Stream Network Movements (SNM) Vignette

Donald T. McKnight

Savanna Field Station

2025-01-17

Table of Contents

Introduction and overview

This package is designed to analyze movement patterns of animals within stream networks. It includes four main functions:

- *prep.data()* = takes the raw input data and formats it for analysis, including creating lookup tables to increase speed.

- *calc.stream.dist()* = calculates the minimum distance along a stream network between two points.
- *movements()* = calculates the total space use (stream length), distance between consecutive points, cumulative movement over time, and the direction of movement
- *dist.over.time()* = calculates distances moved over given time intervals (e.g., distances moved over 24 hours, over 48 hours, etc.)

All results are reported in meters and represent minimum distances moved travelling down the center line of a stream (i.e., they are stream length distances). While stream networks can include any number of branches, they cannot include any braids (i.e., streams cannot split and rejoin, forming islands). See the input section for additional details.

This vignette includes examples of input data and how to run the functions, as well as additional details about how the functions work (with relevant considerations for the accuracy of the data).

Input data

Overview

Three input objects are required:

- Shapefile (polyline) of the stream network, describing the center line of the stream
- Data frame with the coordinates and IDs of each node
- Data frame with the coordinates, IDs, and (optionally) dates and times each time an animal was tracked (dates and times are required for the *dist.over.time()* function and highly recommended for *movements()*)

All three objects must be in the same projected format and coordinate system (this package was written and tested using decimal degrees).

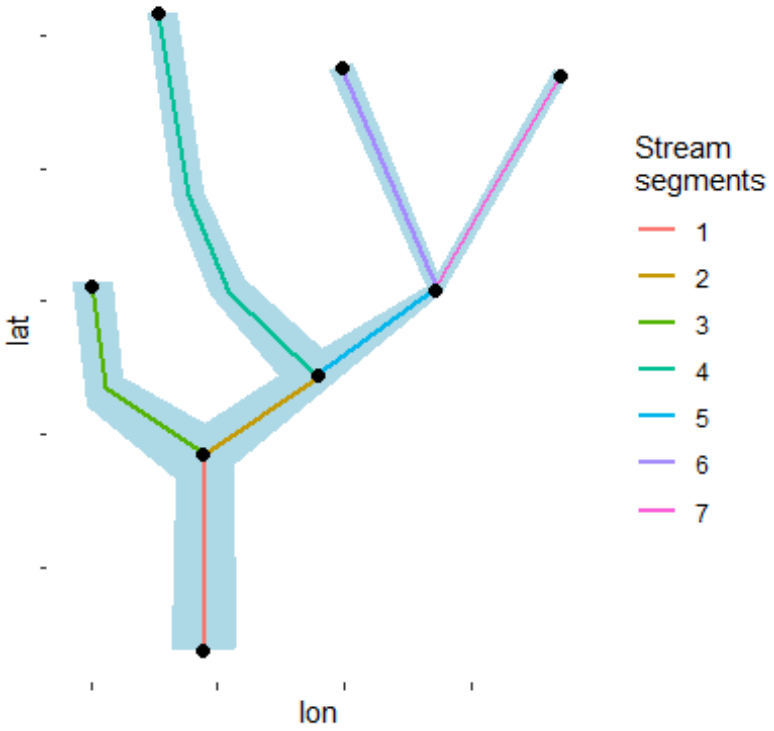
A fourth object (a shapefile showing the boundaries of the stream) has been included for illustrative purposes, but you do not need a boundary shapefile to use the package.

Shapefile

To use this package, you need a shapefile of the stream network you want to analyze. This should be a polyline that runs down the center of the stream, where each segment of the network has a different ID (label). “Segments” refer to “edges” of a network, with the section of stream between any two nodes counting as a unique segment. Nodes must be included anywhere that two segments meet as well as at the ends of any terminal segments.

Let's look at the *stream.line* and *nodes* objects that come with the **SNM** package. We'll also include a polygon shapefile showing the boundaries of the stream. This shapefile is not actually needed for any functions; it is just included here illustrate the point that the polyline you need should run down the center of the stream.

```
library(ggplot2)
library(ggnewscale)
library(SNM)

ggplot()+
  geom_sf(data=stream.boundaries,fill="lightblue",color="lightblue")+
#include the shapefile of the stream area as an illustration
  geom_sf(data=stream.line,aes(color=as.factor(id)),linewidth = 1)+ #include
the required shapefile of the center of the stream network including each
segment
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments",fill="Points")+
  new_scale_color() +
  geom_point(data=nodes,aes(y=lat,x=lon),size=2) #include the nodes
```



Here we can see a branching stream network with 7 segments (colored lines) and 8 nodes (black points). Each segment follows the center of a stream channel, and each node occurs at the junction of two or more segments or the start or end of a terminal segment.

Note that larger stream networks will take longer to run, and you should trim your network to only include the areas used by your animals. Remove any side branches that animals

never visited (to your knowledge) and trim terminal segments to only include the areas you know animals visited.

Stream networks cannot include braids (i.e., sections where the streams fork and reconnect later, forming islands). If your system is braided, you can still analyze it if your tracking efforts were focused on main channels such that you can strategically remove braids from the network and leave a minimum path behind that contains no braids. Alternatively, you may be able to assess movements in small chunks, using subsets of the data that do not include braids.

I recommend creating shapefiles in a program like ArcGIS or QGIS. You can either trace a stream specifying segments as you go, or download an existing shapefile and split it into appropriate segments. Load it into R using the *st_read()* function in the **sf** package.

Note that you must have an ID for each segment, even if your "network" is a single linear stream channel (one segment). Segment IDs should be integers.

Nodes

Nodes occur at the intersection of segments or the start and end of terminal segments (see the Shapefile section). The node object needs to be a data frame with 3 columns containing the latitude, longitude, and ID of each node. You can specify the names of the columns in the *prep.data()* function, but the defaults are "lat", "lon" and "id", respectively. The ID column must be integers. Node IDs do not need to correspond in any meaningful way with segment IDs.

An example node object (*nodes*) is provided

```
nodes
#>         lon      lat id
#> 1 -88.99856 17.17619  1
#> 2 -88.99856 17.17692  2
#> 3 -88.99873 17.17859  3
#> 4 -88.99811 17.17722  4
#> 5 -88.99764 17.17754  5
#> 6 -88.99801 17.17837  6
#> 7 -88.99715 17.17834  7
#> 8 -88.99900 17.17756  8
```

Note that you must have a node object, even if your "network" is a single linear stream channel (i.e., one segment with a node on each end).

Note that for internal calculations, the node object is paired with the segments in the shapefile object by identifying the closest matches between points in both objects. Thus, it automatically detects which nodes go with which segments. Additionally, the ending points for each segment in the shapefile do not need to perfectly match the coordinates of the nodes, but they should be as close as reasonably possible, because large differences will result in inaccurate calculations.

Animal positions

Finally, you need a data frame containing the locations each time an animal was tracked. At a minimum, this data frame should include columns for latitude, longitude, and the ID of each animal. You can specify the names of the columns in the functions, but the defaults are “lat”, “lon” and “id”, respectively.

The ID column does not require a specific format and can include characters.

Optionally, you can include a column of date and time information (in the POSIX format). This column is called “date.time” by default, but you can specify an alternate name in the functions. This column is required for the *dist.over.time()* function and highly recommended for the *movements()* function.

Additional columns of metadata can optionally be included and will be returned alongside the outputs of the *movements()* function.

The *dist.over.time()* and *movements()* functions loop over each individual, providing outputs per individual.

