# Package ‘dynatopGIS’

July 22, 2025

**Title** Algorithms for Helping Build Dynamic TOPMODEL Implementations from Spatial Data

**Version** 0.2.5

**Description** A set of algorithms based on Quinn et al. (1991) <doi:10.1002/hyp.3360050106> for processing river network and digital elevation data to build implementations of Dynamic TOPMODEL, a semi-distributed hydrological model proposed in Beven and Freer (2001) <doi:10.1002/hyp.252>. The 'dynatop' package implements simulation code for Dynamic TOPMODEL based on the output of 'dynatopGIS'.

**License** GPL-2

**Encoding** UTF-8

**Imports** R6, terra, methods, jsonlite

**Depends** R (>= 4.0.0)

**BugReports** https://github.com/waternumbers/dynatopGIS/issues

**URL** https://waternumbers.github.io/dynatopGIS/, https://github.com/waternumbers/dynatopGIS

**RoxygenNote** 7.2.3

**Suggests** knitr, rmarkdown

**VignetteBuilder** knitr

**Language** en-GB

**NeedsCompilation** no

**Author** Paul Smith [aut, cre] (ORCID: <https://orcid.org/0000-0002-0034-3412>)

**Maintainer** Paul Smith <paul@waternumbers.co.uk>

**Repository** CRAN

**Date/Publication** 2023-05-11 09:40:02 UTC

## Contents

`dynatopGIS` *R6 Class for processing a catchment to make a Dynamic TOPMODEL*

## Description

This package contains the code for setting up a dynamic TOPMODEL implementation

## Methods

### Public methods:

- `dynatopGIS$new()`
- `dynatopGIS$get_meta()`
- `dynatopGIS$get_working_directory()`
- `dynatopGIS$set_working_directory()`
- `dynatopGIS$add_dem()`
- `dynatopGIS$add_channel()`
- `dynatopGIS$add_layer()`
- `dynatopGIS$get_layer()`
- `dynatopGIS$plot_layer()`
- `dynatopGIS$sink_fill()`
- `dynatopGIS$compute_areas()`
- `dynatopGIS$compute_properties()`
- `dynatopGIS$compute_flow_lengths()`
- `dynatopGIS$classify()`
- `dynatopGIS$combine_classes()`
- `dynatopGIS$create_model()`
- `dynatopGIS$get_version()`
- `dynatopGIS$get_class_method()`
- `dynatopGIS$clone()`

**Method** `new()`**:** Initialise a project, or reopen an existing project

*Usage:*

```
dynatopGIS$new(meta_file, check = TRUE, verbose = TRUE)
```

*Arguments:*

`meta_file` filename and path of the meta data file

`check` logical, should checks be performed [TRUE]

`verbose` printing of checking output [TRUE]

*Details:* This loads the meta data file found at `meta_path`, or creates it with a warning if no file is present. It check is `TRUE` then the meta data file contents are checked with the level of returned information being controlled by `verbose`.

*Returns:* A new ‘dynatopGIS‘ object

**Method** `get_meta()`**:** Get project meta data

*Usage:*

```
dynatopGIS$get_meta()
```

**Method** `get_working_directory()`**:** Get current working directory

*Usage:*

```
dynatopGIS$get_working_directory()
```

*Details:* Newly generated layers are added to the working directory. By default this is the directory containing the meta date file.

**Method** `set_working_directory()`**:** Set current working directory

*Usage:*

```
dynatopGIS$set_working_directory(file_path, create = TRUE)
```

*Arguments:*

`file_path` the path to the new directory to create

`create` should the directory be created if it doesn't exist

*Details:* Newly generated layers are added to the working directory. By default this is the directory containing the meta date file.

**Method** `add_dem()`**:** Import a dem to the 'dynatopGIS' object

*Usage:*

```
dynatopGIS$add_dem(dem, fill_na = TRUE, verbose = FALSE)
```

*Arguments:*

`dem` a `raster` layer object or the path to file containing one which is the DEM

`fill_na` should NA values in dem be filled. See details

`verbose` Should additional progress information be printed

*Details:* If not a `raster` the DEM is read in using the terra package. If `fill_na` is `TRUE` all NA values other then those that link to the edge of the dem are filled so they can be identified as sinks.

*Returns:* suitable for chaining

**Method** `add_channel()`**:** Import channel data to the 'dynatopGIS' object

*Usage:*

```
dynatopGIS$add_channel(
  channel,
 property_names = c(length = "length", startNode = "startNode", endNode = "endNode",
    width = "width"),
  default_width = 2
)
```

*Arguments:*

`channel` a SpatialLinesDataFrame, SpatialPolygonsDataFrame or file path containing the channel information property_names named vector of columns of the spatial data frame to use for channel properties - see details

default_width default width of a channel if not specified in property_names. Defaults to 2 metres.

