Using BRCindicators

Tom August

10 April, 2017

Contents

Introduction

This document shows how to use the indicator pipeline to create biodiversity indicators such as those for DEFRA's Biodiversity Indicators in Your Pocket. The pipeline is shared in the form of an R package called 'BRCindicators' making it easy to share and maintain.

The functions in BRCindicators work with yearly estimates of species abundance or occurrence and aggregate them into an scaled indicator value with bootstrapped confidence intervals

This package has the ability to read in the output of occupancy models created in the R package sparta, a package for estimating species trends from occurrence data. This package can be installed from Github and details of how to use the package are given in the package vignette. There is no need to use sparta to create your yearly species estimates as BRCindicators can also work with other data.

To create an indicator we first need to have species trends, let's create some using the sparta R package.

Creating yearly estimates of occurrence in sparta

If you already have yearly estimates of abundance or occurrence for your species you can skip this stage. Here we show how you can create these estimates from raw species observation data using sparta.

```
# Install the package from CRAN
# THIS WILL WORK ONLY AFTER THE PACKAGE IS PUBLISHED
install.packages('sparta')
```

```
# Or install the development version from GitHub
library(devtools)
install_github('biologicalrecordscentre/sparta')
```

Let's assume you have some raw data already, we can under take occupancy modelling like this

```
# Load the sparta package
library(sparta)
```

```
## Loading required package: lme4
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'lme4'
```

```
## The following object is masked from 'package:stats':
##
##     sigma
```

For demonstration purposes I have a faked dataset of 8000 species observations. In my dataset the species are named after the letters in the alphabet. Below I show how I can use the Bayesian occupancy models in sparta to create yearly estimates of occurrence. For more information please see the vignette for sparta

```
# Preview of my data
head(myData)
```

```
##   taxa site time_period
## 1    r  A51  1970-01-14
## 2    v  A87  1980-09-29
## 3    e  A56  1996-04-14
## 4    z  A28  1959-01-16
## 5    r  A77  1970-09-21
## 6    x  A48  1990-02-25
```

```
# First format our data
formattedOccData <- formatOccData(taxa = myData$taxa,
                                  site = myData$site,
                                  time_period = myData$time_period)
```

```
## Warning in errorChecks(taxa = taxa, site = site, time_period =
## time_period): 94 out of 8000 observations will be removed as duplicates
```

```
# Here we are going to use the package snowfall to parallelise
library(snowfall)
```

```
## Loading required package: snow
```

```
# I have 4 cpus on my PC so I set cpus to 4
# when I initialise the cluster
sfInit(parallel = TRUE, cpus = 4)
```

```
## Warning in searchCommandline(parallel, cpus = cpus, type = type,
## socketHosts = socketHosts, : Unknown option on commandline:
## rmarkdown::render('W:/PYWELL_SHARED/Pywell Projects/BRC/Tom August/R
## Packages/BRCindicators/vignette/using_BRCindicators.Rmd', encoding
```

```
## R Version:  R version 3.3.2 (2016-10-31)
```

```
## snowfall 1.84-6.1 initialized (using snow 0.4-2): parallel execution on 4 CPUs.
```

```
# Export my data to the cluster
sfExport('formattedOccData')

# I create a function that takes a species name and runs my model
occ_mod_function <- function(taxa_name){

  library(sparta)

  # Note that this will write you results to your computer
  # the location is set to your user folder
  occ_out <- occDetFunc(taxa_name = as.character(taxa_name),
                        n_iterations = 200,
                        burnin = 15,
                        occDetdata = formattedOccData$occDetdata,
                        spp_vis = formattedOccData$spp_vis,
                        write_results = TRUE,
                        output_dir = '~/Testing_indicator_pipe',
                        seed = 123)
}

# I then run this in parallel
system.time({
para_out <- sfClusterApplyLB(unique(myData$taxa), occ_mod_function)
})
```

```
##    user  system elapsed
##    0.49    0.09  146.64
```

```
# Stop the cluster
sfStop()
```

```
##
## Stopping cluster
```

```
# We can see all the files this has created
list.files('~/Testing_indicator_pipe')
```

```
##  [1] "a.rdata" "b.rdata" "c.rdata" "d.rdata" "e.rdata" "f.rdata" "g.rdata"
##  [8] "h.rdata" "i.rdata" "j.rdata" "k.rdata" "l.rdata" "m.rdata" "n.rdata"
## [15] "o.rdata" "p.rdata" "q.rdata" "r.rdata" "s.rdata" "t.rdata" "u.rdata"
## [22] "v.rdata" "w.rdata" "x.rdata" "y.rdata" "z.rdata"
```

Installing BRCindicators

Installing the package is easy and can be done in a couple of lines

```
library(devtools)
install_github(repo = 'biologicalrecordscentre/BRCindicators')
```

Summarising sparta output for an indicator

Now that we have some species trends data to work with (no doubt you already have your own) we can use the first function in BRCindicators. This function reads in all the output files from sparta (which are quite

large and complex) and returns a simple summary table that we can use for calculating the indicator. If you have done your analysis without using sparta you can skip to the next step.

```
library(BRCindicators)

# All we have to supply is the directory where out data is saved
# You will note this is the 'output_dir' passed to sparta above.
trends_summary <- summarise_occDet(input_dir = '~/Testing_indicator_pipe')
```

```
## Loading data...done
```

```
# Lets see the summary
head(trends_summary[,1:5])
```

```
##      year         a         b         c         d
## [1,] 1950 0.6745699 0.7173656 0.4802151 0.5568280
## [2,] 1951 0.6675806 0.6460215 0.5637097 0.6809677
## [3,] 1952 0.4059677 0.6024731 0.5306989 0.5035484
## [4,] 1953 0.1990860 0.5976344 0.4347312 0.5723656
## [5,] 1954 0.5780108 0.5145699 0.6909677 0.5766667
## [6,] 1955 0.2000000 0.4475806 0.5319892 0.4764516
```

Returned from this function is a summary of the data as a matrix. In each row we have the year, specified in the first column, and each subsequent column is a species. The values in the table are the mean of the posterior for the predicted proportion of sites occupied, a measure of occurrence.

Calculating indicator values

Once we have species-year indicies we are in a position to proceed to calculating an indictor. To do this there are a number of mehods available, some of which are presented here in 'BRCindicators'

Geometric mean

The geometric mean method is often used with data that do not have errors associated with them.

The first step is to re-scale the data so that the value for all species in the first year is the same. Once this is done we calculate the geometric mean across species for each year creating the indicator value. This function also accounts for species that have no data at the beginning of the dataset by entering them at the geometric mean for that year, this stops them dramatically changing the indicator value in the year they join the dataset. It also accounts for species that leave the dataset before the end by holding them at their last value. Finally limits to species values can be given, preventing extremely high or low values biasing the indicator.

Rescaling and calculating geometric mean

The data I have generated in 'trends_summary' is very easy to work with but to show off what this function can do I'm going to mess it up a bit.

```
trends_summary[1:3, 'a'] <- NA
trends_summary[1:5, 'b'] <- NA
trends_summary[2:4, 'c'] <- 1000
trends_summary[45:50, 'd'] <- NA

# Let's have a look at these changes
head(trends_summary[,1:5])
```

```
##      year         a         b           c         d
## [1,] 1950        NA        NA   0.4802151 0.5568280
## [2,] 1951        NA        NA 1000.0000000 0.6809677
## [3,] 1952        NA        NA 1000.0000000 0.5035484
## [4,] 1953 0.1990860        NA 1000.0000000 0.5723656
## [5,] 1954 0.5780108        NA    0.6909677 0.5766667
## [6,] 1955 0.2000000 0.4475806    0.5319892 0.4764516
```

```
tail(trends_summary[,1:5])
```

```
##       year         a         b          c  d
## [45,] 1994 0.3546237 0.5100000 0.06005376 NA
## [46,] 1995 0.6591935 0.5248925 0.77193548 NA
## [47,] 1996 0.4116129 0.6036022 0.33881720 NA
## [48,] 1997 0.3696237 0.6756989 0.76268817 NA
## [49,] 1998 0.6983333 0.4795161 0.44989247 NA
## [50,] 1999 0.3657527 0.4788172 0.56693548 NA
```

Now that I have 'messed up' the data a bit we have two species with data missing at the beginning and one species with data missing at the end. We also have one species with some very high values.

