Package 'viscomplexr'

December 11, 2020

Type Package

Title Phase Portraits of Functions in the Complex Number Plane

Version 1.1.0

Date 2020-12-10

Description Functionality for creating phase portraits of functions in the complex number plane. Works with R base graphics, whose full functionality is available. Parallel processing is used for optimum performance.

Language en_US

License GPL-3

Encoding UTF-8

LazyData true

Imports doParallel (>= 1.0.15), grDevices, foreach, parallel, scales, plotrix, Rdpack, Rcpp

RdMacros Rdpack

Depends R (>= 4.0)

RoxygenNote 7.1.1

Suggests knitr, rmarkdown, Cairo, testthat (>= 2.1.0), pracma, covr

VignetteBuilder knitr

LinkingTo Rcpp

URL https://peterbiber.github.io/viscomplexr/, https://github.com/PeterBiber/viscomplexr/

NeedsCompilation yes

Author Peter Biber [aut, cre] (<https://orcid.org/0000-0002-9700-8708>)

Maintainer Peter Biber <castor.fiber@gmx.de>

Repository CRAN

Date/Publication 2020-12-11 05:20:03 UTC

R topics documented:

`blaschkeProd` *Calculate Blaschke products*

Description

This function calculates Blaschke products (https://en.wikipedia.org/wiki/Blaschke_product) for a complex number `z` given a sequence `a` of complex numbers inside the unit disk, which are the zeroes of the Blaschke product.

Usage

```
blaschkeProd(z, a)
```

Arguments

`z`	Complex number; the point in the complex plane to which the output of the function is mapped
`a`	Vector of complex numbers located inside the unit disk. At each `a`, the Blaschke product will have a zero.

Details

A sequence of points `a[n]` located inside the unit disk satisfies the Blaschke condition, if `sum[1:n](1 -abs(a[n])) < Inf`. For each element `a != 0` of such a sequence, `B(a,z) = abs(a)/a * (a -z)/(1 -conj(a) * z)` can be calculated. For `a = 0`, `B(a,z) = z`. The Blaschke product `B(z)` results as `B(z) = prod[1:n](B(a[n],z))`.

Value

The value of the Blaschke product at `z`.

See Also

Other maths: `jacobiTheta()`, `juliaNormal()`, `mandelbrot()`

Examples

```
# Generate random vector of 17 zeroes inside the unit disk
n <- 17
a <- complex(modulus = runif(n, 0, 1), argument = runif(n, 0, 2*pi))

# Portrait the Blaschke product
phasePortrait(blaschkeProd, moreArgs = list(a = a),
  xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
  nCores = 1) # Max. two cores on CRAN, not a limit for your use
```

---

jacobiTheta *Jacobi theta function*

Description

Approximation of "the" Jacobi theta function using the first nn factors in its triple product version

Usage

```
jacobiTheta(z, tau, nn = 30L)
```

Arguments

z	Complex number; the point in the complex plane to which the output of the function is mapped
tau	Complex number; the so-called half-period ratio, must have a positive imaginary part
nn	Integer; number of factors to be used when approximating the triple product (default = 30)

Details

This function approximates the Jacobi theta function theta(z; tau) which is the sum of exp(pi*i*n^2*tau + 2*pi*i*n*z) for n in -Inf, Inf. It uses, however, the function's triple product representation. See https://en.wikipedia.org/wiki/Theta_function for details. This function has been implemented in C++, but it is only slightly faster than well-crafted R versions, because the calculation can be nicely vectorized in R.

Value

The value of the function for z and tau.

See Also

Other maths: blaschkeProd(), juliaNormal(), mandelbrot()

Examples

```
phasePortrait(jacobiTheta, moreArgs = list(tau = 1i/2-1/4),
pType = "p", xlim = c(-2, 2), ylim = c(-2, 2),
nCores = 1) # Max. two cores on CRAN, not a limit for your use
```

```
phasePortrait(jacobiTheta, moreArgs = list(tau = 1i/2-1/2),
pType = "p", xlim = c(-2, 2), ylim = c(-2, 2),
nCores = 1)
```

```
phasePortrait(jacobiTheta, moreArgs = list(tau = 1i/3+1/3),
pType = "p", xlim = c(-2, 2), ylim = c(-2, 2),
nCores = 1)
```

```
phasePortrait(jacobiTheta, moreArgs = list(tau = 1i/4+1/2),
pType = "p", xlim = c(-2, 2), ylim = c(-2, 2),
nCores = 1)
```

`juliaNormal` *Julia iteration with a given number of steps*

Description

This function is designed as the basis for visualizing normal Julia sets with `phasePortrait`. In contrast to usual visualizations of Julia sets, this requires coloring the actual member points of the set and not the points outside. Therefore, for numbers that can be identified as not being parts of the Julia set, this function returns `NaN+NaNi`. All other numbers are mapped to the complex value obtained after a user-defined number of iterations. This function has been implemented in C++; therefore it is fairly fast.

Usage

```
juliaNormal(z, c, R_esc, itDepth = 500L)
```

Arguments

z	Complex number; the point in the complex plane to which the output of the function is mapped
c	Complex number; a parameter whose choice has an enormous effect on the shape of the Julia set. For obtaining useful results with phasePortrait, c must be an element of the Mandelbrot set.
R_esc	Real number; the espace radius. If the absolute value of a number obtained during iteration attains or excels the value of R_esc, juliaNormal will return NaN+NaNi. R_esc = 2 is a good choice for c being an element of the Mandelbrot set. See Details for more information.
itDepth	An integer which defines the depth of the iteration, i.e. the maximum number of iteration (default: itDepth = 500)

Details

Normal Julia sets are closely related to the Mandelbrot set. A normal Julia set comprises all complex numbers z for which the following sequence is bounded for all n > 0: a[n+1] = a[n]^2 + c, starting with a[0] = z. The parameter c is a complex number, and the sequence is certainly unbounded if abs(a[]) >= R with R being an escape Radius which matches the inequality R^2 -R >= abs(c). As the visualization with this package gives interesting pictures (i.e. other than a blank screen) only for c which are elements of the Mandelbrot set, R = 2 is a good choice. For the author's taste, the Julia visualizations become most interesting for c located in the border zone of the Mandelbrot set.

Value

Either NaN+NaNi or the complex number obtained after itDepth iterations

See Also

Other fractals: mandelbrot()

Other maths: blaschkeProd(), jacobiTheta(), mandelbrot()

Examples

```
# This code visualizes a Julia set with some appeal (for the author's
# taste). Zoom in as you like by adjusting xlim and ylim.

phasePortrait(juliaNormal,
  moreArgs = list(c = -0.09 - 0.649i, R_esc = 2),
  xlim = c(-2, 2),
  ylim = c(-1.3, 1.3),
  hsvNaN = c(0, 0, 0),
  nCores = 1)          # Max. two cores on CRAN, not a limit for your use
```

`mandelbrot` *Mandelbrot iteration with a given number of steps*

Description

This function is provided as a basis for visualizing the Mandelbrot set with `phasePortrait`. While usual visualizations color the points *outside* the Mandelbrot set dependent on the velocity of divergence, this function produces the information required for coloring the Mandelbrot set itself. For numbers that can be identified as not being elements of the Mandelbrot set, we obtain a `NaN+NaNi` value; for all other numbers, the function gives back the value after a user-defined number of iterations. The function has been implemented in C++; it runs fairly fast.

