Transitioning from `sp/raster` to `sf/stars/terra`

Richard E. Plant
April 18, 2022
Additional topic to accompany *Spatial Data Analysis in Ecology and Agriculture using R, Second Edition*
http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm
Chapter and section references are contained in that text, which is referred to as SDA2.

1. Introduction

No one ever said that R worries much about backward compatibility. If you have been programming in R for very long, you have probably seen at one time or another the word *deprecated*. This usually happens when you are running some piece of code that you wrote five years ago and you no longer remember the details exactly, and it means that you have to dig back and figure them out, and then figure out how to adapt the code to whatever change was made by the package maintainer. Users of `sp` and related packages are, if they are not prepared, in for such an experience. If you have loaded the `maptools` package recently, you have seen the following message:

```
> library(maptools)
Please note that 'maptools' will be retired by the end of 2023,
plan transition at your earliest convenience;
some functionality will be moved to 'sp'.
```

Although some functionality will be moved to `sp`, the recommendation is that users transition from `sp` to `sf`. This is also discussed in a post on the R-Bloggers website. In addition, the popular `raster` package is being replaced by `terra`. Finally, despite the existence of the excellent `raster` and `terra` packages, the developers of `sf` have seen fit to create their own package, called `stars`, that includes a raster component. In order to keep SDA2 up to date, I have rewritten the SDA2 code so thtat all of the relevant spatial analysis coding is moved from `sp/raster` to `sf/stars/terra`. The hope is that, if and when in the future you run some `sp` code and see the dreaded word "deprecated," you will be able to use these examples to adapt to the new packages.

Although the change affects both figures and calculations, the primary effect is on the latter. I have included two copies of each relevant figure, one made with the original code and one made with the new code. As you will see if you compare the figures, the new ones are sometimes not identical to the old ones. I would like to turn this post into a conversation, and those of you using this document who come up with improvements on my modificatios should feel free to contact me at replant@ucdavis.edu and tell me about them. In particular, I ran out of patience trying to figure out how to set the margins with the `sf` version of `plot()` and with controlling the y-axis with `terra`. I would very much welcome hearing about other improvements in the code.

The accompanying R code is in a folder rcode.New. All of the original code is included. For those R code files in which a modification has been made, the original code is in a file *xx original.r* and new code is in a file *xx.r*, where *xx* is the Section number. If for a given section there is no file *xx original.r* then no change in that code file was necessary.

2. General information

Here is some general information:

An introduction to the `terra` package is here: https://rspatial.org/terra/spatial/index.html.
An introduction to the `sf` package is here: https://r-spatial.github.io/sf/articles.
A pointer to sites describing the `stars` package is here: https://r-spatial.github.io/stars/.

The following two tables give the functions for conversion from one package to another. The tables are copied from a blog posted on the R-bloggers website.

Conversion of vector data

From/to	`sf`	`sp`	`terra`
`sf`		`as(x, "Spatial")`	`vect()`
`sp`	`st_as_sf()`		`vect()`
`terra`	`st_as_sf()`	`as(x, "Spatial")`	

Conversion of raster data

From/to	`raster`	`terra`	`stars`
`raster`		`rast()`	`st_as_stars()`
`terra`	`raster()`		`st_as_stars()`
`stars`	`raster()`	`as(x, "Raster") + rast()`	

Other conversions
To convert point data from `stars` raster to `sf` points use `st_as_sf(…, as_points = TRUE)` See Sec. 5.2.1.

The website https://github.com/r-spatial/sf/wiki/Migrating gives a list of `sp` functions and commands and the corresponding `sf` functions and commands.

There is a collection of similar `sf` functions involving type conversion, including `st_sf()`, `st_sfc()`, `st_as_sfc()`, and `st_as_sf()`, and I find it hard to remember which does what. Here is a table to help with that.

Function	Use	Example
`st_as_sf()`	Convert a data frame to an `sf` point object Convert a map object to an `sf` polygon object	1.2.r, line 27 1.3.1 line 7; 1.3.3, line 6
`st_as_sfc()`	Convert a `spatstat` object to an `sfc` geometry object	1.2.r, line 103, 3.6.1,line 45
`st_sf()`	Convert an `sfc` geometry object into an `sf` object	1.2.r, line 104; 2,4,3, line 40; 3.6.1, line 46
`st_sfc()`	Convert an "`XY point`" object made from `st_point()` into an `sfc` object	1.1.r, line 78

Here is a table of conversions. I have in the last column listed examples of the use of the new (`sf`, `terra`, or `stars`) function and the write up describing these examples contains some further information.

Old function	Package	New Function	Package	Example
`as(`*`polygon`*`,)`	`sp`	`st_as_sfc(`*`thsn poly`*`),` `st_sf()`	`sf`	1.2, 3.6, 4.5.1, 5.3.2
`as(`*`points`*`,)`	`sp`	`st_as_sf()`	`sf`	4.5.2
`bbox()`	`sp`	`st_bbox()`	`sf`	5.2.1

brick()	raster	rast()	terra	2.4.3
cell2nb()	spdep	not.cell2nb()	Homemade	3.2.1
coordinates(*.shp)<- xx	sp	st_read()	sf	1.3.3, 4.6.1
coordinates(*.csv)<- xx		st_as_sf(,coords=)	sf	1.1, 4.6.1
coordinates()	sp	st_coordinates(xx)	sf	1.3.3
expand.grid(), coordinates()	sp	rast(), crs()		6.4.3
extent()	raster	ext()	terra	7.4
gridded()	maptools	make_grid()	stars	2.4.3
image()	rgdal	rast()	terra	1.1
invIrM()	spatialreg	not.invIrM()	Homemade	3.2.1
layout.scale.bar()	sp	Homemade		1.3.1
layout.north.arrow()	sp	Homemade		1.3.1
map2SpatialPolygons()	sp	st_as_sf()	sf	1.3.1
over()	sp	st_intersects()	sf	5.3.4, 7.3, 11.3.2 (see * below)
plot()	sp	plot()	sf	1.1, 1.3.1, 1.3.3
plot()	sp	plot()	terra	1.1
plot()	raster	plot()	terra	2.4.3
plot()	raster	ggplot()+geom_stars()	ggplot2	2.4.3
plot()	Sp	tm_shape()	tmap	3.2.1
Polygons()	sp	st_point(), st_sfc()	sf	1.1
proj4string()	sp	st_crs()	sf	1.1
projection(), CRS()	raster	crs()	terra	2.4.3
spplot()	sp	ggplot()	ggplot2	1.1, 1.2, 1.3.1
spplot()	sp	ggplot()	ggplot2	1.1, 1.2, 1.3.1
spplot()	sp	levelplot()	rastervis	
spplot()	sp	levelplot()	lattice	3.5.3
spplot()	sp	plot()	terra	4.3
SpatialPolygons()	sp	make_grid(), st_sfc()	stars, starsExtra	1.1
SpatialPolygonsDataFrame()	sp	st_sf()	sf	3.6.1
SpatialPolygonsDataFrame()	sp	vect()	terra	1.2
spTransform()	sp	st_transform()	sf	1.3.1

* The `sf` package contains a number of "binary predicate" functions analogous to the `sp` function `over()`. These are described in http://127.0.0.1:24113/library/sf/html/geos_binary_pred.html.

3. Converstions

Chapter 1

Probably the most common conversion of the code in this text involves reading a csv file and then converting into the appropriate object type. The code to create an `sp` object looks like this.

```
data.Set2 <- read.csv("set2\\set2data.csv", header = TRUE)
coordinates(data.Set2) <- c("Longitude", "Latitude")
proj4string(data.Set2) <- CRS("+proj=longlat +datum=WGS84")
```

The key operations are the successive applications of the functions `read.csv()`, `coordinates()`, and `proj4string()` to read the data set, identify the data fields that contain the coordinate information, and assign the projection. This set of steps is accomplished using `sf` objects as

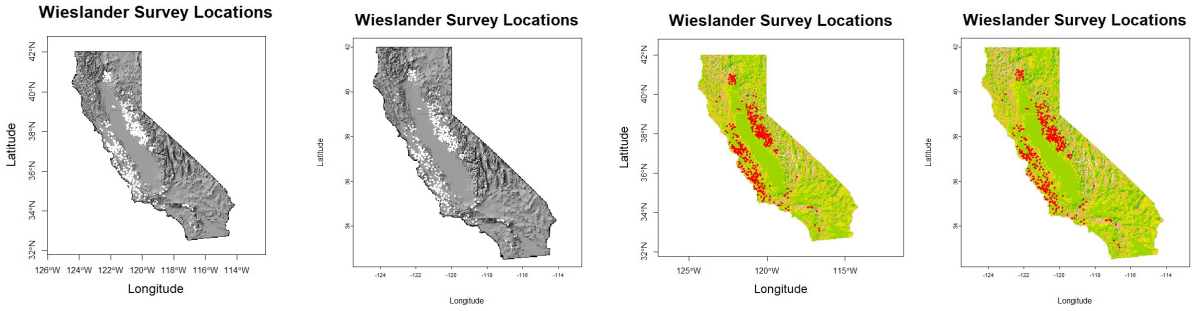
```
> data.Set2 <- read.csv("set2\\set2data.csv", header = TRUE)
> Set2.plot.sf <- st_as_sf(data.Set2, coords = c("Longitude", "Latitude"))
> st_crs(data.Set2.sf) <- 4326  # WGS 84
```

I will try to consistently indicate when something is an `sf` object by appending `.sf` to the name.

Here is a chapter by chapter discussion of changes in the R code. Most but not all of the changes involve the plotting of figures, but I have tried to also list those chapters and sections involving a simple replacement of data types.

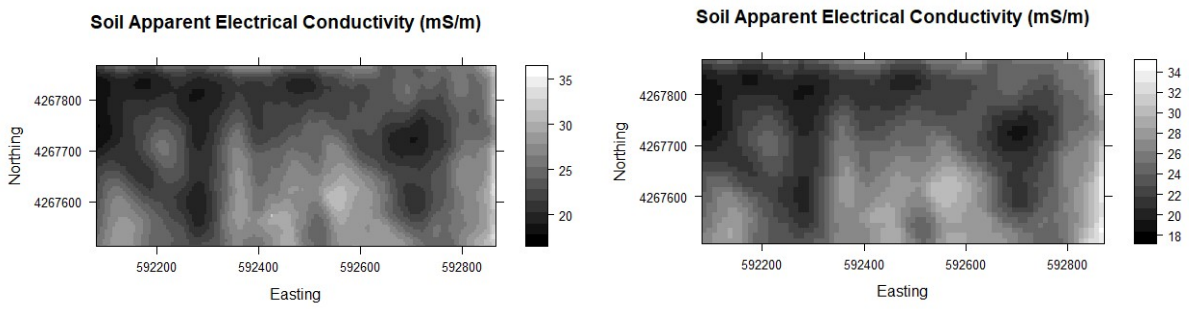
Section 1.1

Fig. 1.1



This is an overlay of a dot plot of locations onto a grid map. The original uses the `sp` functions `image()` and `sp::plot()`. The grid map exists on disk as a GeoTiff file. In the original the grid map is first read using `rgdal::readGDAL()`, and coordinates are assigned using `maptools::proj4string()`. The locations are stored in a csv file which is read using the base function `read.csv()`, and converted to a `SpatialPointsDataFrame` using the function `maptools::coordinates()`. The grid map is first plotted using the base graphics function `image()` and then the locations are plotted using `sp::plot()`. The new version uses `terra::rast()` to read the grid map. The function `image()` does not recognize the `terra` object, so the new version plots it using `terra::plot()`, and the coordinates are set using `terra::crs()`. Note that the `xlab` and `ylab` assignments have to be put in the initial call to `terra::plot()`.
After the California map is plotted using `terra::plot()`, the sample locations are read in, converted to an `sf` object, and added to the `terra` plot using the `sf` plotting function.

