Biodiverse Spatial Conditions

Shawn W Laffan

2026-02-05

Table of contents

1 Introduction

Spatial conditions are core to the Biodiverse system. They are used to specify both neighbourhoods used in the analyses, and also the definition queries used to restrict the calculations to a subset of groups.

This document provides an overview of using these conditions, lists the available conditions and describes how to use variables, both predefined and user defined.

This document describes **version 5.0**

Biodiverse is a tool for the spatial analysis of biological and related diversity.

For more information about Biodiverse see:

Web site: http://shawnlaffan.github.io/biodiverse/

The blog provides updates and tips about functionality. It can be accessed through https://biodiverse-analysis-software.blogspot.com.au/

If you have question about the software then please start a discussion at https://github.com/shawnlaffan/biodiverse/discussions or post a question at https://groups.google.com/forum/#!forum/biodiverse-users

2 Overview

2.1 Uses for spatial conditions

Spatial conditions are used both to define the neighbourhoods of the spatial analyses and the definition queries used to constrain the set of groups used in the analyses.

2.1.1 Neighbourhoods

Neighbourhoods are essential for any spatial analysis, as it is through these that one can define the set of groups to be considered in an analysis. In the moving window analyses these determine which groups are compared with which other groups. In the cluster analyses they determine which groups are considered candidates to be clustered together. It is also possible to define neighbourhoods for spatially constrained randomisations (see Laffan and Crisp, 2003, J Biogeog).

Before we describe the process, some definitions are needed. The **processing group** is the group being considered in the analysis at some iteration, and to which the results for that iteration are assigned. A group is a member of the processing group's set of neighbours (is a **neighbouring group**) if the spatial condition evaluates to true.

A spatial analysis progressively iterates over each group that passes the definition query, assessing every other group for membership in neighbour set 1 or 2. The selected indices are then calculated using the groups that occur in neighbour sets 1 and 2 (and their labels and other properties as required by the calculations).

2.1.2 Definition Queries

These differ from neighbourhoods as they assess only the processing group to determine if calculations should be run for it or not. They use the same syntax as for neighbourhoods, but the system will commonly complain if a condition requiring a neighbouring group is used.

Note that groups that fail the definition query are still considered for membership of neighbour sets of those that pass. Use a definition query in conjunction with an appropriate neighbourhood definition if you want to exclude some groups from neighbour sets as well. For example,

you could use `sp_match_text (axis => 2, text => 'fred', type => 'proc')` for the definition query and `sp_match_text (axis => 2, text => 'fred', type => 'nbr')` for the neighbourhood. This will restrict calculations to those groups with a third axis of 'fred', and also exclude any group without fred in the third axis from the neighbour sets of those groups processed.

2.2 Some details

As with any system, there must be compromises between ease of use and system flexibility. In this case we have opted for system flexibility by direct use of Perl syntax. This means you can use arbitrarily complex functions to define neighbourhoods, including loops and other multiple variable conditions. This may be horrifying to non-perlers, as one of the main complaints about perl is its complex grammar and syntax. To alleviate this we are encapsulating many of the common conditions in subroutines that can be called by name with a set of arguments. We have also provided examples below to assist.

The neighbourhood and definition query interfaces have a syntax verification button to check that the syntax is valid. This does not, however, guarantee your parameters will work, only that it is valid Perl code. (The reality here is that we will just evaluate the parameter statement with some default values and warn you if the system raises some sort of error or exception).

2.2.1 Locale issues

If you are using a locale where the radix character (the decimal place marker) is a comma then you need to remember to use a dot instead. For example, this code `sp_circle (radius => 5,3)` should be `sp_circle (radius => 5.3)` or you will receive warnings about uneven arguments used to define a hash.

2.3 Evaluation

This is a brief description of the evaluation process used to determine the set of neighbours for a group.

Currently the system operates on boolean membership of the set of neighbours, so a group either is or is not a neighbour of the processing group. If no spatial index is used then every group's membership of the processing cell's neighbour set is considered in turn. If a spatial index is used then only a subset of neighbours is considered (those within the relevant spatial index blocks). This is why processing times are usually shorter when using an index (more details here).

Spatial conditions need not return symmetric sets. In this way group i can be in group j's neighbour set, but j need not be in i's neighbour set. This is not an issue for moving window analyses, but can cause asymmetric dissimilarity matrices if used to run a spatially constrained cluster analysis. This is why it is generally a good idea in these cases to set the second neighbourhood to be `sp_select_all()` or `1` (which is the same thing).