Usage

```
mandelbrot(z, itDepth = 500L)
```

Arguments

`z`	Complex number; the point in the complex plane to which the output of the function is mapped
`itDepth`	An integer which defines the depth of the iteration, i.e. the maximum number of iteration (default: `itDepth = 500`)

Details

The Mandelbrot set comprises all complex numbers `z` for which the sequence `a[n+1] = a[n]^2 + z` starting with `a[0] = 0` remains bounded for all `n > 0`. This condition is certainly not true, if, at any time, `abs(a[]) >= 2`. The function `mandelbrot` performs the iteration for `n = 0,...,itDepth -1` and permanently checks for `abs(a[n+1]) >= 2`. If this is the case, it stops the iteration and returns `NaN+NaNi`. In all other cases, it returns `a[itDepth]`.

Value

Either `NaN+NaNi` or the complex number obtained after `itDepth` iterations

See Also

Other fractals: `juliaNormal()`

Other maths: `blaschkeProd()`, `jacobiTheta()`, `juliaNormal()`

Examples

```
# This code shows the famous Mandelbrot figure in total, just in the
# opposite way as usual: the Mandelbrot set itself is colored, while the
# points outside are uniformly black.
# Adjust xlim and ylim to zoom in wherever you like.

phasePortrait(mandelbrot,
```

```
  xlim = c(-2.3, 0.7),
  ylim = c(-1.2, 1.2),
  hsvNaN = c(0, 0, 0),
  nCores = 1)          # Max. two cores on CRAN, not a limit for your use
```

`phasePortrait` *Create phase portraits of complex functions*

Description

`phasePortrait` makes phase portraits of functions in the complex number plane. It uses a technique often (but not quite correctly) called *domain coloring* (https://en.wikipedia.org/wiki/Domain_coloring). While many varieties of this technique exist, this book relates closely to the standards proposed by E. Wegert in his book *Visual Complex Functions* (Wegert 2012). In a nutshell, the argument (`Arg`) of any complex function value is displayed as a color from the chromatic circle. The fundamental colors red, green, and blue relate to the arguments (angles) of 0, 2/3pi, and 4/3pi (with smooth color transitions in between), respectively. Options for displaying the modulus (`Mod`) of the complex values and additional reference lines for the argument are available. This function is designed for being used inside the framework of R base graphics. It makes use of parallel computing, and depending on the desired resolution it may create extensive sets of large temporary files (see Details and Examples).

