SailSim Navigation Interface

By the 2017 HSDC SailSim Team

30 April 2017
The 2017 HSDC SailSim Team is:

*Mike Freeman, Ellery Chan, Scott Minster, Nathan Dymora, Luis Maura, Jeremy Moore, Danny McMillan, Jaron Blackburn, Colin Chan, Taco Bell®*
# Table of Contents

SailSim Navigation Interface ............................................................... 1

Introduction ......................................................................................... 6
  Sailing and Other Nautical Terminology ........................................ 6
  Navigation Client ................................................................. 6
  Simulation Server .............................................................. 6
  Display Client ................................................................. 7
  Navigation Interface ............................................................ 8
  System Overview ............................................................... 8

The Simulation Environment ............................................................... 9
  The Simulation Gaming Area .................................................. 9
  Coordinates ........................................................................... 10
    Location .............................................................................. 11
    Heading ............................................................................... 11
    Sail Angle ........................................................................... 12
  Wind Strength and Wind Heading ........................................... 13
  Obstacles .................................................................................. 14
    Obstacle IDs ....................................................................... 14
    Special Obstacles ............................................................. 14
    Detecting Obstacles .......................................................... 15

The Navigation Interface ........................................................................ 17
  Navigation Interface API .......................................................... 18
  Including the Module ............................................................... 18
    Java ..................................................................................... 18
Module Entry Points

connect() ........................................................................................................... 20
  Java .................................................................................................................... 20
  Python 3 ............................................................................................................. 21
  C++ ..................................................................................................................... 22
  C# ......................................................................................................................... 23
  Node.js .............................................................................................................. 24

send() .................................................................................................................. 26
  Java .................................................................................................................... 26
  Python 3 ............................................................................................................. 27
  C++ ..................................................................................................................... 27
  C# ......................................................................................................................... 28
  Node.js .............................................................................................................. 28

receive() ............................................................................................................ 29
  Java .................................................................................................................... 29
  Python 3 ............................................................................................................. 29
  C++ ..................................................................................................................... 30
  C# ......................................................................................................................... 30
  Node.js .............................................................................................................. 31

disconnect() .................................................................................................... 32
  Java .................................................................................................................... 32
  Python 3 ............................................................................................................. 32
  C++ ..................................................................................................................... 32
  C# ......................................................................................................................... 33
  Node.js .............................................................................................................. 33

isConnected() ................................................................................................. 34
  Java .................................................................................................................... 34
  Python 3 ............................................................................................................. 34
  C++ ..................................................................................................................... 35
  C# ......................................................................................................................... 35
  Node.js .............................................................................................................. 35

Navigation Interface Protocol ................................................................................. 37

Message Formats .................................................................................................. 37
Message Exchange ................................................................................................. 37
Message Rate Throttling .......................................................................................... 38
Query Messages (client → server) ........................................................................... 39
Modification Messages (client → server) ..................................................................... 40
Response Messages (server → client) ............................................................................ 41
Message Details ............................................................................................................ 41
  anchor ....................................................................................................................... 42
  atDestination ............................................................................................................. 43
  atObstacle .................................................................................................................. 44
  boatHeading .............................................................................................................. 45
  boatPosition .............................................................................................................. 47
  boatSpeed .................................................................................................................. 48
  error ............................................................................................................................ 49
  finish ........................................................................................................................... 50
  goalPosition ............................................................................................................... 51
  hello ......................................................................................................................... 52
  obstacle ..................................................................................................................... 53
  obstacleCount ............................................................................................................ 54
  obstacleDistance ....................................................................................................... 55
  obstacles ................................................................................................................... 56
  obstacleScan ............................................................................................................. 57
  sailAngle ................................................................................................................... 58
  time ............................................................................................................................ 58
  windHeading ............................................................................................................. 60
  windStrength ............................................................................................................. 61
  Example ...................................................................................................................... 63
Introduction

**SailSim** is a virtual environment for sailing. You create a sailboat and navigate it to a destination. The virtual environment simulates the physical interaction of the wind, the water, and the boat. You control the boat’s *heading* and *sail angle* to drive the boat where you want it to go.

First, some terminology:

**Sailing and Other Nautical Terminology**

People have been sailing the seven seas for far longer than written history, so it is maybe not surprising that there is a vast assortment of boating words and phrases. If you see an unfamiliar term, you might want to try looking it up here: [https://en.m.wikipedia.org/wiki/Glossary_of_nautical_terms](https://en.m.wikipedia.org/wiki/Glossary_of_nautical_terms).

We hope we are using the traditional nautical terms correctly. If not, let us know! Now, here is some new terminology of our own:

**Navigation Client**

The *Navigation Client* controls the behavior of the sailboat. You will create a Navigation Client to drive your sailboat in a variety of different conditions.

**Simulation Server**

The *Simulation Server* controls the virtual environment and does the physics calculations to determine how your boat moves. Your Navigation Client will send messages to and receive messages from the Simulation Server to control your sailboat.

It is called a "server" because it provides a service to some clients. In this case, the Simulation Server provides the service of doing the physics calculations to figure out how your boat will sail, and reports information about how it is doing.
Display Client

The *Display Client* lets you view the progress of the simulation. The Display Client is a web app that presents a graphical view of the simulation in a web browser. This means you can watch the progress of your simulation run in Chrome, Firefox, or Safari on any Windows, Mac, Linux, iOS, or Android device.

You can also connect a Display Client to any other simulation and watch the boats there, too. Of course, that means that other “guests” could also be checking out your sailing skills!

The screenshot below shows what the Display Client looks like. Its appearance will probably change, as it is still under development, but the main elements will still remain: the blue water, the goal circle, the wind arrow. This simulation course currently has one boat belonging to team *hsdc*.
**Navigation Interface**

Your Navigation Client communicates with the Simulation Server via a *Navigation Interface* using a simple protocol called the *Navigation Interface Protocol*. (Who would have guessed?)

The Navigation Interface and the Navigation Interface Protocol are the primary topics of this document. In order to create a working Navigation Client, you will need to understand how to establish a connection with the Simulation Server, how to get information about the current state of your boat and the simulation environment, and how to send commands to operate your boat.

**System Overview**

The Simulation Server runs *in the cloud*, which is just a vague way of saying it runs on a machine somewhere on the Internet. The Navigation Client runs on your device. The Display Client could also run on the same device, or on a different device. Multiple Display Clients can observe the same simulation. It is also possible for multiple Navigation Clients to control different boats in the same simulation, but this feature may not be enabled.

*Figure 1: How the parts of the simulation environment work together*
The Simulation Environment

Your sailboat will need to know how to navigate the virtual waters of our simulated courses. Ancient sailors navigated by the Sun, Moon, and stars. You will use Cartesian coordinates and heading angles to steer your boat to the goal. Virtual sensors can read the wind strength and direction, and can detect obstacles.

The Simulation Gaming Area

The Simulation Gaming Area is a rectangular body of water bounded on all four sides by the shore. The water extends 2000 meters in the East-West direction and 1500 meters in the North-South direction. Your sailboat must not contact the shore: contacting the shore ends your sailboat’s simulation run. This is explained further in the section on Obstacles below.
Coordinates

Locations in the Simulation Gaming Area are specified by Cartesian coordinates \((x, y)\). The origin of the coordinate system is in the Southwest corner of the lake. \(x\) and \(y\) are floating point numbers.