In the calculations, groups in neighbour set 1 are excluded from neighbour set 2 so there are no overlaps that would violate the comparison calculations.

The conditions are specified using some combination of pre-defined functions, pre-specified variables, and/or user defined variables and functions. These are now described.

3 Functions

Functions are the easiest way to specify conditions as one does not need to wrestle with variables.

Functions also set metadata to tell the system how to use the spatial index. The spatial index saves considerable processing time for large data sets as the system does not need to test many pairs of index blocks to determine which to use. If you use a function for which an index will produce erroneous results then the system sets a flag to ignore it. You can also disable it in the settings for a spatial condition.

3.1 Available functions

The available functions in version 5.0 are: *sp_annulus*, *sp_block*, *sp_circle*, *sp_circle_cell*, *sp_ellipse*, *sp_get_spatial_output_list_value*, *sp_group_not_empty*, *sp_in_label_ancestor_range*, *sp_in_label_range*, *sp_in_line_with*, *sp_is_left_of*, *sp_is_right_of*, *sp_match_regex*, *sp_match_text*, *sp_point_in_cluster*, *sp_point_in_poly*, *sp_point_in_poly_shape*, *sp_points_in_same_cluster*, *sp_points_in_same_poly_shape*, *sp_rectangle*, *sp_redundancy_greater_than*, *sp_richness_greater_than*, *sp_select_all*, *sp_select_block*, *sp_select_element*, *sp_select_sequence*, *sp_self_only*, *sp_spatial_output_passed_defq*, *sp_square*, *sp_square_cell*,

3.1.1 sp_annulus

An annulus. Assessed against all dimensions by default but use the optional `axes => []` arg to specify a subset. Uses group (map) distances.

Required args: `inner_radius`, `outer_radius`

Optional args: `axes`

Example:

```
#  an annulus assessed against all axes
sp_annulus (inner_radius => 2000000, outer_radius => 4000000)

#  an annulus assessed against axes 0 and 1
sp_annulus (inner_radius => 2000000, outer_radius => 4000000, axes => [0,1])
```

3.1.2 sp_block

A non-overlapping block. Set an axis to undef to ignore it.

Required args: `size`

Optional args: `origin`

Example:

```
sp_block (size => 3)
sp_block (size => [3,undef,5]) #  rectangular block, ignores second axis
```

3.1.3 sp_circle

A circle. Assessed against all dimensions by default (more properly called a hypersphere) but you can use the optional `axes => []` arg to specify a subset. Uses group (map) distances.

Required args: `radius`

Optional args: `axes`

Example:

```
#  A circle of radius 1000 across all axes
sp_circle (radius => 1000)

#  use only axes 0 and 3
sp_circle (radius => 1000, axes => [0, 3])
```

3.1.4 sp_circle_cell

A circle. Assessed against all dimensions by default (more properly called a hypersphere) but you can use the optional `axes => []` arg to specify a subset. Uses cell (map) distances.

Required args: `radius`

Optional args: *none*

Example:

```
#  A circle of radius 3 cells across all axes
sp_circle (radius => 3)

#  use only axes 0 and 3
sp_circle_cell (radius => 3, axes => [0, 3])
```

3.1.5 sp_ellipse

A two dimensional ellipse. Use the `axes` argument to control which are used (default is `[0,1]`). The default rotate_angle is 0, such that the major axis is east-west.

Required args: `major_radius`, `minor_radius`

Optional args: `axes`, `rotate_angle`, `rotate_angle_deg`

Example:

```
# North-south aligned ellipse
sp_ellipse (
    major_radius => 300000,
    minor_radius => 100000,
    axes         => [0,1],
    rotate_angle => 1.5714,
)
```

3.1.6 sp_get_spatial_output_list_value

Obtain a value from a list in a previously calculated spatial output.