Usage

```
phasePortrait(
  FUN,
  moreArgs = NULL,
  xlim,
  ylim,
  invertFlip = FALSE,
  res = 150,
  blockSizePx = 2250000,
  tempDir = NULL,
  nCores = parallel::detectCores(),
  pType = "pma",
  pi2Div = 9,
  logBase = exp(2 * pi/pi2Div),
  argOffset = 0,
  darkestShade = 0.1,
  lambda = 7,
  gamma = 0.9,
  stdSaturation = 0.8,
  hsvNaN = c(0, 0, 0.5),
  asp = 1,
```

```
  deleteTempFiles = TRUE,
  noScreenDevice = FALSE,
  autoDereg = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

FUN	The function to be visualized. There are two possibilities to provide it, a quoted character string, or a function object.

Quoted character string A function can be provided as a quoted character string containing an expression R can interpret as a function of a complex number z. Examples: "sin(z)", "(z^2 - 1i)/(tan(z))", "1/4*z^2 - 10*z/(z^4+4)". Names of functions known in your R session can be used in a standalone way, without mentioning z, e.g. "sin", "tan", "sqrt". Obviously, this also works for functions you defined yourself, e.g. "myIncredibleFunction" would be valid if you coded a function with this name before. This is especially useful for functions which require additional parameters beside the complex number they are supposed to calculate with. Such arguments can be provided via the parameter `moreArgs`. One-liner expressions provided as strings are also compatible with `moreArgs` (see examples).
While it is not the way we recommend for most purposes, you can even define more complicated functions of your own as character strings. In this case, you need to use `vapply` as a wrapper for your actual function (see Details, and Examples). Such constructions allow to provide additional input variables as a part of the character string by using the `vapply`-mechanism (see Details and Examples). The helper function `vector2String`) can be useful for that matter. However, the parameter `moreArgs` is not applicable in this context. Probably, the most useful application of the function-as-string concept is when the user defined function, possibly including values for additional arguments, is to be pasted together at runtime.

Function object It is also possible to directly provide function objects to `FUN`. This can be any function known to R in the current session. Simply put, for functions like sin, tan, cos, and sqrt you do not even have to quote their names when passing them to `phasePortrait`. Same applies to functions you defined yourself. It is also possible to hand over an anonymous function to `FUN` when calling `phasePortrait`. In all these cases, the parameter `moreArgs` can be used for providing additional arguments to `FUN`. In general, providing a function as an object, and using `moreArgs` in case additional arguments are required, is what we recommend for user-defined functions.

When executing `phasePortrait`, `FUN` is first evaluated with `match.fun`. If this is not successful, an attempt to interpret `FUN` as an expression will be made. If this fails, `phasePortrait` terminates with an error.

moreArgs	A named list of other arguments to FUN. The names must match the names of the arguments in FUN's definition.

`xlim` The x limits (x1, x2) of the plot as a two-element numeric vector. Follows exactly the same definition as in `plot.default`. Here, `xlim` has to be interpreted as the plot limits on the real axis.

`ylim` The y limits of the plot (y1, y2) to be used in the same way as `xlim`. Evidently, `ylim` indicates the plot limits on the imaginary axis.

`invertFlip` If `TRUE`, the function is mapped over a z plane, which has been transformed to `1/z * exp(1i*pi)`. This is the projection required to plot the north Riemann hemisphere in the way proposed by Wegert (2012), p. 41. Defaults to `FALSE`. If this option is chosen, the numbers at the axis ticks have another meaning than in the normal case. Along the real axis, they represent the real part of `1/z`, and along the imaginary axis, they represent the imaginary part of `1/z`. Thus, if you want annotation, you should choose appropriate axis labels like `xlab = Re(1/z)`, and `ylab = Im(1/z)`.

`res` Desired resolution of the plot in dots per inch (dpi). Default is 150 dpi. All other things being equal, `res` has a strong influence on computing times (double `res` means fourfold number of pixels to compute). A good approach could be to make a plot with low resolution (e.g. the default 150 dpi) first, adjust whatever required, and plot into a graphics file with high resolution after that.

`blockSizePx` Number of pixels and corresponding complex values to be processed at the same time (see Details). Default is 2250000. This value gave good performance on older systems as well as on a high-end gaming machine, but some tweaking for your individual system might even improve things.

`tempDir` `NULL` or a character string, specifying the name of the directory where the temporary files written by `phasePortrait` are stored. Default is `NULL`, which makes `phasePortrait` use the current R session's temporary directory. Note that if you specify another directory, it will be created if it does not exist already. Even though the temporary files are deleted after completing a phase portrait (unless the user specifies `deleteTempFiles = FALSE`, see below), the directory will remain alive even if has been created by `phasePortrait`.

`nCores` Number of processor cores to be used in the parallel computing tasks. Defaults to the maximum number of cores available. Any number between 1 (serial computation) and the maximum number of cores available as indicated by `parallel::detectCores()` is accepted.

`pType` One of the four options for plotting, "p", "pa", "pm", and "pma" as a character string. Defaults to "pma". Option "p" produces a mere phase plot, i.e. contains only colors for the complex numbers' arguments, but no reference lines at all. the option "pa" introduces shading zones that emphasize the arguments. These zones each cover an angle defined by `2*pi/pi2Div`, where p2Div is another parameter of this function (see there). These zones are shaded darkest at the lowest angle (counter clockwise). Option "pm" displays the modulus by indicating zones, where the moduli at the higher edge of each zone are in a constant ratio with the moduli at the lower edge of the zone. Default is a ratio of almost exactly 2 (see parameter `logBase`) for details. At the lower edge, color saturation is lowest and highest at the higher edge (see parameters `darkestShade`, and `stdSaturation`). Option "pma" (default) includes both shading schemes.

`pi2Div` Angle distance for the argument reference zones added for `pType = "pma"` or `pType = "pa"`. The value has to be given as an integer (reasonably) fraction of

2*pi (i.e. 360 degrees). The default is 9; thus, reference zones are delineated by default in distances of 2*pi/9, i.e. (40 degrees), starting with 0, i.e. the color red if not defined otherwise with the parameter `argOffset`. In contrast to the borders delimiting the modulus zones, the borders of the reference zones for the argument always follow the same color (by definition).

`logBase` Modulus ratio between the edges of the modulus reference zones in `pType` `"pm"` and `"pma"`. As recommended by Wegert (2012), the default setting is `logBase = exp(2*pi/pi2Div)`. This relation between the parameters `logBase` and `pi2Div` ensures an analogue scaling of the modulus and argument reference zones (see Details). Conveniently, for the default `pi2Div = 9`, we obtain `logBase == 2.0099...`, which is very close to 2. Thus, the modulus at the higher edge of a given zone is almost exactly two times the value at the lower edge.

`argOffset` The (complex number) argument in radians counterclockwise, at which the argument reference zones are fixed. Default is 0, i.e. all argument reference zones align to the center of the red area.

`darkestShade` Darkest possible shading of modulus and angle reference zones for `pType` `"pm"` and `"pma"`. It corresponds to the value "v" in the hsv color model. `darkestShade = 0` means no brightness at all, i.e. black, while `darkestShade = 1` indicates maximum brightness. Defaults to 0.1, i.e. very dark, but hue still discernible.

`lambda` Parameter steering the shading interpolation between the higher and the lower edges of the the modulus and argument reference zones in `pType` `"pm"` and `"pm"`. Should be > 0, default and reference is `lambda = 7`. Values < 7 increase the contrast at the zone borders, values > 7 weaken the contrast.

`gamma` Parameter for adjusting the combined shading of modulus and argument reference zones in `pType` `"pma"`. Should be in the interval `[0,1]`. Default is 0.9. The higher the value, the more the smaller of both shading values will dominate the outcome and vice versa.

`stdSaturation` Saturation value for unshaded hues which applies to the whole plot in `pType` `"p"` and to the (almost) unshaded zones in `pType` `"pm"` and `"p"`. This corresponds to the value "s" in the hsv color model. Must be between 0 and 1, where 1 indicates full saturation and 0 indicates a neutral grey. Defaults to 0.8.

`hsvNaN` hsv coded color for being used in areas where the function to be plotted is not defined. Must be given as a numeric vector with containing the values h, s, and v in this order. Defaults to `c(0,0,0.5)` which is a neutral grey.

`asp` Aspect ratio y/x as defined in `plot.window`. Default is 1, ensuring an accurate representation of distances between points on the screen.

`deleteTempFiles`

If TRUE (default), all temporary files are deleted after the plot is completed. Set it on FALSE only, if you know exactly what you are doing - the temporary files can occupy large amounts of hard disk space (see details).

`noScreenDevice` Suppresses any graphical output if TRUE. This is only intended for test purposes and makes probably only sense together with `deleteTempFiles == FALSE`. For dimensioning purposes, `phasePortrait` will use a 1 x 1 inch pseudo graphics device in this case. The default for this parameter is `FALSE`, and you should change it only if you really know what you are doing.

`autoDereg` if TRUE, automatically register sequential backend after the phase portrait is completed. Default is FALSE, because registering a parallel backend can be time consuming. Thus, if you want make several phase portraits in succession, you should set `autoDereg` only for the last one, or simply type `foreach::registerDoSEQ` after you are done. In any case, you don't want to have an unused parallel backend lying about.

`verbose` if TRUE (default), `phasePortrait` will continuously write progress messages to the console. This is convenient for normal purposes, as calculating larger phase portraits in higher resolution may take several minutes. The setting `verbose = FALSE`, will suppress any output to the console.

`...` All parameters accepted by the `plot.default` function.

Details

This function is intended to be used inside the framework of R base graphics. It plots into the active open graphics device where it will display the phase plot of a user defined function as a raster image. If no graphics device is open when called, the function will plot into the default graphics device. This principle allows to utilize the full functionality of R base graphics. All graphics parameters (`par`) can be freely set and the function `phasePortrait` accepts all parameters that can be passed to the `plot.default` function. This allows all kinds of plots - from scientific representations with annotated axes and auxiliary lines, notation, etc. to poster-like artistic pictures.

Mode of operation After being called, `phasePortrait` gets the size in inch of the plot region of the graphics device it is plotting into. With the parameter `res` which is the desired plot resolution in dpi, the horizontal and vertical number of pixels is known. As `xlim` and `ylim` are provided by the user, each pixel can be attributed a complex number z from the complex plane. In that way a two-dimensional array is built, where each cell represents a point of the complex plane, containing the corresponding complex number z. This array is set up in horizontal strips (i.e. split along the imaginary axis), each strip containing approximately `blockSizePx` pixels. In a parallel computing loop, the strips are constructed, saved as temporary files and immediately deleted from the RAM in order to avoid memory overflow. After that, the strips are sequentially loaded and subdivided into a number of chunks that corresponds to the number of registered parallel workers (parameter `nCores`). By parallely processing each chunk, the function `f(z)` defined by the user in the argument `FUN` is applied to each cell of the strip. This results in an array of function values that has exactly the same size as the original strip. The new array is saved as a temporary file, the RAM is cleared, and the next strip is loaded. This continues until all strips are processed. In a similar way, all strips containing the function values are loaded sequentially, and in a parallel process the complex values are translated into colors which are stored in a raster object. While the strips are deleted from the RAM after processing, the color values obtained from each new strip are appended to the color raster. After all strips are processed, the raster is plotted into the plot region of the graphics device. If not explicitly defined otherwise by the user, all temporary files are deleted after that.

Temporary file system By default, the above-mentioned temporary files are deleted after use. This will not happen, if the parameter `deleteTempFiles` is set to `FALSE` or if `phasePortrait` does not terminate properly. In both cases, you will find the files in the directory specified with the parameter `tempDir`. These files are `.RData` files, each one contains a two-dimensional array of complex numbers. The file names follow a strict convention, see the following examples:

```
0001zmat2238046385.RData
0001wmat2238046385.RData
```

Both names begin with '0001', indicating that the array's top line is the first line of the whole clipping of the complex number plane where the phase portrait relates to. The array which follows below can e.g. begin with a number like '0470', indicating that its first line is line number 470 of the whole clipping. The number of digits for these line numbers is not fixed. It is determined by the greatest number required. Numbers with less digits are zero-padded. The second part of the file name is either `zmat` or `wmat`. The former indicates an array whose cells contain untransformed numbers of the complex number plane. The latter contains the values obtained from applying the function of interest to the first array. Thus, cells at the same position in both arrays exactly relate to each other. The third part of the file names is a ten-digit integer. This is a random number which all temporary files stemming from the same call of `phasePortrait` have in common. This guarantees that no temporary files will be confounded by the function, even if undeleted temporary files from previous runs are still present.

HSV color model For color-coding the argument of a complex number, `phasePortrait` uses the `hsv` (hue, saturation, value) color model. Hereby, the argument is mapped to a position on the chromatic circle, where the fundamental colors red, green, and blue relate to the arguments (angles) of 0, 2/3pi, and 4/3pi, respectively. This affects only the hue component of the color model. The value component is used for shading modulus and/or argument zones. The saturation component for all colors can be defined with the parameter `stdSaturation`.

Zone definitions and shading In addition to displaying colors for the arguments of complex numbers, zones for the modulus and/or the argument are shaded for `pType` other than "p". The modulus zones are defined in a way that each zone covers moduli whose logarithms to the base `logBase` have the same integer part. Thus, from the lower edge of one modulus zone to its upper edge, the modulus multiplies with the value of `logBase`. The shading of a modulus zone depends on the fractional parts `x` of the above-mentioned logarithms, which cover the interval `[0,1[`. This translates into the value component `v` of the `hsv` color model as follows:

```
v = darkestShade + (1 -darkestShade) * x^(1/lambda)
```

where `darkestShade` and `lambda` are parameters that can be defined by the user. Modifying the parameters `lambda` and `darkestShade` is useful for adjusting contrasts in the phase portraits. The argument zone definition is somewhat simpler: Each zone covers an angle domain of `2*pi / pi2Div`, the "zero reference" for all zones being `argOffset`. The angle domain of one zone is linearly mapped to a value `x` from the range `[0,1[`. The value component of the color to be displayed is calculated as a function of `x` with the same equation as shown above. In case the user has chosen `pType = "pma"`, x-values `xMod` and `xArg` are calculated separately for the modulus and the argument, respectively. They are transformed into preliminary v-values as follows:

`vMod = xMod^(1/lambda)` and vArg = xArg^(1/lambda)

From these, the final v value results as

```
v = darkestShade + (1-darkestShade) * (gamma * vMod * vArg + (1-gamma) * (1 -(1-vMod)
* (1-vArg)))
```

The parameter gamma (range [0,1]) determines they way how vMod and vArg are combined. The closer gamma is to one, the more the smaller of both values will dominate the outcome and vice versa.

Defining more complicated functions as strings with vapply You might want to write and use functions which require more code than a single statement like (z-3)^2+1i*z. As mentioned in the description of the parameter FUN, we recommend to define such functions as separate objects and hand them over as such. There might be, however, cases, where it is more convenient, to define a function as a single long string, and pass this string to FUN. In order to make this work, vapply should be be used for wrapping the actual code of the function. This is probably not the use of vapply intended by its developers, but it works nicely and performs well. The character string has to have the following structure "vapply(z, function(z, *other arguments if required*) {*define function code in here*}, *define other arguments here*, FUN.VALUE = complex(1))". See examples.

References

Wegert E (2012). *Visual Complex Functions. An Introduction with Phase Portraits*. Springer, Basel Heidelberg New York Dordrecht London. ISBN 978-3-0348-0179-9.

Examples

```
# Map the complex plane on itself

# x11(width = 8, height = 8)   # Screen device commented out
                               # due to CRAN test requirements.
                               # Use it when trying this example
phasePortrait("z", xlim = c(-2, 2), ylim = c(-2, 2),
              xlab = "real", ylab = "imaginary",
              verbose = FALSE, # Suppress progress messages
              nCores = 2)      # Max. two cores allowed on CRAN
                               # not a limit for your own use



# A rational function

# x11(width = 10, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
phasePortrait("(2-z)^2*(-1i+z)^3*(4-3i-z)/((2+2i+z)^4)",
              xlim = c(-8, 8), ylim = c(-6.3, 4.3),
              xlab = "real", ylab = "imaginary",
              nCores = 2)     # Max. two cores allowed on CRAN
                              # not a limit for your own use




# Different pType options by example of the sine function.
# Note the different equivalent definitions of the sine
```

```
# function in the calls to phasePortrait

# x11(width = 9, height = 9) # Screen device commented out
                             # due to CRAN test requirements.
                             # Use it when trying this example
op <- par(mfrow = c(2, 2), mar = c(2.1, 2.1, 2.1, 2.1))
phasePortrait("sin(z)", xlim = c(-pi, pi), ylim = c(-pi, pi),
              pType = "p",   main = "pType = 'p'",   axes = FALSE,
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

phasePortrait("sin(z)", xlim = c(-pi, pi), ylim = c(-pi, pi),
              pType = "pm",  main = "pType = 'pm'",  axes = FALSE,
              nCores = 2)

phasePortrait("sin",    xlim = c(-pi, pi), ylim = c(-pi, pi),
              pType = "pa",  main = "pType = 'pa'",  axes = FALSE,
              nCores = 2)

phasePortrait(sin,      xlim = c(-pi, pi), ylim = c(-pi, pi),
              pType = "pma", main = "pType = 'pma'", axes = FALSE,
              nCores = 2)

par(op)


# I called this one 'nuclear fusion'

# x11(width = 16/9*8, height = 8) # Screen device commented out
                                  # due to CRAN test requirements.
                                  # Use it when trying this example

op <- par(mar = c(0, 0, 0, 0), omi = c(0.2, 0.2, 0.2, 0.2), bg = "black")
phasePortrait("cos((z + 1/z)/(1i/2 * (z-1)^10))",
              xlim = 16/9*c(-2, 2), ylim = c(-2, 2),
              axes = FALSE, xaxs = "i", yaxs = "i",
              nCores = 2) # Max. two cores allowed on CRAN
                          # not a limit for your own use

par(op)


# Passing function objects to phasePortrait:
# Two mathematical celebrities - Riemann's zeta function
# and the gamma function, both from the package pracma.
# R's built-in gamma is not useful, as it does not work
# with complex input values.

if(requireNamespace("pracma", quietly = TRUE)) {
# x11(width = 16, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
op <- par(mfrow = c(1, 2))
phasePortrait(pracma::zeta,  xlim = c(-35, 15), ylim = c(-25, 25),
```

```
              xlab = "real", ylab = "imaginary",
              main = expression(zeta(z)), cex.main = 2,
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

phasePortrait(pracma::gammaz, xlim = c(-10, 10), ylim = c(-10, 10),
              xlab = "real", ylab = "imaginary",
              main = expression(Gamma(z)), cex.main = 2,
              nCores = 2) # Max. two cores allowed on CRAN
                          # not a limit for your own use

}



# Using vapply for defining a whole function as a string.
# This is a Blaschke product with a sequence a of twenty numbers.
# See the documentation of the function vector2String for a more
# convenient space-saving definition of a.
# But note that a C++ version of the Blaschke product is available
# in this package (function blaschkeProd()).

