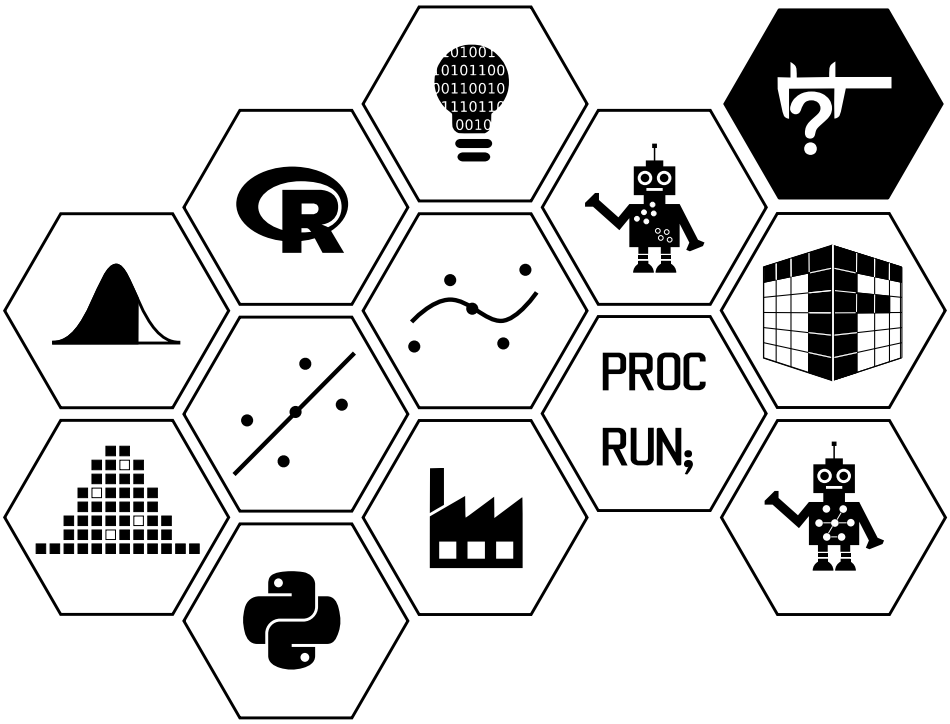
University of Glasgow

Vlad Vyshemirsky

Academic Year 2021-22

Individual Project:

Bayesian Navigation



DATA ANALYTICS GLASGOW

Problem Description

In this project you will build a fully autonomous artificial intelligence algorithm for estimating your location on a map using the position resection method.

Resection and its related method, intersection, are used in surveying as well as in general land navigation (including inshore marine navigation using shore-based landmarks). Both methods involve taking azimuths or bearings to two or more objects, then drawing lines of position along those recorded bearings or azimuths.

Consider the following example (see Figure 1): You are somewhere in the Firth of Clyde and you need to find your precise location. The visibility is good, so you take a compass and find bearings to three easily identifiable landmarks:

- Nardini's cafe in Largs (N55.796819, W4.868940), bearing is 83 degrees.
- The statue of Burns Highland Mary in Dunoon (N55.945681, W4.923315), bearing is 353 degrees.
- A house with a tower in Ascog (N55.826114, W5.022108), bearing is 286 degrees.

Following an instructional video made by the Royal Marines (while disregarding military jargon for angular measurements) you plot the obtained bearings on the map to produce a picture demonstrated in Figure 1. The intersection of three lines indicates your position.