Fig. 1.2a

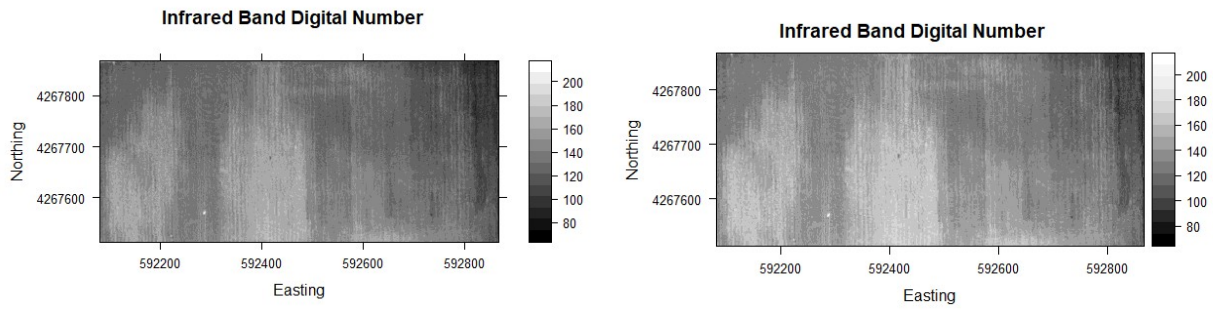


The original uses `gstat::krige()` to compute an interpolation surface. The function `krige()` is a wrapper function for `gstat::gstat()` and therefore we need to use `sf` and `stars` functions as input, since these are the ones that `gstat()` will accept. The creation of the grid to hold the data for `krige()` is based on information in https://rdrr.io/cran/starsExtra/man/make_grid.html.
Plotting in the original is done using `maptools::spplot()`, which is itself a wrapper for the function `lattice::levelplot`. We can use a `ggplot2` function to plot a quick check, but I decided to create my own little wrapper `not.spplot()` for the final plot. The source code for `rasterVis::levelplot()` is given in https://github.com/oscarperpinan/rastervis/blob/master/R/levelplot.R. I used this information to create the wrapper. Within the function `not.spplot()`, the `terra` object is converted into the form used by `levelplot()` using the base function `expand.grid()`, whose output is a data frame of the grid locations, and creating a set of *z* values from the `terra` object using ideas from https://rspatial.org/terra/spatial/4-rasterdata.html.
I ran into an interesting problem here that I had never seen before and is worth discussing. My code has the following `library()` calls:

```
> library(stars)
> library(starsExtra)
```

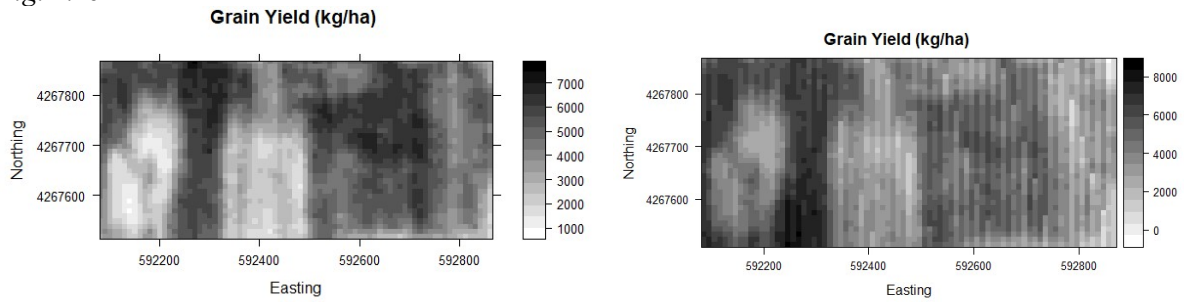
When I tried to run the code, I kept getting an error message saying that there was no package `starsExtra`. A bit of sleuthing (more than I was in the mood for) led to the discovery that my version of `stars` was 1.7 and `starsExtra` required version 1.8 or it would not load. Once I updated `stars` everything went smoothly. If something like this happens to you, be forewarned.

Fig. 1.2b



This displays a cropped GeoTiff image.The original used the function `sp::over()` to do the cropping and `maptools::spplot()` for the display. In the new version the cropping is done using the function `terra::crop()` based on information in https://rspatial.org/terra/rs/2-exploration.html#image-information-and-statistics. As an alternative to what was done with Fig. 1.2a, the display this time uses the function `rasterVis::levelplot()` directly.

Fig. 1.2c



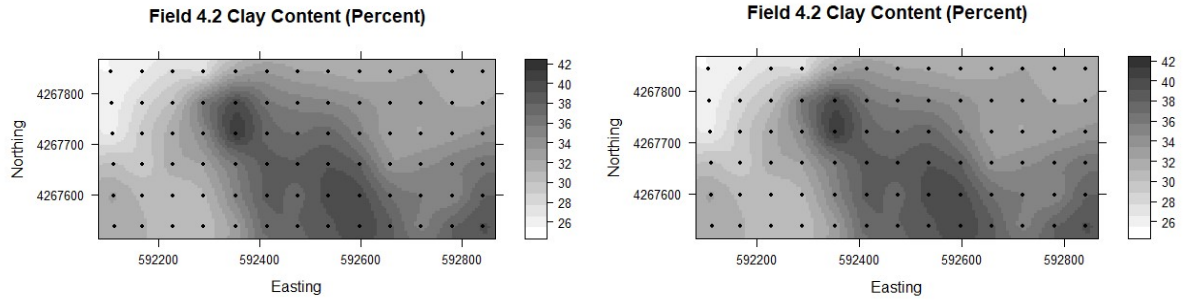
This figure uses `gstat::krige()` to compute an interpolation surface that has been detrended using linear regression to create a trend surface. The creation of a `terra` object with values to input into `krige()` is based on information from https://rspatial.org/terra/spatial/8-rastermanip.html.
The trend is added back into the kriged yield by using the fact that the output of `krige()` is a list in which the first element is `var1.pred`. Use the function `str()` to see this. The `rasterVis` function was used as was done with Fig. 1.2b. To make the grayscale setting the same as the original I used the argument `col.regions` as follows,

```
> greys <- grey(255:0 / 255)
> # par.settings = GrTheme causes a greyscale print
> rasterVis::levelplot(Yield.krig.ter, layers = 1, margin = FALSE,
+   par.settings = GrTheme, xlab = "Easting", ylab = "Northing",
+   main = "Grain Yield (kg/ha)", col.regions = greys)
```

This use of `col.regions` was based on a guess as it is not very clearly explained in the documentation.

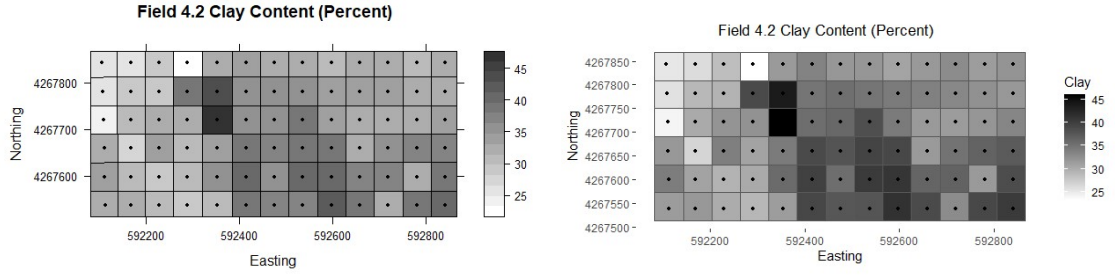
Section 1.2

Fig. 1.3a



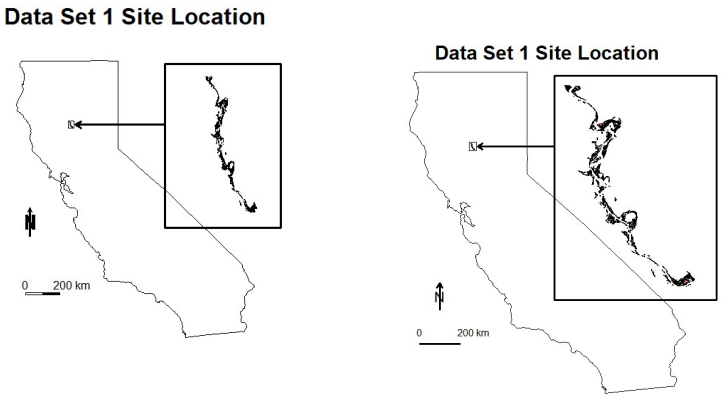
Kriging is carried out as with Fig. 1.2a. Points are added to `latticeExtra::levelplot()` based on the discussion in https://stackoverflow.com/questions/58477129/add-single-points-to-levelplot. I used my homemade "wrapper function" `not.spplot()` as in Fig. 1.2a.

Fig. 1.3b



This one is a bit of a kludge. The creation of Thiessen polygons is based on https://github.com/r-spatial/sf/issues/1233 and presents no problem. The result is a polygonal vector object. The function `levelplot()` works with raster data, I could not figure out how to get the correct grayscale in `sf::plot()`, and I could not figure out how to control the length of the y-axis using `terra::plot()`, so I ended up using `ggplot()`.

Section 1.3.1
Fig. 1.4

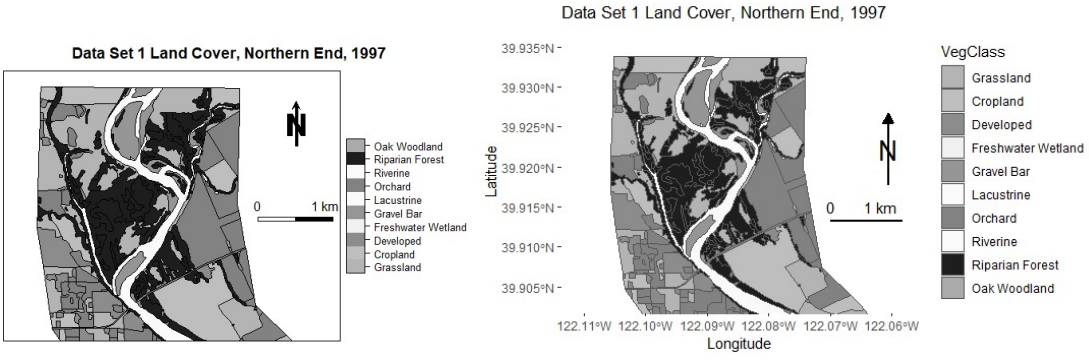


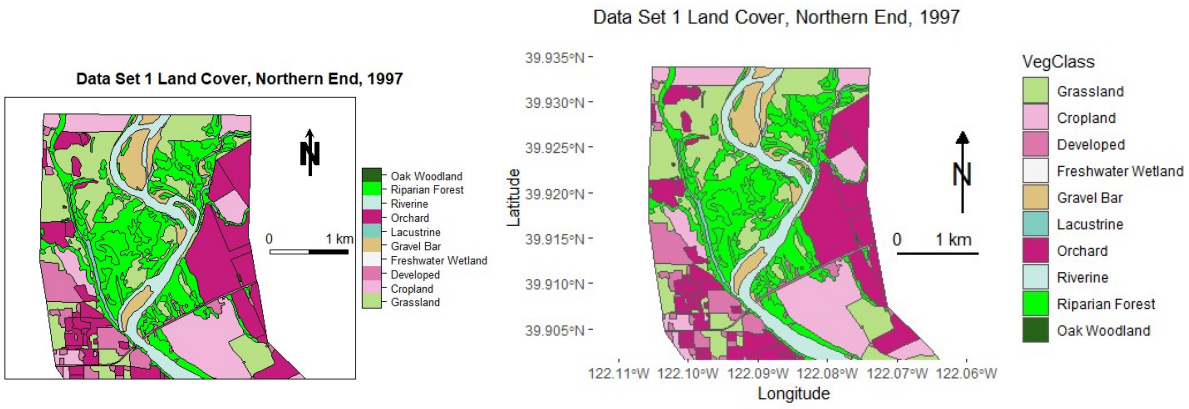
This was fairly straightforward, as the calls to `sp::plot()` for the most part transferred directly to calls to `sf::plot()`. The inlaid map came out a little bigger, which I decided actually looked better, so I made the surrounding box bigger to accommodate it. I made my own scale bar and north arrow. After the last line of code, you will get a warning.

```
> plot(data.Set1.wgs, add = TRUE, axes = FALSE)
Warning message:
In plot.sf(data.Set1.wgs, add = TRUE, axes = FALSE) :
  ignoring all but the first attribute
```