# x11(width = 10, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
phasePortrait("vapply(z, function(z, a) {
                fct <- ifelse(abs(a) != 0,
                  abs(a)/a * (a-z)/(1-Conj(a)*z), z)
                return(prod(fct))
              },
              a = c(0.12152611+0.06171533i,  0.53730315+0.32797530i,
                    0.35269601-0.53259644i, -0.57862039+0.33328986i,
                   -0.94623221+0.06869166i, -0.02392968-0.21993132i,
                    0.04060671+0.05644165i,  0.15534449-0.14559097i,
                    0.32884452-0.19524764i,  0.58631745+0.05218419i,
                    0.02562213+0.36822933i, -0.80418478+0.58621875i,
                   -0.15296208-0.94175193i, -0.02942663+0.38039250i,
                   -0.35184130-0.24438324i, -0.09048155+0.18131963i,
                    0.63791697+0.47284679i,  0.25651928-0.46341192i,
                    0.04353117-0.73472528i, -0.04606189+0.76068461i),
              FUN.VALUE = complex(1))",
              pType = "p",
              xlim = c(-4, 2), ylim = c(-2, 2),
              xlab = "real", ylab = "imaginary",
              nCores = 2) # Max. two cores allowed on CRAN
                          # not a limit for your own use



# Much more elegant: Define the function outside.
# Here comes a Blaschke product with 200 random points.

# define function for calculating blaschke products, even
```

```
# possible as a one-liner
blaschke <- function(z, a) {
  return(prod(ifelse(abs(a) != 0, abs(a)/a * (a-z)/(1-Conj(a)*z), z)))
}
# define 200 random numbers inside the unit circle
n <- 200
a <- complex(modulus = runif(n), argument = runif(n)*2*pi)
# Plot it
# x11(width = 10, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
phasePortrait(blaschke,
  moreArgs = list(a = a),
  pType = "p",
  xlim = c(-2.5, 2.5), ylim = c(-1.7, 1.7),
  xlab = "real", ylab = "imaginary",
  nCores = 2) # Max. two cores allowed on CRAN
              # not a limit for your own use




# A hybrid solution: A one-liner expression given as a character string
# can be provided additional arguments with moreArgs

n <- 73
a <- complex(modulus = runif(n), argument = runif(n)*2*pi)
# x11(width = 10, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
phasePortrait("prod(ifelse(abs(a) != 0,
  abs(a)/a * (a-z)/(1-Conj(a)*z), z))",
  moreArgs = list(a = a),
  pType = "p",
  xlim = c(-2.5, 2.5), ylim = c(-1.7, 1.7),
  xlab = "real", ylab = "imaginary",
  nCores = 1) # Max. two cores allowed on CRAN
              # not a limit for your own use




# Note the difference in performance when using the C++ defined
# function blaschkeProd() provided in this package

n <- 73
a <- complex(modulus = runif(n), argument = runif(n)*2*pi)
# Plot it
# x11(width = 10, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
phasePortrait(blaschkeProd,
  moreArgs = list(a = a),
```

```
  pType = "p",
  xlim = c(-2.5, 2.5), ylim = c(-1.7, 1.7),
  xlab = "real", ylab = "imaginary",
  nCores = 1) # Max. two cores allowed on CRAN
              # not a limit for your own use



# Interesting reunion with Benoit Mandelbrot.
# The function mandelbrot() is part of this package (defined
# in C++ for performance)

# x11(width = 11.7, height = 9/16*11.7) # Screen device commented out
                                        # due to CRAN test requirements.
                                        # Use it when trying this example
op <- par(mar = c(0, 0, 0, 0), bg = "black")
phasePortrait(mandelbrot,
              moreArgs = list(itDepth = 100),
              xlim = c(-0.847, -0.403), ylim = c(0.25, 0.50),
              axes = TRUE, pType = "pma",
              hsvNaN = c(0, 0, 0), xaxs = "i", yaxs = "i",
              nCores = 1) # Max. two cores allowed on CRAN
                          # not a limit for your own use
par(op)



# Here comes a Julia set.
# The function juliaNormal() is part of this package (defined
# in C++ for performance)

# x11(width = 11.7, height = 9/16*11.7) # Screen device commented out
                                        # due to CRAN test requirements.
                                        # Use it when trying this example
op <- par(mar = c(0, 0, 0, 0), bg = "black")
phasePortrait(juliaNormal,
  moreArgs = list(c = -0.09 - 0.649i, R_esc = 2),
  xlim = c(-2, 2),
  ylim = c(-1.3, 1.3),
  hsvNaN = c(0, 0, 0),
  nCores = 1) # Max. two cores allowed on CRAN
              # not a limit for your own use
par(op)
```

---

phasePortraitBw *Create two-color phase portraits of complex functions*

Description

phasePortraitBw allows for creating two-color phase portraits of complex functions based on a polar chessboard grid (cf. Wegert (2012), p. 35). Compared to the full phase portraits that can be made with phasePortrait, two-color portraits omit information. Especially in combination with full phase portraits they can be, however, very helpful tools for interpretation. Besides, two-color phase portraits have a special aesthetic appeal which is worth exploring for itself. In its parameters and its mode of operation, phasePortraitBw is very similar to phasePortrait.

Usage

```
phasePortraitBw(
  FUN,
  moreArgs = NULL,
  xlim,
  ylim,
  invertFlip = FALSE,
  res = 150,
  blockSizePx = 2250000,
  tempDir = NULL,
  nCores = parallel::detectCores(),
  bwType = "ma",
  pi2Div = 18,
  logBase = exp(2 * pi/pi2Div),
  argOffset = 0,
  bwCols = c("black", "gray95", "gray"),
  asp = 1,
  deleteTempFiles = TRUE,
  noScreenDevice = FALSE,
  autoDereg = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

FUN	The function to be visualized. There are two possibilities to provide it, a quoted character string, or a function object. The quoted character string must contain an expression that can be interpreted by R as a function of a complex number z (like e.g. "sin(z)", "(z^2 - 1i)/(tan(z))", "1/4*z^2 - 10*z/(z^4+4)"). See the documentation of phasePortrait for a complete presentation of all options.
moreArgs	A named list of other arguments to FUN. The names must match the names of the arguments in FUN's definition.
xlim	The x limits (x1, x2) of the plot as a two-element numeric vector. Follows exactly the same definition as in plot.default. Here, xlim has to be interpreted as the plot limits on the real axis.
ylim	The y limits of the plot (y1, y2) to be used in the same way as xlim. Evidently, ylim indicates the plot limits on the imaginary axis.

