Potential Field Navigation

ROB 102: Introduction to AI & Programming

Lab Session 5

2021/10/15

Administrative

The next lecture on creating potential fields will be released this weekend.

Field trip! We will be visiting Ford's autonomous cars at MCity next Friday at 12PM.

Today…

1. Updates to the template code
2. Crash course in Git branching
3. “BotLab” (mapping + localization) overview
4. Potential field navigation on the robot

TODO Today:

1. Pull instructor upgrades into autonomous navigation repository
2. (Encouraged) Finish this week’s in-class activities
3. Make an attraction potential & use it to navigate the robot

Code Updates

You should see a couple new commits in your autonomous navigation repositories. Make sure you update your code:

```
git pull
```

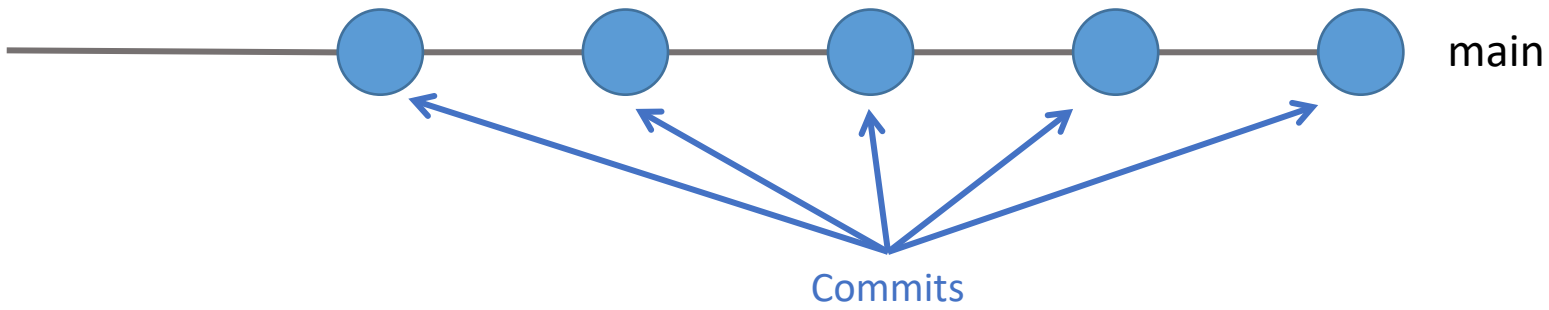
We changed:

1. The Dockerfile: `./docker_build.sh` will now update the nav app if new changes have been pushed to it (rebuild the container to see the updates!)
2. New maps: A couple new maps have been added which are better environments to test potential field navigation.
3. Fixes to local search (which we'll use to drive the robot today).

Git Branching

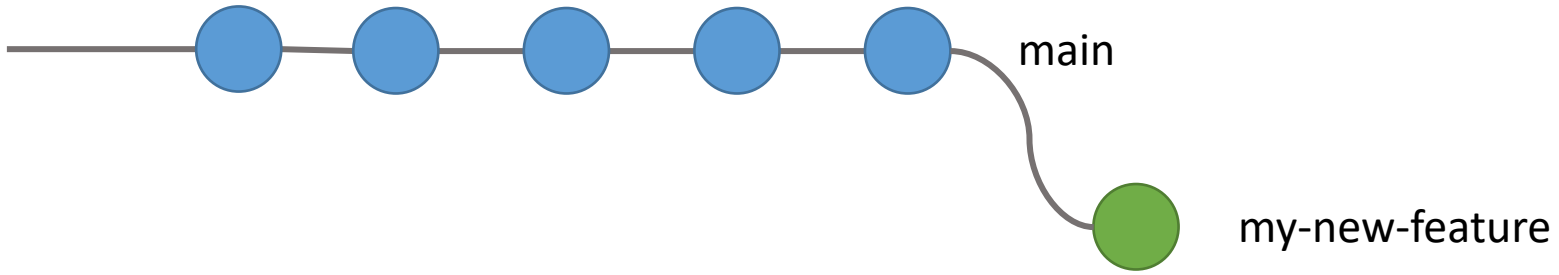
Branches in Git are a useful tool for developing new features in your code, especially if you are collaborating on the code (like we are!).

