The blsplotR package

Brian C. Monsell

6/8/23

Contents

1 Introduction

The `blsplotR` package generates several types of time series plots useful for seasonal adjustment analysis. These routines rely heavily on the seasonal package to extract series and components from the seasonal adjustments generated by the US Census Bureau's X-13ARIMA-SEATS software, and can be generated from a single seas object or a list of seas objects.

Types of plots include line plots, ratio plots, forecast plots, forecast error diagnostic plots, spectral plots, seasonal factor plots, seasonal adjustment component plots.

Users can add grid lines and shade recession regions in selected plots, and in many of the functions there are options to control the margins of plots, suppress the main title, and control the size of titles, axis labels, and subtitles.

1.1 The `seasonal` R package

For most of these plots, output from the `seasonal` package is used to generate the data necessary to in turn generate the plots. This package allows users to run the X-13ARIMA-SEATS program within R and stores the output and diagnostics within efficient data objects within R. There are also utilities to read external data files and spec files used by the X-13ARIMA-SEATS program.

You can download the `seasonal` package from CRAN: you can access the package at https://cran.r-project.org/web/packages/seasonal/index.html.

We'll show examples of using the seasonal package in the examples shown in this document. There are several references online for learning to use R:

- An excellent summary for the seasonal package can be found in the paper "Seasonal Adjustment by X-13ARIMA-SEATS in R" by Christophe Sax and Dirk Eddelbuettel; access this paper at https://cran.r-project.org/web/packages/seasonal/vignettes/seas.pdf.
- A reference document for the individual functions of the `seasonal` package can be accessed at https://cran.r-project.org/web/packages/seasonal/seasonal.pdf.
- Special vignette on more recent developments for using `seasonal` with multiple series can be accessed at https://cran.r-project.org/web/packages/seasonal/vignettes/multiple.html.

These functions were developed using version 1.0.9 of the `seasonal` package.

2 Individual plots

These functions generate different kinds of plots that are useful for seasonal adjustment analysts. Many of them use the seas object generated from the seasonal package.

We'll give a brief description of each function and show an example of how to use the function to generate a plot. For most of the functions, we'll use the Airline Passenger series from Box and Jenkins, since it is part of the basic R distribution.

2.1 plot_cpgram_resid

The `plot_cpgram_resid` function generates a plot of the cumulative periodogram of the regARIMA residuals. This plot tests whether the residuals from the regARIMA model specified by the user are white noise. There are bounds around the cumulative periodogram, and a subtitle shows values for the Ljung-Box Q statistics for the first and second seasonal lag.

```
air_seas <-
  seas(AirPassengers, transform.function= 'log',
       arima.model = '(0 1 1)(0 1 1)')
blsplotR::plot_cpgram_resid(air_seas,
  main_title = 'Cumulative periodogram for Airline Passenger Residuals')
```

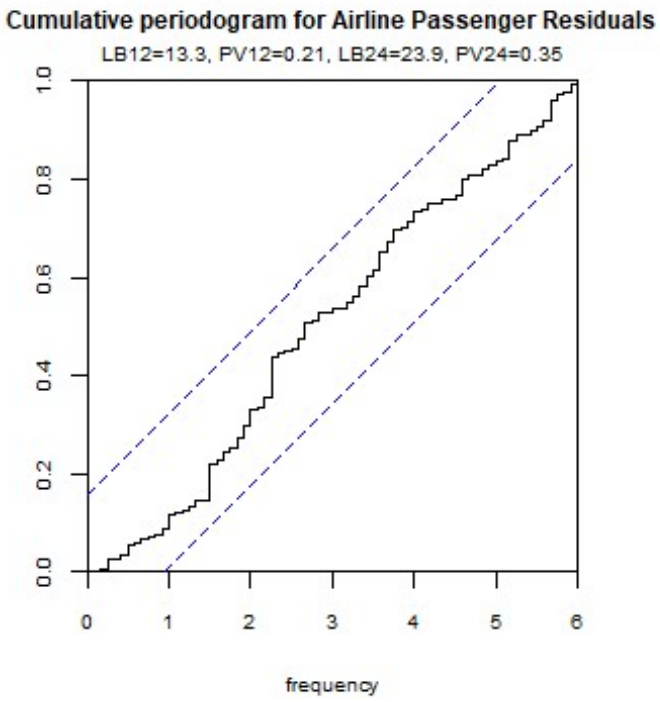

Figure 1: Example of `plot_cpgram_resid`

This plot (Figure 1) shows the residuals of fitting a (0 1 1)(0 1 1) ARIMA model to the Airline Passenger series can be considered white noise.

2.2 plot_double_spectrum

The `plot_double_spectrum` function generates a spectrum of the original series and the seasonally adjusted series on same axis. This allows the user to examine how seasonal the original series is and access if there is residual seasonality in the seasonally adjusted series. The trading day frequencies are flagged as well.

```
blsplotR::plot_double_spectrum(air_seas,
  series_name = 'AirPassengers',
  this_col = c('blue', 'darkgreen', 'darkblue', 'forestgreen',
               'red', 'orange'))
```

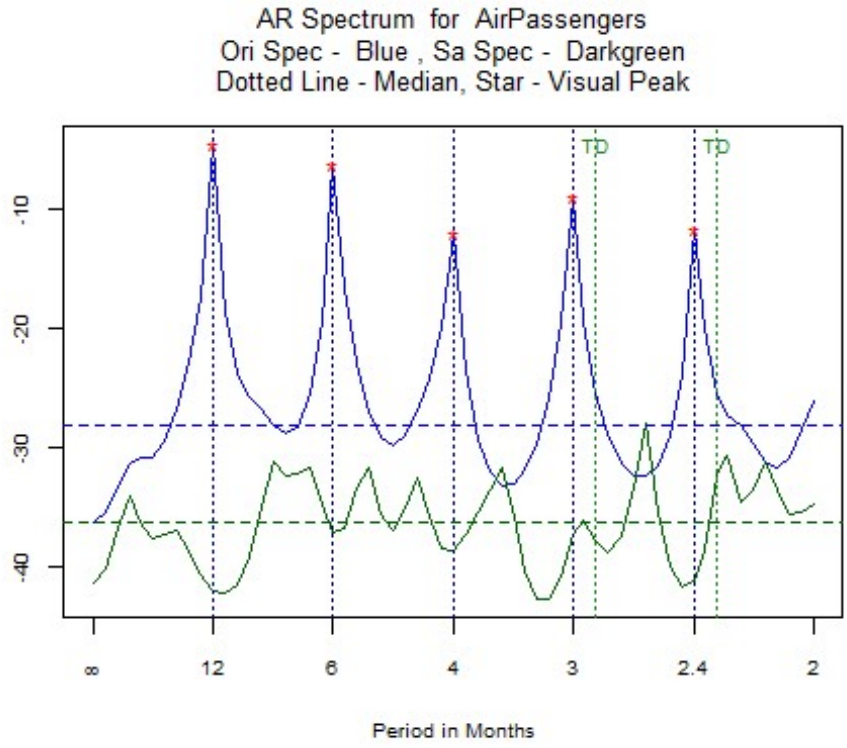

Figure 2: Example of `plot_double_spectrum`

Note that the `this_col` argument controls the colors used in the plot (Figure 2). It is a character array of length 6, and controls the color of the

- spectrum of the original series,
- spectrum of the seasonal adjusted series,
- the vertical line(s) that represents the seasonal frequencies,
- the vertical line(s) that represents the trading day frequencies,
- star(s) for visually significant seasonal frequencies,
- star(s) for visually significant trading day frequency.

The default is `c('blue', 'green', 'grey', 'brown', 'red', 'orange')`.

2.3 plot_fcst

The `plot_fcst` function generates a simply plot of the regARIMA forecasts with confidence bounds. The number of forecasts and probability for the confidence bounds are specified in the X-13ARIMA-SEATS `seasonal` run.

```
air_seas_fcst <-
  seasonal::seas(AirPassengers,
                 arima.model = '(0 1 1)(0 1 1)',
                 forecast.maxlead = 60,
                 forecast.probability = 0.99)
blsplotR::plot_fcst(air_seas_fcst,
          main_title = 'Forecasts for Airline Passengers',
          do_grid = TRUE)
```

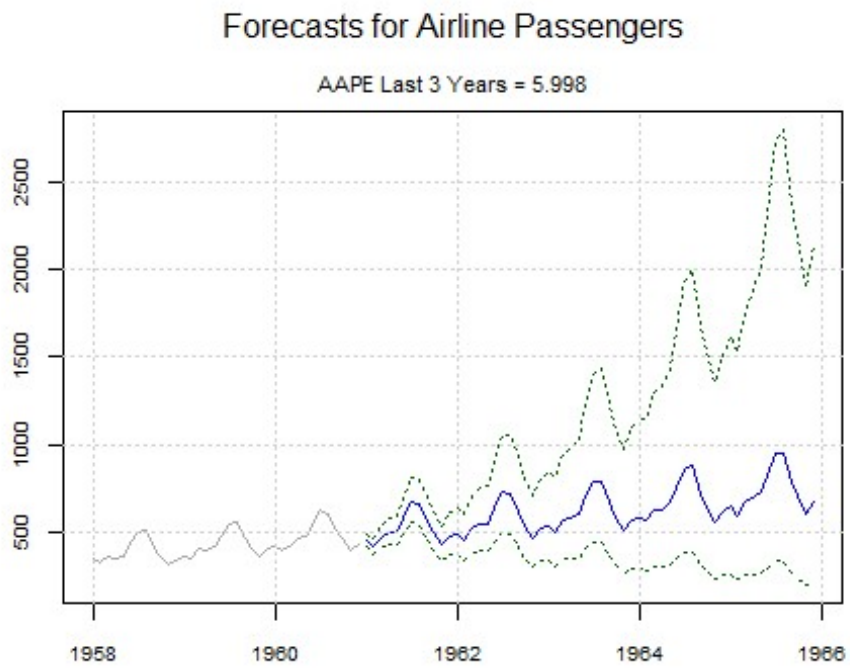

Figure 3: Example of `plot_fcst`

In the `seasonal` run, the number of forecasts are set to 60 (5 years) using the `forecast.maxlead` argument, and the confidence interval probability is set to 0.99 with the `forecast.probability` argument. These settings cannot be set within the `plot_fcst` function - you would need to run the `seasonal` package again with different settings.

2.4 plot_fcst_history

The `plot_fcst_history` function generates a forecast history plot, which compares the sum of squared forecast errors for two models fit to the same series. This plot is a model selection diagnostic - if there is a consistent direction to the difference in the cumulative forecast error, this indicates that one model forecasts better than the other model.

```
air_seas_mdl <-
   seasonal::seas(AirPassengers, x11='', slidingspans = '',
      transform.function = 'log', arima.model = '(0 1 1)(0 1 1)',
      regression.aictest = NULL, outlier = NULL,
      forecast.maxlead = 36, check.print = c( 'pacf', 'pacfplot' ),
      history.fstep = c(1, 12), history.estimates = 'fcst',
      history.save = 'fcsterrors')
air_seas_mdl2 <-
   seasonal::seas(AirPassengers, x11='', slidingspans = '',
      transform.function = 'log', arima.model = '(0 1 1)(0 1 1)',
      regression.variables = c("td"), forecast.maxlead = 36,
      check.print = c( 'pacf', 'pacfplot' ),
      history.fstep = c(1, 12), history.estimates = 'fcst',
      history.save = 'fcsterrors')
blsplotR::plot_fcst_history(air_seas_mdl, air_seas_mdl2,
      start_hist = 1957.0,
      main_title = 'Forecast History Plot for Airline Passengers',
      name_mdl1 = 'Airline model',
      name_mdl2 = 'Airline model + regressors')
```

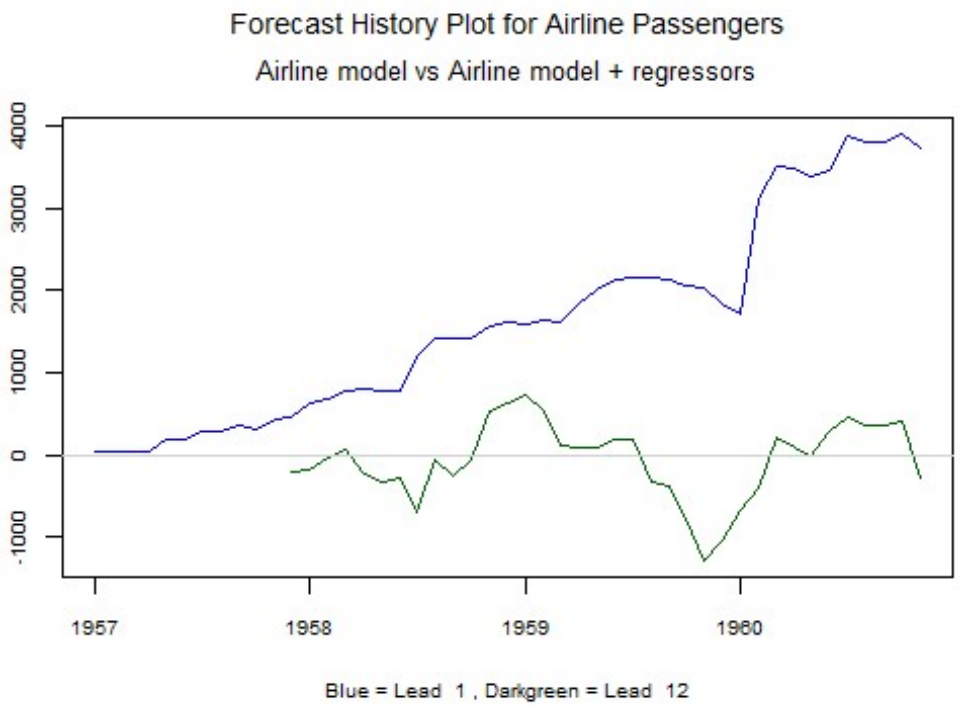

Figure 4: Example of `plot_fcst_history`

You will need to generate `seas` objects using the `seasonal` package for the two models you want to compare before you run the `plot_fcst_history` function.

In this example, the base ARIMA model is the (0 1 1)(0 1 1) model in both cases. In the first case, there are no regressors included in the model. In the second case, a full set of trading day regressors is included in the model, and the automatic outlier identification routine is included.

In the graph (Figure 4), there is a clear preference for the model without regressors for one-step ahead forecasts, since the difference between the forecast errors of the two models is increasing consistently. There is not a clear preference for seasonal forecasts - the differences oscillate around zero.

