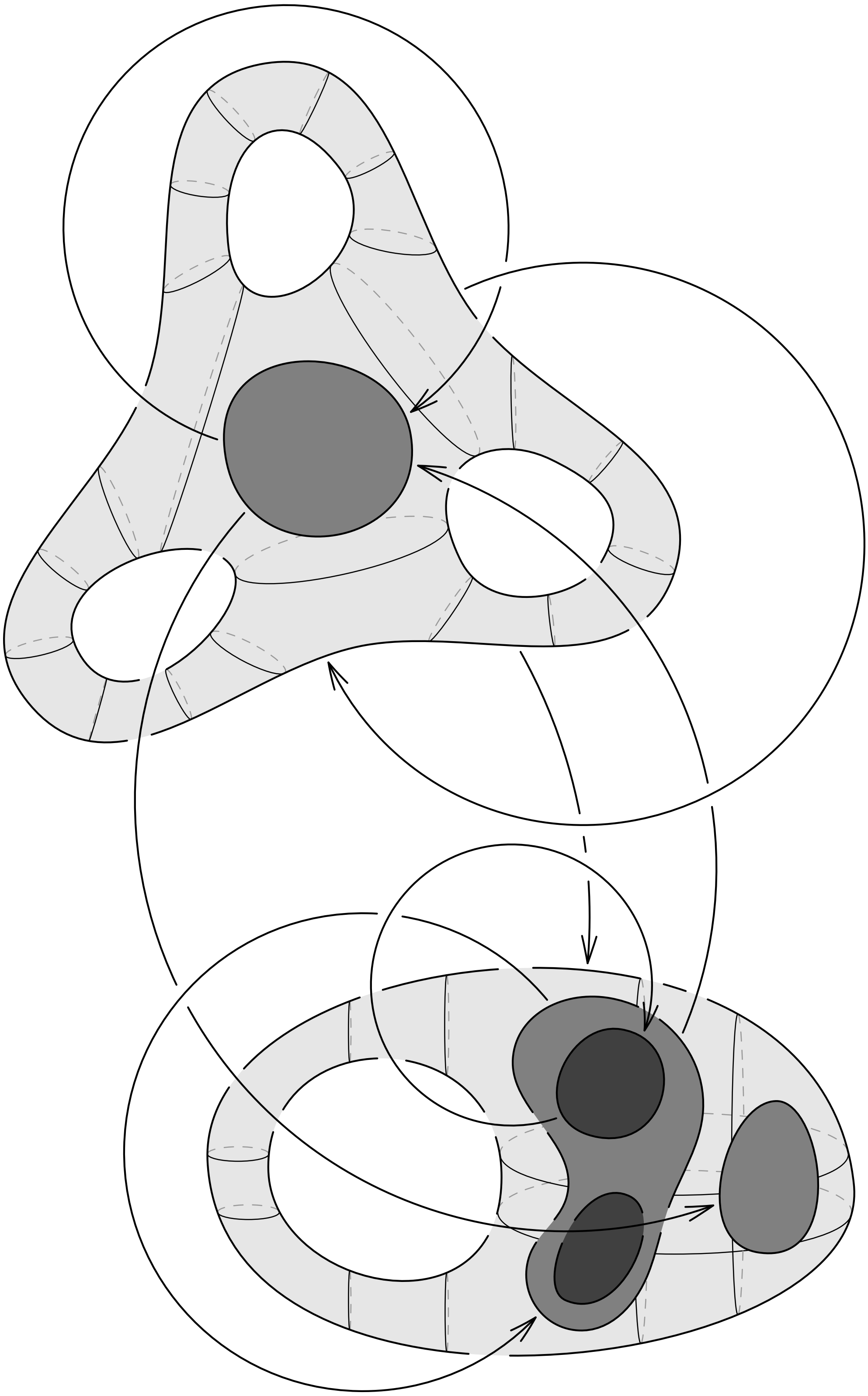
`smoothmanifold.asy - v6.3.1`

Diagrams in higher mathematics with Asymptote

by Roman Maksimovich



Abstract

This document contains the full description and user manual to the `smoothmanifold` Asymptote module, home page https://github.com/thornoar/smoothmanifold.

