SIDFEx R-package user guide

Helge Goessling

2019-07-03

- What is SIDFEx?
- Requirements and setting up the SIDFEx package
- Downloading forecast and observational data
- Exploring and subselecting the index
- Reading forecast and observational data
- Plotting examples
- Remapping and adjusting data in time and space
- Quantative forecast evaluation
- Extracting real-time forecasts
- What else?
- Acknowledgement

What is SIDFEx?

This package provides tools to analyse data of the Sea Ice Drift Forecast Experiment, which is *a community effort to collect and analyse Arctic sea-ice drift forecasts at lead times from days to a year. Forecasts are made with various methods for drifting sea-ice buoys and, ultimately, the trans-Arctic MOSAiC drift campaign*. General information about SIDFEx, which is an activity of the WMO Year of Polar Prediction (YOPP), can be found at https://www.polarprediction.net/yopp-activities/sidfex/. This site also provides a *Background and guidelines* document which describes the plain-text format of the original forecasts submitted to SIDFEx in detail.

SIDFEx comprises forecasts of how certain assets ("targets") drift with the Arctic sea ice. Many of the forecasts are available in (near-)real-time and, depending on the forecast system they stem from, forecasts consist of single trajectories or ensembles of trajectories. Forecasts also vary regarding their length (the forecast lead time), their time resolution, the set of available initial times and targets, and so forth. All this makes the analysis of the data a non-trivial task which is meant to be facilitated by this package.

The package allows you to download SIDFEx forecasts and corresponding observations, to explore which forecasts are available, and to analyse the forecasts and assess their quality. Among other things, functions are provided to remap (interpolate) the time axes of forecasts and observations and to rotate forecast trajectories - for example to correct their initial location (if required) or to derive real-time forecast from slightly outdated forecasts by adjusting the trajectories such that the latest observed location is accounted for.

This user guide is meant to help you getting started with the SIDFEx package, taking you from the installation over data retrieval to a number of analysis examples. After reading this guide and applying the code chunks, possibly with your own modifications, you should be equipped with the basic knowledge needed to start exploring and analysing the SIDFEx data in more detail. We would be more than happy if you could share any insights you get with us, and we are glad to help and/or discuss should you have any technical or scientific questions or suggestions.

Requirements and setting up the SIDFEx package

The package has been developed on and for UNIX (Mac/Linux) systems and might fail on Windows. In case you are eager to analyse SIDFEx data but are restricted to Windows systems and encounter problems, please contact us and we can consider taking better care of platform independence.

Given that the drift trajectories are defined on a sphere, operations such as finding the centroid of a point cloud or adjusting (*rotating*) trajectories necessitate somewhat complex computations. For such tasks, `SIDFEx` makes use of another R-package `spheRlab`, which is not on CRAN but can be obtained from GitHub (just like `SIDFEx`). You can download the package from https://github.com/FESOM/spheRlab and build and install it manually; general explanations for how to do that can be found easily online. Alternatively, you can use the `devtools` package, which is available from CRAN and can thus in turn be installed directly by running `install.packages("devtools")`. Then, to install `spheRlab` with `devtools`, run

```r
library(devtools)
devtools::install_github("FESOM/spheRlab")
```

`spheRlab` can also be used for plotting maps, examples of which are provided below.

The most convenient way to work with the SIDFEx package is to setup a file named `.SIDFEx` in your home directory. This file will be read to determine where the SIDFEx forecasts and observations and the forecast data index are located on your hard drive. For example, you can create a directory `SIDFEx` in your home directory and three subdirectories named `obs`, `fcst`, and `index` therein. The corresponding `.SIDFEx` file should contain the following three lines:

```
data.path.fcst = "~/SIDFEx/fcst"
data.path.obs = "~/SIDFEx/obs"
indexTable.path.in = "~/SIDFEx/index"
```

Alternatively, the functions that use these variables to find the respective data, for example `sidfex.read.fcst()` and `sidfex.read.obs()`, have function arguments `data.path` and/or `indexTable.path` to specify these manually, but that can be inconvenient and error-prone and is thus not recommended. In the following it is assumed that the paths are specified in a `.SIDFEx` file in the home directory.

Now that we've installed (and setup) `spheRlab` and `SIDFEx`, we can load the packages:

```r
library(spheRlab)
library(SIDFEx)
```

Downloading forecast and observational data

Each SIDFEx forecast, once submitted by one of the contributing forecast centres, is automatically processed and made publicly available in real-time (<1h delay) at the Cloud Service of the German Climate Computing Centre (DKRZ). Individual forecast files in plain-text format, ordered by contributor GroupIDs, can be accessed individually at https://swiftbrowser.dkrz.de/public/dkrz_0262ea1f00e34439850f3f1d71817205/SIDFEx_processed/. However, if more than a few forecasts are of interest, it is much more convenient and faster to download a corresponding `.tar.gz` file that holds all the forecast data. To retrieve it through this package, run

```r
res = sidfex.download.fcst()
```

After successful retrieval, you will find the plain-text forecast files in the specified data directory (see above) in subdirectories named by the `GroupID`s of the forecast centres.

In fact, this function retrieves not only the forecast data, but also an index of the data (which you will now also find in the corresponding directory). The index is basically a table stored in a native R file format (`data.frame`), each row representing a single forecast file with specific information about this forecast. Therefore this table can easily be used to find certain forecasts based on criteria on the columns, comparable to tables in a database management system. The index thus facilitates exploring and working with the data considerably. Once a local copy of the data and a corresponding index exist, the index also allows one to compare the local data with the remote data to see what new data is available. To compare, run

```
res = sidfex.download.fcst(comparison.mode = TRUE, from.scratch = FALSE)
#> [1] "Index download ..."
#> [1] "Index download done."
#> [1] "Entries only in local index = obsolete files to be deleted: 0"
#> [1] "Entries only in remote index = new files to be downloaded: 0"
#> [1] "Identical entries = local files already up-to-date: 60736"
#> [1] "Entries for same files with differences = files to be downloaded and to
replace local ones: 0"
#> [1] "To execute these downloads and other changes (if present as indicated above),
resubmit command with comparison.mode=FALSE ."
#> [1] "Returning index of files that would be downloaded with comparison.mode=FALSE,
and (if not from.scratch) the more detailed result of
sidfex.fcst.search.compareIndexTables"
#> Warning in sidfex.download.fcst(comparison.mode = TRUE, from.scratch =
#> FALSE): Note that, with from.scratch=FALSE, it can not be excluded that the
#> local data differs from the remote data in aspects that are not captured by
#> the index. But that's very unlikely.
```

If not too many new forecasts are available and if you want to save download volume, you can rerun the function with `comparison.mode = FALSE` and `from.scratch = FALSE` to retrieve only individual new files. However, `from.scratch = TRUE` will be faster most of the time.

It can be convenient to have the index of the (local) forecast data available in a local variable. To dump the index into a data frame (a type of table common in R) named `index` and to get an overview of its content, run

```
index = sidfex.load.index()
str(index)
#> 'data.frame':    60736 obs. of  20 variables:
#>  $ File                   : chr  "awi001_ClimRunVers2017Jul_127317_2017-120_001"
"awi001_ClimRunVers2017Jul_127317_2017-120_002"
"awi001_ClimRunVers2017Jul_127317_2017-120_003"
"awi001_ClimRunVers2017Jul_127317_2017-120_004" ...
#>  $ GroupID                : chr  "awi001" "awi001" "awi001" "awi001" ...
#>  $ MethodID               : chr  "ClimRunVers2017Jul" "ClimRunVers2017Jul"
"ClimRunVers2017Jul" "ClimRunVers2017Jul" ...
#>  $ TargetID               : chr  "127317" "127317" "127317" "127317" ...
#>  $ InitYear               : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
...
#>  $ InitDayOfYear          : num  120 120 120 120 120 120 120 120 120 120 ...
#>  $ EnsMemNum              : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ SubmitYear             : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
```

```
...
#>  $ SubmitDayOfYear       : num  292 292 292 292 292 ...
#>  $ ProcessedYear         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
...
#>  $ ProcessedDayOfYear    : num  292 292 292 292 292 ...
#>  $ Delay                 : num  172 172 172 172 172 ...
#>  $ nTimeSteps            : int  89 100 87 103 73 110 105 103 76 112 ...
#>  $ FirstTimeStepYear     : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
...
#>  $ FirstTimeStepDayOfYear: num  120 120 120 120 120 120 120 120 120 120 ...
#>  $ LastTimeStepYear      : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
...
#>  $ LastTimeStepDayOfYear : num  208 219 206 222 192 229 224 222 195 231 ...
#>  $ FcstTime              : num  88 99 86 102 72 109 104 102 75 111 ...
#>  $ EnsParentFile         : chr  "awi001_ClimRunVers2017Jul_127317_2017-120_010"
"awi001_ClimRunVers2017Jul_127317_2017-120_010"
"awi001_ClimRunVers2017Jul_127317_2017-120_010"
"awi001_ClimRunVers2017Jul_127317_2017-120_010" ...
#>  $ EnsSize               : num  10 10 10 10 10 10 10 10 10 10 ...
```

The first line of the output tells us how many forecast trajectories (“`obs.`” = number of rows, counting all members of ensemble forecasts individually; note that this has nothing to do with actual *observed* buoy positions, but is R-internal terminology for `data.frame` objects) are available, and how many columns (“`variables`”) the index provides, which are listed underneath. For example, the `$TargetID` column provides the target each forecast corresponds to. We can list all targets by selecting the corresponding column of the index table as follows:

```
unique(index$TargetID)
#>  [1] "127317"          "139939"          "300234060434550"
#>  [4] "300234060436000" "300234060834110" "300234061872720"
#>  [7] "300234062738010" "300234063803010" "300234063991680"
#> [10] "300234065802030" "300234060430010" "300234062880820"
#> [13] "300234065495020" "300234065801030" "300234066030190"
#> [16] "300234066030330" "300234066031190" "300234066036110"
#> [19] "300234066711310" "300234066713470" "300234066830700"
```

To download observational data for all SIDFEx targets, you can use the function `sidfex.download.obs()`. The argument `TargetID` can be used to specify which observations shall be retrieved. However, often the easiest way is to provide the argument `index`, that is, the index of the forecast data for which observations shall be retrieved, directly:

```
obs = sidfex.download.obs(index = index)
```

The above download functions rely on the availability of the respective servers and on finding the data in the expected locations and format. If the download fails, please let us know so we can investigate the cause.