2.5 plot_fts

The `plot_fts` function generates a plot of the final t-statistics for the outlier identification procedure. The `seasonal` run used as input for this function needs to have the outlier identification procedure active, and the additional step of saving the final t-statistics using the `series` function of the `seasonal` package is required.

```
air_seas_outlier <- seasonal::seas(AirPassengers,
    arima.model = '(0 1 1)(0 1 1)', outlier.types = 'all')
air_fts_good <- seasonal::series(air_seas_outlier, "fts")
blsplotR::plot_fts(air_seas_outlier, air_fts_good,
    main_title = 'Outlier T-Values for Airline Passengers')
```

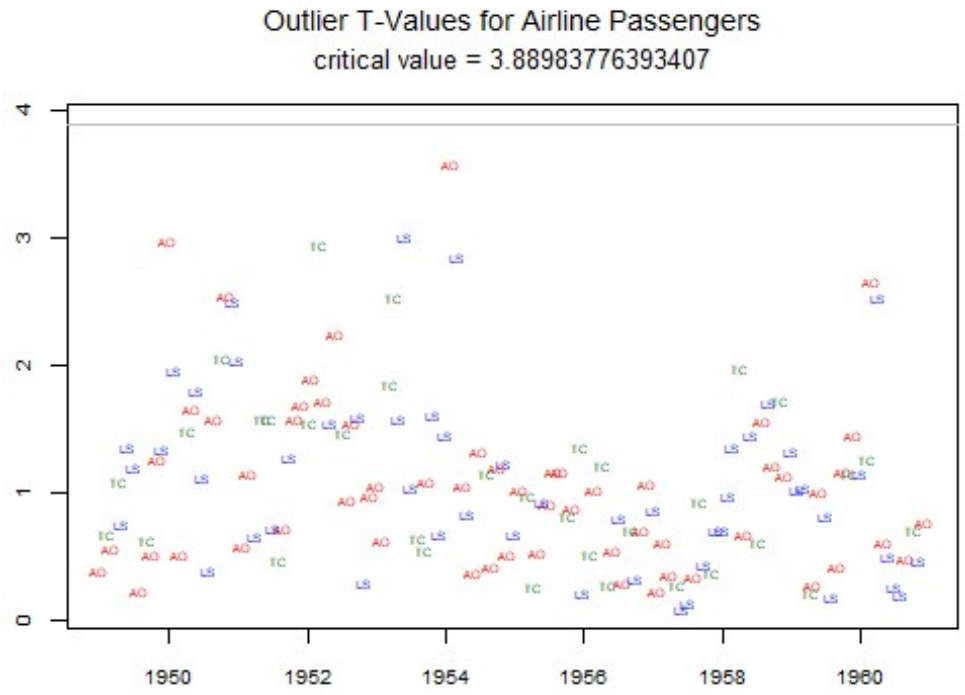

Figure 5: Example of `plot_fts`

In the plot (Figure 5), there is a horizontal line where the outlier critical value lies, and this value appears as a sub-header in the plot. Each type of outlier is noted by a two character text (AO for point outliers, LS for level shift outliers, TC for temporary change outliers), displayed in different colors.

In the next plot (Figure 6), the outlier run has multiple critical values, so there is more than one horizontal line that denotes the critical value. Also, setting the `add_identified_otl` argument to `TRUE` adds the one identified AO to the plot - if you want to see the t-values of the identified outliers, you'll need to set `add_identified_otl = TRUE`.

```
air_seas_outlier <- seasonal::seas(AirPassengers,
    outlier.critical = c(4.0, 5.0, 4.0),
    arima.model = '(0 1 1)(0 1 1)', outlier.types = 'all')
air_fts_good <- seasonal::series(air_seas_outlier, "fts")
blsplotR::plot_fts(air_seas_outlier, air_fts_good,
    add_identified_otl = TRUE,
    main_title = 'Outlier T-Values for Airline Passengers')
```

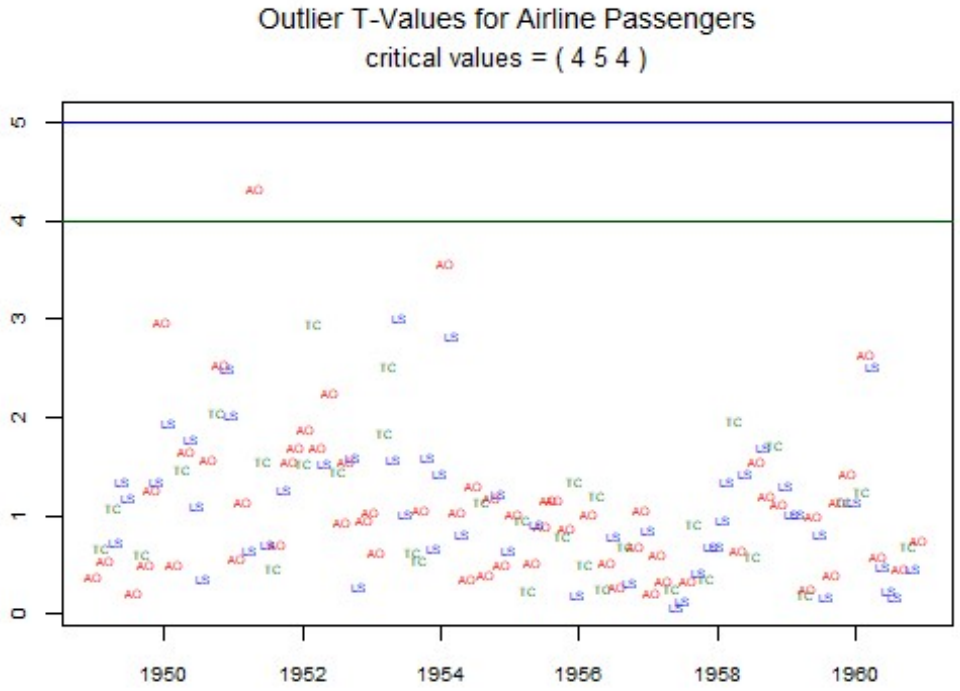

Figure 6: Example of `plot_fts` using `add_identified_otl`

2.6 plot_ratio

The `plot_ratio` function generates a high-definition plot that can be formed around a reference line other than zero. For multiplicative seasonal adjustments, the ratios are centered on one rather than zero.

This function allows the user to generate such a plot with the reference line set at the mean, which for multiplicative seasonal adjustments is assumed to be one.

```
air_seas <-
  seasonal::seas(AirPassengers, transform.function= 'log',
                 arima.model = '(0 1 1)(0 1 1)')
air_sf <- seasonal::series(air_seas, 's10')
blsplotR::plot_ratio(air_sf, ratio_color = 'darkblue',
    main_title = 'SEATS seasonal for Airline Passenger')
```

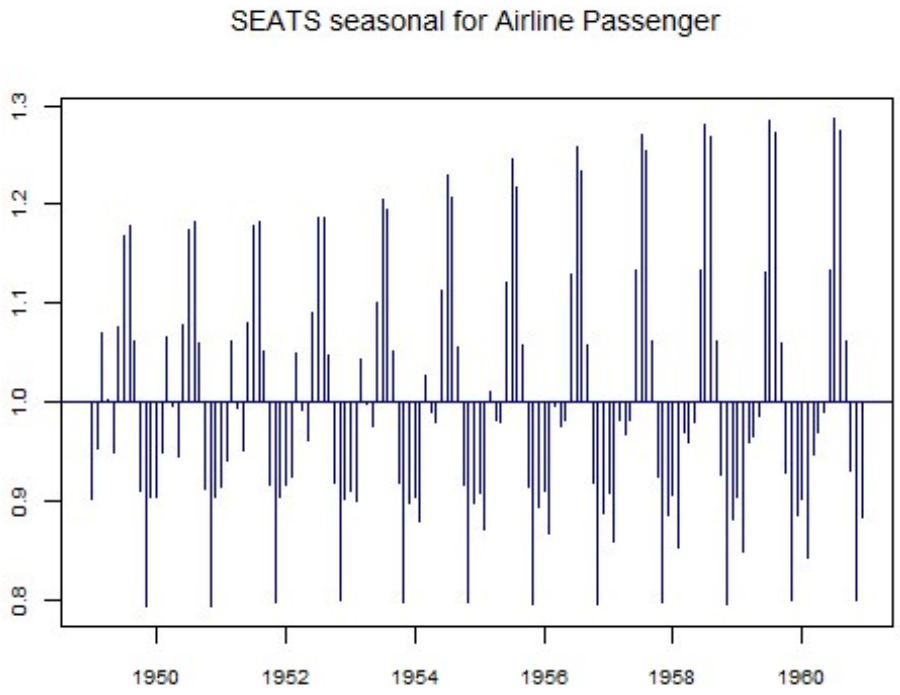

Figure 7: Example of `plot_ratio`

Note that the `plot_ratio` function takes as its first argument the actual series being plotted rather than a seas object. This is why we use the `series` function of the `seasonal` package again to extract the SEATS seasonal factors (table S10) from the seas object.

This function is useful when you need to plot tables from the X-13 output that are expressed as ratios - seasonal factors, SI-ratios, irregular, etc.

You can plot factors from an additive adjustment by setting the `ratio_mean` argument to zero (`ratio_mean = 0.0`).

2.7 plot_resid

The function `plot_resid` generates a plot of the regARIMA residuals with diagnostic information from the regARIMA model estimation incorporated into the sub-headers.

An example of the residual plot is given below (Figure 8). There are several tests for normality in the residuals that are shown in the sub-header of the plot - Geary's a, kurtosis, and test for skewness in the residuals. In this case, all the diagnostics are insignificant.

```
air_seas <-
    seasonal::seas(AirPassengers,
                   arima.model = '(0 1 1)(0 1 1)')
blsplotR::plot_resid(air_seas, use_ratio = TRUE, this_col='darkblue',
   main_title = 'ARIMA Residuals for Airline Passengers')
```

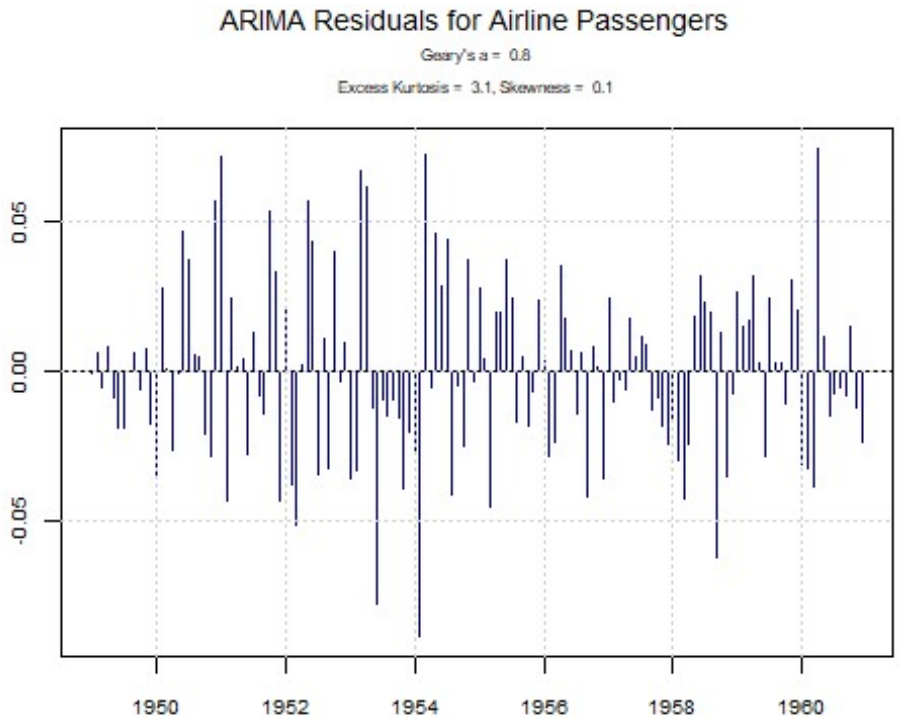

Figure 8: Example of `plot_resid`

Note that the ratios are plotted due to the `use_ratio` argument being set equal to `TRUE`. We could also generate a simple line plot of the residuals by omitting this argument, as we do below (Figure 9).

```
air_seas <-
    seasonal::seas(AirPassengers,
                   arima.model = '(0 1 1)(0 1 1)')
blsplotR::plot_resid(air_seas, this_col='darkblue',
   main_title = 'ARIMA Residuals for Airline Passengers',
   this_sub_cex = 0.75, this_plot_cex = 0.9,
   use_ratio = FALSE)
```

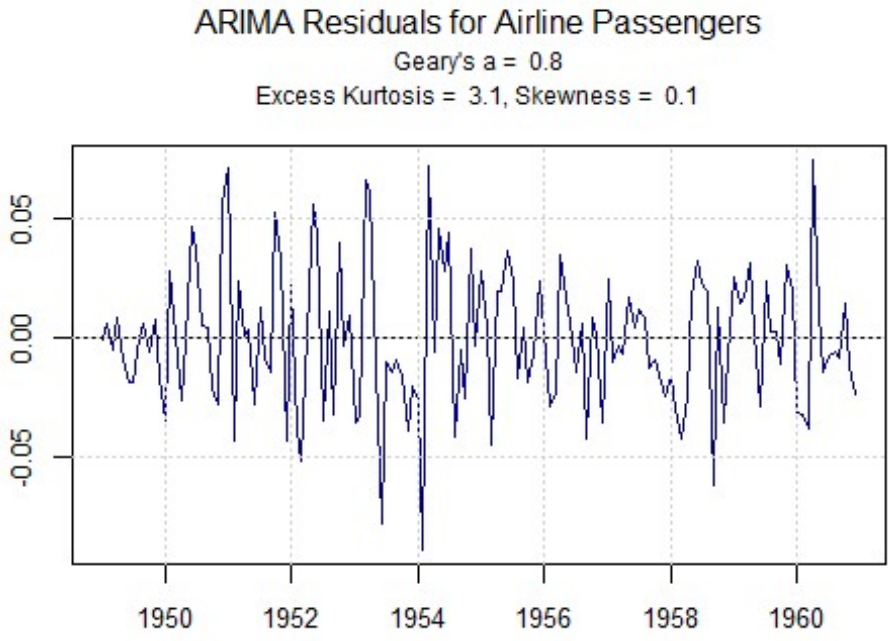

Figure 9: Example of `plot_resid` with `use_ratio = FALSE`

Note also that the size of the subtitle text has been changed by using the `this_sub_cex` argument, and the size of other text and characters in the plot are larger due to the value of the `this_plot_cex` argument. These arguments are examples of how you can fine tune elements of your plots.

2.8 plot_series

The `plot_series` function produces a plot of a given time series. This time series does not have to be the output of a seasonal adjustment process - our first graph (Figure 10) is just such a plot. It shows the Airline Passenger series starting in January of 1958 using the `start_plot` argument.