Contents

1. Introduction

In higher mathematics, diagrams often take the form of “blobs” (representing sets and their subsets) placed on the plane, connected with paths or arrows. This is particularly true for set theory and topology, but other more advanced fields inherit this style. In differential geometry and multivariate calculus, one draws spheres, tori, and other surfaces in place of these “blobs”. In category theory, commutative diagrams are commonplace, where the “blobs” are replaced by symbols. Here are a couple of examples, all drawn with `smoothmanifold`:

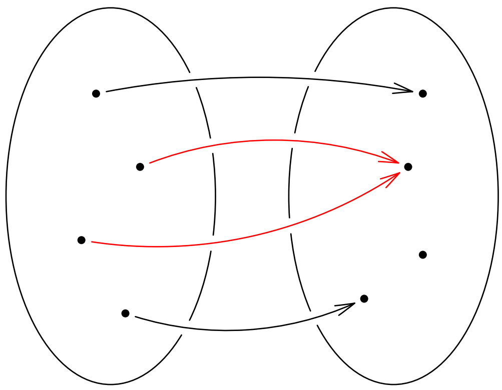


Figure 1: An illustration of non-injectivity (set theory)

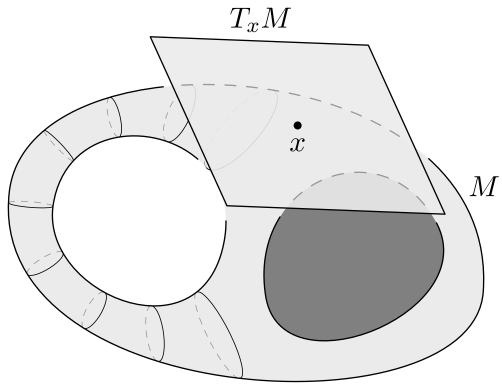


Figure 2: Tangent space at a point on a manifold (diff. geometry)

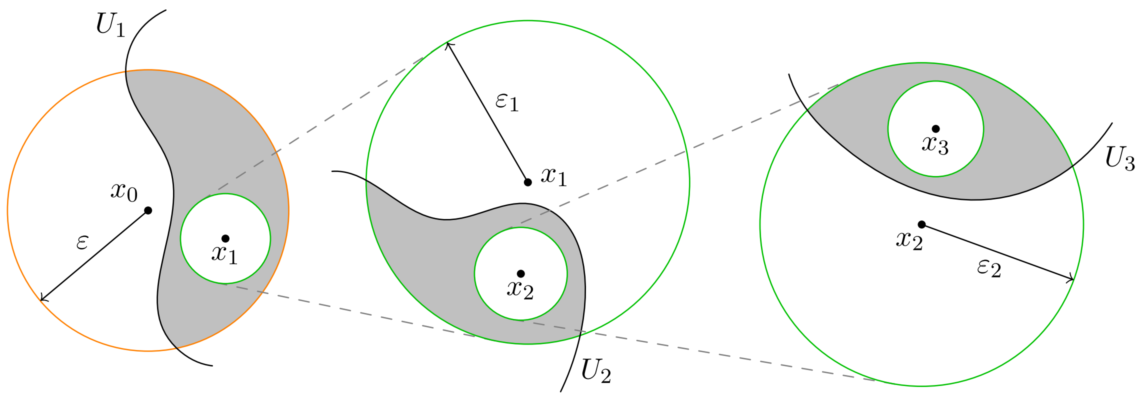


Figure 3: The proof of the Baire category theorem (topology)

Take special note of the gaps that arrows leave on the boundaries of the ovals on Figure 1. I find this feature quite hard to achieve in plain Asymptote, and module `smoothmanifold` uses some dark magic to implement it. Similarly, note the shaded areas on Figure 3. They represent intersections of areas bounded by two paths. Finding the bounding path of such an intersection is non-trivial and also implemented in `smoothmanifold`. Lastly, Figure 2 shows a three-dimensional surface, while the picture was fully drawn in 2D. The illusion is achieved through these cross-sectional “rings” on the left of the diagram.
To summarize, the most prominent features of module `smoothmanifold` are the following:

- **Gaps in overlapping paths**, achieved through a system of deferred drawing;
- **Set operations on paths bounding areas**, e.g. intersection, union, set difference, etc.;
- **Three-dimensional drawing**, achieved through an automatic (but configurable) addition of cross sections to smooth objects.

Do take a look at the source code for the above diagrams, to get a feel for how much heavy lifting is done by the module, and what is required from the user. We will now consider each of the above mentioned features (and some others as well) in full detail.