We already have one branch in our repositories: the main branch.



Git Branching

We can create a new branch off the main branch. Then, we can modify the code and add commits to that branch.



Creating a new branch (from the branch you are on):

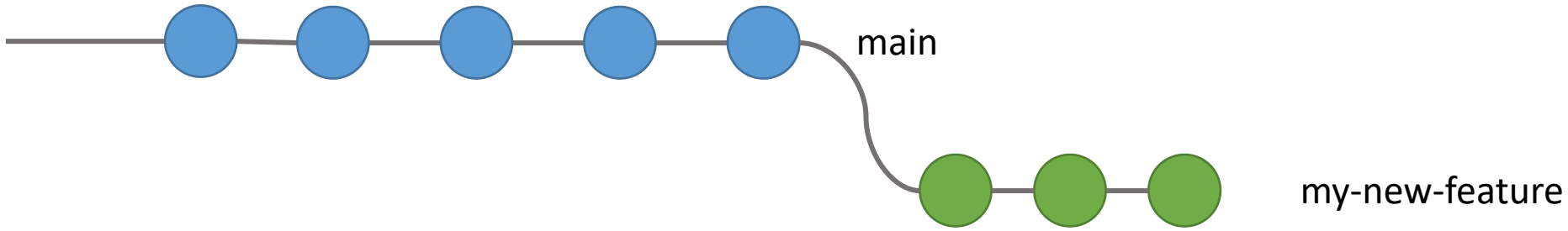
```
git branch my-new-feature
```

Changing branches:

```
git checkout some-branch
```

Git Branching

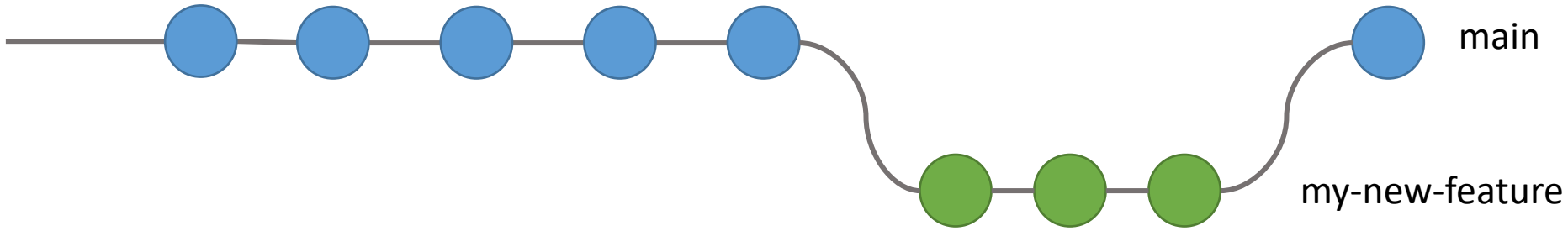
We can create a new branch off the main branch. Then, we can modify the code and add commits to that branch.



The benefit of making a new branch for a new feature is that the main branch still works while you're developing and testing new code.

Git Branching

When you finish developing your new feature, you can **merge** your changes back into the main branch.