Note that we are also including information on the start and end of recessions that occurred during the span of the series using the `draw_recess` argument. Finally, a grid is added to the plot using the `draw_grid` argument.

```
blsplotR::plot_series(AirPassengers, y_label = 'Air Passengers',
            do_grid = TRUE, draw_recess = TRUE,
            this_col = 'black', start_plot = c(1958,1),
            this_point_type = 1)
```

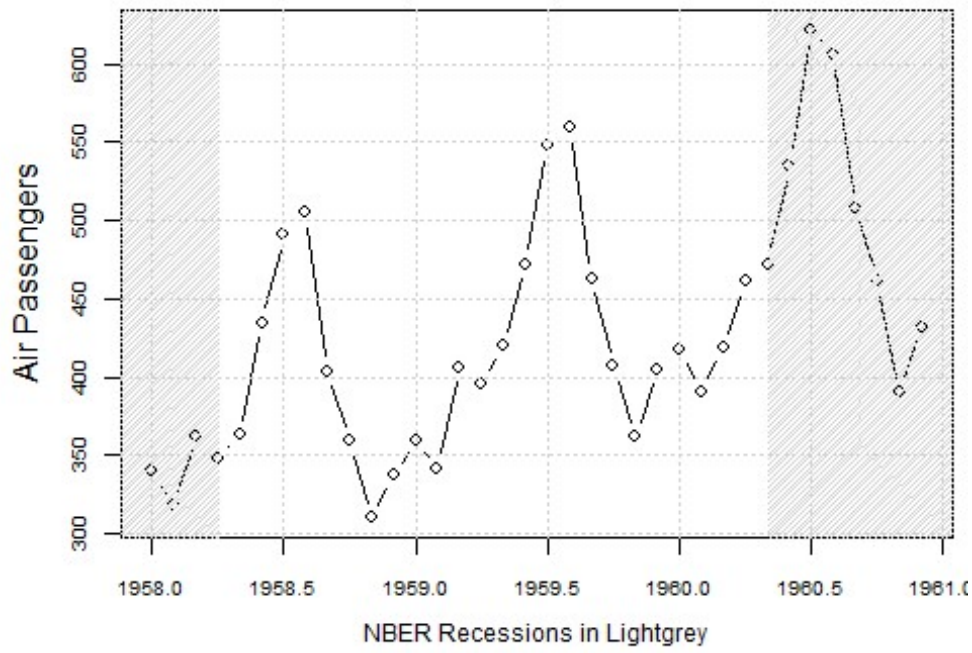

Figure 10: Example of `plot_series`

In the second plot (Figure 11), we generate an X-11 seasonal adjustment, and use the `lines` function to include lines for the X-11 seasonally adjusted series (extracted using the `final` function of the `seasonal` package) and the X-11 trend component (using the `trend` function of the `seasonal` package) in the plot.

In addition, there are arguments that set up a legend for the plot (`add_legend = TRUE`) that will be positioned in the upper left hand corner of the plot (`this_legend_position = "topleft"`); a number of other options for the legend are set as well. This allows users to create more flexible plots - we'll show other examples of plotting multiple series later in this document.

```
air_seas_x11 <-
  seasonal::seas(AirPassengers,  x11 = "",
                 arima.model = '(0 1 1)(0 1 1)',
                 outlier.types = "all",
                 forecast.maxlead = 36)
blsplotR::plot_series(AirPassengers,
    y_label = 'Air Passengers', do_grid = TRUE,
    draw_recess = TRUE, this_col = 'black',
    start_plot = c(1958,1), this_point_type = 1,
    main_title = "X-11 Seasonal Adjustment for Airline Passengers",
    add_legend = TRUE,
    this_legend_position = "topleft",
    this_legend_title = "Air Passengers",
    this_legend_inset = 0,
    this_legend_entry = c("Series", "SA", "Trend"),
    this_legend_col = c("black", "blue", "darkgreen"),
    this_legend_lty = 1:3,
    this_reset = FALSE)
air_x11_sa    <- window(seasonal::final(air_seas_x11), start=c(1958,1))
air_x11_trend <- window(seasonal::trend(air_seas_x11), start=c(1958,1))
lines(air_x11_sa, col = "blue", lty = 2)
lines(air_x11_trend, col = "darkgreen", lty = 3)
blsplotR::reset_par()
```

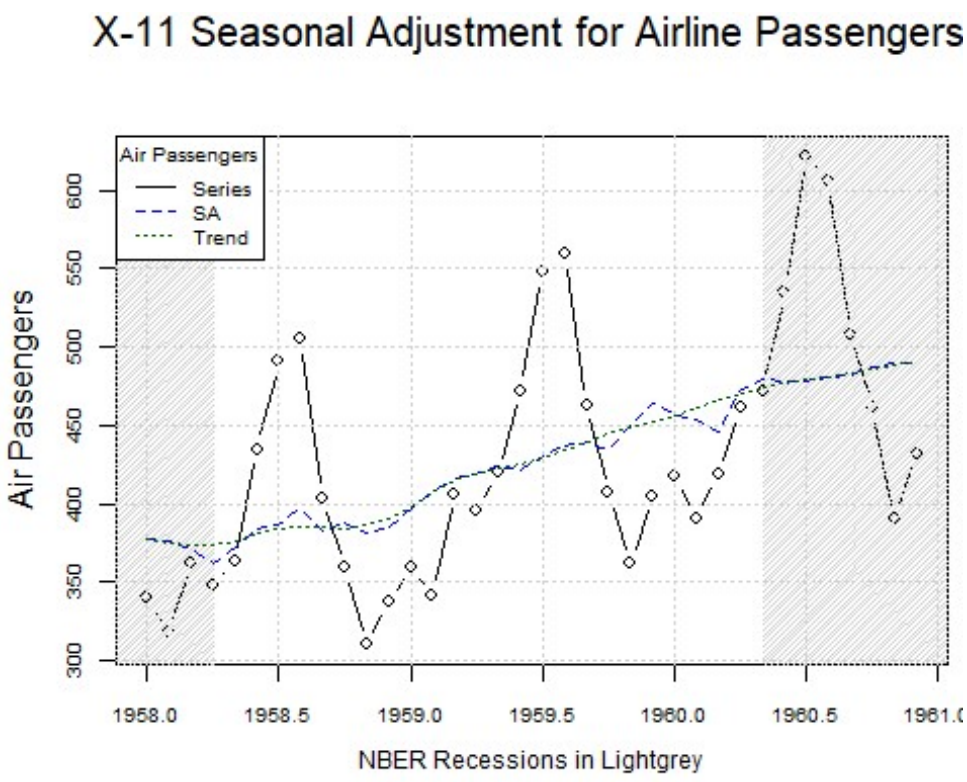

Figure 11: Example of `plot_series` as first plot in sequence with legend

Note that in order for the lines to be added to the plot, you need to set the `this_reset` argument (`this_reset = FALSE`). The default for this argument resets the graphics parameters that are controlled by the `par` function in base R.

At the end of this code snippet, the `reset_par` function resets the graphics parameters to their default.

2.9 plot_sf

The `plot_sf` function generates a special plot of the seasonal factors (and optionally the SI-ratios, a detrended series generated by X-11) grouped by month or quarter.

The plot below (Figure 12) shows an example of a simple seasonal sub-plot. Each month has a line that represents the evolution of the seasonal factors for that month around the mean of the seasonal factors for that month.

```
air_seas <-
  seasonal::seas(AirPassengers,
                 arima.model = '(0 1 1)(0 1 1)',
                 x11='')
blsplotR::plot_sf(air_seas,
        main_title = 'Air Passengers Seasonal Sub-Plots')
```

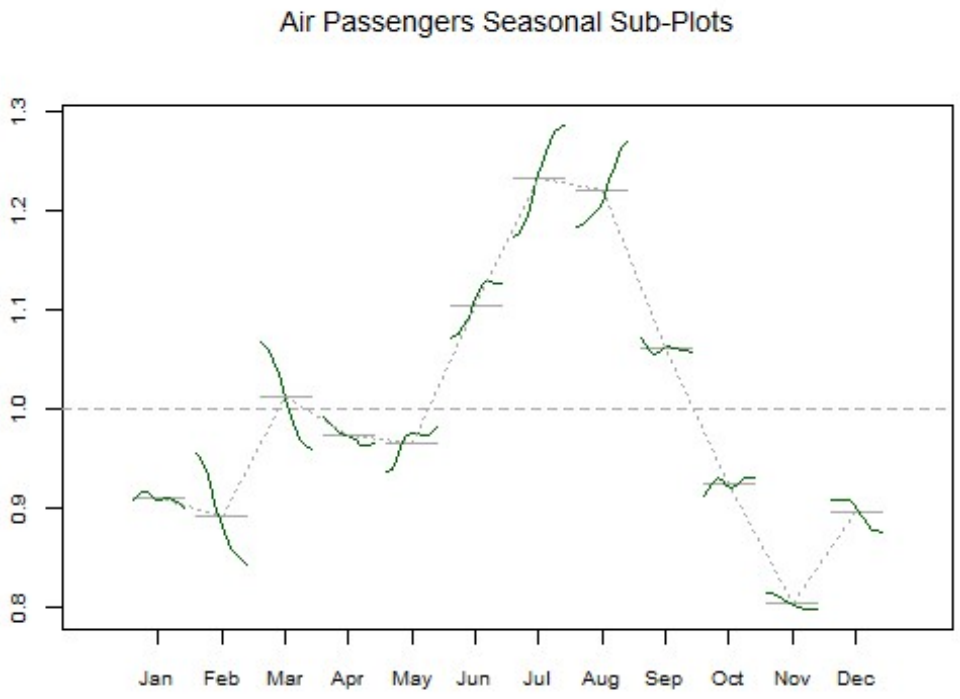

Figure 12: Example of `plot_sf`

In the next plot (Figure 13), including the argument `add_si` includes a high-density plot of the X-11 SI-ratios centered on the mean of the seasonal factors for each month. This plot

also includes a plot legend in the upper left corner of the plot by specifying `add_legend = TRUE`.

```
blsplotR::plot_sf(air_seas,
        add_si = TRUE,
        main_title = 'Air Passengers Seasonal Sub-Plots',
        this_col = c('darkgreen', 'darkblue', 'grey'),
        add_legend = TRUE)
```

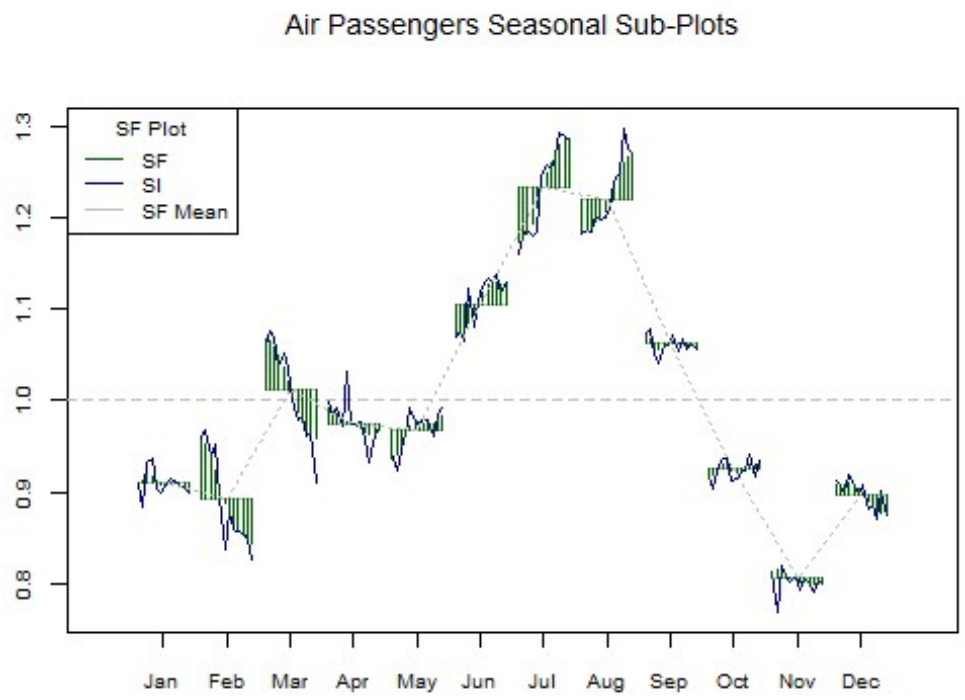

Figure 13: Example of `plot_sf` with `add_si = TRUE`

Note that the color of each type of lines is controlled by the `this_col` argument. This is a character array of length three, and sets the color used for seasonal factors, SI-ratios, and seasonal mean. Note that this order needs to be maintained even if SI-ratios are not included in the plot.

2.10 plot_year_over_year

The `plot_year_over_year` function generates a year-over-year plot - a plot of a user-specified time series with each year as a separate line. This plot is often used to evaluate the seasonality of a given time series.

The only required input is a time series object to be graphed. In this example, we have specifed arguments that adjust the right hand margin of the plot (`this_right_mar = 5.75`) and the amount of space used to position the plot legend outside the main plot (`this_legend_inset = -0.2`). These arguments are only specified when we need to make

adjustments for the position of the legend and to keep the legend from overwriting the main plot.

```
blsplotR::plot_year_over_year(AirPassengers,
    main_title = "Airline Passenger Series (1949 - 1960)",
    this_right_mar = 5.75,
    this_legend_inset = -0.2)
```

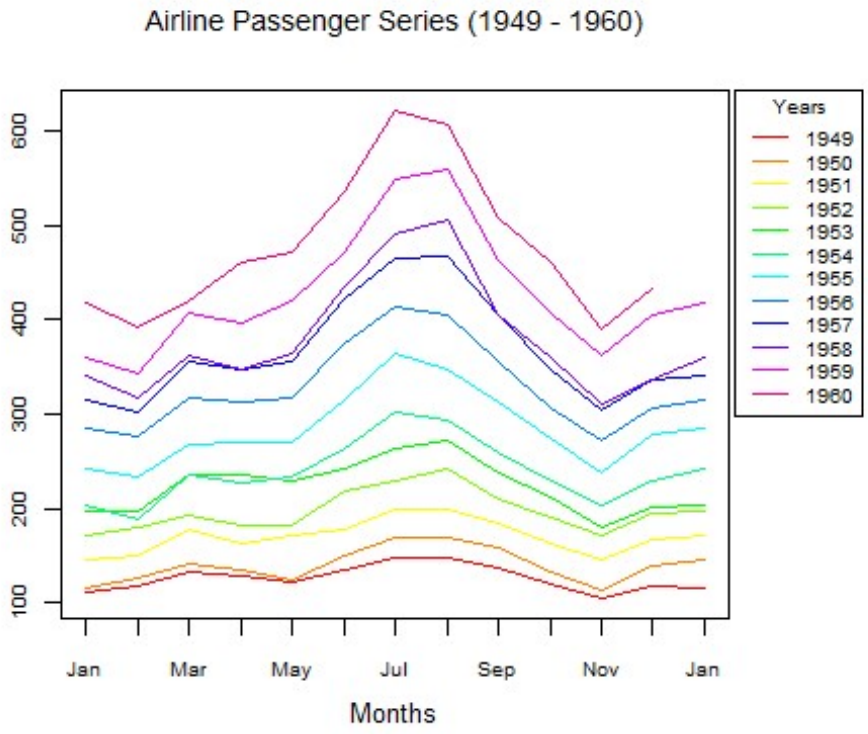

Figure 14: Example of `plot_year_over_year`

This plot (Figure 14) uses the `rainbow` function to generate the colors used for the lines for each year. The legend on the right hand side of the plot is generated when `this_legend = TRUE`.

