sDNA_GH

Authors

- James Parrott
- Crispin Cooper (Cardiff University)

sDNA is a world leading tool for Spatial Design Network Analysis. sDNA_GH is a plug-in for Grasshopper providing components that run the tools from a local sDNA installation, on Rhino and Grasshopper geometry and data. Currently requires Windows.

Shenzhen University and Hong Kong University funded the initial and subsequent development phases respectively. Wedderburn Transport Planning and Alain Chiaradia worked on defining the idea, and staunchly supported the project along with Cardiff University. Sara Nalaskowska, Siddharth Khakhar and Fan Zitian, all provided invaluable help during development and testing. Huge thanks to all.

sDNA_GH functionality

sDNA_GH:

- Reads a network's polyline Geometry from Rhino or Grasshopper.
- Reads data from any User Text on Rhino objects.
- Writes a network's links (formed by one or more polylines), and associated data to shapefiles.
- Initiates an sDNA tool that processes that shapefile, and e.g. carries out a network preparation or an analysis.
- Reads polyline shapefiles produced by the sDNA tool, and any associated data e.g. the results from sDNA.
- Parses the selected data field, and normalises and classifies the data, sorting the polylines if necessary.
- Colours can be allocated to each polyline based on the parsed data, either using a Colour Gradient component, or automatically.
- Visually represents the results from sDNA by colouring a new layer of polylines, or the original Rhino shapes.
- Allows easy adding of a native Legend.

User manual.

version = '3.0.1'

Table of contents

System Requirements.

Software

1. Windows (8.1, 10, or 11)
2. Python. (Please note non-CPythons, e.g. Iron Python, are not supported, as invalid shapefiles will be produced).
3. sDNA.
4. Rhino and Grasshopper (tested in Rhino 7)

Hardware

1. 64-bit Intel or AMD processor (Not ARM)
2. No more than 63 CPU Cores.
3. 8 GB memory (RAM) or more is recommended.
4. 1.2 GB disk space.

Installation.

Experimental installation via pip

`pip install --target=%APPDATA%\Grasshopper\UserObjects\sdna-gh sDNA_GH` There are few modules in sDNA_GH that can be run outside of Grasshopper, let alone Rhino. The wheels (bdist releases) contain launcher components, and import the main sDNA_GH package.
Therefore for most users, installation should target a folder from which Grasshopper will find the sDNA_GH components, not in a venv or standard CPython installation. The folder above is the same as for a standard installation.

Experimental installation in CPython 3 components (Rhino 8's Grasshopper, also via pip)

`#r: sDNA_GH`

Standard installation.

1. Ensure you have an installation of Rhino 3D including Grasshopper (versions 6 and 7 are supported).

sDNA_GH provides some functionality without sDNA installed, e.g. Selecting real Rhino objects, User Text components, reading and writing shapefiles, parsing data, and recolouring objects. But to use the sDNA components, an installation of sDNA is required.

sDNA dependencies.

sDNA itself may require the 64 bit (x64) Visual Studio redistributable [^0].

Official sDNA installer (and Python).

2. Ensure you have a supported version of Python (CPython, the reference implementation). Do not run sDNA in non CPythons, as invalid shape files will be produced. sDNA has been tested with a lot of different CPython versions. But these do not include the CPython shipped with Rhino 8.
3. sDNA Learn requires numpy. Numpy can be installed by opening a `cmd` window and typing: `python -m pip install numpy`. Optional: If this Python installation is also used by other processes, to guarantee there are no dependency conflicts, create a venv to run sDNA in, e.g. firstly by entering: `python -m venv <path_to_sDNA_venv>` `<path_to_sDNA_venv>\Scripts\pip.exe install numpy` (enter `<path_to_sDNA_venv>\Scripts\activate.bat` to test the venv from the command line) and secondly within Grasshopper set `python` to \Scripts` on a config component (or in config.toml) in step 13.iii below.
4. To use sDNA with sDNA_GH, ensure you have an installation of sDNA. sDNA+ is now open source. As above, sDNA may also require a Visual Studio redistributable. If you chose a non-default installation directory, remember it for step 13.ii.

sDNA_GH

5. Download `sdna-gh.zip` from (food4rhino)[https://www.food4rhino.com/en/app/sdnagh] or the sDNA_GH releases page on Github.

6. Ensure `sdna-gh.zip` is unblocked: Open File Explorer and go to your Downloads folder (or whichever folder you saved it in). Right click it and select `Properties` from the bottom of the menu. Then click on the *Unblock* check box at the bottom (right of *Security*), then click `OK` or `Apply`. The check box and *Security* section should

disappear. This should unblock all the files in the zip archive. Please do not automatically trust and unblock all software downloaded from anywhere on the internet [^1].

7. Open Rhino and Grasshopper.

8. In Grasshopper's pull down menus (above the tabs ribbon at the top) click `File` -> `Special folders` -> `User Objects Folder`. The default in Rhino 7 is `%appdata%\Grasshopper\UserObjects`. Note, this is different to the Components Folder used by many other plug-ins (i.e. not `%appdata%\Grasshopper\Libraries`).

9. Copy `sdna-gh.zip` to this folder (e.g. it should be at `%appdata%\Grasshopper\UserObjects\sdna-gh.zip`).

10. Unzip `sdna-gh.zip` to this location (in Windows 10 right click `sdna-gh.zip` and select `Extract All ...`, then click `Extract` to use the suggested location). In the User Objects folder, a new folder called `sdna-gh` should have been created, itself containing a subfolder called `sDNA_GH` (plus license.md and copies of these readme files). If necessary, rename the outer parent folder (inside the User Objects folder) to exactly "sdna-gh" (no quotes). E.g. if you previously downloaded an earlier version of sDNA_GH (or any other file called) sdna-gh.zip, your web browser may have renamed a second download to sdna-gh(1).zip. In this case, you will need to either rename the file or the folder created by unzipping it.

11. Restart Rhino and Grasshopper.

12. The sDNA_GH plug in components should now be available under a new "sDNA" tab in the ribbon amongst the tabs for all the plug-ins you have installed (right of `Mesh`, `Intersect`, `Transform` and `Display` etc).

13. To use sDNA with sDNA_GH, if no preferences are specified, sDNA_GH will search for sDNA and Python installations automatically, using the first one of each it finds. If you are using sDNA Learn with Python in a venv as recommended, or otherwise to ensure sDNA_GH uses a particular version of sDNA and the correct Python interpreter, it is recommended on first usage to: -place a Config component on the canvas (the component with a gear/cog icon in `Extra`). -Specify the file path of the sDNA folder (containing `sDNAUISpec.py` and `runsdnacommand.py`) of the sDNA installation you wish to use in the `sDNA_folders` input (e.g. `C:\Program Files (x86)\sDNA`). -Specify the file path of the sDNA venv's Python (e.g. `\Scripts\python.exe` from step 3), or the path of the chosen Python interpreter's main executable in the `python` input. -Specify any other options you wish to save and reuse on all projects, if necessary by zooming in, and adding a custom input Param with each option's name. -Connect a true Boolean toggle to `go`. An installation wide user options file (`config.toml`) will be created if there isn't one already. -To save options to other project specific `config.toml` files, specify the file path in `save_to` and repeat the previous 4 sub steps.

14. If a newer version of sDNA is used in future with tools unknown to sDNA_GH at the time it was built, if a Config component is placed, and the path of the new sDNA specified in `sDNA_folders`, sDNA_GH will attempt to automatically build components and user objects for the new sDNA tools, and add them to Grasshopper for you. Set `make_new_comps` to false to prevent this.

Usage.

Tools.

Each tool has a component that runs that tool. `auto_` rules may cause a component to also run other tools as well.
For example, by default, sDNA components run Write_Shp before, and Read_Shp after, their sDNA tool.