Required args: `index`, `output`

Optional args: `element`, `list`, `no_error_if_no_index`

Example:

```
#  Get the spatial results value for the current neighbour group
# (or processing group if used as a def query)
sp_get_spatial_output_list_value (
    output  => 'sp1',              #  using spatial output called sp1
    list    => 'SPATIAL_RESULTS',  #  from the SPATIAL_RESULTS list
    index   => 'PE_WE_P',          #  get index value for PE_WE_P
)

#  Get the spatial results value for group 128:254
#  Note that the SPATIAL_OUTPUTS list is assumed if
#  no 'list' arg is passed.
sp_get_spatial_output_list_value (
    output  => 'sp1',
    element => '128:254',
    index   => 'PE_WE_P',
)
```

3.1.7 sp_group_not_empty

Is a basedata group non-empty? (i.e. contains one or more labels)

Required args: *none*

Optional args: `element`

Example:

```
# Restrict calculations to those non-empty groups.
#  Will use the processing group if a def query,
#  the neighbour group otherwise.
sp_group_not_empty ()

# The same as above, but being specific about which group (element) to test.
#  This is probably best used in cases where the element
#  to check is varied spatially.}
sp_group_not_empty (element => '5467:9876')
```

3.1.8 sp_in_label_ancestor_range

(Available from version 5.1)

Is a group within the range of a label's ancestor?

Returns true if the group falls within the range of any of the any of the ancestor's terminal descendant ranges. The range is by default defined as the set of groups in the basedata containing that label. Polygons can also be specified (see below).

The ancestor is by default defined by length along the path to the root node. Setting the `by_depth` option to true uses the number of ancestors. The `by_tip_count` option finds the first ancestor with at least the target number of tips while `by_desc_count` uses the number of descendants (tips and internals). The `by_len_sum` finds the first ancestor for which the sum of its descendant branch lengths plus its own length is greater than the target.

Negative target values with `by_length` or `by_depth` search the path from the root to the specified node. However, if a `by_*_count` option is used then a negative `target` is treated as zero and returns the specified label range is returned.

The `target` argument determines how far up or down the tree the ancestor is searched for. When using length, the distance includes the tipwards extent of the branch. The depth is calculated as the number of ancestors.

If the `target` value exceeds that to (or of) the root node then the root or label node is returned for positive or negative dist values, respectively.

An internal branch can be specified as the label. Specifying a `target` of 0 for an internal node is one means to use the range of an internal node.

Returns false if the label is not associated with a node on the tree.

When the `as_frac` argument is true then target is treated as a fraction of the distance to the root node, the number of tips, or the sum of all branches, as appropriate.

If the `eq` argument is true then the branch lengths are all treated as of equal length (the mean of the non-zero branch lengths), although zero length branches remain zero. This is the same as the alternate tree used in CANAPE.

If the `rw` argument is true then the branches are range weighted. This is the same as the range weighted tree in CANAPE. When both `eq` and `rw` are true then this is the same as the range weighted alternate tree in CANAPE

The underlying algorithm checks each of the terminal ranges using sp_in_label_range(). This means the search can also use the convex/concave hull or circumcircle of each terminal, as well as setting other arguments such as the `buffer_dist` and using a default label in some circumstances.

Note that the range of each of the ancestor's tips is assessed separately, i.e. the union of the hulls/circles is used. The ranges are not aggregated before a hull or circumcircle is calculated.

Required args: `label`, `target`

Optional args: `allow_holes`, `as_frac`, `axes`, `buffer_dist`, `by_depth`, `by_tip_count`, `circumcircle`, `concave_hull`, `convex_hull`, `hull_ratio`, `type`

Example:

```
# Are we in the range of an ancestor of Genus:Sp1?
sp_in_label_ancestor_range(label => 'Genus:Sp1', target => 0.5)

# Are we in the range of the "grandmother" of Genus:Sp1?
sp_in_label_ancestor_range(
  label    => 'Genus:Sp1',
  target   => 2,
  by_depth => 1,
)

# Are we in the range of the first ancestor with 6 or more tips?
sp_in_label_ancestor_range(
  label        => 'Genus:Sp1',
  target       => 6,
  by_tip_count => 1,
)

#  Are we in any of the tips' convex hulls?
sp_in_label_ancestor_range(
  label       => 'Genus:Sp1',
  target      => 0.5,
  convex_hull => 1,
)

#  Are we in any of the tips' concave hulls with a ratio parameter of 0.5?
sp_in_label_ancestor_range(
  label        => 'Genus:Sp1',
  target       => 0.5,
  concave_hull => 1,
  hull_ratio   => 0.5,
)

#  Are we in any of the tips' circumscribing circles?
sp_in_label_ancestor_range(
  label        => 'Genus:Sp1',
  target       => 0.5,
  circumcircle => 1,
)

#  Are we in the range of the ancestor for which the sum of
#  branch lengths below and including it is 10,000?
```

```
sp_in_label_ancestor_range(
  label      => 'Genus:Sp1',
  target     => 10000,
  by_len_sum => 1,
)

#  Are we in the range of the ancestor for which the sum of
#  number of branches below is at least 10?
sp_in_label_ancestor_range(
  label      => 'Genus:Sp1',
  target     => 10,
  by_desc_count => 1,
)
```

3.1.9 sp_in_label_range

(Available from version 5.1)

Is a group within a label's range?