3 Multiple plots

The next set of functions produce plots of multiple time series, or multiple options applied to the same series. Some of these functions allow users to generate a number of different plots for the same series - they also provide options to save these plots into graphics files, such as png or PDF. In these cases, we'll show the code used in those situations, but we may not show examples of the plots.

3.1 plot_all

The `plot_all` function generates 10 different diagnostic plots from a seasonal seas object:

- Unadjusted versus seasonally adjusted series
- Trend
- Seasonal sub-plots
- Seasonal factor plot
- Irregular factor plot
- Forecasts
- Spectrum of the original series and the seasonally adjusted series
- Final outlier t-statistics
- ARIMA residuals
- Cumulative periodogram

Many of these plots are generated by the functions covered in the last section of this document.

```
```{r}
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
      forecast.maxlead = 60, x11='',
      check.print = c("none", "+acf", "+acfplot", "+normalitytest"))

blsplotR::plot_all(air_seas, series_name = 'AirPassengers',
         file_base = 'AirPass',
         this_dir = 'X:/seasonalAdj/graphs/',
         split_plots = TRUE, plot_type = 'png', this_grid = FALSE,
         this_draw_recess = TRUE,
         this_ratio = TRUE, this_add_identified_otl = TRUE,
         col_sa = 'darkred', col_one = 'steelblue')

blsplotR::plot_all(air_seas, series_name = 'AirPassengers',
         file_base = 'AirPass',
         this_dir = 'X:/seasonalAdj/graphs/',
         this_grid = TRUE,
         this_ratio = TRUE, this_add_identified_otl = TRUE,
         col_factor = 'darkorange',
         col_fcst = c('steelblue', 'forestgreen', 'darkred'),
         col_sa = 'orange', col_one = 'violet')
```
```

In the code block above, the `plot_all` function is used twice - one to create separate png files for each type of plot, and the other groups all the plots into a PDF file. The `split_plots` argument is used to generate separate graphics files (`split_plots = TRUE`).

Note that the `devEval` function of the `R.devices` package is used as the engine to generate these graphics files. For more information on the `R.devices` package, access https://cran.r-project.org/web/packages/R.devices/vignettes/R.devices-overview.pdf.

3.2 plot_matrix

The function `plot_matrix` generates a plot from a matrix of user-specified time series. All the series are plotted on the same axis. Users can generate a legend for the plot.

In the example below (Figure 15), a matrix of regional building permits series for one family units is created, and a plot of all four series is generated.

- A legend is placed in the upper left-hand corner.
- Recession periods are drawn and grid lines are drawn, and
- the color of each line drawn is set.

```
BP_Region_Matrix <-
   cbind(xt_data_list$mw1u, xt_data_list$ne1u,
         xt_data_list$so1u, xt_data_list$we1u)
colnames(BP_Region_Matrix) <- names(xt_data_list)
blsplotR::plot_matrix(BP_Region_Matrix,
     main_title = "US Building Permits, 1 Family Units",
     do_grid = TRUE, draw_recess = TRUE,
     this_col = c("red", "steelblue", "forestgreen", "brown"),
     y_label = 'Building Permits', add_legend = TRUE,
     this_legend_title = NULL, this_legend_cex = 0.75,
     this_legend_entry = names(xt_data_list))
```

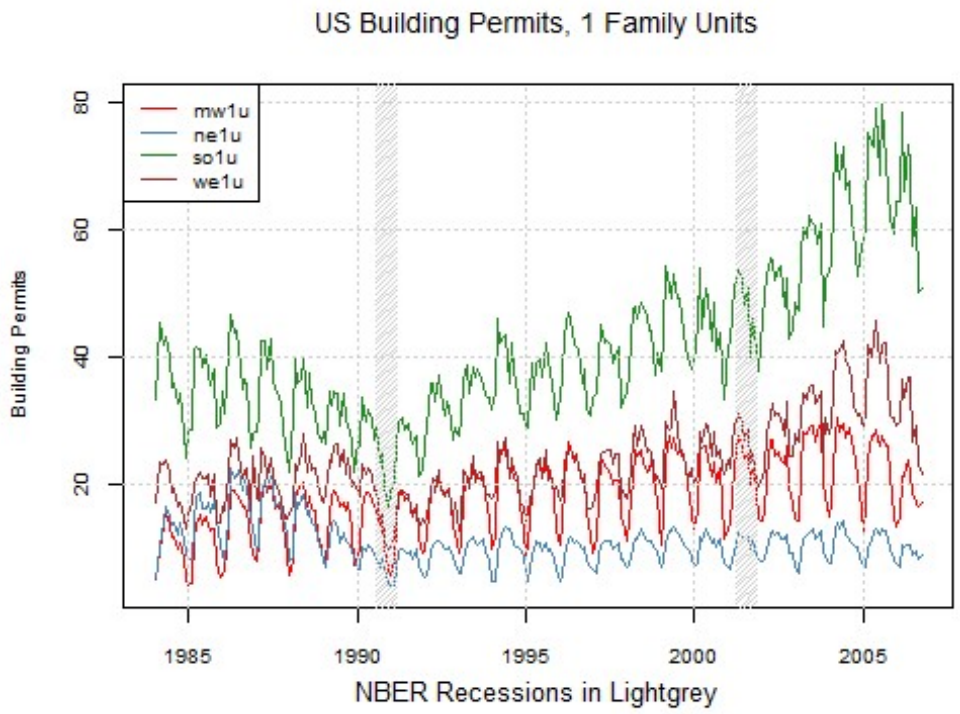

Figure 15: Example of `plot_matrix`

3.3 plot_sa_list

The function `plot_sa_list` generates a page of plots for a time series, seasonal adjustment of the time series, and trend component. Plotting the trend is optional.

The input is list object of seas objects. The `seas_obj_names` argument allows the user to set the names of each plot. Users can control how many plots can appear on the page by setting the number of rows (`this_row`) and columns (`this_col`) of plots appear on the page, and the range of series to be plotted (`first_series`, `last_series`).

In this function, you can specify the name of a PDF file (`pdf_file`) where the plot(s) will be stored as well as the directory the file is saved (`this_dir`).

This example (Figure 16) uses the `plot_start` argument to set the starting date for all the plots generated. Here we are setting the starting date to January 2015. There are also arguments specified that control the size of margins, axis labels, or title text. We'll talk more about that in the next plot.

```r
EM_individual_seas <- seasonal::seas(
    x11 = "", transform.function = "log",
    check.print = c("none", "+acf", "+acfplot", "+normalitytest"),
    regression.aictest = NULL,
    outlier.types = "all",
    arima.model = "(0 1 1)(0 1 1)",
    list = list(
        list(x = blsplotR::employment_list$n2000013),
        list(x = blsplotR::employment_list$n2000014),
        list(x = blsplotR::employment_list$n2000025),
        list(x = blsplotR::employment_list$n2000026)
    )
)

# Use Filter function to grab seas objects
EM_individual_seas_only <-
  Filter(function(x) inherits(x, "seas"), EM_individual_seas)

EM_names <- c("Male 16-19", "Female 16-19", "Male 20+", "Female 20+")
names(EM_individual_seas_only) <- names(blsplotR::employment_list)

blsplotR::plot_sa_list(EM_individual_seas_only,
       this_row = 2, this_col = 2,
       plot_trend=TRUE, seas_obj_names = EM_names,
       group_title='US Employment',
       col_vec = c("grey", "steelblue", "forestgreen"),
       plot_start=c(2015,1), this_main_cex = 0.75,
       this_axis_cex = 0.5, main_title_line = 0.5,
       this_mar = c(2.5, 2.0, 2.0, 0.25))
```

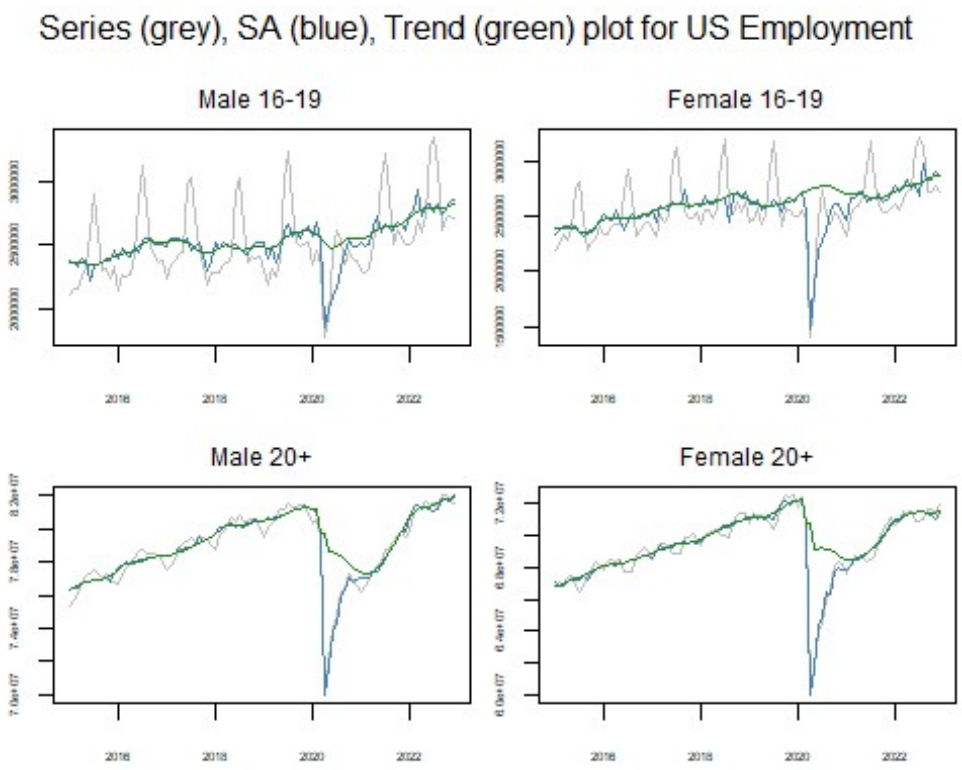

Figure 16: Example of `plot_sa_list`

Note in this second example (Figure 17) the plots are configured differently - there is one column of four plots, which allows users to make comparisons between the different series easier.

To facilitate this, we have specifed arguments that adjust:

- the margins of the plot (`this_mar = c(2.0, 2.0, 1.75, 0.25)`) and the size of the plot titles and
- axis labels(`this_main_cex = 0.5, this_axis_cex = 0.5`).

These arguments are available in other functions to make adjustments in the produced plots.

```
EM_individual_seas <- seasonal::seas(
   x11 = "", transform.function = "log",
   check.print = c("none", "+acf", "+acfplot", "+normalitytest"),
   regression.aictest = NULL, outlier.types = "all",
   arima.model = "(0 1 1)(0 1 1)",
   list = list(
       list(x = blsplotR::employment_list$n2000013),
       list(x = blsplotR::employment_list$n2000014),
       list(x = blsplotR::employment_list$n2000025),
       list(x = blsplotR::employment_list$n2000026)
   )
)
# Use Filter function to grab seas objects
EM_individual_seas_only <-
  Filter(function(x) inherits(x, "seas"), EM_individual_seas)
EM_names <- c("Male 16-19", "Female 16-19", "Male 20+", "Female 20+")
names(EM_individual_seas_only) <- names(blsplotR::employment_list)
blsplotR::plot_sa_list(EM_individual_seas_only,
 this_row = 4, this_col = 1, plot_trend=TRUE,
 seas_obj_names = EM_names, group_title='US Employment',
 col_vec = c("grey", "steelblue", "forestgreen"),
 plot_start=c(2015,1), this_main_cex = 0.75, this_axis_cex = 0.5,
 this_mar = c(2.0, 2.0, 1.75, 0.25), main_title_line = 0.5)
```

Figure 17: Example of `plot_sa_list` with `this_row = 4, this_col = 1`

3.4 plot_sa_list_split

The `plot_sa_list_split` is very similar to the `plot_sa_list` function described earlier, as it produces plots of seasonally adjusted series for seasonal objects in a list. This function differs in that it splits pages of plots into individual graphics files.

Here, the function generates a page of plots for a time series, seasonal adjustment of the time series, and trend component. Plotting the trend is optional. Each page is saved to a different graphics file, the format of which is controlled by the `plot_type` argument (the default is png).

```
```{r}
EM_seas <- seasonal::seas(
       blsplotR::employment_list,
       slidingspans = "",
       transform.function = "log",
       arima.model = "(0 1 1)(0 1 1)",
       forecast.maxlead = 36,
       check.print = c( "pacf", "pacfplot" ))

blsplotR::plot_sa_list_split(EM_seas, this_row = 2, this_col = 1,
       plot_trend = TRUE, this_dir = 'X:/seasonalAdj/graphs/',
       seas_obj_names = c("Male 16-19", "Female 16-19",
                          "Male 20+", "Female 20+"),
       file_name_base='EM_sa_trend', plot_type = 'png',
       group_title='US Employment',
       plot_start=c(2010,1))
```
```

The above code block will produce two png files - `EM_sa_trend_p1.png` and `EM_sa_trend_p2.png` - in the directory specified in the `this_dir` directory.

3.5 plot_sf_mean

The `plot_sf_mean` generates a plot of the means of the seasonal factors for a given series. The input for this function is a time series object of the seasonal factors from a seasonal adjustment generated by the `seasonal` package.

This function can be run multiple times so that you can overlay the seasonal patterns of different series (or different seasonal adjustment options for the same series).

The example shown below (Figure 18) gives an example of how to do this, and how to run multiple series in a single call of the `seasonal` package. Here the list argument specifies data from the four main components of US employment. The other arguments are applied to all four of the series.