This is caused by a property built into the `sf::plot()` function that is supposed to be a feature but sometimes acts as a bug. If no data field of an `sf` object is specified, the function will attempt to plot the first nine fields. In this case that cannot be done because the material is being added to an existing plot, so a warning is posted. Ordinarily this will be corrected by plotting the `st_geometry` of the `sf` object, but for some reason that did not work in this case and I did not have the time or patience to track down the reason. We will see many examples, however, of this workaround.

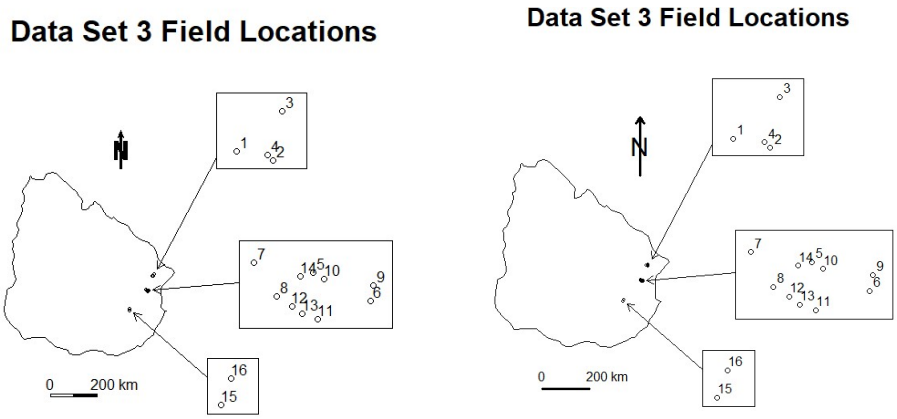
Fig. 1.5





I again did this using `ggplot()`. For a discrete valued quantity like land cover one uses `scale_fill_manual`, whereas for a continuous valued quantity one would use `scale_fill_gradient`. I tried to use the `ggsn` package http://oswaldosantos.github.io/ggsn/ to add a scale bar and north arrow, but was unsuccessful. This may be in part because `ggplot()` badly wants the x and y coordinates to be longitude and latitude and I am using UTM coordinates. In the same way, `ggplot()` badly wants to label the axes, so I had to figure out what in the code the labels correspond to. The idea of the color choices is that "natural" vegetation is various shades of green and "unnatural" vegetation is various shades of red.

Section 1.3.3
Fig. 1.8



This was a relatively easy transition based on Figs. 1.3a and 1.4. Again, I made my own north arrow and scale bar.

Chapter 2

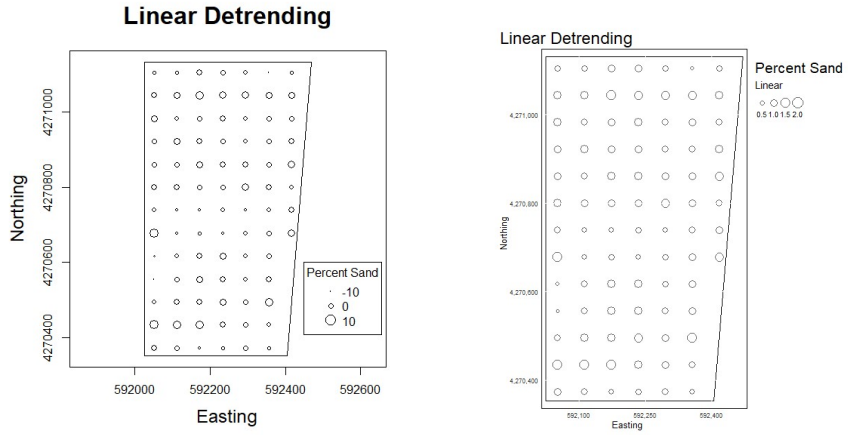
The only section in this chapter requiring modification is 2.4.3

Section 2.4.3
File 2.4.3.r

These plots are not pictured in the text. Two versions are given in the new code:

Chapter 3

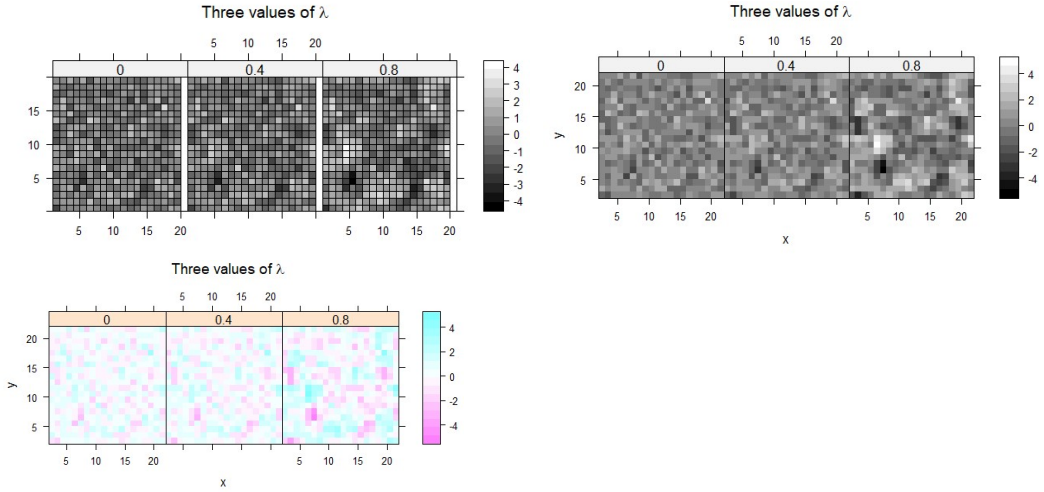
Section 3.2.1



This section uses the `spdep` function `cell2nb()` and the function `invIrM()`, which is now in the `spatialreg` package. The transition of `cell2nb()` to `sf` is discussed here. The package `nngeo` contains a function `st_nn()` that returns the nearest neighbors of grid points and I used this to create a simple function `not.cell2nb()`. I created a simple function `not.invIrM()` using the base functions `diag()` and `solve()`. However, there is no indication that either of these functions will be deprecated any time soon, so I included them as well.
The original bubble plots are made using `maptools::plot()`. The new bubble plots are made using `tmap`. This is a relatively new package that has many useful features and works will with `sf` objects.

Section 3.5.3

Fig. 3.9



I made two versions of this file. The first uses the function `not.cell2nb()` created in the code of Section 3.2.1. The call to `spplot()` used to created Fig. 3.9 is replaced by a direct call to the `lattice` function `levelplot()`. The second uses the old functions cell2nb() and invIrM(). It is not clear whether the packages with these functions will continue to be supported.

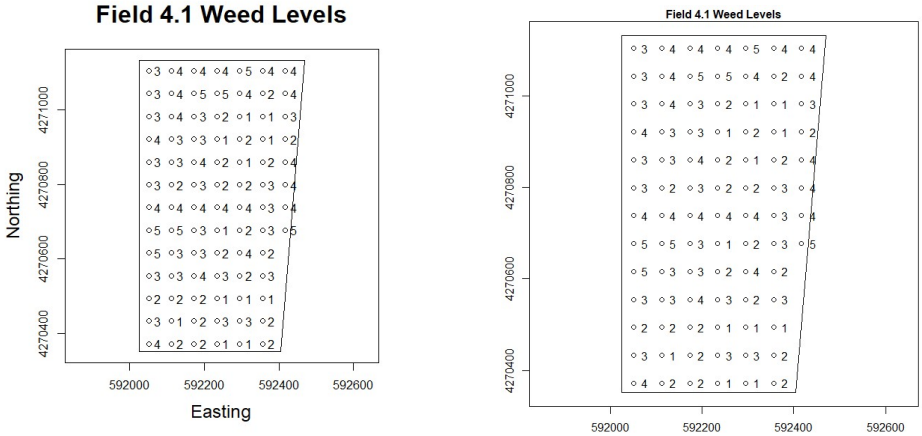
Section 3.6.1

The function `dnearneigh()` in the package `spdep` accepts `sf` objects and so I still used it.

Chapter 4

Section 4.2.1

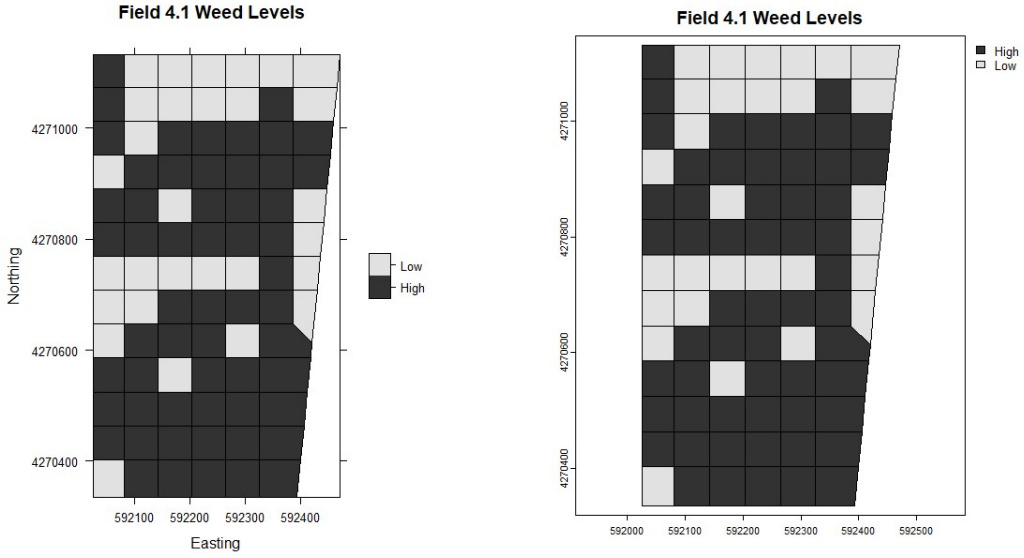
Fig. 4.1



Only Fig. 4.1a is shown since the procedure to construct Fig. 4.1b is identical. This uses a call to the `sf plot()` function. I could not get the `par()` function to affect the margin settings. Some discussion on the web suggests that some plotting functions do not access externally set parameters, but putting the statement inside the call to `plot()` caused an error message. This is one of those instances where one uses the function `st_geometry()` inside `plot()` to avoid getting a warning message.

