# Summer Project Report

Joanna McArthur

## Aim

Distance for Windows, used to conduct analyses on datasets relating to animal abundance estimation, has multiple engines to perform analyses; the two engines of interest for this project were MCDS and MRDS. Both engines can be run from within Distance for Windows, however it can sometimes be useful to work directly with the code. In particular, running the same analysis in multiple engines may help determine which method of analysis is most appropriate. One issue that this raises is that the MCDS engines is written in Fortran, whereas MRDS is written in R, and each engine has different input formats, so direct comparison can be complicated and time consuming.

The aim of this project was to write a function in R which would take the input in the format it is provided to MRDS, and use that input to create a command file that would run an equivalent analysis in MCDS.

## Method

During the first two weeks of this project, I familiarised myself with the broader theory of distance sampling and read through the relevant R code in the MRDS package, in particular ddf.R and associated functions. I also read through some code previously written to investigate comparison of MCDS and MRDS, provided in the optimist repository, which I copied to my own repository.

Once I had familiarised myself with this code, I started to write my own function. The MCDS command file has four main sections, the header section, options section, data section, and estimate section. Within each are commands specifying aspects of how the analysis should be performed; I worked through each section and matched up the input required for MRDS to the commands required for MCDS.

Certain commands translated from one engine to another very directly, for example whether the data is from line or point transects. For MRDS, the user specifies if the data is from a point transect or not in the meta.data input:

```
meta.data=list(point=TRUE,width=5.0)
```

Within MCDS, the `TYPE` command has three possible values, `LINE`, `POINT`, or `CUE`. MRDS has no way of specifying whether the input is from cue counts, so the function will only assign either `POINT` or `LINE` to the `TYPE` command. As the inputs were very similar, the code for this command was fairly simple:

```
  if(meta.data$point == TRUE){
    cat(paste("DISTANCE=RADIAL /UNITS='Meters' /WIDTH=",
        meta.data$width, ";", sep=""), file=command.file.name, "\n",
        append=TRUE)
    cat("TYPE=POINT;", file=command.file.name, "\n", append=TRUE)
  }else{
    cat(paste("DISTANCE=PERP /UNITS='Meters' /WIDTH=",
        meta.data$width, ";", sep=""), file=command.file.name, "\n",
        append=TRUE)
    cat("TYPE=LINE;", file=command.file.name, "\n", append=TRUE)
  }
```

Beyond this, the translation from one engine's input to the other was less direct, so more complex code was required to extract the relevant information from the MRDS input.

### Reformatting the Data

In MRDS, the data is inputted as a data frame, however the MCDS engine requires a data file as its data input. Furthermore, MCDS has very strict requirements for which fields are present and what they are named, therefore it was necessary to rename required fields and remove all others from the data frame.

Firstly, since MRDS is used for mark-recapture distance sampling surveys, the data produced often has multiple entries for each observed object, one for each observer and a column specifying whether the object was recorded by each observer. In MCDS, all objects were observed, so much of the extra information about observers and whether each object was detected was redundant. To deal with this, first only the entries for observed objects were kept:

```
data <- data[data$detected==1,]
```

However, there were still duplicate entries as some objects may have been detected by multiple observers. To remove these duplicates, such objects were identified and only one entry was kept for each object.

```
if(TRUE %in% grepl("^object$",tolower(colnames(data)))){
    # in case the object column isn't named 'object', to remove case sensitivity
    obj_col <- grep("^object$",tolower(colnames(data)))
    colnames(data)[obj_col] <- "object"
    # identifying all objects and taking the first data point for each
    obj_nums <- unique(data$object)
    for(i in 1:length(obj_nums)){
      entries <- grep(TRUE,data$object==i)
      if(length(entries)>1){
        remove <- entries[-1]
        data <- data[-remove,]
      }
    }
  }
```

After this, the required fields were identified. MCDS requires that there are fields for sample label, sample effort, and distance, named `SMP_LABEL`, `SMP_EFFORT`, and `DISTANCE` respectively. If the data is from clusters, then `SIZE` is also a required field. Additionally, `STR_LABEL` and `STR_AREA` may be specified; all other fields must be covariates in the model.