Now that we have all data locally, let’s start by exploring and subselecting the index, then read some data, then inspect some raw data, and finally look at some more sophisticated examples.

Exploring and subselecting the index

Above we have already used `unique(index$TargetID)` as an example how you can use the index directly to get an overview on the available forecasts. In a similar way we can use any of the columns of the index (listed above) to explore the data. The following example shows which forecast systems (GroupID_MethodID combinations) are available, and for each of them (i) how many individual forecast trajectories are available, (ii) the (average) forecast length (lead time), (iii) the (average) ensemble size, and (iv) the (average) number of time steps per forecast:

```
systms = paste(index$GroupID, index$MethodID, sep = "_")
systms_unique = unique(systms)
tbl = as.data.frame(matrix(nrow = length(systms_unique), ncol = 4))
rownames(tbl) = systms_unique
colnames(tbl) = c("nFcst", "leadtimerange", "enssize", "nTimeSteps")
for (i in 1:length(systms_unique)) {
    indices = (systms == systms_unique[i])
    tbl[i, 1] = sum(indices)
    tbl[i, 2] = mean(index$FcstTime[indices])
    tbl[i, 3] = mean(index$EnsSize[indices])
    tbl[i, 4] = mean(index$nTimeSteps[indices])
}
print(tbl)
#>                                nFcst leadtimerange   enssize nTimeSteps
#> awi001_ClimRunVers2017Jul      1090     293.09725 10.000000  294.23578
#> awi001_ClimRunVers2019May      3550     221.04169 10.000000  222.10789
#> awi003_iceocean                 780      78.76562 20.000000   80.08077
#> eccc001_caps                   1226       2.00000  1.000000   97.00000
#> eccc001_giops                  3897      10.00000  1.000000  481.00000
#> ecmwf001_SEAS5                 7191     124.00000 51.000000  125.00000
#> esrl001_SeaIceVelocity         3729      10.00000  1.000000   11.00000
#> metno001_RK2                   9303       9.00000  1.000000    9.00000
#> nrl001_flatearth24              389     120.40231 10.254499  120.90231
#> nrl001_gofs3.1-shortrange      3904       6.50000  1.000000    7.00000
#> nrl001_navyespc-subseasonal    2148      43.50000  4.000000   44.00000
#> ucl001_IceOceanModelTracer      350     192.65714 10.000000  193.65714
#> ukmo001_cplNWP                 7567      10.60513  1.000000   11.60513
#> ukmo001_cplNWP-HR              7658      11.00000  1.000000   12.00000
#> ukmo001_FOAM                   7628       8.00000  1.000000    9.00000
#> uo001_mpasCESM                  114      90.50000  4.438596  363.00000
#> uw001_IceOceanModel             212      81.13208  4.000000   82.13208
```

The table reveals that all quantities considered vary considerably between the systems. We find for example that short-range (`leadtimerange<=10days`) forecasts tend to be deterministic (`enssize=1`) whereas long-range forecasts tend to be probabilistic (`enssize>1`).

Similarly, in the following example we explore with how much delay relative to real-time the forecasts have been submitted to SIDFEx. Using the default parameters of the `quantile()` function, we get the 0%,25%,…,100% quantiles of the delay distributions, looping over the available forecast systems:

```
delay_quantiles = NULL
for (i in 1:length(systms_unique)) {
    delay_quantiles = rbind(delay_quantiles, quantile(index$Delay[systms ==
        systms_unique[i]]))
}
rownames(delay_quantiles) = systms_unique
```

```
print(delay_quantiles)
#>                                   0%     25%     50%       75%     100%
#> awi001_ClimRunVers2017Jul     0.241   1.242  64.960 139.85600  249.057
#> awi001_ClimRunVers2019May    16.436 196.116 468.563 799.06800 1531.582
#> awi003_iceocean              82.117  95.613 117.620 139.91200  166.616
#> eccc001_caps                  0.668   0.668   0.669   0.66900   27.808
#> eccc001_giops                 0.547   0.648   0.649   3.60000  365.649
#> ecmwf001_SEAS5                2.619   4.794   4.800  33.49600  153.658
#> esrl001_SeaIceVelocity        1.167   1.201   1.243   4.93800  145.085
#> metno001_RK2                  2.001   2.043  57.782 386.30800 1175.397
#> nrl001_flatearth24            7.117  34.137  48.459  76.46500  115.462
#> nrl001_gofs3.1-shortrange     0.855   0.863  13.206 102.20775  212.217
#> nrl001_navyespc-subseasonal   1.368   3.368  11.218  92.18100  199.183
#> ucl001_IceOceanModelTracer   16.834  16.836  81.627 320.83500  381.837
#> ukmo001_cplNWP                1.438   1.627  28.435 270.66500  512.540
#> ukmo001_cplNWP-HR             5.245  41.210 195.028 492.82500  762.698
#> ukmo001_FOAM                  1.437   1.627  26.691 268.66500  512.540
#> uo001_mpasCESM               37.806  37.807  67.806  67.80775   97.807
#> uw001_IceOceanModel          12.912  25.042  37.994 117.98500  147.980
```

The 0%-quantile gives the minimum delay (in days), the 100%-quantile the maximum delay. Note that also those contributors that submit forecasts in near-real-time (lower quantiles < few days) typically have also provided retrospective forecasts (higher quantiles > few days).

When it comes to reading the actual forecast data for further analyses (see next sections), it is useful first to make a subselection of the complete SIDFEx forecast index by keeping only selected entries (forecasts). The index can be subselected manually, e.g., by a statement of the type `subindex=index[index$GroupID=="esrl001",]`. However, in particular date-range queries are facilitated using the function `sidfex.fcst.search.extractFromTable()`. We can get the subindex with all `metno001_RK2` forecasts for the target `300234063991680` initialised between 2018 February 1st 00:00 UTC and 2019 April 15th 12:00 UTC as follows:

```
subindex1 = sidfex.fcst.search.extractFromTable(gid = "metno001", mid = "RK2",
    tid = "300234063991680", iy = c(2018, 2019), idoy = c(32, 105.5))
nrow(subindex1)
#> [1] 437
subindex2 = sidfex.fcst.search.extractFromTable(gid = "ecmwf001", mid = "SEAS5",
    tid = "300234063991680", iy = c(2018, 2019), idoy = c(32, 105.5))
nrow(subindex2)
#> [1] 561
```

The output reveals how many forecasts have been selected by these two sets of criteria. No matter if it has been generated manually or with `sidfex.fcst.search.extractFromTable()`, a subindex has the same structure as the complete index, but with only the selected rows retained. Obviously we can explore a subindex as we explored the complete index in the examples above. More importantly, we can use a subindex to read and analyse the corresponding sets of forecasts as described in the following.