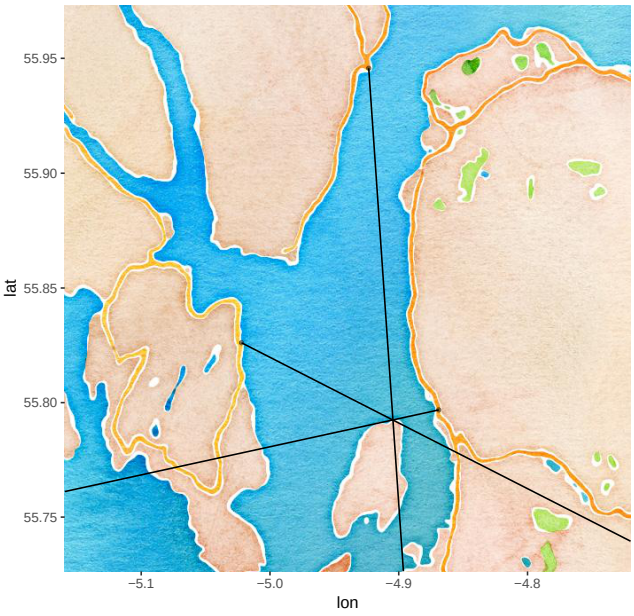


Figure 1: Example of position resection.

Well, the only problem is that the lines do not actually intersect in a single point, see Figure 2. This happens every time you do position resection in practice, and this result is due to errors in measuring angles. Even the most precise surveying equipment (we have no association with this website, just giving an example) inevitably introduces small errors in measurements. Plotting lines on the map is also not the most precise procedure. So, you would typically end up with a triangle formed by those three lines. As the Royal Marines sarge explained, "Your position should be within this triangle".

We want to be more precise about the position. We want to study the uncertainty of the result and obtain the complete distribution of plausible locations. So we build a statistical model for this problem. Let λ be the longitude and ϕ be the latitude for our location. We will treat them as random variables in a fully Bayesian setting. Our three bearings will be angles α, β, and γ, constant data, as we have just measured them. There may be dozens of factors that impact the quality of angle measurement using a compass, so it is fair to assume that the errors in angle measurement are normal. We will assume that measurements are unbiased, therefore the mean of the errors will be zero. The variance of these errors σ^2 is unknown, so we will treat it as an additional unknown, and infer it alongside our position.

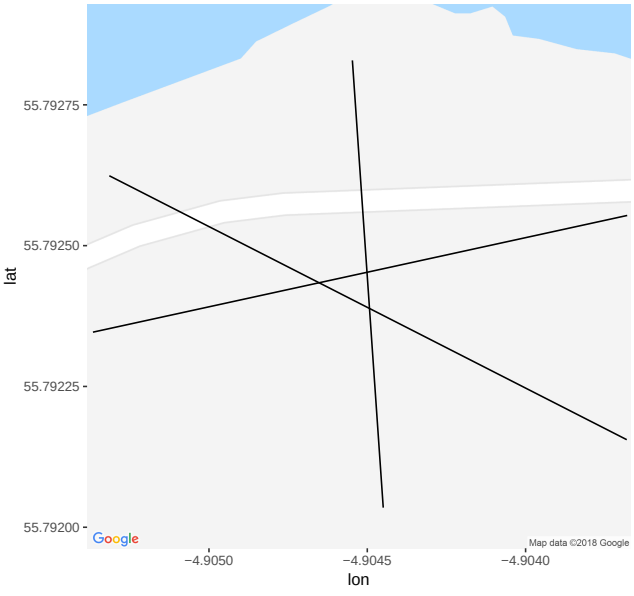


Figure 2: Zooming in on our location.

Given a pair of coordinates $P_1 = (\lambda_1, \phi_1)$ and $P_2 = (\lambda_2, \phi_2)$, we can calculate the bearing angle from point P_1 to point P_2 using the following formula:

$$B(P_1, P_2) = \\ = \left(360 + \arctan\left(\frac{\lambda_2 - \lambda_1}{\phi_2 - \phi_1}\right) \cdot \frac{180}{\pi}\right) \mod 360$$

where $\mod$ is a modulo operation.

This formula, of course, assumes that the Earth is flat. Which may be almost true when working on a small scale. When using radio bearings to remote radio beacons, the curvature of the Earth will begin to matter. In such cases you will need to take bearings along great circles. There is a package `geosphere` in R that can do that.