Now lets run this through the re-scaling function.

```
# Let's run this data through our scaling function (all defaults used)
rescaled_trends <- rescale_species(Data = trends_summary)

# Here's the result
head(rescaled_trends[,c('year', 'indicator', 'a', 'b', 'c', 'd')])
```

```
##      year indicator        a       b          c         d
## [1,] 1950 100.00000       NA      NA   100.0000 100.00000
## [2,] 1951 116.64823       NA      NA 10000.0000 122.29410
## [3,] 1952 126.98409       NA      NA 10000.0000  90.43159
## [4,] 1953 112.18622 112.1862      NA 10000.0000 102.79038
## [5,] 1954  98.31502 325.7127      NA   143.8871 103.56281
## [6,] 1955 102.34902 112.7013 102.349   110.7815  85.56532
```

```
tail(rescaled_trends[,c('year', 'indicator', 'a', 'b', 'c', 'd')])
```

```
##       year indicator        a        b         c        d
## [45,] 1994  93.14009 199.8327 116.6226  12.50560 121.6472
## [46,] 1995 109.22845 371.4597 120.0281 160.74787 121.6472
## [47,] 1996 108.59012 231.9465 138.0267  70.55531 121.6472
## [48,] 1997 115.28010 208.2852 154.5132 158.82221 121.6472
## [49,] 1998 101.41978 393.5152 109.6518  93.68562 121.6472
## [50,] 1999 100.58702 206.1039 109.4919 118.05867 121.6472
```

You can see that species 'a' and 'b' enter the dataset at the geometric mean (the indicator value), all species are indexed at 100 in the first year and the very high values in 'c' are capped at 10000 at the end 'd' has been held at it's end value.

The 'indicator' column that is returned here is our indicator, calculated as the geometric mean of all the species in the data set.

Confidence intervals

We can get confidence intervals for this indicator by bootstrapping across species. We have a function for that too!

```
# This function takes just the species columns
scaled_species <- rescaled_trends[,!colnames(rescaled_trends) %in% c('year', 'indicator')]
indicator_CIs <- bootstrap_indicator(Data = scaled_species)
```

```
## Running bootstrapping for 10000 iterations...done
```

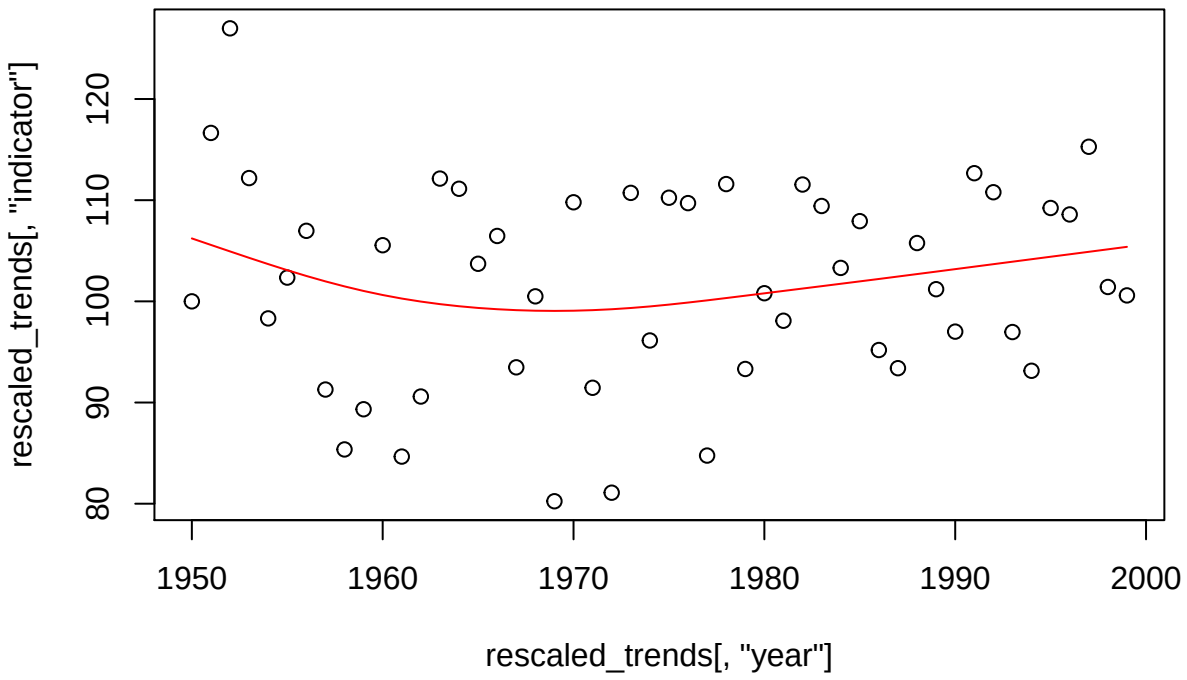
```
# Returned are the CIs for our indicator
head(indicator_CIs)
```

```
##      quant_025 quant_975
## [1,] 100.00000  100.0000
## [2,]  86.34719  184.4499
## [3,]  94.41109  198.6361
## [4,]  78.28526  178.8216
## [5,]  83.02171  117.3654
## [6,]  90.79515  114.9603
```

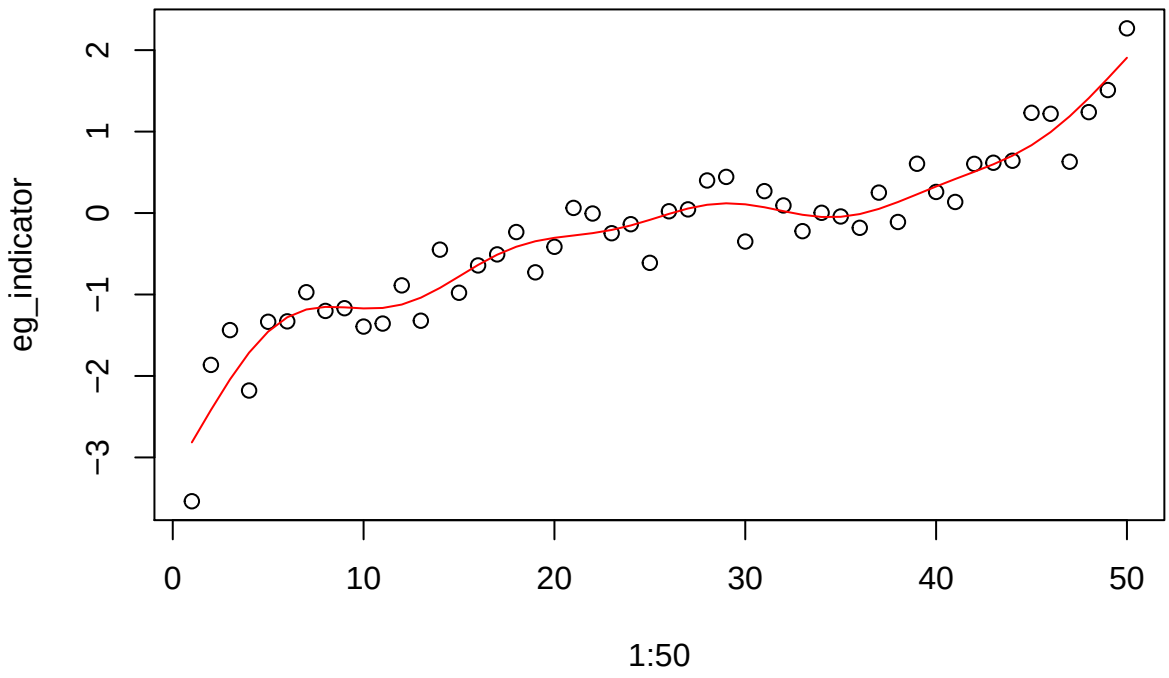
Smoothing

It is sometimes desirable to create a smoothed indicator value from the raw values. This can be achieved by fitting a GAM (general additive model) to the indicator using a spline. This spline is a smoothed curve that goes through the raw values for the indicator and is fitted using the function 'gam' in the 'mgcv' R package.

```
# The smoothing function takes the indicator values
smoothed_indicator <- GAM_smoothing(rescaled_trends[,'indicator'])

# In this example there is little support for a non-linear trend and
# so the line almost linear
plot(x = rescaled_trends[,'year'], y = rescaled_trends[,'indicator'])
lines(x = rescaled_trends[,'year'], y = smoothed_indicator, col = 'red')
```




```r
# But if our indicator did support a non-linear trend it might look
# like this
eg_indicator <- jitter(sort(rnorm(50)), amount = 0.5)
eg_smoothed <- GAM_smoothing(eg_indicator)
plot(x = 1:50, y = eg_indicator)
lines(x = 1:50, y = eg_smoothed, col = 'red')
```