This is by default assessed as a check of whether the label is found in the processing group but can be generalised by using the `convex_hull`, `concave_hull` or `circumcircle` arguments.

The `type` argument determines if the processing or neighbour group is assessed. Normally this can be left as the default.

The `convex_hull` returns true if the processing group is within the convex hull defined by the groups that form the label range.

The `circumcircle` returns true if the processing group is within the minimum circumscribing circle that includes all of the label range groups.

The `concave_hull` returns true if the processing group is within the concave hull defined by the groups that form the label range. The concavity can be controlled by the `hull_ratio` argument, and holes can be allowed by setting the boolean argument `allow_holes` to a true value.

Both the latter two arguments use the first two axes by default and will return an error if there is only one group axis in the basedata. If you have more than two axes and wish to assess different ones then pass the axes argument. (This argument is ignored for the default case as it does a direct comparison of the group names).

If more than one of `convex_hull`, `concave_hull` and `circumcircle` arguments are passed then only one i run. The convex hull takes priority, followed by the circumcircle, and lastly the concave hull.

An optional `buffer_dist` argument can be used to adjust the size of the convex/concave hull or circumcircle. As is standard with GIS buffering, positive values increase the area while negative values shrink it.

The `label` argument should normally be specified but in some circumstances a default is set (e.g. when a randomisation seed location is set).

Required args: `label`

Optional args: `allow_holes`, `axes`, `buffer_dist`, `circumcircle`, `concave_hull`, `convex_hull`, `hull_ratio`, `type`

Example:

```
# Are we in the range of label called Genus:Sp1?
sp_in_label_range(label => 'Genus:Sp1')

#  Are we in the convex hull?
sp_in_label_range(label => 'Genus:Sp1', convex_hull => 1)

#  Are we in the maximally concave hull?
sp_in_label_range(label => 'Genus:Sp1', concave_hull => 1)

#  Are we in a slightly less concave hull?
sp_in_label_range(label => 'Genus:Sp1', concave_hull => 1, hull_ratio => 0.3)

#  Are we in a slightly less concave hull allowing for holes?
sp_in_label_range(
    label        => 'Genus:Sp1',
    concave_hull => 1,
    hull_ratio   => 0.3,
    allow_holes  => 1,
)

#  Are we in the circumscribing circle?
sp_in_label_range(label => 'Genus:Sp1', circumcircle => 1)

#  Are we in the convex hull with a buffer of 100,000 units?
sp_in_label_range(
    label       => 'Genus:Sp1',
    convex_hull => 1,
```

```
    buffer_dist => 100000,
)

#  Buffers can be negative, in which case the
#  convex/concave hull or circumcircle is shrunk
sp_in_label_range(
    label       => 'Genus:Sp1',
    convex_hull => 1,
    buffer_dist => -100000,
)

#  Are we in the convex hull defined using the
#  coordinates from the third and first axes?
sp_in_label_range(
    label       => 'Genus:Sp1',
    convex_hull => 1,
    axes        => [2,0],
)

#  A convex hull with holes and "half" concave.
sp_in_label_range(
    label        => 'Genus:Sp1',
    concave_hull => 1,
    allow_holes  => 1,
    hull_ratio   => 0.5,
)
```

3.1.10 sp_in_line_with

Are we in line with a vector radiating out from the processing cell? Use the `axes` argument to control which are used (default is `[0,1]`).

Required args: *none*

Optional args: `axes`, `vector_angle`, `vector_angle_deg`

Example:

```
sp_in_line_with (vector_angle => Math::Trig::pip2) #  pi/2 = 90 degree angle
```

3.1.11 sp_is_left_of

Are we to the left of a vector radiating out from the processing cell? Use the `axes` argument to control which are used (default is `[0,1]`).