Assuming that the measurements of the three bearings α, β, and γ were made independently, we can now formulate the likelihood:

$$\begin{aligned} p(\alpha, \beta, \gamma | \lambda, \phi, \sigma^2) = & \mathcal{N}\left(\alpha | B((\lambda, \phi), (\lambda_1, \phi_1)), \sigma^2\right) \times \\ & \mathcal{N}\left(\beta | B((\lambda, \phi), (\lambda_2, \phi_2)), \sigma^2\right) \times \\ & \mathcal{N}\left(\gamma | B((\lambda, \phi), (\lambda_3, \phi_3)), \sigma^2\right) \end{aligned}$$

where $(\lambda_1, \phi_1) = (-4.868940, 55.796819)$ is the location of Nardini's cafe in Largs; $(\lambda_2, \phi_2) = (-4.923315, 55.945681)$ is the location of the statue in Dunoon; $(\lambda_3, \phi_3) = (-5.022108, 55.826114)$ is the location of the house in Ascog; $\alpha = 83$, $\beta = 353$, and $\gamma = 286$.

As the first step, we will assume a reasonably wide uniform prior for our location, and a shrinking prior for σ:

$$\begin{aligned} \lambda &\sim \text{Uniform}(-5.5, -4.5) \\ \phi &\sim \text{Uniform}(55, 56) \\ \sigma &\sim \text{Exponential}(20), \text{ i.e. the mean is } 1/20 \end{aligned}$$

The posterior $p(\lambda, \phi, \sigma^2 | \alpha, \beta, \gamma)$ can be evaluated in several ways. You can evaluate it over a fine grid, you can sample from it using the Metropolis-Hastings or the Sequential Importance Sampling algorithms. As the result you will obtain a plot similar to the one depicted in Figure 3. Note, that your most likely location is not in the centre of the triangle, but rather close to the upper corner. This is due to different distances from your location to different reference points. The closer you are to a reference, the smaller will be the location error corresponding to your angular error about the bearing to that reference.

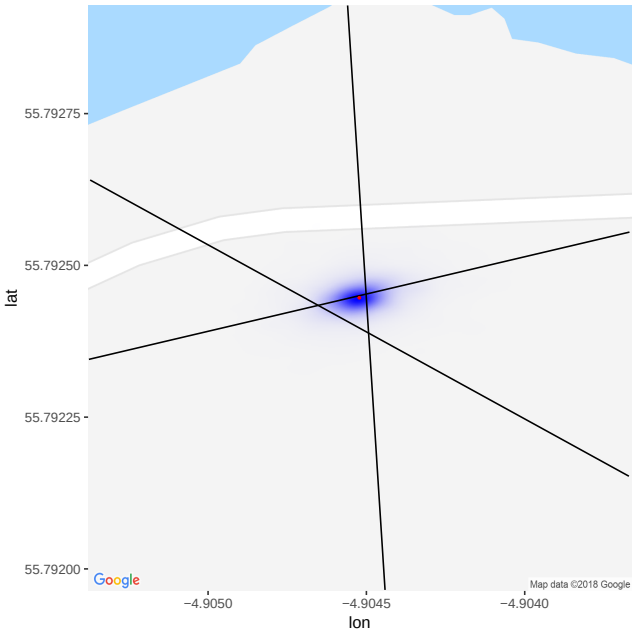


Figure 3: The posterior of the position assuming a wide uniform prior. The measurement error variance was marginalised out.

At the second stage of the study, we assume that we know that we are standing on a road. We will therefore introduce this information as a part of our prior, and observe how the inference result changes.