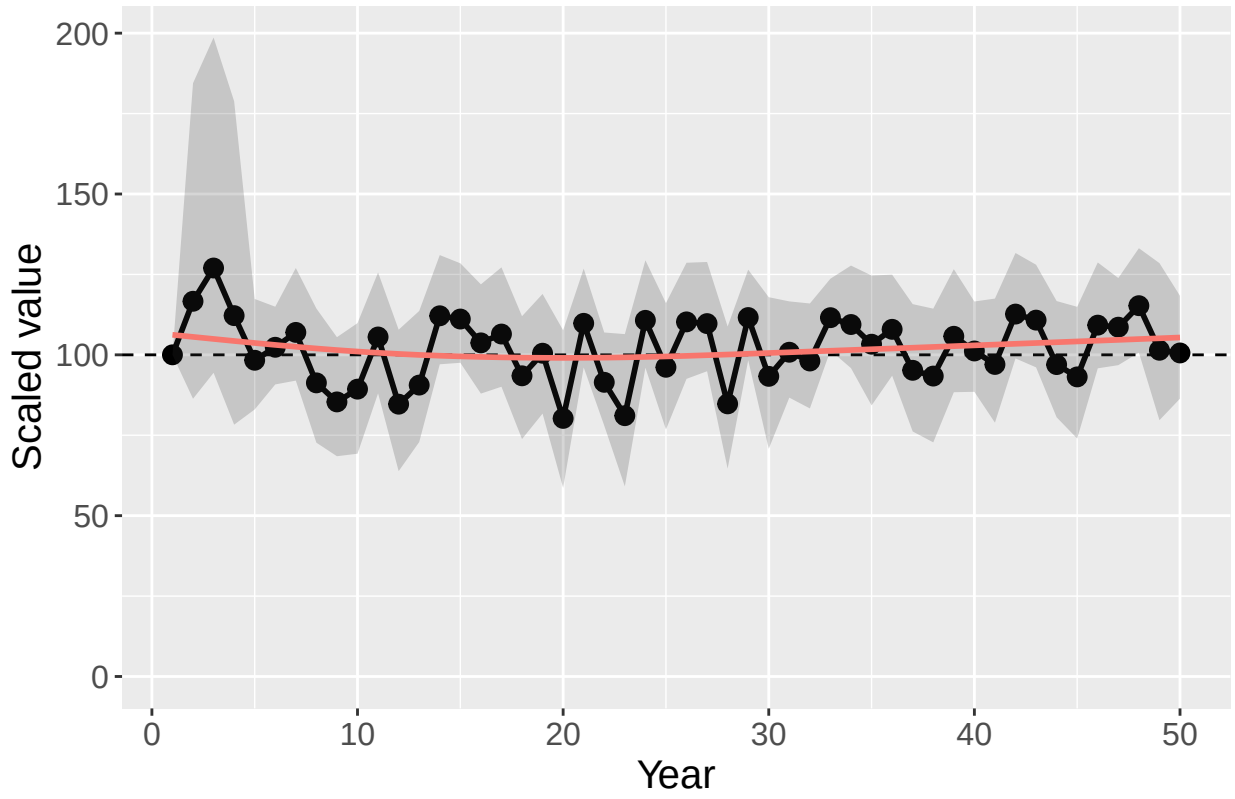


Where there is little support for a non-linear trend a GAM smoothed line will tend towards linear. Where there is good support for a non-linear trend the smoothed line will become more 'bendy'.

Plotting

We now have our indicator and the confidence intervals around it. The next step is to plot it. We have included a function that creates a simple plot using ggplot2, however you could easily create your own plots in R using the data.

```
# Plot our indicator.
plot_indicator(indicator = rescaled_trends[,'indicator'],
               smoothed_line = smoothed_indicator,
               CIs = indicator_CIs)
```



In this plot you can see the high upper confidence interval in years 2-4, this is due to the artificially high values we gave to species 'c'.

Bayesian Meta-Analysis (BMA)

The Bayesian Meta-Analysis method, or BMA, is suited to data with standard errors associated with them. As with other methods we require data from more than one species, across a number of years, with an error for each species-year estimate.

```
# Here is an example dataset for the BMA method
data <- data.frame(species = rep(letters, each = 50),
                   year = rep(1:50, length(letters)),
                   index = runif(n = 50 * length(letters), min = 0, max = 1),
                   se = runif(n = 50 * length(letters), min = 0.01, max = .1))
head(data)
```

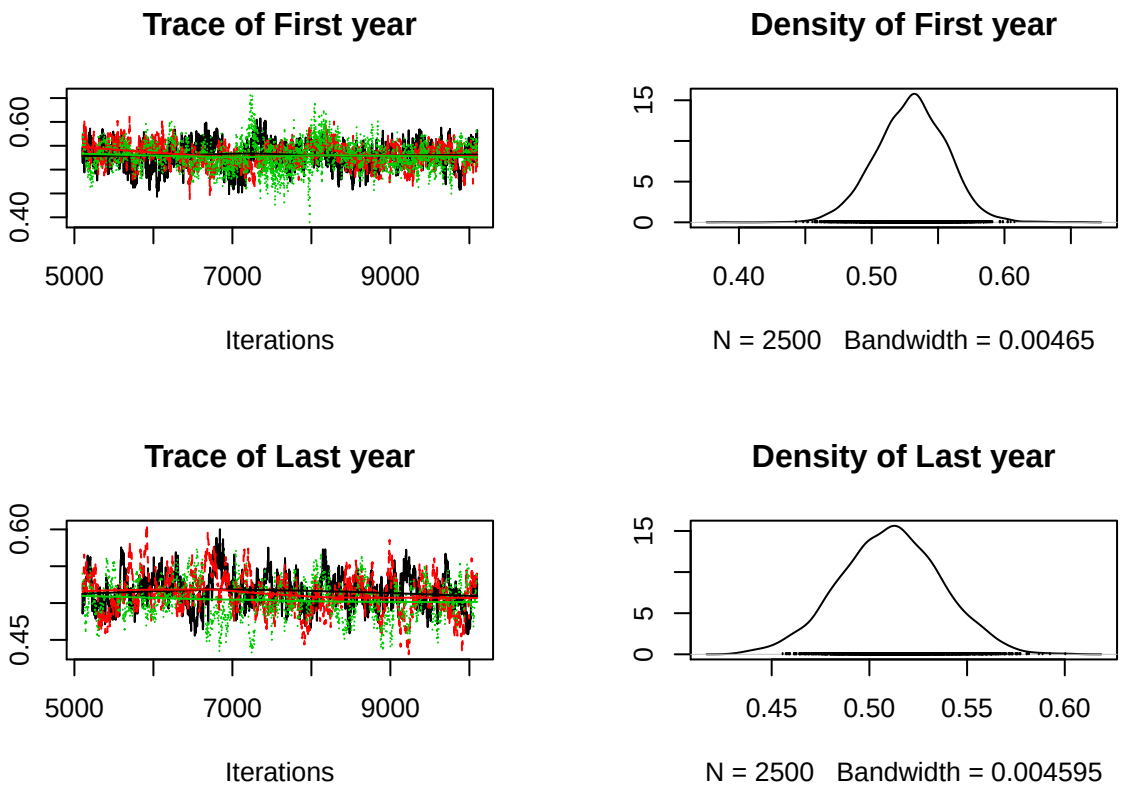
```
##   species year     index         se
## 1       a    1 0.3653818 0.08335768
## 2       a    2 0.7266121 0.04494251
## 3       a    3 0.8660005 0.09993223
## 4       a    4 0.4728771 0.09887020
## 5       a    5 0.7411969 0.09313386
## 6       a    6 0.6477206 0.02236809
```

It is important that your data is in the same format and that your columns are in the same order and have the same names. Remember you can use the function `read.csv()` to read in the data from a .csv on your computer.

