

A map of Singapore with a network overlay of nodes and edges. The nodes are represented by circles of varying sizes and colors (blue, grey, orange). The edges are lines connecting these nodes. The map shows geographical features like the island, water bodies, and urban areas. Labels on the map include 'Pasir Gudang', 'dar Puteri', 'Jurong Island', 'Southwest', 'Singapore', and 'Southe'. A legend in the bottom right corner shows a color scale for values ranging from 0 to 400.

Functional Programming & Parallelization in Spatial Point Pattern Analysis

Clara Chua



Hi there!

- ◎ **Project:** Geospatial Analysis of Airbnb in Singapore
- ◎ **Topic:** Functional Programming & Parallelization in Spatial Point Pattern Analysis (SPPA)
- ◎ **Dataset:** InsideAirbnb, September 2020



For this project...

- ◎ Determine if Airbnb listings are clustered in Singapore and where they occur using Ripley's K-function test
 - Split by room type / region

- ◎ Packages used:

Spatial Data

- spatstat
- sf
- maptools
- tmap
- tidyverse

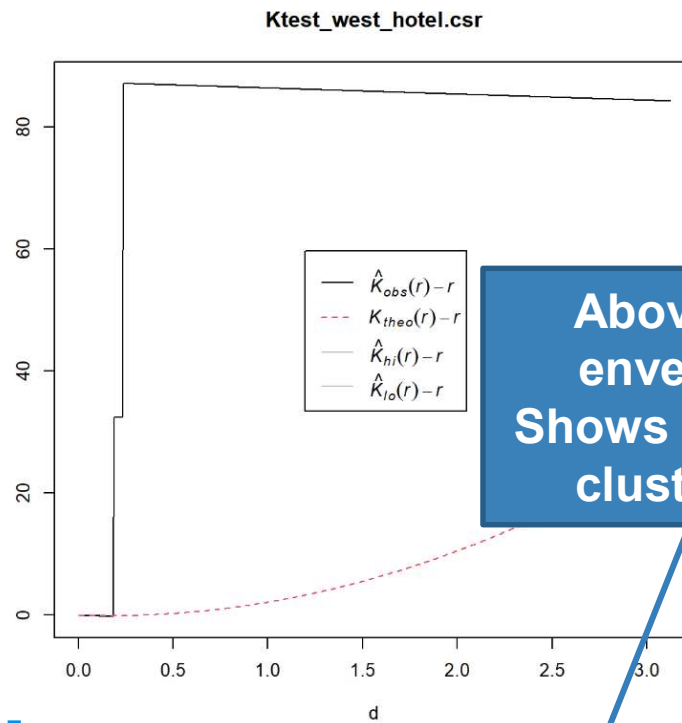
