# 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 Grashopper, and Data from any Usertext 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. 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 ensure sDNA_GH uses the desired version of sDNA and Python 2.7 (and to be sure it can find one at all): -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, or to update the installation wide user options file, specify the file path in save_to and repeat the previous 4 sub steps.
13. For a first test of sDNA_GH, open `\sDNA_GH\tests\5x18_random_grid_network.3dm` (in the folder from the unzip, in the User Objects folder), place an sDNA_Integral component and connect a True boolean toggle to its `go`.

# 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_GH 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 conect in different inputs and outputs between individual tools running, advanced users may prefer to run only one tool at atime turn off any of the `auto_` options: `auto_get_Geom`, `auto_read_Usertext`, `auto_write_Shp`, `auto_read_Shp` and `auto_plot_data` by setting to `false`. How to do this is described below.

## 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 InputParams 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 desynchronise from them. This syncing and desyncing 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.

### Support tools

#### Config (config)

Loads an sDNA_GH project configuration file (`.toml` or `.ini`, e.g. `config.toml`) along with the sDNA_GH Python package and overrides any other specific options the user wishes to set, even if `go` is not set to true. Setting `go` to true saves these options to the `.toml` options file if there is one named in `save_to`. If nothing is connected to `save_to`, then if an installation wide user options file does not already exist one is created, with the current options are saved to it (subsequently to overwrite it, its file path must be specified in `save_to`).

#### Read_Geom (get_Geom)

Read in references to Rhino geometry (polylines) to provide them in the required form for subsequent sDNA_GH tools. Can be merged and override with other supplied geometry and data. The UUIDs of the Rhino objects are converted to strings to preserve the references to them. 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.

### Shapefile tools

#### Write_Shp (write_shapefile)

Writes the provided data and geometry (polylines) to a shapefile. If not specified, a file name based on the Rhino doc or Grasshopper doc name is used (unless `auto_update_Rhino_doc_path = false`). Can overwrite existing files, or create new unique files. If no Data is supplied it will first call read_Usertext (unless `auto_read_Usertext = false`). To work with sDNA, a data point is only written to the Shapefile (in the record asociated with the shape correspnding to its Rhino / GH polyline) if its field (key name if it originated as Usertext) 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)

Read in the polylines and data from the specified shapefile. **WARNING! Read_Shp automatically deletes the shapefile after reading it in the file names match the pattern of the automatically created sDNA output files or if `del_after_read` = true, as long as `strict_no_del` = false, as they are by default.** If a list of existing geometry is provided 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. The bounding box is provided to create a legend frame with in 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 the output Params so that a dropdown 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 data in a Data Tree or GDM (Geometry and Data Mapping), from a specified field, for subsequent colouring and plotting. Use this component separately from Recolour_Objects to calculate colours with a visible Grasshopper Colour Gradient component. Max and Min bounds can be overridden, else they are calculated on the whole data range.
**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 objects are coloured according only to the midpoint of the bin / class they are in, the parsed data will take far fewer distinct values than the number of polylines in a large network.

**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 Usertext or a Shapefile. Grasshopper"s path tools can be used to adjust compatible Datatrees into this format.

**Field to plot** Specify the actual numeric data values to be parsed from all the provided 'Usertext values' by setting `field` to the name of the corresponding 'Usertext 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 'Usertext values' of Data corresponding to the 'Usertext 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 `number_of_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 onthe 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. **Legend class names** Three customisable fields are provided in the options for the first, general and last legend tag names, and a numeric formatting string: `first_leg_tag_str = 'below {upper}'`, `gen_leg_tag_str = '{lower} - {upper}'`, `last_leg_tag_str = 'above {lower}'` and `num_format = '{:.5n}'` respectively. The numeric format string supports a single unnamed field. This is applied to the Classbounds and min and max above, before being substituted into the named fields in the format string for their corresponding legend tag (the general one may be used to produce more than one tag). All must be set to valid Python format strings, with the supported named fields `lower`, `mid_pt` and `upper`.

**Re-normalisation** Finally in order to produce a recolourisation that has a set number of identicla 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).

#### Recolour_Objects (recolour_objects)

Recolour objects (and legend tags) based on pre-parsed and pre-normalised data, or already calculated colours (RGB). Custom colour calculation is possible, as is the Grasshopper Colour Gradient internally via Node In Code. Create a legend by connecting leg_cols, leg_tags and leg_frame to a Grasshopper Legend component. 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`). Recolouring unbaked Grasshopper geometry instead of Rhino Geometry requires the Data and Geometry outputs to be connected to a Grasshopper Custom Preview component. If unparsed data is inputted, Parse_Data is first called.

### Usertext tools

### Data tools

#### Read_Usertext (read_Usertext)

Reads all Usertext from the provided Rhino objects. If no Geometry is provided, Read_From_Rhino is first called (unless `auto_get_Geom` = false).

#### Write_Usertext (write_Usertext)

Write user text to Rhino objects using a customisable pattern for the keys.