BMA is run using the function `bma`, here we will use the default settings and then see what we can change.

```
bma_indicator <- bma(data)
```

```
##
## Processing function input.......
##
## Done.
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 2600
##    Unobserved stochastic nodes: 1380
##    Total graph size: 13125
##
## Initializing model
##
## Adaptive phase, 100 iterations x 3 chains
## If no progress bar appears JAGS has decided not to adapt
##
##
##  Burn-in phase, 5000 iterations x 3 chains
##
##
## Sampling from joint posterior, 5000 iterations x 3 chains
##
##
## Calculating statistics.......
##
## Done.
```



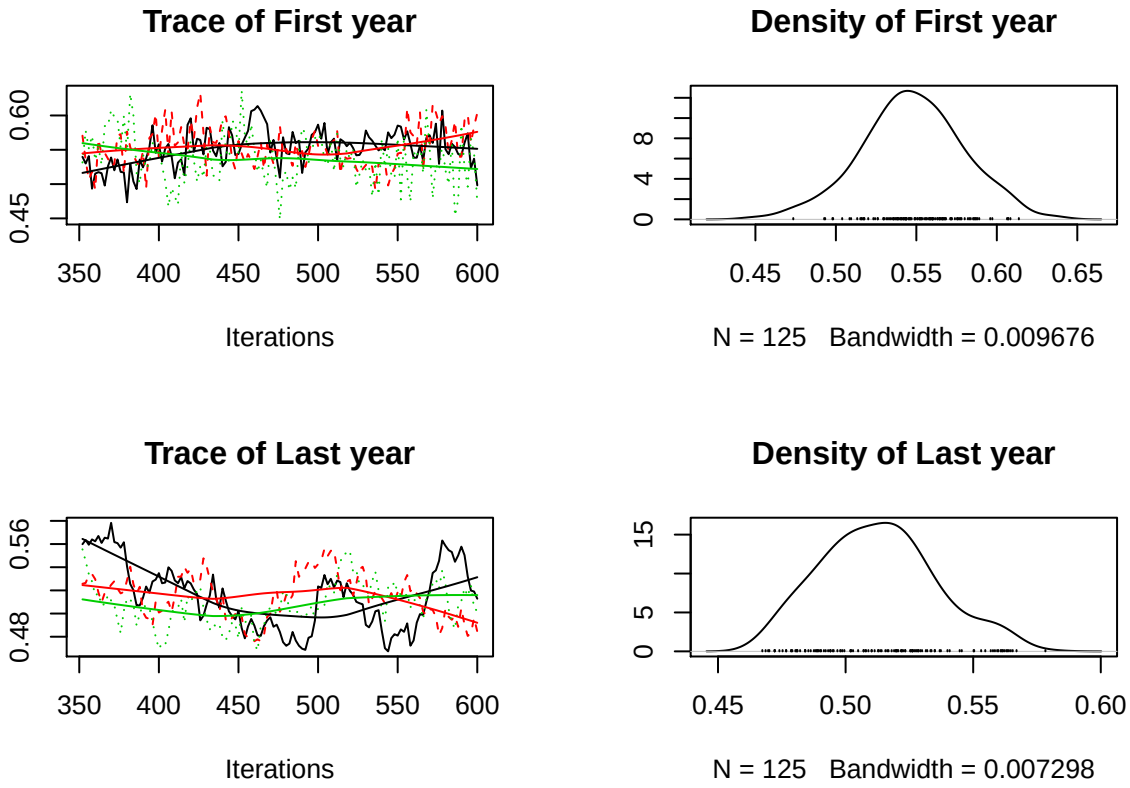
The function returns a plot to your screen which is a diagnostic plot of the model. When the model has converged (i.e. reached a point where the three chains agree on the answer) the lines on the plots on the left will sit on top of one another and the plots on the right will have a nice bell shape. You can turn off this plot by setting `plot` to `FALSE`. By default the method runs the chains in series. Running them in parallel makes the models run faster (about half the time) but will slow down your computer more. We can change this with the parameter `parallel`. The number of iterations the model runs is controlled by `n.iter` and defaults to 10000. If you can it is better to run it for more iterations, though this will take longer. `m.scale` gives the scale your data is on. It is very important that this is correct, choose from 'loge' (natural log, sometimes simply called 'log'), 'log10' (log to the base 10), or 'logit' (output from models of proportions or probabilities).

Let's implement a few of these changes

```
bma_indicator2 <- bma(data,
                      parallel = TRUE,
                      n.iter = 500,
                      m.scale = 'log10')
```

```
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
##
## Parallel processing completed.
##
## Calculating statistics.......
```

```
##
## Done.
```



Because we have reduced the number of interations the model no longer has a good convergence. The lines on the graphs on the left do not overlap and the graphs on the right are no longer a smooth bell shape.

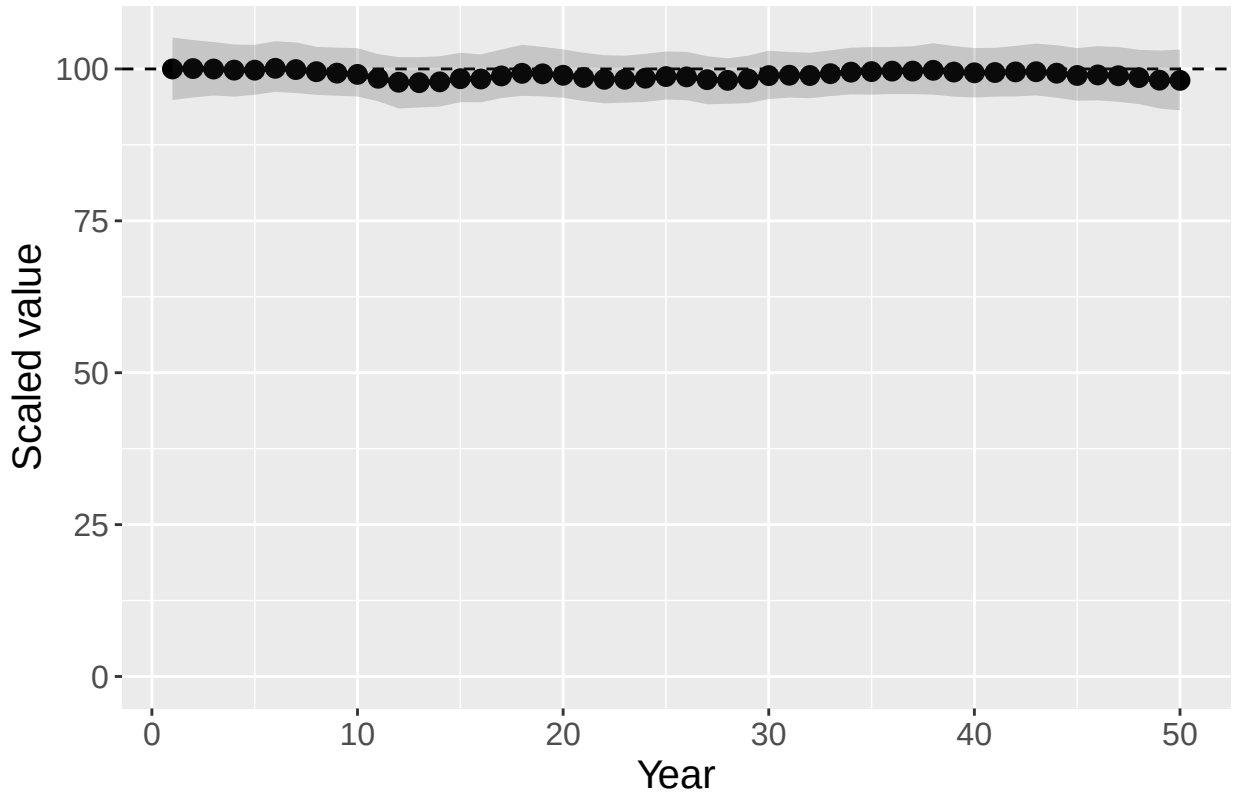
The object that is returned is a data.frame with years as rows and columns giving the year value, index value and confidence intervals. You can write this to a csv using the function `write.csv`.

```
head(bma_indicator)
```

```
##   Year     Index lower2.5 upper97.5
## 1    1 100.00000 94.87787  105.1677
## 2    2 100.06331 95.30973  104.7519
## 3    3  99.98624 95.61992  104.4292
## 4    4  99.80442 95.44291  104.0216
## 5    5  99.79806 95.76269  103.9703
## 6    6 100.10286 96.22112  104.5534
```