`invertFlip` If `TRUE`, the function is mapped over a z plane, which has been transformed to `1/z * exp(1i*pi)`. This is the projection required to plot the north Riemann hemisphere in the way proposed by Wegert (2012), p. 41. Defaults to `FALSE`. If this option is chosen, the numbers at the axis ticks have another meaning than in the normal case. Along the real axis, they represent the real part of `1/z`, and along the imaginary axis, they represent the imaginary part of `1/z`. Thus, if you want annotation, you should choose appropriate axis labels like `xlab = Re(1/z)`, and `ylab = Im(1/z)`.

`res` Desired resolution of the plot in dots per inch (dpi). Default is 150 dpi. All other things being equal, `res` has a strong influence on computing times (double `res` means fourfold number of pixels to compute). A good approach could be to make a plot with low resolution (e.g. the default 150 dpi) first, adjust whatever required, and plot into a graphics file with high resolution after that.

`blockSizePx` Number of pixels and corresponding complex values to be processed at the same time (see Details). Default is 2250000. This value gave good performance on older systems as well as on a high-end gaming machine, but some tweaking for your individual system might even improve things.

`tempDir` `NULL` or a character string, specifying the name of the directory where the temporary files written by `phasePortrait` are stored. Default is `NULL`, which makes `phasePortrait` use the current R session's temporary directory. Note that if you specify another directory, it will be created if it does not exist already. Even though the temporary files are deleted after completing a phase portrait (unless the user specifies `deleteTempFiles = FALSE`, see below), the directory will remain alive even if has been created by `phasePortrait`.

`nCores` Number of processor cores to be used in the parallel computing tasks. Defaults to the maximum number of cores available. Any number between 1 (serial computation) and the maximum number of cores available as indicated by `parallel::detectCores()` is accepted.

`bwType` One of the three options for plotting, "m", "a", and "ma", to be provided as a character string. Defaults to "ma". This parameter has a comparable role to the parameter `pType` in `phasePortrait`. Option "m" produces a plot that colors modulus zones only. In more detail, for each input number's modulus, the logarithm with base `logBase` (see below) is calculated and cut down to the next lower integer value. If this is an even number, the first color given in `bwCols` (see below) is taken. In case of an odd number, the second color is used. Option "a" produces a plot that exclusively colors argument (phase angle) zones. To that end, the full angle (2*pi) is divided into `p2Div` (see below) zones, which are numbered from 0 to pi2Div - 1 with increasing angle. Such an integer number is attributed to the complex number of interest according to the zone it falls into. Even and odd zone numbers are mapped to the first and the second color in `bwCols`, respectively. For normal purposes, the input parameter `pi2Div` should be an even number in order to avoid the first and the last zone having the same color. With option "ma", a chessboard-like alternation of colors is displayed over the tiles formed by the intersecting modulus and argument zones (both determined separately as with the options "m" and "a").

`pi2Div` Angle distance for the argument reference zones added for `pType = "pma"` or `pType = "pa"`. The value has to be given as an integer (reasonably) fraction of

2*pi (i.e. 360 degrees). Unlike with phasePortrait, the default is 18; thus, reference zones are delineated by default in distances of 2*pi/18, i.e. (20 degrees), starting with 0 if not defined otherwise with the parameter argOffset. While the default of pi2Div is 9 with phasePortrait for good reasons (see there), setting pi2Div to an odd number is usually not a good choice with two-color phase portraits, because the first and the last phase angle zone would get the same color. However, as pi2Div here defaults to double the value as with phasePortrait, both plot types can be nicely compared even when using their specific defaults of pi2Div.

logBase — Modulus ratio between the edges of the modulus zones in bwType "m" and "ma". As recommended by Wegert (2012), the default setting is logBase = exp(2*pi/pi2Div). This relation between the parameters logBase and pi2Div ensures an analogue scaling of the modulus and argument reference zones (see Details section in the documentation of phasePortrait). Conveniently, for the default pi2Div = 18, we obtain logBase == 1.4177..., which is very close to the square root of 2. Thus, when crossing two modulus zones, the modulus at the higher edge of the second zone is almost exactly two times the value at the lower edge of the first zone.

argOffset — The (complex number) argument in radians counterclockwise, at which the argument (phase angle) reference zones are fixed, i.e. the lower angle of the first zone. Default is 0.

bwCols — Color definition for the plot provided as a character vector of length 3. Each element of the vector must be either a color name R recognizes, or a hexadecimal color string like e.g. "#00FF11". The first and the second color make the appearance of two-color phase portraits (see bwType above for details), while the third color is reserved for special cases, where the input value cannot sufficiently evaluated (NaNs, partly Inf). Defaults to c("black", "gray95", "gray"), which leads to an alternation of black and very light gray zones or tiles, and uses a neutral gray in special cases.

asp — Aspect ratio y/x as defined in plot.window. Default is 1, ensuring an accurate representation of distances between points on the screen.

deleteTempFiles — If TRUE (default), all temporary files are deleted after the plot is completed. Set it on FALSE only, if you know exactly what you are doing - the temporary files can occupy large amounts of hard disk space (see details).

noScreenDevice — Suppresses any graphical output if TRUE. This is only intended for test purposes and makes probably only sense together with deleteTempFiles == FALSE. For dimensioning purposes, phasePortraitBw will use a 1 x 1 inch pseudo graphics device in this case. The default for this parameter is FALSE, and you should change it only if you really know what you are doing.

autoDereg — if TRUE, automatically register sequential backend after the plot is completed. Default is FALSE, because registering a parallel backend can be time consuming. Thus, if you want make several phase portraits in succession, you should set autoDereg only for the last one, or simply type foreach::registerDoSEQ after you are done. In any case, you don't want to have an unused parallel backend lying about.

verbose	if TRUE (default), phasePortraitBw will continuously write progress messages to the console. This is convenient for normal purposes, as calculating larger phase portraits in higher resolution may take several minutes. The setting verbose = FALSE, will suppress any output to the console.
...	All parameters accepted by the plot.default function.

Details

This function is intended to be used inside the framework of R base graphics. It plots into the active open graphics device where it will display the phase plot of a user defined function as a raster image. If no graphics device is open when called, the function will plot into the default graphics device. This principle allows to utilize the full functionality of R base graphics. All graphics parameters (par) can be freely set and the function phasePortrait accepts all parameters that can be passed to the plot.default function. This allows all kinds of plots - from scientific representations with annotated axes and auxiliary lines, notation, etc. to poster-like artistic pictures. The general mode of operation, including the usage of parallel processing is exactly the same as with phasePortrait, see details section there.

References

Wegert E (2012). *Visual Complex Functions. An Introduction with Phase Portraits*. Springer, Basel Heidelberg New York Dordrecht London. ISBN 978-3-0348-0179-9.

