How to use the \texttt{waypoint} module

Lindsey Kuper

CSCI P545 Embedded and Real-Time Systems, Fall 2009

\section*{Introduction}

This document explains the use of the \texttt{waypoint} Python module for programming ERTS\textsuperscript{1} lab experiments. Rather than working with the bare data provided in lab descriptions, the module provides handy abstractions for waypoints and obstacles (in the form of \texttt{Waypoint}, \texttt{WaypointList}, \texttt{Obstacle}, and \texttt{ObstacleList} objects) so that programmers can work with waypoints and obstacles at a slightly higher level and write more concise and robust driver code.

\section{Waypoint and obstacle data}

\subsection{Waypoint data}

Waypoint data for ERTS lab experiments begins life as a list. As an example, we show the list of nine waypoints that is provided in the descriptions of Labs 2, 3, and 4.

\begin{verbatim}
waypoints = [[1, 39.181956, -86.522103, 4.0, 5.0],
 [2, 39.182119, -86.521690, 4.0, 5.0],
 [3, 39.182119, -86.522102, 4.0, 5.0],
 [4, 39.182235, -86.522292, 4.0, 5.0],
 [5, 39.182169, -86.522420, 4.0, 5.0],
 [6, 39.182070, -86.522292, 4.0, 5.0],
 [7, 39.181956, -86.522421, 4.0, 5.0],
 [8, 39.181907, -86.522294, 4.0, 5.0]]
\end{verbatim}

The data is provided as a Python list in (a rough approximation of) RDDE\textsuperscript{2} format. The list has an element for each waypoint, each of which is itself a list with five elements:

\begin{itemize}
  \item Waypoint number, a positive integer.
  \item Waypoint latitude, a float with up to 7 decimal places.
\end{itemize}

\footnote{\textsuperscript{1}See \url{http://www.cs.indiana.edu/classes/p545-sjoh/Year0fERTS.htm} for the gory details (or don’t).}

\footnote{\textsuperscript{2}See \url{http://www.darpa.mil/grandchallenge05/RDDF_Document.pdf} for the gory details (or don’t).}
• Waypoint longitude, a float with up to 7 decimal places.
• Lateral boundary offset (LBO), or radius of the waypoint, given in meters.
• Course speed, the suggested speed with which to pass through the waypoint, given in meters per second.

These points, like all points in the northern hemisphere and western hemisphere, have a positive latitude and a negative longitude.

1.2 Obstacle data

The obstacle list provided in the description of Lab 4 is as follows:

```python
obstacle_list = 
    ["cone1", (39.181933, -86.521931), 0.25], \
    ["cone2", (39.182036, -86.521747), 0.25], \
    ["cone3", (39.182033, -86.521654), 0.25], \
    ["cone4", (39.182119, -86.522024), 0.25], \
    ["cone5", (39.182154, -86.522190), 0.25], \
    ["cone6", (39.182172, -86.522218), 0.25], \
    ["cone7", (39.182015, -86.522356), 0.25], \
    ["cone8", (39.181957, -86.522167), 0.25], \
    ["cone9", (39.181909, -86.522167), 0.25]
```

Here, each obstacle is represented by a three-element list containing a description string, a tuple containing the latitude and longitude of the obstacle, and the LBO of the obstacle, given in meters. The format of this list is the same as that of the obstacle list provided by the synlaser CartFS component.

As you can see, the format of the obstacle data is just different enough from that of waypoint data to be “interesting”. The waypoint module tries to ameliorate this issue by providing a higher-level abstraction for dealing with both waypoints and obstacles, so that the programmer does not have to be concerned with low-level details.

2 A quick tour of the waypoint module

The waypoint module provides four classes we can use to help streamline the parts of the code that deal with waypoints and obstacles: Waypoint, WaypointList, Obstacle, and ObstacleList. To use the module, we just save `waypoint.py` in the directory in which we are working and add a `from ... import` statement near the top of our code:

```python
from waypoint import *
```

---

3If you’re unfamiliar with modules, you might want to have a quick look at [http://docs.python.org/tutorial/modules.html](http://docs.python.org/tutorial/modules.html).

4Or any place where Python can find it – see [http://docs.python.org/tutorial/modules.html#the-module-search-path](http://docs.python.org/tutorial/modules.html#the-module-search-path).
We can now use any of the methods defined in the `WaypointList` class on `waypoints`. Particularly useful is the `next()` method. The `WaypointList` object keeps track of whichever waypoint we’re concerned with at any given time (usually, the one we’re currently heading toward). For a newly initialized `WaypointList`, `next()` returns a `Waypoint` object representing the first item in the list:

```python
>>> from waypoint import WaypointList
>>> waypoints = [[1,39.181956,-86.522103,4.0,5.0],
... [3,39.182119,-86.521690,4.0,5.0],
... [4,39.182119,-86.522102,4.0,5.0],
... [5,39.182235,-86.522292,4.0,5.0],
... [6,39.182169,-86.522420,4.0,5.0],
... [7,39.182070,-86.522292,4.0,5.0],
... [8,39.181956,-86.522421,4.0,5.0],
... [9,39.181907,-86.522294,4.0,5.0]]
>>> wl = WaypointList(waypoints)
>>> wl.next()
<waypoint.Waypoint instance at 0x96a148>
```

