Package 'flametree'

April 27, 2021

Title Generate Random Tree-Like Images

Version 0.1.2

Description A generative art system for producing tree-like images using an L-system to create the structures. The package includes tools for generating the data structures and visualise them in a variety of styles.

License MIT + file LICENSE

Encoding UTF-8

URL https://github.com/djnavarro/flametree

BugReports https://github.com/djnavarro/flametree/issues

Imports tibble, dplyr, purrr, tidyr, ggplot2, ggforce, paletteer, magrittr

RoxygenNote 7.1.1

Suggests deldir, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

NeedsCompilation no

Author Danielle Navarro [aut, cre] (<https://orcid.org/0000-0001-7648-6578>)

Maintainer Danielle Navarro <d.navarro@unsw.edu.au>

Repository CRAN

Date/Publication 2021-04-27 07:20:03 UTC

R topics documented:

`flametree_grow` *Generate the data specifying a flametree*

Description

Generate the data specifying a flametree

Usage

```
flametree_grow(
  seed = 286,
  time = 6,
  scale = c(0.6, 0.8, 0.9),
  angle = c(-10, 10, 20),
  split = 2,
  trees = 1,
  seg_col = spark_linear(tree = 2, time = 1),
  seg_wid = spark_decay(time = 0.3, multiplier = 5, constant = 0.1),
  shift_x = spark_random(multiplier = 3),
  shift_y = spark_nothing()
)
```

Arguments

`seed`	Integer seed for the random number generator
`time`	Number of generations to run the iterative process
`scale`	Vector of possible values for the "size rescaling" at each iteration
`angle`	Vector of possible angle changes (in degrees) at each iteration
`split`	Number of splits at each time point
`trees`	Number of trees to generate
`seg_col`	Spark function to control the segment colour
`seg_wid`	Spark function to control the segment width
`shift_x`	Spark function to control horizontal jitter
`shift_y`	Spark function to control vertical jitter

Details

Generative art created with flametree is a visualisation of a data structure created by calling `flametree_grow()`. The underlying algorithm is an iterative branching process: each tree starts out as a single vertical segment, to which multiple new segments are added at the end of the first iteration. Over multiple iterations this creates a tree-like structure.

The user can control how this iterative process unfolds. By setting the `seed` argument the random number generator is reset using `set.seed()`. The `trees` argument specifies the number of trees to create using this process, the `time` argument specifies how many iterations of the branching process

will be run (at least two), and the `split` argument specifies how many new segments (at least two) will be created each time abranching occurs.

When a new segment is created, its size and orientation are controlled by the `scale` and `angle` arguments. The `scale` argument takes a vector of at least two positive numbers. One of these numbers is selected at random whenever a new segment is created, and the length of the new segment is equal to the length of the "parent" segment from which it was created, multiplied by this scaling factor. The orientation of the new segment is controlled by the angle argument in an analogous way. Every time a new segment is generated, one of these angles (interpreted in degrees, not radians) is selected at random. The orientation of the new segment is equal to the orientation of the parent segment plus the sampled angle. Like the `scale` argument, `angle` must contain at least two values.

The remaining arguments (`seg_col`, `seg_wid`, `shift_x`, and `shift_y`) all take functions as their input, and are used to control how the colours (`seg_col`) and width (`seg_wid`) of the segments are created, as well as the horizontal (`shift_x`) and vertical (`shift_y`) displacement of the trees are generated. Functions passed to these arguments take four inputs: `coord_x`, `coord_y`, `id_tree`, and `id_time`. Any function that takes these variables as input can be used for this purpose. However, as a convenience, four "spark" functions are provided that can be used to create functions that are suitable for this purpose: `spark_linear()`, `spark_decay()`, `spark_random()`, and `spark_nothing()`.

These functions are documented in their own help files. To give an example, the default behaviour of `flametree_grow()` adds a random horizontal displacement to each tree to give the impression of multiple trees growing side by side. To suppress this horizontal displacement, set `shift_x = spark_nothing()`.

Value

The output of `flametree_grow()‘` is a tibble with the following columns: `coord_x`, `coord_y`, `id_tree`, `id_time`, `id_path`, `id_leaf`, `id_pathtree`, `id_step`, `seg_deg`, `seg_len`, `seg_col`, and `seg_wid`. Each row in the tibble specifies a single point: every curved segment is defined by three such rows.