Section 4.3

Fig. 4.2



This plot was made using terra::plot(). Here the problem of lack of control of the spatial extent of the vertical dimension was not an issue. There does not seem to be a way to include axis labels in the plot.

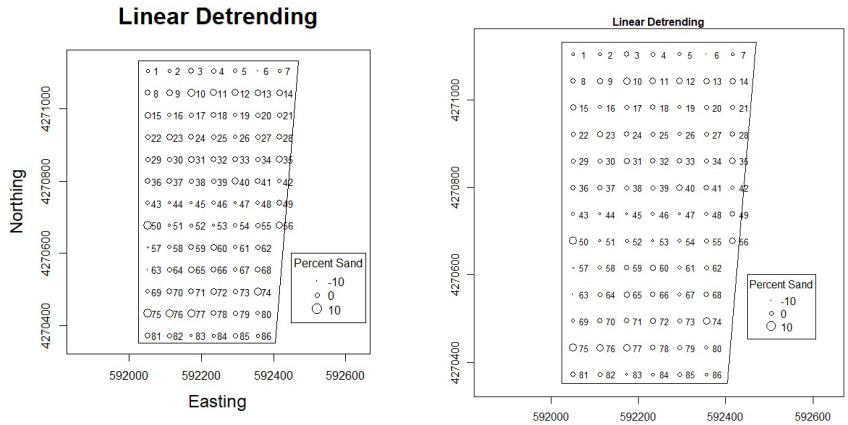
The function `joincount.test()` used in this section is as of this writing still housed in the `spdep` package.

Section 4.2
The functions in this section require the `spdep` package.

Section 4.5.1
This section shows another example of the conversion of a Thiessen polygon to an `sf` object.

Section 4.5.2
This section shows an example of the conversion of a data frame to an `sf` object. It appears that the object returned by the function `moran.plot()` no longer can be made to display the infinite rows.

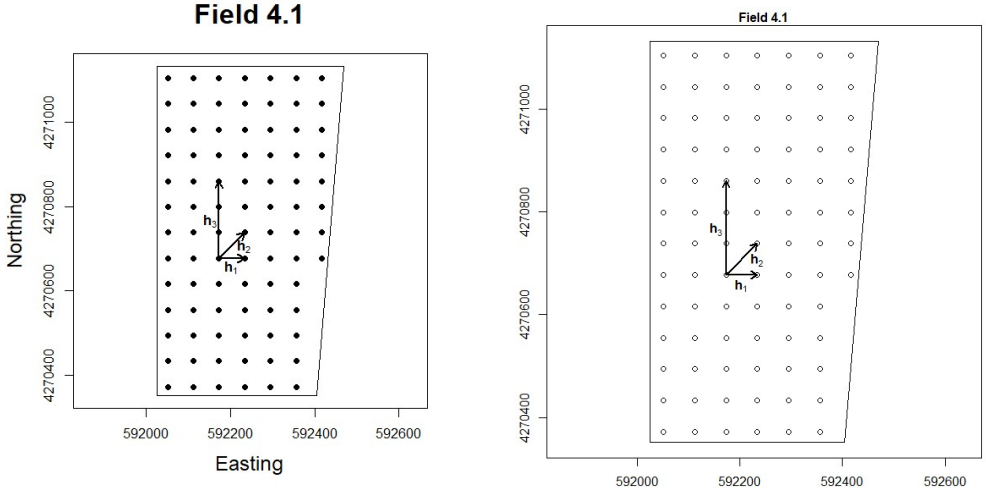
Section 4.5.3
Fig. 4.5



Only Fig. 4.5a is shown. This procedure is identical to that of Sec. 4.2.1.

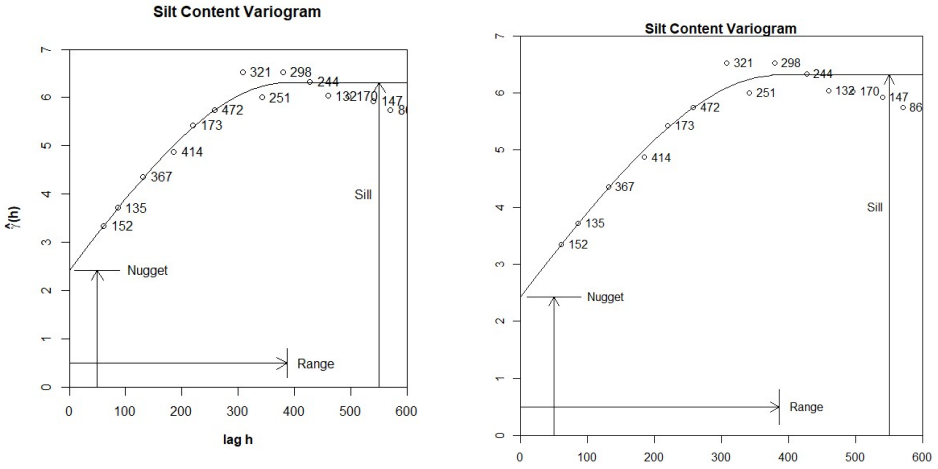
Section 4.5.4
Reprogramming this section takes advantage of the fact that the functions `grw.sel()` and `gwr()` accept a data frame as well as a `Spatial` object.

Section 4.6.1
Fig. 4.6



I changed to open circles, which actually think look better. This is one of those instances where one uses the function `st_geometry()` inside `plot()` to avoid getting a warning message.

Fig. 4.7a
This is essentially the same as Fig. 4.5.



Notice that the use of the sf function plot for Fig. 4.6 affects Fig. 4.7a.
Fig. 4.7b

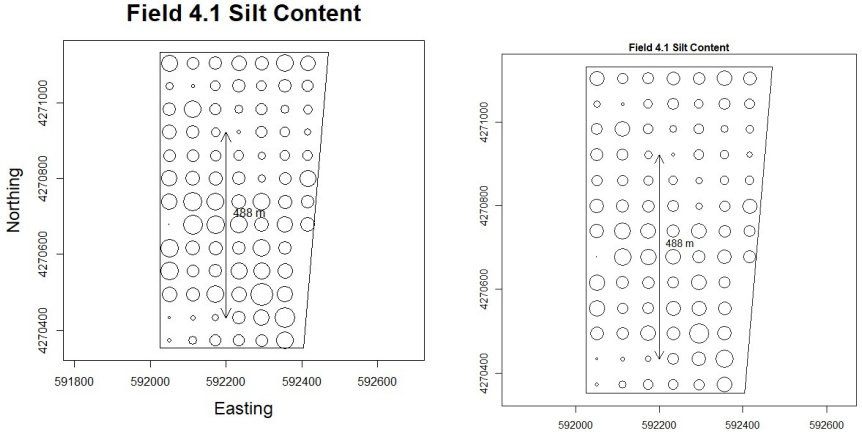
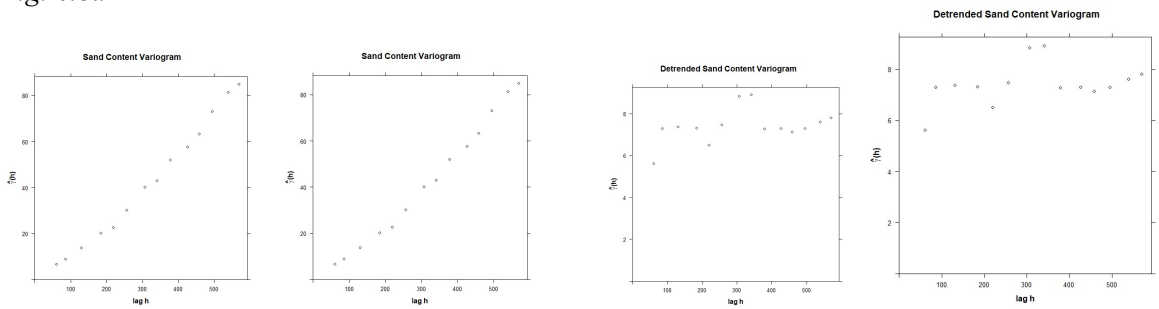


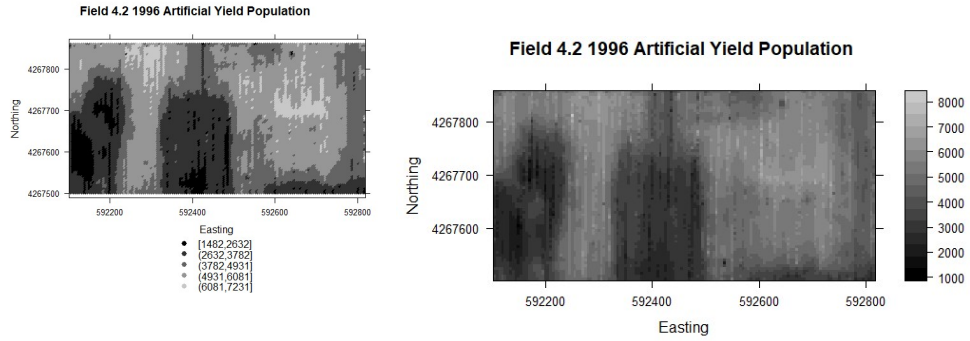
Fig. 4.8a



Chapter 5

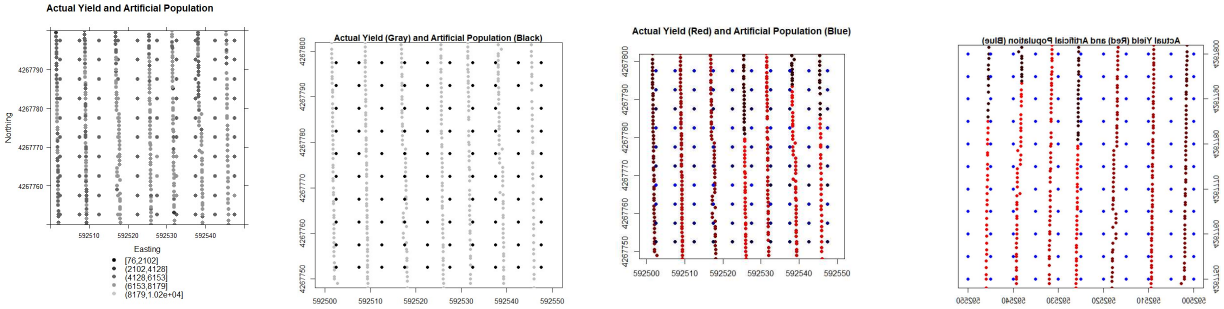
Section 5.2.1

Fig. 5.1a



This is based entirely on Fig. 1.3b of Section 1.2 and uses the homemade function `not.spplot()`. I like the right side legend more than the one in the old figure.

Fig. 5.1b



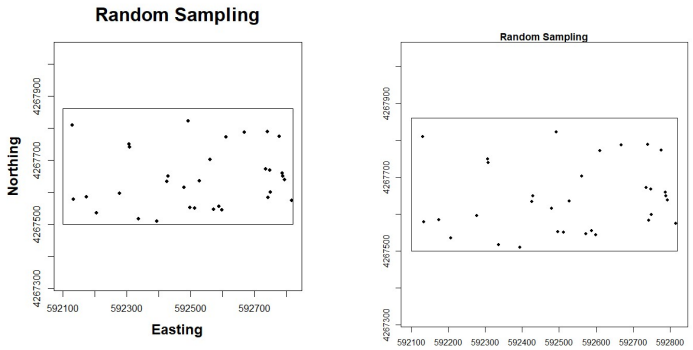
Both the black and white and the color version use `plot()`. I dd not bother with making the grayscale in the black and white version since it did not seem to convey much additional information.