The prior for this case is going to be:

$$\begin{aligned} p(\lambda, \phi) &\propto \mathcal{N}\left(\rho(\lambda, \phi) | 0, 6^2\right) \\ \sigma &\sim \text{Exponential}(20) \end{aligned}$$

where $\rho(\lambda, \phi)$ is the shortest distance (in feet) to the middle of the nearest road estimated from the OpenStreetMap database. *Do not worry, we will give you an R function to calculate this distance later in this project description.*

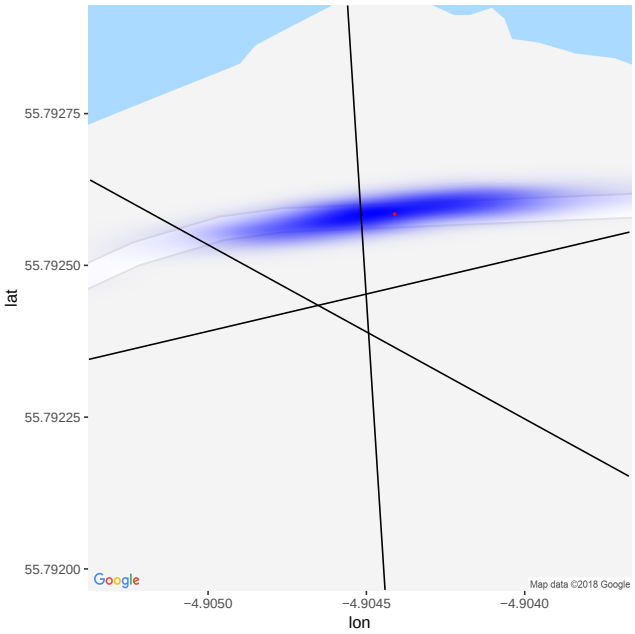


Figure 4: The posterior of the position assuming proximity to a road. The measurement error variance was again marginalised out.

After a reasonably slow calculation (should take about 30 minutes) we arrive to the posterior that accounts for proximity to a road, as depicted in Figure 4.

Project Tasks

For your own project you need to

1. Collect similar data. You can pick three reference points on an online map, and get their coordinates. Then you can either calculate bearing angles to your location to create a simulated data set, or take a compass and perform real measurements. *Extra brownie points to those who will actually measure bearings*.
2. Implement one of the inference approaches you are familiar with (not necessarily the Metropolis-Hasting sampler, you can choose any other method) to calculate the posteriors of your location using a wide uniform prior, and then assuming proximity to a road.
3. Write a report describing your work, and discussing your observations and experience.

Software Requirements

You have freedom to implement the solution for the project in any computational environment you prefer. There are certainly good helper libraries for R, Python, C, and Java.

We will, however, provide some hints assuming that you are using R.

You may need to install the following R packages to perform tasks required in this project: `MASS`, `coda`, `reshape2`, `ggplot2`, `ggmap`, `mvtnorm`, `osmar`, and `geosphere`.

Hints

We understand that some of you don't have experience of working with map data. We therefore provide examples how to perform key mapping tasks required in this project.

Registering for Google Maps API

To plot maps in R using Google Maps as a data source, you need to register for Google Maps API access. This should normally be free for the amount of data we will be using in this project. Go to https://cloud.google.com/maps-platform/ to register. Access to map data can be performed using `ggmap` package in R. Once you have registered with Google and obtained your API key, you can activate it in R with the following commands:

```
require(ggmap)
register_google(key="YOUR_KEY_GOES_HERE",write=TRUE)
```

Plotting a Map

The `ggmap` package provides all the functions required to plot beautiful maps. This package fetches images from online mapping providers such as Google Maps, OpenStreetMap, or Stamen. To plot the basic map of, say, the Firth of Clyde, you need to run the following code:

```
require(ggmap)
map <- get_map(c(-4.94,55.85),zoom=11,maptype="road")
ggmap(map)
```

where $(-4.94, 55.85)$ is the longitude and the latitude of the point in the centre of your map, and zoom level is a parameter of how detailed your map needs to be. Finally, `maptype` parameter defines what type of map you are looking for, see the help page for the get_map function to see possible choices. Function `ggmap` draws the map as a `ggplot` object, therefore any additional drawing should be done using the ggplot2 framework.