An example object (*animal.points*) is included. Note that this object contains 2 sets of latitude and longitude. This is strictly for illustrative purposes and your object should only include one set.

```
animal.points
#>      lon.raw  lat.raw lon.shifted lat.shifted      id           date.time
point
#> 1  -88.99845 17.17668   -88.99855    17.17668 turtle1 2023-01-01 12:00:00
1
#> 2  -88.99848 17.17698   -88.99848    17.17698 turtle1 2023-01-02 12:00:00
2
#> 3  -88.99838 17.17710   -88.99834    17.17707 turtle1 2023-01-03 12:00:00
3
#> 4  -88.99874 17.17835   -88.99869    17.17835 turtle2 2023-01-01 12:30:00
1
#> 5  -88.99849 17.17765   -88.99851    17.17765 turtle2 2023-01-01 13:00:00
2
#> 6  -88.99792 17.17732   -88.99794    17.17734 turtle2 2023-01-02 12:00:00
3
#> 7  -88.99783 17.17794   -88.99781    17.17794 turtle2 2023-01-03 12:00:00
4
#> 8  -88.99747 17.17784   -88.99747    17.17784 turtle2 2023-01-06 12:00:00
5
#> 9  -88.99776 17.17746   -88.99776    17.17746 turtle2 2023-01-07 18:00:00
6
#> 10 -88.99875 17.17700   -88.99871    17.17703 turtle2 2023-01-08 12:00:00
7
#> 11 -88.99889 17.17730   -88.99895    17.17730 turtle2 2023-01-09 12:00:00
8
```

This package can be used with automated stations, where the coordinates describe each station with a row of data each time an animal was detected at a given station.

To calculate distances, for each animal coordinate, it identifies the corresponding point along the network shapefile that is closest to that point. Thus, for the distance between a pair of points, it actually calculates the distance down the center of the stream (following the shapefile) between those two points. As a result, all distances are calculated as stream length distances, and movements across stream channels may not be accurately included. For example, if an animal swims straight across a stream between two points without moving up or down the channel, that will be returned as a movement of 0 meters. This is only problematic when looking at movements over a very small spatial scale or in very wide river systems.

Because each animal point is matched to the closest point in the shapefile object, the frequency of points in the shapefile affects the accuracy of the calculations (see the *prep.data()* section for details on how to easily increase the point frequency).

Note that there may be issues in the rare case where one very wide channel runs parallel to a narrow channel, with only a small strip of land between them. In that situation, an animal on the edge of the wide channel could mistakenly get assigned to the smaller channel. If this is a concern in your system, simply manually adjust relevant points to be closer to the center line of the channel they actually occurred in. Doing so will not affect the accuracy of results. This should not be necessary in the vast majority of data sets.

It is highly recommended that you plot your data in R or elsewhere to check for aberrant or inaccurate points before running this package.

Using the package (example)

prep.data()

Before running any calculations, you need to use the *prep.data()* function to format the stream network for the functions and calculate lookup tables to increase the speed of subsequent functions. This function takes the stream network object (shapefile) and the node object (data frame).

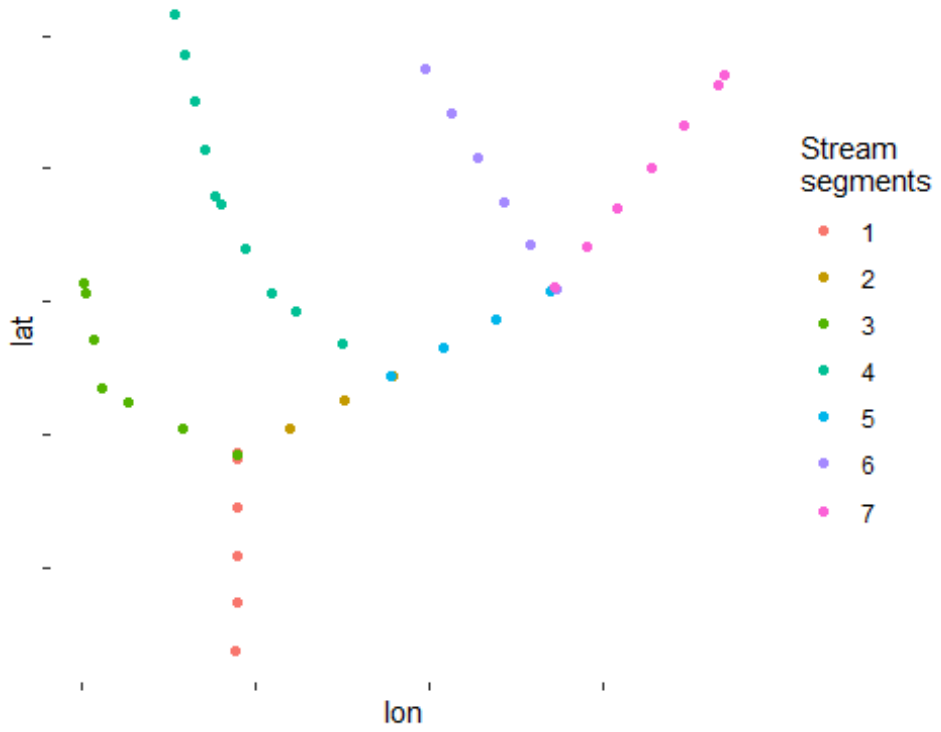
Pay attention to the *freq* argument. This will add additional points between each pair of points in the stream network shapefile, thus greatly improving the accuracy of subsequent calculations. If, for example, your stream included a straight segment that was 1 km long, the shapefile might only have a coordinate included at each end of that segment. As a result, when matching the animal coordinates with the shapefile coordinates, any animals in between those two coordinates would get paired with one of those end coordinates, thus moving them very far from their actual locations. In contrast, setting *freq* = 1 would add an additional coordinate each meter between those two end coordinates, thus providing a close match for each animal point.

The *freq* argument is specified in meters, so the lower the number, the tighter the resolution and the higher the accuracy in the data. However, lower numbers also increase the run time. Thus, you should carefully select a number that balances the two based on your study system and computer resources. If your animals regularly move dozens of kilometers, a higher number like 10 will likely be sufficient. In contrast, if your animals only move very small distances, smaller numbers like 1 or 0.5 may be needed. The total size of the system also greatly affects run time. So a system that includes many kilometers may need a larger value. If you have a large system when looking at all animals, but each individual only uses a small portion of the system, it may be more efficient to run the functions separately on each individual, with a low *freq* per individual and the shapefile trimmed to only the area used by that individual.

To illustrate, let's run the example data through *prep.data()* at several values of *freq* then plot the results.