Section 5.2

This section does not use any `sp` based functions.

Section 5.3.1

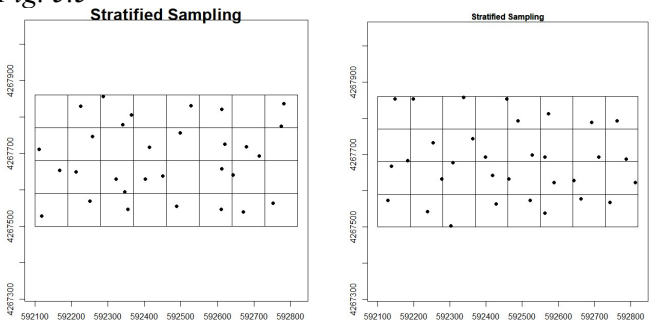
Fig. 5.4



These figures require only minor code changes.

Section 5.3.2

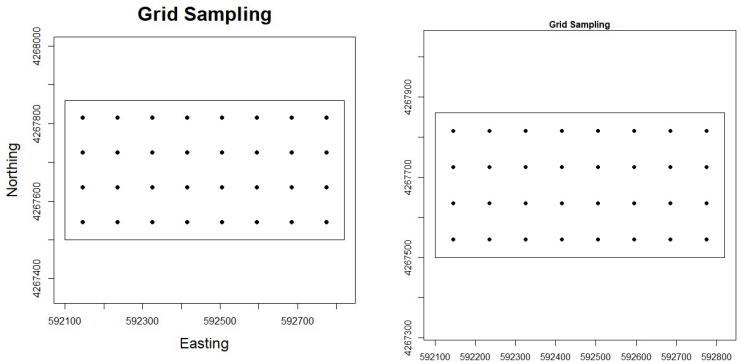
Fig. 5.5



The original code required the use of the `sp` function `GridTopology()`, which does not have an analog in `sf`. Moreover, the `sf` function `st_sample()` at this writing does not include stratified sampling directly. Although some options may be possible, it is relatively easy to construct Thiessen polygons and use these to direct a stratified sampling plan, so that is what I did. Note that in the original stratified sampling plan some cells got two locations and some none. In this sense the revised code is an improvement.

Section 5.3.3

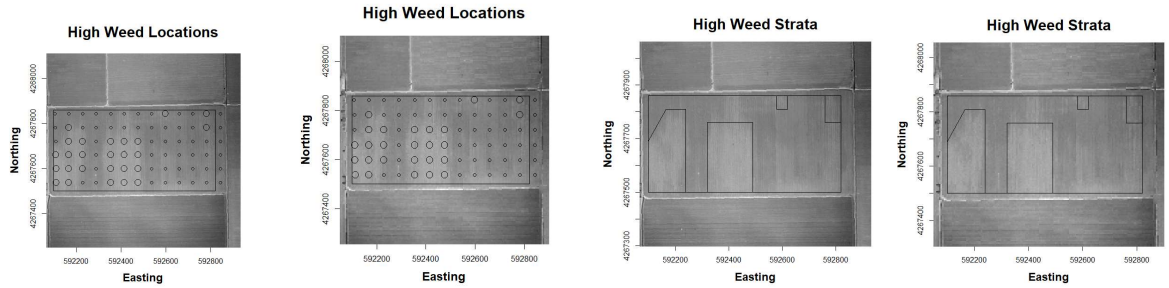
Fig. .5.6

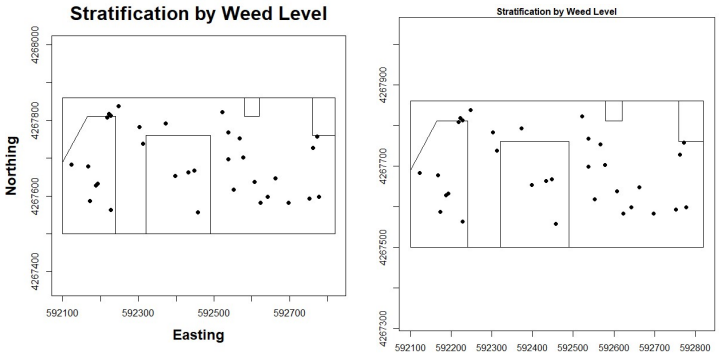


This is handled identically to Fig. 5.5 in Section 5.3.2.

Sec. 5.3.4

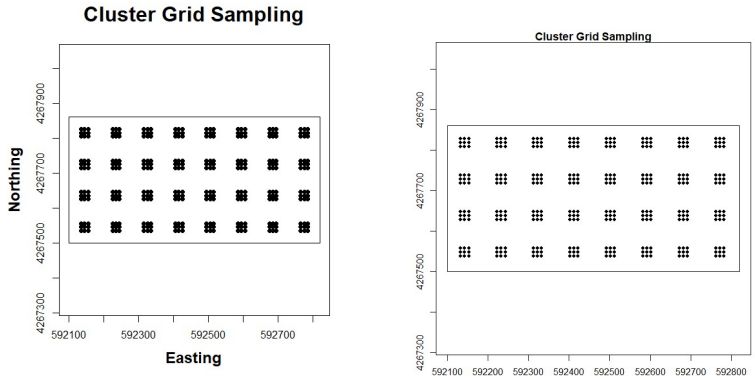
Figs 5.7 and 5.8





I used `terra` to plot the images. I had trouble getting the four individual polygons demartacting the high weed zones to plot as a unit and ran out of patience so I just plotted each one individually. The function `st_intersects()` replaces the `sp` function `over()`.

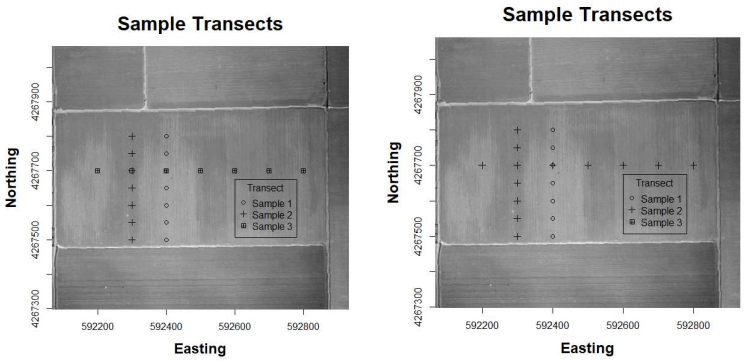
Section 5.3.5
Fig. 5.10



This is handled in exactly the same way as the other similar figures in this chapter.

Section 5.4
The code for the figures is unaltered. Changing the file is a simple matter of replacing `sp` based code with `sf` based code.

Section 5.7
Fig. 5.13



This was handled in exactly the same way as Fig. 5.7. I cheated a little bit in that I could not figure out how to use the `terra` function `extract()` so I did it with `raster`.

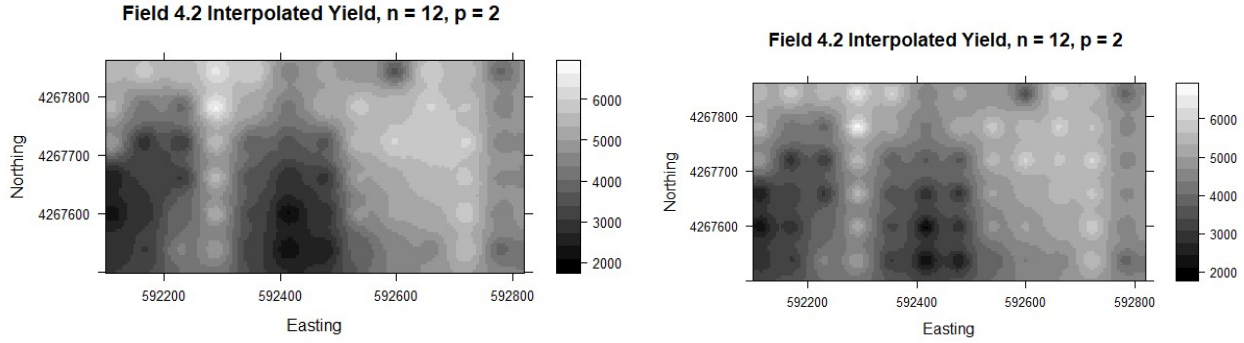
Chapter 6

Section 6.2.3

The plots in this section are unchanged. Interestingly, the form of the output of the function `moran.plot()` has changed so that the original code for this section no longer works.

Section 6.3.1

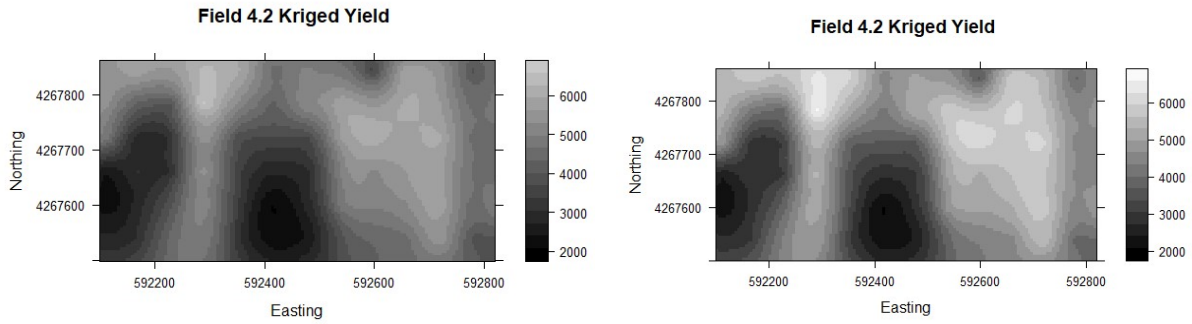
Fig. 6.5a



This is handled in the same way as Sec 5.2.1.

Section 6.3.2

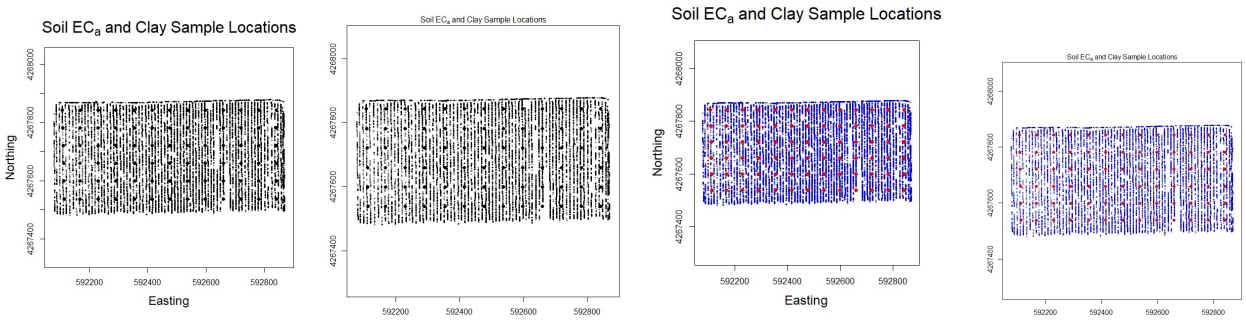
Fig. 6.6



The modifications are identical to those of Sec. 6.3.1.