Examples

```
# Map the complex plane on itself

# x11(width = 8, height = 8)     # Screen device commented out
                                 # due to CRAN test requirements.
                                 # Use it when trying this example
phasePortraitBw("z", xlim = c(-2, 2), ylim = c(-2, 2),
                xlab = "real", ylab = "imaginary",
                verbose = FALSE, # Suppress progress messages
                nCores = 2)      # Max. two cores allowed on CRAN
                                 # not a limit for your own use




# Sinus with default colors and default bwType ("ma")

# x11(width = 8, height = 8)       # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
phasePortraitBw("sin(z)",
                xlim = c(-pi, pi),
                ylim = c(-pi, pi),
                verbose = FALSE,
                nCores = 2)        # Max. two cores allowed on CRAN
                                   # not a limit for your own use
```

```
# Sinus with custom colors and bwType "a"

# x11(width = 8, height = 8)       # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
phasePortraitBw("sin(z)",
                xlim = c(-pi, pi),
                ylim = c(-pi, pi),
                bwType = "a",
                bwCols = c("darkgreen", "green", "gray"),
                verbose = FALSE,
                nCores = 2)        # Max. two cores allowed on CRAN
                                   # not a limit for your own use




# Sinus with custom colors and bwType "m"

# x11(width = 8, height = 8)       # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
phasePortraitBw("sin(z)",
                xlim = c(-pi, pi),
                ylim = c(-pi, pi),
                bwType = "m",
                bwCols = c("darkblue", "skyblue", "gray"),
                verbose = FALSE,
                nCores = 2)        # Max. two cores allowed on CRAN
                                   # not a limit for your own use




# Map the complex plane on itself, show all bwType options

# x11(width = 8, height = 8)       # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
op <- par(mfrow = c(2, 2), mar = c(4.1, 4.1, 1.1, 1.1))
for(bwType in c("ma", "a", "m")) {
  phasePortraitBw("z", xlim = c(-2, 2), ylim = c(-2, 2),
                  bwType = bwType,
                  xlab = "real", ylab = "imaginary",
                  verbose = FALSE, # Suppress progress messages
                  nCores = 2)      # Max. two cores allowed on CRAN
                                   # not a limit for your own use
```

```
}
# Add normal phase portrait for comparison
phasePortrait("z", xlim = c(-2, 2), ylim = c(-2, 2),
              xlab = "real", ylab = "imaginary",
              verbose = FALSE,
              pi2Div = 18,         # Use same angular division as default
                                   # in phasePortraitBw
              nCores = 2)
par(op)




# A rational function, show all bwType options

# x11(width = 8, height = 8)       # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
funString <- "(z + 1.4i - 1.4)^2/(z^3 + 2)"
op <- par(mfrow = c(2, 2), mar = c(4.1, 4.1, 1.1, 1.1))
for(bwType in c("ma", "a", "m")) {
  phasePortraitBw(funString, xlim = c(-2, 2), ylim = c(-2, 2),
                  bwType = bwType,
                  xlab = "real", ylab = "imaginary",
                  verbose = FALSE, # Suppress progress messages
                  nCores = 2)      # Max. two cores allowed on CRAN
                                   # not a limit for your own use
}
# Add normal phase portrait for comparison
phasePortrait(funString, xlim = c(-2, 2), ylim = c(-2, 2),
              xlab = "real", ylab = "imaginary",
              verbose = FALSE,
              pi2Div = 18,         # Use same angular division as default
                                   # in phasePortraitBw
              nCores = 2)
par(op)
```

---

riemannMask *Plot a Riemann sphere mask over a phase portrait*

Description

The function riemannMask can be used for laying a circular mask over an existing phasePortrait (as generated with the function phasePortrait). This mask shades the plot region outside the unit

circle. The unshaded area is a projection on the southern or northern Riemann hemisphere. The standard projection used by phasePortrait, i.e. invertFlip = FALSE hereby corresponds to the southern Riemann hemisphere with the origin being the south pole. If phasePortrait was called with invertFlip = TRUE, then the unit circle contains the northern Riemann hemisphere with the point at infinity in the center (see the vignette for more details). Options for adding annotation, landmark points are available (see Wegert (2012), p. 41). Several parameters are on hand for adjusting the mask's transparency, color, and similar features. some details, this function behaves less nicely under Windows than under Linux (see Details).

Usage

```
riemannMask(
  colMask = "white",
  alphaMask = 0.5,
  circOutline = TRUE,
  circLwd = 1,
  circleSteps = 360,
  circleCol = par("fg"),
  gridCross = FALSE,
  annotSouth = FALSE,
  annotNorth = FALSE,
  xlim = NULL,
  ylim = NULL
)
```

Arguments

colMask	Color for the shaded area outside the unit circle. Defaults to "white". Can be any kind of color definition R accepts. I recommend, however, to use a color definition without a transparency value, because this would be overridden by the parameter alphaMask.
alphaMask	Transparency value for the color defined with colMask. Has to be a value between 0 (fully transparent) and 1 (totally opaque). Defaults to 0.5.
circOutline	Boolean - if TRUE, the outline of the unit circle is drawn. Defaults to TRUE.
circLwd	Line width of the unit circle outline. Obviously relevant only when circOutline == TRUE. Defaults to 1.
circleSteps	Number of vertices to draw the circle. Defaults to 360 (one degree between two vertices).
circleCol	Color of the unit circle, default is the default foreground color (par("fg")).
gridCross	Boolean - if TRUE, a horizontal and a vertical gray line will be drawn over the plot region, intersection in the center of the unit circle. Defaults to FALSE.
annotSouth	Boolean - add landmark points and annotation for a *southern* Riemann hemisphere, defaults to FALSE. This annotation fits to an image that has been created with phasePortrait and the option invertFlip = FALSE.
annotNorth	Boolean - add landmark points and annotation for a *northern* Riemann hemisphere, defaults to FALSE. This annotation fits to an image that has been created with phasePortrait and the option invertFlip = TRUE.

`xlim, ylim` optional, if provided must by numeric vectors of length 2 defining plot limits as usual. They define the outer rectangle of the Riemann mask. If `xlim` or `ylim` is not provided (the standard case), the coordinates of the plot window as given by `par("usr")` will be used for the missing component.

Details

There is, unfortunately, a somewhat different behavior of this function under Linux and Windows systems. Under Windows, the region outside the unit circle is only shaded if the whole unit circle fits into the plot region. If only a part of the unit circle is to be displayed, the shading is completely omitted under Windows (annotation etc. works correctly, however), while it works properly on Linux systems. Obviously, the function `polypath`, which we are using for creating the unit circle template, is interpreted differently on both systems.

References

Wegert E (2012). *Visual Complex Functions. An Introduction with Phase Portraits*. Springer, Basel Heidelberg New York Dordrecht London. ISBN 978-3-0348-0179-9.