We can use the plotting function in BRCindicators to plot the results of this analysis, which in this case are not all that interesting!

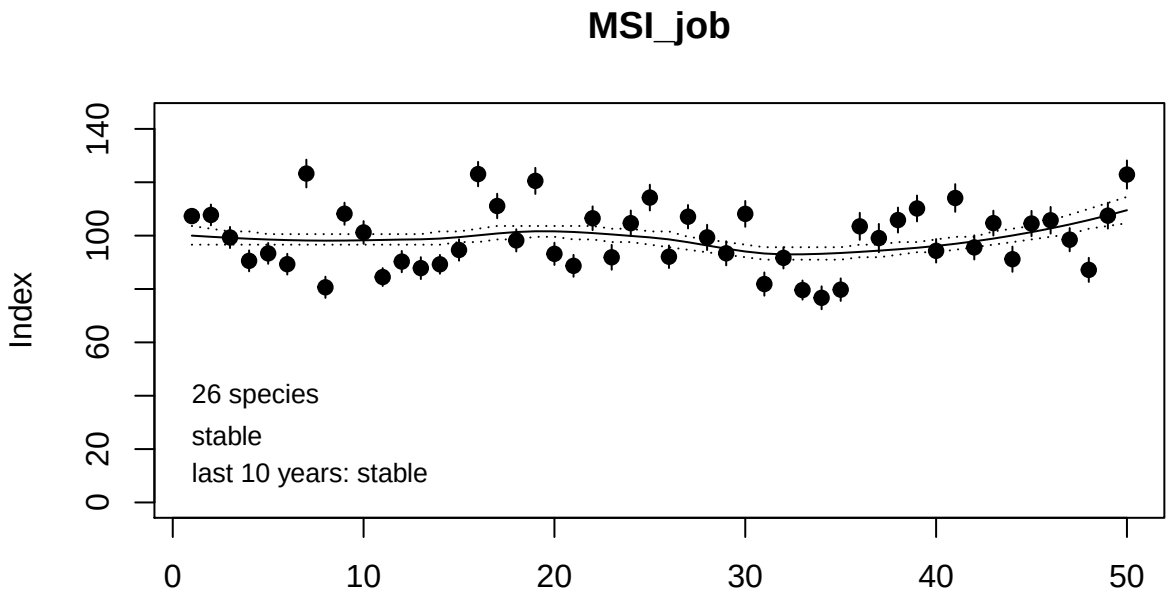
```
plot_indicator(indicator = bma_indicator[,'Index'],
               CIs = bma_indicator[,c(3,4)])
```



Multi-species Indicator

The multi-species indicator method was developed by Statistics Netherlands and the code is made available on their website (https://www.cbs.nl/en-gb/society/nature-and-environment/indices-and-trends--trim--/msi-tool). To find out more about the inner working of this method please read the detailed documentation on the authors website. Here is a simple example of how this method runs in `BRCindicators`.

```
# Create some example data in the format required
data <- data.frame(species = rep(letters, each = 50),
                   year = rep(1:50, length(letters)),
                   index = runif(n = 50 * length(letters), min = 1, max = 10),
                   se = runif(n = 50 * length(letters), min = 0.01, max = .8))

# Alternativly I could read in data from a csv
# read.csv('path/to/my/data.csv')

# Run the MSI function
msi_out <- msi(data)
```

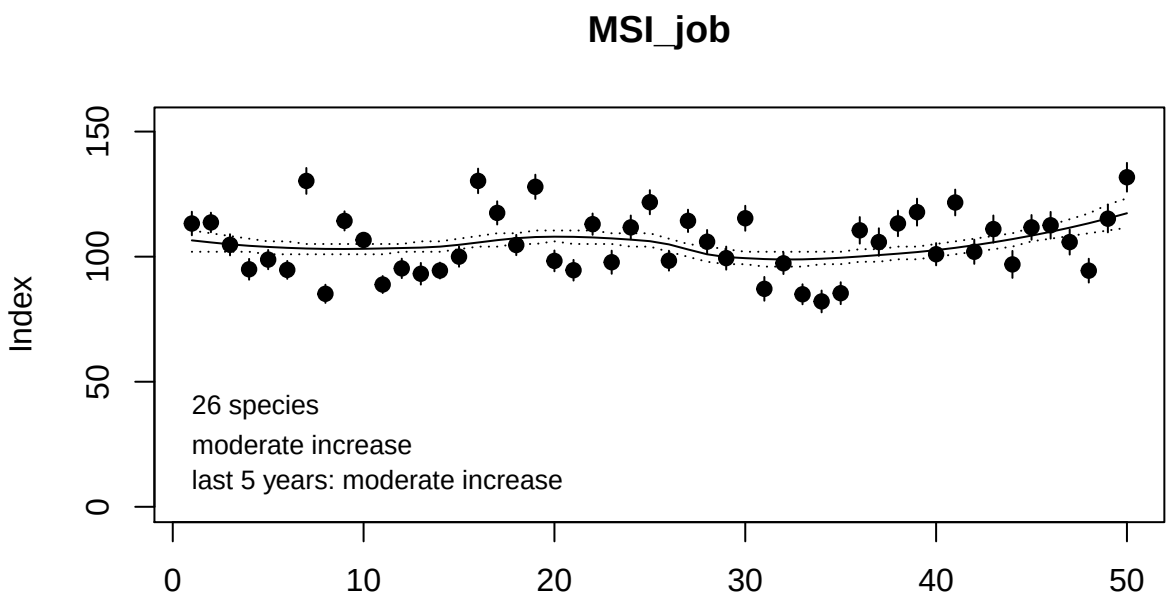



```
# I can capture the output figures too
# pdf('test.pdf')
#   msi_out <- msi(data)
# dev.off()
```

The code returns two plots to the console, the first plot shows the coefficient of variation (CV) for each of the species. Species with high values of CV may adversly effect the relaibility of hte trend estimation. Use this graph to identify the CV values of the species and use the `maxCV` parameter to set a threshold above which species will be excluded. The results of excluding species in this way can be tested by comparing trend plots. The second plot shows the smoothed trend and the MSI values. These two figures can be captured in the usual way in R by using `pdf()` for example. In the example I create a dataset from random numbers but usually you would use `read.csv()` to read in data from a local file.