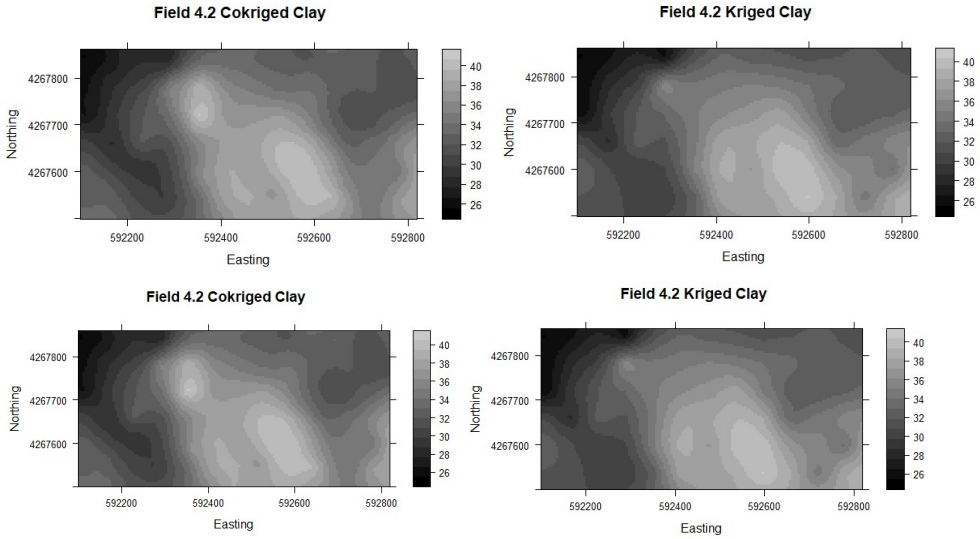
Section 6.3.3

Fig. 6.7



These are handled similarly to Sec. 6.3.1.

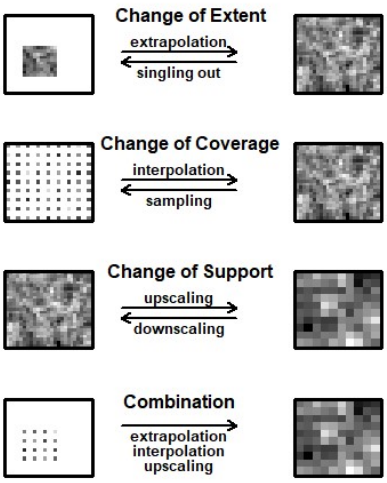
Fig. 6.9



These are handled similarly to Sec. 6.3.1.

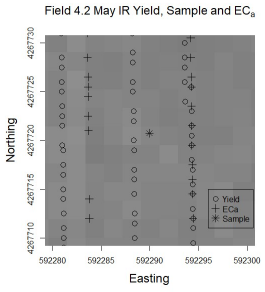
Section 6.4.1

Fig. 6.11



This one is tricky enough that I am going to leave it for now and come back to it.

Section 6.4.2
Fig. 6.11



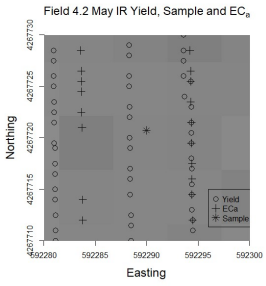
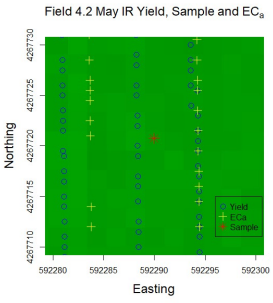


Fig. 6.12



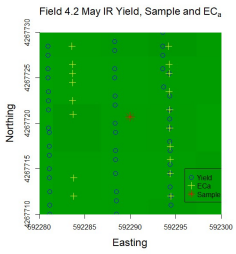
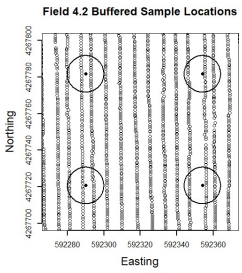
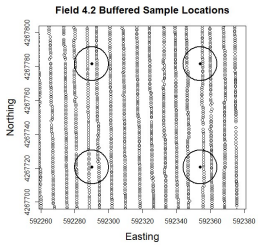
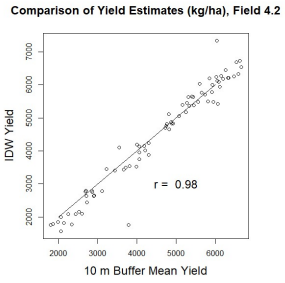
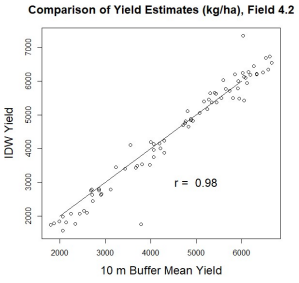


Fig. 6.13



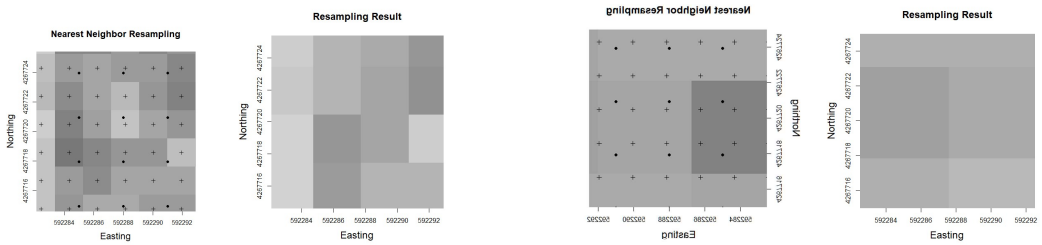






I used `st_multipolygon()` here in place of `spRbind()`.

Section 6.4.3
Fig. 6.14



This section provided an opportunity to work exclusively with terra objects. The expansion of `sf` into grids via `stars` and of `terra` into vector objects creates a problem for the agronomist or ecologist who wishes to use these packages in that there are subtle differences in the commands between the `sf/stars` world and the `terra` world. To the extent possible, it may be better to pick one and try to work within it.

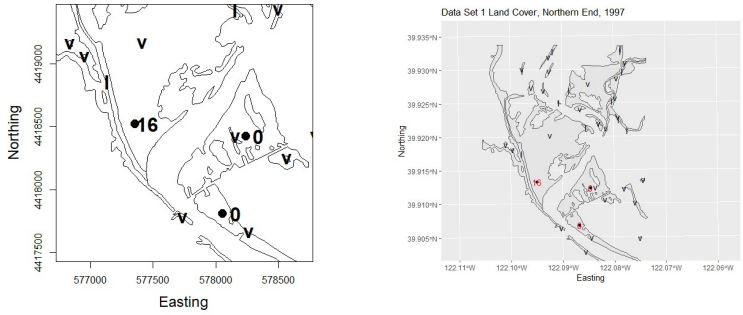
Chapter 7

Section 7.1

No changes in this section.

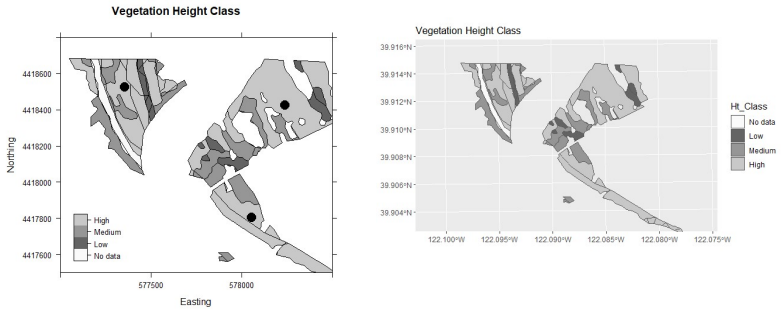
Section 7.2

Fig. 7.2a



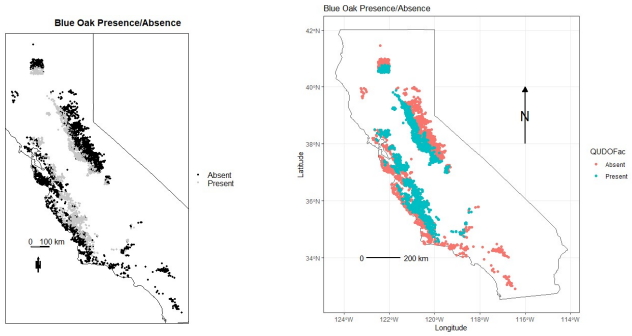
I may get back to this one. I decided to do it using `ggplot()` because the other alternatives seemed worse, but since I don't work with `ggplot2` every day I have a hard time remembering how it works when I do come back to it. Only Fig 7.2a is shown as the other figures follow the same procedure.

Fig. 7.3



Section 7.3

Fig. 7.8



Only Fig. 7.8a is shown. I actually did these figures after Fig. 8.10a below and did not wish to waste any more time on them. See the comments below that figure.

Sections 7.4 – 7.5

These were the last two chapters that I did. By this time I was so tired of dealing with five or six packages, each with incompatible functions having confusingly similar names and syntax, that I tackled these sections in the spirit of data exploration, being satisfied with producing the output without worrying about making it look pretty. This project has made me appreciate the disadvantages of software relying on volunteer developers.

Chapter 8

Section 8.2.1

No changes were necessary.

Section 8.2.2

No changes were necessary.

Section 8.3

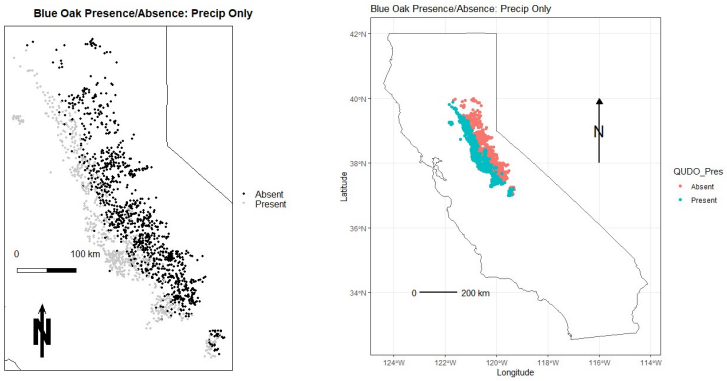
Only minor changes were necessary.

Section 8.4.1

Only minor changes were necessary.

Section 8.4.2

Fig. 8.10a



This figure encapsulates my frustrations with the whole ggplot scene. I tried to clip the figure in the same manner as Fig. 1.5 and for some reason it did not work in this case. I got so sick of it all that I didn't bother to try to figure out how to make the colors appear in grayscale. Any reader who knows how to

make the appropriate adjustments is welcome to contact me and I will include them with proper attribution. As a personal gripe, I find the ggplot syntax and the accompanying tidyverse etc. to be so idiosyncratic, and so different from every other programming language, that knowledge gained from years of programming experience is not transferrable. My guess is that when the inevitable time arrives that R is no longer used (as someone who started programming in Fortran and has since been through Algol, Basic, Pascal, Lisp, Smalltalk, C, C++, and probably a couple of others that I have forgotten, I know what I am talking about), people who only know the tidyverse will be at a disadvantage.