Reading forecast and observational data

Reading all SIDFEx forecast data at once is in most cases not advisable because reading >50.000 ascii files is quite slow. Typically it is more useful to read a meaningful subset of the forecasts. It is possible to read forecasts with `sidfex.read.fcst()` by specifying one or more file names directly. However, the recommended way which also enables ensemble merging (see below) is to provide the function with a subindex (see above). In the following we specify `checkfileformat=FALSE` to save time as we know that any forecasts in the SIDFEx database have already passed an automatic format check. (Note that one can also use the function `sidfex.checkfileformat()` to check the format, which can be useful if one intends to submit forecasts to SIDFEx.)

```
subindex3 = sidfex.fcst.search.extractFromTable(gid = "ecmwf001", tid =
"300234063991680",
    iy = 2019, idoy = c(32, 91))
fcst = sidfex.read.fcst(files = subindex3, checkfileformat = FALSE)
names(fcst)
#> [1] "ens.merge" "res.list"  "index"
length(fcst$res.list)
#> [1] 3
str(fcst$res.list[[1]], list.len = 27)
#> List of 26
#>  $ fl                  : chr [1:51]
"~/AWI/SIDFEx/data/forecasts/ecmwf001/ecmwf001_SEAS5_300234063991680_2019-032_001.txt"
"~/AWI/SIDFEx/data/forecasts/ecmwf001/ecmwf001_SEAS5_300234063991680_2019-032_002.txt"
"~/AWI/SIDFEx/data/forecasts/ecmwf001/ecmwf001_SEAS5_300234063991680_2019-032_003.txt"
"~/AWI/SIDFEx/data/forecasts/ecmwf001/ecmwf001_SEAS5_300234063991680_2019-032_004.txt"
...
#>  $ SubmitYear          : int 2019
#>  $ SubmitDayOfYear     : num 36.8
#>  $ ProcessedYear       : int 2019
#>  $ ProcessedDayOfYear: num 37.3
#>  $ GroupID             : chr "ecmwf001"
#>  $ MethodID            : chr "SEAS5"
#>  $ TargetID            : chr "300234063991680"
#>  $ InitYear            : int 2019
#>  $ InitDayOfYear       : num 32
#>  $ InitLat             : num 74.1
#>  $ InitLon             : num -162
#>  $ EnsMemNum           : int [1:51] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ Ntimesteps          : int 125
#>  $ FirstYear           : int 2019
#>  $ FirstDayOfYear      : int 32
#>  $ FirstLat            : num 74.1
#>  $ FirstLon            : num -162
#>  $ LastYear            : int 2019
#>  $ LastDayOfYear       : int 156
#>  $ LastLat             : num 76.7
#>  $ LastLon             : num -172
#>  $ DaysForecastLength: num 124
#>  $ data                :'data.frame':  125 obs. of  107 variables:
#>   ..$ Year        : int [1:125] 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019
...
#>   ..$ DayOfYear   : int [1:125] 32 33 34 35 36 37 38 39 40 41 ...
#>   ..$ Lat         : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon         : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ DaysLeadTime: num [1:125] 0 1 2 3 4 5 6 7 8 9 ...
#>   ..$ Lat1        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon1        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat2        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon2        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat3        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon3        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat4        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon4        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat5        : num [1:125] 74.1 74.1 74.1 74.2 74.3 ...
#>   ..$ Lon5        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat6        : num [1:125] 74.1 74.1 74.1 74.2 74.3 ...
#>   ..$ Lon6        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat7        : num [1:125] 74.1 74.1 74.1 74.1 74.1 ...
#>   ..$ Lon7        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat8        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon8        : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat9        : num [1:125] 74.1 74.1 74.1 74.1 74.2 ...
#>   ..$ Lon9        : num [1:125] -162 -162 -162 -162 -163 ...
#>   ..$ Lat10       : num [1:125] 74.1 74.1 74.1 74.1 74.1 ...
#>   ..$ Lon10       : num [1:125] -162 -162 -162 -162 -162 ...
#>   ..$ Lat11       : num [1:125] 74.1 74.1 74.1 74.2 74.3 ...
#>   ..$ Lon11       : num [1:125] -162 -162 -162 -162 -162 ...
#>   .. [list output truncated]
#>  $ MergedInitLat     : num [1:52] 74.1 74.1 74.1 74.1 74.1 ...
#>  $ MergedInitLon     : num [1:52] -162 -162 -162 -162 -162 ...
```

The output reveals that the returned object, here `fcst`, is a list with three elements, the most important one being `res.list` which is the "results list" with one element for each forecast. By default (`ens.merge=TRUE`), forecast ensembles have been merged, meaning that all members of each ensemble have been merged into one element of `res.list`. If `ens.merge=FALSE` is specified, all individual forecast files would correspond to one element. Instead, here the trajectory data table has columns `Lat1`, `Lon1`, `Lat2`, `Lon2`, … in addition to `Lat` and `Lon`. The latter are ensemble mean locations (barycentres of the respective points clouds), whereas the former are the locations of the individual ensemble members. Also the additional data outside the trajectory table has additional entries that are specific to merged ensembles. For details, see the `Value` section of the function documentation by typing `?sidfex.read.fcst`.

Behind the scenes, the ensemble merging is mostly a straight-forward task. However, because some SIDFEx contributions are lagged-initial-date ensembles and because some ensembles have inconsistent lengths of the individual trajectories (e.g., due to the use of some termination criterium, for example based on a sea-ice concetration threshold), subtle details may happen in the background. In particular, the resulting time axis is inherited from the "parent" of each ensemble as specified in the (sub)index, and if necessary the trajectories of the other ensemble members are interpolated (and possibly truncated) to that time axis using the `spheRlab` function `sl.trajectory.remaptime`.

Before looking more closely at the data, we consider the reading of corresponding observational data. In the current example, we have read data only for one target, and we could simply use `sidfex.read.obs(TargetID="300234063991680")`. Again, however, the recommended way is to use the corresponding index; in this case the relevant targets are identified automatically.

```
obs = sidfex.read.obs(index = subindex3)
str(obs)
#> List of 3
#>  $ filename: chr "~/AWI/SIDFEx/data/observations/300234063991680.txt"
#>  $ TargetID: chr "300234063991680"
#>  $ data    :'data.frame':    24860 obs. of  10 variables:
```

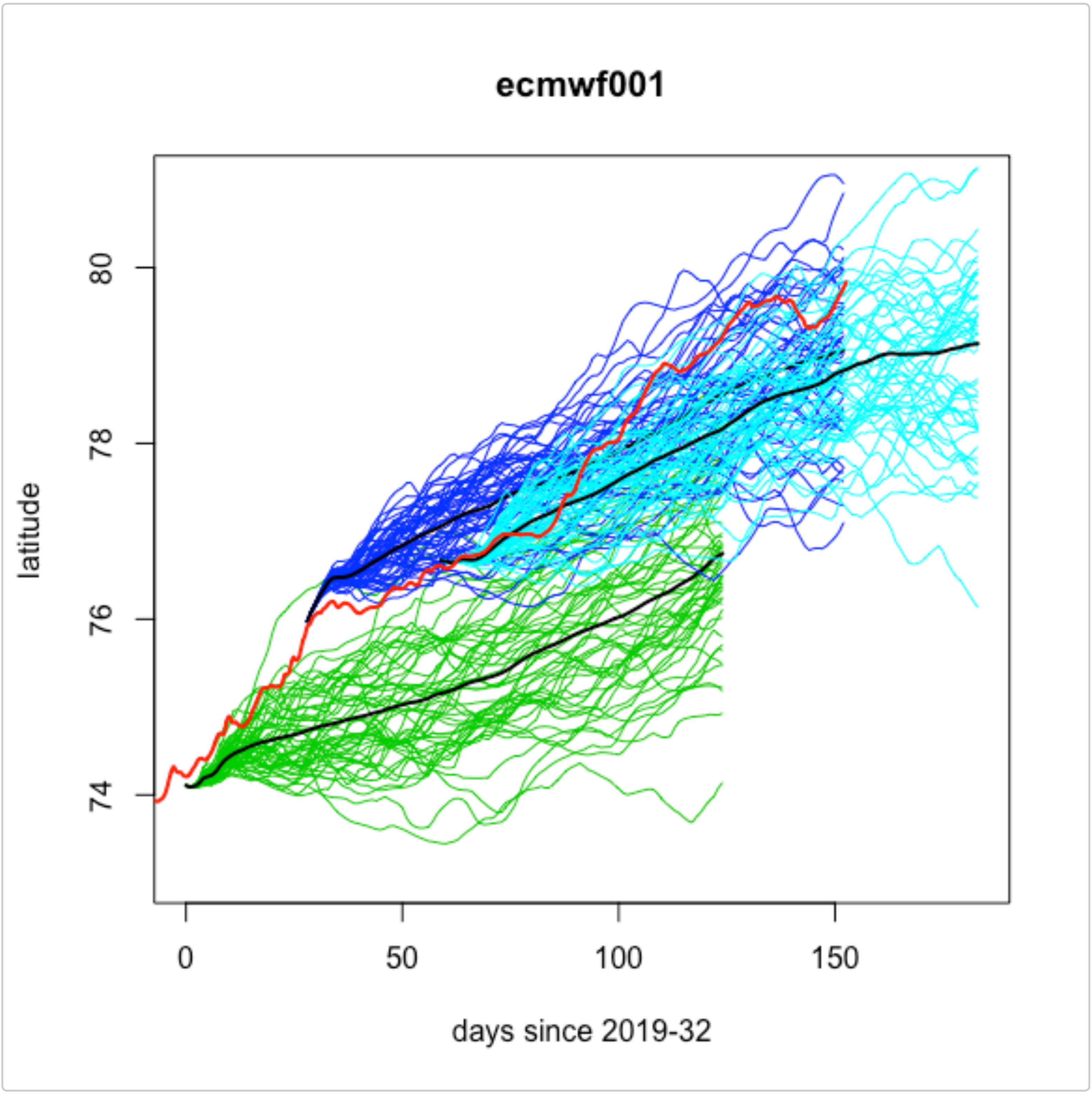
```
#>   ..$ Year      : int [1:24860] 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016
...
#>   ..$ Hour      : int [1:24860] 16 16 16 16 17 17 17 17 17 17 ...
#>   ..$ Min       : int [1:24860] 20 30 40 50 0 10 20 30 40 50 ...
#>   ..$ DOY       : num [1:24860] 163 163 163 163 163 ...
#>   ..$ POS_DOY   : num [1:24860] 163 163 163 163 163 ...
#>   ..$ Lat       : num [1:24860] 33.2 33.2 33.2 33.2 33.2 ...
#>   ..$ Lon       : num [1:24860] -117 -117 -117 -117 -117 ...
#>   ..$ BP        : num [1:24860] 1008 1008 1008 1008 1008 ...
#>   ..$ Ts        : num [1:24860] 15 15 15 15 15 ...
#>   ..$ RelTimeDay: num [1:24860] 0 0.007 0.013 0.02 0.027 ...
```

The most relevant columns of the `data` element are those that define the trajectrories: `Year`, `POS_DOY`, `Lat`, and `Lon`. In this example we see that also co-located physical quantities are available (BP = barometric pressure, Ts = surface temperature), which depends on the considered target and will not be considered further here.