Valid water coordinates will satisfy these relationships:

\[
0 < x < 2000 \\
0 < y < 1500
\]

You may want to use values outside the water range in certain calculations, but don’t try to guide your boat there!
Location

The Simulation Server uses Cartesian coordinates to specify the location of different objects in the simulation: your sailboat, the goal, the shore, and other obstacles. The location of your sailboat will (should!) be changing, but the locations of the other simulation elements will not change. Your Navigation Client can ask for the location of specific objects by sending specific Navigation Interface Protocol messages.

Heading

Your Navigation Client never sends location coordinates to the Simulation Server. You control your boat by specifying the boat’s heading angle and the sail angle (see Sail Angle).

Heading values (sometimes referred to as azimuth values in navigation and astronomy) are angles around the compass, where 0° is North, 90° is East, 180° is South, and 270° is West.
As an example, if the wind heading is 270° (blowing from East to West) and you set the boat heading to 292.5°, then the boat will travel in a west-northwesterly direction. This situation is shown in the following diagram.

### Sail Angle

The sail angle is also specified in degrees, but the sail angle is not measured relative to the compass. Instead, it is relative to the orientation of the sailboat. The sail can move through 180°. The angle is measured from -90° to +90°.

\[-90° \leq \text{sail angle} \leq 90°\]
When the sail angle is 0°, the sail is parallel to the hull centerline of the sailboat. The *sail angle* value describes the rotation of the sail clockwise (looking down) from its center position. Some different sail angles are illustrated in the following diagram.

Wind Strength and Wind Heading

You can query the current simulation wind strength and wind heading direction. These values are constant for a given simulation run. The wind strength is given in units of newtons. The wind direction is a heading value as described above.
Obstacles

Obstacles are things that the sailboat can run into. Sailboats are not obstacles, so if there are multiple sailboats sailing in the same simulation, they will not collide. If a sailboat collides with an obstacle, it stops moving, and its simulation run is considered to be over, since it will never move again in that run.

There are three types of obstacles: “normal” obstacles, the goal, and the shore.

Your Navigation Client can detect when the boat has hit an obstacle. See the atDestination and atObstacle messages in the Message Details section. atDestination will tell the Navigation Client when the boat has reached the goal, indicating a successful run. atObstacle will tell the Navigation Client when the boat has hit a normal obstacle or the shore, indicating a failed run.

All obstacles in the water are represented as circles defined by a center point and a radius. The goal is also a circle defined by a center point and a radius. The shore is a rectangle whose dimensions are the size of the simulation area.

Obstacle IDs

The Simulation Server keeps all the obstacles in a numbered list. Your Navigation Client can reference specific obstacles by their Obstacle IDs which are simply integers which correspond to their positions in the list.

<table>
<thead>
<tr>
<th>ID</th>
<th>Obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shore</td>
</tr>
<tr>
<td>1</td>
<td>Goal</td>
</tr>
<tr>
<td>2</td>
<td>1st Obstacle</td>
</tr>
<tr>
<td>3</td>
<td>2nd Obstacle</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Special Obstacles

The Goal

The goal always has Obstacle ID 1. The goal is a circle. Its radius has been set to 20 m for development purposes, and will probably remain that way for the competition, but your Navigation Client should get its actual radius at race time.
The Shore

The shore always has Obstacle ID 0. It defines a rectangular body of water where all the sailing takes place. Its size is 2000 meters East-West by 1500 meters North-South. Its size will remain unchanged during the competition. But your Navigation Client can get its actual size at race time.

Detecting Obstacles

Your sailboat can detect obstacles using virtual sensors.

Virtual Radar

`obstacleScan` activates the sailboat’s `virtual radar`, which returns a list of `heading, distance` pairs, one pair for each obstacle.

The first pair in the returned list (pair 0) corresponds to the obstacle with ID 0, the shore. Since the shore is reachable in all directions, the heading for the shore obstacle is always set to -1, and the distance is set to 0.

The next pair in the returned list (pair 1) gives the heading from the sailboat’s current location to the goal, and the distance to the goal.

Subsequent pairs return the heading from the sailboat’s current location to the next obstacle in the list, and the distance from the boat to that obstacle.

Virtual Laser Rangefinder

`obstacleDistance` points a virtual laser rangefinder from the boat in a heading direction that the Navigation Client can specify, and returns the ID of the nearest obstacle that was hit, and the distance to the obstacle. Note that this heading direction is independent of the boat’s current heading.

If the ID is 0, there was nothing in the water in the heading direction, so the first obstacle to be hit was the shore. In that case, the distance to the shore in that direction is returned.

If the ID is 1, the goal is the closest object in the specified heading direction, and the distance to the goal is returned.
If the ID is greater than 1, another obstacle was hit, and the distance to that obstacle is returned.

The gray box in the diagram shows a message exchange between a Navigation Client and a Simulation Server. These messages will be described later in this document.

**Virtual Obstacle Database**

The Simulation Server maintains a virtual database of the obstacles. The Navigation Client can find out the number of obstacles in the simulation by sending the `obstacleCount` message, which returns the number of obstacles. The number of obstacles returned will always be greater than or equal to 2, because every simulation scenario will have at least a shore and a goal.

Individual obstacles can be queried by their ID using the `obstacle` message. `obstacle` returns the obstacle’s location coordinates and the obstacle’s radius. For the shore, the radius is the distance from the center of the rectangle to one of the corners.
The Navigation Interface

The Navigation Interface allows the Navigation Client to communicate with the Simulation Server over a network. Here is a technical description that you don’t need to understand in detail. But if you want to find more information, Googling some of the terms here would be a good place to start.

The Navigation Interface uses TCP/IP sockets to perform interprocess communication (IPC) between two processes that may be running on different hosts (machines). To hide a lot of messy, low-level details, the Simulation Server and Navigation Client will both use a network communication library called ZeroMQ (or 0MQ or ZMQ) to establish the connection and send messages back and forth.

We want you to pay attention to your driving and not be distracted by sending text messages, so we have created a Navigation Interface API (Application Programming Interface) that does the dirty work. It hides the details of the ZMQ library and provides a few simple functions for sending and receiving messages and managing the Navigation Interface connection.

We will provide the Navigation Interface API code for (almost) whatever language you wish to use to implement your Navigation Client. It will just be a small module or package that you will include with your program.

We currently have Navigation Interface API modules for the following languages:

- Java
- Python
- C++
- C#
- Node.js
If you would like to use a different programming language, let us know. We can either create a new module for that language, or (less likely) tell you that it would be too difficult and you should choose another language to construct your Navigation Client.

To establish the Navigation Interface connection (which at a lower level is a TCP network socket, since you were surely wondering) you will need to know a host and a port number. For the competition, you will use the hostname `sim.sailsim.org` and the port number `20170`. However, if possible, you should make it possible to change the host or port in your code, in case it should change, or you should need a different value during your development and testing. The host is more likely to change than the port number. An IP address can also be used in place of a host name.