Required args: *none*

Optional args: `axes`, `vector_angle`, `vector_angle_deg`

Example:

```
sp_is_left_of (vector_angle => 1.5714)
```

3.1.12 sp_is_right_of

Are we to the right of a vector radiating out from the processing cell? Use the `axes` argument to control which are used (default is `[0,1]`).

Required args: *none*

Optional args: `axes`, `vector_angle`, `vector_angle_deg`

Example:

```
sp_is_right_of (vector_angle => 1.5714)
```

3.1.13 sp_match_regex

Select all neighbours with an axis matching a regular expression

Required args: `re`

Optional args: `axis`, `type`

Example:

```
#  use any neighbour where the first axis includes the text "type1"
sp_match_regex (re => qr'type1', axis => 0, type => 'nbr')

# match only when the third neighbour axis starts with
# the processing group's second axis
sp_match_regex (re => qr/^$coord[2]/, axis => 2, type => 'nbr')

# match the whole coordinate ID (element name)
# where Biome can be 1 or 2 and the rest of the name contains "dry"
```

```
sp_match_regex (re => qr/^Biome[12]:.+dry/)

# Set a definition query to only use groups where the
# third axis ends in 'park' (case insensitive)
sp_match_regex (text => qr{park$}i, axis => 2, type => 'proc')
```

3.1.14 sp_match_text

Select all neighbours matching a text string

Required args: `text`

Optional args: `axis`, `type`

Example:

```
#  use any neighbour where the first axis has value of "type1"
sp_match_text (text => 'type1', axis => 0, type => 'nbr')

# match only when the third neighbour axis is the same
#   as the processing group's second axis
sp_match_text (text => $coord[2], axis => 2, type => 'nbr')

# match where the whole coordinate ID (element name)
# is 'Biome1:savannah forest'
sp_match_text (text => 'Biome1:savannah forest')

# Set a definition query to only use groups with 'NK' in the third axis
sp_match_text (text => 'NK', axis => 2, type => 'proc')
```

3.1.15 sp_point_in_cluster

Returns true when the group is in a cluster or region grower output cluster.

Required args: `output`

Optional args: `element`, `from_node`

Example:

```
#  Use any element that is a terminal in the cluster output.
#  This is useful if the cluster analysis was run under
#  a definition query and you want the same set of groups.
sp_point_in_cluster (
  output       => "some_cluster_output",
)

#  Now specify a cluster within the output
sp_point_in_cluster (
  output       => "some_cluster_output",
  from_node    => '118___',  #  use the node's name
)

#  Specify an element to check instead of the current
#  processing element.
sp_point_in_cluster (
  output       => "some_cluster_output",
  from_node    => '118___',  #  use the node's name
  element      => '123:456', #  specify an element to check
)
```

3.1.16 sp_point_in_poly

Select groups that occur within a user-defined polygon (see sp_point_in_poly_shape for an alternative)

Required args: `polygon`

Optional args: `point`

Example:

```
# Is the neighbour coord in a square polygon?
sp_point_in_poly (
    polygon => [[0,0],[0,1],[1,1],[1,0],[0,0]],
    point   => \@nbrcoord,
)
```

3.1.17 sp_point_in_poly_shape

Select groups that occur within a polygon or polygons extracted from a shapefile

Required args: `file`

Optional args: `axes`, `field_name`, `field_val`, `no_cache`, `point`

Example:

```
# Is the neighbour coord in a shapefile?
sp_point_in_poly_shape (
    file  => 'c:\biodiverse\data\coastline_lamberts',
    point => \@nbrcoord,
);
# Is the neighbour coord in a shapefile's second polygon (counting from 1)?
sp_point_in_poly_shape (
    file      => 'c:\biodiverse\data\coastline_lamberts',
    field_val => 2,
    point     => \@nbrcoord,
);
# Is the neighbour coord in a polygon with value 2 in the OBJECT_ID field?
sp_point_in_poly_shape (
    file       => 'c:\biodiverse\data\coastline_lamberts',
    field_name => 'OBJECT_ID',
    field_val  => 2,
    point      => \@nbrcoord,
);
```

3.1.18 sp_points_in_same_cluster

Returns true when two points are within the same cluster or region grower group, or if neither point is in the selected clusters/groups.