In order to allow for slight variation in the data format, I allowed for different names for these fields, based upon the example data given in Distance for Windows; this also made testing the code with that data much easier. Therefore, the code was able to identify which field in the data frame corresponded to each required field, and renamed them if necessary. For the three required fields the function also created a dummy field if it were not present in the initial dataframe. An example of this for the `SMP_LABEL` field:

```
# find if SMP_LABEL is a field; if not, add it
  if(TRUE %in% grepl("^SMP_LABEL",toupper(colnames(data)))){
    colnames(data)[grep("^SMP_LABEL",toupper(colnames(data)))] <- "SMP_LABEL"
  }else if(TRUE %in% grepl("^sample.label$",tolower(colnames(data)))){
    colnames(data)[grep("^sample.label$",tolower(colnames(data)))] <- "SMP_LABEL"
  }else{
    data$SMP_LABEL <- rep(1,nrow(data))
  }
```

Once these specified fields had been identified, only the covariate fields were left to identify. To do this, the covariates were extracted from the model inputted in dsmodel and the matching fields were identified. MCDS

does not allow any of the aforementioned required fields to be covariates, with the exception of `SIZE`, so an additional check was included to ensure that this did not happen:

```
    # extracting the list of covariates
    covars <- all.vars(dsmodel)
    # creating a list of the fields for each covariate
    covar_fields <- rep("",length(covars))
    for(i in 1:length(covars)){
      # identifying the field name for each covariate
      index <- grep(covars[i],tolower(colnames(data)))
      covar_fields[i] <- colnames(data)[index]
    }
    # the required fields cannot be covariates in the model, with the exception of size
    if(length(intersect(tolower(req_fields),tolower(covar_fields))) > 0){
      if(TRUE %in% grepl("size",tolower(covar_fields))){
        # specify whether SIZE is a covariate
        size_cov <- TRUE
      }
      # remove any required fields from the list of covariates
      covar_fields <- covar_fields[! covar_fields %in% intersect(req_fields,covar_fields)]
    }else{
      size_cov <- FALSE
    }
    # add covariates to the fields that are kept for analysis
    req_fields <- c(req_fields,covar_fields)
    # if SIZE is a covariate, add it back to the list of covariates
    if(size_cov == TRUE){
      covar_fields <- append(covar_fields,"SIZE")
    }
```

The vector of required fields was then used to specify which fields were to be kept in the data used for analysis. The data could then be written into a data file that can be accessed by the MCDS command file.

Since the columns of the data frame had then been renamed, they could be inputted into the `FIELDS` command:

```
cat(paste("FIELDS=", fields_comb, ";", sep=""),
      file=command.file.name, "\n", append=TRUE)
```

One additional complication was the factor covariates must be specified in MCDS, however this is not required for MRDS, therefore there was no input specifying which fields are factors. When the levels of a factor are numeric, the `as.factor()` function will not recognise it as a factor, therefore when inputting factor data the user must use characters as the levels. To identify and specify which covariates are factors and write a `FACTOR` command for each, the following code was used:

```
    factor_fields <- c()
    for(i in 1:length(colnames(data))){
      if(is.factor(data[,i]) && (TRUE %in% grepl(colnames(data)[i],covar_fields))){
        factor_fields <- append(factor_fields,colnames(data)[i])
        labels <- paste(levels(data[,i]), collapse=",")
        cat(paste("FACTOR /NAME=", toupper(colnames(data)[i]),
                  " /LEVELS=", length(levels(data[,i])), " /LABELS=",
                  labels, sep=""), file=command.file.name, "\n",
            append=TRUE)
      }
    }
```

### Accessing Model Parameters

Due to the hybrid formula/function notation used to specify `dsmodel`, the usual `$` notation does not work to access the parameters. Instead, a fairly clunky method of accessing them which has been used in code elsewhere in the `mrds` package was used:

```
  mod_paste <- paste(dsmodel)
  mod_vals <- try(eval(parse(text=mod_paste[2:length(modpaste)])))
```