Here is a second example which sets some additional parameters. The parameters for `msi` get passed to `msi_tool` so to see a list of all the parameters you can change look at the help documentation in `msi_tool` usign `?msi_tool` at the R console. I cover most of hte important ones here.

```
msi_out <- msi(data,
               nsim = 500, # The number of Mote Carlo simulations
               SEbaseyear = 10, # The year to index on
               plotbaseyear = 15, # The year to set as 100 in plots
               index_smoot = 'INDEX', # plotbaseyear uses MSI not trend
               span = 0.7, # 'wigglyness' of line, between 0 and 1
               lastyears = 5, # last X years of time series for short-term trends
               maxCV = 10, # maximum allowed Coefficient of Variation
               changepoint = 25, # compare trends before and after this year
               truncfac = 8, # max year-to-year index ratio
               TRUNC = 5, #set all indices below TRUNC to this
               plot = TRUE # should the plots be returned?)
               )
```



This set of parameters is unrealistic but shows the options available. Note that in the second graph the year 10 point now has a se = 0, year 15 MSI is set to 100, and the short term trend is reported for the last 5 years.

The analysis also returns data which provide more insights into the analysis and let you create your own plots if required.

```
# The returned object has 2 elements
head(msi_out$results)
```

```
##   year    MSI sd_MSI lower_CL_MSI upper_CL_MSI  Trend lower_CL_trend
## 1    1 108.78   5.24        98.98       119.55 120.53         115.55
## 2    2 121.93   5.11       112.31       132.37 119.13         114.40
## 3    3 139.76   5.54       129.32       151.03 117.84         114.40
## 4    4 124.11   5.29       114.18       134.93 116.74         113.26
## 5    5 113.84   5.27       103.98       124.64 115.78         113.26
## 6    6 103.93   5.05        94.49       114.31 114.97         112.14
##   upper_CL_trend      trend_class
## 1         126.43 moderate_decline
## 2         123.93 moderate_decline
## 3         121.48 moderate_decline
## 4         120.27           stable
## 5         119.07           stable
## 6         117.89           stable
```

The first of the two elements (`results`) returned gives all the data, and a little more, that is presented in the second figure.

```
# The returned object has 2 elements
msi_out$trends
```

```
##                          Measure   value     significance
## 1                  overall trend  0.9983 moderate decline
```

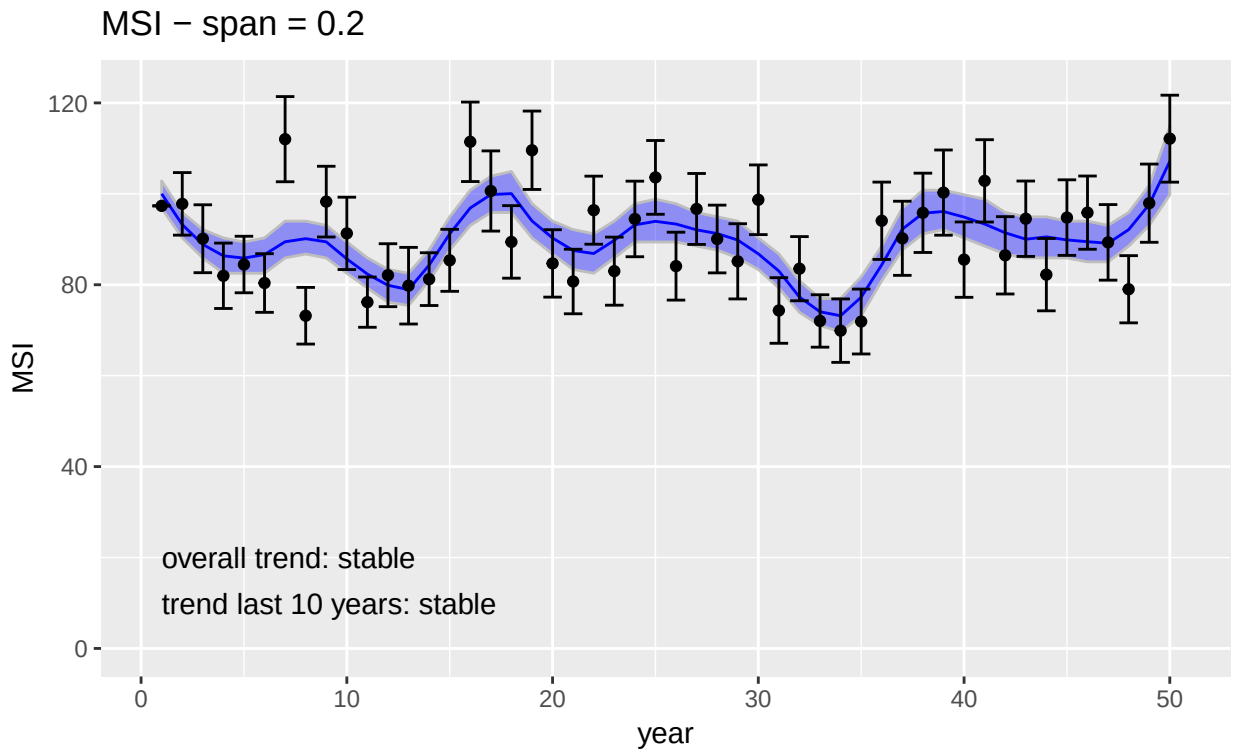
```
## 2                  SE overall trend  0.0005
## 3                trend last 5 years  0.9877           stable
## 4             SE trend last 5 years  0.0172
## 5                  changepoint (25) 25.0000
## 6     trend before changepoint (25)  0.9999           stable
## 7  SE trend before changepoint (25)  0.0013
## 8      trend after changepoint (25)  0.9966 moderate decline
## 9   SE trend after changepoint (25)  0.0013
## 10                        % change -9.8900           p<0.01
## 11                     SE % change  3.4200
## 12            % change last 5 years  0.4310             n.s.
## 13         SE % change last 5 years  1.5260
## 14                     changepoint      NA             n.s.
```

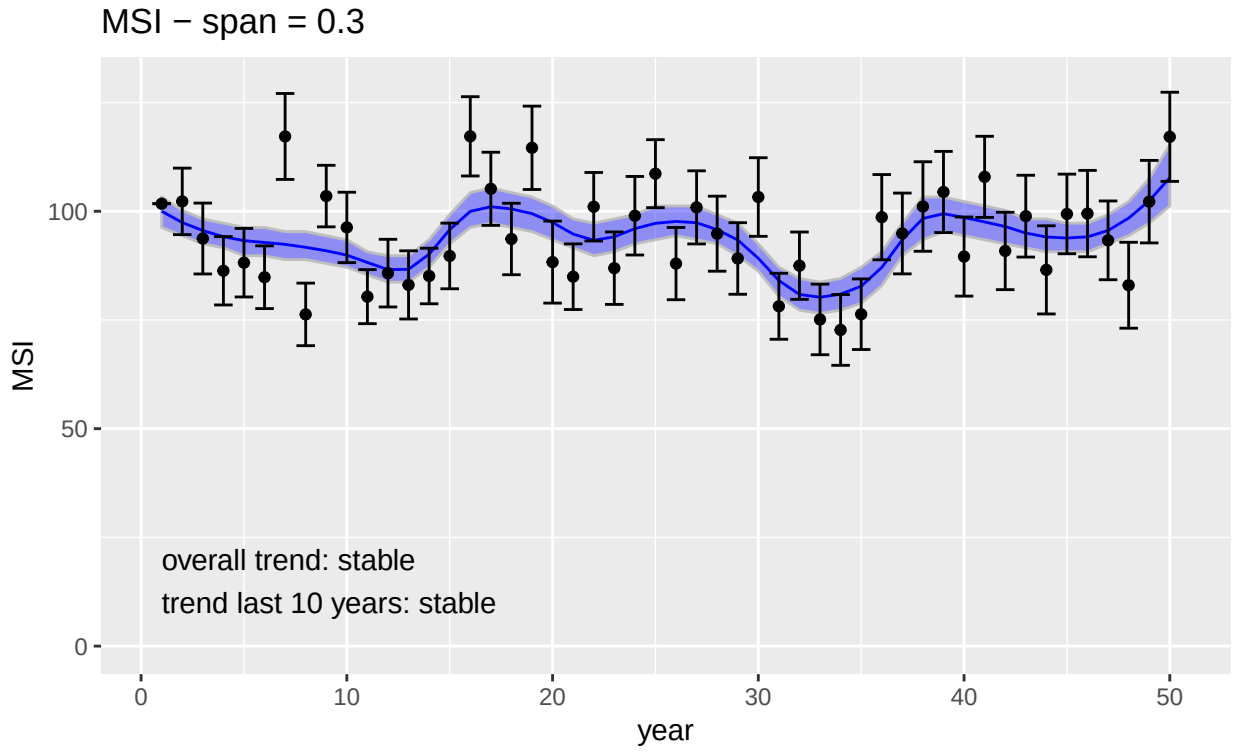
```
# I could write this as a csv too
# write.csv(msi_out$trends, file = 'path/to/my/output.csv')
```