Plotting examples

Now that we have read some forecasts and corresponding observations, let’s have a look at them. For example, let’s plot latitude versus time. Using the subindex from the previous section, the time axis can be defined easily based on the day of the year because all forecast data in the subindex are within 2019. Nevertheless, here we define a new time axis relative to the initial time of the first forecast which would also work for data spanning more than one year. To this end we use `sidfex.ydoy2reltime()`. The remainder of the following code uses just basic `R`:

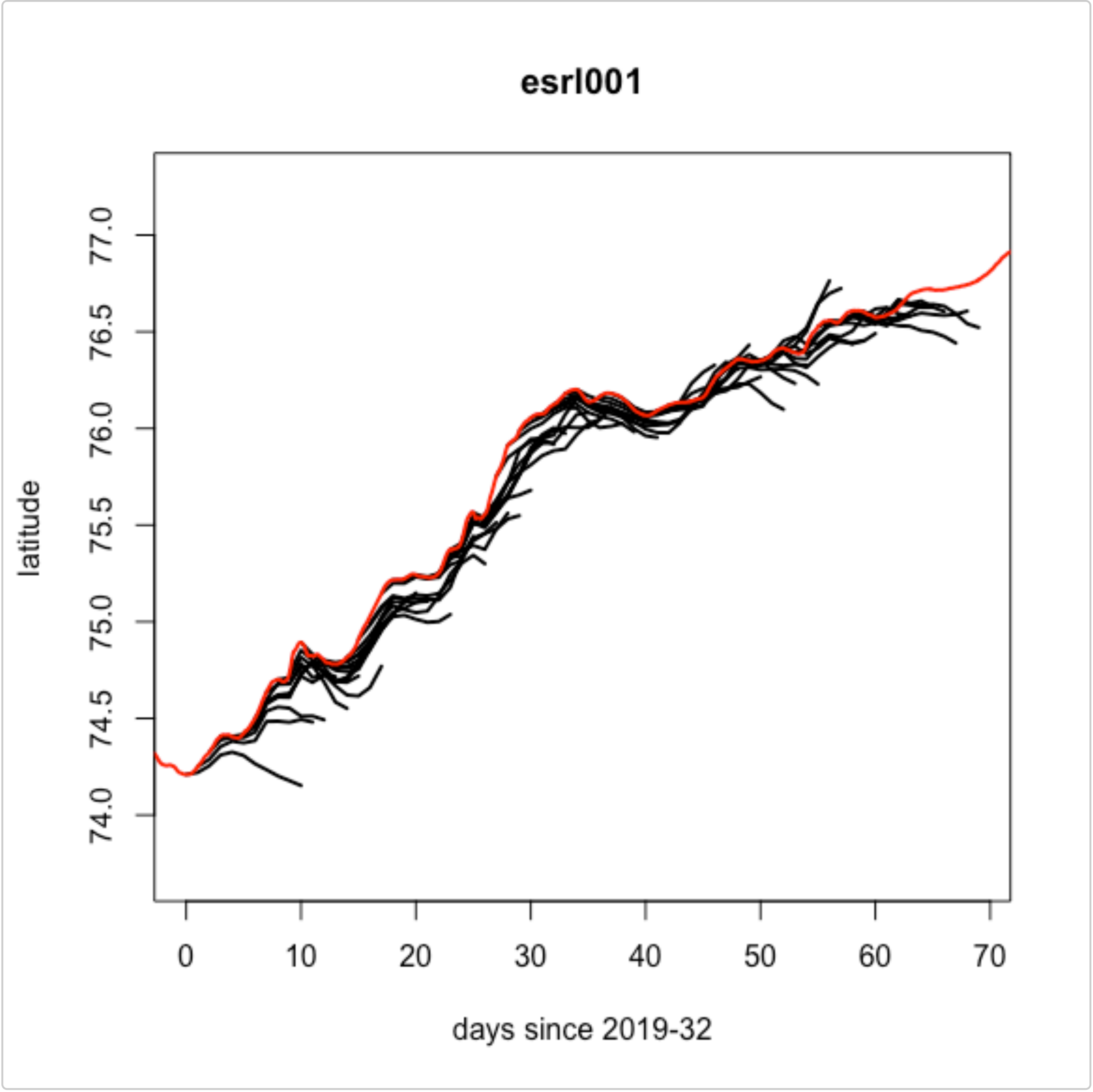
```
RefYear = fcst$res.list[[1]]$InitYear
RefDayOfYear = fcst$res.list[[1]]$InitDayOfYear
nFcst = length(fcst$res.list)
last.time = sidfex.ydoy2reltime(Year = fcst$res.list[[nFcst]]$LastYear, DayOfYear =
fcst$res.list[[nFcst]]$LastDayOfYear,
    RefYear = RefYear, RefDayOfYear = RefDayOfYear)
obs.time = sidfex.ydoy2reltime(Year = obs$data$Year, DayOfYear = obs$data$POS_DOY,
    RefYear = RefYear, RefDayOfYear = RefDayOfYear)

xlim = c(0, last.time)
ylim = extendrange(obs$data$Lat[obs.time >= 0 & obs.time <= last.time], f = 0.2)
plot(NA, xlim = xlim, ylim = ylim, xlab = paste0("days since ", RefYear, "-",
    RefDayOfYear), ylab = "latitude", main = "ecmwf001")
for (i in 1:nFcst) {
    fcst.time = sidfex.ydoy2reltime(Year = fcst$res.list[[i]]$data$Year, DayOfYear
= fcst$res.list[[i]]$data$DayOfYear,
        RefYear = RefYear, RefDayOfYear = RefDayOfYear)
    LatColumns = which(substr(names(fcst$res.list[[i]]$data), start = 1, stop = 3)
==
        "Lat")
    if (length(LatColumns) > 1) {
        for (latcol in LatColumns[2:length(LatColumns)]) {
            # plot the individual ensemble members
            lines(fcst.time, fcst$res.list[[i]]$data[, latcol], col = i + 2)
        }
    }
    # plot the ensemble mean
    lines(fcst.time, fcst$res.list[[i]]$data$Lat, col = "black", lwd = 2)
}
# plot the observations
lines(obs.time, obs$data$Lat, col = "red", lwd = 2)
```



How do the forecasts from the system `esrl001_SeaIceVelocity` compare? If we rerun the same code for another subindex that differs from `subindex3` only in terms of the `GroupID` (both groups submit only one `MethodID`) used in the call to `sidfex.fcst.search.extractFromTable()` (keeping the reference time fixed), we obtain the following plot:

```
subindex4 = sidfex.fcst.search.extractFromTable(gid = "esrl001", tid =
"300234063991680",
    iy = 2019, idoy = c(32, 91))
fcst = sidfex.read.fcst(files = subindex4, checkfileformat = FALSE)
nFcst = length(fcst$res.list)
last.time = sidfex.ydoy2reltime(Year = fcst$res.list[[nFcst]]$LastYear, DayOfYear =
fcst$res.list[[nFcst]]$LastDayOfYear,
    RefYear = RefYear, RefDayOfYear = RefDayOfYear)
```

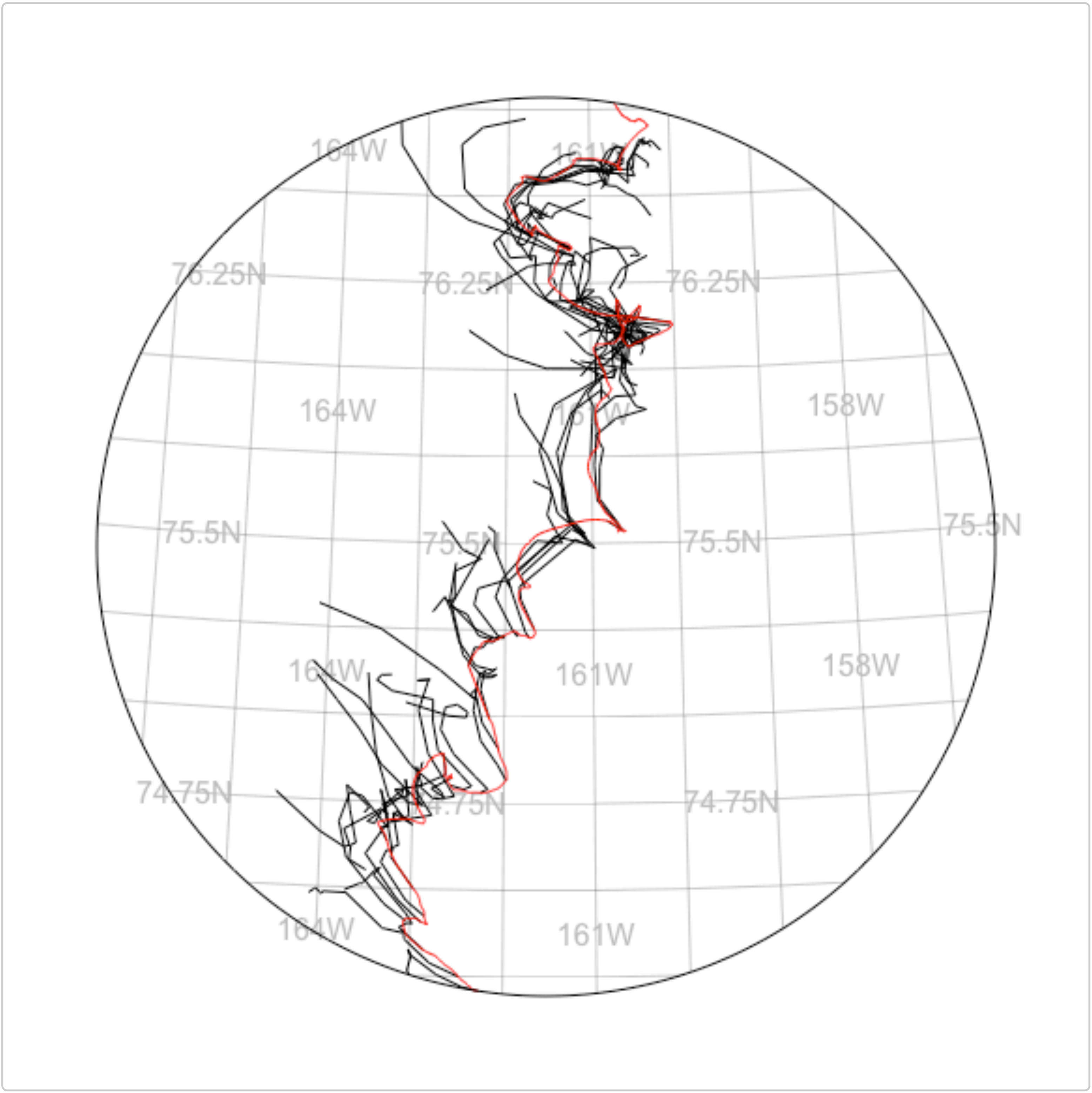
```r
xlim = c(0, last.time)
ylim = extendrange(obs$data$Lat[obs.time >= 0 & obs.time <= last.time], f = 0.2)
plot(NA, xlim = xlim, ylim = ylim, xlab = paste0("days since ", RefYear, "-",
    RefDayOfYear), ylab = "latitude", main = "esrl001")
for (i in 1:nFcst) {
    fcst.time = sidfex.ydoy2reltime(Year = fcst$res.list[[i]]$data$Year, DayOfYear
= fcst$res.list[[i]]$data$DayOfYear,
        RefYear = RefYear, RefDayOfYear = RefDayOfYear)
    LatColumns = which(substr(names(fcst$res.list[[i]]$data), start = 1, stop = 3)
==
        "Lat")
    if (length(LatColumns) > 1) {
        # WILL NOT BE TRUE IN THIS CASE (ONLY SINGLE FORECASTS)
        for (latcol in LatColumns[2:length(LatColumns)]) {
            # plot the individual ensemble members
            lines(fcst.time, fcst$res.list[[i]]$data[, latcol], col = i + 2)
        }
    }
    # plot the ensemble mean
    lines(fcst.time, fcst$res.list[[i]]$data$Lat, col = "black", lwd = 2)
}
# plot the observations
lines(obs.time, obs$data$Lat, col = "red", lwd = 2)
```



