sDNA_GH

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.

sDNA

sDNA is able to calculate Betweenness, Closeness, Angular distance, and many other quantities including custom hybrid metrics, and is able to perform many other advanced functions as well. Please note, for results of a network analysis to be meaningful, it must be ensured that the network is first properly prepared.

sDNA_GH functionality

sDNA_GH:

- Reads a network's polyline Geometry from Rhino or Grasshopper, and Data from any User Text on it.
- Writes the network polylines (formed by one or more polylines) and user Data to a Shapefile.
- Initiates an sDNA tool that processes that shapefile, and e.g. carries out a network preparation or an analysis.
- Reads the shapefile produced by the sDNA tool.
- Displays the results from sDNA by colouring a new layer of new polylines or the original ones.

User manual.

System Requirements.

Software

1. Windows 10 or 8.1 (not tested in Windows 11)
2. Python 2.7. Please note Iron Python 2.7 does not run sDNA correctly (as incorrect shapefiles are 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.

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

2. Ensure you have an installation of Python 2.7 [^0] that can run sDNA correctly from the command line. sDNA_GH runs sDNA from the command line. Command line use of sDNA has been tested with Python versions 2.6 and 2.7 . Do not run sDNA with Iron Python 2.7, as invalid shape files may be produced (it is not possible to access the Iron Python shipped with Rhino from the command line, in any case).

3. To use sDNA with sDNA_GH, ensure you have an installation of sDNA. sDNA itself may require the 64 bit Visual Studio 2008 redistributable, available here or here).

4. Download `sDNA_GH.zip` from food4Rhino or the sDNA_GH releases page on Github.

5. 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 the zip archive. If any files still need to be unblocked, a PowerShell script is provided in the zip file: `\sDNA_GH\dev_tools\batch_files\unblock_all_files_powershell.bat`[^2] Please do not automatically trust and unblock all software downloaded from anywhere on the internet [^1].

6. Open Rhino and Grasshopper.

7. 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 not the Components Folder used by many other plug-ins (i.e. not `%appdata%\Grasshopper\Libraries`).

8. Copy in sDNA_GH.zip to this folder (e.g. it should be at `%appdata%\Grasshopper\UserObjects\sDNA_GH.zip`).

9. 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 single subfolder called sDNA_GH should have been created.

10. Restart Rhino and Grasshopper.

11. The sDNA_GH plug in components should now be available under a new "sDNA_GH" tab in the ribbon tabs amongst any other plug-ins installed (right of `Mesh`, `Intersect`, `Transform` and `Display` etc).

12. To use sDNA with sDNA_GH, if no preferences are specified, sDNA_GH will search for sDNA and Python 2.7 installations automatically, using the first one of each it finds. However to be sure sDNA_GH uses a particular version of sDNA and the correct Python 2.7 interpreter(and to be sure it can find one at all) it is recommended (on first usage only) to: -place a Config component on the canvas (the component with a lightbulb icon, under Support Tools). -Specify the file path of the sDNA folder (containing sDNAUISpec.py and runsdnacommand.py) of the sDNA you wish to use in the `sDNA_folders` input. -Specify the file path of the Python 2.7 interpreter's main executable in the `python` input. -Specify any other options you wish to save and reuse on all projects, if necessary by adding custom input Params with the 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 to other project specific `config.toml` files, specify the file path in save_to and repeat the previous 4 sub steps.

13. For a first test of sDNA_GH using sDNA, open `\sDNA_GH\tests\5x18_random_grid_network.3dm` (in the previously unzipped folder in the User Objects folder), place an sDNA_Integral component and connect a True boolean toggle to its `go`.

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.

Components.

Automatic multi-tools.

Each sDNA tool has its own a 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 (unless an `auto_` option is set not to) some tools by default also automatically run other tools before or after they run themselves. For example, this allows an entire sDNA process to be run on Rhino Geometry, with the results from the sDNA calculation being used to recolour the Rhino geometry, from one single sDNA tool component. When placed on the canvas, each component adds in Params for all its required Input and Output arguments (if the Params are not already present), including those of any extra automatically added tools. Extra customisation can be carried out by adding in extra Params too. Such extra user added Params are not removed. This can make the components quite large, but any Params not being specified can be removed. Another way to make the components smaller, that also allows any other custom work to be carried out in between individual tools, is to set the `auto_` options for that component to `false`. A newly placed component will then only add in the Params for that component's own tool.

[note] All except the Config tool which always loads its options when placed or its Inputs are updated (saving options when `go` is true), and Unload_sDNA on which `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. To work with a Grasshopper Colour Gradient tool, to show fewer Input and Output Params on each component, or to customise sDNA_GH behaviour, e.g. to connect in different inputs and outputs between individual tools running, advanced users may prefer to run only one tool per sDNA_GH component. To do this, simply 'turn off' the `auto_` options: `auto_get_Geom`, `auto_read_User_Text`, `auto_write_Shp`, `auto_read_Shp` and `auto_plot_data` by setting them to `false`, e.g. on a Config component. How to do this is described below in more detail.