## sDNA Tools

### Analysis tools

By default an sDNA tool component will show all the possible inputs on its Input Parms. To show only the essential ones instead (and make the components a lot smaller) set the *meta* `show_all` =

false. **WARNING! All sDNA tools will delete the shapefile 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 wrapper. This and all sDNA tools below, automatically calls other support tools, handling the normal 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). To analyse Grasshopper Geometry and to customise work flows between sDNA_GH components, e.g. using a Grasshopper Colour Gradient component, set the corresponding `auto_` option to false in `config.toml`. Connect a Grasshopper Legend component to plot a legend. The component attempts to check if any Write_Usertext or Read_Usertext components are already connected to its inputs (upstream) or to its outputs (downstream), before running the extra tools before or afterwards respectively.

#### sDNA_Skim (sDNASkim)

Skim Matrix. 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. 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.

### 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.

#### 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_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_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.

### 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 analagous 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 $\lambda$ 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 inpect a plot of cross validated $r^2$ against $\lambda$ 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_GH

Unload the sDNA_GH Python package and all sDNA modules, by removing their keys from sys.modules.
The next sDNA_GH component to run will then reload the package and installation-wide options file (config.toml), and any specified options including a project options config.toml, without otherwise having to restart Rhino to clear its cache.

#### User buildable dev tools.

##### sDNA_general

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

##### Comp_Names

Output the names of all the sDNA tool classes for the sDNA installation provided in opts, as well as all the sDNA_GH support tool names.

##### Self_test

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 to the functions in the launcher, can also be added to the launcher code.

##### Build_components

Easily build all the other components for the sDNA installation provided. User Objects still need to be built manually, but components are all the same launcher code in a Gh_Python component, but with different names. Functionality is provided by main.py in the sDNA_GH Python package, so new components are only needed to be built for tools sDNA_GH doesn't know about yet.

# Example Grasshopper definitions.

## Running sDNA Integral on a random grid read from Rhino.

#### Selecting and specifying an sDNA Results field.

#### Reading shapefile data with existing geometry.

#### Using a Grasshopper Colour gradient component.

#### Adding a legend with a Legend component.

#### Customising legend class boundaries and tag names.

#### Running sDNA Integral on a random grid of Grasshopper geometry (colouring with the Custom Preview component).

#### Running sDNA Integral on a network of polylines, approximating a network of arcs from intersecting circles.

##### Recolouring the arcs instead of polylines.

#### Writing polylines and data to shapefiles.

#### Reading in polylines and data from shapefiles.

#### Writing Usertext.

#### Reading Usertext for sDNA (e.g. User weights).

#### Baking (saving Grasshopper objects to a Rhino document) with Usertext.

### License.



### Copyright.



# Contact.

grasshopper.sdna@gmail.com

# Developer manual.

### Dependencies.

To bulk unblock files, to avoid unblocking every file manually it is necessary to install (Powershell)[https://docs.microsoft.com/en-us/powershell/scripting/install/installing-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" https://github.com/GeospatialPython/pyshp/blob/master/shapefile.py Toml (MIT License) https://github.com/uiri/toml/blob/master/toml/decoder.py Latest commit 230f0c9 on 30 Oct 2020

### Build instructions.

1. Open `\dev_tools\sDNA_build_components.gh` in Grasshopper.
2. Right click the Path Component and ensure it points to `\sDNA_GH \sDNA_GH_launcher.py`
3. Ensure the File Reader Component (that the Path Component is connected to) is set to read

the whole file, and also is connected to the *launcher code* input param on the Build_components GhPython component. Set the plug-in name on *plug in name*.
4. In the main Grasshopper Display pull down menu, ensure Draw Icons is turned off (this displays comoponent names instead).
5. Change the Boolean toggle to True, and connect it to the `go` input param of Build_components.
6. A slight delay may occur as sDNA_GH/main.py is imported, and the 23 or so components are created.
7. Turn the Boolean toggle to False (connected to the go input param of Build_components). This ensures no further components are created (unnecessary duplicates). The components are disabled, otherwise the next update will makes each one ask Grasshopper what its name is, connect to sDNA_GH.main.py, and update its own Input and Output params.
8. Click the pull down menu *Solution* and select *Disable Solver*.
9. Right click each new component (on its name not its Params) and select Enable.

10. Select each component one at a time, and go to the main Grasshopper File pull down menu, and select *Create User Object ...*
11. Ensure both the Name and Nickname are the same as the automatically created component name (as some versions of Rhino 7 read their component names from the descriptive human-readable name). Ensure the main category is sDNA_GH or sDNA. Look up the sub category in the main.py meta option categories. Description text can be used from the tool's description in this readme file itself (above).
12. From `%appdata%\Grasshopper\UserObjects` or the Grasshopper User objects folder, copy (or move) all the .ghuser files just created into `\sDNA_GH` in the main repo, next to config.toml
13. For non-Github users, a good quality pdf of this file (`README.md`) can be created in VS Code with this extension: (print, PD Consulting VS Marketplace Link)[ https://marketplace.visualstudio.com/items?itemName=pdconsec.vscode-print]. This will render the markdown file in your web browser. Print it to a pdf with the name `README.pdf` in the same directory.
14. Run `create_release_sDNA_GH_zip.bat` to create the zip file for release.
15. Note: The components are only GhPython launchers with different names, so steps 1 - 12 above (in particular, the most laborious step, number 10.) only need to be repeated if the code in `\sDNA_GH\sDNA_GH_launcher.py` has been changed, or if new components need to be built e.g. for new tools . 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`

## 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)