We can use `print` to get a human-readable representation of an individual waypoint or of the whole list:

```python
>>> print wl.next()
1, 39.181956, -86.522103, 4.0, 5.0

>>> print wl
1, 39.181956, -86.522103, 4.0, 5.0
3, 39.182119, -86.521690, 4.0, 5.0
4, 39.182119, -86.522102, 4.0, 5.0
5, 39.182235, -86.522292, 4.0, 5.0
6, 39.182169, -86.522420, 4.0, 5.0
7, 39.182070, -86.522292, 4.0, 5.0
8, 39.181956, -86.522421, 4.0, 5.0
9, 39.181907, -86.522294, 4.0, 5.0
```

The `Waypoint` object returned by `next()` has `description`, `lat`, `lon`, `lbo`, and `speed` attributes that we may access:

```python
>>> wl.next().description
'1'
>>> wl.next().lat
39.181956
>>> wl.next().lon
-86.522103000000001
>>> wl.next().lbo
4.0
>>> wl.next().speed
5.0
```

And we can call `inc_next()` and `dec_next()` to navigate within the list:
If we are already at the first or last waypoint, \texttt{inc\_next()} and \texttt{dec\_next()} behave themselves and “roll over” to the end or beginning of the list, respectively:

\begin{verbatim}
>>> print wl.next()
1, 39.181956, -86.522103, 4.0, 5.0
>>> wl.dec\_next()
>>> print wl.next()
9, 39.181907, -86.522294, 4.0, 5.0
>>> wl.inc\_next()
>>> print wl.next()
1, 39.181956, -86.522103, 4.0, 5.0
\end{verbatim}

The methods shown here are just a few; we can see a complete list of methods available on an instance of \texttt{WaypointList} by calling \texttt{help()} on the instance name.

### 3 A simple example: using the \texttt{waypoint} module in Lab 2 driver code

Suppose we are writing a \texttt{Driver} class for the Lab 2 assignment, in which we navigate a sequence of GPS waypoints. (Assume our code is in a file called \texttt{driver.py}.) An outline of \texttt{Driver} might look something like this:

```python
class Driver():
    def __init__(self):
        """Initializes a new Driver object.""
        ...
    def update(self, lat, lon, head, dist):
        """Returns a tuple containing the updated throttle percentage and the inverse turn radius.""
        ...
```

To start using the \texttt{waypoint} module, we can add \texttt{from waypoint import *} at the top of \texttt{driver.py}. Of course, we can also choose to only import the classes we
need. For this lab, we will not be working with obstacles, or even individual waypoints, so we need only import `WaypointList`:

```python
from waypoint import WaypointList
```

Now we’re ready to create instances of `WaypointList` using the `WaypointList()` constructor. Our `Driver` class starts with an `__init__()` method that, among other things, creates a class variable, `self.waypoints`, that contains a `WaypointList` object initialized from the bare `waypoints` list we saw in section 1.1. (We’ll assume that the original, bare `waypoints` list is defined elsewhere in our code, possibly in a separate file.) `__init__()` looks something like this:

```python
def __init__(self):
    """Initializes a new Driver object.""
    # other class variables ...
    self.waypoints = WaypointList(waypoints)
```

The `next()` and `inc_next()` methods shown in section 2, together with the `lat` and `lon` attributes of `Waypoint` objects, are all the waypoint abstractions we need to write a complete Lab 2 driver. An outline of such a `Driver` class could be written as follows.

```python
class Driver():
    def __init__(self):
        """Initializes a new Driver object.""
        # ... other class variables ...
        self.waypoints = WaypointList(waypoints)

    def update(self, lat, lon, head, dist):
        """Returns a tuple containing the updated throttle percentage and the inverse turn radius.""
        percent_throttle = 50.0

        waypoint_latlon = (self.waypoints.next().lat, self.waypoints.next().lon)

        # ... code to find distance and angle to next waypoint ...

        # If we’re within 3 meters of the next waypoint, move on.
        if (meters_to_target < 3):
            self.waypoints.inc_next()

        # ... code to set desired heading and calculate turn radius ...

        return (percent_throttle, turn_rad)
```
### 4 Using the waypoint module to manage obstacles

When we write a driver to avoid obstacles, as in Labs 3 and 4, we could be receiving obstacle data from a CartFS **synlaser** component, or we could simply have a static list of obstacles to avoid (for debugging, for instance). In either case, let’s assume that our approach to obstacle avoidance is to create synthetic waypoints and add them to our waypoint list on the fly.