Common component input and output Params

OK This output is true when a component has executed successfully.

go Set this input to true (e.g. from a boolean toggle component) to run a component's tool.

file Specifies the path of a file to write to, or that was written to.

Data It is possible to do most tasks in sDNA_GH without ever looking at `Data` in detail. The order of the deepest branches must correspond with the order of the Geometric objects. But for example to specify weights elsewhere than on their corresponding Polylines, it is also possible to construct `Data` manually. If so, `Data` must be a Data Tree 2 branches deep at the first level: a branch each for keys {0;0} and values {0;1}. The two nodes of this structure should have a branch for each geometric object (so the nth's keys and values should have paths {0;0;n} and {0;1;n}). The lists at these nodes must be of equal length. The mth key and value of the nth geometric object should be {0;0;n}[m] and {0;1;n}[m] respectively.
Read_Shp supplies a Data Tree in this required format, if the data is read from User Text or from a Shapefile. Grasshopper's path tools can be used to adjust compatible Data Trees into this format.

Geom Accepts a list of geometric objects (Guids of Rhino objects or native Grasshopper objects). Data trees of objects need to be flattened into lists. To use Rhino objects referenced from Grasshopper parameter objects (instead of their Grasshopper versions which are often obscured unless the Rhino shapes are set to Hidden) run the output of the Geometry (Geo) or Curve (Crv) through a Guid (ID) parameter object first.

gdm An alternative to both **Geom** and **Data**. Accepts a Geometry-data-mapping, a list of nested dictionaries (standard Python objects). The keys are the UUIDs of geometric objects. The values are also dictionaries, containing key/value pairs for use as User Text.

opts Accepts an options data structure (a nested dictionary of named tuples) from another sDNA_GH component. Only of use if they are not synced to the global module options.

config The path of a TOML file (e.g. `config.toml`) to be read in containing sDNA_GH options settings. Shared between synchronised components.

Support tools

Config (config)

Loads custom user options and configuration files (`.toml`). Saves options to a `.toml` file if go is true. If a `.toml` file is specified in `save_to`, it is saved to. Otherwise the default value of `save_to` is the installation-wide user options file. One is created if it does not already exist. This will overwrite existing files.

If not using a `config.toml` file, then e.g. when using a config component to set true `auto_` options, to guarantee your components are setup correctly when reloading a saved `.gh` file, this component must run before all your others. To ensure a component runs first, select it and press `Ctrl` + `B` (or from the pull-down menu select `Edit` -> `Arrange` -> `Put To Back`) before saving the `.gh` file.

Read_Geom (get_Geom)

Gets real strings of the uuid references to Rhino polylines and degree-1 Nurbs curves for subsequent sDNA_GH tools. Grasshopper Geom and Curve params create Grasshopper references to Rhino objects. The actual Rhino uuid is required to read and write User Text on a Rhino object.

Set `selected` to true, to only read objects that are selected. Similarly, specify `layer` to the name of a layer, to only read objects from that layer.

Note, if `sync = True` it will remember its previous setting, in which case to go back to selecting all layers, `layer` must be set to any value that is not the name of a layer. Similarly, to go back to selecting everything (not just selected geometry) set `selected = false`. If the component has `sync = false`, an input Params that has had its value set, when subsequently disconnected will fallback to whatever its previous value was.

Shapefile tools

Write_Shp (write_shapefile)

Writes a DataTree in `Data` and a list of polylines in `Geom` to a shapefile. If not specified in `file`, a default file name based on the Rhino doc or Grasshopper doc name is used (unless `auto_update_Rhino_doc_path = false`). `overwrite_shp` = true overwrites existing files, otherwise if it is false Write_Shp creates new automatically named files up to a maximum of `max_new_files` (20 by default). **WARNING! Shapefiles created with default names (due to no valid file path being specified in `file` by the user) will be deleted by subsequent sDNA tools if `strict_no_del` = false, `overwrite_shp` = false, and `del_after_sDNA` = true.**

To create a projection (.prj) file for the new shapefile, specify the path of an existing .prj file in `prj`.

If no Data is supplied, if no read_User_Text component is connected to its input, and if `auto_read_User_Text` is true, this tool will first call read_User_Text.

To work with sDNA, data records are only written to the Shapefile (associated with a shape corresponding to a Rhino / GH polyline) if its field matches the template string specified in `input_key_str`. The field name has a maximum of 10 characters long, and is taken from the `{name}` value (in the key name if it originated as User Text). To write all data with any key name (shorter then 11 characters) to the Shapefile, set `input_key_str` to `{name}`.

Shapefile data entries (records) only allow a maximum width of 254 Ascii characters, or 254 bytes for Unicode strings. Encodings can require more than one byte per code point. So some UTF-8 and UTF-16 encoded Unicode strings may only be 120 code points long, or even shorter.
This is an intrinsic limitation of the shapefile format [https://en.wikipedia.org/wiki/Shapefile#Data_storage].

Rhino and Grasshopper shapes must support certain methods (to retrieve the points list of their vertices) to be written to shp files of the following types: `PolylineVertices` for `POLYLINE`s and `POLYGON`s, `PointCoordinates` for `POINT`s, `PointCloudPoints` for `MULTIPOINT`, and `MeshVertices` for `MULTIPATCH`. Please note: `MULTIPATCH`s are not supported, and `POINT`s and `MULTIPOINT`s are experimental and untested.

Write_Shp attempts to coerce data types to set the Shapefile field type and size correctly. The full coercion order is bool -> int -> (float / Decimal) -> Date -> string (Shapefile field types L -> N -> F -> D -> C respectively). If attempts at boolean and integer conversion fail, by default the data is then coerced to a Decimal. The number of significant figures for Decimals is 12 by default. This can be altered in `precision`. If `decimal` is set to false, Write_Shp coerces the value to a float instead of a Decimal object. The maximum number of decimal places is `max_dp`. Dates are attempted to be built using Python's `datetime.date` object. If a datetime.date cannot be built, if `yyyy_mm_dd` is true, a date must be in yyyy mm dd format. Otherwise if `yyyy_mm_dd` is false as it is by default, dd mm yyyy and mm dd yyyy are also supported. In both cases, the supported separators are one of: -.,;:/\ or a single space.

Read_Shp (read_shapefile)

Reads in polylines and associated data records from a shapefile of polylines. Creates new objects if `new_geom` = true or no objects corresponding to the shapefile are specified in `Geom`. Specify the path of the .shp file to read in `file`. **WARNING! If a valid file path was not specified in `file` on a preceding Write_Shp component, and that file was used by an sDNA tool, Read_Shp deletes sDNA output shapefiles with default names if `strict_no_del` = false, `overwrite_shp` = false, and `del_after_read` = true.** If a list of existing geometry is provided in `Geom` that corresponds to (is the same length as) the data records in the shapefile, and if new_geom = false, only the data is read from the shapefile. Otherwise the polylines in the shapefile are added as new Rhino Polyline objects if bake =

true; otherwise as Grasshopper Polyline objects.

If an attempt to add a polyline fails due to Rhino's validity rules, a degree-1 Nurbs curve is added instead. If this fails, error handling according to Rhino's validity rules is carried out, to inform the user which shape is not valid. A list of the shape numbers, and any known reason they are invalid (if any) is outputted in `invalid`, if the user wishes to fix their data. Alternatively, invalid shapes can simply be skipped by setting `ignore_invalid` to true (but in this case data associated with these shapes is lost).

The bounding box output `bbox` is provided to create a legend frame within Recolour_Objects (its value is calculated from the shape file). The abbreviations and field names from an sDNA results field file (if a file with the same name ending in .names.csv exists) are also read in, and supplied on `abbrevs` so that a drop-down list may be created, for easy selection of the data field for subsequent parsing and plotting. If no separate Recolour_Objects Component is detected connected to the component's outputs downstream and `auto_plot_data = true`, Recolour_Objects is called afterwards.