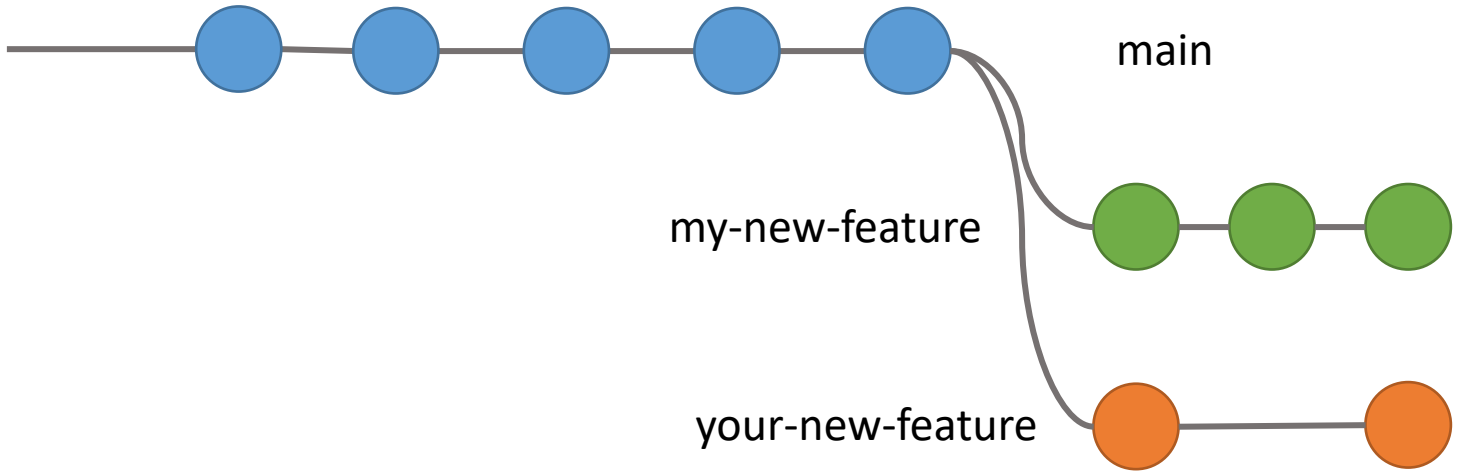
Merging a branch into the current branch:

```
git merge some-branch
```

Now all the changes from my-new-feature will be in main.

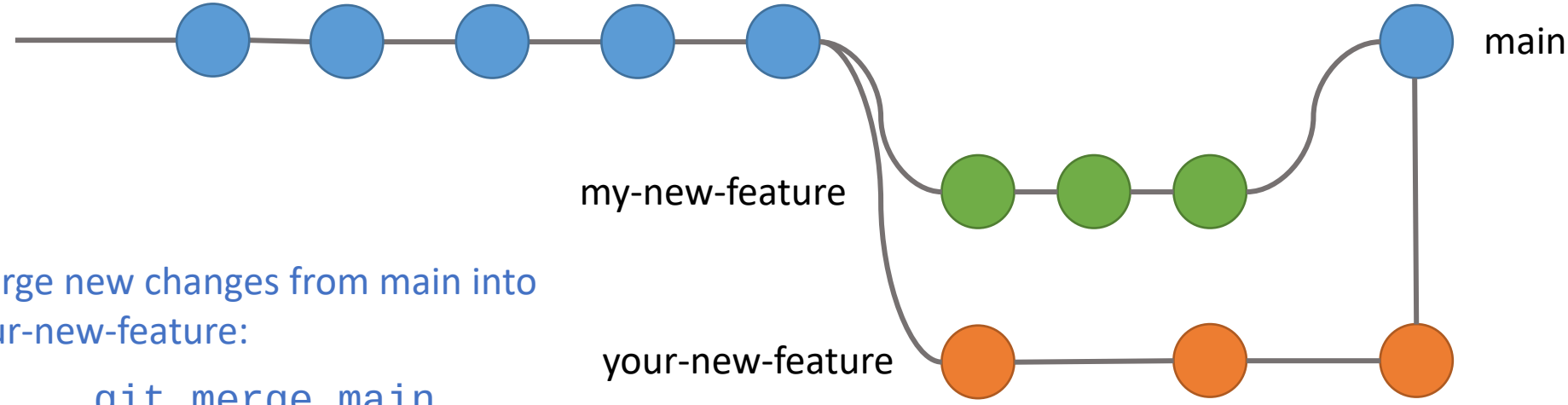
Git Branching: Collaborative Development

Branches are a very useful tool when multiple people are working on the same code.



Git Branching: Collaborative Development

Branches are a very useful tool when multiple people are working on the same code.