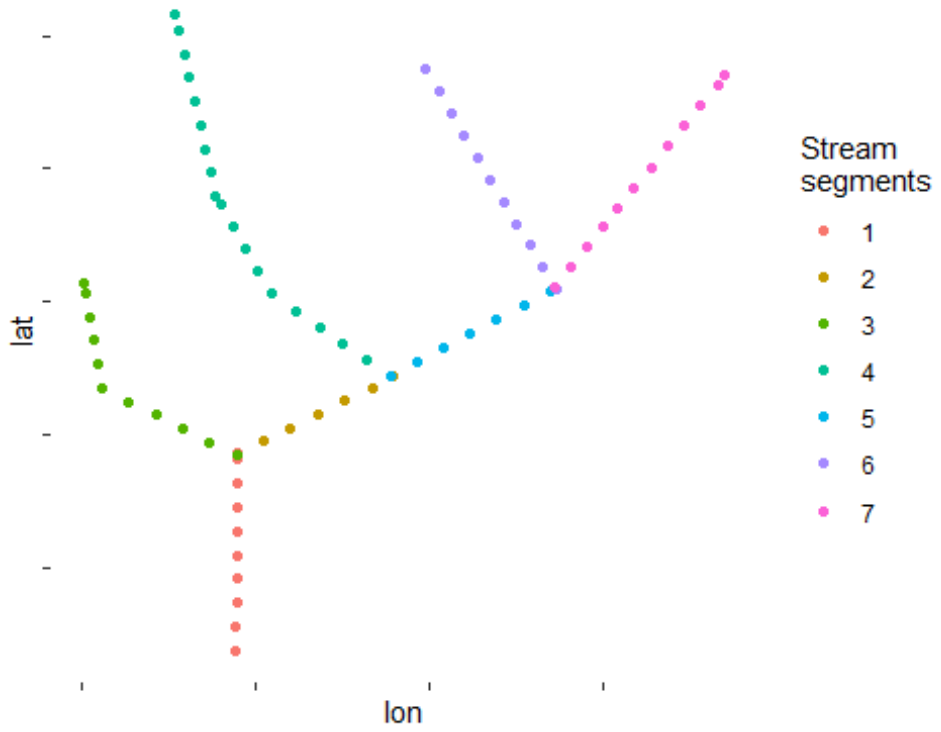
First, let's use a high value (*freq*=20). We'll provide the function with the shapefile (l = stream.line) and nodes data frame (nodes = nodes). The lat, lon, and id columns are already named following the defaults, so we do not actually have to specify them, but for illustrative purposes, we'll go ahead and do so.

```
network.20 <-
prep.data(l=stream.line,freq=20,nodes=nodes,lon.name="lon",lat.name="lat",nod
e.name="id")

#the increased.line object contains the new coordinates that were added
ggplot(network.20$increased.line,aes(x=lon,y=lat,color=as.character(id)))+
  geom_point()+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments")
```



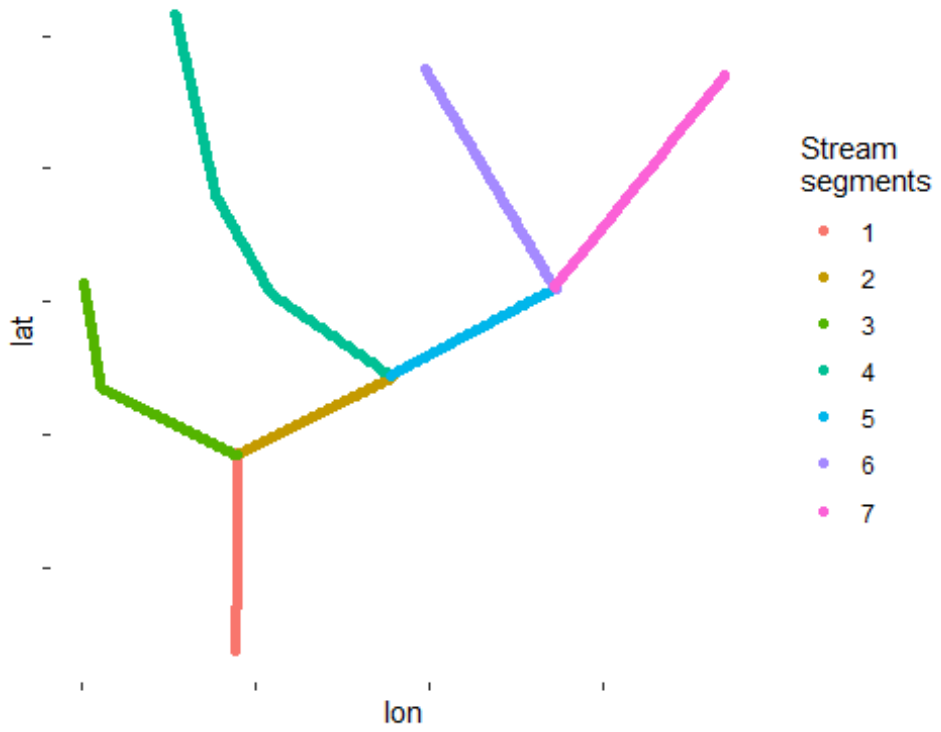
As you can see, now instead of a line for each segment, we have a series of points separated by no more than 20 m (sometimes less at the ends if a segment was not evenly dividable by 20). This will give us a better resolution than the initial shapefile, but is still pretty crude. Let's do it again at 10 m

```
network.10 <-
prep.data(l=stream.line,freq=10,nodes=nodes,lon.name="lon",lat.name="lat",nod
e.name="id")

ggplot(network.10$increased.line,aes(x=lon,y=lat,color=as.character(id)))+
  geom_point()+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments")
```



This is an obvious improvement with closer points, but for such a small area and small data set, we can go narrower, so let's use 1 m

```
network.1 <-
prep.data(l=stream.line,freq=1,nodes=nodes,lon.name="lon",lat.name="lat",node
.name="id")

ggplot(network.1$increased.line,aes(x=lon,y=lat,color=as.character(id)))+
  geom_point()+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments")
```



Now the points are so close together that they almost look like a solid line. We'll use this resolution moving forward. The *network.1* object is a list containing all the information on the stream network necessary for all subsequent calculations.

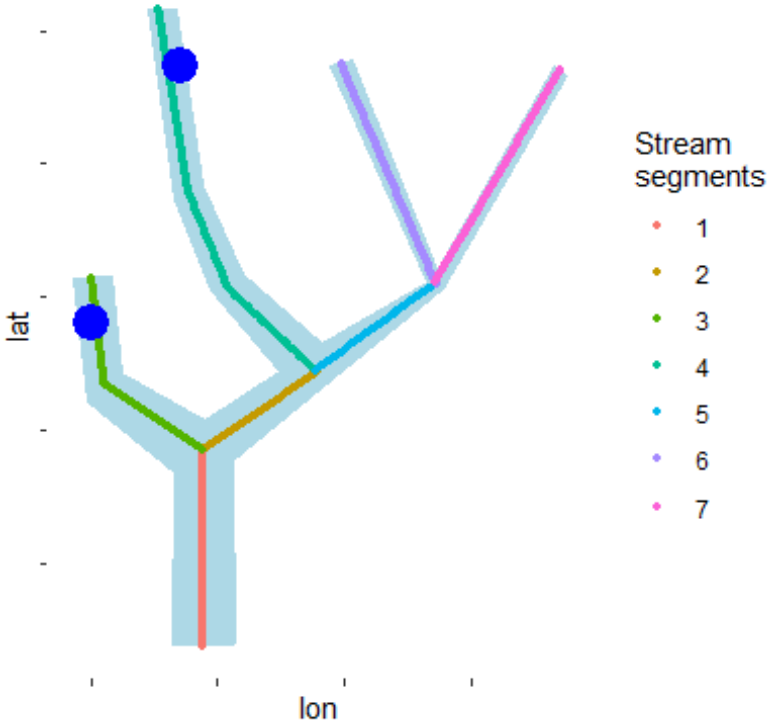
calc.stream.dist()

This is a core function used internally by the other functions and simply calculates the shortest distance between two points within the stream network by following the center line of the stream (i.e., this is a stream length distance). To use it separately from the other functions, we simply supply it with two coordinates (in the same projected system as the stream network) and the stream network object made previously using *prep.data()*. Coordinates need to be entered as vectors with longitude followed by latitude.