For more information on ways to process multiple series with the seasonal package, access the vignette at https://cran.r-project.org/web/packages/seasonal/vignettes/multiple.html.

```r
EM_individual_seas <- seasonal::seas(
    x11 = "", transform.function = "log",
    check.print = c("none", "+acf", "+acfplot", "+normalitytest"),
    regression.aictest = NULL,
    outlier.types = "all",
    arima.model = "(0 1 1)(0 1 1)",
    list = list(
        list(x = employment_list$n2000013),
        list(x = employment_list$n2000014),
        list(x = employment_list$n2000025),
        list(x = employment_list$n2000026)
    )
)

# Use Filter function to grab seas objects
EM_individual_seas_only <-
  Filter(function(x) inherits(x, "seas"),  EM_individual_seas)

names(EM_individual_seas_only) <-
  c("n2000013", "n2000014", "n2000025", "n2000026")

EM_Comp_Sf <-
 cbind(seasonal::series(EM_individual_seas_only$n2000013, "d10"),
       seasonal::series(EM_individual_seas_only$n2000014, "d10"),
       seasonal::series(EM_individual_seas_only$n2000025, "d10"),
       seasonal::series(EM_individual_seas_only$n2000026, "d10"))
this_sf_limit <- range(EM_Comp_Sf)

blsplotR::plot_sf_mean(EM_Comp_Sf[,1], cycle(EM_Comp_Sf[,1]),
  this_col = 'steelblue', y_limit = this_sf_limit,
  this_freq = 12, forecast = 0,
  this_title = 'US Employment Seasonal Means',
  add_legend = TRUE,
  this_legend_position = "topleft",
  this_legend_title = "SF Means",
  this_legend_inset = 0,
  this_legend_entry = c("M 16-19", "F 16-19", "M 20+", "F 20+"),
  this_legend_col = c("steelblue", "red", "darkgreen", "purple"),
  this_legend_lty = rep(1,4),
  this_legend_cex = 0.6)

blsplotR::plot_sf_mean(EM_Comp_Sf[,2], cycle(EM_Comp_Sf[,2]),
  this_col = 'red', this_freq = 12, forecast = 0,
  add_line = TRUE)

blsplotR::plot_sf_mean(EM_Comp_Sf[,3], cycle(EM_Comp_Sf[,3]),
  this_col = 'darkgreen', this_freq = 12, forecast = 0,
  add_line = TRUE)
```

```
blsplotR::plot_sf_mean(EM_Comp_Sf[,4], cycle(EM_Comp_Sf[,4]),
  this_col = 'purple', this_freq = 12, forecast = 0,
  add_line = TRUE, this_reset = TRUE)
```

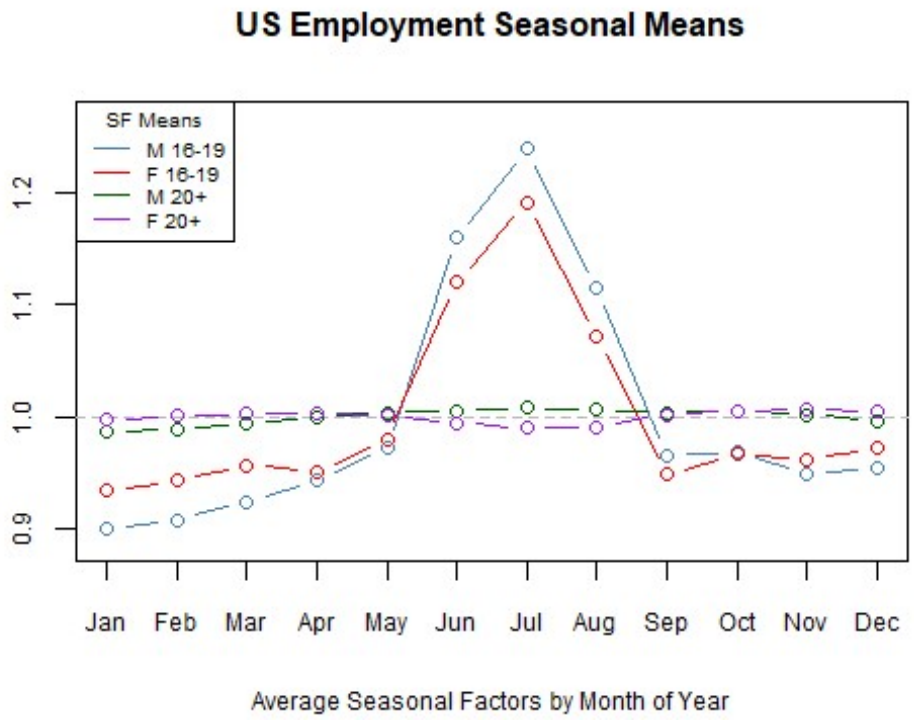

Figure 18: Example of `plot_sf_mean`

Once the seasonal adjustments are generated (and the `Filter` function is used to isolate the seas objects into the list `EM_individual_seas_only`), the seasonal factors are extracted using the `series` function of the seasonal package.

Next, each of the seasonal components are plotted using a different call of the `plot_sf_mean` function. The first two arguments of every call should be time series objects of the seasonal factors (`this_sf`) and an indicator variable that serves as an index for the month or quarter for each observation (`this_period`). Other arguments that should be specified for every call are the frequency of the time series (`this_freq`), the number of forecasts (`forecast`), and the color of the line for that seasonal factors (`this_col`).

The first call of `plot_sf_mean` generates the main plot frame, with arguments that control

- the main title(`this_title`),
- legend (`add_legend`, `this_legend_position`, and other options starting with `this_legend`), and
- limits for the y axis (`y_limit`).

The remaining calls to `plot_sf_mean` adds line to the main plot, indicated by the `add_line` argument.

3.6 plot_sf_series

The plot_sf_series function generates a special plot of the seasonal factors grouped by month/quarter. This can be done for up to two sets of seasonal factors, and allows for users to compare the seasonal factors for two different adjustments of the same series.

The example shown below (Figure 19) compares the factors for the default X-11 seasonal adjustment with those from a SEATS model based seasonal adjustment for the Airline Passengers series.

```
air_seas       <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
air_seats_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)')
air_sf         <- seasonal::series(air_seas, "d10")
air_seats_sf   <- seasonal::series(air_seats_seas, "s10")
sf_range       <- range(air_sf, air_seats_sf)

blsplotR::plot_sf_series(air_sf, air_seats_sf, y_limit = sf_range,
  add_mean_line = TRUE, add_legend = TRUE,
  main_title = 'Air Passengers Seasonal Sub-Plots',
  this_col = c('darkgreen', 'darkblue', 'lightgreen',
               'lightblue'),
  this_legend_text = c("sf(x11)", "sf(seats)", "mean(x11)",
                       "mean(seats)"),
  this_legend_color = c('darkgreen', 'darkblue', 'lightgreen',
                        'lightblue'),
  main_title_line = 1.25, this_main_cex = 0.95,
  this_plot_cex = 0.6, this_axis_cex = 0.75,
  this_mar = c(3,3,3,0.5), this_legend_cex = 0.75)
```

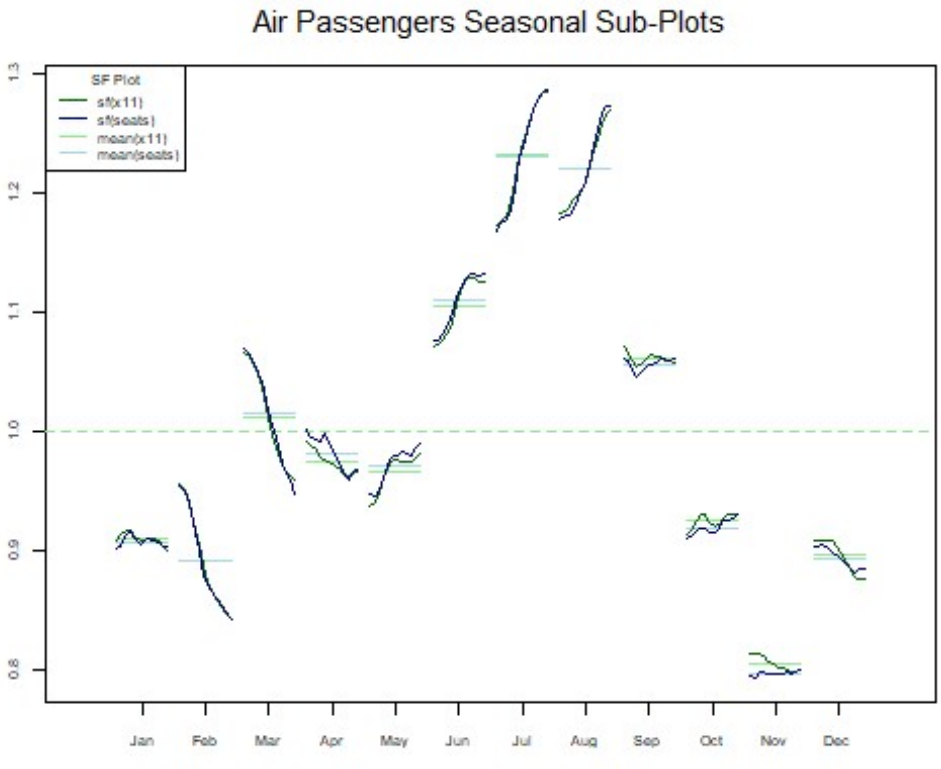

Figure 19: Example of `plot_sf_series`

3.7 plot_table

The `plot_table` function generates a plot of at least one user-specified time series. The user specifies the table from the X-13ARIMA-SEATS output, and the function extracts those tables from a seas object provided by the user.

The example below (Figure 20) shows `plot_table` generating a plot of the Airline Passenger series (table A1), the prior adjusted original series (table B1), and the seasonally adjusted series (D11). In addition to these lines, which are noted in the subtitle, the function can add lines for outliers either specified by the user or identified in the regARIMA modeling process when `add_otl = TRUE`.

```
air_seas <-
    seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                   x11='', series.save = 'b1')
blsplotR::plot_table(air_seas, c('a1', 'b1', 'd11'),
        y_label = 'AirPassengers',
        main_title = 'Airline Passengers',
        main_title_cex = 1.0, add_sub_title = TRUE,
        do_grid = TRUE, draw_recess = TRUE,
        use_ratio = TRUE, add_otl = TRUE,
        this_col = c('grey', 'darkgreen', 'darkblue'))
```

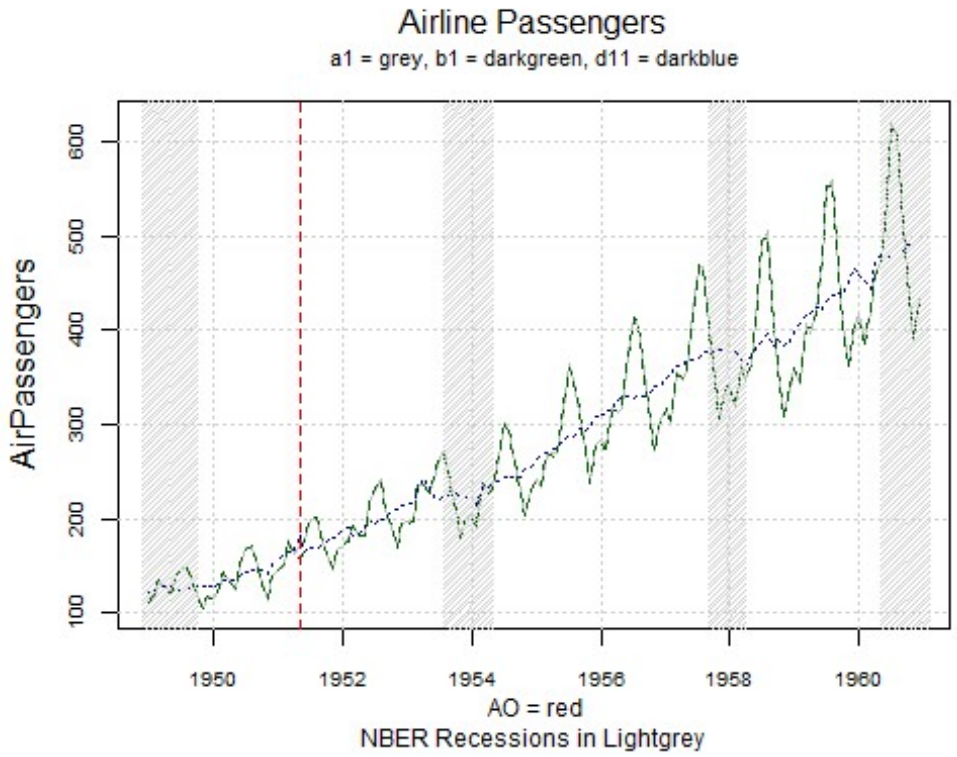

Figure 20: Example of `plot_table`

4 Utility functions

The three functions listed below are used within the functions described above to generate plots. They are usually not used by themselves, but they could be.

4.1 plot_multiple

The function `plot_multiple` generates a page of plots for a time series, seasonal adjustment of the time series, and trend component. Plotting the trend is optional. This function is called from the `plot_sa_list_split` function, or can be called on its own.

The input is list object of seas objects. In the example shown below (Figure 21), the names of the plots (`seas_obj_names`) and the number of rows and columns of plots on the page (`this_row`, `this_col`) are controlled in the same way as the `plot_sa_list` function.

```
EM_individual_seas <-
 seasonal::seas(blsplotR::employment_data_mts,
    x11 = "", transform.function = "log",
    check.print = c("none", "+acf", "+acfplot", "+normalitytest"),
    regression.aictest = NULL,
    outlier.types = "all",
    arima.model = "(0 1 1)(0 1 1)",
)

# Use Filter function to grab seas objects
EM_individual_seas_only <-
  Filter(function(x) inherits(x, "seas"), EM_individual_seas)

EM_names <- c("Male 16-19", "Female 16-19", "Male 20+", "Female 20+")

blsplotR::plot_multiple(EM_individual_seas_only,
   first_series = 1, last_series = 4, this_row = 2, this_col = 2,
   plot_trend = FALSE, col_vec = c("grey", "steelblue"),
   seas_obj_names = EM_names,
   outer_title = 'Series (grey), SA (blue) plot',
   group_title='U. S. Employment Series',
   do_grid = TRUE, draw_recess = TRUE, recess_sub = FALSE)
```

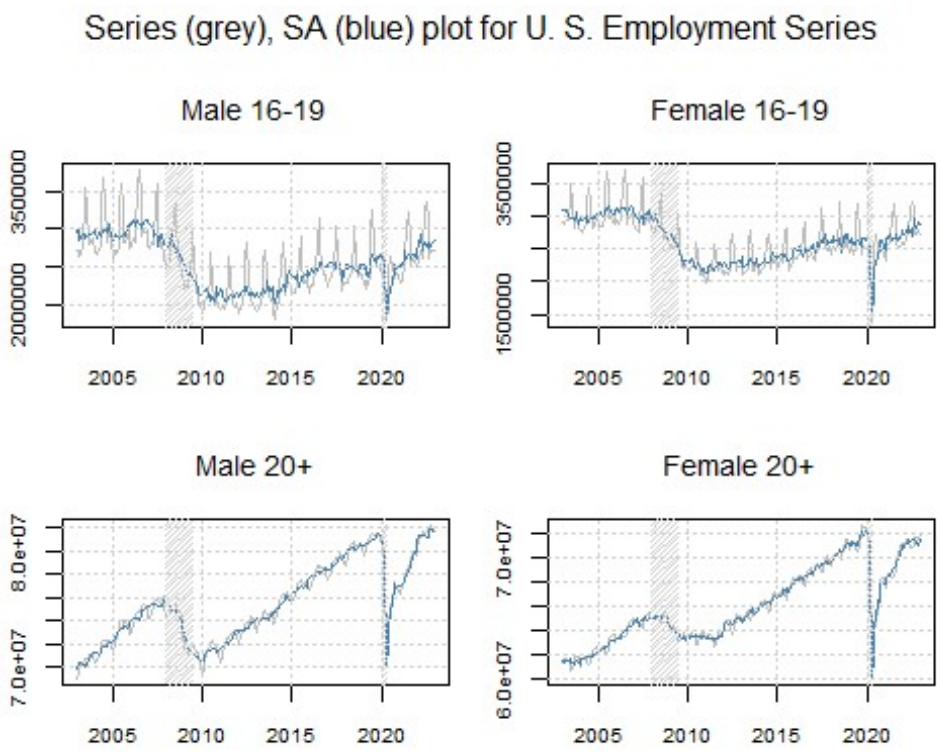

Figure 21: Example of `plot_multiple`

4.2 plot_single_cell

The `plot_single_cell` function generates a single plot of a time series, seasonal adjustment of the time series, and trend component. Plotting the trend is optional. The series name is used for the title.