Adding Reference Points and Bearing on the Map

Once you have the basic map, you may want to plot on top of it. Let's first create the objects that we want to plot. A data set for the locations of the reference points can be created using:

```
landmarks<-data.frame(lon=c(-4.868940,-4.923315, -5.022108),lat=c(55.796819, 55.945681,55.826114))
alpha <- 83 # First bearing
beta <- 353 # Second bearing
gamma <-286 # Third bearing
```

Now we need to define a data set for the bearing lines:

```
d <- seq(0,0.4,0.0001) # Length of the line
line1 <- data.frame(lon=landmarks[1,1] + d*sin(alpha*pi/180+pi),
                    lat=landmarks[1,2] + d*cos(alpha*pi/180+pi))
line2 <- data.frame(lon=landmarks[2,1] + d*sin(beta*pi/180+pi),
                    lat=landmarks[2,2] + d*cos(beta*pi/180+pi))
line3 <- data.frame(lon=landmarks[3,1] + d*sin(gamma*pi/180+pi),
                    lat=landmarks[3,2] + d*cos(gamma*pi/180+pi))
```

And finally draw the reference points and the bearing lines on the map using ggplot:

```
map <- get_map(c(-4.94,55.85),zoom=11,maptype="watercolor")
mapPlot <- ggmap(map)+
 geom_point(aes(x = lon, y = lat), size = 1, data = landmarks, alpha = .5) +
 geom_line(aes(x=lon,y=lat),data=line1) +
 geom_line(aes(x=lon,y=lat),data=line2) +
 geom_line(aes(x=lon,y=lat),data=line3)

mapPlot
```

That will produce the plot displayed in Figure 1.

Bearing Line Intersections

If you are implementing the Metropolis-Hastings algorithm, it may struggle to converge fast enough if initialised randomly from the prior. It may be worth trying to initialise it somewhere near the expected mode of the posterior.

You can pick one of the corners of the bearing line triangle as a general location around which you can randomise your initial samples.

The following function calculates the coordinates for the intersection of two bearing lines:

```
intersectBearings <- function(p1,b1,p2,b2) {
    x1 <- p1[1]
    x2 <- p1[1] + 0.1*sin(b1*pi/180)
    x3 <- p2[1]
    x4 <- p2[1] + 0.1*sin(b2*pi/180)
    y1 <- p1[2]
    y2 <- p1[2] + 0.1*cos(b1*pi/180)
    y3 <- p2[2]
    y4 <- p2[2] + 0.1*cos(b2*pi/180)
    x <- ((x1*y2-y1*x2)*(x3-x4)-(x1-x2)*(x3*y4-y3*x4))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-x4))
    y <- ((x1*y2-y1*x2)*(y3-y4)-(y1-y2)*(x3*y4-y3*x4))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-x4))
    return(as.numeric(c(x,y)))
}
```

We can use it to calculate the point at which the first and the second bearings intersect:

```
intersectBearings(landmarks[1,],alpha,landmarks[2,],beta)
```

```
[1] -4.904501 55.792453
```

Now it may be a good idea to initialise your Metropolis sampler around that point:

```
intersection <- intersectBearings(landmarks[1,],alpha,landmarks[2,],beta)
draws <- array(0,dim=c(4000,3,3))
draws[4000,1,] <- runif(3,intersection[1] - 0.01, intersection[1] + 0.01)
draws[4000,2,] <- runif(3,intersection[2] - 0.01, intersection[2] + 0.01)
draws[4000,3,] <- rexp(3,20)
```

Proposals in Metropolis-Hastings

If you are implementing the Metropolis-Hastings algorithm, it may be tricky to achieve convergence. A good idea to find a suitable proposal distribution is to run a pilot chain with arbitrary proposal, collect the samples from this chain, and then use the covariance of those samples as the variance-covariance matrix for the multivariate normal proposal.