For example, let's look at two hypothetical positions. Note that the points do not perfectly align with the network object because, in a real system, the network object represents only the center line down a stream channel, and animals may be anywhere within the channel. We'll include the *stream.boundaries* object to illustrate this.

```
ggplot()+
  geom_sf(data=stream.boundaries,fill="lightblue",color="lightblue")+

geom_point(data=network.1$increased.line,aes(x=lon,y=lat,color=as.character(i
d)),size=1)+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments")+
```

```
  new_scale_color()+
  geom_point(aes(x=-88.99900,y=17.1774),color="blue",size=6)+
  geom_point(aes(x=-88.99865,y=17.17837),color="blue",size=6)
```



To calculate the distance between these points following the stream network, we simply run the following (note that we must supply the network object):

```
calc.stream.dist(p1=c(-88.99900,17.1774),p2=c(-
88.99865,17.17837),data=network.1)
#> [1] 280.0319
```

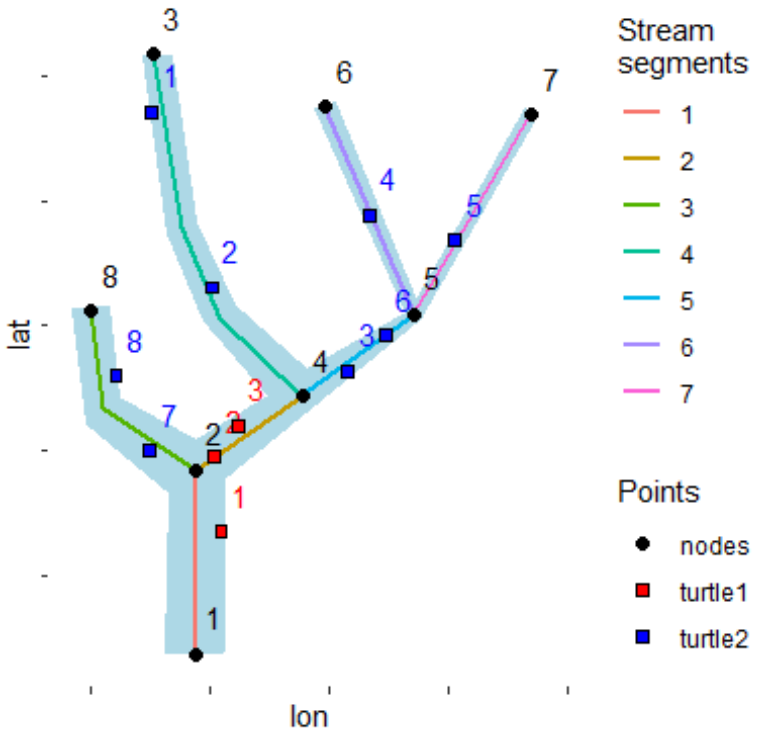
This shows that the points are 280.03 m apart following the center lines of the stream network.The calculation was performed by using the lookup tables to determine the distance from the point on the left to the lower node on segment 3, the entire length of segment 2, and the distance from the lower node on segment 4 to the second point. Those distances were then summed.

Movements()

In most cases, we want to look at large chunks of data, not two individual points. The *movements()* function lets us do this.

To illustrate, let's use the *animal.points* data frame that comes with **SNM**. It contains two sets of coordinates. The "raw" set contains the actual data and is the only set needed for analyses. The "shifted" points are simply to illustrate how the functions work. Let's start by plotting the raw data.

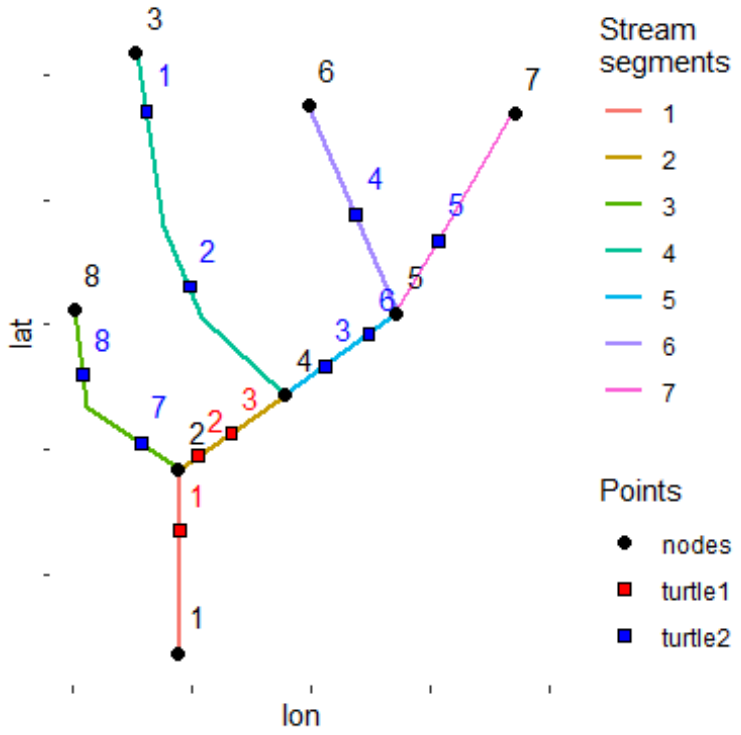
```
nodes$nodes <- "nodes"

ggplot()+
   geom_sf(data=stream.boundaries,fill="lightblue",color="lightblue")+
  geom_sf(data=stream.line,aes(color=as.factor(id)),linewidth = 1)+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments",fill="Points",x="lon",y="lat")+
  new_scale_color() +
  geom_point(data=animal.points,aes(x=lon.raw,y=lat.raw,fill=id))+

geom_text(data=animal.points,label=animal.points$point,aes(x=lon.raw,y=lat.ra
w,color=id),nudge_y=0.00015,nudge_x=0.00008)+
  scale_color_manual(values=c("red","blue"))+
  guides(color="none")+
  geom_point(data=nodes,aes(y=lat,x=lon,fill=nodes),size=2)+

geom_text(data=nodes,label=nodes$id,aes(x=lon,y=lat),nudge_y=0.00015,nudge_x=
0.00008)+

geom_point(data=animal.points,aes(x=lon.raw,y=lat.raw,fill=id),shape=22,size=
2)+
  scale_fill_manual(values=c("black","red","blue"))
```



Here we have coordinates for two turtles. As you can see, they are within the stream boundaries, but aren't lined up with the center line object. The functions in **SNM** will take care of that by matching each point with the closest point on the stream network using the

object generated in *prep.data()*, but for illustrative purposes, let's plot roughly what that looks like using the "shifted" columns (these were generated manually and are not overly accurate; again, just for illustration).

```
ggplot()+
   #geom_sf(data=stream.boundaries,fill="lightblue",color="lightblue")+
  geom_sf(data=stream.line,aes(color=as.factor(id)),linewidth = 1)+
  theme(axis.text=element_blank(),panel.background=element_blank())+
  labs(color="Stream\nsegments",fill="Points",x="lon",y="lat")+
  new_scale_color() +
  geom_point(data=animal.points,aes(x=lon.shifted,y=lat.shifted,fill=id))+

geom_text(data=animal.points,label=animal.points$point,aes(x=lon.shifted,y=la
t.shifted,color=id),nudge_y=0.00015,nudge_x=0.00008)+
  scale_color_manual(values=c("red","blue"))+
  guides(color="none")+
  geom_point(data=nodes,aes(y=lat,x=lon,fill=nodes),size=2)+

geom_text(data=nodes,label=nodes$id,aes(x=lon,y=lat),nudge_y=0.00015,nudge_x=
0.00008)+

geom_point(data=animal.points,aes(x=lon.shifted,y=lat.shifted,fill=id),shape=
22,size=2)+
  scale_fill_manual(values=c("black","red","blue"))
```



Now you can see that the points are perfectly aligned with the center of lines of the stream network and the boundary shapefile is irrelevant. Again, you do not need to do this alignment manually. This is simply an illustration to show what the functions are doing internally (you also do not need the boundary shapefile object).