Parallelization

- foreach
- doParallel

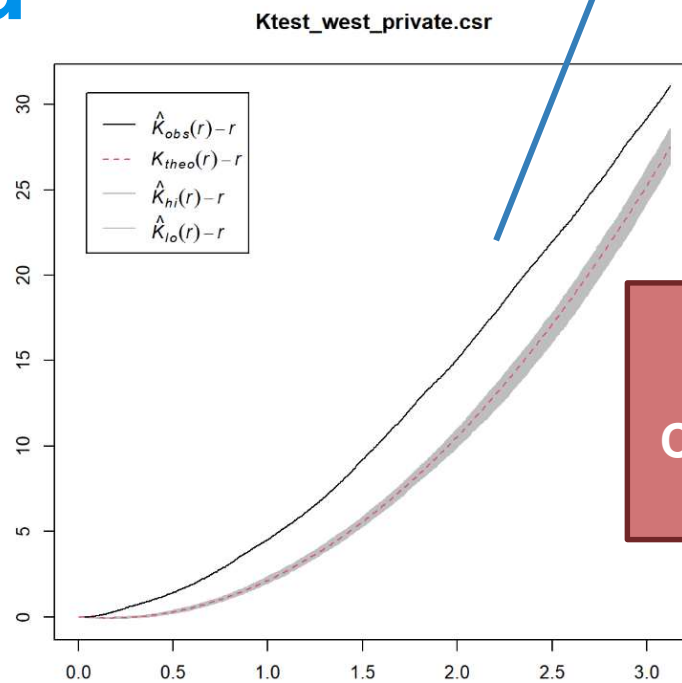
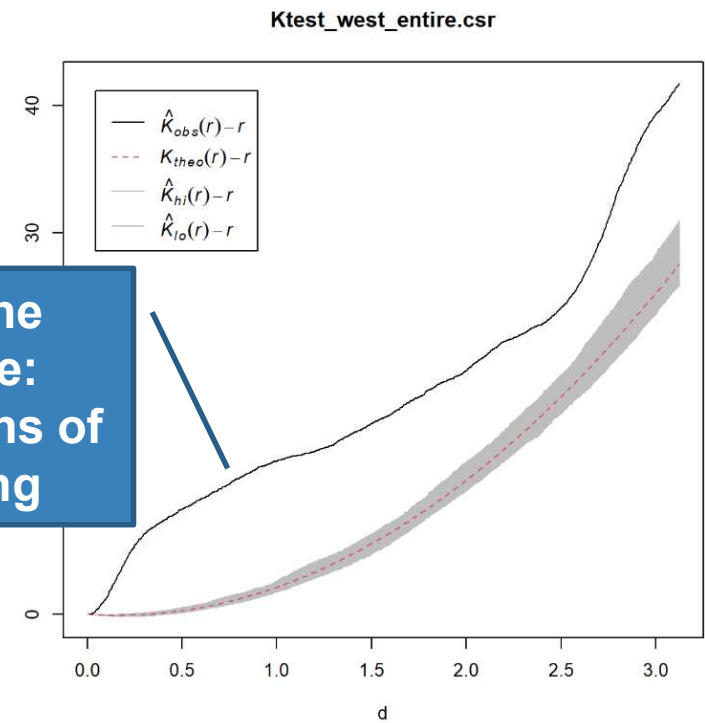
How does the K-Test work?

- ◎ spatstat provides a function `Kest()` to compute an unbiased estimate of $K(t)$
- ◎ Run a series of Monte Carlo simulations using the `envelope()` function to compare it to the theoretical value.
- ◎ H_0 : Complete Spatial Randomness
- ◎ H_1 : Not
 - **Clustering**: simulated values fall higher than the upper bounds of the theoretical values.
 - **Competitive processes**: simulated values lower than the lower bound of theoretical values

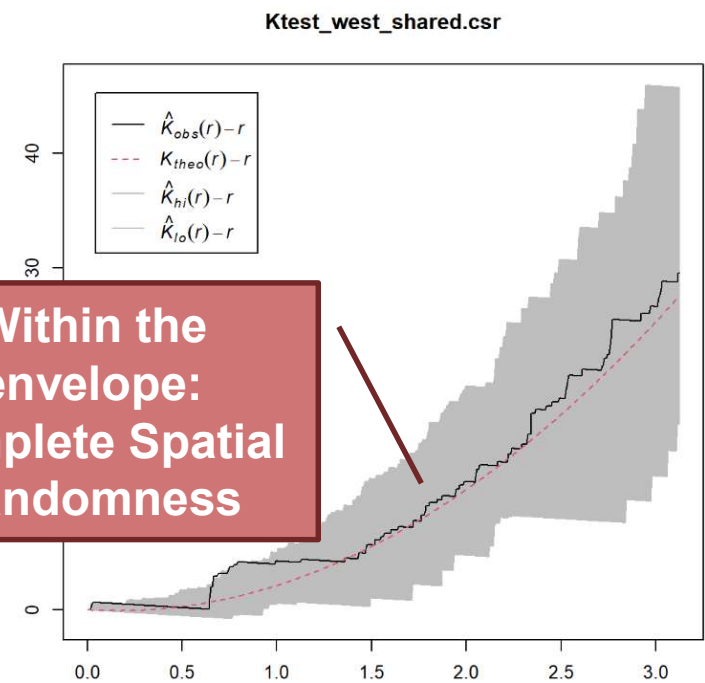
Graphs produced



Above the envelope:
Shows signs of clustering



Within the envelope:
Complete Spatial Randomness



Issues

1. Computationally intensive for large datasets (~7,000 points) or data with non-uniform observation windows.
2. No in-built parallelization methods in spatstat. Parallelisation is also dependent on OS – functions such as ``mclapply`` from the base parallel package works for Mac and Linux, but not for Windows.

