, the reliability of the algorithmic approach is greatly improved.

Exercise: Groucho's Program

- Construct a program to control Groucho through its algorithmic strategy (use Groucho mock-up or HandyBug)
- Write these simple movement and sensor functions before getting started:
  - **Movement functions,** assuming Groucho is a standard "turtle" style robot: forward(), backward(), spin left() and spin right() (rotate in place), and veer left() and veer right() (turn while moving forward by operating one motor and leaving the other motor off).
  - **Sensor functions:** left touch() and right touch() for the two touch sensors, which return 0 if not pressed and 1 if pressed, and left eye() and right eye() for the dual light sensors, which return 0 if positioned above the dark surface and 1 if above the light surface.

```
/* grchmain.c */
void groucho()
{
while (1) {
    inside_corner(); /* position 1 to 2 */
    drive_to_top(); /* 2 to 3 */
    align_with_edge(); /* 3 to 4 */
    follow_edge_to_wall(); /* 4 to 5 */
    dump_ball_shuffle(); /* 5 to 6 to 7 */
    drive_to_bottom(); /* 7 to 8 */
    inside_corner(); /* 8 to 9 */
}
}
```

Exercise: Groucho's Program

1. Write a subroutine to negotiate the inside corner, represented by the transition from step 1 to step 2. Name the subroutine inside corner(); upon entry, it should assume that the *Groucho* has just struck the wall with either or both of its touch sensors. The subroutine should exit after *Groucho* has turned the corner and has been following along the wall on its right side for at least two seconds
2. Write drive to top(), a subroutine that takes control after inside corner() (position 2) and drives *Groucho* until it senses the light/dark boundary (position 3). At this point, the routine should stop *Groucho* from moving and return.
3. Write align with edge(), which should drive *Groucho* forward for a second or two (dumping any collected balls over to the opponent's side), drive back, and then rotate to parallel to the light/dark edge (position 4). Use the light sensor to determine when the robot has rotated back to cross over the light/dark edge, and open-loop timing for the forward and backward motions. Return when the robot has crossed over the light/dark edge and is ready to follow the edge across the table.
4. Write follow edge to wall(), which uses the dual light sensor to follow the light/dark edge across the playing field. The routine should attempt to keep one light sensor on the dark side and the other on the light side, thereby tracking the edge and traversing the table. Terminate when either of the touch sensors is pressed, signally that the robot has reached the far wall of the playing field (position 5).

Exercise: Groucho's Program

5. Write dump ball shuffle(), which takes *Groucho* from having contacted the far wall head on (position 5), to facing down the opponent's side (position 6), to turning around 180 degrees heading down its own side with the wall to its right (position 7). This function is difficult to accomplish using feedback; this exercise should best be done with an actual robot. Alternatively, this function may be replaced with another call to inside corner.
6. Write drive to bottom(), which drives *Groucho* straight ahead and terminates when a touch sensor is pressed (position 8).
7. The final cornering action (positions 8 to 9) may be done with another call to inside corner().
8. At this point it should be possible to test the groucho() program loop presented at the beginning of this exercise. If you are working with an actual robot, attempt the unified program and describe the results.

Exit Conditions

- **Problem with simple algorithmic approach:** there is no provision for detecting, no less correcting for, problem situations
  - Consider *Groucho*'s program: most of the time, it is waiting for a touch sensor to trigger the next phase of action. If something were to impede its travel, without striking a touch sensor, *Groucho* would just sit there, unable to take corrective action. Many a robot has failed in this way in contest situations.
  - There are a variety of other ways *Groucho* could fail. Suppose it is crossing the top plateau, weaving its way back and forth across the light/dark boundary, and the opponent robot gets in the way, and *does* trigger a touch sensor. Then *Groucho* would begin its behavior that is normally activated when it reaches the opposing wall. Depending on the interaction between *Groucho* and the opponent robot, this might or might not be a sensible thing to do.
- **Solution:** techniques for error detection, and discuss recovery within an algorithmic framework.
  - Instead, suppose it were possible for *Groucho* to "know" that it had struck the opponent, and not the opposite wall. Then it might be possible for *Groucho* to take an action that would have a better likelihood of a desirable result, rather than just blindly proceeding with the algorithmic plan.