The two "coord" columns are numeric variables that specify the location of the point itself. The "id" columns are used as indicators of various kinds. The `id_tree` column contains numbers specifying which tree each point belongs to, and similarly the `id_time` column is a numeric identifier that specifies the time point at which the point was generated (i.e., the iteration of the generative process). The `id_step` column contains a number (0, 1, or 2) indicating whether the point is the first point, the midpoint, or the end point of the relevant curved segment in a tree. In addition, there are two identifier columns used to denote the segments themselves. The `id_path` column is numeric, and assigns value 1 to the "first" segment (i.e., the lowest part of the tree trunk) for every tree, with values increasing numerically for each subsequent segment. Values for `id_path` will uniquely identify a segment within a tree, but when multiple trees are generated there will be multiple segments that have the same `id_path` value. If a unique identifier across trees is needed, use the `id_pathtree` column, which is a character vector constructed by pasting the `id_path` and `id_tree` values into a string, with an underscore as the separator character.

In addition to the two coordinate columns and the six identifier columns, the data generated by `flametree_grow()` contains four "seg" columns that are intended to map onto different visual characteristics of a plot. The `seg_deg` column specifies the orientation of the segment, whereas `seg_len` denotes the length of the segment, `seg_col` specifies the colour (as a numeric value that could be interpreted by a palette), and `seg_wid` specifies the width of the segment. Note that this

information use used differently by the `flametree_plot()` function, depending on what style of plot is generated.

Examples

```
# flametree data structure with default parameters
flametree_grow()

# setting time = 10 runs the generative process
# longer resulting in a table with more rows
flametree_grow(time = 10)

# default behaviour is to randomly displace trees
# by random horizontal perturbation: to switch this
# off use the spark_nothing() function
flametree_grow(shift_x = spark_nothing())
```

---

flametree_plot *Create a plot from a flametree data frame*

Description

Create a plot from a flametree data frame

Usage

```
flametree_plot(
  data,
  background = "black",
  palette = c("#1E2640", "#F3EAC0", "#DC9750", "#922C40"),
  style = "plain"
)
```

Arguments

`data`	The data frame specifying the flametree
`background`	The background colour of the image
`palette`	A vector of colours
`style`	Style of tree to draw

Details

The `flametree_plot()` function provides several ways to visualise the data created by the generative system implemented by `flametree_grow()`. The `background` argument sets the background colour of the image, and should either be a string specifying an RGB hex colour (e.g., "#000000") or the of a colour recognised by R (see the `colours()` function for details). Analogously, the `palette` argument should be a vector of colours. However, the `palette` argument is interpreted

slightly differently depending on which style of plot is created, discussed below. To set the `style` of the resulting plot, pass one of the following style names: "plain" (the default), "voronoi", "wisp", "nativeflora", "minimal", or "themegray".

Plots in the "plain" style have the following properties. Branches of the trees vary in width using the `seg_wid` data column. Each branch is shown as a curved segment created using `geom_bezier2()`, and the colour of the segments is mapped to the `seg_col` column in the data. No leaves are drawn. In this style, the elements of the `palette` are used to create a continuous n-colour gradient using `scale_colour_gradientn()`.

Plots in the "voronoi" style draw the shape of the tree the same way as the plain style, except that the segments do not vary in colour and are rendered using `geom_bezier()` instead of `geom_bezier2()`. Unlike the plain style, stylised "leaves" are drawn by constructing a Voronoi tesselation of the terminal nodes in the tree. Note that computing the tesselation is computationally expensive and this will likely produce errors if there are too many nodes (typically when the `time` parameter to `flametree_grow()` is large). The interpretation of the `palette` argument is slightly different: the first element of the palette is used to set the colour of the trees, and the rest of the palette colours are used to create the gradient palette used to colour the tiles depicted in the Voronoi tesselation.

The `style = "nativeflora"` style creates a plot in which tree branches are rendered as thin segments, with a proportion of those segments removed, and small points are drawn at the end of each terminal segment. The width of the branches does not vary (i.e., `seg_wid` is ignored) and the colour of the branches is constant within tree, but does vary across trees, ignoring the continuous valued `seg_col` variable and using only the `id_tree` variable to do so. As with the plain style, the `palette` colours are used to define an n-colour gradient.

The "wisp" style is similar to nativeflora, but no segments are removed, and the wdith of the branches is mapped to `seg_wid`. It only uses the first two elements of `palette`: the first element specifies the colour of the branches, and the second element specifies the colour of the leaf dots.

The final two styles are simplifications of other styles. The "minimal" style is similar to the plain style but does not use curved segments, relying on `geom_path()` to draw the branches. The "themegray" style does this too, but it ignores the `palette` argument entirely, rendering the trees in black, set against the default gray background specified by the ggplot2 `theme_gray()` function.

