Potential Fields Lab

Chris Monson

May 31, 2005

The purpose of this lab is to give you experience with simple reactive agents using potential fields as a form of “intelligence”. You will do the following:

- Code up attractive fields
- Code up repulsive fields
- Code up tangential fields
- Use all three fields to help a BZFlag agent capture an enemy flag

No internal states will be allowed for your agent. All actions should be based purely on the current state of the world.

1 What To Turn In

To pass off this lab, you will turn in the following electronically, in an email to cs470ta@cs.byu.edu:

- All of your agent code
- A declaration of time spent by each lab partner
- A writeup of your experience with the lab (may be submitted electronically or on paper)

You will need to demonstrate to the TA that your code works.

The writeup should describe in detail what all of the fields do and why you coded them in the manner that you did. Don’t just say, “I used an attractive field.” There are lots of types of attractive fields, so go into detail. Where appropriate, plot graphs of what the field looks like and put those into the paper.

Though the paper need not be very long, do a nice job and make it look professional. One of the best ways to get a professional looking paper nearly for free is to use \L\TeX, so that’s what I recommend. PDF files are always appreciated, Postscript is okay, and Word makes us grumpy but we’ll read it anyway.

2 Implementation Requirements

You will be given a distribution of BZFlag (http://www.bzflag.org/) code. BZFlag is a computer game in which tanks play capture the flag and shoot each other if they so desire. You will be writing code to control a team of these tanks in a modified version of BZFlag (so make sure that you get it from the class website).

BZFlag has a nice client/server architecture, which makes it easy to do tournament play and to have agents spread out around the world. We won’t take advantage of that latter characteristic, but we will take advantage of the former. In a nutshell, a server is started on a machine. The server has control of the game options and the world map, and it regulates all of the play. Clients connect to the server via TCP/IP. Observers do, too. To be an observer, the first character of your call sign should be ‘@’.

To simplify the coding experience and to limit your agents’ knowledge and power, we have introduced a layer between your agent code and the communication code. You will code to this layer.
2.1 The World

CS 470 is not a physics class. BZFlag is not virtual reality. Don’t get worried if things aren’t quite like real life. Just have fun (and blow up lots of tanks). However, it does try to make things somewhat reasonable.

You are a tank—a metric tank. You are 6.0 meters long (TankLength), 2.8 meters wide (TankWidth), and 2.05 meters tall (TankHeight). You live in a world which is 800 meters by 800 meters square (WorldSize) and which is surrounded by 6.15 meter high walls (WallHeight). Your maximum speed is 25.0 meters per second (TankSpeed), with maximum acceleration of 5.0 meters per second squared (TankAccel), and maximum rotational speed of 45 degrees per second (TankAngVel). Your shots go 100.0 meters per second (ShotSpeed) with a range of 350.0 meters (ShotRange). These constants are all listed in include/global.h and are automatically included in your code. Don’t use magic constants!

You will program agents to make decisions in this world. Your only direct decisions are how fast to accelerate, how much to turn, and whether or not to fire a shot. Your agent will constantly be given new information and prompted to make a decision. If your code makes its decisions reasonably quickly, it can be called over a hundred times each second.

2.2 The Distribution

All of these instructions apply to writing the code under Linux on the Open Lab machines. If you are interested in working from home, it is strongly encouraged that you do so using a relatively recent Linux distribution with 3D acceleration. It also works on Mac OS X. If you attempt to make this work on Windows, you are welcome to try, but you should know that nobody has made it work reliably on that platform for this class. Deadlines will not be extended if the reason for the request is that time was spent trying to make it work on Windows.

The first thing you should do is download, unpack, configure, and compile the distribution. You probably should not install it:

```
    tar xzf bzflag470-1.7g2-4.tar.gz
    cd bzflag470-1.7g2-4
    ./configure
    make
```

In the distribution, several directories are of potential interest:

```
src/bzfs : the server directory
src/bzflag : the graphical client directory (with patched observer code to enhance the observer interface)
src/470bot : the 470 bots including sample agents and your code
```

The distribution has been successfully compiled on Linux (RedHat 8.0, 9.0, Fedora Core 2, Gentoo) as well as on Mac OS X 10.2 and 10.3.