Now that we have plotted the data, let's do the actual analysis. There are several options for what calculations to run. We will run them all by setting *space.use*, *from.previous*, and *cumulative* to TRUE. We also need to include the name of the node that is furthest downstream (in this case 1). The date.time column is not strictly necessary, but without it, it is important that the data are sorted from oldest to newest (the *dist.from.prev()* and *cumulative.dist()* functions will not give correct results otherwise). If the date.time column is included, the data will be sorted internally (including it is highly recommended).

```
move.results <- movements(data=network.1,
          space.use=T,
          from.previous=T,
          cumulative=T,
          downstream.node=1,
          coords=animal.points,
          lon.name="lon.raw",
          lat.name="lat.raw",
          id.name="id",
          date.time.name="date.time")
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"

move.results
#>    lon.shifted lat.shifted           date.time point space.use
dist.from.prev
#> 1    -88.99855    17.17668 2023-01-01 12:00:00     1        NA
NA
#> 2    -88.99848    17.17698 2023-01-02 12:00:00     2  37.16384
37.00000
#> 3    -88.99834    17.17707 2023-01-03 12:00:00     3  53.16384
16.00000
#> 4    -88.99869    17.17835 2023-01-01 12:30:00     1        NA
NA
#> 5    -88.99851    17.17765 2023-01-01 13:00:00     2  81.84138
81.84138
#> 6    -88.99794    17.17734 2023-01-02 12:00:00     3 169.12858
87.28720
#> 7    -88.99781    17.17794 2023-01-03 12:00:00     4 256.56658
87.43800
#> 8    -88.99747    17.17784 2023-01-06 12:00:00     5 293.56658
86.00000
#> 9    -88.99776    17.17746 2023-01-07 18:00:00     6 293.56658
52.43800
#> 10   -88.99871    17.17703 2023-01-08 12:00:00     7 373.41909
125.85252
```

```
#> 11   -88.99895    17.17730 2023-01-09 12:00:00     8 414.46992
41.05082
#>    cumulative.dist dist.from.downstream  direction time.diff       lon
lat
#> 1               NA              55.0000       <NA>        NA -88.99845
17.17668
#> 2         37.00000              92.0000   upstream      24.0 -88.99848
17.17698
#> 3         53.00000             108.0000   upstream      24.0 -88.99838
17.17710
#> 4               NA             285.9811       <NA>        NA -88.99874
17.17835
#> 5         81.84138             204.1397 downstream       0.5 -88.99849
17.17765
#> 6        169.12858             162.8525 downstream      23.0 -88.99792
17.17732
#> 7        256.56658             250.2905   upstream      24.0 -88.99783
17.17794
#> 8        342.56658             238.2905 downstream      72.0 -88.99747
17.17784
#> 9        395.00457             185.8525 downstream      30.0 -88.99776
17.17746
#> 10       520.85709             104.0000 downstream      18.0 -88.99875
17.17700
#> 11       561.90791             145.0508   upstream      24.0 -88.99889
17.17730
#>         id
#> 1  turtle1
#> 2  turtle1
#> 3  turtle1
#> 4  turtle2
#> 5  turtle2
#> 6  turtle2
#> 7  turtle2
#> 8  turtle2
#> 9  turtle2
#> 10 turtle2
#> 11 turtle2
```

This outputs a data frame with all of the input data, plus multiple additional columns:

- "space.use" was returned by setting *space.use* = T and is the entire length of stream used by an animal up to and including a given point. No portion of stream is counted more than once, so on a straight system with no branches, it would simply be the distance between the furthest two points. Notice that the "space.use" does not increase from points 5 to 6 for turtle2, because the movement from 5 to 6 was entirely within an area the turtle had already used. The final value per individual indicates the total length of stream used and can be viewed as a measure of home range (depending on the system and study organism).

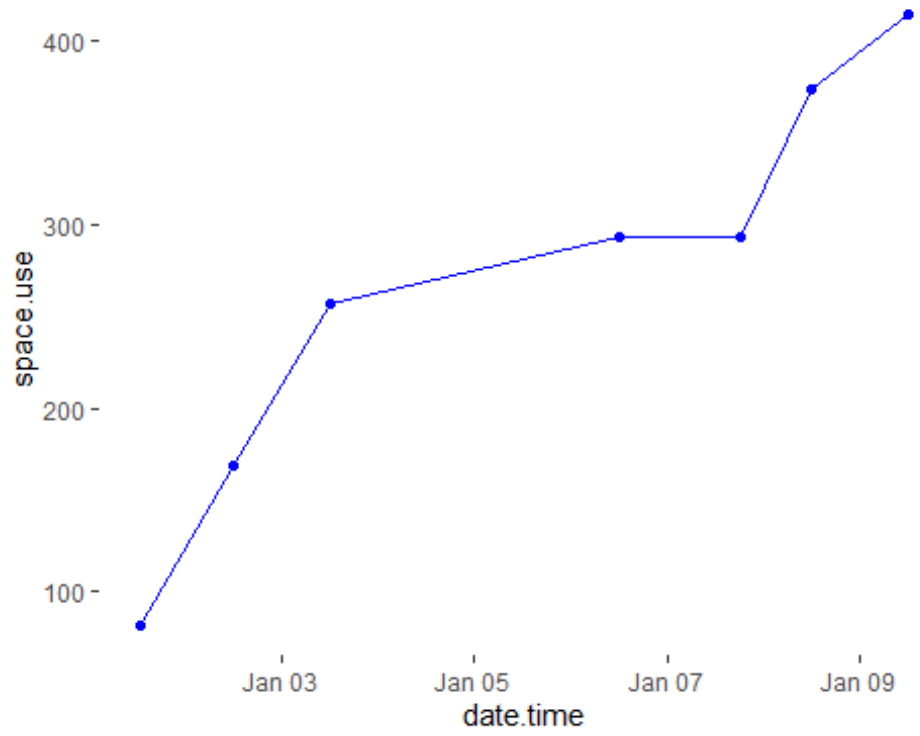
    - In this example, for turtle2, the final space use was the sum of: most of segments 3 and 4, all of segments 2 and 5, and the lower portion of segments 6 and 7 (i.e., it is the shortest path connecting all the points without duplicating any paths). Nothing from segment 1 was included, because there was no evidence of the turtle visiting it. Keep in mind, however, that these are minimum estimates. In many cases, it may have been possible that the turtle did go down segment 1, but was simply not tracked while it was in that segment.
    - Note also that the "space.use" for turtle2 did not increase from points 5 to 6, because that movement was entirely within an area where the turtle had already been documented (see contrast with "cumulative.dist")
- "dist.from.prev" was returned by setting *from.prev* = T and simply shows the distance between each point and the previous point for an individual.
- "cumulative.dist" was returned by setting *cumulative* = T and is the cumulative area used by an individual up to and including the given point. This differs from "space.use" in that areas are counted every time an animal uses them. Thus, this is a measure of the total distance traveled from the start and is the same as the sum of all "dist.from.prev" measurements up to and including the given measurement.
- "dist.from.downstream" was returned by including the *downstream.node* and shows the distance from each point to the furthest node downstream.
- "direction" was returned by including the *downstream.node* and shows whether the animal moved further (upstream) or closer (downstream) to the furthest downstream node
- "time.diff" was returned by including a date.time column and shows the amount of time (in hours) that passed between two points.

Note that for each individual, the first row is mostly NA because the animal has, by definition, not moved yet.

Plotting some of these outputs can give useful insights into animal movement patterns. For example, if we plot the space use of turtle 2, we can see that at first, its space use increased rapidly, before stagnating while it moved around in an area where it had already been documented, followed by another large movement. If you have tracked an animal long enough to establish its home range, you should see this line asymptote. That clearly is not the case here.

```
ggplot(move.results[move.results$id=="turtle2",],aes(x=date.time,y=space.use)
)+
  geom_point(color="blue")+
  geom_line(color="blue")+
  theme(panel.background=element_blank())
#> Warning: Removed 1 row containing missing values or values outside the
scale range
#> (`geom_point()`).
#> Warning: Removed 1 row containing missing values or values outside the
```

```
scale range
#> (`geom_line()`).
```



dist.over.time()

The final function is designed to look at movement patterns over specified amounts of time. There are a lot of options for this function, so we'll go through a few examples.