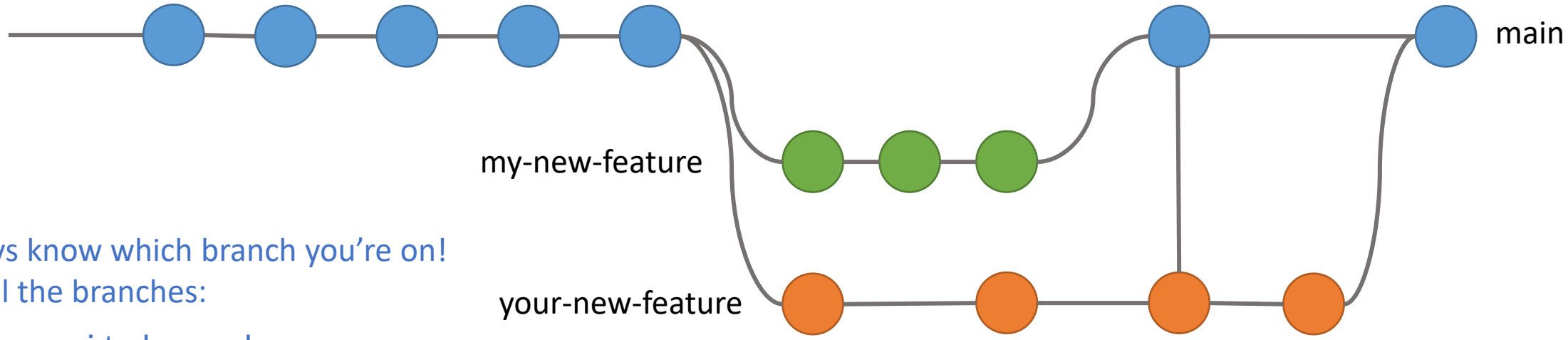


Merge new changes from main into your-new-feature:

```
git merge main
```

Git Branching: Collaborative Development

Branches are a very useful tool when multiple people are working on the same code.



Always know which branch you're on!
List all the branches:

```
git branch
```

Or check with `git status`.

Git Branching: Guidelines

1. Each person should be working on their own branch
2. Create one branch per feature
   - Giant branches with many changes (like branch “janas-code” where I write all my code) are a bad idea because they get out of sync with other branches.
3. Merge changes from main often so your branch doesn’t deviate too much
4. The main branch should always work
   - Only merge changes into main when you’ve thoroughly tested your code.

BotLab: Localization & Mapping Code

The code we call “BotLab” performs Simultaneous Localization and Mapping (SLAM) on the robot. It has the following executables:

1. `timesync`: Synchronizes time between sensors.
2. `rplidar_driver`: Reads from the LiDAR and shares the data with the other processes
3. `motion_controller`: Listens for goals (like a path or a clicked point in the GUI) and sends control commands
4. `slam`: Maps and localizes.
5. `botgui`: Visualizes robot data like map, pose and LiDAR scan.

BotLab: Motion Controller

Motion controller *must* be running to control the robot by clicking the BotGUI, or with the in-class potential field control activity.

Motion controller *must not* be running if you are running your own code that calls the drive() function. They will interfere with each other. To launch without starting motion controller:

```
./launch_botlab.sh -l
```

BotLab: SLAM

SLAM can be run in mapping + localization mode or localization only mode (with a given map).

SLAM must be running in *mapping + localization mode* to make a map. The map is saved in `~/botlab-bin/maps/current.map`.

SLAM must be running in *localization only mode* when you run code for Project 2 and 3 (since you will define the field over a fixed map). To run SLAM in localization only mode:

```
./launch_botlab.sh -m [PATH/TO/MAP]
```

BotLab: Common Issues

Always stop the code with `./cleanup_botlab.sh` before launching again to avoid having many versions of the executables running together.

Use BotGUI to make sure your robot is localized correctly.

Make sure your battery is charged.

BotLab: Common Issues

My robot is lost!

- Sometimes this happens! Restart by calling the cleanup and launch scripts again. (A hack: pick up your robot and move it to where it thinks it is!)
- Did your map change? Small changes to the map can have a big effect on localization.
- Did you make a good map? If your map doesn't have enough interesting features for the robot to recognize (it's in a huge open area, or many parts of the map look exactly the same), localization will be difficult.
- Are you in the map? Our SLAM code does not have the ability to deal with moving obstacles (yet!).
- Did your robot bump into an obstacle or slip? Did you move the robot? Localization can be sensitive to big differences in where the robot thinks it should be moving and where it actually moved.