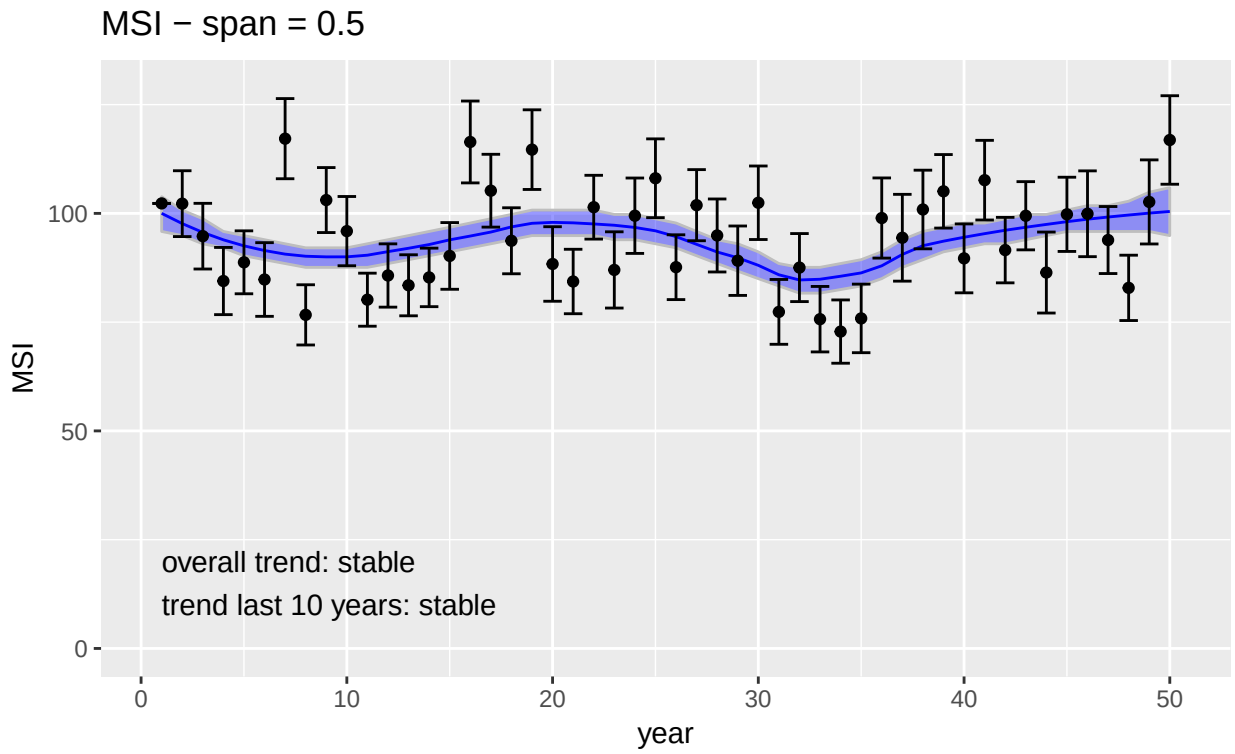
The second element (`trends`) returned give a summary of various trend assessments across the time series.

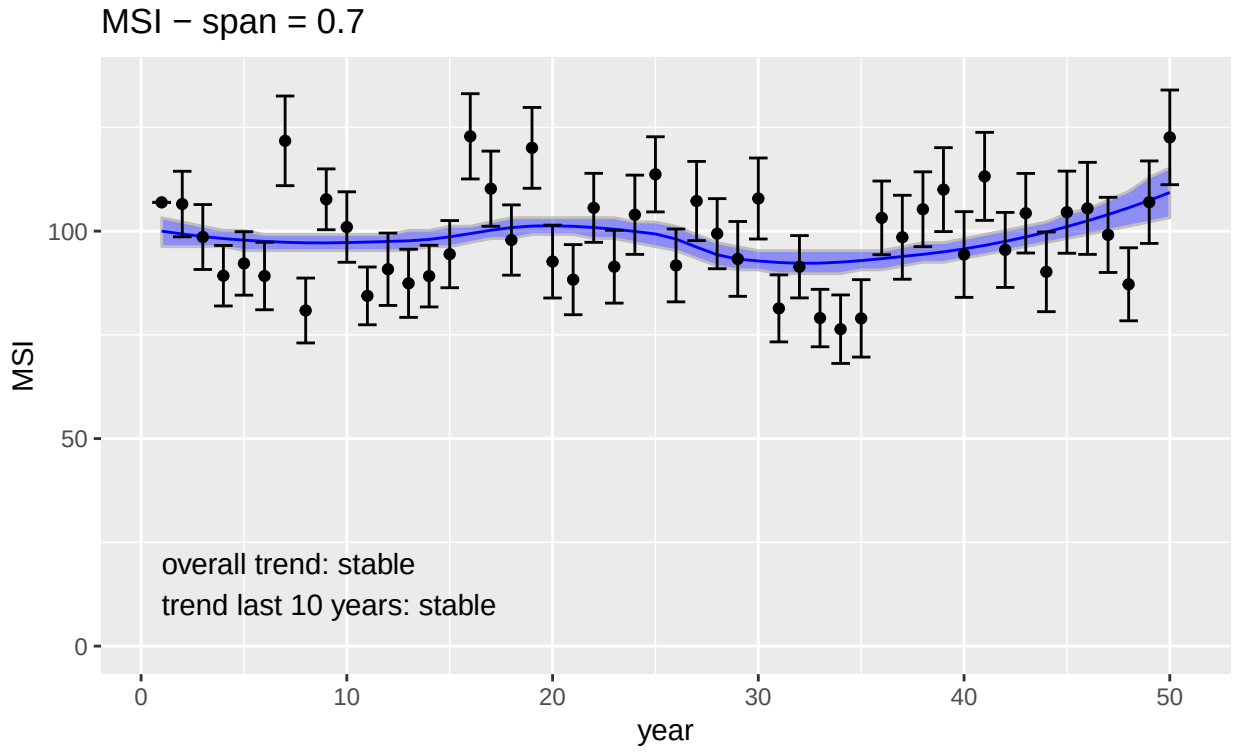
We have also added a plot method for the MSI output which provides a plot similar to that of the second figure we have seen already. Lets use this plot method to explore the effect of changing the span value in the analysis

```
for(i in c(0.2, 0.3, 0.5, 0.7)){ # use a range of values for span

  msi_out <- msi(data, span = i, # span is set to i
                 nsim = 200, plot = FALSE)
  print( # print makes the plot visible in the for loop
    plot(msi_out, title = paste('MSI - span =', i)) # plot
  )

}
```


MSI − span = 0.2
120
80
40
0
MSI
overall trend: stable
trend last 10 years: stable
0
10
20
30
40
50
year


MSI − span = 0.3
100
50
0
MSI
overall trend: stable
trend last 10 years: stable
0
10
20
30
40
50
year





As the value of span gets closer to 1 the trend line gets smoother.