In the example given in this project description the following proposal worked reasonably well:

```
prop.cov <- c(1e-8,1e-8,1e-4)*diag(3)
proposed <- rmvnorm(1,draws[step-1,,chain],prop.cov)
```

It may be a good idea to plot your chains as they are being sampled, to observe what's going on with the algorithm. You can add the following lines in the end of the burn-in loop:

```
while (!converged) {
  draws[1,,] <- draws[4000,,]
  accepted <- 1
  for (step in 2:4000) {
    for (chain in 1:3) {
       . # Sampler goes here
       .
       .
    }
  }
  print(sprintf("Acceptance rate: %f",accepted/120))
  chainlist <- mcmc.list(Chain1=mcmc(draws[,,1]),
                         Chain2=mcmc(draws[,,2]),
                         Chain3=mcmc(draws[,,3]))
  converged <- all((gelman.diag(chainlist)$psrf[,2])<1.2)
  plot(chainlist) # This plots current state of the chains
  Sys.sleep(0.1)
}
```

If you still struggle to make the algorithm converge, feel free to disregard the Gelman-Rubin diagnostics and rely on visual inspection of the chains.

Distance to the Nearest Road

You will need to evaluate the distance from an arbitrary location to the nearest road. We can do that by fetching road information from the OpenStreetMap database first in the beginning of your script:

```
require(osmar)
require(geosphere)
intersection <- intersectBearings(landmarks[1,],alpha,landmarks[2,],beta)

roads.box <- center_bbox(center_lon=intersection[1],
                         center_lat=intersection[2],
                         width=100,
                         height=100)
api <- osmsource_api(url="https://api.openstreetmap.org/api/0.6/")
roads <- get_osm(roads.box, source=api)
ways <- find(roads, way(tags(k=="highway")))
ways <- find_down(roads, way(ways))
ways <- subset(roads, ids=ways)
hw_lines <- as_sp(ways,"lines")
```

The above code fetches information about all the roads within approximately 164 feet (50 meters) of the intersection of the first and the second bearing lines. The width and the hight of the box for the roads are 100 meters in this example.

If the precision of your bearing measurement is poor, you may need to consider a larger box.

And then you can define the distance function (in feet to the nearest road):

```
distanceToRoad <- function(par) {
  distance <- dist2Line(c(par[1],par[2]), hw_lines)
  proper.distance <- distance[1] * 3.28084
  return(proper.distance)
}
```

Plotting the Density Estimate on the Map

Finally, you may need to visualise your posteriors on the map. Assuming that they are stored in an array called `sample`, you can call `kde2d` for density estimation and then plot the result:

```
require(MASS)
require(reshape2)
D <- kde2d(as.vector(sample[,1,]),as.vector(sample[,2,]),
```

```
          h=c(sd(sample[,1,]),sd(sample[,2,])),
          n=1024,
          lims=c(-4.9055,-4.9035,55.7915,55.7935)) # Enough to cover map
z <- melt(D$z)
z$Var1<-D$x[z$Var1]
z$Var2<-D$y[z$Var2]

map <- get_map(c(mean(sample[,1,]),mean(sample[,2,])),zoom=19,maptype="road")
mapPoints <- ggmap(map)+
 geom_point(aes(x = lon, y = lat), size = 1, data = landmarks, alpha = .5) +
 geom_raster(data=z,aes(x=Var1,y=Var2,fill=value))+
 guides(fill=FALSE,alpha=FALSE)+
 scale_fill_gradientn(colours=c("#0000FF00","#0000FFFF"))+
 coord_cartesian() +
 geom_line(aes(x=lon,y=lat),data=line1) +
 geom_line(aes(x=lon,y=lat),data=line2) +
 geom_line(aes(x=lon,y=lat),data=line3)+
 geom_point(aes(x=lon,y=lat),
            data=data.frame(lon=mean(sample[,1,]),lat=mean(sample[,2,])),
            size=0.5,colour="#FF0000")

mapPoints
```