Robot Nav Code

```
47      std::vector<float> field(graph.width * graph.height, 0);
48      /**
49       * TODO (P2): Call your function to create a potential field, and store the
50       * result in field.
51       **/
52
53      // Initialize the pose listener. The robot's state will be stored in (x, y, theta).
54      float x = 0, y = 0, theta = 0;
55      initPoseListener();
56
57      /**
58       * TODO (P2): Define any variables you need.
59       **/
60
61      while (true)
62      {
63          // Check for a new pose.
64          handle();
65          getPose(x, y, theta);
66
67          // Do local search on the graph from the current position.
68          std::vector<float> v_grad = localSearch(x, y, theta, graph, field);
69
70          /**
71           * TODO (P2): Use the vector pointing in the direction of steepest
72           * descent of the potential field to drive the robot.
73           *
74           * v_grad contains 3 elements: {vx, vy, grad}. vx and vy form a vector
75           * with magnitude 1 that point in the direction of steepest decrease
76           * starting from the robot's current position. grad is the maginutude of
77           * the decrease in potential.
78           **/
79
80          if (ctrl_c_pressed) break;
81      }
```

Calculate the field given the goal cell and the graph (**your code**)

Listens for poses from the localization system.

Waits for a new pose

Gets the direction and magnitude of the largest decrease in potential

Your code: Navigate using the direction of largest decrease.

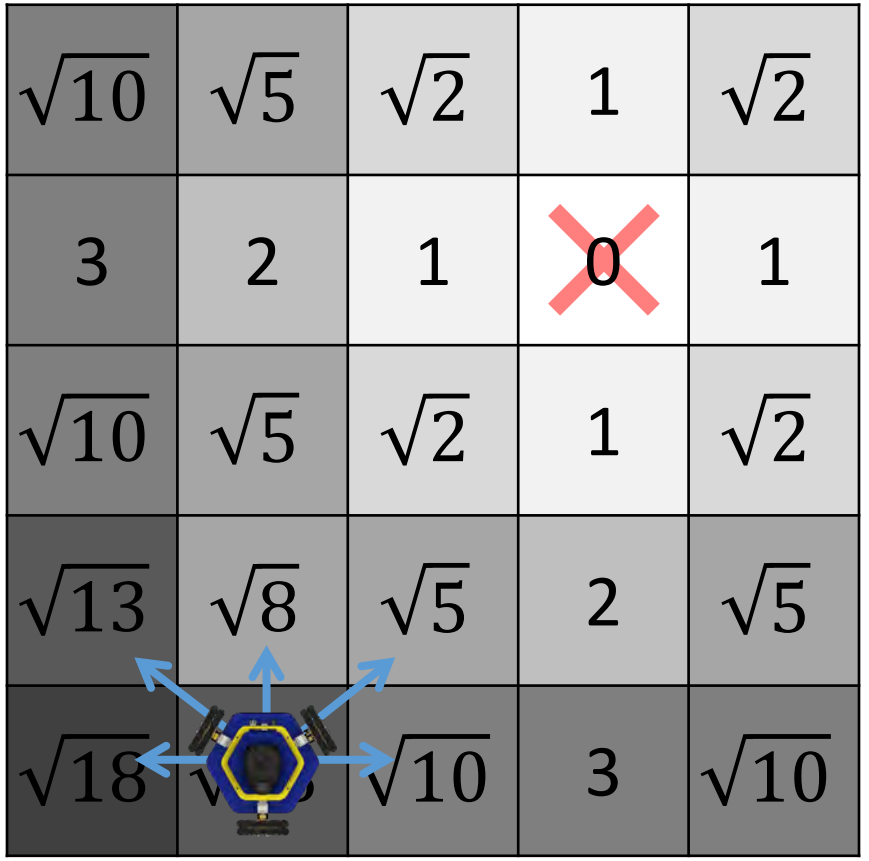
src/robot_potential_field.cpp

Potential Field Navigation on the Robot

Start by getting the robot to drive using an attraction potential only.