What we did

- ◎ Split the analysis by the 5 different subregions (East, West, Central, North and Northwest) and room types (Private room, Entire home/apt, Hotel room, shared room)
- ◎ Create functions that benefit from functional programming:
 - ``envelope()`` function over the different subregions, and for different room types.
 - plot the envelope results for the K-tests.





Parallelization (1) Set up Clusters

Parallelisation is done by splitting the number of simulations between the different cores, then combining the results back together.

```
# Set up clusters  
# Set up number of clusters (6 in this case - 2 less than max no. of cores)  
nclus <- detectCores() - 2  
  
# Make the clusters and register doParallel with the number of clusters  
cl <- makeCluster(nclus)  
registerDoParallel(cl)
```



Parallelization – (2) Set up simulations and distribute objects

```
# Set up minimum total number of simulations for parallel function to split evenly between them (may be slightly more than 100)
nsims = 100
```

```
# Distribute the necessary R Objects
clusterExport(cl, c('ppp_store2', 'nsims', 'nclus'))
clusterEvalQ(cl, library(tidyverse, spatstat))
```

```
## [[1]]
## [1] "forcats" "stringr" "dplyr" "purrr" "readr" "tidyr"
## [7] "tibble" "ggplot2" "tidyverse" "stats" "graphics" "grDevices"
## [13] "utils" "datasets" "methods" "base"
##
## [[2]]
## [1] "forcats" "stringr" "dplyr" "purrr" "readr" "tidyr"
## [7] "tibble" "ggplot2" "tidyverse" "stats" "graphics" "grDevices"
## [13] "utils" "datasets" "methods" "base"
##
## [[3]]
## [1] "forcats" "stringr" "dplyr" "purrr" "readr" "tidyr"
## [7] "tibble" "ggplot2" "tidyverse" "stats" "graphics" "grDevices"
## [13] "utils" "datasets" "methods" "base"
##
```

Parallelization – (3) Set up helper functions

```
# Set up functions to be used in the parallelisation

# Function to replicate the ppp object for each core in use
rep_ppp <- function(pppobj) {
  replicate(nclus, pppobj, simplify=FALSE)
}

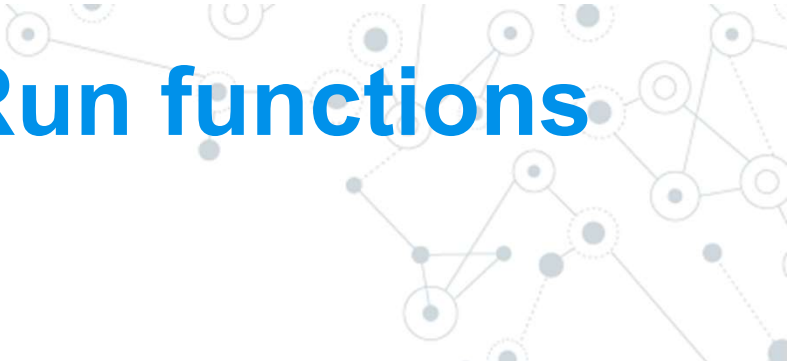
# Function to run an envelope simulation for 1 ppp object. This results in a list of envelopes.
runKobjpar <- function(ppp_obj) {
  ppp_par <- rep_ppp(ppp_obj)
  foreach(i=1:length(ppp_par)) %dopar% {
    set.seed(123)
    spatstat::envelope(ppp_par[[i]], spatstat::Kest, nsim=ceiling(nsim/nclus), savefuns = TRUE, rank=1)
  }
}

# Function to pool envelope simulation results into 1 envelope
poolpar <- function(x) {
  do.call(pool, x)
}
```