Plotting tools

Parse_Data (parse_data)

Parse the data in a Data Tree of numerical data (in `Data`) from a specified `field`, for subsequent colouring and plotting. If browsing different results fields, to reset `plot_max` and `plot_min` between different data sets, set `sync` to false on a Parse_Data component. Be sure to supply the list of the data's associated geometric objects (in `Geom`), as legend tags and class midpoint values are appended to the outputted `Geom` and `Data` lists respectively. Some classifiers sort the data into ascending order (if supplied, the geometry objects will then be reordered too, preserving their correspondence). To force a sort, according to `field` regardless, set `sort_data` to true. To make each parsed data point, take the same value as its class midpoint, set `colour_as_class` to true. Use this component separately from Recolour_Objects to calculate colours with a visible Grasshopper Colour Gradient component. Max and Min bounds can be overridden (in `plot_max` and `plot_min`). **WARNING! Parsing is for the purpose of colourisation, e.g. in order to produce the desired result from Recolour_Objects. Therefore, although the inputted Data is not changed, the Data outputted almost certainly will be changed, so should be assumed to be false.**
After parsing, the legend tags are the definitive reference for what each colour means, not the outputted data values. In particular, if `colour_as_class` = true, the parsed data will take far fewer distinct values than the number of polylines in a large network. To parse numerical data that uses a numerical format different to your system's normal setting (e.g. with a different radix character: ',' or '.' or thousands separator: ',' or '_'), set `locale` to the corresponding IETF RFC1766, ISO 3166 Alpha-2 code (e.g. `fr`, `cn`, `pl`).

Field to plot Specify the actual numeric data values to be parsed from all the provided 'User Text values' by setting `field` to the name of the corresponding 'User Text key'. Valid `field` values for sDNA output shapefiles are in `fields`.

Bounds The domain this data is parsed against can be customised by setting the options `plot_min`, `plot_max`, shifting it, widening it or narrowing it, e.g. to exclude erroneous outliers. If `plot_min`, `plot_max` are both numbers and `plot_min` < `plot_max`, their values will be used; otherwise the max and min are automatically calculated from the list of values in the 'User Text values' of Data corresponding to the 'User Text key' named in `field`. To go back to automatic calculation after an override, choose invalid values that satisfy `plot_min` >= `plot_max`. Set `exclude` to true to exclude data points lower than `plot_min` or higher than `plot_max` from the output altogether (and their corresponding objects from Geom). If `exclude = false` the `plot_min`, `plot_max` will be applied to limit the values of outlying data points (cap and collar).

Classes (bins / categories for the legend) Either, specify the number of classes desired in the legend in `num_classes` (the default is 7), or specify a list of the actual class boundaries desired in `class_bounds` manually. Note these are the inter-class bounds. Use `plot_min` for the lower bound of the bottom class and `plot_max` for the upper bound of the top class. There should be n-1 inter-class bounds, n classes and n+1 class bounds including the `plot_max` and `plot_min`.

If no valid inter-class boundaries are manually specified in `class_bounds`, sDNA_GH will automatically calculate them based on the following methods (each are valid values for `class_spacing`):

- `quantile` - classify 'spikes' in the frequency distribution containing more data points than the normal class size, narrower than a specified width (in `max_width`). Then classify the remaining data values according to `adjuster`. Sorts the data.
- `adjuster` - Sort the data and place inter class bounds in ascending order so that classes contain approximately the same number of data points, adjusting the inter-class boundaries to the closest gap, if one would otherwise be placed between identical data values.
- `linear` - space the inter-class boundaries evenly between `plot_min` and `plot_max`.
- `exponential` - space the inter-class boundaries between `plot_min` and `plot_max` but with a skewed spacing determined from an exponential curve (customisable `base`).
- `log` - space the inter-class boundaries between `plot_min` and `plot_max` but with a skewed spacing determined from an logarithmic curve (customisable `base`).
- `simple` - Uncomplicated quantile classification. Sort the data and divide it into classes containing approximately the same number of data points. Take no action if this places an interclass bound between identical values.
- `max_deltas` - place the inter-class boundaries at the largest gaps between consecutive data points. Prone to distortion from outlying values. Sorts the data.

If after one of the above classification methods (especially `simple`), inter-class bounds have still been placed between indistinguishable data points (closer than `tol`), sDNA_GH can simply remove them (meaning there will be one few class for each) if `remove_overlaps` is set to true.

Legend class names Three customisable fields are provided in the options for the first, general and last legend tag names respectively: `first_leg_tag_str = 'below` {upper}', gen_leg_tag_str = '{lower} - {upper}', last_leg_tag_str = 'above {lower}'. A formatting string e.g. num_format = '{:.5n}' is applied to all numbers before displaying in the legend tags - it can be customised to set any desired number of decimal places or significant figures. If set, all must be valid Python format strings, with the supported named fields lower, mid_pt and upper, except num_format which supports a single unnamed field.

Re-normalisation Finally in order to produce a recolouring that has a set number of identical colours (the same for each member of the same class) it is possible to assign the value of the midpoint of its class to each parsed data point. The parsed values may then additionally be renormalised, in order to tinker with the spread of colours against different colour gradients and other possible colourisations and applications, it is possible to 'renormalise' the parsed data points - the default value of re_normaliser is linear (for no re-normalisation) but exponential or log curves are also supported (with customisable base as above).

Finally, the errors raised if there are small classes or class overlaps can be suppressed by setting suppress_small_classes_error or suppress_class_overlap_error to true respectively.

Recolour_Objects (recolour_objects)

Recolour objects (and legend tags) based on pre-parsed and pre-normalised data, or already calculated colours (as RGB triples). Recolouring Rhino Geometry can be much slower than recolouring Grasshopper Geometry (the latter via a custom preview component connected to Geom and Data). Nonetheless, to recolour Rhino objects from Grasshopper Referenced Curves (instead of their actual Grasshopper copies, which may be obscured unless the overlying Rhino geometry is set to Hidden), connect the Referenced Curves to a Guid (ID) parameter object first.

If unparsed data is inputted, Parse_Data is first called. Custom colour curves are supported using a 3D quadratic spline between the triples of numbers: rgb_min, rgb_mid and rgb_max. Otherwise, use the Grasshopper Colour Gradient internally (via Node In Code) by setting Col_Grad to true and picking a setting from 0 to 7 for Col_Grad_num (0 : 'EarthlyBrown', 1 : 'Forest', 2 : 'GreyScale', 3 : 'Heat', 4 : 'Pink', 5 : 'Spectrum', 6 : 'Traffic', 7 : 'Zebra'). Set line_width to control the width of the line of Rhino geom objects (the default is 4).

Create a legend by connecting leg_cols, leg_tags and leg_frame to a Grasshopper Legend component. The coordinates of the corners of the Rectangle provided in leg_frame may be overridden by specifying leg_extent (xmin, ymin, xmax, ymax); alternatively any rectangle object can be passed into leg_frame on the GH Legend component itself. Custom legend tag templates and class boundaries are supported via four format strings (first_leg_tag_str, gen_leg_tag_str, last_leg_tag_str and num_format) as per Parse_Data.

To recolour Grasshopper geometry instead of Rhino Geometry (i.e. unbaked objects), connect the `Data` and `Geom` outputs to a Grasshopper Custom Preview component (line widths of GH objects cannot be increased).

User Text tools

Data tools

Read_User_Text (read_User_Text)

Reads all User Text from the list of Rhino objects in `Geom` - these must be Rhino objects from Read_Geom (Grasshopper references to Rhino objects, e.g. from a geom Param will not work). If `auto_get_Geom` = true and if no Read_Geom component is connected to its inputs, Read_Geom is first called. If `compute_vals` = true, values starting and ending in `%` referring to a Rhino object's UUID (e.g. `%<CurveLength("ac4669e5-53a6-4c2b-9080-bbc67129d93e")>%`) are computed using Rhino.RhinoApp.ParseTextField.