2. Deferred drawing and path overlapping

2.1. The general mechanism

In the `picture` structure, the paths drawn on a picture are not stored in an array, but rather indirectly stored in a `void` callback. That is, when the `draw` function is called, the *instruction to draw* the path is added to the picture, not the path itself. This makes it quite impossible to "modify the path after it is drawn". To go around this limitation, `smoothmanifold` introduces an auxiliary struct:

```
struct deferredPath {
    path[] g;
    pen p;
    int[] under;
    tarrow arrow;
    tbar bar;
}
```

It stores the path(s) to draw later, and how to draw them. Now, `smoothmanifold` executes the following steps to draw a "mutable" path `p` to a picture `pic` and then draw it for real:

1. Have a global two-dimensional array, say `arr`, of `deferredPath`'s;
2. Construct a `deferredPath` based on `p`, say `dp`;
3. Exploit the `nodes` field of the `picture` struct to store an integer. Retrieve this integer, say `n`, from `pic` (or create one if the `nodes` field doesn't contain it).
4. Store the deferred path `dp` in the one-dimensional array `arr[n]`;
5. Move on with the original code, perhaps modifying the deferred path `dp` in `arr` as needed, e.g. adding gaps;
6. At shipout time, when processing the picture `pic`, retrieve the index `n` from its `nodes` field and draw all `deferredPath` objects in the array `arr[n]`.

All these steps require no extra input from the user, since the shipout function is redefined to do them automatically. One only needs to use the `fitpath` function instead of `draw`.

2.2. The `tarrow` and `tbar` structures

Similarly to drawing paths to a picture, arrows and bars are implemented through a function type `bool(picture, path, pen, margin)`, typedef'ed as `arrowbar`. Moreover, when this arrowbar is called, it automatically draws not only itself, but also the path is was attached to. This makes it impossible to attach an arrowbar to a path and then mutate the path — the arrowbar will remember the path's original state. Hence, `smoothmanifold` implements custom arrow/bar implementations:

```
struct tarrow {
    arrowhead head;
    real size;
    real angle;
    filltype ftype;
    bool begin;
    bool end;
    bool arc;
}
```

```
struct tbar {
    real size;
    bool begin;
    bool end;
}
```

These structs store information about the arrow/bar, and are converted to regular arrowbars when the corresponding path is drawn to the picture. For creating new `tarrow`/`tbar` instances and converting them to arrowbars, the following functions are available:

```
tarrow DeferredArrow(
    arrowhead head = DefaultHead,
    real size = 0,
    real angle = arrowangle,
    bool begin = false,
    bool end = true,
    bool arc = false,
    filltype filltype = null
)
arrowbar convertarrow(
    tarrow arrow,
    bool overridebegin = false,
    bool overrideend = false
)
```

```
tbar DeferredBar(
    real size = 0,
    bool begin = false,
    bool end = false
)
arrowbar convertbar(
    tbar bar,
    bool overridebegin = false,
    bool overrideend = false
)
```

The `overridebegin` and `overrideend` options let the user force disable the arrow/bar at the beginning/end of the path.

2.3. The `fitpath` function

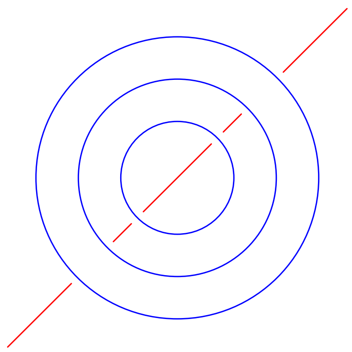
This is a substitute for the plain `draw` function. The `fitpath` function implements steps 1-4 of the deferred drawing system describes above.

```
void fitpath (picture pic, path gs, bool overlap, int covermode, bool drawnow, Label L,
pen p, tarrow arrow, tbar bar)
```

Arguments:

- `pic` — the picture to fit the path to;
- `gs` — the path to fit;
- `overlap` — whether to let the path overlap the previously fit paths. A value of `false` will lead to gaps being left in all paths that `gs` intersects;
- `covermode` — if the path `gs` is cyclic, this option lets you decide what happens to the parts of previously fit paths that fall "inside" of `gs`. Suppose a portion `s` of another path falls inside the cyclic path `gs`. Then
  - `covermode == 2`: The portion `s` will be erased completely;
  - `covermode == 1`: The portion `s` will be "demoted to the background" — either temporarily removed or drawn with dashes;
  - `covermode == 0`: The portion `s` will be drawn like the rest of the path;
  - `covermode == -1`: If the portion `s` is "demoted", it will be brought "back to the surface", i.e. drawn with solid pen. Otherwise, it will be draw as-is.

