A Python mapping package

New Mexico Tech
Computer Center www.nmt.edu/tcc

John W. Shipman

2006-11-01 15:34

Table of Contents

1. Introduction

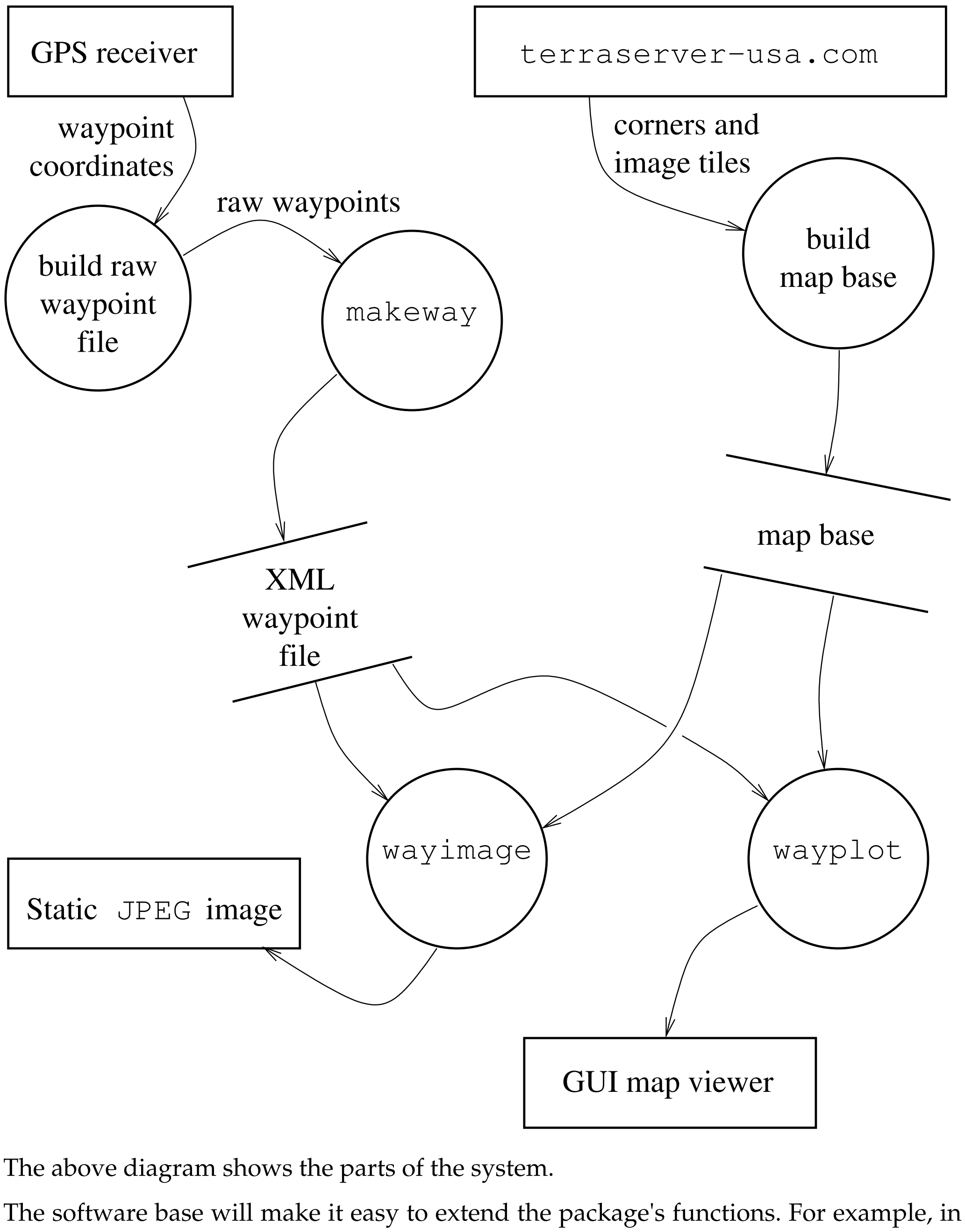
Thanks to a number of fine public domain sources, it is possible to display data on base maps with free software. The package described here is for displaying GPS waypoints on a base map, but the techniques can be generalized to other applications.