Value

A ggplot object.

Examples

```
# the default tree in the plain style
flametree_grow() %>% flametree_plot()

# 10 trees drawn in the nativeflora style
flametree_grow(trees = 10, shift_x = spark_nothing()) %>%
  flametree_plot(style = "nativeflora")

# changing the palette
shades <- c("#A06AB4", "#FFD743", "#07BB9C", "#D773A2")
flametree_grow() %>% flametree_plot(palette = shades)
```

`flametree_save` *Save the plot*

Description

Save the plot

Usage

```
flametree_save(plot, filename, ...)
```

Arguments

`plot`	The ggplot object
`filename`	The path to the file
`...`	Other arguments to be passed to ggsave

Details

The `flametree_save()` function provides a very thin wrapper around the `ggsave()` function from ggplot2. It reverses the order of the first two arguments: the plot argument comes before filename, in order to be more pipe-friendly. The second thing it does is inspect the plot object to determine the background colour, and ensures that colour is also used to specify the background colour for the graphics device (e.g., the bg argument to `png()`). The reason for doing this is that plots created using `flametree_plot()` typically force the coordinates to be on the same scale using `coord_equal()`. As a consequence, if the aspect ratio of the image differs from the aspect ratio of the ggplot there will be sections of the image that show the background colour of the graphics device rather than the background colour specified by the ggplot object. By overriding the default behaviour of `ggsave()`, the `flametree_save()` function ensures that the image has the same background colour everywhere.

Value

Invisibly returns NULL.

Examples

```
## Not run:
# typical usage
flametree_grow(trees = 5, time = 8) %>%
  flametree_plot(style = "voronoi") %>%
  flametree_save(filename = "~/Desktop/myfile.png")

# passing additional arguments to ggsave()
flametree_grow(trees = 5, time = 8) %>%
  flametree_plot(style = "voronoi") %>%
  flametree_save(
    filename = "~/Desktop/myfile.png",
```

```
    height = 8,
    width = 8
  )

## End(Not run)
```

---

sparks *Spark functions to control tree growth*

Description

Spark functions to control tree growth

Usage

```
spark_linear(x = 0, y = 0, tree = 0, time = 0, constant = 0)

spark_decay(x = 0, y = 0, tree = 0, time = 0, multiplier = 2, constant = 0)

spark_random(multiplier = 3, constant = 0)

spark_nothing()
```

Arguments

x	Weight given to the horizontal co-ordinate
y	Weight given to the horizontal co-ordinate
tree	Weight given to the tree number
time	Weight given to the time point
constant	Constant value to be added to the output
multiplier	Scaling parameter that multiplies the output

Details

Some arguments to `flametree_grow()` take numeric input, but `seg_col`, `seg_wid`, `shift_x`, and `shift_y` all take functions as their input, and are used to control how the colours (`seg_col`) and width (`seg_wid`) of the segments are created, as well as the horizontal (`shift_x`) and vertical (`shift_y`) displacement of the trees are generated. Functions passed to these arguments take four inputs: `coord_x`, `coord_y`, `id_tree`, and `id_time` as input. Any function that takes these variables as input and produces a numeric vector of the same length as the input can be used for this purpose. However, as a convenience, four "spark" functions are provided that can be used to create functions that are suitable for this purpose: `spark_linear()`, `spark_decay()`, `spark_random()`, and `spark_nothing()`. Arguments passed to one of the spark functions determine the specific function is generated. For example, `spark_linear()` can be used to construct any linear combination of the inputs: `spark_linear(x = 3,y = 2)` would return a function that computes the sum `(3 *`

coord_x) + (2 * coord_y). Different values provided as input produce different linear functions. Analogously, spark_decay() returns functions that are exponentially decaying functions of a linear combination of inputs. The spark_random() generator can be used to generate functions that return random values, and spark_nothing() produces a function that always returns zero regardless of input.

Value

A function that takes coord_x, coord_y, id_tree, and id_time as input, and returns a numeric vector as output.

Examples

```
# returns a linear function of x and y
spark_linear(x = 3,  y = 2)

# returns a function of time that decays
# exponentially to an asymptote
spark_decay(time = .1, constant = .1)

# returns a numeric vector containing
# copies of the same uniform random number
# constrained to lie between -2.5 and 2.5
spark_random(multiplier = 5)

# returns a function that always produces
# a vector of zeros
spark_nothing()
```

Index