3 470bot Agents

The 470bot directory includes two sample agents: DumbAgent.cxx and Lab1TAAgent.cxx. The makefiles depend on the existence of files called “redteam.cxx” and “greenteam.cxx”, but they should be symbolic links, allowing you to swap different agents in and out without renaming or mixing up files.

3.1 Really Dumb Agents

These agents simply move forward and then turn left when they run into something. They fire every 0.5 seconds or so. These are to illustrate some simple state. It’s best to run these in an empty world with about 5 agents, otherwise mayhem will ensue (it’s kind of fun to watch, actually).

To use the really dumb agents:
The “redteam” and “greenteam” executables will now contain a Really Dumb Agent™. This means that you can watch two teams fight which are both operating from the same source code.

3.2 Potential Field Defenders

These are the defenders that we will use for the Potential Fields Lab.

To use the potential field defenders:

```plaintext
  cd src/470bot
  ln -fs DumbAgent.cxx redteam.cxx
  make clean
  make
```

3.3 The Empty Agent

The file “EmptyAgent.cxx” is just the bare 470bot framework. Copy this file and fill it in to create your own exciting and intelligent agents. Read more about how to do this later on in the lab description.

4 Starting Things Up

There are three steps:

1. Start the bzfs server
2. Start the bot(s)
3. Connect with an observer

Open up four windows, one for each program that you will be running.

First, you need to start the server (bzfs). Use the “ss.sh” script to run it with the common options that we will be using:

```plaintext
  cd src/bzfs
  ./ss.sh -world empty_world
```

Note that the CS Department will lock your account if you forget to kill all instances of bzfs before you leave a computer (run “killall bzfs” before logging out to make sure you don’t have any stray BZFlag servers lying around. There is more information available about starting the server in Section 4.2.

Next, start up the 470 bots and have them connect to the server. As you’re first becoming familiar with the setup, you may use the same TA bot for both red and green teams. Assuming that you are running the bzflag server on the same machine as the clients will be running on, you can use the “redteam.sh” and “greenteam.sh” scripts to quickly start up two teams of 5 agents:

```plaintext
  cd src/470bot
  ./redteam.sh
  ...and...
  cd src/470bot
  ./greenteam.sh
```
There is more information about starting clients in Section 4.3. Currently, they will spew a lot of debugging information. Sorry. We’ll fix that.

Once the clients have successfully connected to the server, they will begin to play. You may connect as an observer to watch them. Note that you may also connect as a human player to play against them!

Note that you can use any pre-existing bzflag client to connect to the server for this lab – you do not have to use our patched observer code (but it is much nicer). There is more information about starting observers in Section 4.4.

### 4.1 This Lab

For the Potential Fields Lab, we will run the server as follows:

```
./ss.sh -world empty_world
```

(note the `-world empty_world` parameter – if you omit it, bzflag will serve a random world). Note that this is equivalent to

```
We will run both your agent and the ta agent with these changes in options from redteam.sh and greenteam.sh:
```

```
-solo 1 \n-collision_check on \n-collision_same_team_action gameover \n-collision_diff_team_action gameover \n-flag_action gameover \n-shooting off
```

### 4.2 Starting the Server

#### 4.2.1 Typical options

```
cd src/bzfs
./bzfs -c -mo 2 -mp 30 -ms 5 -tk -r
```

If you want to start the server with the usual options, you can use the provided “ss.sh” script instead. The above options will start the server with the default world. If you want to use an empty world, try this:

```
./bzfs -c -mo 2 -mp 30 -ms 5 -tk -r -world empty_world
```

which is equivalent to this:

```
./ss.sh -world empty_world
```

#### 4.2.2 Server options

- **-c** : capture the flag style
- **-g** : one game only
- **-mo n** : max observers
- **-mp n** : max players
-ms n : max shots
-time float : time limit
-d n : debug level
-tk : team killer does not die
-r : allow rogues (tanks without a team)
-world worldfile : use “worldfile” as the world

There is also another nice reference (http://www.shellshock.dutchrai.com/references-bzfscmd.htm) for command line options if you want to know more.

4.3 Starting Clients

In order to start a client, there are several required options: a callsign, a team color, and a server to connect to. It is also sometimes necessary to tell bzflag where to find your data directory.