Parallelization – (4) Set up functions

```
runKregionpar <- function(ppp_region) {  
  reg_name = paste(deparse(substitute(ppp_region)) %>% gsub("\\$", " ", .) %>% word(., -1))  
  # create temporary variable removing 'all' from the region list and any room types that are zero  
  temp_ppp = ppp_region %>% discard(., names(ppp_region)=="all") %>% discard(., map(., "n") ==0)  
  foreach(i=1:length(temp_ppp)) %do% {  
    tempname <- names(temp_ppp)[i]  
    envtemp[[tempname]] <- runKobjpar(temp_ppp[[i]])  
  }  
  Ktestcsr3[[reg_name]] <- map(envtemp, poolpar)  
}
```

Parallelization – (3) Run functions & Stop Clusters



```
# Run selected region  
ptm <- proc.time()  
runKregionpar(ppp_store2$west)  
proc.time() - ptm
```

```
# Stop Cluster (Not run)  
# stopCluster(cl)
```



Parallelization is 4x faster

Used microbenchmark and ran simulation 10 times

Region	expr	min	mean	median	max	
East	forloop	657.55	658.07	658.00	658.54	4.40
	functional prog	657.46	658.06	658.03	658.70	
	parallelization	149.31	149.73	149.61	150.46	
North	forloop	129.95	130.80	130.67	132.57	4.23
	functional prog	130.45	131.05	130.87	133.11	
	parallelization	30.73	30.98	30.92	31.29	
Northeast	forloop	235.78	236.69	236.77	237.57	4.39
	functional prog	235.95	236.72	236.78	237.21	
	parallelization	53.80	54.10	53.98	54.92	
West	forloop	1015.69	1021.08	1021.67	1022.50	4.40
	functional prog	1021.03	1021.56	1021.36	1023.59	
	parallelization	231.50	232.80	231.96	237.07	

Alternative: Kest-fft()

The Kest.fft() function is an alternative to analyse large pattern of points.

Kest() computes the distance between each pair of points analytically and would therefore take a long time in computing large datasets ($N-1$ points * No. of simulations).

Kest.fft() function discretises the point pattern onto a rectangular raster and applies Fast Fourier Transform (FFT) techniques to estimate the K-function.

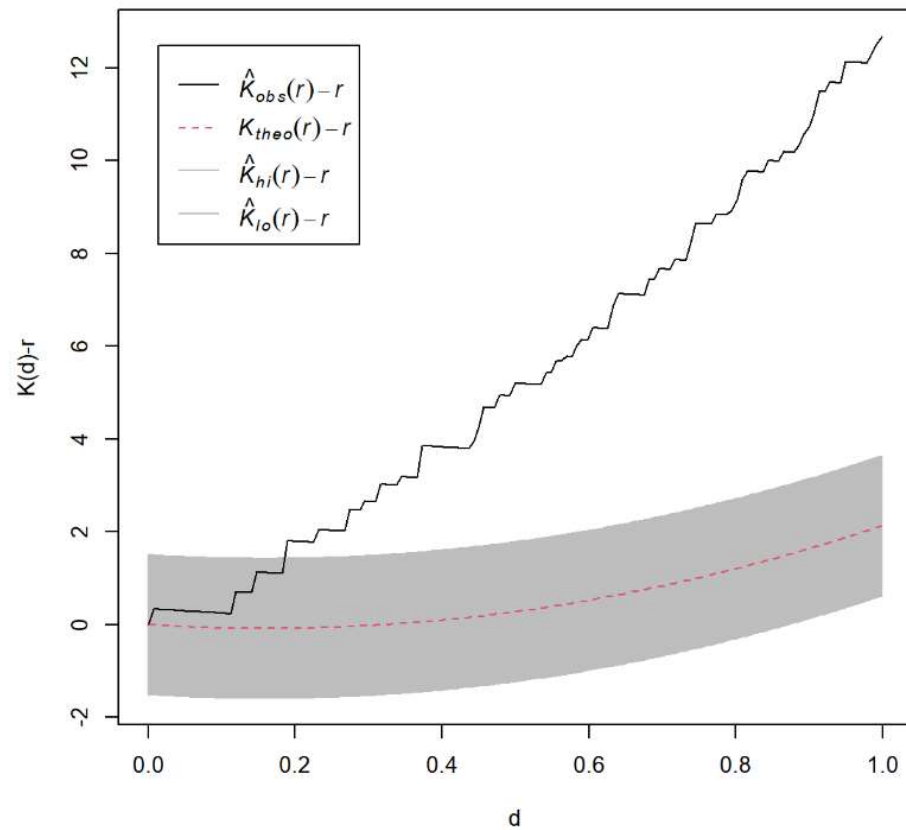
In essence, the Fast Fourier Transform algorithm allows the estimates to be computed in parallel, by splitting them pairwise, and aggregating them again.