For this example, suppose we are working on Lab 4, and we wish to write a **Driver** class that avoids the static list of obstacles given in **obstacle_list** in section 1.2. (As before, we’ll assume that **obstacle_list** is already defined.) In addition to our usual **__init__()** and **update()** methods, we’ll need a method that is capable of inserting a synthetic waypoint into the waypoint list, so let’s outline our **Driver** class as follows:

```python
class Driver():
    def __init__(self):
        """Initializes a new Driver object."""
        ...

    def update(self, lat, lon, head, dist, obstacle_list):
        """Returns a tuple containing the updated throttle percentage and the inverse turn radius."""
        ...

    def insert(self, lat, lon, meters_to_obstacle, angle_to_target):
        """Creates a synthetic waypoint and adds it to the waypoint list in the appropriate place so that the next obstacle can be avoided."""
        ...
```

We’ll need to import the **Waypoint** and **ObstacleList** classes from **waypoint**, in addition to the **WaypointList** class we’ve already been using:

```python
from waypoint import Waypoint, ObstacleList, WaypointList
```

And now we’re ready to define **__init__()**:

```python
class Driver():
    def __init__(self):
        """Initializes a new Driver object."""
        # ...other class variables ...

        self.waypoints = WaypointList(waypoints)
        self.obstacle_list = ObstacleList(obstacle_list)
```

The only change that we’ve made from the **__init__()** shown in section 3 is that we’ve created a second class variable, **self.obstacle_list**, using the
ObstacleList() constructor. (Here we see the benefit of the abstractions that the waypoin\nt module provides: the code to create self.waypoints and self.obstacle_list looks the same, despite the different original formats.)

We can now use any of the methods defined in the WaypointList and ObstacleList classes on waypoints and obstacle_list, respectively. Again, all we need to write update() are next() and inc_next(). We can outline a simple version of update() as follows:

```python
def update(self, lat, lon, head, dist):
    """Returns a tuple containing the updated throttle percentage and the inverse turn radius."""

    percent_throttle = 60.0

    next_waypoint = self.waypoints.next()
    waypoint_latlon = (next_waypoint.lat, next_waypoint.lon)

    next_obstacle = self.obstacle_list.next()
    obstacle_latlon = (next_obstacle.lat, next_obstacle.lon)

    # ... code to find distance and angle to next waypoint and next obstacle ...

    if (# ... code to determine if an obstacle is present ...):
        self.insert(lat, lon, meters_to_obstacle, angle_to_target)

    if (# ... code to determine if we are finished with the current obstacle ...):
        self.obstacle_list.inc_next()

    # If we're within 3 meters of the next waypoint, move on.
    if (meters_to_target < 3):
        self.waypoints.inc_next()

    # ... code to set desired heading and calculate turn radius ...
    return (percent_throttle, turn_rad)
```

Again, our code has changed little from the version shown in section 3. All that remains for us to do is to define the insert() method that is called by update().

The insert() method will be responsible for creating synthetic waypoints and inserting them into the waypoint list, so we use the Waypoint() constructor to create a new, single Waypoint object from a list of data. Then, we use the insert_waypoint() method defined on WaypointList instances. insert_waypoint() takes a Waypoint object and an index position and inserts the object into the WaypointList instance, making adjustments if necessary so that next() will work correctly. In this case, since we want to add the new waypoint just before the
current “next” waypoint, we simply use the `next_waypoint` attribute of `waypoints` as the index value to pass to `insert_waypoint()`. Here is an outline of `insert()` that builds a list of waypoint data, creates a new `Waypoint` object, and inserts it into `waypoints`. It finishes up by adjusting our “next” pointers with a pair of calls to `dec_next()` and `inc_next()`, both of which are defined on `ObstacleList` instances just as they are on `WaypointList` instances.

```python
def insert(self, lat, lon, meters_to_obstacle, angle_to_target):
    """Creates a synthetic waypoint and adds it to the waypoint list in the appropriate place so that the next obstacle can be avoided.""

    # ... code to calculate swp_lat and swp_lon, the latitude and longitude of the synthetic waypoint ...

    # Create the synthetic waypoint using the latitude and longitude of the point, the LBO of the obstacle, and the speed 5.0.
    next_obstacle = self.obstacle_list.next()
    lbo = next_obstacle.lbo
    speed = 5.0
    swp_data = ["ignored", swp_lat, swp_lon, lbo, speed]
    swp = Waypoint(swp_data)

    self.waypoints.insert_waypoint(swp, self.waypoints.next_waypoint)

    # By the time we're adding the synthetic waypoint, the next_waypoint counter is on the waypoint *after* it, so we have to decrement the next_waypoint field so that it points to the synthetic waypoint rather than the following one.
    self.waypoints.dec_next()

    # Increment the next_obstacle field.
    self.obstacle_list.inc_next()
```