First, imagine that you want to look at movement patterns as the number of days increased between points. So, you want to know how far animals moved over 24 hours, over 48 hours, over 72 hours, etc. This function lets you do that by taking each point and looking for a previous point that matches the specified time interval, then getting the distance between those two points. We can set this up by telling the function to use days as the units and increase the days by 1 for each time interval (set this with the *time.diff* argument; note that we could also have set *units* to "hours" and *time.diff* to 24 to get the same result). We'd like to look at the maximum possible spread of days, so for now we'll leave *diff.max* on it's default NULL setting. Also, for now, let's shut off the sensitivity arguments by setting them to 0.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
```

```
               units="days",
               time.diff=1,
               diff.max=NULL,
               sensitivity.min=0,
               sensitivity.max=0,
               sensitivity.change=0)
#> [1] "Calculating every 1 days from 1 days to 8 days"
#> [1] "Sensitivity starts at +/- 0 days and increases by 0 days every 1
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>           lon      lat      id           date.time      dist actual.diff
#> 2   -88.99848 17.17698 turtle1 2023-01-02 12:00:00  37.00000           1
#> 3   -88.99838 17.17710 turtle1 2023-01-03 12:00:00  16.00000           1
#> 31  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.00000           2
#> 7   -88.99783 17.17794 turtle2 2023-01-03 12:00:00  87.43800           1
#> 8   -88.99747 17.17784 turtle2 2023-01-06 12:00:00  86.00000           3
#> 81  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.43800           4
#> 10  -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.29051           2
#> 101 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 190.29051           5
#> 102 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 102.85252           6
#> 11  -88.99889 17.17730 turtle2 2023-01-09 12:00:00  41.05082           1
#> 111 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 219.34133           3
#> 112 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 231.34133           6
#> 113 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 143.90334           7
#>     time.diff sensitivity
#> 2           1           0
#> 3           1           0
#> 31          2           0
#> 7           1           0
#> 8           3           0
#> 81          4           0
#> 10          2           0
#> 101         5           0
#> 102         6           0
#> 11          1           0
#> 111         3           0
#> 112         6           0
#> 113         7           0
```

This gives us an output with the distances moved (dist) over a given number of days. Carefully note that this is simply distance moved between a pair of points separated by a given number of days, rather than distance moved from a fixed point over a given number of days.

Also note that because we set sensitivity to 0, only pairs of points that exactly matched our time interval could be used (this also means that “actual.diff” and “time.diff” are identical). As a result, for turtle2, there is no result for the point recorded on 2023-01-02, because there was no exact match 1 day previously. Unless you are using GPS data

loggers, this may be problematic, because your points are unlikely to be separated by such precise time intervals. We can modify this with the sensitivity arguments. These specify a buffer of time around each time interval in which points can be included. As an example, let's use 0.1 days as the buffer (i.e., the time interval is X days +/- 0.1 days) and lets make that fixed by leaving *sensitivity.change* = 0 (note that both *sensitivity.min* and *sensitivity.max* arguments should be 0.1).

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               time.diff=1,
               diff.max=NULL,
               sensitivity.min=0.1,
               sensitivity.max=0.1,
               sensitivity.change=0)
#> [1] "Calculating every 1 days from 1 days to 8 days"
#> [1] "Sensitivity starts at +/- 0.1 days and increases by 0 days every 1
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>           lon      lat      id           date.time      dist actual.diff
#> 2   -88.99848 17.17698 turtle1 2023-01-02 12:00:00  37.00000   1.0000000
#> 3   -88.99838 17.17710 turtle1 2023-01-03 12:00:00  16.00000   1.0000000
#> 31  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.00000   2.0000000
#> 6   -88.99792 17.17732 turtle2 2023-01-02 12:00:00 169.12858   0.9791667
#> 7   -88.99783 17.17794 turtle2 2023-01-03 12:00:00  87.43800   1.0000000
#> 71  -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.56658   1.9791667
#> 8   -88.99747 17.17784 turtle2 2023-01-06 12:00:00  86.00000   3.0000000
#> 81  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.43800   4.0000000
#> 82  -88.99747 17.17784 turtle2 2023-01-06 12:00:00 244.56658   4.9791667
#> 10  -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.29051   2.0000000
#> 101 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 190.29051   5.0000000
#> 102 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 102.85252   6.0000000
#> 103 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 225.98110   6.9791667
#> 11  -88.99889 17.17730 turtle2 2023-01-09 12:00:00  41.05082   1.0000000
#> 111 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 219.34133   3.0000000
#> 112 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 231.34133   6.0000000
#> 113 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 143.90334   7.0000000
#> 114 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.03192   7.9791667
#>     time.diff sensitivity
#> 2           1         0.1
#> 3           1         0.1
#> 31          2         0.1
#> 6           1         0.1
#> 7           1         0.1
```

```
#> 71          2         0.1
#> 8           3         0.1
#> 81          4         0.1
#> 82          5         0.1
#> 10          2         0.1
#> 101         5         0.1
#> 102         6         0.1
#> 103         7         0.1
#> 11          1         0.1
#> 111         3         0.1
#> 112         6         0.1
#> 113         7         0.1
#> 114         8         0.1
```

Now you can see that there are more results returned, and the "actual.diff" column (showing the actual amount of time separating points) is often different from the "time.diff" column (showing the interval it was set to). Also note that now there is a result for turtle2 on 2023-01-02 because there was a point 0.979 days previously, which is within the time interval of 1 day +/- 0.1 days. Also, note that there were actually two possible matches, one at 2023-01-01 12:30:00 and one at 2023-01-01 13:00:00. In cases like this, only one result is returned, and it is the result for the pair where the "actual.diff" is closest to the "time.diff" (in this case, 2023-01-01 12:30:00).

You'll also notice that the buffer (sensitivity) remains fixed as the number of days increases. This may or may not be desirable. In some cases, you might expect that as a study continues, the odds of an animal being tracked at the exact same time of day decrease as the length of time between points increases. Likewise, if you are looking at movements over 1 day, having points from the same time of day is likely important, whereas if you are looking at movements over 100 days, it is much less important that the points were at the same time of day. You can incorporate that into this function by using the *sensitivity.change* argument to specify an amount of additional buffer to be added each time interval. You can also cap this increase by using the *sensitivity.max* argument. So, as an example, let's start with a sensitivity of 0.1, but increase it by 0.02 each day, with a maximum sensitivity of 0.2.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               time.diff=1,
               diff.max=NULL,
               sensitivity.min=0.1,
               sensitivity.max=0.2,
               sensitivity.change=0.02)
#> [1] "Calculating every 1 days from 1 days to 8 days"
```