Caveat: This only applies to the K-function test, so hopefully the last few slides weren't a complete waste of your time.

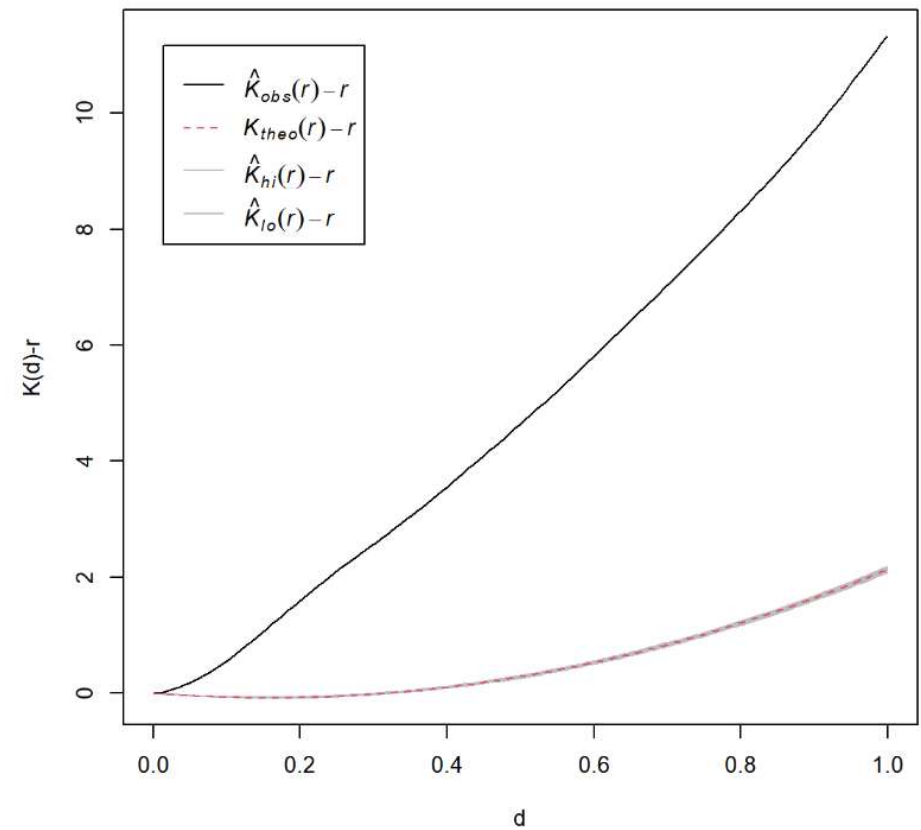
Kest-fft() Results



Kfft_cen_private.csr



Ktest_central_private.csr



Takeaways

1. Learn parallelisation so you can apply it to your problems
2. Learn how to set up a VM with better hardware specs
3. RTFM – there might be a function written specifically for your problem



Thank You!

RPubs : <https://rpubs.com/clarachua/airbnbA1>

 : <https://clarachua.netlify.app/projects/>