  Consider the following example:



```
import smoothmanifold;
path l = (-1.2,-1.2) -- (1.2,1.2);
path c1 = unitcircle;
path c2 = scale(.7) * unitcircle;
path c3 = scale(.4) * unitcircle;
fitpath(l, red);
fitpath(c1, blue, covermode = 1);
fitpath(c2, blue, covermode = -1);
fitpath(c3, blue, covermode = 0);
```

Figure 4: A showcase of the `fitpath` function

- `drawnow` — whether to draw the path `gs` immediately to the picture. When `drawnow == true`, the path `gs` leaves gaps in other paths, but is immutable itself, i.e. later fit paths will not leave any gaps in it.

When `drawnow == false`, the path `gs` is not immediately drawn, but rather saved to be mutated and finally drawn at shipout time;
- `L` — the label to attach to `gs`. This label is drawn to `pic` immediately on call of `fitpath`, unlike `gs`;
- `p` — the pen to draw `gs` with;
- `arrow` — the arrow to attach to the path. Note that the type is `tarrow`, not `arrowbar`;
- `bar` — the bar to attach to the path. Note that the type is `tbar`, not `arrowbar`.

Apart from different types of the `arrow`/`bar` arguments, the `fitpath` function is identical to `draw` in type signature, and they can be used interchangeably. Moreover, there are overloaded versions of `fitpath`, where parameters are given default values (one of these versions is used in the example above):

```
void fitpath (
    picture pic = currentpicture,
    path g,
    bool overlap = config.drawing.overlap,
    int covermode = 0,
    Label L = "",
    pen p = currentpen,
    bool drawnow = config.drawing.drawnow,
    tarrow arrow = null,
    tbar bar = config.arrow.currentbar
)
```

```
void fitpath (
    picture pic = currentpicture,
    path[] g,
    bool overlap = config.drawing.overlap,
    int covermode = 0,
    Label L = "",
    pen p = currentpen,
    bool drawnow = config.drawing.drawnow
)
```

Here, `config` is the global configuration structure, see Section 5. Furthermore, there are corresponding `fillfitpath` functions that serve the same purpose as `filldraw`.

2.4. Other related routines

```
int extractdeferredindex (picture pic)
```

Inspect the `nodes` field of `pic` for a string in a particular format, and, if it exists, extract an integer from it.

```
deferredPath[] extractdeferredpaths (picture pic, bool createlink)
```

Extract the deferred paths associated with the picture `pic`. If `createlink` is set to `true` and `pic` has no integer stored in its `nodes` field, the routine will find the next available index and store it in `pic`.

```
path[] getdeferredpaths (picture pic = currentpicture)
```

A wrapper around `extractdeferredpaths`, which concatenates the `path[] g` fields of the extracted deferred paths.

```
void purgedeferredunder (deferredPath[] curdeferred)
```

For each deferred path in `curdeferred`, delete the segments that are "demoted" to the background (i.e. going under a cyclic path, drawn with dashed lines).

```
void drawdeferred (
    picture pic = currentpicture,
    bool flush = true
)
```

Render the deferred paths associated with `pic`, to the picture `pic`. If `flush` is `true`, delete these deferred paths.

```
void flushdeferred (picture pic = currentpicture)
```

Delete the deferred paths associated with `pic`.

```
void plainshipout (...) = shipout;
shipout = new void (...)
{
    drawdeferred(pic = pic, flush = false);
    draw(pic = pic, debugpaths, red+1);
    plainshipout(prefix, pic, orntn, format, wait, view, options, script, lt, P);
};
```

A redefinition of the `shipout` function to automatically draw the deferred paths at shipout time. For a definition of `debugpaths`, see Section 4.4.

The functions `erase`, `add`, `save`, and `restore` are redefined to automatically handle deferred paths.

3. Operations on paths

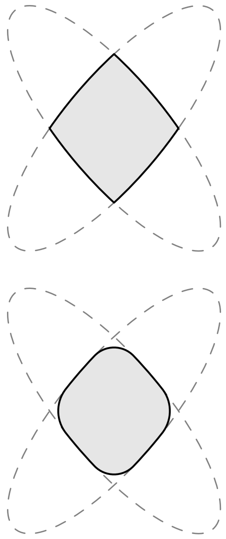
3.1. Set operations on bounded regions

Module `smoothmanifold` defines a routine called `combination` which, given two *cyclic* paths `p` and `q`, calculates a result path which encloses a region that is a combination of the regions `p` and `q`:

```
path[] combination (path p, path q, int mode, bool round, real roundcoeff)
```

This function returns an array of paths because the combination of two bounded regions may be bounded by multiple paths. Rundown of the arguments:

- `p` and `q` — cyclic paths bounding the regions to combine;
- `mode` — an internal parameter which allows to specialize `combination` for different purposes;
- `round` and `roundcoeff` — whether to round the sharp corners of the resulting bounding path(s).



```
<...>
filldraw(
    combination(p, q, 1, false, 0), // <- no rounding
    drawpen = linewidth(.7),
    fillpen = palegrey
);
<...>
<...>
filldraw(
    combination(p, q, 1, true, .04), // <- yes rounding
    drawpen = linewidth(.7),
    fillpen = palegrey
);
<...>
```

Figure 5: A showcase of the `round` and `roundcoeff` parameters

Based on different values for the `mode` parameter, the module defines the following specializations:

```
path[] difference (
  path p,
  path q,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] operator - (path p, path q)
{ return difference(p, q); }
```

Calculate the path(s) bounding the set difference of the regions bounded by `p` and `q`. The `correct` parameter determines whether the paths should be "corrected", i.e. oriented clockwise.

```
path[] symmetric (
  path p,
  path q,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] operator :: (path p, path q)
{ return symmetric(p, q); }
```

Calculate the path(s) bounding the set symmetric difference of the regions bounded by `p` and `q`.

```
path[] intersection (
  path p,
  path q,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] operator ^ (path p, path q)
{ return intersection(p, q); }
```

Calculate the path(s) bounding the set intersection of the regions bounded by `p` and `q`. The following array versions are also available:

```
path[] intersection (
  path[] ps,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] intersection (
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
  ... path[] ps
)
```

Inductively calculate the total intersection of an array of paths.

```
path[] union (
  path p,
  path q,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] operator | (path p, path q)
{ return union(p, q); }
```

Calculate the path(s) bounding the set union of the regions bounded by `p` and `q`. The corresponding array versions are available:

```
path[] union (
  path[] ps,
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
)
```

```
path[] union (
  bool correct = true,
  bool round = false,
  real roundcoeff = config.paths.roundcoeff
  ... path[] ps
)
```

Inductively calculate the total union of an array of paths. Here is an illustration of the specializations:

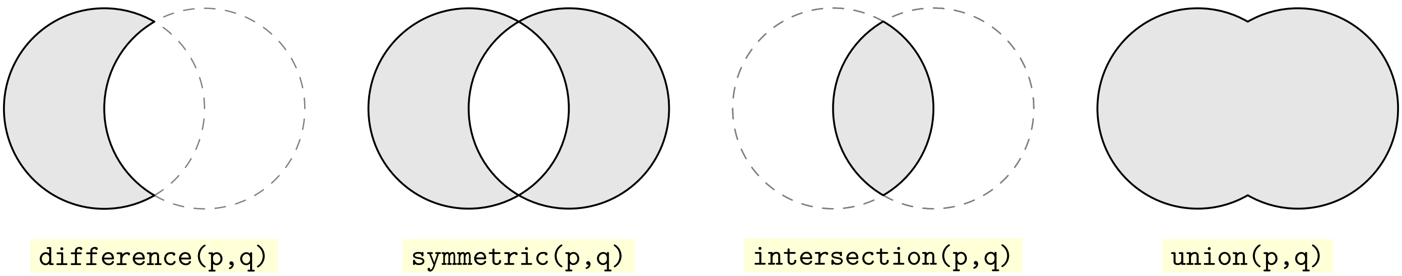


Figure 6: Different specializations of the `combination` function

3.2. Other path utilities

Module `smoothmanifold` features dozens of useful auxiliary path utilities, all of which are listed below.

```
path[] convexpaths = { ... }
path[] concavepaths = { ... }
```

Predefined collections of convex and concave paths (14 and 7 paths respectively), added for user convenience.

```
path randomconvex ()
path randomconcave ()
```

Allows the user to sample a random path from the above arrays.

```
path ucircle = reverse(unitcircle);
path usquare = (1,1) -- (1,-1) -- (-1,-1) -- (-1,1) -- cycle;
```

Slightly changed versions of the `unitcircle` and `unitsquare` paths. Most notably, these are *clockwise,* since most of this module-s functionality prefers to deal with clockwise paths.

```
pair center (path p, int n = 10, bool arc = true, bool force = false)
```

Calculate the center of mass of the region bounded by the cyclic path `p`. If `force` is `false` and the center of mass is outside of `p`, the routine uses a heuristic to return another point, inside of `p`.

```
bool insidepath (path p, path q)
```

Check if path `q` is completely inside the cyclic path `p` (directions of `p` and `q` do not matter).

```
real xsize (path p) { return xpart(max(p)) - xpart(min(p)); }
real ysize (path p) { return ypart(max(p)) - ypart(min(p)); }
```