4.3.1 Typical Options

These are the typical options that we will be using to start the bzflag bots:

```
cd src/470bot
./redteam \\n  -solo NUMROBOTS \\n  -callsign COLORteam \\n  -team COLOR -directory \\n  ../.../data localhost
```

where “NUMROBOTS” will be replaced with the number of robots appropriate for the lab, and “COLOR” will be similarly replaced.

If you want to start a client with the usual options, you can use the provided “greenteam.sh” or “redteam.sh” scripts.

4.3.2 Client Options

See this reference (http://www.shellshock.dutchrai.com/references-bzflagcmd.htm) for a good list of client command-line options and their explanations. Note that many of the options documented here do not apply to the headless bots, because all graphics support has been removed.

We have added the following custom 470 options:

- -solo <num-robots>
- -can_capture_own_flag <on|off>
- -resurrect <on|off>
- -shooting <on|off>
- -collision_check <on|off>
- -collision_diff_team_action <msg|freeze|stuck|explode|gameover>
- -collision_same_team_action <msg|freeze|stuck|explode|gameover>
- -flag_check <on|off>
- -flag_action <normal|gameover>
- -random_seed <{some number}>

The random_seed option can be particularly useful for debugging. Whenever the random seed is the same, bots will be placed in the world in the same place. To reproduce a problem situation, run the agents again with the same random seed, which the redteam/greenteam programs print out as they start up.
4.4 Starting an Observer

In order to start an observer, the first character of your callsign must be the ‘@’ character. We recommend that you connect as a “rogue” player to avoid activating additional flags!

4.4.1 Typical Options

```
  cd src/bzflag
  ./bzflag -callsign @observer -directory ../.. / data -team rogue localhost
```

4.4.2 Observer Options

There are several options that you can use to control the way that the bzflag scene is rendered on the observer. For example, you can render in a window, render fullscreen, render at lower quality levels, turn on lighting for bullets, etc.

You can access configuration menus by pressing “ESC”. When you quit the client, all of your settings will be saved in a configuration file (typically `~/bzflag`).

See this reference (http://www.shellshock.dutchrai.com/references-bzflagconfig.htm) for an explanation of the config file. Additionally, see this reference (http://www.shellshock.dutchrai.com/references-bzflagcmd.htm) for a good list of client command-line options and their explanations.

4.4.3 Observer Keys

F11 : jump to a useful view: birdseye, mouse tracking off, black render
F10 : toggle mouse tracking
F5 : reset to the initial view
F4 : bird’s eye view
F3 : toggle HUD
F2 : toggle black render
F1 : set some nice defaults
up arrow, down arrow, left arrow, right arrow : translate forward, backward, left or right
pageup : move up vertically
shift-up : move up vertically
pagedown : move down vertically
shift-down : move down vertically
left mouse : zoom in
right mouse : zoom out
middle mouse : reset zoom

5 Implementing Your Agent

A simple abstraction layer has been placed between the code that you will write and the actual bzflag client. This is both to reduce your learning curve and to prevent the possibilities of cheating.

You will implement one file: `src/470bot/TheNameOfYourReallyCoolAgent.cxx`

As with the other agents, make a symlink to compile your code:
The idea behind the interface is simple: the server will call a special function (that you implement) once for each of the robots that you control. You will instruct the server what to do for that robot by filling out an instruction structure.

### 5.0.4 Functions You Must Write

- `void world_init( world_t *w, team_t team );`
- `void robot_kill( player_t *r, int your_robot_num );`
- `void robot_resurrect( player_t *r, int your_robot_num );`
- `void robot_pre_update( void );`
- `void robot_update( player_t *r, int your_robot_num, inst_t *instructions );`
- `void robot_post_update( void );`

Any (or all!) of the functions may be empty, but they must be defined. All of the relevant functions and data structures are prototyped for you in `470bot.h`.

### 5.0.5 Order of Calls

The function `world_init` will be called once on startup. The `world_t` structure has lots of helpful information about the world you will be in (described below).

During game play, the order of calls is as follows:

```c
robot_pre_update();
robot_update( first_robot ... );
robot_update( second_robot ... );
robot_update( third_robot ... );
/* ... */
robot_update( nth_robot ... );
robot_post_update();
```

This code is executed in a loop as often as is possible.