This function is used within `plot_multiple` and `plot_sf_list` to generate the plots within those functions.

In this example (Figure 22), the user specifies the original series (`this_series`), seasonally adjusted series (`this_sadj`), and optionally the trend component (`this_trend`) in the first three arguments - if the trend component is not specified, the function will not plot the trend.

```
air_seas  <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
air_final <- seasonal::final(air_seas)
air_trend <- seasonal::trend(air_seas)
blsplotR::plot_single_cell(this_series = AirPassengers,
  this_sadj = air_final, this_trend = air_trend,
  this_name = 'Air Passengers')
```

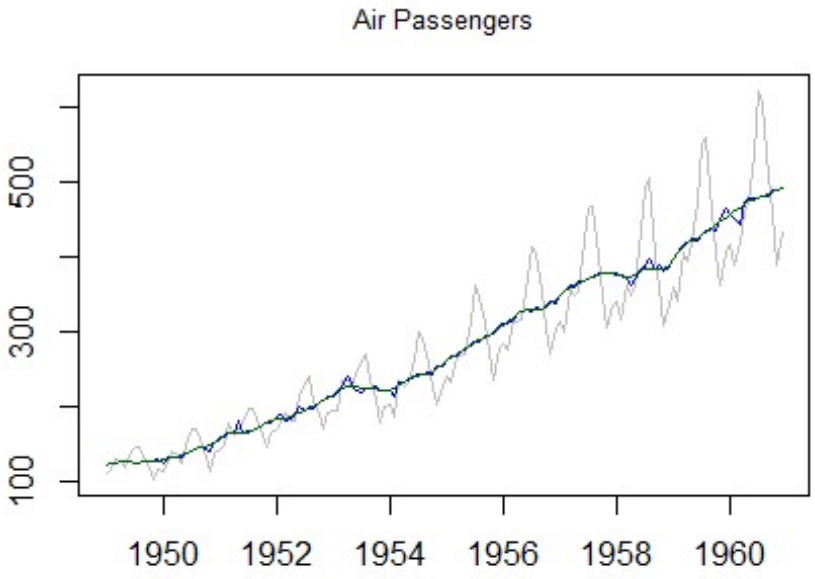

Figure 22: Example of `plot_single_cell`

4.3 reset_par

The `reset_par` function resets the graphics parameters controlled by the `par` function for plots. In many of the functions in this package, many of the default plotting parameters are changed and adjusted; once the plot is produced, it is useful to reset the graphics parameters back to their default.

When you want to add additional lines, points, or text to an existing plot, one should consider not resetting the graphics parameters until all elements of the desired plot are complete. In that case, `reset_par` should be called after all elements are added to the plot.

This function is taken from the stackoverflow post found at https://stackoverflow.com/questions/9292563/reset-the-graphical-parameters-back-to-default-values-without-use-of-dev-off.

```
par(mar = c(3.1, 3.1, 3.1, 0.5), mfrow = c(2,2))
xt_names <- names(blsplotR::xt_data_list)
for (i in 1:4) {
     plot(blsplotR::xt_data_list[[i]],
          main = xt_names[i], type = "l",
          ylab = " ", xlab = " ")
}

cat(paste0("Before reset_par: par()$mfrow = c(",
           par()$mfrow[1], ", ", par()$mfrow[2], ")"))
```

```
Before reset_par: par()$mfrow = c(2, 2)
```

```
blsplotR::reset_par()

cat(paste0("After reset_par: par()$mfrow = c(",
           par()$mfrow[1], ", ", par()$mfrow[2], ")"))
```

```
After reset_par: par()$mfrow = c(1, 1)
```

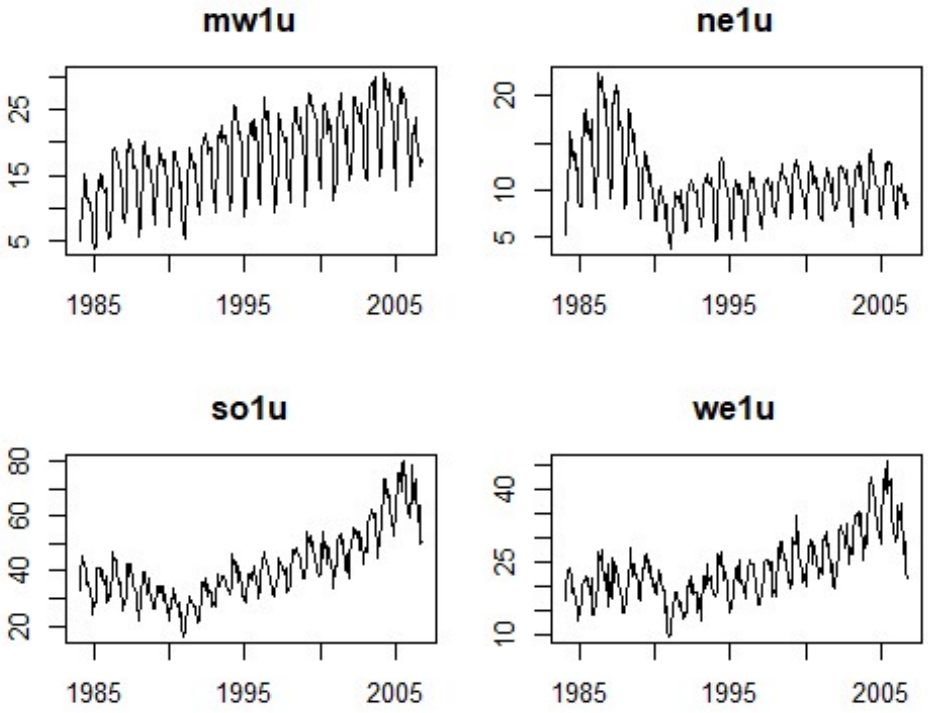

Figure 23: Example of `reset_par`

In the above example (Figure 23), the par function is used to specify that there will be two rows and two columns of plots (`mfrow`). After `reset_par` is called, `par` is reset so that there is a single plot in the panel.

5 Color related functions

There are three functions that work with colors in plots - they are used within several of the functions in this package.

5.1 cnv_color_codes

The function `cnv_color_codes` generates a vector of the closest color names from an input of hexidecimal color codes. This is used to generate color names for subheaders and other labels in plots. It uses the `color.id` function of the `plotrix` package to do the conversion.

```
Moonrise_Codes <-
   c("#F3DF6C", "#CEAB07", "#D5D5D3", "#24281A", "#798E87",
     "#C27D38", "#CCC591", "#29211F", "#85D4E3", "#F4B5BD",
     "#9C964A", "#CDC08C", "#FAD77B")
Moonrise_All <- blsplotR::cnv_color_codes(Moonrise_Codes)
Moonrise_All
```

```
 [1] "lightgoldenrod"  "gold3"           "gray83"          "gray13"
 [5] "lightcyan4"      "peru"            "wheat3"          "gray14"
 [9] "skyblue"         "rosybrown2"      "khaki4"          "wheat3"
[13] "lightgoldenrod2"
```

In this set of example code, we set up a color pallete based on that for the Wes Anderson movie Moonlight Kingdom, taken from the Wes Anderson Palette Generator developed by Karthik Ram (CRAN repository: https://cran.r-project.org/web/packages/wesanderson/index.html\%7B.uri%7D). The function gives the closest representation of the hexadecimal color code into a defined word based color code.

This is useful for plot labels and other text where a hexedecimal color code would be meaningless. We'll show an example of how close this conversion gets to the actual hexidecimal color in the discussion of the `display.color` function.

5.2 color_blind_palette

The function `color_blind_palette` generates a color palettes that can be distinguished by color-blind people and can be used in your plots. These palettes are taken from the website **Cookbook for R - Colors (ggplot2)**, which can be accessed at http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette.

The only input for this function is the argument `with_grey`, which controls whether the first element in the palette is grey (`with_grey = TRUE`) or black (`with_grey = FALSE`).

Examples of the function for both possible values of `with_grey` is shown below (Figure 24).

```
this_color_blind      <- blsplotR::color_blind_palette(FALSE)
this_color_blind_grey <- blsplotR::color_blind_palette(TRUE)

par(mfrow = c(2,1), cex = 0.75, mar= c(1.0,1.0,3.0,1.0))

barplot(rep(1, length(this_color_blind)),
        axes = F, col = this_color_blind,
        main = "Color-blind palette (with_grey = FALSE)")
barplot(rep(1, length(this_color_blind_grey)),
        axes = F, col = this_color_blind_grey,
        main = "Color-blind palette (with_grey = TRUE)")

blsplotR::reset_par()
```

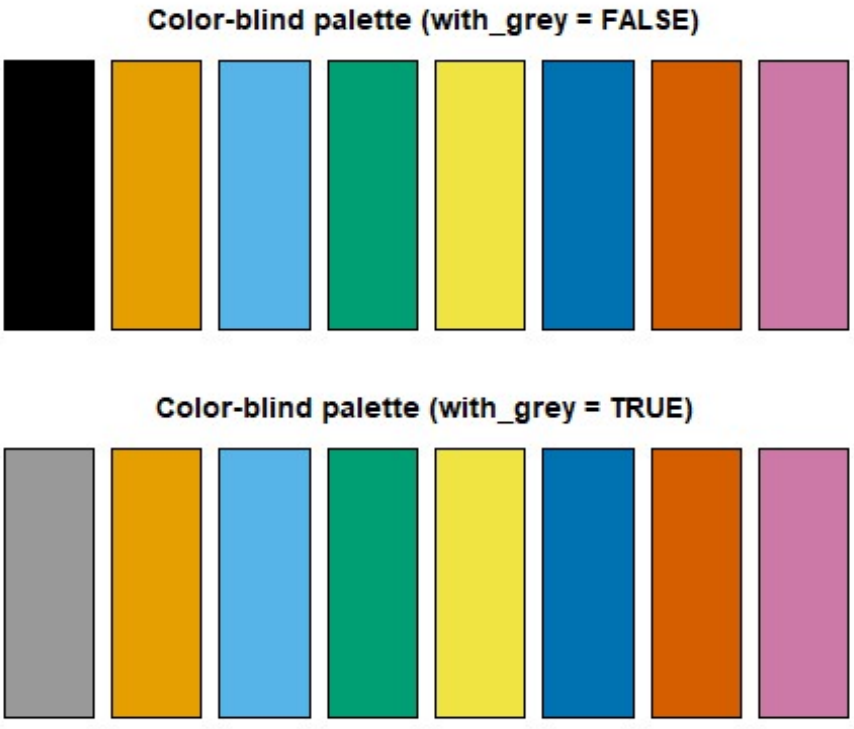

Figure 24: Example of `color_blind_palette (with_grey = FALSE)`

5.3 display_color

The function `display_color` generates color names for display on plot labels and subheaders. This function calls the `cnv_color_codes` function to convert hexidecimal color codes to the closest color name representation of these colors.

Note that this can only convert one color code at a time; as in the example below (Figure 25), if you have a palette with more than one color, you'll need to use some kind of loop.

```r
this_color_blind <- blsplotR::color_blind_palette(FALSE)
this_color_blind_names <-
  blsplotR::cnv_color_codes(this_color_blind)
this_color_blind_display <- array(NA, dim = 8)
for (i in 1:8) {
  this_color_blind_display[i] <-
       blsplotR::display_color(this_color_blind[i],
                               strip_numbers = TRUE)
}

par(mfrow = c(3,1), cex = 0.75, mar= c(1.5,1.0,3.0,1.0))

barplot(rep(1, length(this_color_blind)), axes = F,
        col = this_color_blind,
        main = "Color-blind palette from color_blind_palette")
barplot(rep(1, length(this_color_blind)), axes = F,
        col = this_color_blind_names,
        main = "Color-blind palette after cnv_color_codes")
barplot(rep(1, length(this_color_blind)), axes = F,
        col = this_color_blind_display,
        main = "Color-blind palette after display_color")

blsplotR::reset_par()
```

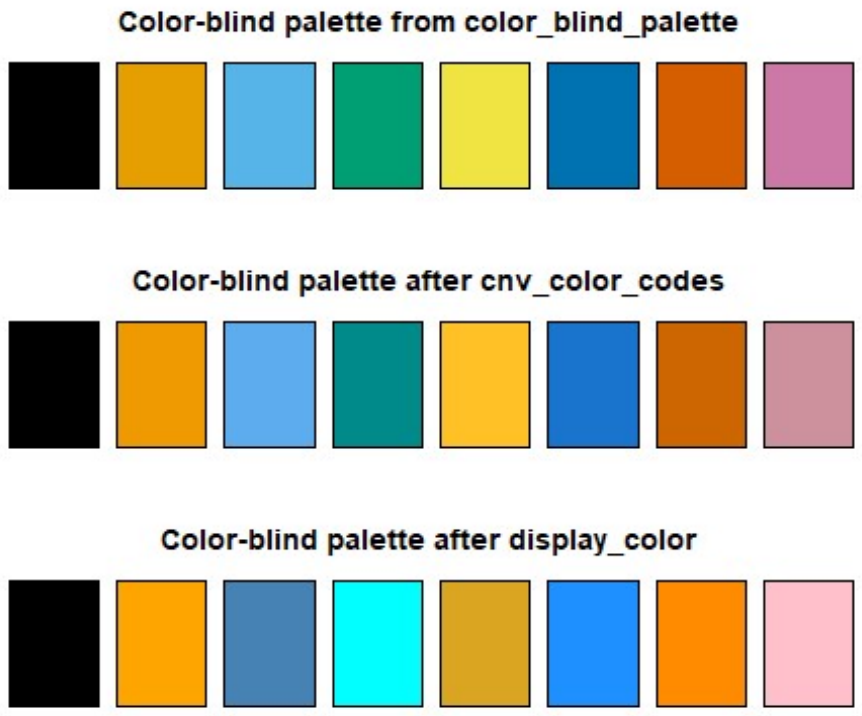

Figure 25: Compare `color_blind_palette`, `cnv_color_codes`, `display_color`

Here we are comparing the color blind colors produced by the function `color_blind_palette` and the way the functions `cnv_color_codes` and `display_colors` try to represent those colors with words for displays.