Calculate the horizontal and vertical size of a path.

```
real radius (path p) { return (xsize(p) + ysize(p))*.25; }
```

Calculate the approximate radius of the region enclosed by `p`.

```
real arclength (path g, real a, real b) { return arclength(subpath(g, a, b)); }
```

A more general version of `arclength`.

```
real relarctime (path g, real t0, real a)
```

Calculate the time at which arclength `a` will be traveled along the path `g`, starting from time `t0`.

```
path arcsubpath (path g, real arc1, real arc2)
```

Calculate the subpath of `g`, starting from arclength `arc1`, and ending with arclength `arc2`.

```
real intersectiontime (path g, pair point, pair dir)
```

Calculate the time of the intersection of `g` with a beam going from `point` in direction `dir`

```
pair intersection (path g, pair point, pair dir)
```

Same as `intersectiontime`, but returns the point instead of the intersection time.

```
path reorient (path g, real time)
```

Shift the starting point of the cyclic path `g` by time `time`. The resulting path will be same as `g`, but will start from time `time` along `g`.

```
path turn (path g, pair point, pair dir)
{ return reorient(g, intersectiontime(g, point, dir)); }
```

A combination of `reorient` and `intersectiontime`, that shifts the starting point of the cyclic path `g` to its intersection with the ray cast from `point` in the direction `dir`.

```
path subcyclic (path p, pair t)
```

Calculate the subpath of the *cyclic* path p, from time `t.x` to time `t.y`. If `t.y < t.x`, the subpath will still go in the direction of `g` instead of going backwards.

```
bool clockwise (path p)
```

Determine if the cyclic path `p` is going clockwise.

```
bool meet (path p, path q) { return (intersect(p, q).length > 0); }
bool meet (path p, path[] q) { ... }
bool meet (path[] p, path[] q) { ... }
```

A shorthand function to determine if two (or more) paths have an intersection point.

```
pair range (path g, pair center, pair dir, real ang, real orient = 1)
```

Calculate the begin and end times of a subpath of `g`, based on `center`, `dir`, and `angle`, as such:

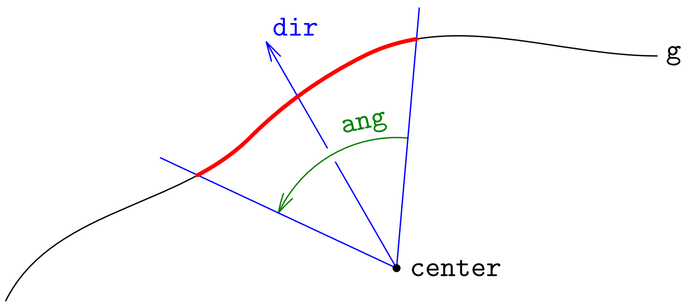


Figure 7: An illustration of the `range` function

If `orient` is set to `-1` instead of `1`, then the returned times are switched.

```
bool outsidepath (path p, path q)
```

Check if `q` is completely outside (that is, inside the complement) of the region enclosed by `p`.

```
path ellipsepath (pair a, pair b, real curve = 0, bool abs = false)
```

Produce half of an ellipse connecting points `a` and `b`. Curvature may be relative or absolute.

```
path curvedpath (pair a, pair b, real curve = 0, bool abs = false)
```

Constuct a curved path between two points. Curvature may be relative (from 0 to 1) or absolute.

```
path cyclepath (pair a, real angle, real radius)
```

A circle of radius `radius`, starting at `a` and turned at `angle`.

```
path midpath (path g, path h, int n = 20)
```

Construct the path going “between” `g` and `h`. The parameter `n` is the number of sample points, the more the more precise the output.

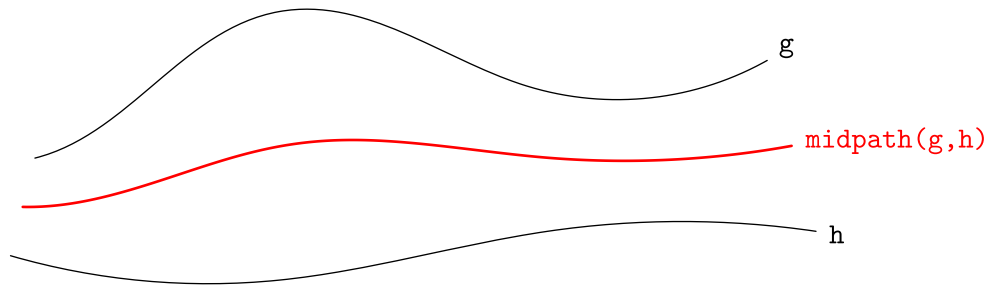


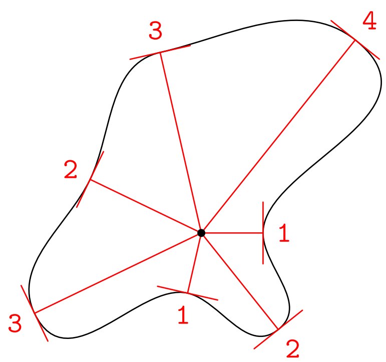
Figure 8: An illustration of the `midpath` function

```
path connect (pair[] points)
path connect (... pair[] points)
```