**Navigation Interface API**

**Including the Module**

The code needed to incorporate the Navigation Interface API into your Navigation Client depends on the programming language you use. (This module is called the *Navigation Interface Client* because it implements the *client* end of the Navigation Interface. There is also a *Navigation Interface Server* piece on the other end of the connection. Do not confuse the *Navigation Interface Client* module with your *Navigation Client* program even though the names are similar.)

Here are examples of how to include the Navigation Interface Client module using the languages we currently support:

**Java**

```java
import NavInterfaceClient;
```

**Python 3**

```python
from NavInterfaceClient import NavInterfaceClient
```
Module Entry Points

The Navigation Interface API has a few simple entry points.

- `connect()` – establish a connection between the Navigation Client and the Simulation Server
- `send()` – send a message to the Simulation Server
- `receive()` – receive a response message from the Simulation Server
- `disconnect()` – close the connection between the Navigation Client and the Simulation Server
- `isConnected()` – check the status of the connection

Each of these operations will be more fully describe, with examples, in the following sections.
**connect()**

**Purpose:** Establish a connection between the Navigation Client and the Simulation Server

**Returns:** A Boolean value that is True if the connection was successfully established

To establish a connection with the Simulation Server, you will need four pieces of information:

1. Your team’s *username* (a string)
2. Your team’s *password* (a string)
3. The Simulation Server *hostname* or IP address (a string)
4. The Simulation Server connection *port number* (an integer)

`connect()` establishes communication between your Navigation Client process and the Simulation Server, then it sends the `hello` message with your team’s *username* and *password* (see `hello` in Message Details below) and receives a response. If the response is `hello` back from the Simulation Server, the connection is successful.

### Java

**Declaration**

Throws an IOException if an error occurs.

```java
public boolean connect(String host,
                      int port,
                      String username,
                      String password) throws IOException;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>A String containing the Simulation Server connection hostname (ex. “sim.sailsim.org”) or an IP address (ex. “192.168.1.5”)</td>
</tr>
<tr>
<td>port</td>
<td>An integer containing the Simulation Server connection port number (ex. 20170)</td>
</tr>
<tr>
<td>username</td>
<td>A String containing the team’s team username (ex. “keel haulers”)</td>
</tr>
<tr>
<td>password</td>
<td>A String containing the team’s team password (ex. “n3v3r_surr3nd3r”)</td>
</tr>
<tr>
<td>Return value</td>
<td>true if the connection was made successfully, or false if something went wrong (check all your parameters)</td>
</tr>
</tbody>
</table>
Example

```java
/** Include the module in your code */
import NavInterfaceClient;

/** Create the NavInterfaceClient object */
NavInterfaceClient simAPI = new NavInterfaceClient();

/** Get the parameters you will need to establish the connection. */
* These are hardcoded constants, but you may want to get them
* from the command line or from user input.
*/
String serverHost = "192.168.1.3";
int serverPort = 20170;
String username = "keelhaulers";
String password = "n3v3r_surr3nd3r";

/** Attempt to establish a connection with the server */
boolean connected;
connected = simAPI.connect(serverHost, serverPort, username, password);

/** Check whether the connection was successful */
if (connected) {
    System.out.println("Connection established.");
} else {
    System.err.println("Something went wrong…");
}
```

Python 3

Declaration

<table>
<thead>
<tr>
<th>NavInterfaceClient.connect(username, password, host, port)</th>
</tr>
</thead>
</table>

- **username**: A string containing the team’s team username (ex. “keelhaulers”)
- **password**: A string containing the team’s team password (ex. “n3v3r_surr3nd3r”)
- **host**: A string containing the Simulation Server connection hostname (ex. “sim.sailsim.org”) or an IP address (ex. “192.168.1.5”)
- **port**: An integer containing the Simulation Server connection port number (ex. 20170)
- **Return value**: **True** if the connection was made successfully, or **False** if something went wrong (check all your parameters)
import sys

# Include the module in your code
from NavInterfaceClient import NavInterfaceClient

...

# Get the parameters you will need to establish the connection.
# These are hardcoded constants, but you may want to get them
# from the command line or from user input.
serverHost = "sim.sailsim.org"
serverPort = 20170
username = "keelhaulers"
password = "n3v3r_surr3nd3r"

...

# Create the NavInterfaceClient object
simAPI = NavInterfaceClient()

# Attempt to establish a connection with the server
connected = simAPI.connect(username, password, serverHost, serverPort)

# Check whether the connection was successful
if connected:
    print("Connection established.")
else:
    print("Something went wrong…", file=sys.stderr)

---

C++

Declaration

```cpp
public void connect(const std::string& hostname,
                    int port,
                    const std::string& username,
                    const std::string& password);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostname</td>
<td>An <code>std::string</code> containing the Simulation Server connection hostname (ex. &quot;sim.sailsim.org&quot;) or an IP address (ex. &quot;192.168.1.5&quot;)</td>
</tr>
<tr>
<td>port</td>
<td>An integer containing the Simulation Server connection port number (ex. 20170)</td>
</tr>
<tr>
<td>username</td>
<td>An <code>std::string</code> containing the team’s team username (ex. &quot;keelhaulers&quot;)</td>
</tr>
<tr>
<td>password</td>
<td>An <code>std::string</code> containing the team’s team password (ex. &quot;n3v3r_surr3nd3r&quot;)</td>
</tr>
</tbody>
</table>
Example

```cpp
/** Include the module in your code */
#include "Client.h";

...

/** Create the NavInterfaceClient object */
Client *simAPI = new Client();

/** Get the parameters you will need to establish the connection. *
* These are hardcoded constants, but you may want to get them *
* from the command line or from user input. */
std::string serverHost = "sim.sailsim.org";
int serverPort = 20170;
std::string username = "keelhaulers";
std::string password = "n3v3r_surr3nd3r";

...

/** Attempt to establish a connection with the server */
simAPI->connect(serverHost, serverPort, username, password);

/** Check whether the connection was successful */
if (simAPI->isConnected()) {
    std::cout << "Connection established." << std::endl;
} else {
    std::cerr << "Something went wrong…" << std::endl;
};
```

C#

Declaration

```csharp
public bool connect (string host,
    int port,
    string username,
    string password);
```

| **host** | A string containing the Simulation Server connection hostname (ex. “sim.sailsim.org”) or an IP address (ex. “192.168.1.5”) |
| **port** | An integer containing the Simulation Server connection port number (ex. 20170) |
| **username** | A string containing the team’s team username (ex. “keelhaulers”) |
| **password** | A string containing the team’s team password (ex. “n3v3r_surr3nd3r”) |
Return value: **true** if the connection was made successfully, or **false** if something went wrong (check all your parameters)

Example

```csharp
/** Include the module in your code */
using NavInterfaceClient;

/** Create the NavInterfaceClient object */
Client simAPI = new Client();

/** Get the parameters you will need to establish the connection. 
* These are hardcoded constants, but you may want to get them 
* from the command line or from user input. */
string serverHost = "sim.sailsim.org";
int serverPort = 20170;
string username = "keelhauliers";
string password = "n3v3r_surr3n3r";

/** Attempt to establish a connection with the server */
bool connected = simAPI.connect(serverHost, serverPort, username, password);

/** Check whether the connection was successful */
if (connected)
{
    Console.WriteLine("Connected to server");
}
else
{
    Console.WriteLine("Something went wrong");
}
```