Exit Conditions

Timeouts

- Consider: *Groucho* going from **position 4 to 5**: traverses light/dark edge across the playing field. In the earlier exercise, we named this subroutine follow _edge_to_wall(); the core loop of this function is feedback based on the dual light sensors, with an exit check from the touch sensors
- **Problem:** only way this loop can exit: if one of the touch sensors is pressed; if *Groucho* were to hit something without the touch sensors being pressed, the loop simply would never end
- **Solution:** allow the subroutine to **time out.** After a predetermined period of time has elapsed, the subroutine exits even if a touch sensor was not pressed.

```
/* "eye" sensors return 1 if above light,
 0 if above dark; try to keep left on light,
 right on dark */
while (1) {
/* if left eye sees black, turn left */
    if (left_eye() == 0) veer_left();
/* if right eye sees white, turn right */
    else if (right_eye() == 1) veer_right();
/* otherwise, go straight */
    else forward();
/* check for touch sensors */
    if (left_touch() || right_touch()) break;
}
```

- Built-in IC functions for measuring elapsed time:
  - **seconds()** function reports the number of seconds since the last system reset, as a floating point number (avoid floating point operations on Handy Board - they are too slow)
  - **mseconds()** function reports the same quantity as the number of *milliseconds* since reset, as a long integer (use mseconds() for timing functions)

Exit Conditions

Timeouts

- Keep track of how much time has elapsed since a subroutine began execution, and take action at some point in time.
- A long integer variable, **timeout**, is declared and initialized with the value of the current time plus four seconds (4,000 milliseconds, or 4000L)
- Inside the body of the feedback loop, the current time is compared with the timeout point; if the current time is later than the timeout value, the loop exits
- The last statement performs the elapsed time check, and breaks the loop if too much time has elapsed. (It would also be possible to put the timeout check as the condition of the while loop.)

```
/* declare and initialize timeout variable */
long timeout= mseconds() + 4000L;
while (1) {
    if (left_eye() == 0) veer_left();
    else if (right_eye() == 1) veer_right();
    else forward();
    if (left_touch() || right_touch()) break;
/* check for timeout */
    if (mseconds() > timeout) break;
}
```


Exit Conditions

Timeouts

- In most cases, the higher level control program should take a special action when a subroutine exits because of a timeout rather than the normal conclusion of its duties.
- Function has a return value that indicates whether the routine terminated normally (with a touch sensor press) or abnormally (because of a timeout)
- Robot's main program would need to be modified to take action based upon the success or lack thereof of the program's subroutines.
- At least, though, the timeout technique allows the master program to have the opportunity to take corrective action.

```
/* define exit codes */
int NORMAL= 0;
int TIMEOUT= 1;
int follow_edge_to_wall() {
long timeout= mseconds() + 4000L;
while (1) {
    if (left_eye() == 0) veer_left();
    else if (right_eye() == 1) veer_right();
    else forward();
    if (left_touch() || right_touch()) return NORMAL;
    if (mseconds() > timeout) return TIMEOUT;
}
}
```

Exit Conditions

Premature Exits

- **Problem:** routine finishes in too *little* time
- Suppose *Groucho* is happily traversing that center median and the opposing robot is in the way. If circumstances are fortunate, one of *Groucho*'s touch sensors will be triggered.
- *Groucho* realizes that something unusual has happened, because from past empirical observation there is no way that *Groucho* could already have reached the far wall.
- The "too-long" timeout has been made into a program constant (TOO LONG), along with a new parameter, the "too-short" timeout (TOO SHORT).
- When the touch sensors are hit, the elapsed time is checked, and if it is less than the TOO SHORT amount returns with the EARLY error result rather than the NORMAL exit result

```
/* define exit codes */
int NORMAL= 0;
int TIMEOUT= 1;
int EARLY= 2;
/* sample timing parameters */
long TOO_LONG= 4000L;
long TOO_SHORT= 1500L;
int follow_edge_to_wall() {
    long start= mseconds();
    long timeout= start + TOO_LONG;
    while (1) {
     if (left_eye() == 0) veer_left();
     else if (right_eye() == 1) veer_right();
     else forward();
     if (left_touch() || right_touch())
     if (mseconds() < (start + TOO_SHORT))
     return EARLY;
     else return NORMAL;
     if (mseconds() > timeout) return TIMEOUT;
}
}
```


Exit Conditions

Premature Exits

- Final piece of the timeout puzzle: While a robot is performing a feedback process like following the light/dark edge, it is typically shuttling back and forth between the various modes.
- That is to say, when *Groucho* is traversing the edge, it is continuously correcting its movements, one moment veering left, then going straight, then going right, then another way.
- In the **edge-following algorithm**, *Groucho* may stay in the "go straight" mode for a fair while, but it shouldn't stay in the "veer left" or "veer right" modes for very long.
- These transitions between the **different modes of the feedback loop** can be monitored along with the overall performance.
- Three new timeout parameters are necessary:
  - VL_TIME (veer left timeout), VR_TIME (veer right time), and GS_TIME (go straight timeout). Parameters represent longest time that *Groucho* may spend *continuously* in any given state.
- Two new state variables are necessary:
  - last_mode, to keep track of the loop's state the last time through, and last_time, to keep track of the time the last state began.
- Three new return codes, VL_STUCK, VR_STUCK,and GS_STUCK, are re-used as codes to represent the states.