Open-source and public domain resources used in this project include:

- The `terraserver-usa.com` web site makes this entire project possible. A flagship public domain resource supported by Microsoft, this site provides both topographic maps and aerial photos of the entire continental United States.
- Files using XML (eXtended Markup Language) structures take care of storing a number of types of data required by the system.
- The Python programming language is the glue that holds everything together. In addition to the base language, a number of open-source Python library modules allow use of database systems, graphical user interfaces (GUIs), interaction with data stored in XML files, and manipulation of images.

Here is the general workflow for the system:

- Take the GPS into the field and record the coordinates of points of interest.
- Prepare a data file of those coordinates and their descriptions.
- Prepare base maps by downloading map images from `terraserver-usa.com`.
- Display the GPS waypoints on the base maps.



The above diagram shows the parts of the system.

The software base will make it easy to extend the package's functions. For example, in the future, we may add support for public-domain files describing elevations. This would allow derivation of the elevation gain for a given hiking route.

1.1. How to get this publication

This publication is available in Web form [1] and also as a PDF document [2]. Please forward any comments to **`tcc-doc@nmt.edu`**.

2. Definitions

These terms are used throughout:

2.1. Kinds of tiles

Two kinds of image data are available from TerraServer:

- *Digital Orthoquad* or *DOQ* tiles represent aerial photos. The *tile-kind* or image type code for such tiles is 1.
- *Topographic map tiles* are scanned from topographic maps, such as USGS quadrangle maps. The tile-kind for these tiles is 2.

2.2. Magnification codes

TerraServer's map data is available at a number of different magnifications. Each pixel of the map covers a square whose edges may be 1, 2, 4, 8, up to 512 meters.

When we refer to a *magnification code* or *mag-code*, we mean this dimension. For example, the aerial photographs are available at magnifications up to 1 meter/pixel, a mag-code of 1. Topographic map images are easiest to read at a mag-code of 4, so that each pixel of the image represents a 4×4 meter square of ground.

2.3. Tiles

All the imagery available from the TerraServer site comes in the form of *tiles*, image files containing 200×200 pixels. For example, a tile with a mag-code of 4 will represent an 800×800-meter area.

The programs of this suite will take care of assembling the tiles into images. The size of these images will be a multiple of the tile size.

2.4. Flexible angle coordinates

You can use a variety of input formats to specify coordinates. To save typing, we use these conventions:

- Numbers with 1, 2 or 3 digits are assumed to be in degrees. For example, "**`34`**" is assumed to be 34°, and "**`107`**" means 107°.
- Numbers with 4 or 5 digits are assumed to be in the format *`DDMM`* or *`DDDMM`*, where *`DD`* or *`DDD`* is degrees and *`MM`* is minutes. So, for example, **`3456`** means 34° 56', and **`10703`** means 107° 3'.
- Numbers with 6 or 7 digits are assumed to have be in format *`DDMMSS`* or *`DDDMMSS`*, where *`SS`* is seconds. Examples: **`341638`** means 34° 16' 38", and **`1075301`** means 107° 53' 1".
- A trailing decimal point and fraction are always allowed. The fraction is assumed to be in the same units as the smallest unit to the left of the decimal point. For example, "**`3401.57`**" is interpreted as 34° 01.57'.