Figure 25 shows three rows of bar charts:

- the first bar chart is the color-blind palette generated by the function `color_blind_palette`;
- the second bar chart is how those hexadecimal codes are translated to words by the `cnv_color_codes` function;
- the third bar chart is how those colors would be displayed by the `display_colors` function if `thestrip_numbers = TRUE` is specified in the call. This argument simplifies the text by stripping any numeric text from the end of the color name.

While the names provided are more recognizable than the hexadecimal codes used in the original palette, there are differences in the colors produced by these codes. Comparing the first two rows, the colors compare reasonably well, except for the choice of `goldenrod1` for the `#F0E442` color code. Stripping the numbers from the end of those names cause even more differences from the original color.

5.4 from_rgb_to_hue

The function `from_rbg_to_hue` converts color codes formatted in RGB notations to a measure of the hue of that specific color, which refers to the dominant color family of the specific color specified. This measure is used in other representations of particular colors (HSL (hue-saturation-lightness), HSV (hue-saturation-value) or HSI (hue-saturation-intensity)).

This function appears in the post **How to sort HEX colors based on hue values in R** by BIOLINFO at https://www.biolinfo.com/sort-hex-colors-in-r/.

Figure 26 is an example of using hue values to sort a color vector. Using the `sample_shade` function, vectors with blue and red colors are generated, and the array `this_color` contains a mix of blue and red colors, illustrated by the first barchart. By generating the hue of the different colors (after converting them to RGB format) in the array `this_hue`, We can sort by hue and separate the different colors, as seen in the second plot.

Note that all of the “red” colors are not grouped together - the last three colors are some combination of violet and red, and are considered to be more violet than red, and are thus grouped together.

```r
par(mfrow = c(2,1), mar= c(1.0,1.0,3.0,1.0))
n_colors <- 6
# create index for blue colors
blue_length <- length(blsplotR::sample_shades("blue", NULL))
blue_index <- round(seq(blue_length/(2*n_colors), blue_length,
                        by=blue_length/n_colors))
# create blue color palatte. Since blue_index is used,
# colors will not be sampled.
blue_color <-
  blsplotR::sample_shades("blue", n_colors, blue_index)
# create index for red colors
red_length <- length(blsplotR::sample_shades("red", NULL))
red_index <- round(seq(red_length/(2*n_colors), red_length,
                       by=red_length/n_colors))
# create red color palatte. Since blue_index is used,
# colors will not be sampled.
red_color <-
  blsplotR::sample_shades("red", n_colors, red_index)
# createmix of blue and red color palattes.
this_color <-
  c(blue_color[1:3], red_color[1:3],
    blue_color[4:6], red_color[4:6])
barplot(rep(1, length(this_color)), axes = F,
    col = this_color, main = "Blue and Red")
# convert color code to RGB format
this_rgb <-
  sapply(this_color, function(x){col2rgb(x)})
# generate hues for RGB colors
this_hue <-
  apply(this_rgb, 2,
        function(x){blsplotR::from_rgb_to_hue(x)})
# generate colors sorted by hue
this_color_sorted_by_hue <- this_color[order(this_hue)]
barplot(rep(1, length(this_color_sorted_by_hue)), axes = F,
    col = this_color_sorted_by_hue, main = "Blue and Red")
blsplotR::reset_par()
```

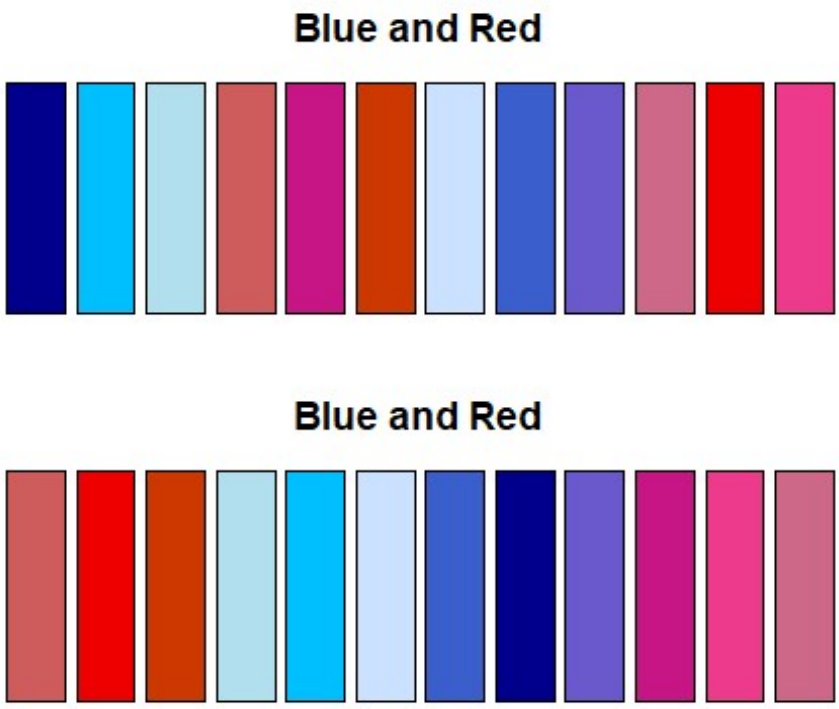

Figure 26: Example of `from_rgb_to_hue`

5.5 sample_shades

The function `sample_shades` searches the possible color codes to match a text string supplied by the user, and then samples from that list to create a color palette.

Users can specify a length for the color palette with the `n_colors` argument - if the user sets `n_colors = NULL`, the function will return every color that has a match for the color specified.

An set of indices can be supplied with the `this_index` argument that allows the function to return specific entries; this way, the function returns the same values rather than drawing a sample from the possible values.

Figure 27 shows how different options of the `sample_shades` function can be used. In this example, the first two calls to `sample_shades` are the same, but they do not generate the same color palette, as the function will sample from the available colors, and the sampling can change between calls of the function.

The third call of `sample_shades` returns all possible colors that match the color `blue`. This is used to create an index for the colors that are approximately equally spaced through the list of colors.

The next two calls of `sample_shades` have the same input, but since an index is specified by the user, the color palettes are identical. This allows for someone to be able to reproduce the same color palette for different calls of `sample_shades`.

```
par(mfrow = c(4,1), cex = 0.75, mar= c(1.0,1.0,3.0,1.0))

blue_color <-
  blsplotR::sample_shades("blue", n_colors = 12)
blue_color2 <-
  blsplotR::sample_shades("blue", n_colors = 12)

blue_all <- blsplotR::sample_shades("blue", n_colors = NULL)
blue_length <- length(blue_all)
blue_index <- round(seq(blue_length/(2*12), blue_length,
                        by=blue_length/12))
blue_color3 <-
  blsplotR::sample_shades("blue", n_colors = 12, blue_index)
blue_color4 <-
  blsplotR::sample_shades("blue", n_colors = 12, blue_index)

barplot(rep(1, 12), col = blue_color, axes = F,
        main = "Blue Shades 1")
barplot(rep(1, 12), col = blue_color2, axes = F,
        main = "Blue Shades 2")
barplot(rep(1, 12), col = blue_color3, axes = F,
        main = "Blue Shades 3")
barplot(rep(1, 12), col = blue_color4, axes = F,
        main = "Blue Shades 4")

blsplotR::reset_par()
```

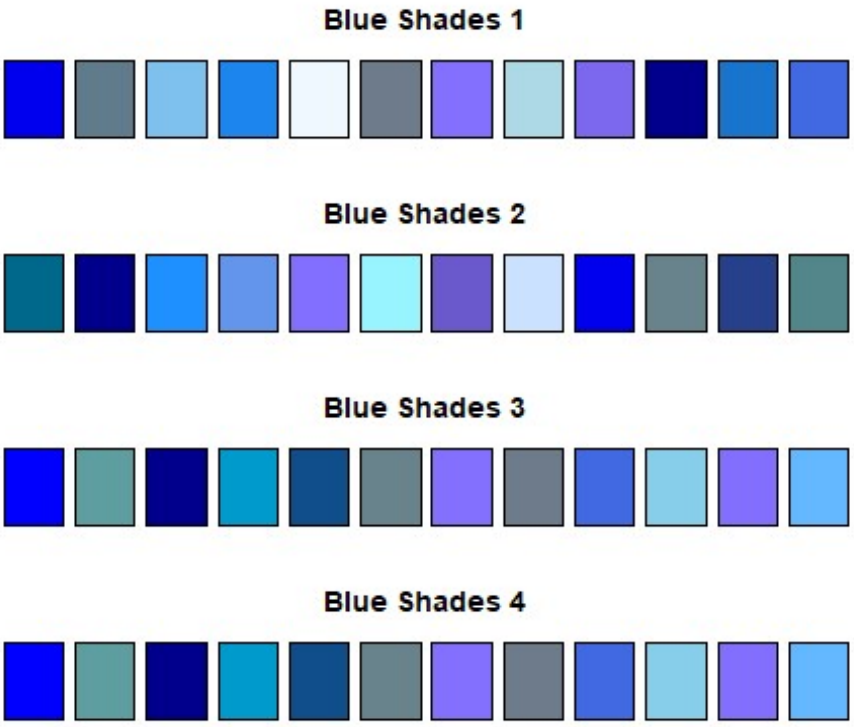

Figure 27: Example of `sample_shades`

5.6 sort_hex_by_hue

A function `sort_hex_by_hue` sorts a vector hexidecimal color colors by hue, which refers to the dominant color family of the specific color specified.

The code for this function first appeared in the post **How to sort HEX colors based on hue values in R** by BIOLINFO at https://www.biolinfo.com/sort-hex-colors-in-r/.

Figure 28 is an example of using hue values to sort a color vector. Using the `sample_shade` function, vectors with blue and red colors are generated, and the array `this_color` contains a mix of blue and red colors, illustrated by the first barchart. This is similar to the plots produced in Figure 26, except that once the colors are converted to RGB format, the sorting is done using the `sort_hex_by_hue` function.

```r
par(mfrow = c(2,1), mar= c(1.0,1.0,3.0,1.0))
n_colors <- 6
# create index for blue colors
blue_length <- length(blsplotR::sample_shades("blue", NULL))
blue_index <- round(seq(blue_length/(2*n_colors), blue_length,
                        by=blue_length/n_colors))
blue_color <- blsplotR::sample_shades("blue", n_colors, blue_index)

# create index for red colors
red_length <- length(blsplotR::sample_shades("red", NULL))
red_index <- round(seq(red_length/(2*n_colors), red_length,
                       by=red_length/n_colors))
red_color <- blsplotR::sample_shades("red", n_colors, red_index)
this_color2 <-
  c(red_color[1:3], blue_color[1:3], red_color[4:6], blue_color[4:6])
barplot(rep(1, length(this_color2)),
        col = this_color2,
        main = "Red and Blue",
        axes = F)

this_color2_hex <- gplots::col2hex(this_color2)
this_color2_sorted_by_hue <-
  blsplotR::sort_hex_by_hue(this_color2_hex)
barplot(rep(1, length(this_color2_sorted_by_hue)),
        col = this_color2_sorted_by_hue,
        main = "Red and Blue, Sorted by Hue",
        axes = F)
blsplotR::reset_par()
```

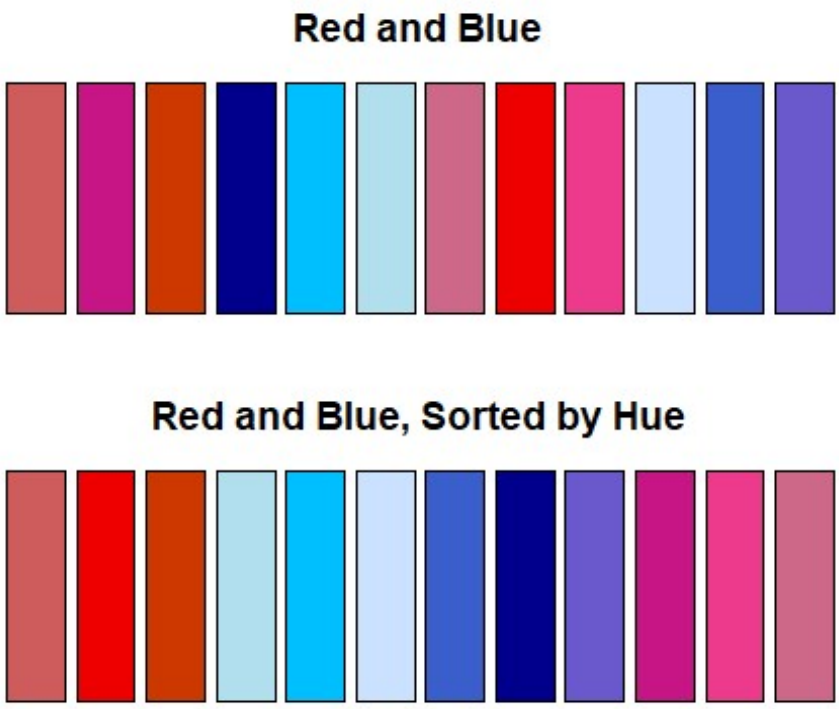

Figure 28: Example of `sort_hex_by_hue`

As before, all of the "red" colors are not grouped together - the last three colors are some combination of violet and red, and are considered to be more violet than red, and are thus grouped together.

5.7 wheel_invisible

The function `wheel_invisible` attempts to reproduce the values of the function `wheel` from the `colortools` package without producing the plot of the resulting color wheel. Since the `colortools` package is no longer available in CRAN, we needed to try and duplicate the functionality of the `wheel` function.

This function uses code from a couple of online posts:

- Code documented in the post **How to sort HEX colors based on hue values in R** by BIOLINFO at https://www.biolinfo.com/sort-hex-colors-in-r/, which shows how to sort hexidecimal color codes by hue using R.
- Additional code that sorts colors from light to dark from a response within the StackOverflow plot https://stackoverflow.com/questions/61193516/how-to-sort-colours-in-r.