It is obvious that the `esrl001` forecasts are shorter (10-days range), but initialised more frequently (daily).

Obviously we can do the same but plot longitudes instead of latitudes. Instead, we now plot the `esrl001` data onto a map using `spheRlab`:

```
which.obs = which(obs.time >= 0 & obs.time <= last.time)
plot.domain = sl.boundingcircle(lon = obs$data$Lon[which.obs], lat =
obs$data$Lat[which.obs],
    verbose = FALSE)
pir = sl.plot.init(projection = "polar", polar.lonlatrot = c(plot.domain$center_lon,
    plot.domain$center_lat, 0), polar.latbound = 90 - plot.domain$radius,
do.init.device = FALSE)
# use the following line to plot coastlines (if within the plot domain).
# Note that this would initiate a download of the corresponding data from
# naturalearthdata.com when called for the first time.
# sl.plot.naturalearth(pir,what='coastline',resolution='medium')
sl.plot.lonlatgrid(pir, labels = TRUE)
# now plot the forecast trajectories
for (i in 1:nFcst) {
    LatColumns = which(substr(names(fcst$res.list[[i]]$data), start = 1, stop = 3)
==
        "Lat")
    if (length(LatColumns) > 1) {
        # WILL NOT BE TRUE IN THIS CASE (ONLY SINGLE FORECASTS)
        for (latcol in LatColumns[2:length(LatColumns)]) {
            # plot the individual ensemble members
            sl.plot.lines(pir, lon = fcst$res.list[[i]]$data[, latcol + 1],
                lat = fcst$res.list[[i]]$data[, latcol], col = i + 2)
        }
    }
    # plot the ensemble mean
    sl.plot.lines(pir, lon = fcst$res.list[[i]]$data$Lon, lat =
fcst$res.list[[i]]$data$Lat,
        col = "black")
}
# now the observed trajectory (only the part relevant for the forecasts)
sl.plot.lines(pir, lon = obs$data$Lon[which.obs], lat = obs$data$Lat[which.obs],
    col = "red")
sl.plot.end(pir, do.close.device = FALSE)
```



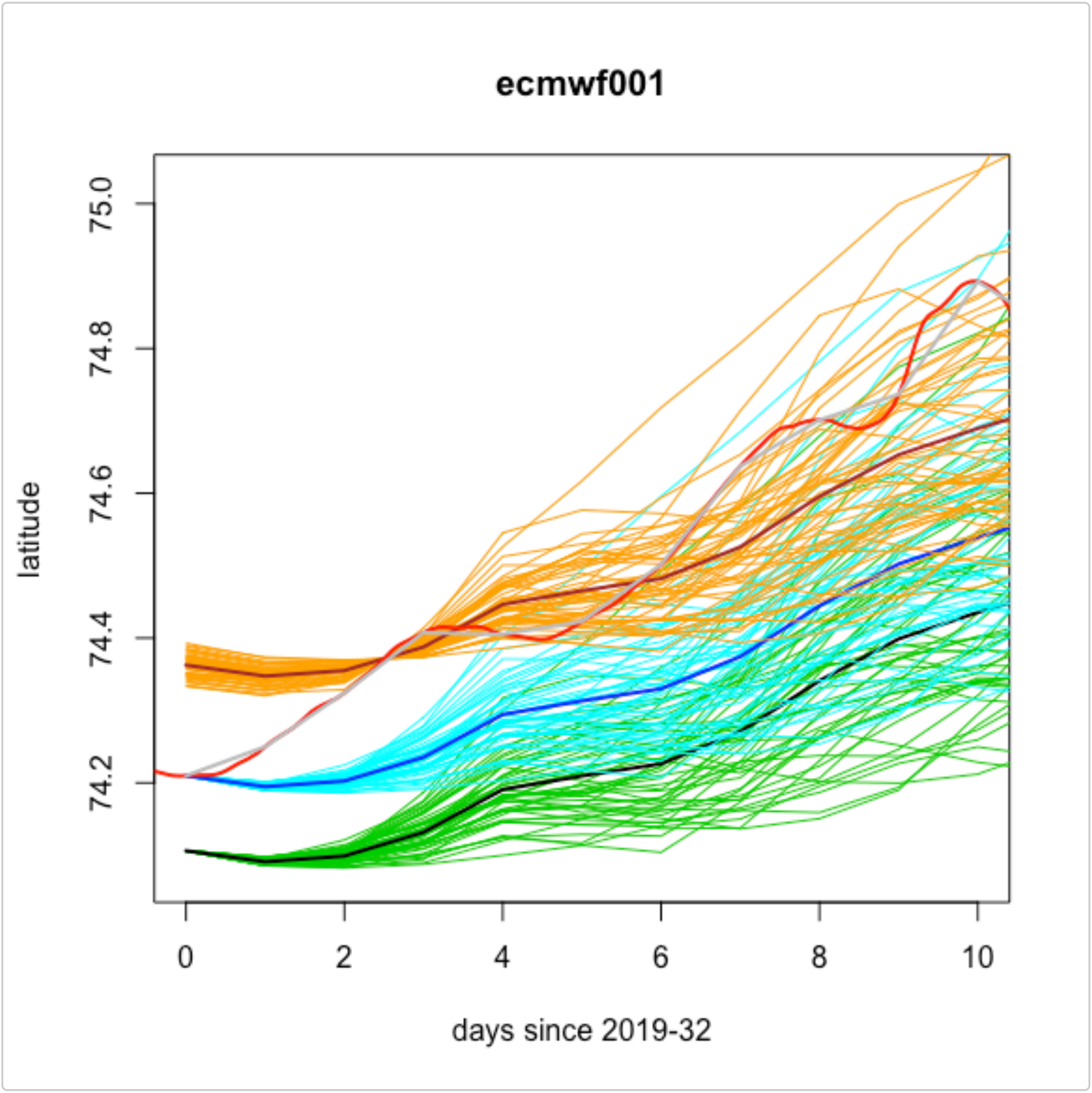
Remapping and adjusting data in time and space

Remapping and adjusting SIDFEx data in time and space can be useful for example to (i) align observational data with the forecasts so that errors can be quantified, (ii) adjust initial locations of forecasts if those were inaccurate, (iii) harmonise time axes for multi-model or single-model ensembles, or (iv) adjust forecasts in real-time applications when more recent observations are available. In fact, the harmonisation mentioned in point (iii) is applied automatically in `sidfex.read.fcst()` when ensembles are merged that contain ensemble members with inhomogenous time axes (mostly because they are lagged-initial-date ensembles). Remapping and adjusting the data needs to account correctly for the geometry on the sphere, in particular when trajectories are close to the geographic (lon-lat) pole and/or the date line. For this purpose `spheRlab` is used, in particular the functions `sl.trajectory.remaptime()` and `sl.rot()`. In the following we consider points (ii), (iv), and (i) of the above list, in that order.