[1] http://www.nmt.edu/tcc/help/lang/python/mapping/doc/
[2] http://www.nmt.edu/tcc/help/lang/python/mapping/doc/py-mapping.pdf

2.5. Latitude and longitude coordinates

The usual format for representing a location as a latitude and longitude is:

```
  lat{n|s}lon{e|w}
```

That is, start with the latitude using the flexible angle format described above, followed by **`n`** or **`s`** for north or south latitude, followed by the longitude in flexible angle format, then **`e`** or **`w`** for east or west longitude.

Here are some examples:

`5130s0007e`	51°30' S. Lat., 0°7' E. Long.
`34n107w`	34° N. Lat., 107° W. Long.
`34.0242n107.1811w`	34.0242° N. Lat., 107.1811° W. Lat.

3. Preparation of GPS waypoint files

The primary immediate application of this system is to display, superimposed on a base map, waypoints (locations with descriptions) collected with a Global Positioning System (GPS) receiver.

Accordingly, the first step is to record GPS waypoints and enter these data into a file. Such files will be input to the steps described in later sections.

Waypoints are organized into *transects* that record the order in which the locations were visited. This is important to give meanings to terms like “left” and “ahead”, which refer to the direction of travel with the GPS. Each transect has a name that can be used for loading it into the display applications.

3.1. Fieldwork

When taking GPS waypoints in the field, you will need to record:

- The coordinates of the waypoint. Currently, the program supports latitude and longitude coordinates. You can use any of these formats: degrees with decimal (e.g., 33.98253°), whole degrees and minutes with decimal (e.g., 33° 58.95'), or degrees, minutes, and seconds with optional decimal (e.g., 33° 58' 57" or 33° 58' 57.1"). Support for UTM (Universal Transverse Mercator) coordinates may be added in the future.
- A brief description of what is at that waypoint. This isn't mandatory, but in most cases you'll want to provide it so that users can request the description of a point displayed on a map and get back something like “End of Forest Road 505” or “Abandoned water well for the Rienhardt Ranch.”
- You may wish to record elevations as well. GPS units do not provide very good elevations (50 feet is a typical error), but they may be sufficient for your purposes. More accurate values from barometric or other measurements may be available, and elevations interpolated from topographic map contours can always be added to the data files later.

3.2. Creating the raw waypoint file

A simple text editor suffices for creating the raw file of waypoints. The format of this file is optimized for rapid entry. A later step (see Preparing the XML waypoint file (p. 5) below) checks this file for validity and transforms it to the XML-based format that is used by later processing steps.

Here is an fragment of a waypoint file:

```
!--
! NM: Magdalena Mts: Copper Canyon Road
!--
340127n 1070759w  Magdalenas: Copper Cyn x N. Fork Rds
....25n ...0804w  Copper Cyn Rd: bends L across streambed ~6980'
....06n .....25w  Sign `Copper Cyn Tr. No. 10', fork: L=mines, R=trail
....04n .....27w  Copper Cyn Rd: gate to private inholding, ~7120'
```

The first three lines are comments. Lines starting with a bang (!) character are ignored.

Each of the remaining lines specifies a waypoint. The fourth line shown above gives the location as 34° 01' 27" N. Lat, 107° 07' 59" W. Long. You can use a number of different formats to enter coordinates; see the section above on flexible angle format.

The remainder of the line is the description.

Because the degrees don't change through this set, the program allows you to use period (.) characters at the beginning of a coordinate value to mean “copy the value from the corresponding position on the previous line.” So the first four characters of “**`....25n`**” and the first three characters of “**`...0804`**” are copied from the previous values, giving effective coordinates of “**`340125n 1070804`**”. Similarly, the next line is the equivalent of “**`340106n 1070825w`**”.

3.3. Preparing the XML waypoint file

A script called `makeway` is used to translate the raw waypoint file, described in the previous section, to an XML-based form. The XML format has the advantage that it is easier for programs to read and use. It also allows you to use an XML file editor (such as the XML editing mode in *emacs*) to maintain and enhance the waypoint data after its initial entry.