Examples

```
# Tangent with fully annotated Riemann masks.
# The axis tick marks on the second diagram (Northern hemisphere)
# have to be interpreted as the real and imaginary parts of 1/z
# (see vignette). The axis labels in this example have been adapted
# accordingly.

# x11(width = 16, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
op <- par(mfrow = c(1, 2), mar = c(4.7, 4.7, 3.5, 3.5))
phasePortrait("tan(z)", pType = "pma",
              main = "Southern Riemann Hemisphere",
              xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
              xlab = "real", ylab = "imaginary",
              xaxs = "i", yaxs = "i",
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

riemannMask(annotSouth = TRUE, gridCross = TRUE)

phasePortrait("tan(z)", pType = "pma",
              main = "Northern Riemann Hemisphere",
              invertFlip = TRUE,
              xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
              xlab = "real (1/z)", ylab = "imaginary (1/z)",
              xaxs = "i", yaxs = "i",
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

riemannMask(annotNorth = TRUE, gridCross = TRUE)
par(op)
```

```
# Rational function with Riemann masks without annotation.
# The axis tick marks on the second diagram (Northern hemisphere)
# have to be interpreted as the real and imaginary parts of 1/z
# (see vignette). The axis labels in this example have been adapted
# accordingly.

# x11(width = 16, height = 8) # Screen device commented out
                              # due to CRAN test requirements.
                              # Use it when trying this example
op <- par(mfrow = c(1, 2), mar = c(4.7, 4.7, 3.5, 3.5))
phasePortrait("(-z^17 - z^15 - z^9 - z^7 - z^2 - z + 1)/(1i*z - 1)",
              pType = "pma",
              main = "Southern Riemann Hemisphere",
              xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
              xlab = "real", ylab = "imaginary",
              xaxs = "i", yaxs = "i",
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

riemannMask(annotSouth = FALSE, gridCross = FALSE, circOutline = FALSE,
            alphaMask = 0.7)

phasePortrait("(-z^17 - z^15 - z^9 - z^7 - z^2 - z + 1)/(1i*z - 1)",
              pType = "pma",
              main = "Northern Riemann Hemisphere",
              invertFlip = TRUE,
              xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
              xlab = "real (1/z)", ylab = "imaginary (1/z)",
              xaxs = "i", yaxs = "i",
              nCores = 2) # Max. two cores on CRAN, not a limit for your use

riemannMask(annotNorth = FALSE, gridCross = FALSE, circOutline = FALSE,
            alphaMask = 0.7)
par(op)
```

`vector2String` *Convert a vector into a comma-separated string*

Description

A simple utility function that transforms any vector into a single character string, where the former vector elements are separated by commas. This is can be useful, in some circumstances, for feeding a series of constant numeric values to `phasePortrait` (see examples). For most applications we recommend, however, to use `phasePortrait`'s parameter `moreArgs` instead.

Usage

```
vector2String(vec)
```

Arguments

vec	The (usually real or complex valued) vector to be converted.

Value

A string, where the former vector elements are separated by commas, enclosed between "c(" and ")".

See Also

Other helpers: `xlimFromYlim()`, `ylimFromXlim()`

Examples

```
# Make a vector of 77 complex random numbers inside the unit circle
n <- 77
a <- complex(n, modulus = runif(n), argument = 2*pi*runif(n))
a <- vector2String(a)
print(a)


# Use this for portraying a Blaschke product

# x11(width = 9.45, height = 6.30) # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
op <- par(mar = c(1, 1, 1, 1), bg = "black")
n <- 77
a <- complex(n, modulus = runif(n), argument = 2*pi*runif(n))
a <- vector2String(a)
FUN <- paste("vapply(z, function(z, a){
                    return(prod(abs(a)/a * (a-z)/(1-Conj(a)*z)))
                  }, a =", a,
             ", FUN.VALUE = complex(1))", sep = "")
phasePortrait(FUN, pType = "p", axes = FALSE,
              xlim = c(-3, 3), ylim = c(-2.0, 2.0),
              nCores = 2) # Max. two cores allowed on CRAN
                          # not a limit for your own use
par(op)
```

`xlimFromYlim` *Adjust xlim to ylim*

Description

This simple function is useful for adjusting x and y coordinate ranges `xlim` and `ylim` in order to maintain a desired display ratio. The latter must be given, the former will be adjusted.

Usage

```
xlimFromYlim(ylim, centerX = 0, x_to_y = 16/9)
```

Arguments

`ylim`	Numeric vector of length 2; the fixed lower and upper boundary of the vertical coordinate range
`centerX`	The horizontal coordinate which the output range is to be centered around (default = 0)
`x_to_y`	The desired ratio of the horizontal (x) to the vertical (y) range. Default is 16/9, a display ratio frequently used for computer or mobile screens

Details

For certain purposes, e.g. producing a graph that exactly matches a screen, the x and y coordinates must be adjusted to match a given display ratio. If the vertical range, `ylim`, the desired ratio, `x_to_y` and the desired center of the x-range, `centerX`, are provided, this function returns an adpated vertical range, that can be used as `ylim` in any plot including `phasePortrait`.

Value

A numeric vector of length 2; the lower and upper boundary of the resulting vertical coordinate range

See Also

Other helpers: `vector2String()`, `ylimFromXlim()`

Examples

```
# Make a phase portrait of a pretty function that fully covers a
# plot with a display aspect ratio of 5/4.

# 9 inch wide window with 5/4 display ratio (x/y)

# x11(width = 9, height = 9 * 4/5) # Screen device commented out
                                   # due to CRAN test requirements.
                                   # Use it when trying this example
ylim <- c(-8, 7)
xlim <- xlimFromYlim(ylim, centerX = 0, x_to_y = 5/4)
op <- par(mar = c(0, 0, 0, 0), bg = "black") # Omit all plot margins
phasePortrait("exp(cosh(1/(z - 2i + 2)^2 * (1/2i - 1/4 + z)^3))", pType = "pm",
xlim = xlim, ylim = ylim, # Apply the coordinate ranges
xaxs = "i", yaxs = "i",   # Allow for now room between plot and axes
nCores = 2) # Max. two cores allowed on CRAN
```

```
            # not a limit for your own use
par(op)
```

`ylimFromXlim` *Adjust ylim to xlim*

Description

This simple function is useful for adjusting x and y coordinate ranges `xlim` and `ylim` in order to maintain a desired display ratio. The former must be given, the latter will be adjusted.

Usage

```
ylimFromXlim(xlim, centerY = 0, x_to_y = 16/9)
```

Arguments

`xlim`	Numeric vector of length 2; the fixed lower and upper boundary of the horizontal coordinate range
`centerY`	The vertical coordinate which the output range is to be centered around (default = 0)
`x_to_y`	The desired ratio of the horizontal (x) to the vertical (y) range. Default is 16/9, a display ratio frequently used for computer or mobile screens

Details

For certain purposes, e.g. producing a graph that exactly matches a screen, the x and y coordinates must be adjusted to match a given display ratio. If the horizontal range, `xlim`, the desired ratio, `x_to_y` and the desired center of the y-range, `centerY` are provided, this function returns an adapted vertical range, that can be used as `ylim` in any plot including `phasePortrait`.

Value

A numeric vector of length 2; the lower and upper boundary of the resulting vertical coordinate range

See Also

Other helpers: `vector2String()`, `xlimFromYlim()`

Examples

```
# Make a phase portrait of a Jacobi theta function that fully covers a
# plot with a display aspect ratio of 4/3.
# 10 inch wide window with 4/3 display ratio (x/y)

# x11(width = 10, height = 10 * 3/4) # Screen device commented out
                                     # due to CRAN test requirements.
                                     # Use it when trying this example
xlim <- c(-3, 3)
ylim <- ylimFromXlim(xlim, centerY = -0.3, x_to_y = 4/3)
op <- par(mar = c(0, 0, 0, 0), bg = "black") # Omit all plot margins
phasePortrait(jacobiTheta, moreArgs = list(tau = 1i/2 - 1/3),
  xlim = xlim, ylim = ylim, # Apply the coordinate ranges
  xaxs = "i", yaxs = "i",   # Allow for now room between plot and axes
  nCores = 1) # Max. two cores allowed on CRAN
              # not a limit for your own use
par(op)
```

Index