Connect an array of points with a path.

```
path wavypath (real[] nums, bool normaldir = true, bool adjust = false)
path wavypath (... real[] nums)
```

Generate a clockwise cyclic path around the point `(0,0)`, based on the `nums` parameter. If `normaldir` is set to `true`, additional restrictions are imposed on the path. If `adjust` is `true`, then the path is shifted and scaled such that its `center` [page 9] is `(0,0)`, and its `radius` [page 9] is `1`. Consider the following example:




```
real[] nums = {1,2,1,3,2,3,4};
bool normaldir = true;

draw(wavypath(nums, normaldir));

for (int i = 0; i < nums.length; ++i) {
  <...> // draw numbers
}

dot((0,0));
```

Figure 9: A showcase of the `wavypath` function

```
path connect (path p, path q)
```

Connect the paths `p` and `q` smoothly.

```
pair randomdir (pair dir, real angle)
{ return dir(degrees(dir) + (unitrand()-.5)*angle); }
path randompath (pair[] controlpoints, real angle)
```

Create a pseudo-random path passing through the `controlpoints`. The `angle` parameter determines the "spread" of randomness. Here's an example:

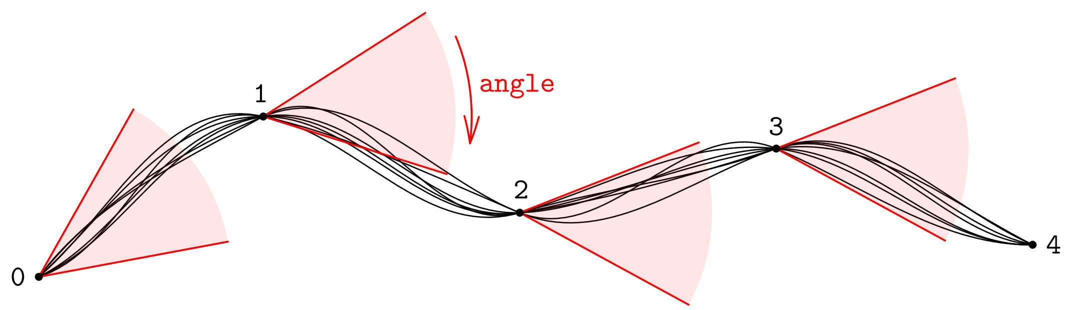


Figure 10: A showcase of the `randompath` function

```
path neigharc (
    real x = 0,
    real h = config.paths.neighheight,
    int dir = 1,
    real w = config.paths.neighwidth
)
```

Draw an "open neighborhood bracket" on the real line, like so:

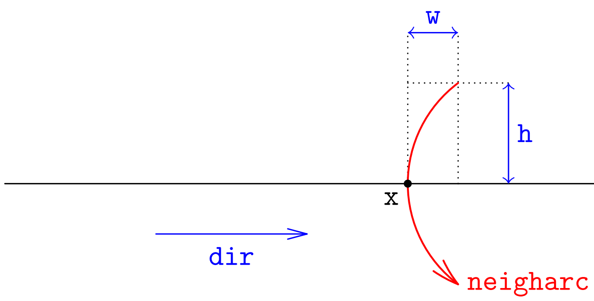


Figure 11: A showcase of the `neigharc` function

4. Smooth objects

4.1. Definition of the smooth, hole, subset, and element structures

The `smoothmanifold` module's original purpose was to introduce a suitable abstraction to simplify drawing blobs on the plane. The `smooth` structure is, perhaps, the oldest part of `smoothmanifold`, that has persisted through countless updates and changes. In its current form, here is how it's defined:

```
struct smooth {
    path contour;
    pair center;

    string label;
    pair labeldir;
    pair labelalign;

    hole[] holes;
    subset[] subsets;
    element[] elements;

    transform unitadjust;

    real[] hratios;
    real[] vratios;

    bool isderivative;

    smooth[] attached;

    void drawextra (dpar, smooth);

    static smooth[] cache;
}
```

```
struct hole {
    path contour;
    pair center;
    real[][] sections;
    int scnumber;
}

struct subset {
    path contour;
    pair center;
    string label;
    pair labeldir;
    pair labelalign;
    int layer;
    int[] subsets;
    bool isderivative;
    bool isonboundary;
}

struct element {
    pair pos;
    string label;
    pair labelalign;
}
```