To run this program:

```
    makeway <raw-file >out-file.xml
```

where ***`raw-file`*** is the name of the raw waypoint file created as described above, and ***`out-file`*****`.xml`** is the file to be created.

Once the file is built, you will need to edit the XML file in order to break the contents up into “routes” by enclosing sets of `<pt>` elements within a `<route name="`*`route-name`*` id="`*`route-id`*`">`...`<//route>` element. Later steps in this processing chain (described below) will allow you to display whole waypoint files on a base map, and the route name and ID elements will allow you to select only certain routes for display.

To continue the example in the preceding section, here is the start of the XML that might result from the input example, with a `<route>` element added by hand, naming these points as part of the Copper Canyon route (ID **`copper`**):

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE waypoint-set SYSTEM "waypoints.dtd">
<!-- $Revision: 1.27 $  $Date: 2006/04/26 18:47:55 $
 !-->
<waypoint-set>
  <route name="Magdalenas: Copper Canyon" id="copper">
    <pt latd='34' latm='01' lats='27' ns='n'
```

```
        lond='107' lonm='07' lons='59' ew='w' mapdatum='WGS-84'
    >Magdalenas: Copper Cyn x N. Fork Rds</pt>
    <pt latd='34' latm='01' lats='25' ns='n'
        lond='107' lonm='08' lons='04' ew='w' mapdatum='WGS-84'
    >Copper Cyn Rd: bends L across streambed ~6980'</pt>
    ...
  </route>
</waypoint-set>
```

4. Building the map base

Once you have entered your waypoints, the next step is to build up local copies of the TerraServer's map data for the area of interest.

First, designate or create a specific directory as the root of the *map base*. By default, this is directory `tiles/` in the current directory. You will also need to create a subdirectory, under the map base root directory, named `tiles-`*`n`*, for each magnification you will need, where ***`n`*** is the magnification code (p. 3).

For each area, you will need to find the geographic coordinates that delimit it. In practice, an area is defined by minimum and maximum latitude and longitude values, which define a rectangle in the usual Mercator projection.

Once you have the coordinates of the target area, you will then run the `buildmapbase` script to download local copies of the relevant map data.

4.1. The `routebox` script

To find the geographic limits of one route or all routes in a given waypoint file, use the `routebox` script:

```

    routebox wayfile [route-id]
```

where ***`wayfile`*** is the name of the waypoint file, and ***`route-id`*** is the optional route name. The program will print a line like this:

```
34.02417n107.18114w 34.05033n107.13056w
```

4.2. The `buildmapbase` script

Then, assuming you have an Internet connection, the `buildmapbase` script will add the necessary data to your map base root directory:

```

    buildmapbase [option ...] c1 c2
```

where the options include:

`-m` ***`mag`***
Specifies the magnification code (p. 3) of the desired map base. Good first guesses for this argument are 1 for photoquad tiles and 4 for topo map tiles. The default value is 4.

`-k` ***`kind`***
specifies a kind of tile, 1 for photoquad tiles, 2 for topo map tiles. The default is 2.

`-b` ***`map-base`***
Specifies the directory containing the map base (see building the map base, above). The default is subdirectory `tiles/` in the current directory.

`-x` ***`margin`***
Specifies an extra margin to be shown around the rectangle defined by the waypoints. The units of ***`margin`*** are in tiles. For example, if you are using topo maps at the recommended 4 meters/pixel scale, the default margin is half a tile, or 400 meters; if you set the option **`-x 2.0`**, you would get a margin of 1600 meters.

Two positional arguments are required:

`c1`
Specifies one corner of the area to be covered. as a The notation is like that used in the raw waypoints file, except that the latitude and longitude are concatenated into a single item. For example, **`3404n10654w`** means 34°4' N. Lat., 106°54' W. Long.

`c2`
Specifies the opposite corner of the area to be covered, in the same way.

So, if the output of the `routebox` script looks like this:

```
34.02417n107.18114w 34.05033n107.13056w
```

To load up a map base in directory `mtns` for this area with 4-meter/pixel topo maps, insure that there is a subdirectory `mtns/tiles-4`, and then use this command:

```
buildmapbase -b mtns -m 4 -k 2 34.02417n107.18114w 34.05033n107.13056w
```

5. The `wayplot` script: GUI display of waypoints on a base map

Once you have built up a map base for a given area, and created a set of GPS waypoints in that area, use the `wayplot` script to display the waypoints on the base map in a graphical user interface.

To run this program, use this command line:

```
wayplot [options ...] wayfile [route-id]...
```

where the options include:

`-m` ***`mag`***
specifies a magnification as a magnification code; the default is 4.

`-k` ***`kind`***
specifies a kind of tile, 1 for photoquad tiles, 2 for topo map tiles. The default is 2.

`-b` ***`map-base`***
Specifies the directory containing the map base (see building the map base, above). The default is subdirectory `tiles` in the current directory.

`-x` ***`margin`***
Specifies an extra margin to be shown around the rectangle defined by the waypoints. The units of ***`margin`*** are in tiles, and the default value is 0.5, that is, half a tile. For example, if you are using

photo tiles maps at 1 meters/pixel scale, a tile is 200 meters on a side, and the default margin is 100 meters. If you set the option **`-x 2.0`**, you would get a margin of 400 meters.

Positional arguments to the **`wayplot`** script:

`wayfile`
Specifies the waypoint file to be read. If no route in this file is selected by name or ID, all the waypoints in the file are displayed.

`route-id`
Selects one `<route>` element in the waypoint file, by specifying its **`id`** attribute. Any number of route IDs may be specified; if none are given, all the routes in the waypoint file are displayed.

You must be in an environment where you can create new windows. The script will bring up a window showing the base map in a large frame, with assorted controls (described below). You can drag the map around in the big frame by dragging with the middle mouse button.

The waypoints will be shown as circled numbers. Clicking mouse button 1 on a waypoint will display the descriptive text on that location from the waypoint file.

Other controls and indicators include:

- An indicator that reads out the latitude and longitude of the current mouse position. If there are missing corners in the corner file at that location, such that some or all of the four corners of the tile containing the mouse are not defined, this indicator will be blank. Another indicator displays the X and Y coordinates of the cursor relative to the image.
- An indicator that shows the text description of the last waypoint clicked on using mouse button 1, and its latitude and longitude.
- A Quit button.

6. The `wayimage` script: Plotting waypoints on an image

To display a set of waypoints statically on an image, you can use the `wayimage` script to produce a JPEG file.

To run `wayimage`:

```
wayimage [option] ... wayfile route-id ...
```

where the options include:

`-m` ***`mag-code`***
specifies a magnification code; default is 4 meters/pixel.

`-k` ***`tile-kind`***
specifies a kind of tile, 1 for photoquad tiles, 2 for topo map tiles. The default is 2.

`-b` ***`map-base`***
Specifies the directory containing the map base (see building the map base, above). The default is subdirectory `tiles` in the current directory.

`-x` ***`margin`***
Specifies an extra margin to be shown around the rectangle defined by the waypoints. The units of ***`margin`*** are in tiles. For example, if you are using topo maps at the recommended 4 meters/pixel scale, each tile is 800×800 meters. The default margin is half a tile, or 400 meters; if you set the option **`-x 2.0`**, you'll get a margin of 1600 meters.

-o *image-out*

Specifies the name of the output image file to be written. The default is `wayimage.jpg`; most images will be written in JPEG format, but topo images for mag-codes 2, 8, and 32 will be written as GIF images.

Positional arguments to the **`wayimage`** script:

wayfile

Specifies the waypoint file to be read. If no route in this file is selected by name or ID, all the waypoints in the file are displayed.

route-id

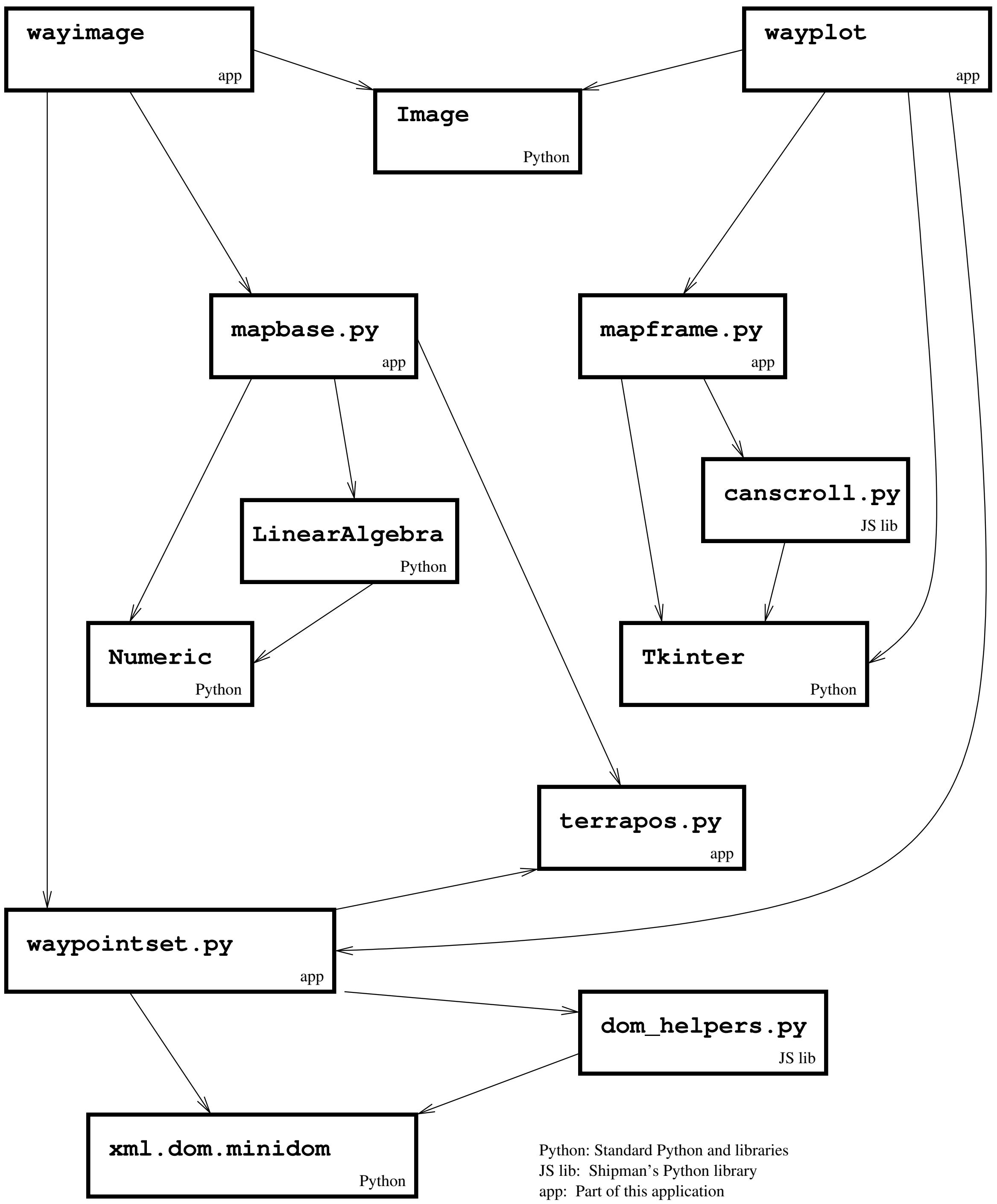
Selects one `<route>` element in the waypoint file, by specifying its **`id`** attribute. Any number of route IDs may be specified; if none are given, all the routes in the waypoint file are displayed.

Points are plotted on the base image as white squares containing the waypoint numbers in red, with one black pixel showing the exact location of the waypoint.

A report showing all waypoints is also written to the standard output stream. Points outside the known map tiles, if any, will be shown with the prefix "Outside:". The plotted points are listed with their waypoint numbers so you can relate them to the image produced.

7. Internals

This section describes some of the inner workings of the coordinate transforms used in the map objects. Here is a diagram showing the modules and their sources:



7.1. The `terrapos.py` module

This Python module provides a number of functions and classes for representing and manipulating terrestrial coordinates. Functions include:

`degRad(deg)`
Given an angle in degrees, this function returns the corresponding angle in radians.

`radDeg(rad)`
Given an angle in radians, this function returns that angle as degrees.

`feetToAngle(feet)`
Given a distance on the earth's surface in feet, this function returns the angle subtended by that distance from the center of the earth, in radians.

`angleToFeet(angle)`
The inverse function of **`feetToAngle`**.

`degDMS(deg)`
Given an angle in degrees, this function returns the equivalent angle as a tuple **`(d,m,s)`** where **`d`** is whole degrees, **`m`** is whole minutes, and **`s`** is seconds as a float.

`dmsDeg(dms):`
Inverse function of **`degDMS`**; the argument can be either a 2-tuple **`(d,m)`** or a 3-tuple **`(d,m,s)`**, and each value can be either **`int`** or **`float`** type.