```
#> [1] "Sensitivity starts at +/- 0.1 days and increases by 0.02 days every 1
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>           lon      lat      id           date.time      dist actual.diff
#> 2   -88.99848 17.17698 turtle1 2023-01-02 12:00:00  37.00000   1.0000000
#> 3   -88.99838 17.17710 turtle1 2023-01-03 12:00:00  16.00000   1.0000000
#> 31  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.00000   2.0000000
#> 6   -88.99792 17.17732 turtle2 2023-01-02 12:00:00 169.12858   0.9791667
#> 7   -88.99783 17.17794 turtle2 2023-01-03 12:00:00  87.43800   1.0000000
#> 71  -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.56658   1.9791667
#> 8   -88.99747 17.17784 turtle2 2023-01-06 12:00:00  86.00000   3.0000000
#> 81  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.43800   4.0000000
#> 82  -88.99747 17.17784 turtle2 2023-01-06 12:00:00 244.56658   4.9791667
#> 10  -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.29051   2.0000000
#> 101 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 190.29051   5.0000000
#> 102 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 102.85252   6.0000000
#> 103 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 225.98110   6.9791667
#> 11  -88.99889 17.17730 turtle2 2023-01-09 12:00:00  41.05082   1.0000000
#> 111 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 219.34133   3.0000000
#> 112 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 231.34133   6.0000000
#> 113 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 143.90334   7.0000000
#> 114 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.03192   7.9791667
#>     time.diff sensitivity
#> 2           1        0.10
#> 3           1        0.10
#> 31          2        0.12
#> 6           1        0.10
#> 7           1        0.10
#> 71          2        0.12
#> 8           3        0.14
#> 81          4        0.16
#> 82          5        0.18
#> 10          2        0.12
#> 101         5        0.18
#> 102         6        0.20
#> 103         7        0.20
#> 11          1        0.10
#> 111         3        0.14
#> 112         6        0.20
#> 113         7        0.20
#> 114         8        0.20
```

Notice now that the “sensitivity” column increases as the “time.diff” increases, but it never exceeds 0.2.

So far so good, but for large data sets, getting a result every day may result in a very long run time. So let’s get every other day instead. To do this, we’ll simply change time.diff to 2 and, to be consistent, we’ll double the sensitivity increase.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               time.diff=2,
               diff.max=NULL,
               sensitivity.min=0.1,
               sensitivity.max=0.2,
               sensitivity.change=0.04)
#> [1] "Calculating every 2 days from 2 days to 8 days"
#> [1] "Sensitivity starts at +/- 0.1 days and increases by 0.04 days every 2
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>           lon      lat      id           date.time     dist actual.diff
#> 3   -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.0000    2.000000
#> 7   -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.5666    1.979167
#> 8   -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.4380    4.000000
#> 10  -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.2905    2.000000
#> 101 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 102.8525    6.000000
#> 11  -88.99889 17.17730 turtle2 2023-01-09 12:00:00 231.3413    6.000000
#> 111 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.0319    7.979167
#>     time.diff sensitivity
#> 3           2        0.10
#> 7           2        0.10
#> 8           4        0.14
#> 10          2        0.10
#> 101         6        0.18
#> 11          6        0.18
#> 111         8        0.20
```

There are several other ways that you can control the number of comparisons being made (run time increases dramatically as number of comparisons increases). One option is to set a cap on the maximum time interval you want to analyze. To do this, simply set the *diff.max* to the largest time interval you want to examine.

Another option is to supply a vector of custom times, rather than using a fixed interval of increase. For example, let's just look at the results for 2, 4, and 8 days.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               sensitivity.min=0.1,
```

```
               sensitivity.max=0.2,
               sensitivity.change=0.02,
               custom.times = c(2,4,8))
#> [1] "Using custom sequence of times"
#> [1] "Sensitivity starts at +/- 0.1 days and increases by 0.02 days every 1
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>          lon      lat      id           date.time     dist actual.diff
#> 3  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.0000    2.000000
#> 7  -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.5666    1.979167
#> 8  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.4380    4.000000
#> 10 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.2905    2.000000
#> 11 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.0319    7.979167
#>    time.diff sensitivity
#> 3          2        0.12
#> 7          2        0.12
#> 8          4        0.16
#> 10         2        0.12
#> 11         8        0.20
```

Now we have a very reduced output. This is especially useful for large data sets that span a long time period. If you have a year of data, for example, running this every day will produce very long run times and is probably unnecessary, so something like *custom.times* = c(1,2,3,4,5,6,7,30,60,90,120,160,200,240,300,330,360) may be sufficient while greatly reducing the run time.

Note that when we use 2, 4 and 8 as our time interval, the "sensitivity" increased linearly with each 1 time unit increase, even though most time intervals were not returned (i.e., it increased as if we had all time intervals from 1-8). Also, *sensitivity.min* is always applied to 1 of the specified units. So even though *sensitivity.min* = 0.1, the lowest sensitivity value returned was 0.12 because the lowest time interval used was 2 days (i.e., for day 1 [not returned] sensitivity = 0.1, and for day 2 sensitivity = 0.12). You can, alternatively, specify a vector of custom sensitivity values, which will override the other sensitivity arguments. For example, let's make it day 2 = 0.1, day 4 = 0.15, and day 8 = 2.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               custom.times = c(2,4,8),
               custom.sensitivity = c(0.1,0.15,.2))
#> [1] "Using custom sequence of times"
#> [1] "Using custom sequence of sensitivities"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
```

```
#>          lon      lat      id           date.time     dist actual.diff
#> 3  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.0000    2.000000
#> 7  -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.5666    1.979167
#> 8  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.4380    4.000000
#> 10 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.2905    2.000000
#> 11 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.0319    7.979167
#>    time.diff sensitivity
#> 3          2        0.10
#> 7          2        0.10
#> 8          4        0.15
#> 10         2        0.10
#> 11         8        0.20
```

Now the sensitivity values simply pair with the time values supplied.

Note that you can use *custom.sensitivity* without using *custom.times* if, for example, you want a non-linear increase in sensitivity. If doing this, you must ensure that the length of your sensitivity vector is the same as the total number of time intervals you are using.

```
dist.over.time(data=network.1,
               coords=animal.points,
               lon.name="lon.raw",
               lat.name="lat.raw",
               id.name="id",
               date.time.name="date.time",
               units="days",
               time.diff=1,
               custom.sensitivity = c(.1,.11,.13,.16,.20,.25,.31,.37))
#> [1] "Calculating every 1 days from 1 days to 8 days"
#> [1] "Using custom sequence of sensitivities"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"
#>           lon      lat      id           date.time      dist actual.diff
#> 2   -88.99848 17.17698 turtle1 2023-01-02 12:00:00  37.00000   1.0000000
#> 3   -88.99838 17.17710 turtle1 2023-01-03 12:00:00  16.00000   1.0000000
#> 31  -88.99838 17.17710 turtle1 2023-01-03 12:00:00  53.00000   2.0000000
#> 6   -88.99792 17.17732 turtle2 2023-01-02 12:00:00 169.12858   0.9791667
#> 7   -88.99783 17.17794 turtle2 2023-01-03 12:00:00  87.43800   1.0000000
#> 71  -88.99783 17.17794 turtle2 2023-01-03 12:00:00 256.56658   1.9791667
#> 8   -88.99747 17.17784 turtle2 2023-01-06 12:00:00  86.00000   3.0000000
#> 81  -88.99747 17.17784 turtle2 2023-01-06 12:00:00  75.43800   4.0000000
#> 82  -88.99747 17.17784 turtle2 2023-01-06 12:00:00 244.56658   4.9791667
#> 9   -88.99776 17.17746 turtle2 2023-01-07 18:00:00 110.28720   6.2083333
#> 10  -88.99875 17.17700 turtle2 2023-01-08 12:00:00 178.29051   2.0000000
#> 101 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 190.29051   5.0000000
#> 102 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 102.85252   6.0000000
#> 103 -88.99875 17.17700 turtle2 2023-01-08 12:00:00 225.98110   6.9791667
#> 11  -88.99889 17.17730 turtle2 2023-01-09 12:00:00  41.05082   1.0000000
#> 111 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 219.34133   3.0000000
#> 112 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 231.34133   6.0000000
```

```
#> 113 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 143.90334   7.0000000
#> 114 -88.99889 17.17730 turtle2 2023-01-09 12:00:00 267.03192   7.9791667
#>     time.diff sensitivity
#> 2           1        0.10
#> 3           1        0.10
#> 31          2        0.11
#> 6           1        0.10
#> 7           1        0.10
#> 71          2        0.11
#> 8           3        0.13
#> 81          4        0.16
#> 82          5        0.20
#> 9           6        0.25
#> 10          2        0.11
#> 101         5        0.20
#> 102         6        0.25
#> 103         7        0.31
#> 11          1        0.10
#> 111         3        0.13
#> 112         6        0.25
#> 113         7        0.31
#> 114         8        0.37
```