Component Execution Order.

Warning! If you did not create a `config.toml` file (in Installation step 12 above), and if you rely instead on components inside your .gh file itself to set option values, e.g. `Config` components to determine options, be sure to select 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 whn you reload the .gh file. E.g. send to the back all components that set options values that you wish other components to later depend on. In particular if you are running with `auto_` options set to false, saved single-tool components will rebuild themselves as multi-tool components when a .gh file is loaded, if sDNA_GH's hard-coded default options are read before your custom options. As even if these were set and read correctly, they are on a component that hasn't run yet, that Grasshopper doesn't know about yet.

Errors.

Errors that occur while the components import the main sDNA_GH Python package and when they initialise themselves must be cleared by deleting the component and placing a new one. It may also be necessary to first unload the sDNA_GH package using the unload component, to clear sDNA_GH from GPython's cached modules, and force it to be reimported.

Options.

sDNA_GH is highly customisable. This customisation is controlled by setting options. Any option in a component can be read by adding an Output Param and renaming it to the name of the option. Similarly, any 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 components as well, via Grasshopper connections.

Adding Component Input and Output Params.

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

Logging options

When the first sDNA_GH component is placed (or when the first one runs when a .gh file is loaded) you may notice a slight delay. This is because the main sDNA_GH code base is being imported from the installation sub folder in the User Objects folder. Subsequent components simply link to it once it has already been imported. This happens before a component knows what its input Params are. The installation wide options file is read and the main logger for trouble shooting is created during this first import. Therefore unlike other options, custom logger options for configuring the logger (e.g. making it less verbose by raising the logging level from `DEBUG`) must be set in the installation wide options file (`config.toml`). Like other options, logger options are overridden by other files, components and Input Params afterwards. Only as the logger has already been set up by the time the Input Params are created, it's simply too late by then for changes to logger options (made using what are normally the higher priority channels) to have any effect. Supported values for logging levels are: `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`.

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.
3. Options from another sDNA_GH component override the installation-wide options file (e.g. `%appdata%\Grasshopper\UserObjects\sDNA_GH\config.toml`).
4. 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`.

Name map (abbreviations, NickNames and work flows)

After `config`, the next most important meta option is `name_map`, in which custom NickNames for user-created sDNA_GH components (and entire work sequences of tools) may be defined.

Component NickName.

The particular tool or tools a module runs is controlled by its NickName (accessible in the local meta option NickName), but which can be changed by simply renaming the component. The component then looks up in the meta option name_map to see which (if any) tool or tools (including other NickNames) its NickName corresponds to, then retrieves these tools from the cache (building them if they do not exist already). sDNA tools automatically add their own default options and syntax in their initialiser (and add new ones if their NickName or the version of sDNA subsequently changes).

Tool options.

Each NickName in name map creates a new set of options. These contain options for each tool (real name) in the list of tools used under that NickName. Each of these has a set of options for each version of sDNA encountered by the component so far. Primarily this is where the settings for sDNA wrapper tools are stored (apart from a couple of helper overrides, like `file`). Support tools and sDNA_GH tools use options and meta options in the common name root space (which is the same for all sDNA versions sDNA_GH has found).

Local meta options.

By default all sDNA_GH components share (and may change) the same global dictionary of options (module options, tool options, and *meta* options, together in *opts*) 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.

For advanced users, each component with a given NickName in name_map also has its own set of tool options (one for each version of sDNA). However, for one support component to have a different set of options to another, one of them must no longer update the global options dictionary - it must de-synchronise from them. This syncing and de-syncing is controlled by each component's individual *local meta options*, (in `local_metas`) `sync_to_module_opts`, `read_from_shared_global_opts`. By default these booleans are both equal to True. More than one project specific options file (`config.toml`) can then be supported on the same canvas. Just create the files, and specify the name of the one you wish to be used on the `config` input of the desired components (or connect their `opts` to one with this `config`). Just like other options, *local metas* can be set in Input Params, read from Output Params, shared between components via Grasshopper connections in the same way as opts, can be set by project config files(`config.toml`) and set in the installation-wide config file (e.g. `%appdata%\Grasshopper\UserObjects\sDNA_GH\config.toml`). But unlike *meta options*, *local metas* are not updated automatically by syncing to the main module options (as this would defeat their entire purpose).

Tools.

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** Accepts a data tree of keys and values. **Geom** Accepts a list of geometric objects (Rhino or Grasshopper). Data trees of objects need to be flattened. **gdm** Accepts a Geometry-data-mapping, a python nested dictionary. The keys are the IDs 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.

Support tools

Config (config)

To guarantee this component runs before all others when your .gh file is loaded, ensuring your options are loaded, select it and press Ctrl + B before saving. 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.