Initial locations of SIDFEx forecasts can be slightly inaccurate, for example if Lagrangian tracers are initialised at nearest-neighbour grid-points instead of the exact locations. This is the case for example for the `ecmwf001` forecasts plotted above. In the following we zoom into the first 10 days of the first forecast ensemble shown above (latitude versus time). We first plot the data as is (colours as above) and then correct the initial location using `sidfex.rot.fcst()` and re-plot the forecast ensemble in blue. We then assume that the most recent position has been observed at YEAR=2019, DOY=34.5 (2.5 days after the initial time) and again adjust the forecast ensemble so that the trajectories (in brown/orange) pass the observed position at that time. Finally, we remap the observations to the time axis of the forecast ensemble (in grey). The code is somewhat simpler than above because the reduced subindex holds only one forecast ensemble:

```r
subindex3.1 = sidfex.fcst.search.extractFromTable(gid = "ecmwf001", tid =
"300234063991680",
    iy = 2019, idoy = c(32))
fcst = sidfex.read.fcst(files = subindex3.1, checkfileformat = FALSE)
obs.time = sidfex.ydoy2reltime(Year = obs$data$Year, DayOfYear = obs$data$POS_DOY,
    RefYear = RefYear, RefDayOfYear = RefDayOfYear)
xlim = c(0, 10)
ylim = extendrange(obs$data$Lat[obs.time >= 0 & obs.time <= 10], f = 0.2)
plot(NA, xlim = xlim, ylim = ylim, xlab = paste0("days since ", RefYear, "-",
    RefDayOfYear), ylab = "latitude", main = "ecmwf001")
fcst.time = sidfex.ydoy2reltime(Year = fcst$res.list[[1]]$data$Year, DayOfYear =
fcst$res.list[[1]]$data$DayOfYear,
    RefYear = RefYear, RefDayOfYear = RefDayOfYear)
LatColumns = which(substr(names(fcst$res.list[[1]]$data), start = 1, stop = 3) ==
    "Lat")
for (latcol in LatColumns[2:length(LatColumns)]) {
    lines(fcst.time, fcst$res.list[[1]]$data[, latcol], col = 3)
}
lines(fcst.time, fcst$res.list[[1]]$data$Lat, col = "black", lwd = 2)

# now with adjustment of the initial location (default of sidfex.rot.fcst())
fcst.adj.init = sidfex.rot.fcst(obs = obs, fcst = fcst)
for (latcol in LatColumns[2:length(LatColumns)]) {
    lines(fcst.time, fcst.adj.init$res.list[[1]]$data[, latcol], col = "cyan")
}
lines(fcst.time, fcst.adj.init$res.list[[1]]$data$Lat, col = "blue", lwd = 2)

# now with adjustment of the location at DOY=34.5
```

```
fcst.adj.34.5 = sidfex.rot.fcst(obs = obs, fcst = fcst, obsref.Year = 2019,
    obsref.DayOfYear = 34.5)
for (latcol in LatColumns[2:length(LatColumns)]) {
    lines(fcst.time, fcst.adj.34.5$res.list[[1]]$data[, latcol], col = "orange")
}
lines(fcst.time, fcst.adj.34.5$res.list[[1]]$data$Lat, col = "brown", lwd = 2)

# plot the observations
lines(obs.time, obs$data$Lat, col = "red", lwd = 2)

# now remap observations to fcst time axis and re-plot
obs.remap = sidfex.remaptime.obs2fcst(obs = obs, fcst = fcst)
obs.remap.time = sidfex.ydoy2reltime(Year = obs.remap$res.list[[1]]$data$Year,
    DayOfYear = obs.remap$res.list[[1]]$data$DayOfYear, RefYear = RefYear,
RefDayOfYear = RefDayOfYear)
lines(obs.remap.time, obs.remap$res.list[[1]]$data$Lat, col = "grey", lwd = 2)
```



The return values of `sidfex.rot.fcst()` and `sidfex.remaptime.obs2fcst()` are structured like the forecast data that have been provided as input. The latter function produces one remapped observed trajectory for every forecast contained in the `res.list` of the forecast object provided as input. This is why the arguments of the two last calls to `lines()` in the above code are structurally different (`obs$data...` versus `obs.remap$res.list[[1]]$data...`).

Quantative forecast evaluation

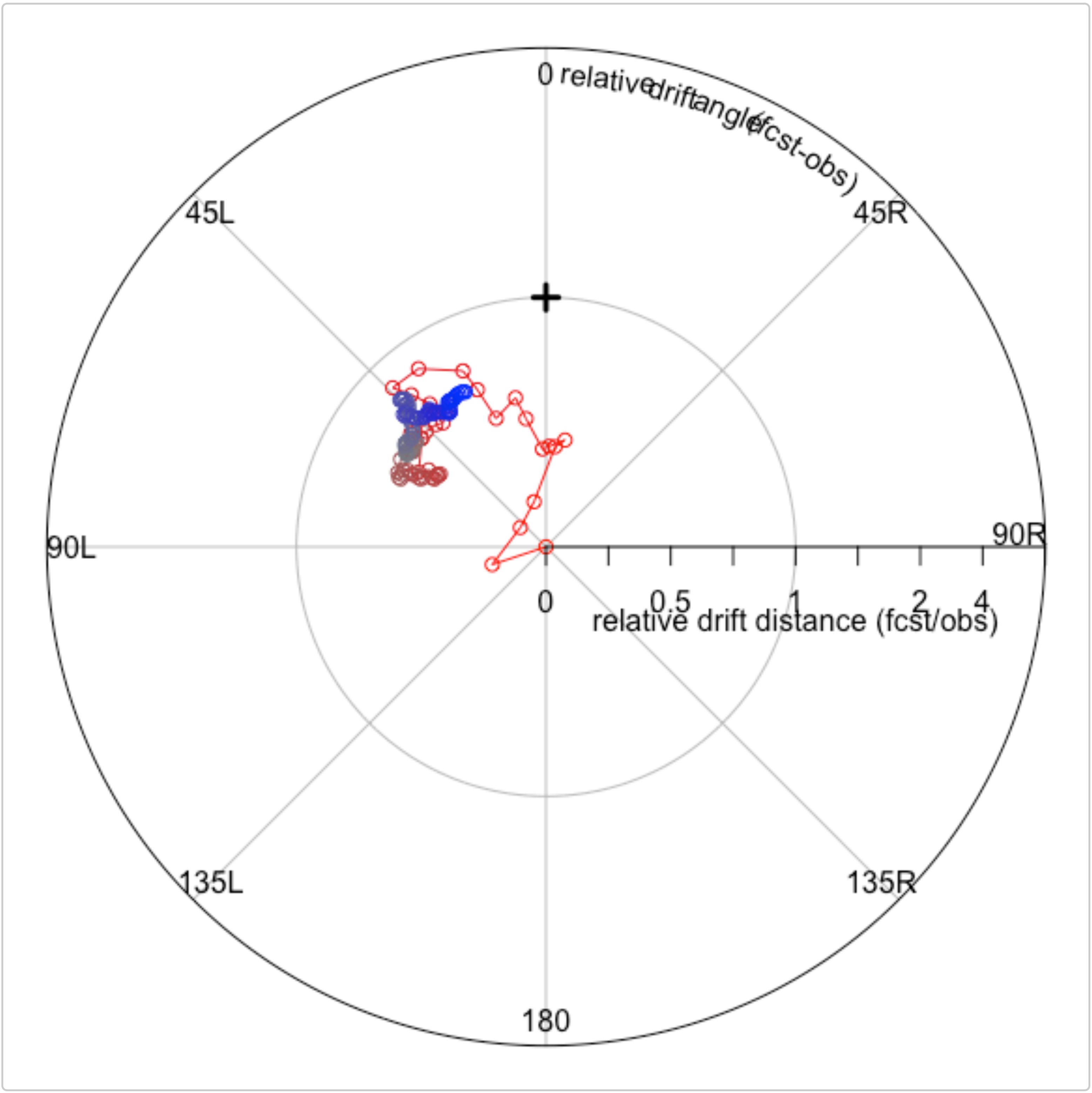
With `sidfex.remaptime.obs2fcst()` it is in principle straight-forward to compute forecast errors for concurrent values of the forecasts and the observations, that is, to *evaluate* or *verify* the forecasts. To facilitate the computation of errors (and ensemble spread), the function `sidfex.evaluate()` can be used. For example, an evaluation for one of the previous examples can be obtained as follows:

```
errs = sidfex.evaluate(fcst = fcst.adj.init)
str(errs$res.list[[1]])
#> List of 21
#>  $ ens.mean.gc.dist              : num [1:125] 0 8437 20114 26834 17068 ...
#>  $ ens.mean.lat.err              : num [1:125] 2.84e-12 -5.51e-02 -1.22e-01 -1.72e-
01 -1.11e-01 ...
#>  $ ens.mean.lon.err              : num [1:125] -1.71e-12 1.92e-01 4.93e-01 6.26e-01
3.94e-01 ...
#>  $ ens.individual.gc.dist        : num [1:125, 1:51] 0 8350 19957 27812 18257 ...
#>  $ ens.individual.lat.err        : num [1:125, 1:51] 2.84e-12 -5.41e-02 -1.18e-01
-1.77e-01 -1.20e-01 ...
#>  $ ens.individual.lon.err        : num [1:125, 1:51] -1.71e-12 1.92e-01 4.98e-01
6.54e-01 4.17e-01 ...
#>  $ ens.individual.gc.dist.mean   : num [1:125] 0 8439 20123 26859 17459 ...
#>  $ ens.individual.lat.err.mean   : num [1:125] 2.86e-12 -5.51e-02 -1.22e-01 -1.72e-
01 -1.11e-01 ...
#>  $ ens.individual.lat.err.meanabs: num [1:125] 2.86e-12 5.51e-02 1.22e-01 1.72e-01
1.11e-01 ...
#>  $ ens.individual.lon.err.mean   : num [1:125] -1.70e-12 1.92e-01 4.93e-01 6.26e-01
3.94e-01 ...
#>  $ ens.individual.lon.err.meanabs: num [1:125] 1.70e-12 1.92e-01 4.93e-01 6.26e-01
3.94e-01 ...
#>  $ ens.spread.gc.dist            : num [1:125] 0 436 955 2584 4391 ...
#>  $ ens.spread.lat                : num [1:125] 1.67e-14 -1.72e-07 -5.57e-07 -4.78e-
06 -1.14e-05 ...
#>  $ ens.spread.lon                : num [1:125] 8.36e-15 -1.72e-06 -4.08e-06 -5.53e-
05 -5.48e-05 ...
#>  $ ens.mean.relspeed             : num [1:125] 0 0.227 0.129 0.187 0.403 ...
#>  $ ens.mean.angle                : num [1:125] -175.5 108.26 52.99 14.77 -5.16 ...
#>  $ ens.individual.relspeed       : num [1:125, 1:51] 0 0.213 0.118 0.157 0.361 ...
#>  $ ens.individual.angle          : num [1:125, 1:51] -175.5 106.81 45.32 15.81
-5.17 ...
#>  $ ens.individual.relspeed.mean  : num [1:125] 0 0.232 0.136 0.196 0.416 ...
#>  $ ens.individual.angle.mean     : num [1:125] -175.6 106.12 53.26 19.84 -1.12 ...
#>  $ ens.individual.angle.meanabs  : num [1:125] 175.6 106.1 53.3 20.6 12.2 ...
```