Build the field using function:

```
createAttractivePotential()
```

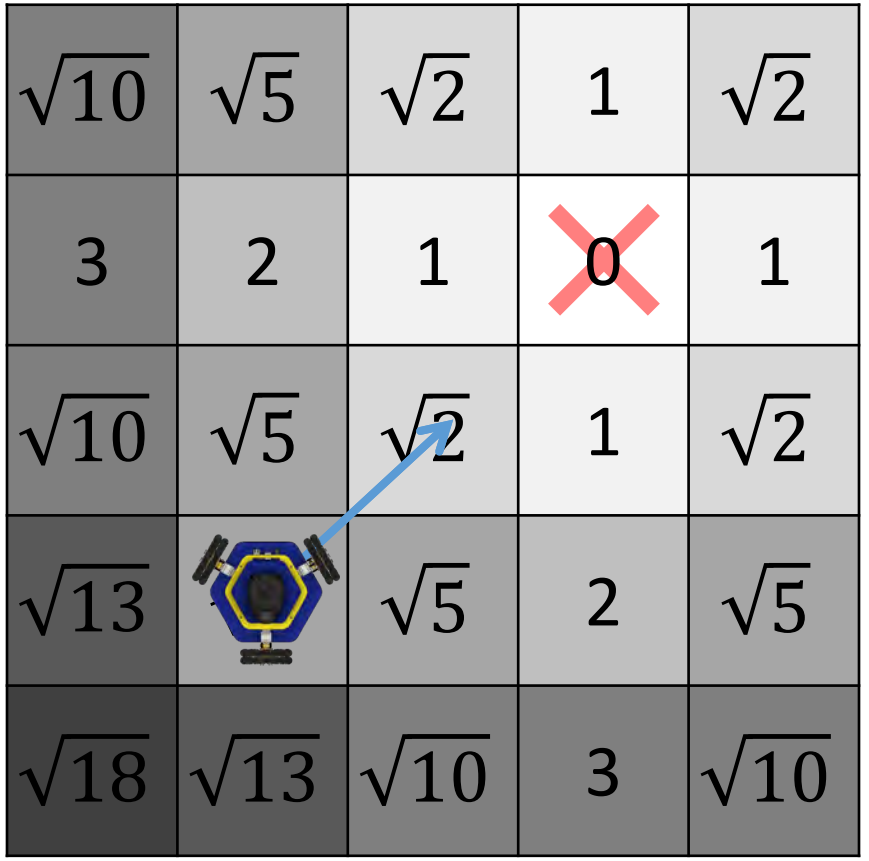


Local Search

Local search returns vector:

```
{vx, vy, grad}
```

The robot should drive in the direction of this vector.

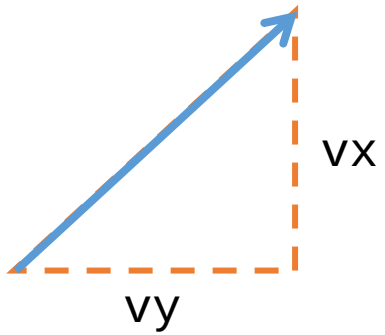


Local Search

Local search returns vector:

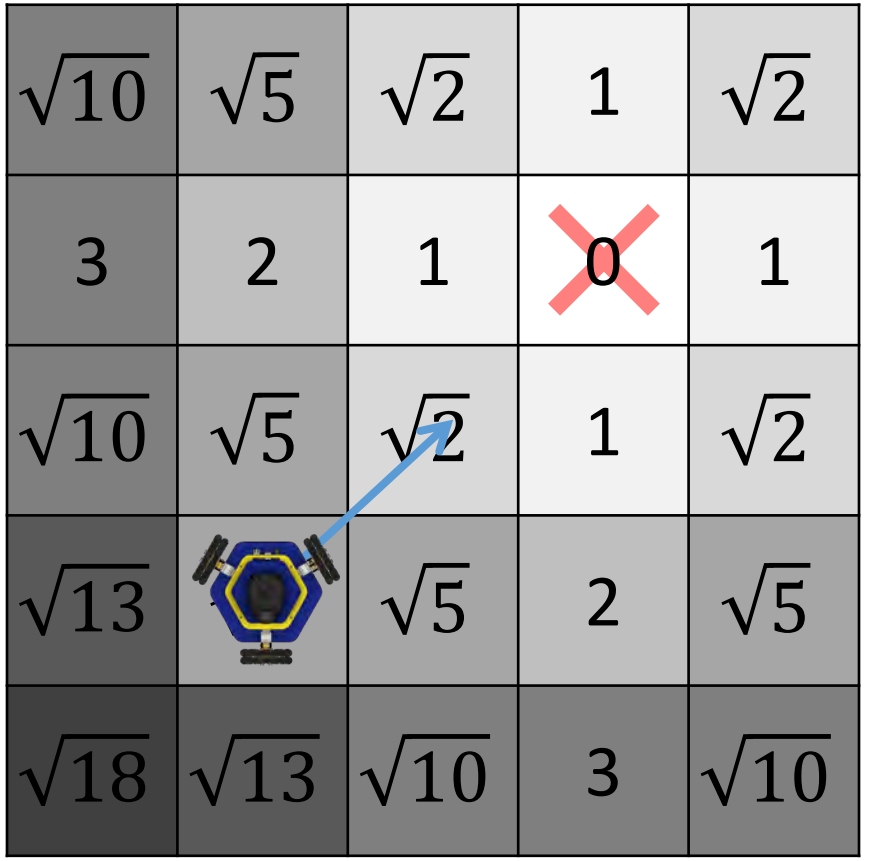
```
{vx, vy, grad}
```

The robot should drive in the direction of this vector.




```
grad = sqrt(8)-sqrt(2)
```

The magnitude of potential decrease. Can be used to control the velocity of the robot.



Local Search

To make control smoother, local search performs a lookahead.

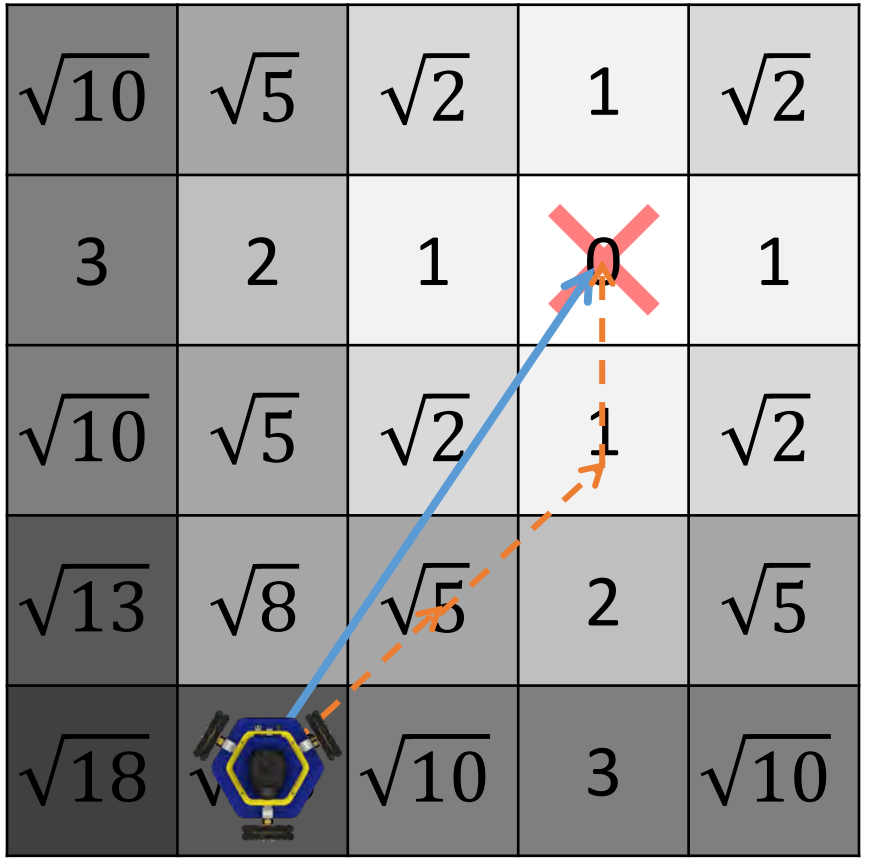
It will search for the lowest potential `depth` neighbors away.

You can tune the size of the lookahead if you are trying to get better performance:

```
local_search(x, y, theta, graph, field, depth)
```

Optional! If not provided, default is 5.