Write_User_Text (write_User_Text)

Writes User Text to Rhino objects, using a specified pattern for the keys. Specify the data tree to write in `Data`, and the list of Rhino objects to write to in `Geom`. The format of the User Text key can be customised in the Python format string `output_key_str`, accepting two named fields (e.g. = `sDNA output={name} run time={datetime}`).

A field specifying an originating Rhino object's UUID `uuid_field` will be omitted. If a key of that name already exists, it will be overwritten if `overwrite_UserText` is true. Otherwise a suffix will be appended to it, based on an integer counter and the format string in `dupe_key_suffix`, until a unique key name is found, up to a limit of `max_new_keys` (overwrite warnings can be suppressed by setting `suppress_overwrite_warning` to true).

sDNA Tools

By default, sDNA components run the Write_Shp tool before, and Read_Shp tool after, the actual sDNA tool of the component, passing in and out the names of the special temporary shapefiles. To prevent this, e.g. if using separate Write_Shp or Read_Shp components, set `auto_write_Shp` or `auto_read_Shp` to false respectively.

Analysis tools

- sDNA tools run sDNA from the command line, using the Python interpreter in `python`.
- All sDNA tools try to load an sDNA installation. The first pair of sDNAUISpec.py and runsdnacommand.py files matching the names in `sDNAUISpec` and `runsdnacommand`, found in a folder in sDNA_paths are loaded (if the corresponding sDNA is not already loaded). This is used to run the correct sDNA tool in the corresponding `/bin` sub

folder, and to add Input Params to the sDNA component for each of its sDNA tool's inputs.
- By default an sDNA tool component will show all the possible inputs on its input Params. To show only the essential inputs instead (and make the components a lot smaller) set `show_all` = false.
- sDNA tools require a shapefile to be specified in `file` or `input`. If Write_User_Text is run beforehand and a file name is not specified, a default file name will be used.
- if the user Param `make_advanced` is true, all other unrecognised user params on the component will be added into the advanced config string.

Auto-run tool rules.

The `auto_` options control automatic insertion of extra support tools, into an sDNA component's list of tools. These extra tools can allow sDNA components to do anything between simply being pure wrappers around the sDNA command line interface, to carrying out an entire workflow from Rhino geometry and User Text, finishing with recolouring the Rhino Geometry with the sDNA results. Running tools within a single component should be faster and require less memory, as the Data and Geometry is all kept as the internal data structure (a `gdm`) between tools, instead of being transformed into Grasshopper native form after each component runs, and into Python `gdm` form again when the next component runs.

If `auto_read_User_Text` is true, all sDNA components attempt to check, if any read_User_Text components are already connected to its inputs (upstream). If not, the sDNA component will insert the read_User_Text tool before all instances of the auto_write_Shp tool (which is run before each sDNA tool if `auto_write_Shp` is true, as above). Similarly, if `auto_get_Geom` is true, Read_Geom is run before all instances of read_User_Text. So if all `auto_` options are true, an sDNA component will take in geometry from Rhino directly, write it to a shapefile, run the analysis in sDNA, read in the output shapefile (data and shapes), parse the data, and recolour the original Rhino polylines. **WARNING! If a valid file path was not specified in `file` or `input` on a preceding Write_Shp component, and that file was used by an sDNA tool, sDNA components will delete input shapefiles with default names if `strict_no_del` = false, `overwrite_shp` = false, and `del_after_sDNA` = true.**
The sDNA tool descriptions below are copied almost verbatim from the sDNA manual:

sDNA_Integral (sDNAIntegral)

sDNA Integral is the core analysis tool of sDNA. It computes several flow, accessibility, severance and efficiency measures on networks.

This automatically calls other support tools, handling an entire Rhino geometry workflow from this one component, additionally running Read_Geom and Write_Shp before the sDNA tool itself, and then Read_Shp and Recolour_Objects afterwards (unless `auto_write_Shp` = false or `auto_read_Shp` = false respectively). **WARNING! All sDNA tools will delete the shapefile named in input after it has been read in, if `del_after_sDNA` = true and `strict_no_del` = false (as they are by default).**

To add or remove existing `Geometry` before the results file is read in (to control creation of new geometry objects), set `auto_read_Shp` = false and connect a Read_Shp component. To analyse a network of Grasshopper Geometry set `auto_get_Geom` and `auto_read_User_Text` to false. To access the data and geom objects before parsing and recolouring, `auto_plot_data` = false (and connect Parse_Data and Recolour_Objects components). This allows picking a results field from `abbrevs` to parse without repeating the whole analysis, and using a Grasshopper Colour Gradient component on the canvas to generate colours. Connect a Grasshopper Legend component to plot a legend. To recolour Grasshopper geometry instead of Rhino Geometry (i.e. unbaked objects), connect the `Data` and `Geom` outputs to a Grasshopper Custom Preview component.

To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it); the `advanced` config string can then be saved to a `config.toml` file with an sDNA_GH Config component). Alternatively create an `advanced` config string manually. The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the and connect it to `advanced`. See the readme for the list of supported advanced config options.

Advanced config options for sDNA Integral

Option Default Description
startelev= Name of field to read start elevation from
endelev= Name of field to read end elevation from
metric= angular Metric – angular, euclidean, custom or one of the presets
radius= n List of radii separated by commas
startelev= Name of field to read start elevation from
endelev= Name of field to read end elevation from
origweight= Name of field to read origin weight from
destweight= Name of field to read destination weight from
origweightformula= Expression for origin weight (overrides origweight)
destweightformula= Expression for destination weight (overrides destweight)
weight= Name of field to read weight from. Applies weight field to both origins and destinations.
zonesums= Expressions to sum over zones (see zone sums below)
lenwt Specifies that weight field is per unit length
custommetric= Specified field name to read custom metric from
xytol= Manual override xy tolerance for fixing endpoint connectivity.
ztol= Manual override z tolerance for fixing endpoint connectivity.
outputgeodesics Output geometry of all pairwise geodesics in analysis (careful – this can create a lot of data)
outputdestinations Output geometry of all pairwise destinations in analysis (careful – this can create a lot of data). Useful in combination with origins for creating a map of distance/metric from a given origin.
outputhulls Output geometry of all convex hulls in analysis
outputnetradii Output geometry of all network radii in analysis