There are three classes in module `terrapos.py`:

`TerraPosition`
This class represents an arbitrary position on the earth's surface as a latitude and longitude in radians. It may optionally include an elevation above sea level in feet and a map datum value (such as WGS-84 and NAD-27) defining the overall coordinate system in use. (Most GPS coordinates are given as WGS-84, while most topo maps use NAD-27. The difference can be as much as 200 feet.)

Objects of this class support a number of operations in spherical geometry, such as the ability to find the surface (rhumbline) distance between two points, or to find a new position given a starting position, bearing, and distance.

`LatLon`
This class is derived from the **`TerraPosition`** class. It differs from the base class in that it explicitly represents lat-long coordinates in degrees (as opposed to UTM coordinates).

You might ask, why doesn't the **`TerraPosition`** class suffice for representing lat-lon coordinates, since we can easily convert the radian values to degrees? Well, we certainly could do it that way, but there is an annoying little pathology of such conversions that the **`LatLon`** class is designed to circumvent.

Suppose we take an angle, such as 107° 50' 0" and convert it to radians. For some values, converting back to mixed units, and then rounding the individual units, can cause this value to be displayed as 107° 49' 60.0".

To prevent this kind of ugliness, the **`LatLon`** class remembers the original units and regurgitates those units when displaying the value in string form, rather than converting them from radians. The class constructor also takes care of converting the original values to radians for computational convenience in the base class.

`TerraBox`
An instance of class **`TerraBox`** represents a rectangular area in lat-long coordinates. This is useful for applications such as producing a map of a specific area. The class also supports methods that can tell you whether a point is inside such an area; whether two areas overlap; and the rectangles defining the intersection and the union of two rectangles.

7.2. The `mapbase.py` module

Module **`mapbase.py`** contains mechanisms for building up a composite image from individual tiles and supporting transformations between image coordinates and terrestrial coordinates.

Principle classes include:

`MapBase`
Represents the entire image pyramid: all sizes and kinds of tiles.

`MagBox`
Represents all the available map information for a specific rectangular area, that is, for a range of latitudes and longitudes. Such an area will appear as a rectangle in the usual Mercator projection with latitude running vertically and longitude running horizontally.

A **`MagBox`** object manages the geometry of assembling a set of tiles into a single rectangle, and can produce either photo or topo images.

`CornerSet`
An object of this class represents the `corners` file for a given magnification. Its purpose is to manage just the geometry of *tile slots*, the square holes into which tiles fit. It is a container for **`TileCorner`** and **`TileSlot`** objects.