This example: depth = 3



Potential Field Navigation on the Robot

To compile the code on the robot, do:

```
cd build/
cmake -DOMNIBOT=On ..
make
```

Important! Robot code is only compiled on the robot (not in Docker)

Start the localization (you might want to use NoMachine and the BotGUI):

```
~/botlab-bin/launch_botlab.sh -l -m [PATH/TO/MAP]
```

Use a fixed map.

No motion control since our code will drive the robot.

Run your code on the robot:

```
./robot_potential_field [PATH/TO/MAP] [goal_x goal_y]
```

Can provide goal x and y position in meters

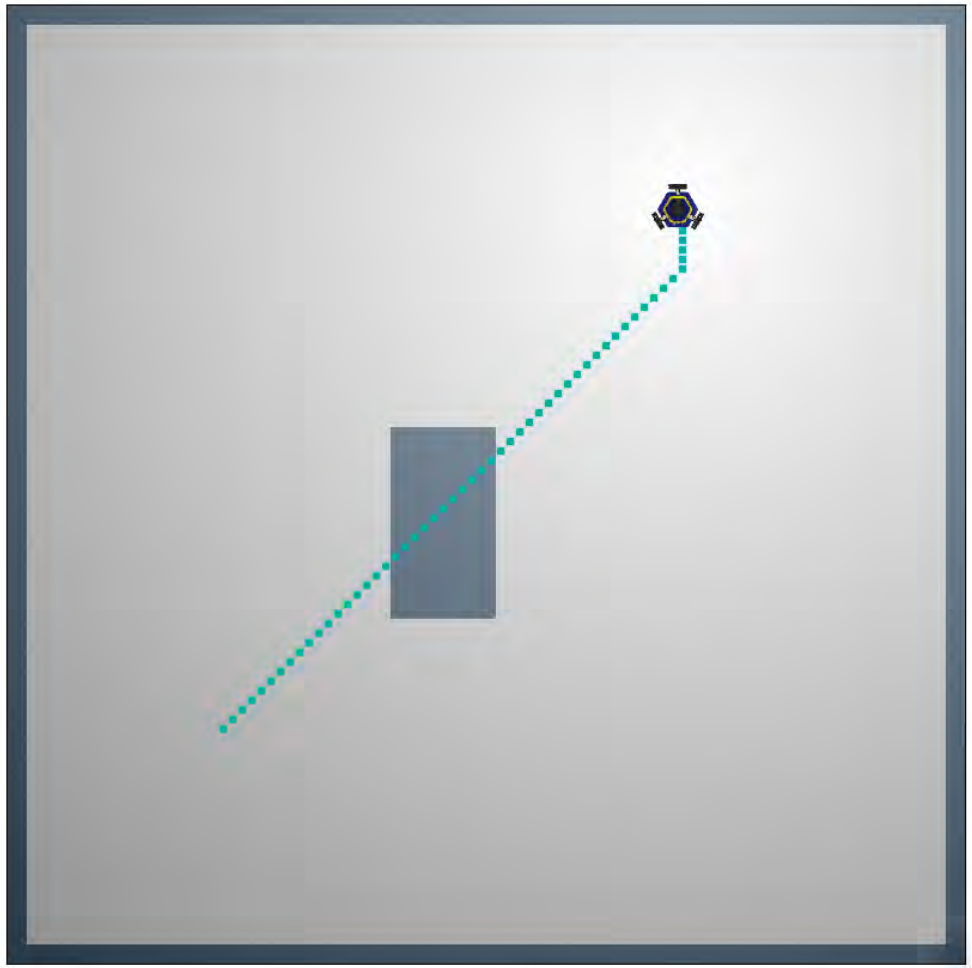
Use the same map to compute the field

Attraction Field on the Webapp

Modify `createPotentialField()` to simply call `createAttractionField()` and set `potential_field` equal to the result.

Compile and run the web app server (`nav_app_server`).

In the webapp, pick a map and a goal, and planning algorithm “Potential Field.” Click “Plan!” to see the resulting path. You can visualize the field.



TODO Today

1. Update your code with instructor changes (git pull).
2. (Encouraged) Finish Monday's in-class activity (draw a field).
3. Work on P2.1: Attractive potential navigation on the robot.

Focus on the robot code today! If you have extra time, or for homework:

- (Encouraged) Finish Wednesday's in-class activity (distance transform).