origins= Only compute selected origins (provide feature IDs as comma separated list). Useful in conjunction with outputgeodesicsm, outputdestinations, outputhulls, outputnetradii.
destinations= Only compute selected destinations (ditto)
nonetdata Don't output any network data (used in conjunction with geometry outputs)
pre= Prefix text of your choice to output column names
post= Postfix text of your choice to output column names
nobetweenness Don't calculate betweenness (saves a lot of time)
nojunctions Don't calculate junction measures (saves time)
nohull Don't calculate convex hull measures (saves time)
linkonly Only calculate individual link measures.
outputsums Output sum measures SAD, SCF etc as well as means MAD, MCF etc.
probroutes Output measures of problem routes – routes which exceed the length of the radius
forcecontorigin Force origin link to be handled in continuous space, even in a discrete analysis. Prevents odd results on very long links.
nqpdn= 1 Custom numerator power for NQPD equation
nqpdd= 1 Custom denominator power for NQPD equation
skipzeroweightorigins Skips calculation of any output measures for origins with zero weight. Saves a lot of time if many such origins exist.
skipzeroweightdestinations 1 Zero weight destinations are skipped by default. Note this will exclude them from geometry outputs; if this is not desired behaviour then set skipzeroweightdestinations=0
skiporiginifzero= Specified field name. If this field is zero, the origin will be skipped. Allows full customization of skipping origins.
skipfraction= 1 Set to value n, skips calculation for (n-1)/n origins. Effectively the increment value when looping over origins.
skipmod= 0 Chooses which origins are calculated if skipfraction?1. Effectively the initial value when looping over origins: every skipfractionth origin is computed starting with the skipmodth one.
nostrictnetworkcut Don't constrain geodesics to stay within radius. This will create a lot more 'problem routes'. Only alters behaviour of betweenness measures (not closeness).
probrouteaction= ignore Take special action for problem routes that exceed the radius by a factor greater than probroutethreshold. Can be set to ignore, discard or reroute. Reroute changes geodesic to shortest Euclidean path. Only alters betweenness output, not closeness.
probroutethreshold= 1.2 Threshold over which probrouteaction is taken. Note this does not affect computation of probroutes measures, which report on all routes which exceed the radius length regardless of this setting.
outputdecomposableonly output only measures which are decomposable i.e. can be summed over different origins (useful for parallelization)
linkcentretype= Angular for angular analysis, Euclidean otherwise Override link centre types – angular or Euclidean
lineformula= Formula for line metric in hybrid analysis (see below)
juncformula= 0 Formula for junction turn metric in hybrid analysis (see below)
bidir Output betweenness for each direction separately
oneway= Specified field name to read one way data from (see note 1 below)
vertoneway= Specified field name to read vertical one way data from (see note 1 below)
oversample= 1 Number of times to run the analysis; results given are the mean of all runs.

Useful for sampling hybrid metrics with random components.
odmatrix Read OD matrix from input tables (a 2d table must be present)
zonedist= euc Set expression to determine how zone weights are distributed over links in each zone, or 0 to skip distribution (all lines receive entire zone weight)
intermediates= Set expression for intermediate link filter. Geodesics are discarded unless they pass through link where expression is nonzero.
disable= Set expression to switch off links (links switched off when expression evaluates nonzero)
outputskim Output skim matrix file
skimorigzone Origin zone field (must be text) for skim matrix
skimdestzone Destination zone field (must be text) for skim matrix
skimzone Skim matrix zone field for both origin and destination (sets both skimorigzone and skimdestzone)
bandedradii Divide radius into bands: for each radius only include links outside the previous radius
datatokeep= List of field names for data to copy to output

sDNA_Skim (sDNASkim)

Skim Matrix outputs a table of inter-zonal mean distance (as defined by whichever sDNA Metric is chosen), allowing high spatial resolution sDNA models of accessibility to be fed into existing zone-base transport models.

sDNA_Int_From_OD (sDNAIntegralFromOD)

A simplified version of sDNA Integral geared towards use of an external Origin Destination matrix. Note that several other tools (including Integral) allow Origin Destination matrix input as well.

The file must be formatted correctly, see Creating a zone table or matrix file. All geodesic and destination weights are replaced by values read from the matrix. The matrix is defined between sets of zones; polylines must contain text fields to indicate their zone.

sDNA_Access_Map (sDNAAccessibilityMap)

This tool may produce an entire copy of the network for each origin, all in the same Data Tree layer - take care when interpreting plotted data.

Outputs accessibility maps for specific origins, including metric between each origin-destination, Euclidean path length and absolute diversion (difference between Euclidean path length and crow flight path length, similar to circuity, notated here as ‘Div’).

The accessibility map tool also allows a list of origin polyline IDs to be supplied (separated by commas). Leave this parameter blank to output maps for all origins. If outputting “maps” for multiple origins, these will be output in the same feature class as overlapping polylines. It may be necessary to split the result by origin link ID in order to display results correctly.

sDNA Accessibility Map is a different interface applied to sDNA Integral, so will in some cases accept its advanced config options as well. To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it). The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the `advanced` config string manually and connect it to `advanced`. See the readme for the list of supported advanced config options.

Preparation tools

sDNA_Prepare (sDNAPrepare)

Prepares spatial networks for analysis by checking and optionally repairing various kinds of error. Note that the functions offered by sDNA prepare are only a small subset of those needed for preparing networks. A good understanding of Network Preparation is needed, and other (free) tools can complement sDNA Prepare.

The errors fixed by sDNA Prepare are:

- endpoint near misses (XY and Z tolerance specify how close a near miss)
- duplicate lines
- traffic islands (requires traffic island field set to 0 for no island and 1 for island). Traffic island lines are straightened; if doing so creates duplicate lines then these are removed.
- split links. Note that fixing split links is no longer necessary as of sDNA 3.0 so this is not done by default.
- isolated systems.

To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it). The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the `advanced` config string manually and connect it to `advanced`. See the readme for the list of supported advanced config options.

Advanced config options for sDNA Prepare

Option Description startelev= Name of field to read start elevation from endelev= Name of field to read end elevation from island= Name of field to read traffic island information from. Anything other than zero will be treated as traffic island islandfieldstozero= Specifies additional data fields to set to zero when fixing traffic islands (used for e.g. origin or destination weights) data_unitlength= Specifies numeric data to be preserved by sDNA prepare (preserves values per unit length, averages when merging links) data_absolute= Specifies numeric data to be preserved by sDNA prepare (preserves absolute values, sums when merging links) data_text= Specifies text data to be preserved (merges if identical, concatenates with semicolon otherwise) xytol= Manual override xy tolerance for fixing endpoint connectivity ztol= Manual override z tolerance for fixing endpoint connectivity merge_if_identical= Specifies data fields which can only be merged if identical, i.e. split links will not be fixed if they differ (similar to 'dissolve' GIS operation)

sDNA_Line_Measures (sDNALineMeasures)

Individual Line Measures. Outputs connectivity, bearing, euclidean, angular and hybrid metrics for individual polylines. This tool can be useful for checking and debugging spatial networks. In particular, connectivity output can reveal geometry errors.

Geometric analysis tools

sDNA_Geodesics (sDNAGeodesics)

Outputs the geodesics (shortest paths) used by Integral Analysis. Geodesics can appear different to the original network so to make sure the changes are visible, ensure the pre existing network is hidden to avoid obscuring the new one.

The geodesics tool also allows a list of origin and destination polyline IDs to be supplied (separated by commas). Leave the origin or destination parameter blank to output geodesics for all origins or destinations. (Caution: this can produce a very large amount of data).

sDNA Geodesics is a different interface applied to sDNA Integral, so will in some cases accept its advanced config options as well. To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it). The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the `advanced` config string manually and connect it to `advanced`. See the readme for the list of supported advanced config options.

sDNA_Hulls (sDNAHulls)

Outputs the convex hulls of network radii used in Integral Analysis.

The convex hulls tool also allows a list of origin polyline IDs to be supplied (separated by commas). Leave this parameter blank to output hulls for all origins.

sDNA Convex Hulls is a different interface applied to sDNA Integral, so will in some cases accept its advanced config options as well. To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it). The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the `advanced` config string manually and connect it to `advanced`. See the readme for the list of supported advanced config options.

sDNA_Net_Radii (sDNANetRadii)

Outputs the network radii used in Integral Analysis. This tool will return a Data Tree in Geom of all the sub networks for each origin. The cluster component, Unpack_Network_Radii in sDNA/Extras can assist plotting and interpreting this.

The network radii tool also allows a list of origin polyline IDs to be supplied (separated by commas). Leave this parameter blank to output radii for all origins.

sDNA Network Radii is a different interface applied to sDNA Integral, so will in some cases accept its advanced config options as well. To use sDNA's advanced config options in sDNA_GH, add in an input Param to an sDNA component with the same name as each advanced config option you wish to include (omitting a trailing equals sign and leaving the Param unconnected, unless you wish to provide a value for it). The sDNA tools in that component will gather all user-specified input Params and construct the `advanced` config string from them. Alternatively, prepare the `advanced` config string manually and connect it to `advanced`. See the readme for the list of supported advanced config options.