*Details:* Takes the input channel converts it a SpatialPolygonDataFrame with properties length, startNode and endNode. The variable names in the sp_object data frame which corresponding to these properties can be specified in the property_names vector. In the channel is a SpatialLinesDataFrame (or read in as one) an additional property width is used to buffer the lines and create channel polygons. If required the width property is created using the default value. Note that any columns called length, startNode, endNode and width are overwritten. Any column called id is copied to a column original_id then overwritten.

*Returns:* suitable for chaining

**Method** add_layer()**:** Add a layer of geographical information

*Usage:*

```
dynatopGIS$add_layer(layer_name, file_path)
```

*Arguments:*

layer_name name to give to the layer

file_path the location of the file containing the new layer

*Details:* The file given is read by the terra package and checked against the project meta data. Only layer names not already in use (or reserved) are allowed. If successful the meta data for the project are altered to reflect the new layer name and file location.

*Returns:* suitable for chaining

**Method** get_layer()**:** Get a layer of geographical information or a list of layer names

*Usage:*

```
dynatopGIS$get_layer(layer_name = character(0))
```

*Arguments:*

layer_name name of the layer give to the layer

*Returns:* a ‘raster‘ layer of the requested information if layer_name is given else a vector of layer names

**Method** plot_layer()**:** Plot a layer

*Usage:*

```
dynatopGIS$plot_layer(layer_name, add_channel = TRUE)
```

*Arguments:*

layer_name the name of layer to plot

add_channel should the channel be added to the plot

*Returns:* a plot

**Method** sink_fill()**:** The sink filling algorithm of Planchona and Darboux (2001)

*Usage:*

```
dynatopGIS$sink_fill(
  min_grad = 1e-04,
  max_it = 1e+06,
  verbose = FALSE,
  hot_start = FALSE
)
```

*Arguments:*

`min_grad` Minimum gradient between cell centres

`max_it` maximum number of replacement cycles

`verbose` print out additional diagnostic information

`hot_start` start from filled_dem if it exists

*Details:* The algorithm implemented is that described in Planchona and Darboux, "A fast, simple and versatile algorithm to fill the depressions in digital elevation models" Catena 46 (2001). A pdf can be found at (<https://horizon.documentation.ird.fr/exl-doc/pleins_textes/pleins_textes_7/sous_copyright/01003

**Method** `compute_areas()`**:** Computes area maps and presence of channel in dem pixels

*Usage:*

```
dynatopGIS$compute_areas()
```

*Details:* The algorithm calculates the land and channel area for each DEM pixel assigning a channel_id to each pixel with a channel area.

**Method** `compute_properties()`**:** Computes statistics e.g. gradient, log(upslope area / gradient) for raster cells

*Usage:*

```
dynatopGIS$compute_properties(min_grad = 1e-04, verbose = FALSE)
```

*Arguments:*

`min_grad` gradient that can be assigned to a pixel if it can't be computed

`verbose` print out additional diagnostic information

*Details:* The algorithm passed through the cells in decreasing height. Min grad is applied to all cells. It is also used for missing gradients in pixels which are partially channel but have no upslope neighbours.

**Method** `compute_flow_lengths()`**:** Computes flow length for each pixel to the channel

*Usage:*

```
dynatopGIS$compute_flow_lengths(verbose = FALSE)
```

*Arguments:*

`verbose` print out additional diagnostic information

*Details:* The algorithm passed through the cells in increasing height. For measures of flow length to the channel are computed. The shortest length (minimum length to channel through any flow path), the dominant length (the length taking the flow direction with the highest fraction for each pixel on the path) and expected flow length (flow length based on sum of downslope flow lengths based on fraction of flow to each cell) and band (strict sequence to ensure that all contributing cell have a higher band value). By definition cells in the channel that have no land area have a length (or band) of NA.

**Method** `classify()`: Create a catchment classification based cutting an existing layer into classes

*Usage:*

```
dynatopGIS$classify(layer_name, base_layer, cuts)
```

*Arguments:*

`layer_name` name of the new layer to create

`base_layer` name of the layer to be cut into classes

`cuts` values on which to cut into classes. These should be numeric and define either the number of bands (single value) or breaks between band (multiple values).