Creating a custom pipeline function

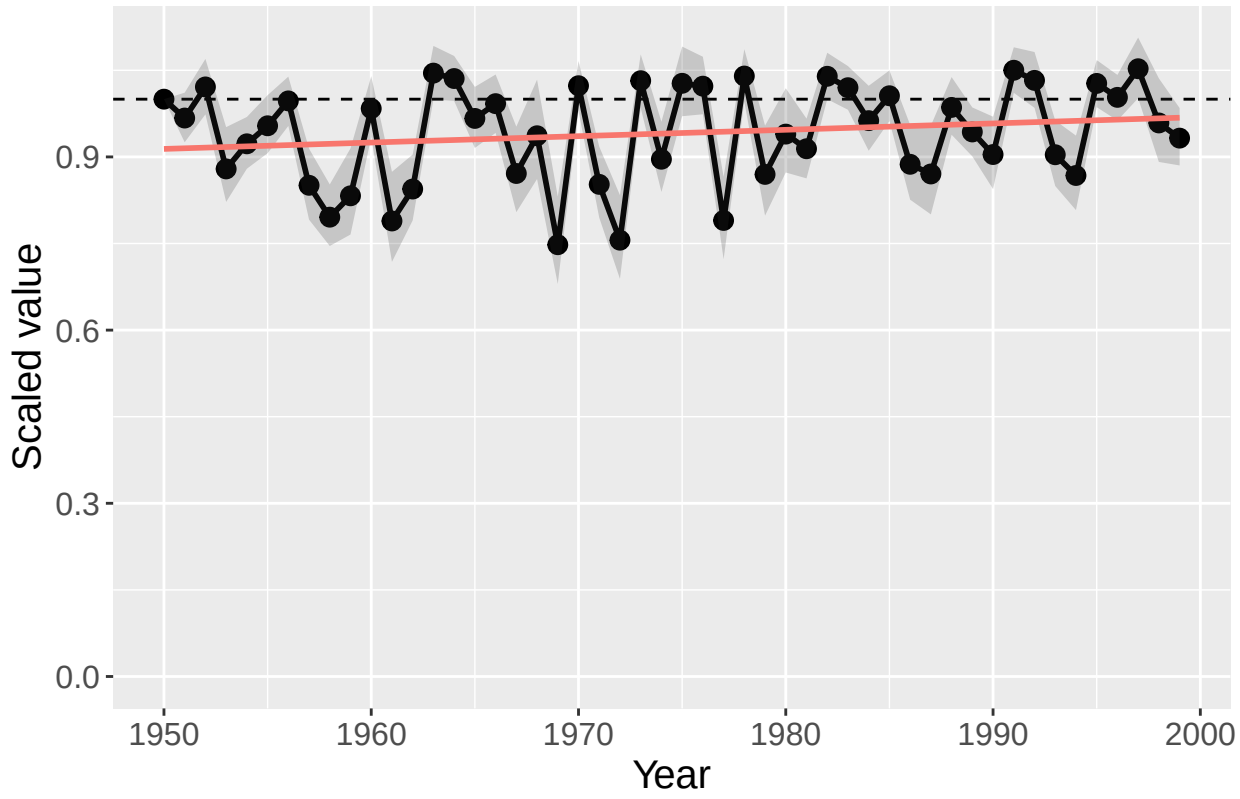
We have demonstrated how you might run the indicator functions one at a time, however in a 'pipeline' we want data to flow through seamlessly. Additionally there are a number of parameters in the functions that we have not shown you that you might find useful. Here is an example of how you can create your own pipeline function. Our function will wrap around the functions described above, setting the parameters to meet our needs. Once we have done this it will allow use to execute our pipeline in one line.

```
# I call my function 'run_pipeline' and the only arguement it
# takes is the directory of sparta's output
run_pipeline <- function(input_dir){

  require(sparta)
  require(BRCindicators)

  # Create the trends summary
  trends_summary <- summarise_occDet(input_dir = input_dir)

  # Rescale the values and get the indicator values
  # Here I set the index to 1 and change the value limits
  rescaled_trends <- rescale_species(Data = trends_summary,
                                     index = 1,
                                     max = 100,
                                     min = 0.001)

  # Bootstrap the indicator to get CIs
  scaled_species <- rescaled_trends[,!colnames(rescaled_trends) %in% c('year', 'indicator')]
  # This time I set the iterations to twice the default and
  # use custom confidence intervals
  indicator_CIs <- bootstrap_indicator(Data = scaled_species,
                                       CI_limits = c(0.25, 0.75),
                                       iterations = 20000)

  # Get the smoothed indicator line
  smoothed_indicator <- GAM_smoothing(rescaled_trends[,'indicator'])

  # This time I specify the years and index value
  plot_indicator(indicator = rescaled_trends[,'indicator'],
                 year = rescaled_trends[,'year'],
                 index = 1,
                 CIs = indicator_CIs,
                 smoothed_line = smoothed_indicator)

  ## I'll return all my data
  return(cbind(smoothed_indicator, indicator_CIs, as.data.frame(trends_summary)))
 }
```

Once we have created this function we can run this pipeline on a directory in one line, or put it in a loop to run across many directories.

```
# Now we can run the pipeline in one line, like a boss
indicator_data <- run_pipeline(input_dir = '~/Testing_indicator_pipe')
```

```
## Loading data...done
## Running bootstrapping for 20000 iterations...done
```




```
head(indicator_data)
```

```
##   smoothed_indicator  quant_25  quant_75 year         a         b
## 1          0.9138432 1.0000000 1.0000000 1950 0.6745699 0.7173656
## 2          0.9149464 0.9253322 1.0113132 1951 0.6675806 0.6460215
## 3          0.9160497 0.9737014 1.0696681 1952 0.4059677 0.6024731
## 4          0.9171529 0.8219554 0.9515864 1953 0.1990860 0.5976344
## 5          0.9182562 0.8794916 0.9691775 1954 0.5780108 0.5145699
## 6          0.9193594 0.9064665 1.0069338 1955 0.2000000 0.4475806
##           c         d         e         f         g          h         i
## 1 0.4802151 0.5568280 0.5884946 0.5057527 0.7402151 0.52607527 0.3404301
## 2 0.5637097 0.6809677 0.4640860 0.4447312 0.7330645 0.28741935 0.6885484
## 3 0.5306989 0.5035484 0.6218280 0.3743548 0.8120430 0.68682796 0.8302688
## 4 0.4347312 0.5723656 0.6150538 0.6258602 0.7101075 0.05032258 0.5569892
## 5 0.6909677 0.5766667 0.7106452 0.8705376 0.6783333 0.21607527 0.5248387
## 6 0.5319892 0.4764516 0.8084946 0.5811290 0.8016129 0.79473118 0.1360215
##           j         k         l         m         n         o         p
## 1 0.7691935 0.7781720 0.7627419 0.5490860 0.9055914 0.5089247 0.5126344
## 2 0.5700000 0.3858065 0.8473656 0.6853763 0.7496237 0.9110753 0.8044086
## 3 0.4464516 0.9057527 0.7373118 0.7177419 0.8656452 0.8820968 0.5419355
## 4 0.8590860 0.5333871 0.5558602 0.7541935 0.8749462 0.8529032 0.3788710
## 5 0.6415591 0.4744624 0.7397312 0.3079032 0.5227419 0.8681183 0.5830108
## 6 0.5844086 0.5594624 0.7416129 0.8600000 0.8168817 0.8223656 0.9411828
##           q         r         s         t         u         v         w
## 1 0.8418280 0.3869355 0.8094086 0.9240323 0.8798925 0.7795161 0.9172581
## 2 0.7586022 0.5087097 0.7891935 0.5512366 0.9248925 0.4275806 0.8763441
```

```
## 3 0.9354301 0.8917204 0.9319892 0.4596237 0.9089247 0.7392473 0.7803763
## 4 0.9145161 0.6189247 0.7883333 0.8891398 0.6998387 0.8875269 0.7385484
## 5 0.8787634 0.6071505 0.6269355 0.8125806 0.5819892 0.9118280 0.7530108
## 6 0.7818280 0.5579570 0.7590860 0.7274194 0.9422581 0.8439785 0.8212903
##           x         y         z
## 1 0.8746774 0.8625806 0.9073118
## 2 0.7978495 0.9104301 0.9095161
## 3 0.6747849 0.9007527 0.7707527
## 4 0.8668817 0.8252688 0.6954301
## 5 0.5406452 0.9445161 0.6822581
## 6 0.8452688 0.8245699 0.6967742
```