Advanced config options for sDNA geometry tools

Calibration tools

sDNA_Learn (sDNALearn)

sDNA Learn selects the best model for predicting a target variable, then computes GEH and cross-validated R^2. If an output model file is set, the best model is saved and can be applied to fresh data using sDNA Predict.

Available methods for finding models are (valid options for `algorithm`):

- `Single best variable` - performs bivariate regression of target against all variables and picks single predictor with best cross-validated fit
- `Multiple variables` - regularized multivariate lasso regression
- `All variables` - regularized multivariate ridge regression (may not use all variables, but will usually use more than lasso regression)

Candidate predictor variables can either be entered as field names separated by commas, or alternatively as a *regular expression*. The latter follows Python regex syntax. A wildcard is expressed as `.*`, thus, `Bt.*` would test all Betweenness variables (which in abbreviated form begin with `Bt`) for correlation with the target.

Box-Cox transformations can be disabled, and the parameters for cross-validation can be changed.

Weighting lambda (`weightlambda`) weights data points by $y^{\lambda-1}$, where y is the target variable. Setting to 1 gives unweighted regression. Setting to around 0.7 can encourage selection of a model with better GEH statistic, when used with traffic count data. Setting to 0 is somewhat analogous to using a log link function to handle Poisson distributed residuals, while preserving the model structure as a linear sum of predictors. Depending on what you read, the literature can treat traffic count data as either normally or Poisson distributed, so something in between the two is probably safest.

Ridge and Lasso regression can cope with multicollinear predictor variables, as is common in spatial network models. The techniques can be interpreted as frequentist (adding a penalty term to prevent overfit); Bayesian (imposing a hyperprior on coefficient values); or a mild form of entropy maximization (that limits itself in the case of overspecified models). More generally it's a machine learning technique that is tuned using cross-validation. The r^2 values reported by learn are always cross-validated, giving a built-in test of effectiveness in making predictions.

Regularization Lambda allows manual input of the minimum and maximum values for regularization parameter λ in ridge and lasso regression. Enter two values separated by a comma. If this field is left blank, the software attempts to guess a suitable range, but is not always correct. If you are familiar with the theory of regularized regression you may wish to inspect a plot of cross validated r^2 against λ to see what is going on. The data to do this is saved with the output model file (if specified), with extension `.regcurve.csv`.

sDNA_Predict (sDNAPredict)

Predict takes an output model file from sDNA Learn, and applies it to fresh data.

For example, suppose we wish to calibrate a traffic model, using measured traffic flows at a small number of points on the network:

- First run a Betweenness analysis at a number of radii using Integral Analysis.
- Use a GIS spatial join to join Betweenness variables (the output of Integral) to the measured traffic flows.
- Run Learn on the joined data to select the best variable for predicting flows (where measured).
- Run Predict on the output of Integral to estimate traffic flow for all unmeasured polylines.

Dev tool(s)

Unload_sDNA (Unload_sDNA)

Unload the sDNA_GH Python package and all sDNA modules, by removing them from Grasshopper Python's shared cache (sys.modules).

The next sDNA_GH component to run after this one (that's not also an Unload_sDNA) will then reload the sDNA_GH Python package and installation-wide options file (config.toml), and any specified options including a project specific config.toml, without otherwise having to restart Rhino to clear Grasshopper's cache. sDNA tools will also try to load an sDNA installation.

Self_test (selftest)

Runs the unit tests of the sDNA_GH module and launcher.py.

Not a tool in the same sense as the others (this has no tool function in sDNA). The name `Self_test` (and variations to case and spacing) are recognised by the launcher code, not the main package tools factory. In a component named "Self_test", the launcher will cache it, then replace the normal RunScript method in a Grasshopper component class entirely, with a function (`unit_tests_sDNA_GH.run_launcher_tests`) that runs all the package's unit tests (using the Python unittest module). Unit tests of the functions in the launcher, can also be added to the launcher code.

sDNA_GH_API_test_xxxx

These components are not formally provided with releases (except inside `src\sDNA_GH\Rhino_8_API_tests.gh` - to debug this file add `'CHEETAH_GH_NON_INTERACTIVE' : 'False'` to the dict set to the kwarg `extra_env_vars` in `hatch_build.py`). But for developers, launcher components whose names start with `sDNA_GH_API_test_` (defined in `launcher.py:APITEST_PREFIX`), named `sDNA_GH_API_test_xxxx` will run API test "xxxx". "xxxx" can also be "all" to run all the API tests.

Advanced Usage.

Automatic multi-tools.

Each sDNA tool has its own Grasshopper component. To run a tool, a true value, e.g. from a Boolean toggle component, must be connected to its component's `go` Input Param [^note]. To group together common work flows, if an `auto_` option is set to true, some components also automatically run other tools before and after they run their own tools (if components for those tools are not connected to it). For example, this allows an entire sDNA process to be run on Rhino Geometry from a single sDNA tool component.
When an sDNA_GH component is first placed on the canvas, or a grasshopper file with an sDNA_GH component on the canvas is first loaded, each component adds in Params for all its required Input and Output arguments (if the Params are not already present). These added Params include those of any extra automatically added tools if an `auto_` option is true, that would other wise require separate components. Extra customisation can be carried out by adding in user specified Params too, that have the correct name of a supported option. Similarly any Params not being specified can be removed.

[note] The Config component tool always loads its options when placed or its Inputs are

updated for any value of `go`. On the Unload_sDNA component, `unload` does the same thing as `go`.

Running individual tools.

Multiple sDNA_GH components can be chained together to run in sequence by connecting the `OK` Output Param of one component, to the `go` Input Param of the component(s) to be run afterwards. A Grasshopper Colour Gradient tool can be connected in between a Parse_Data component and Recolour_Objects component.

Component Execution Order.

Multiple sDNA_GH tools can be run from a single sDNA_GH component by setting any of the `auto_` options to true: `auto_get_Geom`, `auto_read_User_Text`, `auto_write_Shp`, `auto_read_Shp` and `auto_plot_data`, on a Config component, before placing the chosen sDNA_GH tool on the canvas (as long as components for those tools are not connected to it).
Warning! If you did not create a `config.toml` file (in Installation step 13 above), and if you rely instead on Config components inside your .gh file itself to set option values, immediately before saving your .gh file, be sure to select the Config component determining any `auto_` options, and press `Ctrl` + `B` (or from the pull-down menu select `Edit` -> `Arrange` -> `Put To Back`) to to send to the back, any components that should run first when you reload the .gh file. This ensures the Config component will run before other sDNA_GH components, which rely on settings controlled by it to configure themselves correctly.

Options.

sDNA_GH is highly customisable. This customisation is controlled by setting options. To give an option a value, connect a Grasshopper Param or text panel containing that value, to the Param with that option name on any sDNA_GH component (except Unload_sDNA). If a Param is subsequently disconnected, its latest value will be remembered. Some Text Params can be cleared by connecting an empty Text Panel to them. Any option in a component can be read by adding an Output Param and renaming it to the name of the option. Options whose names are all in capitals are read only. Any other option in a component can be changed by adding an Input Param and renaming it to the name of the option, and connecting it to the new value. Entire options data structures (`opts`) may also be passed in from other sDNA_GH components as well, via normal Grasshopper connections.

Adding Component Input and Output Params.

To add a new Input or Output Param, zoom in on the component until plus and minus symbols can be seen between the params. Click on the plus symbol where you want the new Param. Right click the new Param's name (e.g. x, y or z for an Input Param, or a,b or c for an Output Param) to rename it to name of the desired option you wish to set.