Section 8.4.3
No changes were necessary.

Section 8.4.4
Only minor changes were necessary.

Chapter 9

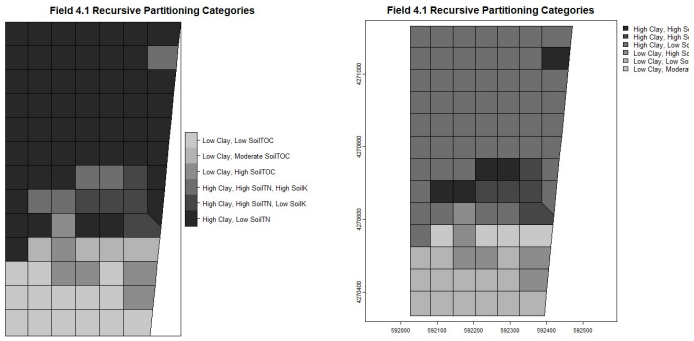
Section 9.2
Only minor changes were necessary.

Section 9.3.1
No changes were necessary.

Section 9.3.3
No changes were necessary.

Section 9.3.4
No changes were necessary.

Section 9.3.5
Fig. 9.16a

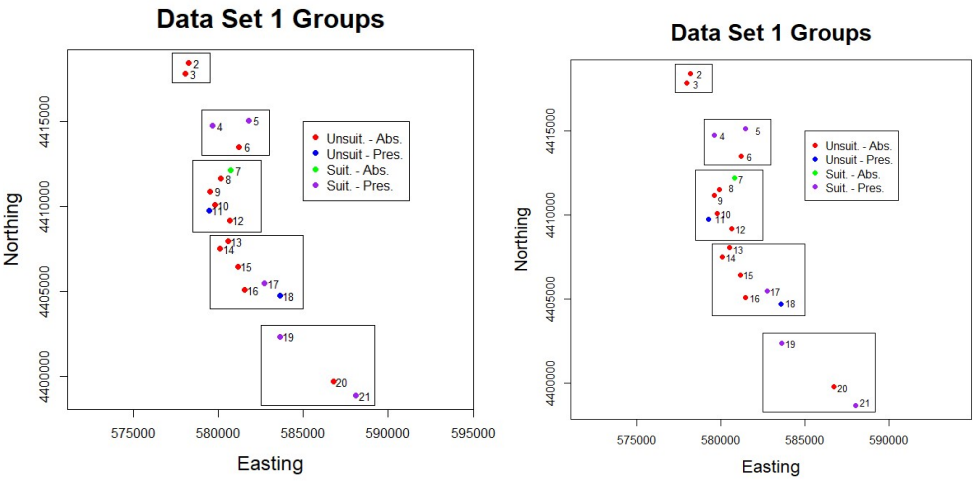


This was a simple repeat of the process used to create Fig. 4.2.

Chapter 10
No changes were necessary in this chapter.

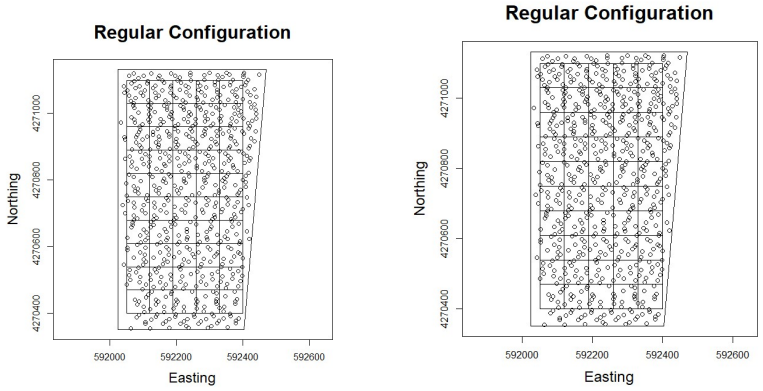
Chapter 11
The only sections in which changes were necessary are 11.3.2 and 11.5.1.

Section 11.3.2
Fig. 11.4



This one is a bit tricky. The original used the `sp` function `over()`, and the replacement uses `st_intersects()`. The tricky part is that `over()` returns a data frame that can be used to construct point data via the data frame coordinates, while `st_intersects()` returns an `sf` object with polygon geometry. For this reason we have to plot the centroids of the polygon objects to get the sppropriate point locations in the figure. See the discusion here https://r-spatial.github.io/sf/articles/sf5.html.

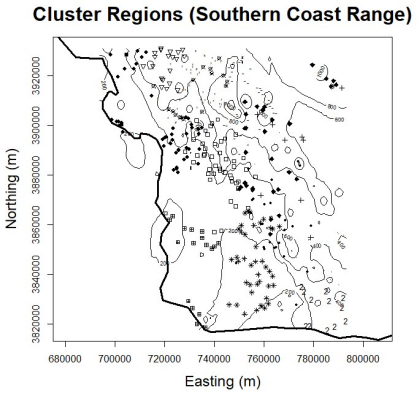
Section 11.5.1
Fig. 11.5.1

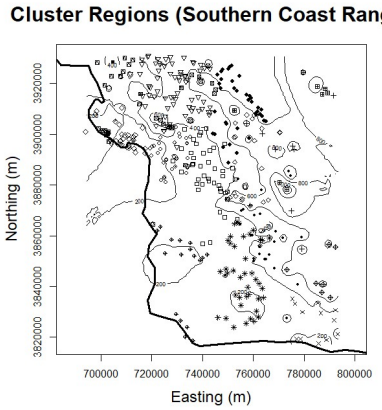


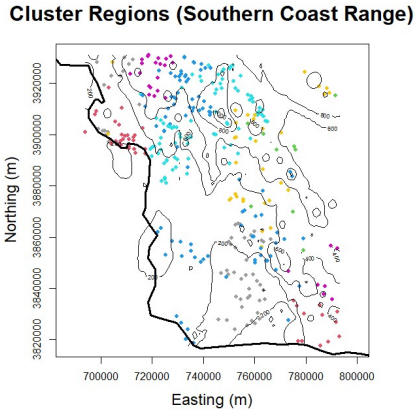
The code for this section makes use of the NDVI computed from the May aerial photos of Field 4.1. The original code used `rgdal` to read in the image file and the new code uses `terra`. The computation of the NDVI is taken from https://rspatial.org/terra/rs/4-unsupclassification.html. The Thiessen polygon coding is similar to that of Section 4.5.1. The correlation between yield and NDVI is obtained by extracting the values of the `terra` objects, converting them to `sf` objects, and using st.`intersects()` to overlay the yield and NDVI data on the Thiessen polygons, and computing the means for each polygon. Again to plot the boundary we have to use the command `plot(st_geometry(bdry.sf), …)` to get only one copy of the boundary.

Chapter 12

Section 12.6.1
Fig. 12.7







This works with terra spatRaster objects, drawing from Sections 1.3.1 and 6.4.3. For some reason that I did not take the time to investigate, `krige()` works here while `idw()` does not, even though the former produces an idw interpolation.

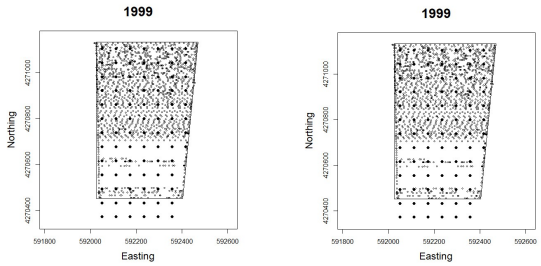
Section 12.6.2
There were no figures involved here, just simple changes from `sp` to `sf` objects.

Chapter 13
The files in this chapter involve the use of functions that require the package `spdep` and thus cannot be modified at this time.

Chapter 14
No files need to be changed.

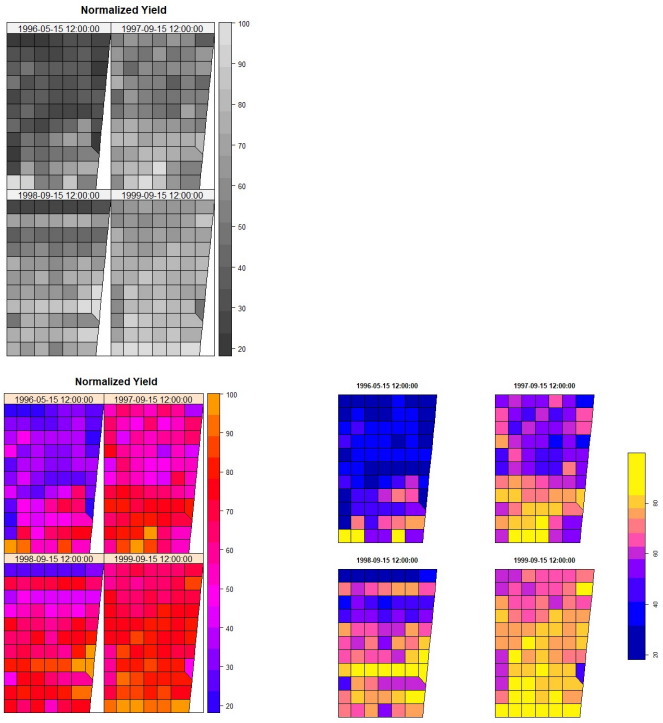
Chapter 15

The only sections that required modification in this chapter were 15.2.1 and 15.4.1
Section 15.2.1
Fig. 15.1



These involved a straightforward change from an `sp` object to an `sf` object.

Fig. 15.2



This involved converting `spacetime` objects to `stars` objects. As of this writing there is very little documentation on `stars` and its use. I used https://keen-swartz-3146c4.netlify.app/sf.html. The function `stars::plot()` did not plot greyscale and the main argument did not work, and I noticed that none of the example plots in the documentation include a main argument. The package `spacetime` requires the use of `sp` objects, but until `stars` is further developed it may be advisable to stay with `spacetime`.

Section 15.2.2

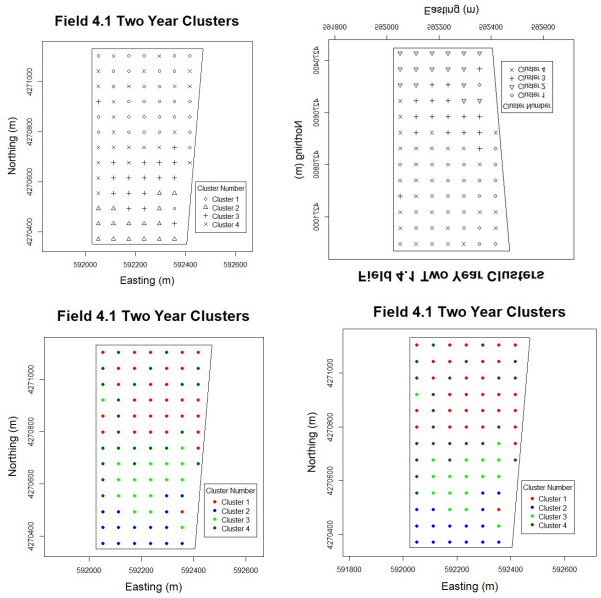
This section involves the use of the function `variogramST()`. In examining the code of this function, I found that, although it will accept a `stars` object, the first thing it does with that is to convert it into a `spacetime` object. This is no doubt convenient and expedient, but it seems to defeat the purpose of converting the code in this section from `spacetime` to `stars`. For that reason I left it alone.

Section 15.3.2

Again because section involves the use of the function `variogramST()` I did not change the code.