### 5.0.6 Passing Back Instructions

An empty instruction structure is passed to `robot_update` for each robot. You are expected to fill out the structure. The instructions are very simple:

```c
typedef struct inst_t {
    float accel, angvel;
    int fire_shot;
} inst_t;
```

The `accel` field specifies your acceleration in meters per second squared. If you don’t set acceleration, you will continue merrily along your way at the current speed until you get shot and/or run into a wall. If you want to accelerate to a noticable degree, you must set the `accel` field each turn.
The angvel field sets the angular velocity in radians per second. Zero radians is due east, $\frac{\pi}{2}$ is north, and so on. This means that right is negative and left is positive. Deal. Note that $\pi$ is defined to be pi by math.h, so you don’t have to set it on your own.

The fire shot field specifies whether or not to fire a shot this turn. Do to rate limiting, it is not “intelligent” to set fire_shot to 1 every turn.

angvel is the amount of turn (a number between $\pm TankAngVel$, i.e. $\pm \frac{\pi}{4}$)
accel is the amount of acceleration (a number between $\pm TankAccel$, i.e. $\pm 5.0$)
fire_shot is whether or not you want this tank to fire a shot

5.0.7 Data structures

The world_t data structure contains information about the world.

```c
typedef struct world_t {
    float world_size;
    float reload_time;
    float time;
    float dt;

    int max_ever_players;
    int players_you_control;
    int current_num_players;
    player_t *players;

    flag_t flags[MAX_TEAMS];
    base_t bases[MAX_TEAMS];

    int num_obstacles;
    obstacle_t *obstacles;

    int shots_per_player;
} world_t;
```

world_size is the size of the world. The coordinates of the west side of the world are $-0.5*world_size$; the east side is $0.5*world_size$. Same goes for the north and south sides.

reload_time is the amount of time before a reload can occur. When determining whether or not a shot can fire, the time since the last shot is examined; if it is more than reload_time / shots_per_player, then a shot is fired.

time is the absolute amount of time that has elapsed since game start.

dt is the amount of time since the last tick (the last time your agents’ functions were called).

max_ever_players is the number of players that will ever be allowed into the game. It will not change during play.

players_you_control is the number of robots you are controlling. It will not change during play.

current_num_players is the current number of players, humans, and robots that are connected. It may change during play.

players is a linked list of player_t structures. One of these structures will be defined for each player. This linked list may change during play.
flags is an array of flag_t. It holds information about each of the four flags. The length of this array will not change.

num_obstacles is the number of obstacles in the world. This will not change during play.

obstacles is an array of obstacle_t. Each element describes one obstacle in the world. The obstacles are constant throughout the game, so this array will not change.

shots_per_player indicates how many shots each player receives.

The player_t data structure contains information about each player. It is implemented as a simple linked list, which may shrink or grow as players are added (and removed) from the game.

```c
typedef struct player_t {
  const char *callsign;
  team_t team;
  float old_position[3];
  float cur_position[3];
  float velocity[3];
  float angle, angvel;
  status_t status;

  int has_flag;

  int shots_used;
  int shots_avail;
  shot_t *shots;

  /* these are private. Read, but do not frob. */
  struct player_t *next;
} player_t;
```

callsign is the player’s callsign (null terminated string).

team is the player’s team (TEAM_RED through TEAM_PURPLE).

old_position, cur_position and velocity should be self-explanatory. The 3 values are x, y, and z coordinates.

angle is the direction the player is facing (in radians). This ranges from zero to 2π.

angvel is the angular velocity (radians per second).

status is a bitwise-OR of several status flags (see the STATUS_* defines).

has_flag is −1 if this tank has no flag. Otherwise, it is the number of the flag’s team. Um. So, if you have the red team’s flag, this number will be TEAM_RED. And so forth. Yup.

shots_used is the number of shots this player has used.

shots_avail is the number of shots left.

The shot type is shown below. There is one of these for every shot in the game, whether or not the bullet is currently active.

```c
typedef struct shot_t {
  float pos[3];
  float vel[3];
  bool used;
} shot_t;
```
pos and vel should be self-explanatory.
used is a boolean indicating whether or not the shot is active.