Node.js

Declaration

```javascript
NavInterfaceClient.prototype.connect = function(host,
    port,
    username,
    password);
```
<table>
<thead>
<tr>
<th><strong>host</strong></th>
<th>A string containing the Simulation Server connection hostname (ex. “sim.sailsim.org”) or an IP address (ex. “192.168.1.5”)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>port</strong></td>
<td>An integer containing the Simulation Server connection port number (ex. 20170)</td>
</tr>
<tr>
<td><strong>username</strong></td>
<td>A string containing the team’s team username (ex. “keelhauliers”)</td>
</tr>
<tr>
<td><strong>password</strong></td>
<td>A string containing the team’s team password (ex. “n3v3r_surr3nd3r”)</td>
</tr>
</tbody>
</table>

**Example**

```javascript
this.sim = new NavInterfaceClient(true);
this.sim.connect('localhost', 20170, 'keelhauliers', 'n3v3r_surr3nd3r');
```
**send()**

**Purpose:** Send a command to the Simulation Server

The command consists of a keyword with optional remainder. The command format is described below in the Navigation Interface Protocol section.

Every message sent to the Simulation Server results in a response message sent back from the Simulation Server. You must not call `send()` twice without calling `receive()` in between. This will result in a protocol error. You must alternate calls to `send()` and `receive()`.

**Returns:** Some versions return a Boolean value that is True if the connection was successfully established. Others return `void`. See the specific declaration for your language for details.

### Java Declaration

There are two versions of `send()` for the Java API. One takes just a keyword parameter, and the other takes a keyword parameter and a remainder parameter.

Throws an IOException if an error occurs.

```java
public void send(String keyword) throws IOException;
public void send(String keyword, String remainder) throws IOException;
```

<table>
<thead>
<tr>
<th>keyword</th>
<th>The command keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>remainder</td>
<td>The command remainder (optional)</td>
</tr>
</tbody>
</table>

**Example**

```java
simAPI.send("boatHeading", "33.3");
```
**Python 3**

**Declaration**

```python
NavInterfaceClient.send(keyword, value="")
```

<table>
<thead>
<tr>
<th>keyword</th>
<th>The command keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The command remainder (optional)</td>
</tr>
</tbody>
</table>

**Example**

```python
simAPI.send("boatHeading", "33.3")
```

---

**C++**

**Declaration**

There are two versions of `send()` for the C++ API. One takes just a keyword parameter (`command`), and the other takes a keyword parameter and a remainder parameter (`args`).

```cpp
public bool send(const std::string& command);
public bool send(const std::string& command, const std::string& args);
```

<table>
<thead>
<tr>
<th>command</th>
<th>The command keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>The command remainder (optional)</td>
</tr>
<tr>
<td>Return value</td>
<td><strong>true</strong> if the command was sent successfully, or <strong>false</strong> if an error occurs</td>
</tr>
</tbody>
</table>

**Example**

```cpp
simAPI->send("boatHeading", "33.3");
```
C#

Declaration

```csharp
public void send(string message, string value = "");
```

<table>
<thead>
<tr>
<th>Message</th>
<th>The command keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>The command remainder (optional)</td>
</tr>
</tbody>
</table>

Example

```csharp
simAPI.send("boatHeading", "33.3");
```

Node.js

Declaration

```javascript
NavInterfaceClient.prototype.send = function(keyword);
NavInterfaceClient.prototype.send = function(keyword, argument);
```

<table>
<thead>
<tr>
<th>Message</th>
<th>The command keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>The command remainder (optional)</td>
</tr>
</tbody>
</table>

Example

```javascript
this.sim.send('boatHeading', "33.3");
```
**receive()**

**Purpose:** Receive a response message from the Simulation Server. The response consists of a keyword with optional remainder. The response format is described below in the Navigation Interface Protocol section.

Every message sent to the Simulation Server results in a response message sent back from the Simulation Server. You may not call `send()` twice without calling `receive()` in between. This will result in a protocol error. You must alternate calls to `send()` and `receive()`.

**Returns:** a single string or multiple strings, depending on the format of the received message.

### Java

**Declaration**

Throws an IOException if an error occurs.

```java
public String[] receive() throws IOException;
```

**Return value** An array of Strings, where the first element is the keyword, and the second element (if present) is the remainder of the message.

**Example**

```java
String[] msg_parts = simAPI.receive();
```

### Python 3

**Declaration**

```python
NavInterfaceClient.receive()
```

**Return value** A list containing 0, 1, or 2 elements. The first element is a string containing the response keyword. The second element (if present) is a string containing the remainder of the response. If the list contains 0 elements, an error occurred or the connection is broken.
Example

```c++
response = simAPI.receive()
```

**C++**

**Declaration**

```c++
public std::string receive();
```

**Return value**  
An std::string containing the entire response. It may need to be separated into keyword and remainder.

**Example**

```c++
std::string response = simAPI->receive();
```

**C#**

**Declaration**

Throws an IOException if an error occurs.

```c#
public string[] receive();
```

**Return value**  
An array of strings. The first element is the keyword of the received command. The second element (if present) is the remainder of the command.

**Example**

```c#
string[] response = simAPI.receive();
```
Node.js

Declaration

NavInterfaceClient.prototype.receive = function();

Example

var receivedMessage = this.sim.receive();
**disconnect()**

**Purpose:** Terminate a connection with a Simulation Server  
Your Navigation Client should do this when it has completed a simulation run.

**Returns:** nothing.

### Java

**Declaration**

Throws an IOException if an error occurs.

```
public void disconnect() throws IOException;
```

**Example**

```
simAPI.disconnect();
```

### Python 3

**Declaration**

```
NavInterfaceClient.disconnect()
```

**Example**

```
simAPI.disconnect()
```

### C++

Calling `disconnect()` for the C++ interface is optional. The destructor will call `disconnect()` if you do not.
Declaration

```java
public void disconnect();
```

Example

```java
simAPI->disconnect();
```

C#

Declaration

```csharp
public void disconnect();
```

Example

```csharp
simAPI.disconnect();
```

Node.js

Declaration

```javascript
NavInterfaceClient.prototype.disconnect = function();
```

Example

```javascript
this.sim.disconnect();
```
**isConnected()**

**Purpose:** Query the status of a connection to a Simulation Server

**Returns:** A Boolean value that is True if the connection was successfully established

**Java**

**Declaration**

```java
public boolean isConnected();
```

**Return value** true if the connection is currently established, otherwise false

**Example**

```java
if (simAPI.isConnected()) {
    System.out.println("Connection is operational.");
}
```

**Python 3**

**Declaration**

```python
NavInterfaceClient.isConnected()
```

**Return value** True if the connection is currently established, otherwise False

**Example**

```python
if simAPI.isConnected():
    print("Connection is operational.")
```
### C++

**Declaration**

```
public bool isConnected() const;
```

**Return value**  
true if the connection is currently established, otherwise false

**Example**

```
if (simAPI->isConnected()) {
    cout << "Connection is operational." << std::endl;
}
```

### C#

**Declaration**

In the C# interface, Connected is a property of the object, not a method, so it does not have parentheses following it when used in an expression. See the example below.

```
public bool Connected;
```

**Value**  
true if the connection is currently established, otherwise false

**Example**

```
if (simAPI.Connected) {
    Console.WriteLine("Connection is operational.");
}
```

### Node.js

**Declaration**

```
NavInterfaceClient.prototype.isConnected = function();
```
Example

```javascript
if (this.sim.isConnected()) {
    /* Print something out however Node.js does it */
}
```
Navigation Interface Protocol

The Navigation Interface sends messages between the Navigation Client and the Simulation Server. The protocol is designed to be very simple. All messages are human-readable text. Each message is terminated by a newline character. Leading and trailing whitespace is ignored.