Required args: `output`

Optional args: `from_node`, `group_by_depth`, `num_clusters`, `target_distance`

Example:

```
#  Try to use the highest four clusters from the root.
#  Note that the next highest number will be used
#  if four is not possible, e.g. there are five
#  siblings below the root.  Fewer will be returned
```

```
#  if the tree has insufficient tips.
sp_points_in_same_cluster (
  output       => "some_cluster_output",
  num_clusters => 4,
)

#  Cut the tree at a distance of 0.25 from the tips
sp_points_in_same_cluster (
  output          => "some_cluster_output",
  target_distance => 0.25,
)

#  Cut the tree at a depth of 3.
#  The root is depth 1.
sp_points_in_same_cluster (
  output          => "some_cluster_output",
  target_distance => 3,
  group_by_depth  => 1,
)

#  work from an arbitrary node
sp_points_in_same_cluster (
  output       => "some_cluster_output",
  num_clusters => 4,
  from_node    => '118___',  #  use the node's name
)

#  target_distance is ignored if num_clusters is set
sp_points_in_same_cluster (
  output          => "some_cluster_output",
  num_clusters    => 4,
  target_distance => 0.25,
)
```

3.1.19 sp_points_in_same_poly_shape

Returns true when two points are within the same shapefile polygon

Required args: `file`

Optional args: `axes`, `no_cache`, `point1`, `point2`

Example:

```
#  define neighbour sets using a shapefile
sp_points_in_same_poly_shape (file => 'path/to/a/shapefile')

#  return true when the neighbour coord is in the same
#  polygon as an arbitrary point
sp_points_in_same_poly_shape (
    file   => 'path/to/a/shapefile',
    point1 => [10,20],
)

#  reverse the axes
sp_points_in_same_poly_shape (
    file => 'path/to/a/shapefile',
    axes => [1,0],
)

#  compare against the second and third axes of your data
#  e.g. maybe you have time as the first basedata axis
sp_points_in_same_poly_shape (
    file => 'path/to/a/shapefile',
    axes => [1,2],
)
```

3.1.20 sp_rectangle

A rectangle. Assessed against all dimensions by default (more properly called a hyperbox) but use the optional `axes => []` arg to specify a subset. Uses group (map) distances.

Required args: `sizes`

Optional args: `axes`

Example:

```
#  A rectangle of equal size on the first two axes,
#  and 100 on the third.
sp_rectangle (sizes => [100000, 100000, 100])

#  The same, but with the axes reordered
#  (an example of using the axes argument)
```

```
sp_rectangle (
    sizes => [100000, 100, 100000],
    axes  => [0, 2, 1],
)

#  Use only the first an third axes
sp_rectangle (sizes => [100000, 100000], axes => [0,2])
```

3.1.21 sp_redundancy_greater_than

Return true if the sample redundancy for an element is greater than the threshold.

Required args: `threshold`

Optional args: `element`

Example:

```
#  Uses the processing group for definition queries,
#  and the neighbour group for spatial conditions.
#  In this example, # any group with a redundncy
#  score of 0.5 or fewerless will return false
sp_redundancy_greater_than (
    threshold => 0.5,
)

sp_redundancy_greater_than (
    element   => '128:254',  #  an arbitrary element
    threshold => 0.2,          #  with a threshold of 0.2
)
```

3.1.22 sp_richness_greater_than

Return true if the richness for an element is greater than the threshold.

Required args: `threshold`

Optional args: `element`

Example:

```
#  Uses the processing group for definition queries,
#  and the neigbour group for spatial conditions.
sp_richness_greater_than (
    threshold => 3, # any group with 3 or fewer labels will return false
)

sp_richness_greater_than (
    element   => '128:254',  #  an arbitrary element
    threshold => 4,          #  with a threshold of 4
)
```

3.1.23 sp_select_all

Select all elements as neighbours

Required args: *none*

Optional args: *none*

Example:

```
sp_select_all() #  select every group
```

3.1.24 sp_select_block

Select a subset of all available neighbours based on a block sample sequence

Required args: `size`

Optional args: `count`, `prng_seed`, `random`, `reverse_order`, `use_cache`

Example:

```
# Select up to two groups per block with each block being 5 groups
on a side where the group size is 100
sp_select_block (size => 500, count => 2)

#  Now do it non-randomly and start from the lower right
sp_select_block (size => 500, count => 10, random => 0, reverse => 1)

#  Rectangular block with user specified PRNG starting seed
sp_select_block (size => [300, 500], count => 1, prng_seed => 454678)
```

```
# Lower memory footprint (but longer running times for neighbour searches)
sp_select_block (size => 500, count => 2, clear_cache => 1)
```