Exit Conditions

```
/* define exit codes */
int NORMAL= 0;
int TIMEOUT= 1;
int EARLY= 2;
int VL_STUCK= 3;
int VR_STUCK= 4;
int GS_STUCK= 5;

/* sample timing parameters */
long TOO_LONG= 4000L;
long TOO_SHORT= 1500L;
long VL_TIME= 2000L;
long VR_TIME= 2000L;
long GS_TIME= 3000L;

int follow_edge_to_wall() {
long start= mseconds();
long timeout= start + TOO_LONG;
int last_mode= 0;
long last_time= 0;
```

```
while (1) {
    if (left_eye() == 0) {
    veer_left();
    if (last_mode == VL_STUCK)
      if ((mseconds() - last_time) > VL_TIME)
        return VL_STUCK;
      else {
        last_mode= VL_STUCK;
        last_time= mseconds();
    }
}
else if (right_eye() == 1) {
  veer_right();
  if (last_mode == VR_STUCK)
    if ((mseconds() - last_time) > VR_TIME)
      return VR_STUCK;
    else {
      last_mode= VR_STUCK;
      last_time= mseconds();
    }
}
```


Exit Conditions

- Notice sample values for the timeout durations VL TIME and VR TIME are different than the value of GS TIME. This is to point toward one potential problem with this approach:
  - **what if Groucho tracks the line too well?**
- Possible that it might stay in the "go straight" state for a long uninterrupted period.
- If a given robot seems to perform very well when it's directly centered and going straight, one would not want the going-straight timeout to interrupt it from the fine job it's doing.
- These sort of cases must be determined experimentally; if necessary, the GS TIME value can be increased to as large as the EARLY timeout value to ensure that the robot is allowed to go straight without being disturbed by the monitoring software.

```
else {
forward();
if (last_mode == GS_STUCK)
  if ((mseconds() - last_time) > GS_TIME)
    return GS_STUCK;
  else {
    last_mode= GS_STUCK;
    last_time= mseconds();
  }
}
if (left_touch() || right_touch())
  if (mseconds() < (start + TOO_SHORT))
    return EARLY;
  else return NORMAL;
    if (mseconds() > timeout) return TIMEOUT;
  }
}
```

Exit Conditions

Taking Action

- **Question:** how to take advantage of knowledge about the performance of feedback loops while they are running: it is very difficult for most algorithmic controls to know what action to take upon learning that a problem has occurred.
  - *Groucho* crossing the center median: suppose it is running the latest version of the follow_edge_to_wall() function, and the function exits with the VL _STUCK error code.
  - What has occurred? Perhaps *Groucho* has run into the opponent robot part of the way across the playing field, or perhaps *Groucho* has mistracked the median edge and driven itself into a playing field wall, or perhaps something else has gone wrong—there is simply no way to exactly determine the situation.
- It is difficult for the algorithmic control program to take the "appropriate" course of action based on the error result.
- One possibility is that an error condition from a feedback routine could prompt a re-examination of all other sensors to try to make sense of the situation.
  - If *Groucho*'s touch sensors went off only part way across the playing field, in the context of a contest situation, it could be because the opponent robot got in the way. If *Groucho* had any other sensors that could be used to detect the opponent robot, now would be the time to check them.
- Even if it is difficult to design an appropriate reaction to each various situation that might be detected by the timeout methods, it is often possible to figure out a single recovery behavior that would suffice for many circumstances.
  - In *Groucho*'s case, heading downhill until hitting the bottom wall and then proceeding with the cornering routine should allow *Groucho* to recover from a variety of problems.

Exit Conditions

Exercise: Groucho with Timeouts

1. *Timeout detection.* For each of the following *Groucho* subroutines, decide upon suitable methods for determining timeout conditions, and re-write the functions to implement them. Which one or which combination of the techniques—simple timeout, premature timeout, or feedback loop monitoring—are most appropriate for each subroutine?
   - inside_corner()
   - drive_to_top()
   - align_with_edge()
   - drive_to_bottom()

2. *Corrective action.* For each of previous four subroutines, postulate reasons that the routine could fail with a timeout, how your routines would detect the failures, and possible corrective actions.

3. *Putting it together.* Re-write the main *Groucho* program, which previously was just a loop running the subroutines in order, to choose the appropriate corrective action based on the error codes reported by the modified subroutines.