Those quantities starting `ens.mean` relate to the ensemble-mean position or the single deterministic position of a non-ensemble forecast. Those starting `ens.individual` (provided only for ensembles) are either valid for individual ensemble members or, if they end `.mean` or `.meanabs`, they are mean (absolute) values across the corresponding statistic for individual ensemble members. Quantities starting `ens.spread` provide the ensemble spread (mean deviations of ensemble members from the ensemble-mean location). To see how these different diagnostics are defined exactly, please consult the function documentation by typing `?sidfex.evaluate`.

A rather specific function of this package can be used to produce formatted plots of the relative drift speed and angle. (The function works currently only with ensemble-mean data, or with non-merged ensembles; aspect ratios are more optimised with `device="pdf"`, or can be adjusted with other arguments, see `?sidfex.plot.speedangle`.)

```
colbar = sidfex.plot.speedangle(read.fcst.res = fcst.adj.init, points.type = "b",
    col.by = "DaysLeadTime", device = NULL)
#> [1] "values for 'DaysLeadTime' to determine the colourbar are in the following
range:"
#>    0%  25%  50%  75% 100%
#>     0   31   62   93  124
```



The quantity and unit defining the colours is *lead time (days)*, in this case from red=0 to blue=124; to plot a corresponding colourbar, type `sl.plot.colbar(colbar,do.init.device=FALSE)`. The cross in the upper middle marks the perfect-forecast point. The plot reveals for this particular case that during the first couple of days (red colours) the ensemble-mean forecast drift is considerably slower than the observed drift, but in a realistic direction (at least after the very first days). At longer lead times, the ensemble-mean drift is still slightly slower (approx. 75% relative drift speed) and about 45° to the left of the observed trajectory. Note that these relative speed and angle errors, like those generated with `sidfex.evaluate()`, consider the *integrated* drift since the initial time and location, not the *current* drift.

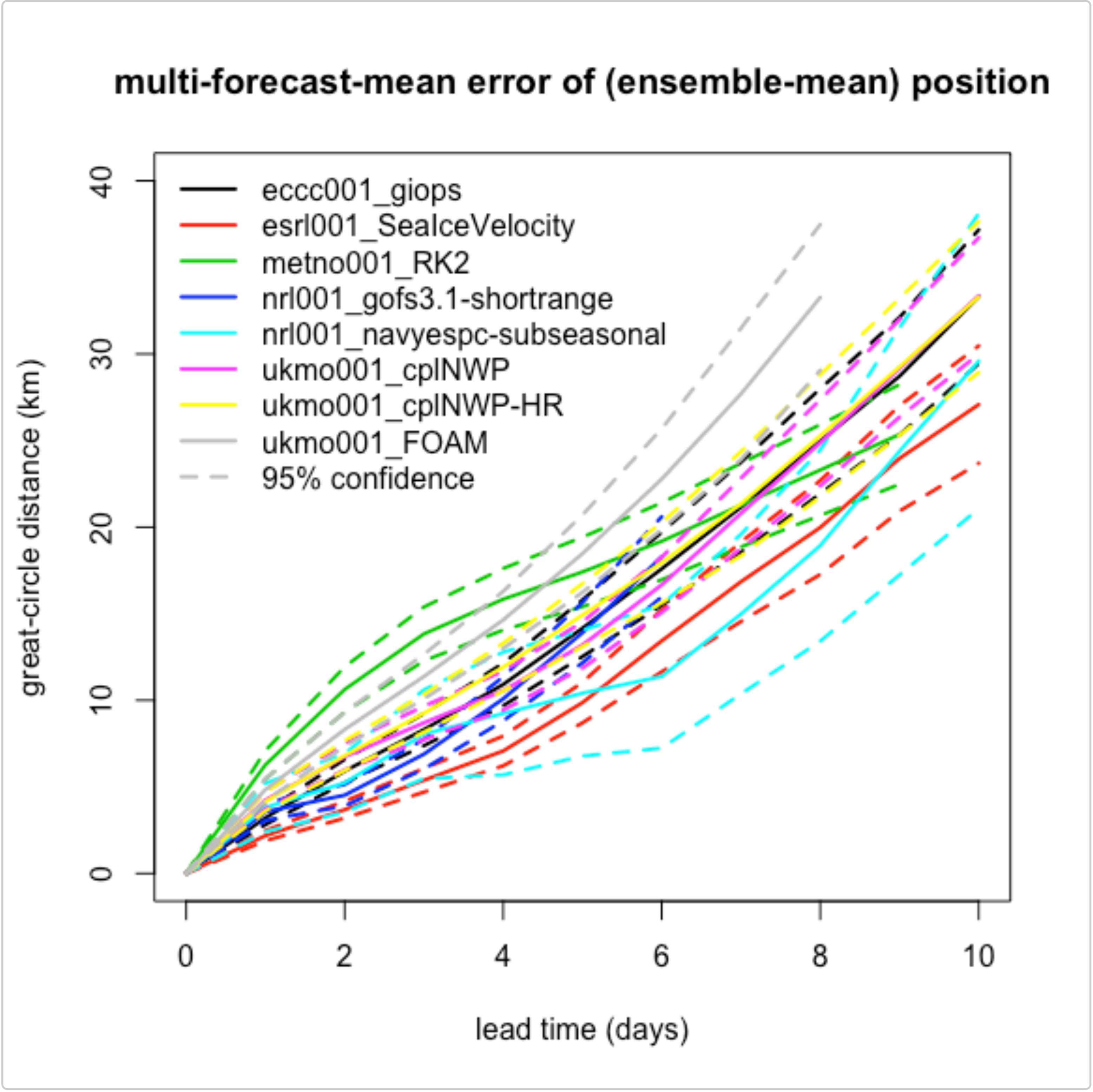
The following is an example that comes closer to a useful quantitative (and comparative) analysis than the mostly single-case-based examples considered above. We consider all forecasts for `TargetID="300234061872720"` initialised between December 1st 2018 and March 15th 2019 from all forecast methods that have provided at least weakly forecasts, tolerating gaps (missing forecast for certain initial times) where they exist. We then compare some forecast error statistics for the first ten days between the forecast methods, including an assessment of statistical confidence by adding two-standard-errors-margins to the plots. On the way, we recycle different code chunks from above to inspect the data.

```
subind.allmethods = sidfex.fcst.search.extractFromTable(tid = "300234061872720",
    iy = c(2018, 2019), idoy = c(335, 74), InheritFromParent = TRUE)
# setting 'InheritFromParent=TRUE' makes sure that lagged-initial-date
# ensembles are not split
systms = paste(subind.allmethods$GroupID, subind.allmethods$MethodID, sep = "_")
systms_unique = unique(systms)
tbl = as.data.frame(matrix(nrow = length(systms_unique), ncol = 4))
rownames(tbl) = systms_unique
colnames(tbl) = c("nFcst", "leadtimerange", "enssize", "nTimeSteps")
for (i in 1:length(systms_unique)) {
    indices = (systms == systms_unique[i])
    tbl[i, 1] = sum(indices)
    tbl[i, 2] = mean(subind.allmethods$FcstTime[indices])
    tbl[i, 3] = mean(subind.allmethods$EnsSize[indices])
    tbl[i, 4] = mean(subind.allmethods$nTimeSteps[indices])
}
print(tbl)
#>                              nFcst leadtimerange enssize nTimeSteps
#> awi001_ClimRunVers2019May       30      363.6667      10        365
#> eccc001_giops                  100       10.0000       1        481
#> ecmwf001_SEAS5                 153      124.0000      51        125
#> esrl001_SeaIceVelocity         101       10.0000       1         11
#> metno001_RK2                   105        9.0000       1          9
#> nrl001_gofs3.1-shortrange      104        6.5000       1          7
#> nrl001_navyespc-subseasonal     60       43.5000       4         44
#> ukmo001_cplNWP                 101       11.0000       1         12
#> ukmo001_cplNWP-HR              101       11.0000       1         12
#> ukmo001_FOAM                   102        8.0000       1          9
```

Note that in case of ensemble forecasts `nFcst` here needs to be divided by the ensemble size to get the actual number of forecast *ensembles*. In the following we keep only those methods with more than 10 forecasts (forecast ensembles) available for the considered 3.5-months time period, allowing some meaningful statistics. This criterium is met by all deterministic forecast methods (`enssize=1`) and one of the ensemble forecast methods (`nrl001_navyespc-subseasonal`) because it provides weekly ensembles. Since we want to derive forecast error statistics across different initial times for each of the selected forecast methods separately, and only then compare the results between the systems, we execute a loop where we define different subindices for each forecast method and apply the same analysis chain to all of them separately. We store only the respective final output of `sidfex.evaluate()` in a list with one element for each forecast method:

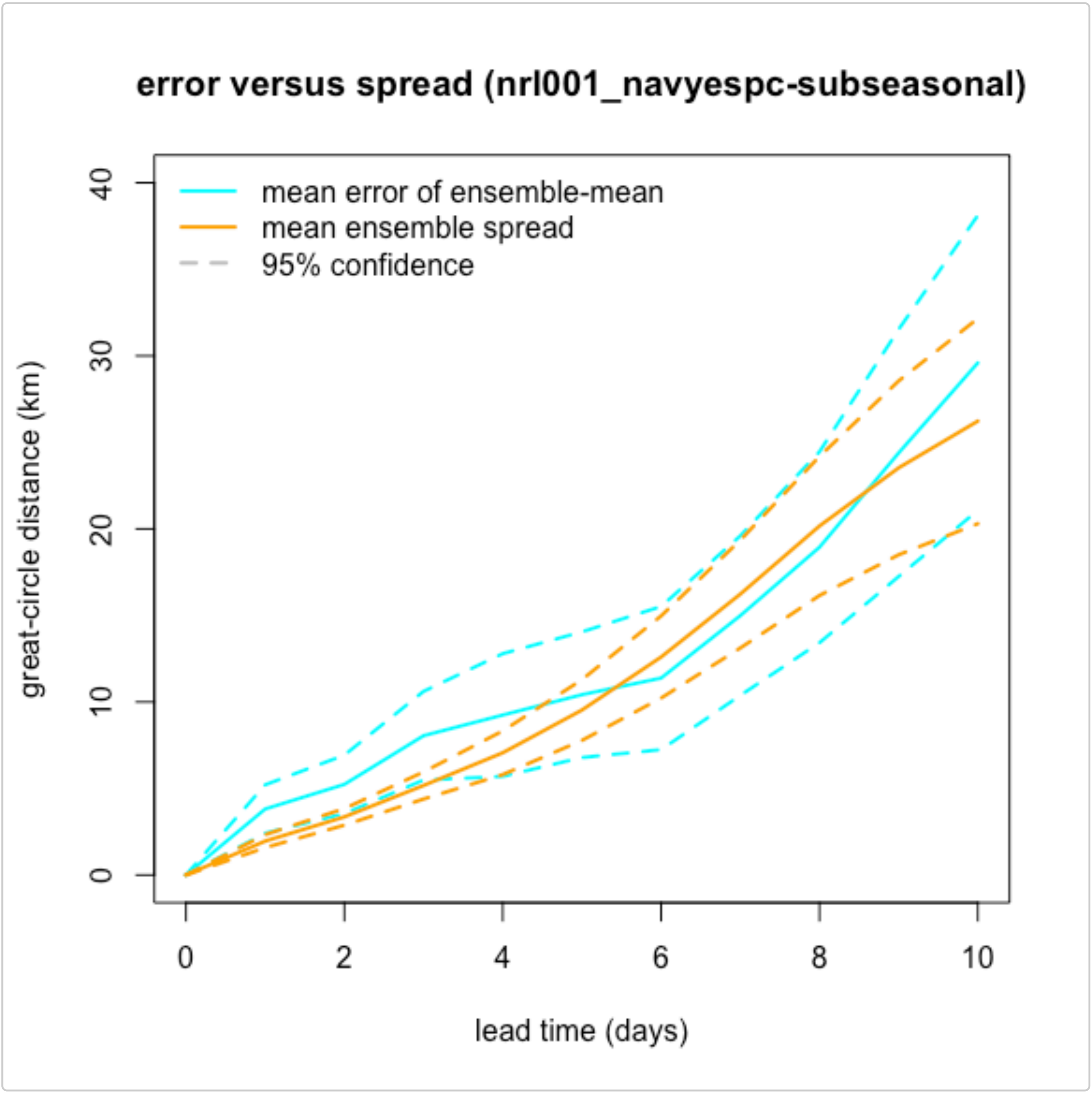
```
systms_selected = systms_unique[(tbl$nFcst/tbl$enssize > 10)]
```

```r
Nmethods = length(systms_selected)
remap.time = seq(0, 10)
eval.results = list()
for (i in 1:Nmethods) {
    # this time we subselect the index manually instead of using
    # sidfex.fcst.search.extractFromTable()
    subind = subind.allmethods[systms %in% systms_selected[i], ]
    fcst.orig = sidfex.read.fcst(files = subind, checkfileformat = FALSE, verbose =
FALSE)
    # remap everything to identical relative time axes to enable compution of
    # statistics across forecasts and clean comparison between methods
    fcst.remap = sidfex.remaptime.fcst(fcst = fcst.orig, newtime.DaysLeadTime =
remap.time,
        verbose = FALSE)
    # make sure that initial offsets are corrected for (where they exist)
    fcst.adjust = sidfex.rot.fcst(fcst = fcst.remap)
    # keep only the multi-forecast statistics from the output of
    # sidfex.evaluate()
    eval.results[[i]] = sidfex.evaluate(fcst = fcst.adjust, verbose =
FALSE)$multifcst.stats
}
#> Warning in sidfex.read.fcst(files = subind, checkfileformat = FALSE,
#> verbose = FALSE): Some forecast ensemble members have been remapped
#> temporally to match the parent forecast time axis
# now plot mean great-circle distances of (ensemble-mean) positions for each
# method
plot(NA, xlim = range(remap.time), ylim = c(0, 40), main = "multi-forecast-mean
error of (ensemble-mean) position",
    xlab = "lead time (days)", ylab = "great-circle distance (km)")
for (i in 1:Nmethods) {
    dat = eval.results[[i]]$ens.mean.gc.dist/1000
    # 95% confidence bands of the mean error based on standard-error*2
    lines(remap.time, dat$mean + 2 * dat$st.err, col = i, lty = 2, lwd = 2)
    lines(remap.time, pmin(dat$mean - 2 * dat$st.err), col = i, lty = 2, lwd = 2)
    # now the mean error
    lines(remap.time, dat$mean, col = i, lwd = 2)
}
legend("topleft", legend = c(systms_selected, "95% confidence"), lty = c(rep(1,
    Nmethods), 2), bty = "n", col = c(1:Nmethods, 8), lwd = 2)
```



Finally, we inspect the spread-error relationship for the only ensemble-based forecast method from the previous example, namely `nrl001_navyespc-subseasonal`:

```
plot(NA, xlim = range(remap.time), ylim = c(0, 40), main = "error versus spread
(nrl001_navyespc-subseasonal)",
    xlab = "lead time (days)", ylab = "great-circle distance (km)")
i = which(systms_selected == "nrl001_navyespc-subseasonal")
dat = eval.results[[i]]$ens.mean.gc.dist/1000
lines(remap.time, dat$mean + 2 * dat$st.err, col = "cyan", lty = 2, lwd = 2)
lines(remap.time, pmin(dat$mean - 2 * dat$st.err), col = "cyan", lty = 2, lwd = 2)
lines(remap.time, dat$mean, col = "cyan", lwd = 2)
# and now the same for the ensemble spread, in orange
dat = eval.results[[i]]$ens.spread.gc.dist/1000
lines(remap.time, dat$mean + 2 * dat$st.err, col = "orange", lty = 2, lwd = 2)
lines(remap.time, pmin(dat$mean - 2 * dat$st.err), col = "orange", lty = 2,
    lwd = 2)
lines(remap.time, dat$mean, col = "orange", lwd = 2)
legend("topleft", legend = c("mean error of ensemble-mean", "mean ensemble spread",
    "95% confidence"), col = c("cyan", "orange", "grey"), bty = "n", lty = c(1,
    1, 2), lwd = 2)
```



In this case the mean spread is somewhat smaller than the mean error in the beginning, which generally may point to under-dispersive (that is, over-confident) ensembles. However, here (i) the 95% confidence bands overlap, meaning that the difference is statistically not significant, and (ii) a required correction factor has not been applied to the estimate of the ensemble spread (which is relative to the ensemble-mean position and thus low-biased).

Extracting real-time forecasts

An example how to extract real-time forecasts will be added soon.

What else?

There are more functions in the `SIDFEx` R-package than have been used in this user guide. To get an overview, type `help(package="SIDFEx")`. Most functions have an individual function documentation (accessible by typing `?` followed by the function name, without white space) that explains their purpose and usage. Note however that some of the functions are very specific and rather for internal usage, e.g., as part of the processing chain installed at the DKRZ.

An R-Shiny App that builds on the `SIDFEx` R-package is planned to be released in July 2019. The App will allow anyone to use some of the essential functionality of this package in a browser-based online tool (without programming). The App will be hosted on a server of the Alfred Wegener Institute (Bremerhaven, Germany).

Please provide any feedback on `SIDFEx`, the `SIDFEx` R-package, and/or this user guide to helge.goessling@awi.de.

Acknowledgement

We are indepted to all groups that have contributed and/or continue contributing forecasts to `SIDFEx`, as well as to many others that have supported `SIDFEx` in one or the other way. We thank the German Climate Computing Centre (DKRZ) for hosting the `SIDFEx` data, including the real-time processing chain, and colleagues at the International Arctic Buoy Program (IABP) for their support. Moreover, we acknowledge the generous funding of the German Ministry for Education and Research (BMBF) for the Young Investigator Group Seamless Sea Ice Prediction (SSIP, grant 01LN1701A), which has enabled us to develop and maintain `SIDFEx`, including this R-package.