From this, the required information about the model specifications could be obtained to pass through to MCDS, such as the key function to be used:

```
cat("ESTIMATOR /KEY=", file=command.file.name, append=TRUE)
  if(mod_vals$key == "hn"){
    cat("HNORMAL", file=command.file.name, append=TRUE)
  }else if(mod_vals$key == "hr"){
    cat("HAZARD", file=command.file.name, append=TRUE)
  }else if(mod_vals$key == "unif"){
    cat("UNIFORM", file=command.file.name, append=TRUE)
  }else{
    cat("NEXPON", file=command.file.name, append=TRUE)
  }
```

### Rearranging Initial Values

It is possible to enter initial values as part of the control input of MRDS, however accessing these values in a way that could be inputted into MCDS was challenging. The initial input is a named list with three available parameters: scale, shape, and adjustment. The scale parameter will contain the initial values for any covariates in the key function, which must be inputted into the MCDS command first, before any initial values for the adjustment parameters.

In order to input the initial values in the same order as the covariates are specified in the covariate command, I created a string starting with `control$initial$shape$` and concatenated the covariate name, then evaluated the text to access the initial value.

```
      inits <- c()
      for(i in 1:length(covars)){
        # find the index of the covariate field
        index <- grep(toupper(covar_fields[i]),toupper(colnames(data)))
        # create the text for the parameter that must be accessed
        access_covar <- paste("control$initial$scale$",
                              colnames(data)[index],sep="")
        # evaluate the text in order to access the initial value
        inits <- append(inits,eval(parse(text=access_covar)))
      }
```

When a covariate is a factor, an initial value must be given for each level. The order in which they are inputted varies between MRDS and MCDS, however the first level is absorbed for the analysis; this level must be inputted last in MCDS to be absorbed, so the initial value for the first factor level is placed at the end when listed in the `/START` switch in MCDS. In order to do this, I included an if statement in the above for loop which identified and dealt with factors:

```
       if(TRUE %in% grepl(covar_fields[i],factor_fields)){
          for(j in 2:length(levels(data[,index]))){
            # create the text for the parameter that must be accessed
            access_covar <- paste("control$initial$scale$",
                                  colnames(data)[index],"[",j,"]",sep="")
            # evaluate the text in order to access the initial value
```

```
            inits <- append(inits,eval(parse(text=access_covar)))
          }
          # the first level has to be last in MCDS
          access_covar <- paste("control$initial$scale$",
                                colnames(data)[index],"[1]",sep="")
          inits <- append(inits,eval(parse(text=access_covar)))
        }
```

These could then be combined and inputted into the `/START` switch:

```
cat(paste(" /START=", paste(inits,collapse=","), sep=""),
        file=command.file.name, append=TRUE)
```

The full code can be found in the repository: https://github.com/jonah417/summer_project

## Next Steps

There are still a few issues with this code, and areas I unfortunately did not have time to cover, which I have detailed here.

Currently the function outputs the file name of the command file, however I was not able to access and run the MCDS engine with this command file from within R itself. While testing the code, I generated the file in R and then downloaded it to my own device to test it in MCDS. In order to conduct comparisons more easily, it would be ideal if it were possible to run the command file in MCDS from R.

There is also currently no way of specifying data where `TYPE=LINE` but `DISTANCE=RADIAL`. A potential way to include this may be to check if there is a column within the dataset for the angle, which would be a required column in MCDS in this case, however that may not be a watertight way to identify the input type correctly. There is also currently no way of specifying if cue counts were used, therefore `TYPE=CUE` is never used in this function.

Additionally, it would be beneficial to perform more test on the code. I was able to test some standard inputs, as well as testing that the function deals with incomplete input (for example models with no adjustment parameters, and empty control inputs), however more testing of fringe cases should be considered.

Finally, the code desperately wants to be modular; this is something I would like to try to do in future.