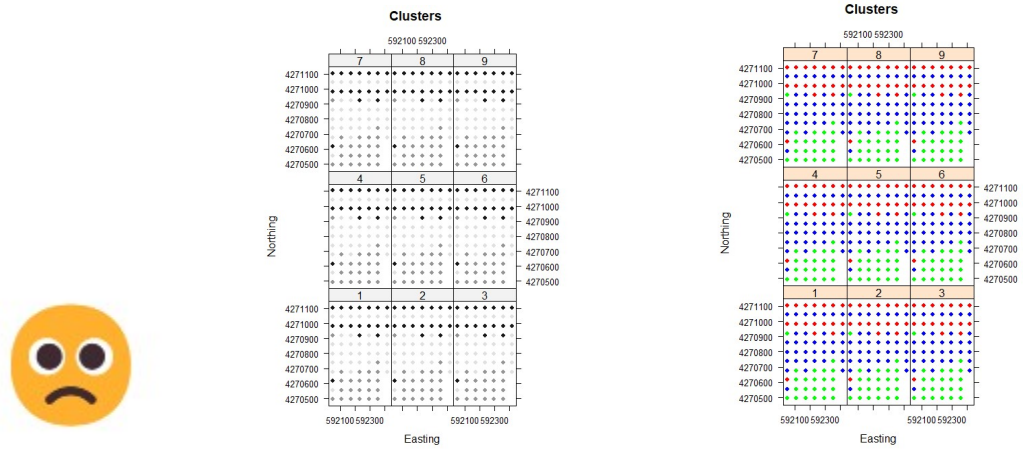
Section 15.4.1

Fig. 15.10



This is a simple replacement of the `sp` file with an `sf` file.

Fig. 15.11

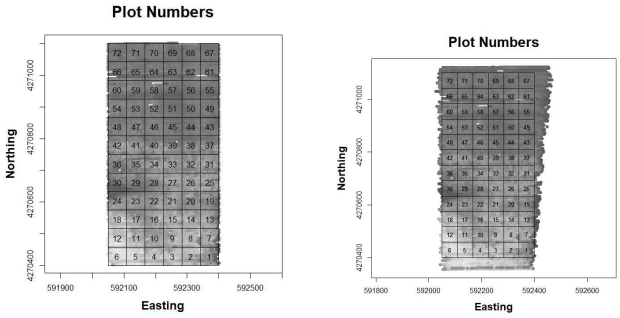


As of this writing I am running R version 4.2.0. When SDA2 was written I was running version 3.4.3 and I have maintained that version on my computer in anticipation of situations like this. When I attempted to execute the code to generate Fig. 15.11 using version 4.2.0 I got an error message. The code still executes as intended using version 3.4.3, so somebody changed something. There is no point in trying to debug this; it is better to just move on. Rather than trying to figure out what is going on with `spplot()` I decided to work directly with the `lattice` function `xyplot()`, which at this point seems much more stable. A good, simple introduction to `lattice` is in this website https://www.statmethods.net/advgraphs/trellis.html.

Chapter 16

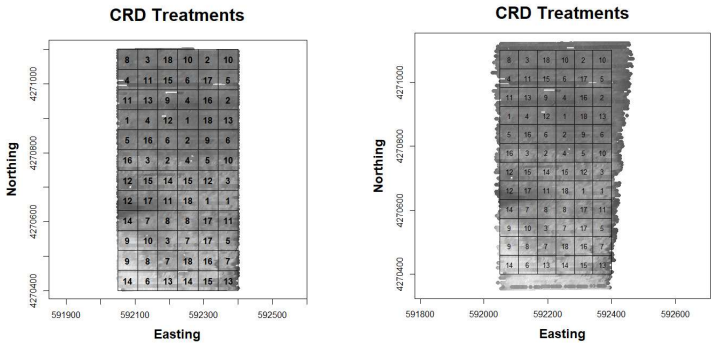
Section 16.2

Fig. 16.1



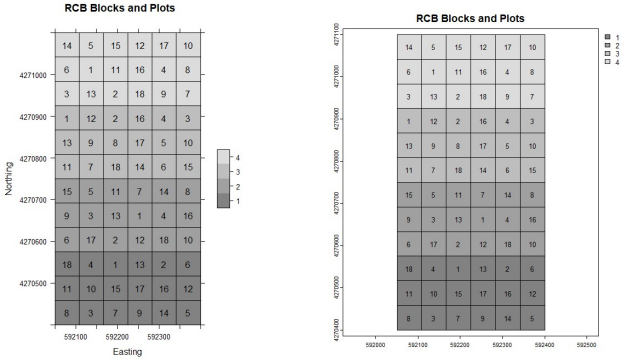
This figure took a lot more work than it should have. I basically combined programming from Sections 4.5.3, 5.2.1, and 11.5.1 to get it. The problem came in figuring out how to add the labels. It says here https://r-spatial.github.io/sf/articles/sf1.html that the function as.data.frame() applied to an sf object returns a data frame rather than an sf object, but it that was ever true it is not true now. It tool a long while to determine how to extract the coordinates of the centroids of the thiessen polygons so the labels could be placed there.

Fig. 16.2



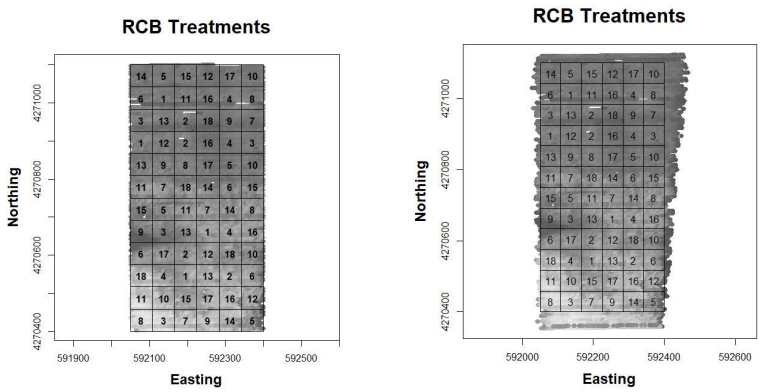
This is the same as Fig. 16.1.

Fig. 16.3a



This one was handled similarly to the code in Sections 4.5.3 and 9.3.5. As usual I could not figure out how to squeeze in the axia labels. Anyone who solves this is welcome to contact me.

Fig. 16.3b



This is the same as Fig. 16.1. The subsequent analysis variance shows how to replace `over()` with `st_intersects()` in an analytical task.

Sec. 16.3.1

There are no figures in this section to modify, but the `sp` objects are replaced with `sf` objects.

Sec 16.3.2

This section relies on the function `nb2listW()` and so the `sp` structures were retained.

Sec. 16.3.3

There are no figures in this section to modify, but the `sp` objects are replaced with `sf` objects.

Sec. 16.4.1

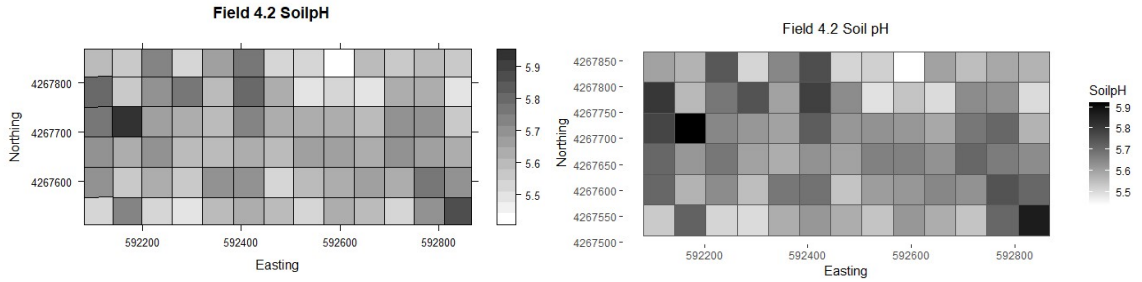
There are no figures in this section to modify, but the `sp` objects are replaced with `sf` objects.

Sec 16.4.2

This section relies on the function `nb2listW()` and so the `sp` structures were retained.

Sec. 16.4.3

Fig. 16.5



This figure presented the same issues as Fig. 1.3b in Section 1.2, and once again I resorted to `ggplot()` to produce it. The calculations done in the remainder of the file use the function `errorsarlm()` which, after poking around the Internet for a while, I found has been moved to the package `spatialreg`. There are also several functions that require the package `spdep`.

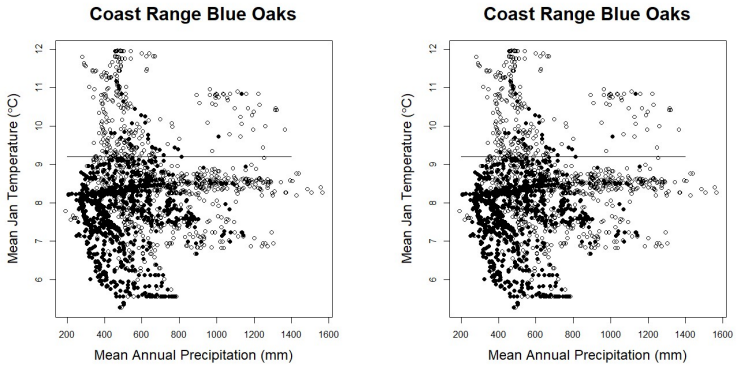
Chapter 17

Section 17.2

No code needs to be changed in this section.

Section 17.3

Fig. 17.2a and 17.2b

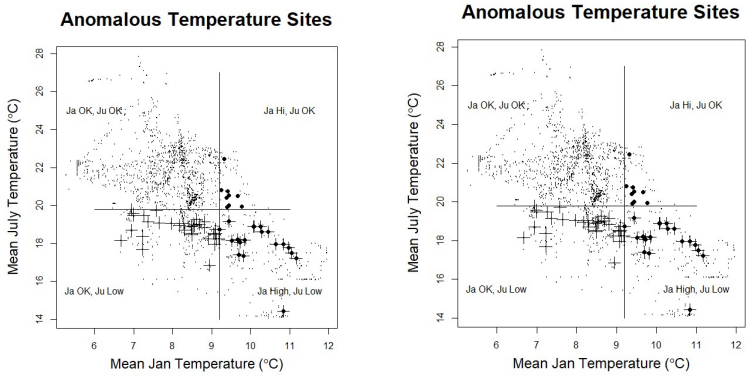


The code for these plots is identical.

Fig. 17.3

The code for these figures is the same, needing only the addition of `st_geometry`.

Fig. 17.4



This involves simply changing `sp` objects to `sf` objects.

Figs. 17.5 and 17.6

These involve simply changing `sp` objects to `sf` objects.

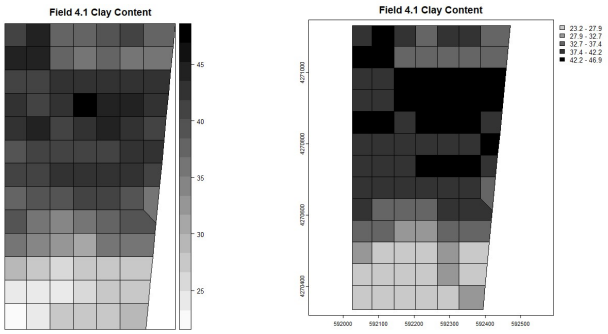
Section 17.4

Fig. 17.7

No change is required

Section 17.5

Fig. 17.8



This was a sgtrightforward change to the `terra` package.