Message Formats

There are two message formats:

Form 1: [keyword]

This form is for messages that consist of only a keyword, and carry no other data. These messages typically request information from the Simulation Server. The response to such a query will normally be a message of Form 2. The message is terminated by a newline character.

Form 2: [keyword remainder]

This form is normally used as a response to a query, or to set a value. keyword and remainder are separated by whitespace. The remainder is the rest of the message after the keyword and whitespace following the keyword are removed. Or specifying it another way, the remainder is the sequence of characters starting from the first non-whitespace character following the keyword to the end of the message. remainder may contain whitespace.

Your code should be prepared to handle the presence of trailing whitespace or a newline character at the end of a message, in case it is not removed by the Navigation Interface API.

Message Exchange

A message exchange consists of a pair of messages. A message exchange is always initiated by the Navigation Client, by sending a message (using the Navigation Interface send() API call) to the Simulation Server. When the Simulation Server receives the message, it sends a response message which is received by the Navigation Client. Your code will accept the response message using the Navigation interface receive() API call.

In the following example, the Navigation Client sends a query to request the boat’s current speed. The Simulation Server responds with a boat speed of 5.25 meters/sec.
In the next example, the Navigation Client sends a message to change the boat’s heading to 90 degrees. The Simulation Server responds that the boat’s heading has been changed to 90 degrees.

The Navigation Client must alternately send and receive messages, using the `send()` and `receive()` methods previously described. Calling `send()` twice in a row will cause an error. Really. Just don’t do it. We can’t tell you exactly what will happen, but it will mess up your communication. Calling `receive()` twice in a row will cause the Navigation Client to get stuck in the second `receive()`, waiting for a message that will never arrive.

**Message Rate Throttling**

To prevent denial-of-service (go ahead and Google it if you’re wondering) due to excessive message traffic, the Simulation Server can limit the rate at which messages are sent and received by adding a delay between its responses. Your Navigation Client should be tolerant of a variable message rate. Your Navigation Client should still be able to send and receive several messages per second, even with throttling.
Query Messages (client → server)

These are messages sent by the Navigation Client to query the state of the sailboat or the environment.

Reference table key:

<table>
<thead>
<tr>
<th>keyword</th>
<th>Message description</th>
<th>➡ example msg sent by client</th>
<th>➡ example response from server</th>
</tr>
</thead>
</table>

These are the messages that the Navigation Client can use to query (get information from) the Simulation Server:

- **anchor**
  - Request the anchor state
  - ➡ anchor
  - ➡ anchor down

- **atDestination**
  - Request whether the boat is at the destination location
  - ➡ atDestination
  - ➡ atDestination false

- **atObstacle**
  - Request whether the boat has hit an obstacle
  - ➡ atObstacle
  - ➡ atObstacle false

- **boatHeading**
  - Request the boat direction
  - ➡ boatHeading
  - ➡ boatHeading 123.45

- **boatPosition**
  - Request the boat location
  - ➡ boatPosition
  - ➡ boatPosition 234.5 67.8

- **boatSpeed**
  - Request the boat speed
  - ➡ boatSpeed
  - ➡ boatSpeed 12.3

- **goalPosition**
  - Request the destination location
  - ➡ goalPosition
  - ➡ goalPosition 123.4 567.8

- **obstacle**
  - Request info about a specific obstacle
  - ➡ obstacle 1
  - ➡ obstacle 1,234.5,67.8,9.0

- **obstacleCount**
  - Request the number of obstacles in the simulation
  - ➡ obstacleCount
  - ➡ obstacleCount 3

- **obstacleDistance**
  - Request the nearest obstacle in the heading direction
  - ➡ obstacleDistance 45
  - ➡ obstacleDistance 1,2.34

- **obstacles**
  - Request the location and size of every obstacle
  - ➡ obstacles
  - ➡ obstacles x,y,\text{radius} ...
### obstacleScan
Request the heading and distance of every obstacle

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ obstacleScan</td>
<td>obstacleScan heading, distance...</td>
</tr>
</tbody>
</table>

### sailAngle
Request the sail angle relative to the boat

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ sailAngle</td>
<td>sailAngle 12.3</td>
</tr>
</tbody>
</table>

### time
Request the current simulation time

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ time</td>
<td>time 123.456</td>
</tr>
</tbody>
</table>

### windHeading
Request the wind direction

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ windHeading</td>
<td>windHeading 123.4</td>
</tr>
</tbody>
</table>

### windStrength
Request the wind strength (magnitude of the wind force)

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ windStrength</td>
<td>windStrength 12.3</td>
</tr>
</tbody>
</table>

---

**Modification Messages (client → server)**

A few messages allow the Navigation Client to change the state of the boat or the simulation. The Simulation Server will update the state and respond with a message containing the same keyword as the query plus the updated value (which should match the value from the modification message).

These are the messages that the Navigation Client can send to the Simulation Server to change the state of the boat or the simulation.

<table>
<thead>
<tr>
<th>Request</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ anchor</td>
<td>Set the anchor state anchor true</td>
</tr>
<tr>
<td>➤ boatHeading</td>
<td>Set the boat direction boatHeading 123.45</td>
</tr>
<tr>
<td>➤ finish</td>
<td>Close connection with simulation server finish</td>
</tr>
<tr>
<td>➤ hello</td>
<td>Establish connection with simulation server hello team1 p455wd</td>
</tr>
<tr>
<td>➤ sailAngle</td>
<td>Set the sail angle relative to the boat sailAngle 12.3</td>
</tr>
</tbody>
</table>
Response Messages (server → client)

These messages are sent by the Simulation Server to the Navigation Client in response to a message sent by the Navigation Client. The **keyword** of the response from the Simulation Server matches the **keyword** of the previous message sent by the Navigation Client, with one exception: it could be **error**. The Simulation Server could respond with **error** if something was wrong with the previous message sent by the Navigation Client. The **remainder** of the **error** message will contain text that indicates the nature of the error.

Response message examples are shown in the rightmost column of the message tables above, and are also described in the following sections.