Logging options

If `log_file` is not an empty string, a log file is created before Input Params are created on components, so to have any effect, the options `propagate`, `working_folder`, `logs_dir` and `log_file` must be set in an installation wide config.toml file, e.g. by setting those options and `go` to true on Config component, and restarting Rhino. Logging levels can be changed dynamically. Supported values for `log_file_level` and `log_console_level` are: `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. To suppress output from the `out` Output param entirely, set `log_console_level` to `CRITICAL`. To propagate log messages higher than the sDNA_GH package's parent logger (which will most likely result in a lot of output), set `propagate` to true.

Options override priority order

1. The component input Param options override options in a project specific options file (`config`).
2. A project specific options file (specified in `config`) overrides options from another sDNA_GH component (connected in `opts`).
3. Options from another sDNA_GH component override the installation-wide options file (e.g. `%appdata%\Grasshopper\UserObjects\sdna-gh\sDNA_GH\config.toml`).
4. On start up only (and every time thereafter if `sync = false`, `no_state = true` and `read_only = false`) the installation-wide options file overrides the sDNA_GH hard-coded default options in `main.py` [^note] [note] Dev note: the options in `main.py` themselves override every individual tool's default options in `tools.py`.

Local meta options.

By default all sDNA_GH components, across all Grasshopper canvases in the same Rhino process, share (and may change) the same global `opts` (module options, tool options, and *meta* options) in the `main.py` module. If only one of each tool is needed (and there is only one version of sDNA), that will suffice for most users.

To give an sDNA_GH component different options to another of the same type using the shared global options, it must de-synchronise from the global options. De-syncing occurs if a component's *local meta option* `sync` is false (if `read_only` is true, it will still read from but not update the global options). *local metas* are like any other option, except a) they are shared using `l_metas` instead of `opts`, and b) they are not updated automatically from the global module options (as this would defeat their entire purpose).

Components referring to the global options share their settings. This means they do not know if a particular setting in the global options came from a different component sharing with it, or from itself previously. Shared states mean even a component in isolation has its own historical state. To use components without any options sharing and a minimum of state, set `sync` to false and `no_state` to true (its default).

Shared state vs desynchronised components.

sDNA_GH has two fundamentally different modes of operation controlled by the Boolean `sync` local meta option: synchronised and desynchronised. Local meta options of components can only be changed directly on each component via user Params, by project options files (`config` params) or by the installation wide options file (`config.toml`). By default, components are desynchronised (`sync` = false). A few useful features are available from synchronised components, but they may behave in unexpected ways, and may even feel buggy and glitchy. It is recommended to undertake a one off setup process to save your commonly used custom options in an installation wide options file (`config.toml`). This is necessary to run sDNA if sDNA_GH fails to automatically find sDNA or the Python run time; even if this is successful but takes a while, it is still a good idea. Thereafter all components can be used as desynchronised:

Synchronisation (Advanced user only)

To set all components to synchronised, set `sync` = true on a Config component, leave `save_to` unconnected, and set `go` = true. This writes `sync` = true to the installation wide options file (`config.toml`). As long as no higher priority options source (Params, project specific options files, or local metas from other components) sets `sync` = false, all desynchronised components will then resynchronise the next time they run.

Applying synchronisation changes.

Finally, one of three alternatives is necessary to make these changes take effect, as synchronised components only read the installation wide options file on start up: a) Restart Rhino, b) Set `unload = true` on an Unload_sDNA component, set `unload = false` on it immediately afterwards, then manually re-initialise each component (double click its name or icon to see its source code and click OK, or delete it and replace it from the sDNA_GH ribbon). c) Set the `config` on each component to be resynchronised to the file path of the installation wide options file (e.g. `%appdata%\Grasshopper\UserObjects\sdna-gh\sDNA_GH\config.toml` - expand `%appdata%` in a File Explorer).

De-synchronisation.

Components are already desynchronised by default. Global de-synchronisation is only necessary if you have already synchronised them, e.g.to save options from other components to a `config.toml` file, and wish to undo this.

To desynchronise all synchronised components, set `sync` = false on a Config component, leave `save_to` unconnected, and set `go` = true. This writes `sync` = false to the sDNA_GH `config.toml`.

Apply the synchronisation changes as in the previous subsection.

Desynchronised components (`sync` = false):

-must have any `auto_` options directly set on each of them. A config component cannot be used for this. -Plot min and plot max will be automatically calculated anew each time, if viewing multiple results fields. -Input params will be revert to their defaults when disconnected. -Config components cannot affect the behaviour of desynchronised components, or save their options to `config.toml` files.
-If sDNA components are run with `auto_write_Shp` = false or `auto_read_Shp` = false, deletion of temporary files requires opts to be connected between the sDNA component and write_Shp and read_Shp component. -Read all the defaults in the override order on every run, so respond dynamically to changes in the installation wide `config.toml` options file. -Support multiple project specific `.toml` files. -Require the `.toml` file to be set on a `config` Input on every desynchronised component that refers to it.

Synchronised components (sync = true):

-are affected by relevant `auto_` rules set on a Config component.
-can have their options saved to `config.toml` files. -only require one project specific `.toml` file to be set to a `config` Input between all of them. -can save automatically created advanced config strings (the `advanced` option) to `.toml` files by sharing them with a config component, and thence saved to file. -only once read the installation wide `config.toml` file user options file, when the first sDNA_GH component is placed (synchronised or desynchronised).
-in one `.gh` file are affected by the global options set by other synchronised sDNA_GH components in previous `.gh` files opened in the same Rhino session.

The following Inputs and Outputs are never shared (unless connected by the user in Grasshopper): `OK`, `go`, `Geom`, `Data`, `file`, `input`, `output`, `gdm`.

License.

See license.md

Copyright.

Cardiff University 2022

Contact.

grasshopper.sdna@gmail.com

Developer manual.

Dependencies.

[IronPyShp (MIT License)](IronPyShp) v2.3.1, an Iron Python cross-port of Joel Lawhead and Karim

Baghat et al's PyShp.

toml_tools (MIT License) v2.0.0, a Python 2 back-port and Iron Python cross-port of Taneli Hukkinen's tomli (behind tomllib in Python 3.11 and later) and tomli_w, with a few small extras.

mapclassif-Iron (BSD 3-Clause License), a stripped down minimal fork of mapclassify, only containing a pure Python Fisher-Jenks classifier.

Cheetah_GH (MIT License). A framework that can place, virtually connect, and runs Grasshopper components to the canvas, in code (e.g. test code). Also runs Grasshopper definitions from the command line, pipes output from the internal Grasshopper env run using Cheetah_GH (e.g. unittest) back to the command line, and if few changes are made to the Grasshopper definition, quit Rhino and return to the command line.

Anteater_GH (MIT License). Fuzz testing helper functions for testing within Grasshopper and Rhino.

Testing.

Unit tests.

Most of the code in sDNA_GH is coupled with the APIs of sDNA and Rhino and Grasshopper, and does not carry out calculations of significant complexity (to avoid unnecessarily slowing the user's machine).
However the code in data_cruncher.py is a little more complex, and so is important to test. Its unit tests can be run within Grasshopper by placing a self_test component (with a test tube icon under Extras).

API Tests.

The API Tests can be run locally from within Rhino by opening `test_cases\Rhino_8_API_tests.gh`. They can also be run locally from the cmd command line by running `Cheetah_GH .\src\sDNA_GH\Rhino_8_API_tests.gh NUM_TESTS 5` (Python >= 3.8 and Cheetah_GH required in the test env). Cheetah_GH will listen over UDP for 'stderr' output from the unittest process Cheetah is also running within Rhino/ It also attempts to return an error code indicating success or failure of the tests. Test output is also saved to `src\sDNA_GH\sDNA_GH_unit_test_results.log`.

If RhinoCompute allows command line access, this could potentially be developed into a cloud CI pipeline.

Iteration.

When working on the source tree outside of Rhino, if no changes to `.\launcher.py` have been made (that would necessitate running `.\build_components.bat`), then `.`

`\create_install_and_test_release.bat` can be run to create a release from the current source tree (and whatever component launcher files are there from a previous build), install it, and run the API tests on it.