`TileSlot`
A **`TileSlot`** object describes one tile slot: its corners, and its tile number, which is called **`tileCR`** (for tile row and column) internally.

A tile slot is not considered to exist unless all four of its corners are known.

`TileCorner`
Each **`TileCorner`** object describes one corner of a tile, corresponding to one line of the `corners` file for a given magnification.

`UTM`
An object of this class represents one UTM coordinate. In the current version, there is no obvious way to convert between UTM and lat-long coordinates. When such algorithms become available, they will be moved into the `terrapos.py` module.

7.3. Tile interpolation algorithm

Working with tiles requires that we convert back and forth between two coordinate systems:

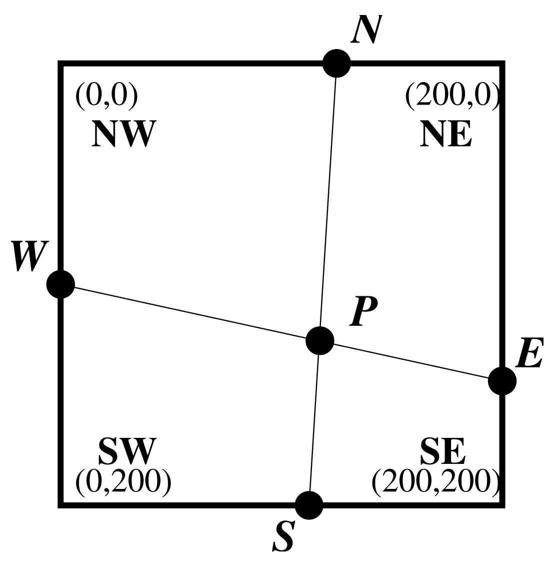
- *Display coordinates* are the coordinates of pixels on the screen. As usual for windows, y increases from top to bottom, an inversion of the usual Cartesian y coordinate.

  Each map tile has an upper left (northwest) corner at display coordinate (0,0) and occupies a 200 x 200 pixel area on the screen. So, for example, the pixel in the southwest corner is at display coordinate (0,199).

- *Terrestrial coordinates* are latitude and longitude (although UTM coordinates may be supported in the future).

For assorted complex historical reasons, longitude and latitude lines are not exactly parallel to the x and y coordinates of display space. Accordingly, the algorithm for converting between display coordinates and terrestrial coordinates is not trivial, but it is reasonably straightforward.

Here is a diagram of an idealized map tile:



In this figure, the four corners are labeled NW, NE, SW, and SE. Consider a point P within the tile. Point W is the point on the west edge of the tile with the same latitude as point P, and point E is the point on the east edge with the same latitude as P. Assuming that the tile is an accurate Mercator projection of the terrain, there should be a straight line EPW connecting all points in the tile with the same latitude. Similarly, line NPS connects the points with the same longitude.

Therefore, deriving the display coordinate of P from its lat-long coordinates is straightforward:

1. Find the coordinates of point W by interpolating the latitude of P between the latitudes of points NW and SW, and then project that fraction onto the display length of 200.
2. Find the coordinates of point E by interpolating along the east side similarly.
3. Find the coordinates of points N and S by interpolating P's longitude along the north and south edges respectively.
4. Derive the coordinates of P by finding the intersection of lines WE and NS. We use the two-point form of the equation of a line, $(y-y_a)/(x-x_a)=(y_b-y_a)/(x_b-x_a)$, where the coordinates of the two points are (x_a,y_a) and (x_b,y_b). This gives us equations for lines WE and NS, and it is simple linear algebra to solve the system of these two equations in two variables.

The inverse transform uses the same procedure, simply exchanging the two coordinate systems.