<table>
<thead>
<tr>
<th>error</th>
<th>Respond to an unrecognized or badly formed message</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ soilAngel 12.3</td>
<td>➡️ error Unknown keyword ‘soilAngel’</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>boatSpeed</th>
<th># nav --→ sim: ask for current boat speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>boatSpeed 12.3</td>
<td># sim --→ nav: boat speed is 12.3 m/s</td>
</tr>
</tbody>
</table>

Message Details

The following sections describe each message in more detail.
Get or set the state of the boat’s anchor. Set the state to **down**, **true**, or **1** to drop the anchor or **up**, **false**, or **0** to raise the anchor. When the anchor is lowered, the boat will not move. The Simulation Server responds to this message with the (new) current state of the anchor.

Note that while anchored the boat is not moving, but simulation time is still advancing!

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>anchor</td>
<td></td>
<td>Request the current anchor state</td>
</tr>
<tr>
<td>➔ Set</td>
<td>anchor(newState)</td>
<td>boolean</td>
<td>Set the anchor state to <strong>down</strong>, <strong>true</strong>, or <strong>1</strong> to lower; or <strong>up</strong>, <strong>false</strong>, or <strong>0</strong> to raise</td>
</tr>
<tr>
<td>← Response</td>
<td>anchor(state)</td>
<td>boolean</td>
<td>Report the state of the anchor as <strong>down</strong> or <strong>up</strong></td>
</tr>
</tbody>
</table>

Example:

```
anchor # nav --> sim: ask for current anchor state
anchor false # sim --&gt; nav: anchor is currently up
anchor true # nav --&gt; sim: drop anchor
anchor down # sim --&gt; nav: anchor is now down
```
atDestination

You can use **atDestination** to determine whether your boat has reached the goal. Once it has reached the goal the simulation is completed, and your Navigation Client can disconnect from the Simulation Server. See the **finish** message.

Once your boat has reached the goal (or collided with any other obstacle), the boat will no longer move or change state and the boat’s total race time will no longer advance. The boat has collided with an obstacle when the boat’s location (defined by the coordinates of the boat’s center) intersects the obstacle.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td><strong>atDestination</strong></td>
<td></td>
<td>Request whether the boat has reached the goal</td>
</tr>
<tr>
<td>➔ Response</td>
<td><strong>atDestination</strong></td>
<td>state</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Report whether the boat has reached the goal (true or false)</td>
</tr>
</tbody>
</table>

Example:

```
 atDestination              # nav --> sim: has boat reached the goal?
atDestination false        # sim --> nav: nope!
```
\textbf{atObstacle} can be used to detect when the boat has collided with an obstacle or reached the goal (the goal is just a special obstacle).

Once your boat has collided with an obstacle (or reached the goal), the boat will no longer move or change state. The boat has collided with an obstacle when the boat’s location (defined by the coordinates of the boat’s center) intersects the obstacle.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>atObstacle</td>
<td></td>
<td>Request whether the boat has run into an obstacle</td>
</tr>
<tr>
<td>← Response</td>
<td>atObstacle state</td>
<td>boolean</td>
<td>Report whether the boat has run into an obstacle (\textbf{true} or \textbf{false})</td>
</tr>
</tbody>
</table>

Example:

\begin{verbatim}
\texttt{atObstacle} \\
\texttt{atObstacle false}
\end{verbatim}

# nav --&gt; sim: has boat hit an obstacle? \\
# sim --&gt; nav: nope!
**boatHeading**

`boatHeading` can be used to get the sailboat’s current heading, or to set the boat on a new heading. The *heading* and *newHeading* values are specified in decimal degrees, where 0° is North and 90° is East.

Changing the boat’s heading does not immediately change the boat’s direction of travel, but it instantaneously re-orients the boat to the new heading (not entirely realistic). The new orientation will change how the wind and water forces act on the boat, so its direction of travel is likely to change.

You may specify any value for the heading, but the returned value will be normalized to 0-360.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ Get</td>
<td><code>boatHeading</code></td>
<td></td>
<td>Request the current boat heading</td>
</tr>
<tr>
<td>➤ Set</td>
<td><code>boatHeading</code> newHeading float</td>
<td>Set the boat’s heading to <em>newHeading</em>, where <em>newHeading</em> is in degrees</td>
<td></td>
</tr>
<tr>
<td>➤ Response</td>
<td><code>boatHeading</code> heading float</td>
<td>Report the boat’s current <em>heading</em>, where <em>heading</em> is in degrees (0 &lt;= <em>heading</em> &lt; 360)</td>
<td></td>
</tr>
</tbody>
</table>
Example:

<table>
<thead>
<tr>
<th>boatHeading 123.456</th>
<th># nav --&gt; sim: which direction is the boat pointing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>boatHeading 270</td>
<td># sim --&gt; nav: sort of southeasterly</td>
</tr>
<tr>
<td>boatHeading 270.0</td>
<td># nav --&gt; sim: head west</td>
</tr>
<tr>
<td>boatHeading -90</td>
<td># sim --&gt; nav: west it is</td>
</tr>
<tr>
<td>boatHeading 270.0</td>
<td># sim --&gt; nav: normalized it for you!</td>
</tr>
</tbody>
</table>
The Simulation Server returns the current boat location as an XY coordinate pair.

The location is a Cartesian coordinate pair of floating point numbers separated by a comma. The units of the coordinates are meters East for \( x \), and meters North for \( y \). The coordinates are relative to the Southwest corner of the simulation area.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>boatPosition</td>
<td></td>
<td>Request the boat’s current position</td>
</tr>
<tr>
<td>➩ Response</td>
<td>boatPosition ( x \ y )</td>
<td>float float</td>
<td>Report the boat’s current position as a Cartesian coordinate pair</td>
</tr>
</tbody>
</table>

Example:

```
boatPosition  # nav --› sim:  where am I?
boatPosition 1000.0 750.0  # sim --› nav:  right in the center of the lake
```
The speed of the boat is measured in meters/second. The speed does not indicate the direction of motion.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>boatSpeed</td>
<td></td>
<td>Request the current speed of the boat</td>
</tr>
<tr>
<td>➡️ Response</td>
<td>boatSpeed</td>
<td>float</td>
<td>Report the current boat speed in meters/second</td>
</tr>
</tbody>
</table>

Example:

```
boatSpeed
boatSpeed 3.45
```

# nav --> sim: how fast are we going
# sim --> nav: 3.45 m/s
The **error** message is never sent by the Navigation Client. However, if the Navigation Client sends an invalid message to the Simulation Server, the Simulation Server may respond with an **error**. The remainder of the **error** message is a string indicating the reason for the error.