Plotting and summarizing dist.over.time() results

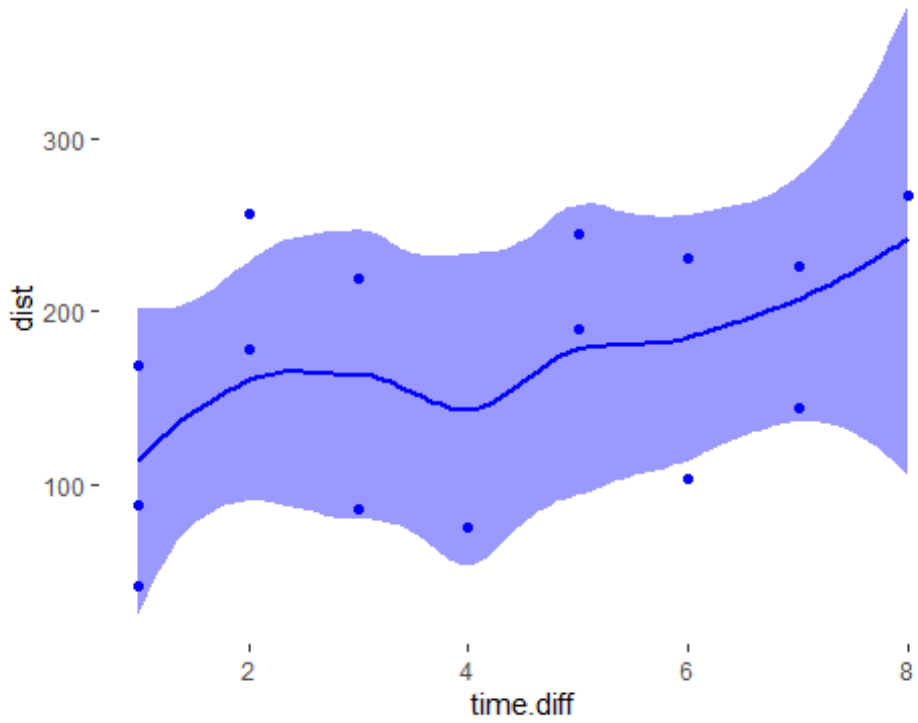
The results of this function can be highly informative for understanding animal movement patterns. At the most simple level, you can use functions like *group_by()* in the **dplyr** package or *ddply()* in the **plyr** package to calculate the average distance moved over given amounts of time.

You may also be able to use these data to assess things like seasonal differences in movement patterns. For example, you could get results for movement over 24 hours during two seasons, then run a mixed effects model with “dist” as your response, the season as your fixed effect, and “id” as your random effect. This would tell you whether the average distance moved over 24 hours differed between the seasons.

If you have data over many time points, you can also make excellent GAM visualizations using **ggplot2**

```
dist.res <- dist.over.time(data=network.1,
                coords=animal.points,
                lon.name="lon.raw",
                lat.name="lat.raw",
                id.name="id",
                date.time.name="date.time",
                units="days",
                time.diff=1,
                diff.max=NULL,
                sensitivity.min=0.1,
```

```
                sensitivity.max=0.2,
                sensitivity.change=0.02)
#> [1] "Calculating every 1 days from 1 days to 8 days"
#> [1] "Sensitivity starts at +/- 0.1 days and increases by 0.02 days every 1
days"
#> [1] "Analyzing individual turtle1"
#> [1] "Analyzing individual turtle2"

ggplot(dist.res[dist.res$id=="turtle2",],aes(x=time.diff, y=dist))+
  geom_point(color="blue")+
  geom_smooth(color="blue",fill="blue")+
  theme(panel.background=element_blank())
#> `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



This particular example is not that interesting, showing little or no change in distance moved over time (it is hypothetical and only over 8 days); however, with real data over longer time periods, interesting patterns often emerge. For example, a line that continues trending up suggests that an individual is not settling into one area and instead is continuing to move, with distance increasing as time increases.

In other cases, you may get a pattern where distance increases quickly for a few days, then largely levels off. This would suggest a daily activity pattern where an animal moves around quickly within a small area. So over a short time period the distance goes up quickly, but over longer times, the distance remains stable because it is always within the same

general area. So distance moved over 20 days is no different from distance moved over 40 days.

You can also get punctuated patterns, where there is little movement over many days, followed by a sudden increase in distance, suggesting that an animal was making small daily movements in a given area before rapidly transitioning to a new area.

Many other movement patterns are possible and should be interpreted in light of other available data, but these sorts of visualizations can help to understand animal movements and behaviors. Here again, comparison in different times of year, between sexes, etc. may be informative.

Troubleshooting and suggestions

Long run times

Some of these functions, especially *dist.over.time()* may take a long time to run. Here are some suggestions if you are having issues with run time

- Ensure that you have removed any tangential stream segments (i.e., segments that animals never used).
- Increase the *freq* argument in *prep.data()*. This will reduce the accuracy, so you have to weigh that against run time, but low *freq* values dramatically increase run time.
- Run the data in chunks (e.g., one individual at a time)
- Reduce the number of time intervals you are testing (in *dist.over.time()*)
- Prune your data to minimize repeated points within a time interval. As an example, if you are using automated receiving stations, and an animal sits near one of those stations for several days. You may wrack up thousands of data points, which will greatly reduce run time. Pruning that to one point per hour, for example, will greatly reduce run time without impacting your results (depending on your question, of course)
- Borrow a faster computer

Errors, warnings, and other issues

- Ensure that all your data are in the same projected format
- Plot your data to make sure your stream networks are correct and your points are within the network
- Double check that you have a node at each intersection of segments and each end of a segment.
- Double check that all your nodes are named as an integer and there are no duplicate names
- Double check that segments are designated correctly

- Double check that segments are named as an integer and there are no duplicate names
- Check for and remove NA entries in lat, lon, id, and date.time columns
- Make sure your date.time column is in the correct POSIX format

When running *prep.data()* you may get the following warning: "In increase.stream.points(l = l, freq = freq) : LINESTRING EMPTY for segment removing that segment because there are no coordinates in it"

This simply means that there is a segment ID with no geometry attached to it. This can easily happen when creating shapefiles in a program like QGIS and is usually not an issue. Plot the shapefile and ensure that all the segments that are supposed to be there are present. If they are, you can probably ignore this message, but it would be a good idea to also plot the "increased.line" object created by *prep.data()* to double check that things worked correctly.

When running *dist.over.time()* you may encounter the error: "Error in seq.default(sensitivity.min, sensitivity.max, sensitivity.change) : invalid '(to - from)/by'"

This means that your *sensitivity.max* value is smaller than your *sensitivity.min* value.

Pseudoreplication (multiple points per period)

Especially with the *dist.over.time()* function, you should be cautious about potential pseudoreplication. If, for example, you tracked an animal multiple times a day, then ran an analysis to look at movement over 24 hours, the function will take each point and try to pair it with another point 24 hours previously. The result is that you may get multiple points on one date matched with points on the previous date. This is arguably pseudoreplicated because the time span covered by each pair of points overlaps with the other pairs. Thus, those measurements of movement over 24 hours are not independent and actually repeat some of the same chunk of movement data.

To avoid this problem, you may need to filter your data before or after running *dist.over.time()* to remove multiple values for a given time interval ending on a given date (or hour or whatever your units are). Filtering before running the function will reduce run time, but could limit your matches unless tracking was done at very consistent times of day. Alternatively, filtering after running the function will take longer, but will maximize the usable data and will give the closest matches for the specified time interval (because when a single point pairs with several others for a given time interval, only the pair closest to the specified time interval is used).