Read_Geom (get_Geom)

Reads in references to Rhino geometry (polylines) to provide them in the required form for subsequent sDNA_GH tools. Set the option `selected` to true, to only read selected Rhino objects (of the specified type - polylines). Similarly, specify `layer` = your_layer_name to only read Rhino objects from the layer named your_layer_name. To go back to selecting all layers, set `layer` to any value that is not the name of a layer.

The UUIDs of the Rhino objects are converted to strings to preserve the references to them.

Shapefile tools

Write_Shp (write_shapefile)

Writes the DataTree in `Data` and list of `Geom`etric objects (polylines) to a shapefile. If not specified in `file`, a file name based on the Rhino doc or Grasshopper doc name is used (unless `auto_update_Rhino_doc_path = false`). Overwrites existing files, or creates new files. 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 it will first call read_User Text (unless `auto_read_User_Text` is false). 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 (key name if it originated as User Text) matches the template string specified in `input_key_str`. To write all data with any key name to the Shapefile, set it to `{all}`.

Read_Shp (read_shapefile)

Reads in polylines and associated data records from a shapefile. Creates new objects unless existing objects are specified in `Geom`. Specify the path of the .shp file to read in `file`. **WARNING! Read_Shp automatically deletes the shapefile after reading it in,if `strict_no_del` = false, if either a) the file name matches the pattern in either `output_fmt` or `prepped_fmt`, or b) if `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, only the data is read from the shapefile. Otherwise the shapes in the shapefile are outputted as new Grasshopper Geometry objects. The bounding box output `bbox` is provided to create a legend frame within Recolour_Objects. 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 (unless `auto_plot_data = false`), 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. Be sure to supply the list of the data's associated geometric objects (in `Geom`), as legend tags and parsed values are appended to the output `Data` list and `Geom` list. 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`). See the readme for further details.

Data Tree A Data Tree connected to `Data`'s first level should be 2 branches deep. The first level should contain a branch {n;0} for each geometric element n in the corresponding list connected to `Geom` . Each of these top level branches should themselves contain a branch with only two items: keys {n;0} and values {n;1}. The two nodes of this structure should be a pair of corresponding (equal length) lists; a list of 'keys' or field names, and a list of 'values' corresponding to the actual numerical data items for that field, for that geometric object. The mth key and value of the nth geometric object should be {n;0}[m] and {n;1}[m] respectively. Read_Shp supplies a Data Tree in this required format, if the data is read from User Text or a Shapefile. Grasshopper's path tools can be used to adjust compatible Data Trees into this format.

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'.

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). 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 provided Rhino objects. If no Geometry is provided in `Geom`, Read_From_Rhino is first called (unless `auto_get_Geom` = false).

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

Analysis tools

All the sDNA tools are run from the command line, using the Python interpreter in `python`. By default an sDNA tool component will show all the possible inputs on its input Params. To show only the essential ones instead (and make the components a lot smaller) set `show_all` = false.
All sDNA_GH components attempt to check, e.g. if any Write_User Text or Read_User Text components are already connected to its inputs (upstream) or outputs (downstream), before running extra tools. But if it is unconnected and run with all `auto_` options on, an sDNA component will take in geometry from Rhino directly, write it to a shapefile, run the analysis, read in the output shapefile, and recolour the Rhino polylines. Otherwise, sDNA tools that require input must have the path to the shapefile you wish to run the analysis on specified in `file` or `input`. **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).**
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)

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.

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.

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 GhPython's shared cache (sys.modules).

The next non-Unload_sDNA sDNA_GH component to run after this one will then reload the 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_General (sDNA_General)

Run any other component by feeding the name of it into the "tool" input param.

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.

License.

See license.md

Copyright.

Cardiff University 2022

Contact.

grasshopper.sdna@gmail.com

Developer manual.

Dependencies.

To bulk unblock files, to avoid unblocking every file manually it is necessary to install Powershell. Otherwise no additional dependencies are required. sDNA_GH is shipped with files from the following python packages included: PyShp (MIT License) "version: 2.2.0" Toml (MIT License) Latest commit 230f0c9 on 30 Oct 2020

Build instructions.

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. 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.
5. Run `create_release_sDNA_GH_zip.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`.

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 this file, `README.md` starting on the line

after (`tool Class name`) in brackets, ending in two blank lines to this very file. Save it to `%appdata%\Grasshopper\UserObjects\sDNA_GH\README.md` (overwriting the previous one).
- for each new component: Prepare an icon file and save it to `%appdata%\Grasshopper\UserObjects\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 12 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.

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 Python 2.7 download can be verified using this certificate and Gpg4win.

[^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 visible, except the .ghuser files which each contain the launcher code under a different name, and are compiled. It is a little repetitive, but see the Build Instructions above to build them for yourself from the source code.

[^2] This script is largely code from Ed Wilson of Microsoft's Dev Blog or try this alternative method)