Other Tests.

Grasshopper is a visual programming language. It has been challenging to run a test framework within it it from a local command line, let alone a modern CI/CD system. So unfortunately all other test Grasshopper definitions must be opened manually. They are included as examples for the end user.

Contributions.

Thankyou for your interest in contributing. If you are considering writing code for inclusion in the main sDNA_GH project, please reach out to us first. More details below.

Dependencies.

Contributions to toml_tools, IronPyShp, and mapclassif-Iron that satisfy the requirements of their copyright holders are welcome.
These forks were created after the original release of sDNA_GH. Their copyright is still owned by the same owners as their parent projects. James maintains these forks. toml_tools in particular has high test coverage (thanks to tomli and tomli_w's tests), and new features are being proposed for TOML, so is particularly suitable for future development.

sDNA_GH

Rhino/Grasshopper Python development is a plentiful source of bugs. Contributions to sDNA_GH must not fail any regression tests. Unfortunately many other tests (in '/test_cases', other than the Unit tests and API tests mentioned above) are Grasshopper definitions that must each be run manually. Automating the remainder of these tests in some way, e.g. also using Cheetah_GH would be an excellent and most welcome contribution. Very simple contributions may be accepted on a discretionary basis, e.g. that add entries to the dictionaries the factories refer to, that would let Read_Shp and Write_Shp support Points shapefile types. But beware the Contributors may also need to satisfy IP transfer requirements (t.b.c.) of the copyright holder, for Cardiff University to be able to include them.

Build instructions.

A Python env is needed with pyproject-build (build) installed, to kick off the whole show. Build will create a venv for sDNA_GH's Python deps from PyPi. An an sDNA installation is also needed, so that the build process knows which sDNA components to build (otherwise the user must build their own using a Config component). The sDNA env can now be any venv in which sdna_plus is installed from PyPi. If so, when using sDNA_GH, `python` needs to be the venv's Python for the sDNA Python subprocess to find PyShp.

Build env.

- Install build either as a tool (in its own venv automatically, e.g. via `uv tool install build`), or in the currently activated Python environment (best practise is a venv, but pyproject-build creates venvs itself for its builds by default anyway, so I use a global Python env on Windows).

- Install sdna_open or sdna_plus. This is solely to get copies of `sDNAUIspec.py` and `runsdnacommand.py` for the builder to know which sDNA tools to build components for. If sDNA is in a non-standard location, if the sDNA_GH builder cannot find sDNA automatically (via a non-exhaustive list of hardcoded directories), it will the path of the parent dir of those two files specifiying in `sDNA_paths`.

- Make a local clone of sDNA_GH (at the branch and commit a rlease is to be built from).

Steps.

0. The deps are installed in the build environment, and the path to these deps in the build environment (`sys.path[-1]`) is passed by the build hook (`hatch_build.py`) into the builder tool running within Rhino and Grasshopper (so to simply run `dev\sDNA_build_components.gh` as before, the environment variable SDNA_GH_BUILD_DEPS must be set).
1. If sDNA_GH has not automatically found the sDNA installation you wish to build components for, place a Config component (the one with a lightbulb icon) and add its path to `sDNA_paths`.
2. In the repository's root dir, run `build_components.bat` (if necessary open it and adjust the paths to your local folders, and the paths in `\dev\sDNA_build_components.gh`).
3. For non-Github users, a good quality pdf of this file (`README.md`) can be created in VS Code with the extension: print, PD Consulting VS Marketplace Link. This will render the markdown file in your web browser. Print it to a pdf with the name `README.pdf` in the same directory (using Save to Pdf in Mozilla instead of Microsoft Print to Pdf will preserve the URLs in the links).
4. Manually create `Unload_sDNA_GH` and `Readme.txt` components if required (the latter can use `readme_component.gh`; the former is in `sDNA_build_components.gh`), and the cluster components in `Shp_Data_Tree_component.gh` and `unpack_network_radii.gh`. Copy these to `\sDNA_GH\components\manually_built`.
5. Run `create_release_sdna-gh.zip.bat.bat` to create the zip file for release.
6. Note: The components are only GhPython launchers with different names and different docstrings. As much code as possible has been shifted into the python package and the other sDNA_GH Python package files. If no changes to the launcher code have been made and no new components/tools are required, a new release can simply reuse the

.ghuser files from an old release, and the new release's zip files can be created simply by re running `create_release_sdna-gh.zip.bat.bat`.

To build new sDNA components.

sDNA_GH will attempt to automatically build components and user objects for the sDNA tools in an `sDNAUISpec.py`, that it doesn't already have .ghuser GH component / User Object file for. It will also look for an .png icon file with the same name as the tool Class in sDNAUISPec in `\sDNA_GH\components\icons`, and will parse this very file (`README.md`) for a tool description, to swap in as the launcher code's docstring (which will become the User Object and component descriptions, and its mouse over text). Therefore:

- for each new component: Add a description to `%appdata%\Grasshopper\UserObjects\sdna-gh\sDNA_GH\README.md` starting on the line after (`tool Class name`) in brackets, ending in two blank lines. Save it to `README.md` (overwriting the previous one in the User Objects folder).
- for each new component: Prepare an icon file and save it to `%appdata%\Grasshopper\UserObjects\sdna-gh\sDNA_GH\components\icons`. 24x24 is recommended by the Grasshopper developers, but it seems fairly flexible - see sDNA_Integral. A format compatible with .Net's `System.Drawing.Bitmap` Class is required. `.png` has been tested.
- Open a new Grasshopper canvas with sDNA_GH installed.
- Place an sDNA_GH Config component.
- Setup sDNA_GH to use the new version of sDNA by specifying it in `sDNA_folders` (following Installation step 13 above).
- Ensure `make_new_comps` is true.
- If necessary Recompute the sheet - press F5.
- The new user objects for the new components will be automatically created, and added to the sDNA section of the Grasshopper Plug-ins Ribbon. Copy the relevant `.ghuser` file(s) from `%appdata%\Grasshopper\UserObjects\`, and paste them in `\sDNA_GH\components\automatically_built` in the repo. Place copies of the updated README.md file and new icon files in there too for posterity. If wished, move the new `.ghuser` file(s) from `%appdata%\Grasshopper\UserObjects\` to `%appdata%\Grasshopper\UserObjects\sdna-gh\sDNA_GH\components\automatically_built`

The supported data types for inputs (forced to lower case) are in sDNA_ToolWrapper.sDNA_types_to_Params in tools.py:

- fc = Param_FilePath
- ofc = Param_FilePath
- bool = Param_Boolean
- field = Param_String
- text = Param_String
- multiinfile = Param_FilePath
- infile = Param_FilePath

- outfile = Param_FilePath

Misc

To compile C# code to a grasshopper assembly (.gha file): Install Visual Studio 2017 community edition with VB / C# / .Net workflow [https://developer.rhino3d.com/guides/grasshopper/installing-tools-windows/#fnref:3] Install Rhino & templates as above [https://developercommunity.visualstudio.com/t/net-framework-48-sdk-and-targeting-pack-in-visual/580235] Install .Net v4.8. Change .csproj target to v4.8 [https://stackoverflow.com/questions/58000123/visual-studio-cant-target-net-framework-4-8]

GHPython for .ghuser: Select GHPython component. Optionally compile to .ghpy. File -> Create User Object

[^0] The the old builds of sDNA_open require the 64 bit (x64) Visual Studio 2008 redistributable. It is available here or here.

[^1] The entire source code for sDNA_GH is visible on Github. All the source code is also visible in the download itself as the component launcher and Python package is uncompiled, except the .ghuser files which each contain the launcher code under a different name, and are compiled. See the Build Instructions above to build them for yourself from the source code.