Using the `wheel_invisible` function produces a vector of color codes based on a color provided by the user and, in the code block below, provides a color palette for the year-over-year plot produced for the Airline Passenger series (Figure 29).

```
this_wheel <- blsplotR::wheel_invisible("blue", 12)
blsplotR::plot_year_over_year(AirPassengers,
    main_title = "Airline Passenger Series (1949 - 1960)",
    this_col = this_wheel, this_right_mar = 5.75,
    this_legend_inset = -0.2)
```

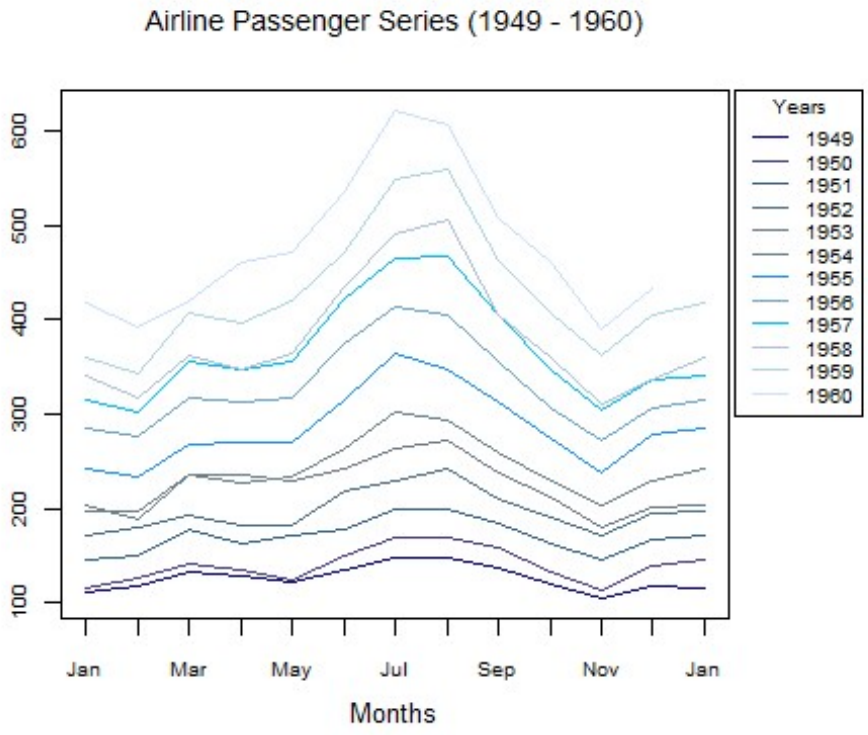

Figure 29: Example of `wheel_invisible`

Note that wheel not only returns a color palatte sorted by hue and brightness (light to dark), but it indexes the colors so that the hue values are equally spaced. This means you get the same set of colors every time you call `wheel_invisible`.

6 Recession related functions

In many of the plotting functions, you can specify that recession periods can be marked on the plotting surface. In this section, we'll go through the functions used to generate this on the plots.

6.1 convert_date_to_tis

The `convert_date_to_tis` function convert dates used for monthly (or quarterly) `ts` objects to a date object that can be used with `tis` objects.

The function used to generate the recession periods used by this code are stored as `tis` dates. Some time series objects in `R` are formed as time indexed series, which are

compatible with frequencies from the FAME database program. These time series objects are supported by the tis R package.

In the code block above, the current date for a monthly ts series is generated, and the function 'convert_date_to_tisuses this to generate the starting and ending date for the month astis` index values.

```
this_month <- as.numeric(substr(Sys.Date(),1,4)) +
    (as.numeric(substr(Sys.Date(),6,7)) - 1) / 12
end_this_month_tis <-
    blsplotR::convert_date_to_tis(this_month, this_freq = 12,
                                  is_start = FALSE, return_tis = TRUE)
start_this_month_tis <-
    blsplotR::convert_date_to_tis(this_month, this_freq = 12,
                                  is_start = TRUE, return_tis = TRUE)

c(start_this_month_tis, end_this_month_tis)
```

```
[1] 20230601 20230630
class: ti
```

6.2 draw_recession

The function draw_recession draws shaded areas in plots corresponding to NBER recessions. The recession dates are generated by the get_recession_dates function described later.

Arguments for this function allow users to set the color of shading done for the recession (this_col_recess), the density of the shading used for the recession periods (this_density), the border surrounding the recession periods (this_border), and whether there's a subtitle related to the recession (this_sub_recess).

```
air_seas <-
    seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                   x11='')
blsplotR::plot_table(air_seas, 'd12', 'AirPassengers',
    do_grid = TRUE, draw_recess = FALSE, use_ratio = FALSE,
    add_sub_title = TRUE, this_col = 'forestgreen')
blsplotR::draw_recession(this_col_recess = 'lightblue',
    this_density = 25, this_border = 1, this_sub_line = 2.25,
    this_sub_cex = 0.75)
```

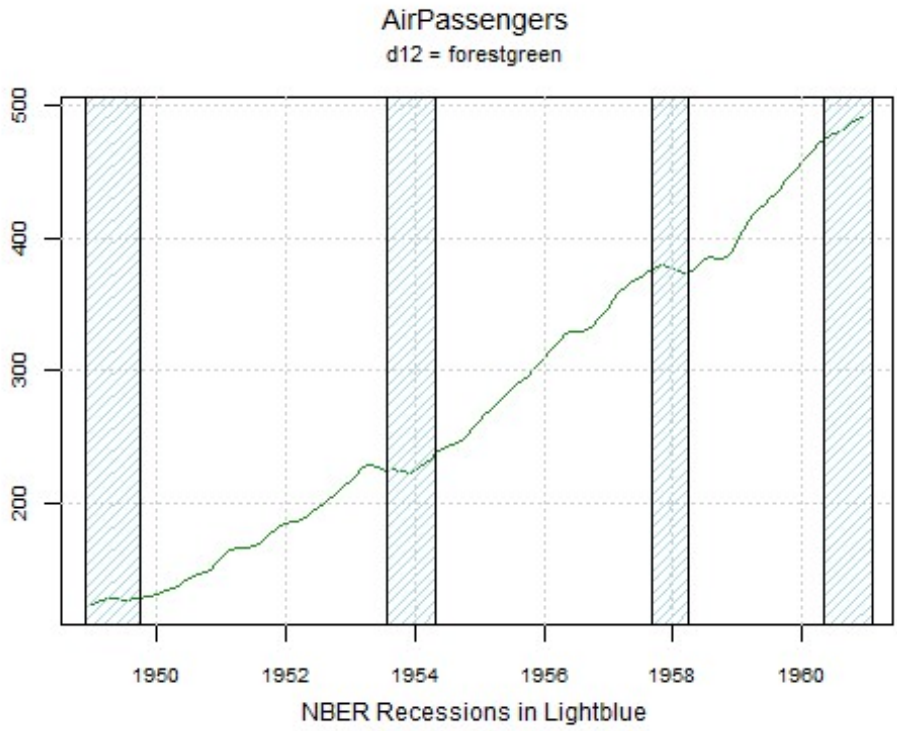

Figure 30: Example of `draw_recession`

This code chunk adds light blue regions for the NBER recessions to a plot of the X-11 trend generated by the `plot.table` function (Figure 30). The color of the regions are set to light blue (`this_col_recess = 'lightblue'`), and the shading for the region is set to be bit lighter than the default (`this_density = 25`; the default is 50). Compare this to Figure 10, which uses the default value.

The position and size of the subtitle related to the recession are also specified (`this_sub_line = 2.25, this_sub_cex = 0.75`).

6.3 get_recession_dates

The `get_recession_dates` function generates the starting and ending dates for NBER recessions between two monthly (or quarterly) dates for 'ts' time series objects. The function takes these dates, converts them to `tis` time indexes, uses the function nberDates from the tis package to get the time indexes for all NBER recessions, and returns just those that occur in the range specified.

There is also an option to add a starting date for a recession that hasn't ended yet (`this_add_recess_start`). This will allow users to include a region for a recession that started recently.

This code generates the NBER recessions for the time frame used for published CPS series. Note that the returned dates can be used with `ts` objects.

```
cps_years <- c(2003, 2023)
cps_recession <-
   blsplotR::get_recession_dates(start_recess = cps_years[1], end_recess =
cps_years[2])

cps_recession
```

```
     [,1]     [,2]
[1,] 2008 2009.417
[2,] 2020 2020.250
```

7 Spectrum related functions

There are a few special functions that are used with the `plot_double_spectrum` that flag spectral peak and determine if a given peak is significant using a visual criteria. The following two functions can be used to for this purpose, using output generated by the X-13ARIMA-SEATS program.

7.1 flag_peak

The `flag_peak` function determines the positions of visual significant peaks in the spectra generated by X-13ARIMA-SEATS. This function is called from the visual_sig_peaks function.

If visually significant peaks are found by the `flag_peak` function, a numeric vector of the position of the significant frequencies are returned. If no peaks are found, the function returns `0`.

This first block of code examines the spectrum of the original series for significant seasonal frequencies. The function returns the index for the first 5 seasonal frequencies - the series has multiple visually significant seasonal frequencies.

```
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
ori_flagged_peak_seas <- blsplotR::flag_peak(air_seas, 'ori', 's', 5)

ori_flagged_peak_seas
```

```
[1] 10 20 30 40 50
```

The next code block looks for significant trading day peaks in the seasonally adjusted series. The function returns the index for the second trading day frequency.

```
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
sa_flagged_peak_td <- blsplotR::flag_peak(air_seas, 'sa', 't', 2)

sa_flagged_peak_td
```

```
[1] 52
```

7.2 visual_sig_peaks

The function `visual_sig_peaks` has the same purpose as the function `flag_peak` - it determine the positions of visually significant peaks in spectra generated by X-13ARIMA-SEATS runs.

The user specifies the type of spectrum to be examined. The choices of spectrum includes:

- the original series adjusted for extreme values and outliers (`spec_type = 'ori'`);
- the seasonally adjusted series (`spec_type = 'sa'`);
- the irregular component (`spec_type = 'irr'`);
- the regARIMA residuals (`spec_type = 'rsd'`).

Users can specify which type of frequency to examine:

- seasonal (`spec_freq_code = 'seas'`)
- trading day (`spec_freq_code = 'td'`).

As with the `flag_peak` function, if visually significant peaks are found, a numeric vector of the position of the significant frequencies are returned. If no peaks are found, the function returns `0`.

This first block of code examines the spectrum of the seasonally adjusted series for significant seasonal frequencies. The function returns the index for the first seasonal frequencies.

```
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
ori_visual_sig_peaks_seas <-
    blsplotR::visual_sig_peaks(air_seas,'sa','seas')

ori_visual_sig_peaks_seas
```

```
[1] 10
```

The next code block looks for significant trading day peaks in the seasonally adjusted series. The function returns the index for the second trading day frequency.

```
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
sa_visual_sig_peaks_td <-
   blsplotR::visual_sig_peaks(air_seas, 'sa', 'td')

sa_visual_sig_peaks_td
```

```
[1] 52
```

The spectral plot below (Figure 31) shows the AR spectrum for the original series and the seasonally adjusted series, and you can see that there are significant peaks at the frequencies indicated by the code blocks for both the `visual_sig_peaks` and `flag_peaks` functions.

```
air_seas <-
  seasonal::seas(AirPassengers, arima.model = '(0 1 1)(0 1 1)',
                 x11='')
blsplotR::plot_double_spectrum(air_seas,
  this_col = c("blue", "forestgreen", "grey", "brown", "red", "orange"))
```

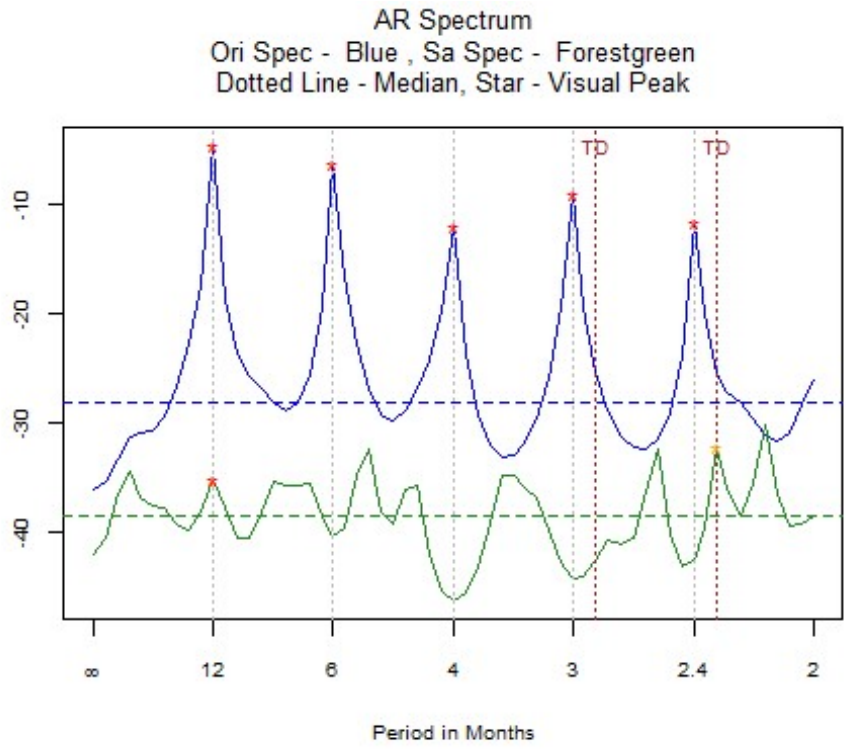

Figure 31: Example of `plot_double_spectrum` with an X-11 seasonal adjustment

Note that the seasonal adjustment used in this plot was from X-11; in an earlier plot (Figure 2), the spectrum of the SEATS adjustment for the same series shows not residual seasonality or trading day peaks.

8 Data

There are three data sets included in the `blsplotR` package. Details on each are given below.

8.1 employment_data_mts

The `employment_data_mts` object is an `mts` object (time series matrix object) containing the four main components of US Employment expressed as time series objects that end in December, 2022.

The matrix has four named columns with US Employment data:

- **n2000013**, Employed Males 16-19
- **n2000014**, Employed Females 16-19
- **n2000025**, Employed Males 20+
- **n2000026**, Employed Females 20+

Each column is a `ts` time series object - `employment_data_mts[,1]` is a time series object for employed males aged 16 to 19. This series can also be accessed using `employment_data_mts[,1]`.

8.2 employment_list

The `employment_list` object is a list object containing the four main components of US Employment expressed as time series objects that end in December, 2022. It is exactly the same data as in `employment_data_mts`, stored in a different type of object.

The list has four named elements with US Employment data:

- **n2000013**, Employed Males 16-19
- **n2000014**, Employed Females 16-19
- **n2000025**, Employed Males 20+
- **n2000026**, Employed Females 20+

Each element is a `ts` time series object - `employment_list$n2000013` is a time series object for employed males aged 16 to 19. This series can also be accessed using `employment_list[[2]]`.

8.3 xt_data_list

The xt_data_list object is a list object of US One Family Building Permits from the US Census Bureau for four regions of the United States. The elements of the list are expressed as time series objects that end in October, 2006

The list has four named elements with regional Building Permit data:

- **mw1u**, Midwest 1 family building permits
- **ne1u**, Northeast 1 family building permits
- **so1u**, South 1 family building permits
- **we1u**, West 1 family building permits

Each element is a `ts` time series object - `xt_data_list$ne1u` is a time series object for Northeast 1 family Building Permits. This series can also be accessed using `xt_data_list[[2]]`.