*Details:* This applies the given cuts to the supplied landscape layer to produce areal groupings of the catchment. Cuts are implement using `terra::cut` with `include.lowest = TRUE`. Note that is specifying a vector of cuts values outside the limits will be set to NA.

**Method** `combine_classes()`: Combine any number of classifications based on unique combinations and burns

*Usage:*

```
dynatopGIS$combine_classes(layer_name, pairs, burns = NULL)
```

*Arguments:*

`layer_name` name of the new layer to create

`pairs` a vector of layer names to combine into new classes through unique combinations. Names should correspond to raster layers in the project directory.

`burns` a vector of layer names which are to be burnt on

*Details:* This applies the given cuts to the supplied landscape layers to produce areal groupings of the catchment. Burns are added directly in the order they are given. Cuts are implement using `terra::cut` with `include.lowest = TRUE`. Note that is specifying a vector of cuts values outside the limits will be set to NA.

**Method** `create_model()`: Compute a Dynamic TOPMODEL

*Usage:*

```
dynatopGIS$create_model(
  layer_name,
  class_layer,
  dist_layer,
  transmissivity = c("exp", "bexp", "cnst", "dexp"),
  channel_solver = c("histogram"),
  dist_delta = 0,
  rain_layer = NULL,
  rain_label = character(0),
  pet_layer = NULL,
  pet_label = character(0),
  verbose = FALSE
)
```

*Arguments:*

`layer_name` name for the new model and layers `class_layer` the layer defining the topographic classes

`dist_layer` the layer defining the distances to the channel

`transmissivity` transmissivity profile to use

`channel_solver` channel solver to use

`dist_delta` used in computing flow linkages see details

`rain_layer` the layer defining the rainfall inputs

`rain_label` Prepended to rain_layer values to give rainfall series name

`pet_layer` the layer defining the pet inputs

`pet_label` Prepended to pet_layer values to give pet series name

`verbose` print more details of progress

*Details:* The `class_layer` is used to define the HRUs. Flow between HRUs is based on the distance to a channel. For each HRU the shortest distance to a channel is computed. Flow from a HRU can only go to a HRU with a lower shortest distance to the channel. Flow from a HRU can occur from any raster cell within the HRU whose distance to the channel is within dist_delta of the shortest distance within the HRU. Setting the transmissivity and channel_solver options ensure the model is set up with the correct parameters present. The `rain_layer` (`pet_layer`) can contain the numeric id values of different rainfall (pet) series. If the value of `rain_layer` (`pet_layer`) is not `NULL` the weights used to compute an averaged input value for each HRU are computed, otherwise an input table for the models generated with the value "missing" used in place of the series name.

**Method** `get_version()`**:** get the version number

*Usage:*

```
dynatopGIS$get_version()
```

*Details:* the version number indicates the version of the algorithms within the object

*Returns:* a numeric version number

**Method** `get_class_method()`**:** get the cuts and burns used to classify

*Usage:*

```
dynatopGIS$get_class_method(layer_name)
```

*Arguments:*

`layer_name` the name of layer whose classification method is returned

*Returns:* a list with two elements, cuts and burns

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
dynatopGIS$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## The vignettes contains more examples of the method calls.

## create temport directory for output
demo_dir <- tempfile("dygis")
dir.create(demo_dir)

## initialise processing
ctch <- dynatopGIS$new(file.path(demo_dir,"meta.json"))

## add digital elevation and channel data
dem_file <- system.file("extdata", "SwindaleDTM40m.tif", package="dynatopGIS", mustWork = TRUE)
dem <- terra::rast(dem_file)
ctch$add_dem(dem)
channel_file <- system.file("extdata", "SwindaleRiverNetwork.shp",
package="dynatopGIS", mustWork = TRUE)
sp_lines <- terra::vect(channel_file)
property_names <- c(channel_id="identifier",endNode="endNode",startNode="startNode",length="length")
ctch$add_channel(sp_lines,property_names)

## compute properties
ctch$compute_areas()
ctch$sink_fill() ## fill sinks in the catchment

ctch$compute_properties() # like topograpihc index and contour length
ctch$compute_flow_lengths()

## classify and create a model

ctch$classify("atb_20","atb",cuts=20) # classify using the topographic index
ctch$get_class_method("atb_20") ## see the details of the classification
ctch$combine_classes("atb_20_band",c("atb_20","band")) ## combine classes
ctch$create_model("new_model","atb_20_band","band") ## create a model
list.files(demo_dir,pattern="new_model*") ## look at the output files for the model

## tidy up
unlink(demo_dir)
```

# Index