This is the *only* case where the Simulation Server message will have a keyword in its response that is *different* from the keyword in the message sent by the Navigation Client.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>← Response</td>
<td><strong>error</strong> reason</td>
<td>string</td>
<td>Report an error from the previously sent message</td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
hello there # nav --> sim: attempt
error Missing user name or password in hello message # sim --> nav: rejected!
```
The Navigation Client should send this message to close the connection with the Simulation Server. `finish` notifies the Server that the client is finished and allows it to remove the boat from the simulation.

If the Navigation Client closes the connection too soon after issuing `finish`, the response may not arrive before the connection is terminated.

No messages may be sent by the Navigation Client after `finish`.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Set</td>
<td><code>finish</code></td>
<td></td>
<td>Notify the Simulation Server that the Navigation Client is about to close the connection</td>
</tr>
<tr>
<td>← Response</td>
<td><code>finish</code></td>
<td></td>
<td>Acknowledge that the connection is closing</td>
</tr>
</tbody>
</table>

Example:

```
finish       # nav --> sim: done
finish       # sim --> nav: and done!
```
Get the location of the goal as an XY coordinate pair.

The location is a Cartesian coordinate pair of floating point numbers separated by a comma. The units of the coordinates are meters East for \( x \), and meters North for \( y \). The coordinates are relative to the Southwest corner of the simulation area.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>\texttt{goalPosition}</td>
<td></td>
<td>Request the current goal position</td>
</tr>
<tr>
<td>➖ Response</td>
<td>\texttt{goalPosition \ x \ y}</td>
<td>float float</td>
<td>Report the current goal position as a Cartesian coordinate pair</td>
</tr>
</tbody>
</table>

Example:

```
goalPosition                   # nav --> sim:  where do I want to go today?
goalPosition 1000.0 750.0      # sim --> nav:  how about here?
```
**hello** is the first message sent by a Navigation Client to establish a connection with the Simulation Server. You pass your **team_id** and **password** to the Server. The Server will respond either with a **hello** response, if your credentials were accepted, or with an **error** if they were not.

The **hello** message also enters your team’s boat into the simulation. If the simulation already had a boat entered by your team, that boat’s simulation run will be terminated and recorded. Then a new boat will be entered at the start of the course, signaling the start of a new simulation run.

Your boat could be replaced by a second **hello** if, for example, your Navigation Client crashed and you had to restart it. Or you might have a bug in your code that sends **hello** more than once during the same run. So be careful! During the competition you will only have a limited number of runs, and you won’t want to waste them.

**Important:** **hello** is automatically sent by the **connect()** API call, so you probably *do not* want to send **hello** yourself, unless you need to reset your boat for a subsequent run. You can also accomplish the same thing by calling **disconnect()**, then calling **connect()** again. However, it may be simplest to just terminate your Navigation Client and restart it.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Set</td>
<td><strong>hello</strong> team_id password</td>
<td>string string</td>
<td>Establish a connection with the Simulation Server</td>
</tr>
<tr>
<td>← Response</td>
<td><strong>hello</strong> team_id</td>
<td></td>
<td>Acknowledge that a connection has been established</td>
</tr>
</tbody>
</table>

Example:

```
hello keelhauliers badpasswd          # nav --> sim: log us in please
error Invalid username or password    # sim --> nav: try again please
hello keelhauliers n3v3r_surr3nd3r   # nav --> sim: log us in please
hello keelhauliers                    # sim --> nav: entering keelhauliers in sim
```
**obstacle** is used to retrieve information about a single specific obstacle in the simulation. The goal is also considered an obstacle, because the boat can collide with it, and it ends the simulation for that boat. Colliding with the goal is a successful end. Colliding with any other obstacle results in a not completed excursion.

Obstacles are numbered sequentially, starting with 0 being the shore, 1 being the goal, and 2 being the first obstacle to be avoided. That number is the obstacle’s *id*.

The information returned by the Simulation Server about the obstacle is: *id,x,y,r* where

- *id* is the obstacle’s ID as described above
- *x,y* are the Cartesian coordinates of the center of the obstacle
- *r* is the radius of the obstacle

The units of the coordinates are meters East for *x*, and meters North for *y*. The units of *radius* are also meters. The radius of the shore obstacle is the distance from the center of the rectangle to one of the corners.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td><strong>obstacle</strong> <em>id</em></td>
<td>integer</td>
<td>Request information about obstacle with ID <em>id</em></td>
</tr>
<tr>
<td>❯ Response</td>
<td><strong>obstacle</strong> <em>id</em>,<em>x</em>,<em>y</em>,<em>r</em></td>
<td>integer,float,float,float</td>
<td>Report information about the obstacle</td>
</tr>
</tbody>
</table>

Example:

| obstacle 0 | # nav --> sim: tell me about the shore |
| obstacle 0,1000.0,750.0,1250.0 | # sim --> nav: here are it’s center and radius |
| obstacle 1 | # nav --> sim: tell me about the goal |
| obstacle 1,300.0,400.0,20.0 | # sim --> nav: here are it’s center and radius |
| obstacle 2 | # nav --> sim: tell me about an obstacle |
| obstacle 2,750.0,345.6,50.0 | # sim --> nav: don’t hit this one! |
| obstacle 7 | # nav --> sim: tell me about another obstacle |
| error Invalid obstacle ID | # sim --> nav: there is no such obstacle! |
obstacleCount can be used to get the number of obstacles in the simulation. The shore and goal are included in the count, so the smallest obstacle count to be returned would be 2.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>obstacleCount</td>
<td></td>
<td>Request the number of obstacles in the simulation</td>
</tr>
<tr>
<td>← Response</td>
<td>obstacleCount $n$</td>
<td>integer</td>
<td>Report the number of obstacles in the simulation</td>
</tr>
</tbody>
</table>

Example:

```
obstacleCount # nav --> sim: what could the boat run into?
obstacleCount 3 # sim --> nav: shore, goal, plus one more obstacle
```
obstacleDistance can be used to request the distance from the boat to the nearest obstacle in the specified heading direction. The id and distance of the nearest thing in that direction are returned. id 0 is the shore, id 1 is the goal, and id’s greater than 1 are obstacles to be avoided. See the Virtual Laser Rangefinder section above for more information about obstacle id’s.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>obstacleDistance  heading float</td>
<td>Request the distance between the boat and a specific obstacle</td>
<td></td>
</tr>
<tr>
<td>← Response</td>
<td>obstacleDistance  id,distance</td>
<td>Report the distance between the boat and a specific obstacle</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
obstacleDistance 270 # nav --> sim: what’ll I hit if I go west?
obstacleDistance 0,920.5 # sim --> nav: the shore, in a bit less than 1 km
obstacleDistance 90  # nav --> sim: what’ll I hit if I go east?
obstacleDistance 1,103 # sim --> nav: the goal is 103 m from the boat
```
The **obstacles** message retrieves information about every obstacle in the system. The response contains a sequence of $x,y,radius$ tuples separated by spaces. Each tuple consists of three numbers separated by commas, where:

- $x,y$ are two floating point values that give the location of the obstacle center
- $radius$ is the radius of the obstacle, in meters

The **obstacleCount** message can be used to get the number of $x,y,radius$ tuples that will be returned by **obstacles**.

Remember that obstacle ID 0 is the shore and obstacle ID 1 is the goal.

See the section on Obstacles above for more information about obstacle IDs, etc.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td><strong>obstacles</strong></td>
<td></td>
<td>Request information about every obstacle in the simulation</td>
</tr>
<tr>
<td>← Response</td>
<td><strong>obstacles</strong> $x,y,r$ ...</td>
<td>float,float,float ...</td>
<td>Report the $x,y,radius$ for every obstacle in the simulation</td>
</tr>
</tbody>
</table>

Example:

```
obstacles # nav --> sim: what could I run into out there?
obstacles 1000.0,750.0,1250.0 1000.0,750.0,20.0 100.0,100.0,50.0
obstacle ID:
          0    1    2
# sim --> nav: the shore, the goal, or one other thing
```
**obstacleScan**

*obstacleScan* works like radar. It provides information (heading and distance) about each obstacle relative to the current sailboat location. It returns a sequence of *heading, distance* tuples separated by spaces. Each *heading, distance* tuple consists of two numbers separated by a comma, where:

- *heading* is the heading angle from the sailboat’s location to the obstacle center
- *distance* is the distance from the sailboat’s location to the closest point on the edge of the obstacle

The *obstacleCount* message can be used to get the number of *heading, distance* tuples that will be returned by *obstacleScan*.