3.1.25 sp_select_element

Select a specific element. Basically the same as sp_match_text, but with optimisations enabled

Required args: `element`

Optional args: `type`

Example:

```
# match where the whole coordinate ID (element name)
# is 'Biome1:savannah forest'
sp_select_element (element => 'Biome1:savannah forest')
```

3.1.26 sp_select_sequence

Select a subset of all available neighbours based on a sample sequence (note that groups are sorted south-west to north-east)

Required args: `frequency`

Optional args: `cycle_offset`, `first_offset`, `reverse_order`, `use_cache`

Example:

```
# Select every tenth group (groups are sorted alphabetically)
sp_select_sequence (frequency => 10)

#  Select every tenth group, starting from the third
sp_select_sequence (frequency => 10, first_offset => 2)

#  Select every tenth group, starting from the third last
#  and working backwards
sp_select_sequence (
    frequency     => 10,
    first_offset  =>  2,
    reverse_order =>  1,
)
```

3.1.27 sp_self_only

Select only the processing group

Required args: *none*

Optional args: *none*

Example:

```
sp_self_only() #  only use the proceessing cell
```

3.1.28 sp_spatial_output_passed_defq

Returns 1 if an element passed the definition query for a previously calculated spatial output

Required args: *none*

Optional args: `element`, `output`

Example:

```
#  Used for spatial or cluster type analyses:
#  The simplest case is where the current
#  analysis includes a def query and you
#  want to use it in a spatial condition.
sp_spatial_output_passed_defq();

#  Using another output in this basedata
#  In this case the output is called 'analysis1'
sp_spatial_output_passed_defq(
    output => 'analysis1',
);

#  Return true if a specific element passed the def query
sp_spatial_output_passed_defq(
    element => '153.5:-32.5',
);
```

3.1.29 sp_square

An overlapping square assessed against all dimensions (more properly called a hypercube). Uses group (map) distances.

Required args: `size`

Optional args: *none*

Example:

```
#  An overlapping square, cube or hypercube
#  depending on the number of axes
#   Note - you cannot yet specify which axes to use
#   so it will be square on all sides
sp_square (size => 300000)
```

3.1.30 sp_square_cell

A square assessed against all dimensions (more properly called a hypercube). Uses 'cell' distances.

Required args: `size`

Optional args: *none*

Example:

```
sp_square_cell (size => 3)
```

4 Variables

There are several different sets of variables implemented that the system recognises. Any undeclared variable you use that does not occur in this list will be treated as a zero or as undefined (depending on where it is used), which means it will probably not behave as you expect. An example declaring variables is given below.

As a general rule, uppercase letters denote absolute values, lower case letters denote signed values (positive or negative). Positive values are north, east, above, or to the right. Negative values are south, west, below or to the left.

`$D` is the absolute euclidean distance from the processing group to a candidate neighbour group across all dimensions.

`$D[0]`, `$D[1]` are the absolute euclidean distances in dimension 0 and 1. In most cases `$D[0]` will be the X (longitude) dimension, `$D[1]` will be the Y (latitude) dimension. The library functions can actually handle more dimensions than this (eg `$D[2]` for altitude or depth), but the GUI is not set up to display them (it will plot the data using the first two axes, so only the first of any overlapping groups will be visible).

`$d[0]`, `$d[1]` and so forth are the signed euclidean distance in dimension 0, 1 etc. This allows us to extract all groups within some distance in some direction. As with standard Cartesian plots, negative values are to the left or below (west or south), positive values to the right or above (east or north). As with `$D[0]`, `$d[0]` will normally be the X dimension, `$d[1]` will be the Y dimension.

Note that using `abs($d[1])` is the same as using `$D[1]`.

`$C`, `$C[0]`, `$C[1]`, `$c[0]`, `$c[1]` are the same as the euclidean distance variables (`$D` etc) but operate directly in group (cell) units. If your groups were imported using a cellsize of 100,000, then `$D[1] < 100000` is the same as `$C[1] < 1`. Note, however, that if you used a different resolution in each dimension, then the map and cell distances are not directly comparable. For example, if cell sizes of 100 and 200 were used for axes 0 and 1 then `$C<1` is the same as `sqrt($C[0]**2 + $C[1]**2) < 1` which is `sqrt(($D[0]/100)**2 + ($D[1]/200)**2) < 1`, and *not* `$D<100`.