6 Miscellaneous TA Hints

6.1 Notes on the world_t structure

In world_init, a pointer to a world_t structure is passed to your function. You should probably create a
global variable and save this pointer. You will need it to access information about the world (such as the
location of players, flags, etc.) in your robot_update function.

You should not make a private copy of the structure. Much information is volatile, and changes regularly
(such as player positions, flag positions, etc.). Of course, these changes will not be reflected in your private
copy.

You should also not make a copy of the players member of the world_t structure. This is a volatile
pointer which points to the head of a linked list. As players are added and deleted, the value of this pointer
will change. If you copy the pointer, and decide to use it later (after other players have been added, for
example), a traversal of the linked list may not end up visiting every player_t structure.

Of course, nothing in the world_t structure will change during a call to robot_update, but information
may change between calls.

6.2 Notes on the player_t linked list

The player_t linked list is a singly linked list. It is not doubly-linked, nor is it circular. The head of the
linked list is always the players member of the world_t structure. For the purposes of traversing the linked
list, code should always start at this pointer, and continue until a NULL value is encountered.

There is a player_t pointer which is passed into the robot_update function. This is a pointer to an entry
somewhere in the linked list, and is provided as a convenience to you. If you attempt to traverse the linked
list from this pointer, your traversal will probably be incomplete.

6.3 Notes on when flags are active

For the potential fields lab, you may assume that we will run the TA agents first, then your agent. This will
ensure that their flag is active when your functions are called.

However, no flags are active when the world_init function is called (this is a deficiency that we intend to
remedy).

6.4 Notes on the world coordinate system

- (0,0,0) is right in the middle of the world. The world extends to $\pm \frac{1}{2}$ world_size (a member of the
  world_t structure) in both x and y dimensions.
- All angle measurements are in radians. 0 rads is due east. Some functions or variables use values
  ranging from $-\pi$ to $\pi$ while other functions or variables deal with ranges between 0 and $2\pi$. If you
  aren’t absolutely certain in a particular situation, do a test or make verbose output to make sure you
  know what’s happening. This is a consequence of dealing with periodic functions. Be very careful.
  Work things out on paper.
- Passing a 1.0 as the angvel instruction will turn the agent to the LEFT. Passing $-1.0$ will turn the
  agent to the right. This may be counter-intuitive in our right-handed culture, but it makes sense from
  an angle perspective (a 1.0 implies that your angle will get larger).
6.5 Notes on observers in windows

To facilitate testing, you may want to know that you can start the observer in a window. To do so, just run:

```
$ cd bzflag
$ ./observe.sh --window --geometry 600x600
```

(of course, you can replace "600x600" with whatever you want).

This should allow you to have other windows (like terminals) open to start and stop bots, etc. You may have problems with the mouse. In order to allow the mouse to leave the bzflag window, you need to add the following line to the bzflag resource file (typically `~/.bzflag`):

```
mousegrab no
```

You may still have problems with getting the mouse out of the bzflag window and into another window without sending the roamer’s camera crazy. We use the following procedure, which seems to work quite well:

- Open a terminal window. Reposition the terminal so that it is in the upper-left corner of your desktop.
- Start an observer in a window. The observer will cover the terminal (in part, or in full; it doesn’t matter).
- Press alt-tab to cycle through windows, until the terminal which is covered by the observer becomes selected. Assuming that your window manager is set up so that selected windows are raised, you should be able to grab the terminal window and drag it somewhere else. This will have the effect of allowing you to move the mouse out of the observer window and into another area of the desktop.

Of course, you don’t have to use a terminal. Any window will do.

7 Known Issues

- Once activated, a flag is never deactivated. It should be deactivated once all players of a given color have been removed.
- When a robot decides that the game is over, it typically just exit()’s. This is not very nice. Should exit gracefully.
- Currently, the resurrect command-line switch is implemented, but not functional. Robots always resurrect once they finish exploding.
- Occasionally, a client will repeatedly process resurrect messages. This may result in multiple calls to `robot:resurrect` for a robot that is already alive.
- When a player is booted out of the game, that player’s entry in the players linked list (stored in the `world` structure) is not removed.