The position in the list of each tuple corresponds to the obstacle ID, starting at zero. So the first *heading, distance* tuple describes obstacle ID 0: the shore; the next tuple describes obstacle ID 1: the goal; etc..

Since the shore can be reached by going in any direction, *obstacleScan* always returns a *heading* of -1 and a *distance* of 0 for the shore.

See the section on Obstacles above for more information about obstacle IDs, etc. See the section on Virtual Radar above for more information on *obstacleScan*.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td><em>obstacleScan</em></td>
<td></td>
<td>Request an obstacle scan</td>
</tr>
<tr>
<td>✅ Response</td>
<td><em>obstacleScan</em> heading, dist ...</td>
<td>list of float, float</td>
<td>Report the heading and distance to each obstacle in the simulation</td>
</tr>
</tbody>
</table>

Example:

```
obstacleScan
obstacleScan -1,0 45,999.9 45,100

obstacle ID: 0 1 2
```

# nav --> sim: where are things?
# sim --> nav: if you go straight for
# the goal, you won’t
# make it!
Sail angle is in degrees. The sail angle is relative to the boat. Changing the boat's heading (with `boatHeading`) does not change the relative sail angle. However, it will change the sail's angle to the wind. The sail angle will be clamped to the range $-90^\circ \leq \text{angle} \leq 90^\circ$.

Also, the `new_angle` value is an absolute value (relative to the boat), not a displacement from its previous position. So sending `sailAngle 30` will always result in the sail at $30^\circ$ relative to the boat, regardless of its previous position.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td><code>sailAngle</code></td>
<td></td>
<td>Request the boat’s current sail angle</td>
</tr>
<tr>
<td>➔ Set</td>
<td><code>sailAngle new_angle</code></td>
<td>float</td>
<td>Set the boat’s sail angle</td>
</tr>
<tr>
<td>← Response</td>
<td><code>sailAngle angle</code></td>
<td>float</td>
<td>Report the boat’s current sail angle</td>
</tr>
</tbody>
</table>

$sailAngle 0$  $sailAngle 45$  $sailAngle -45$

$sailAngle 0$  $sailAngle 45$  $sailAngle -45$
Example:

<table>
<thead>
<tr>
<th>sailAngle</th>
<th># nav --&gt; sim: where is my sail?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td># sim --&gt; nav: amidships</td>
</tr>
<tr>
<td>-45</td>
<td># nav --&gt; sim: rotate it counterclockwise (looking down)</td>
</tr>
<tr>
<td>-45.0</td>
<td># sim --&gt; nav: aye, skipper!</td>
</tr>
<tr>
<td>135</td>
<td># nav --&gt; sim: rotate it a lot the other way</td>
</tr>
<tr>
<td>90.0</td>
<td># sim --&gt; nav: sorry, skipper! it only goes this far.</td>
</tr>
</tbody>
</table>
time reports the elapsed time since the start of the simulation. This is the time in seconds since the simulation began, not how long your boat has been sailing. If you want to keep track of how long your sailboat has been sailing, you will want to save the time when your boat starts (which happens when your Navigation Client sends hello), for use in duration calculations.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>time</td>
<td></td>
<td>Request the elapsed time in the simulation</td>
</tr>
<tr>
<td>➩ Response</td>
<td>time sec</td>
<td>float</td>
<td>Report the elapsed time in seconds</td>
</tr>
</tbody>
</table>

Example:

```plaintext
# nav --> sim:  what time is it?
time

# sim --> nav:  this many seconds since the dawn of simulated time

# nav --> sim:  what time is it now?
time 2789.345

# sim --> nav:  1.1 seconds have elapsed since you last asked me
```
**windHeading** requests the direction of the wind. The *heading* value that is returned is in degrees, where 0° is North and 90° is East.

The wind heading indicates the direction of movement of the air. So, for example, if the wind heading is 0, the air is moving from South to North. If the wind heading is 270, the air is moving from East to West. The wind heading remains constant throughout the duration of the simulation run, but it may be different in a subsequent run. The wind heading is also the same at every location. Therefore, you should only need to request the value once per simulation.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ Get</td>
<td>windHeading</td>
<td></td>
<td>Request the current wind heading</td>
</tr>
<tr>
<td>← Response</td>
<td>windHeading</td>
<td>heading</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Report the current wind heading</td>
</tr>
</tbody>
</table>

Example:

```
windHeading 0  # nav --> sim: tell me the wind direction
windHeading 315 # sim --> nav: any way the wind blows,
windHeading 180.0  # doesn’t really matter,
# to me, to me...
# but it might matter to you, so it’s
# a Northerly wind (blowing from N to S)
```

61
windStrength requests the strength of the wind, which is the magnitude of the wind force vector. The direction of the wind force vector is given by windHeading. The Simulation Server reports the wind strength in newtons.

The wind strength remains constant throughout the duration of a simulation run, but it may be different in a subsequent run. The wind strength is also the same at every location. Therefore, you should only need to request the value once per simulation.

<table>
<thead>
<tr>
<th>Send/Rcv</th>
<th>Msg Format</th>
<th>Arg Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Get</td>
<td>windStrength</td>
<td></td>
<td>Request the current wind strength</td>
</tr>
<tr>
<td>↔ Response</td>
<td>windStrength strength</td>
<td>float</td>
<td>Report the current wind strength</td>
</tr>
</tbody>
</table>

Example:

```
windStrength 10.0  # nav --&gt; sim: tell me the wind strength
windStrength 10.0  # sim --&gt; nav: it’s a steady 10 newtons
```
Example

The following example shows several message exchanges from the start of a typical simulation run:

<table>
<thead>
<tr>
<th>Message</th>
<th>Action/Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello team1 p455wd</td>
<td># nav --&gt; sim: establish connection</td>
</tr>
<tr>
<td>hello team1</td>
<td># sim --&gt; nav: connection established</td>
</tr>
<tr>
<td></td>
<td># credentials validated</td>
</tr>
<tr>
<td></td>
<td># boat “team1” added</td>
</tr>
<tr>
<td>boatHeading</td>
<td># nav --&gt; sim: ask for current boat heading</td>
</tr>
<tr>
<td>boatHeading 45.0</td>
<td># sim --&gt; nav: boat is heading northeast</td>
</tr>
<tr>
<td>boatHeading 180.0</td>
<td># nav --&gt; sim: change heading to south</td>
</tr>
<tr>
<td>boatHeading 180.0</td>
<td># sim --&gt; nav: boat is heading south</td>
</tr>
</tbody>
</table>

Now the rest is up to you!

Happy Sailing!

The 2017 HSDC SailSim Team