`$coord_id1` is the name of the processing coord, `$coord_id2` is the name of the neighbour coord.

`$coord[0]`, `$coord[1]` are the coordinate values of the processing group in the first and second dimensions. As per the above, think of these as X and Y, except that `$coord[5]` will also

work if your groups have six or more dimensions. Note that the `$coord[]` variables do not necessarily work properly with the spatial index, so you might need to turn the index off when using them.

`$nbrcoord[0]` etc are analogous to `$coord[0]` etc, except that they are the coordinates for the current neighbour group.

Note that the array index starts from zero, so `$coord[1]` is the second coordinate axis and not the first. This differs from systems like Fortran, R and AWK, but is consistent with many other programming languages like C, Rust and Python.

4.1 Examples using variables

- Set the neighbours to be those groups where the absolute distance from the processing group is less than 100,000.

```
$D <= 100000
```

- Select all groups to the west of the processing group.

```
$d[0] < 0
```

- Select all groups to the north-east of the processing group.

```
$d[0] > 0 && $d[1] > 0
```

- The absolute distance in the first (eg x) dimension is less than 100,000 AND the signed distance is greater than 100,000. This will result in a neighbourhood that is a column of groups 200,000 map units east-west, and including all groups 100,000 map units north of the processing group. Not that you would normally want a neighbourhood like this...

```
$D[0] <= 100000 && $d[1] >= 100000
```

- Select everything north of 6000000 (e.g. if using UTM coordinates as axes 0 and 1). This is an example that could be used as a definition query, and will not work well as a neighbourhood (use `$nbr_y` instead of `$y` for that).

```
$y > 6000000
```

- Select everything within a rectangle. This is another useful definition query.

```
$y > 6000000 && $y <= 6100000 && $x > 580000 && $x <= 600000
```

- Select a specific processing coord (`495:595`), useful as a definition query to use only one group. Note the use of the `eq` operator - this matches text.

```
$coord_id1 eq '495:595'
```

4.2 Declaring variables and using more complex functions

Variable declaration is done as per Perl syntax. For example:

```
my $some_var = 10;
return ($D / $some_var) <= 100;
```

This trivial example evaluates to true if the absolute distance divided by 10 (the value in variable `$some_var`) is less than 100. The semicolon denotes a separation of statements to be processed in sequence, such that this example could be written on one line. The result of the last statement is what is returned to the analysis to determine if the group is part of the neighbourhood or not. It is evaluated as true or false. The word `return` is not actually needed in this case, but does make things clearer when there are multiple lines of code.

A more complex function might involve an ellipse (although you could just use `sp_ellipse (major_radius => 300000, minor_radius => 100000, rotate_angle => 1.5714)`)

```
my $major_radius = 300000; # longest axis
my $minor_radius = 100000; # shortest axis

# set the offset in radians, anticlockwise (1.5714 = PI/2 = north)
my $rotate_angle = 1.5714;

# now calc the bearing to rotate the coords by
my $bearing = atan2 ($d[0], $d[1]) + $rotate_angle;

#  and rotate them
my $r_x = cos ($bearing) * $D; # rotated x coord
my $r_y = sin ($bearing) * $D; # rotated y coord

#  get the scaled distances in each direction
my $a_dist = ($r_y ** 2) / ($major_radius ** 2);
my $b_dist = ($r_x ** 2) / ($minor_radius ** 2);
```

```
# this last line evaluates to 1 (true) if the candidate
#   neighbour is within or on the edge of the ellipse,
#   and 0 (false) otherwise
return ($a_dist + $b_dist) <= 1;
```

Note the use of the word `my`. This is required to declare your own variables in the correct scope. If it is not used then the variables will not work properly. Do not declare any of the pre-calculated Biodiverse variables with this (`$D` etc) - they already exist and redeclaring will overprint them, causing unpredictable results.

If you wish to know if your function works with the spatial index then run a moving window analysis twice, once with and once without the spatial index, using the list indices generated by the Element Counts calculations to get the lists of neighbours. Export the results to CSV and use a difference tool to compare the results.

Functions available by default are those in the Math::Trig library, plus POSIX::fmod().

To access environment variables you have set, just use `$ENV{variable_name}`, eg `$ENV{my_default_radius}`.