Every `smooth` object has a:

- `contour` — the clockwise cyclic path that serves as a boundary of the object;
- `center` — the center of the object, usually inferred automatically with `center(contour)` [page 9];
- `label` — a string label, e.g. `"$A$"` or `"$S$"`, that will be displayed when drawing the smooth object;

- `labeldir` and `labelalign` — they determine where the label is to be drawn, namely at `intersection(contour, center, labeldir)` [page 9], with `labelalign` as `align`.

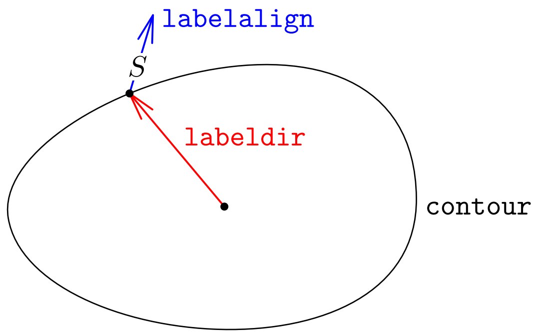


Figure 12: A showcase of the `labeldir` and `labelalign` fields

- `holes` — an array of `hole` structures, each of which has a:
  - `contour` — the clockwise cyclic boundary of the hole;
  - `center` — the center of the hole, typically calculated automatically by `center(contour)` [page 9];
  - `sections` — a two-dimensional array that determines where to draw the cross sections seen on Figure 2. For a detailed description, see Section 4.7.2;
  - `scnumber` — the maximum amount of cross sections that the hole allows other holes to share with it. These are sections *between holes,* not ones between a hole and the contour of the `smooth` object. For details, see Section 4.7.2;
- `subsets` — an array of `subset` structures, each of which has a:
  - `contour` — the clockwise cyclic boundary of the subset;
  - `center` — the center of the subset, likewise usually determined by `center(contour)` [page 9];
  - `label`, `labeldir`, `labelalign` — serve the same purpose as the respective fields of the `smooth` object;
  - `layer` — an integer determining the "depth" of the subset. A toplevel subset will have `layer == 0`, its subsets will have `layer == 1`, their subsets will have `layer == 2`, etc. This way, a hierarchy of subsets is established. For details, see Section 4.4;
  - `subsets` — an index `i` is an element of this array if and only if the subset `subsets[i]` (taken from the `subsets` field of the parent `smooth` object) is a subset of the current subset. For details, see Section 4.4;
  - `isderivative` — a flag that marks all automatically created subsets (i.e. those that represent intersections of existing subsets). For details, see Section 4.4;
  - `isonboundary` — a flag that marks if the current subset touches the boundary of another subset.
- `elements` — an array of `element` structures, each of which has a:
  - `pos` — the position of the element;
  - `label` — the label attached to the element, e.g. `"$x$"` or `"$y_0$"`;
  - `labelalign` — how to align the label when drawing the element;
- `unitadjust` — a transform that converts from unit coordinates of the smooth object to the global user coordinates (see Section 4.5);
- `hratios` and `vratios` — two arrays that determine where to draw cross sections in the `cartesian` mode. For details, see Section 4.7.3;
- `isderivative` — similarly to `subset`, this field marks those `smooth` objects which are obtained from preexisting objects through operations of intersection, union, etc.;
- `attached` — this field allows to bind an array of smooth objects to the current one. Drawing the current object will trigger drawing all of its `attached` objects. For example, the tangent space seen on Figure 2 is `attached` to the main object;
- `drawextra` — a callback to be executed *after the `smooth` object is drawn.* It takes as parameters a drawing configuration of type `dpar` (see Section 5.5) and the current `smooth` object;
- `static cache` — a global array of all `smooth` objects constructed so far. It is used mainly to search for smooth objects by label. See Section 4.6.

4.2. Construction

Each of the four structures is equipped with a sophisticated `void operator init` that will infer as much information as possible. To construct a `smooth`, `hole`, or `subset`, it is only necessary to pass a `contour`. All other fields can be set in the constructor, but they are optional. To construct an `element`, it is only necessary to pass a `pos`, the `label` and `labelalign` fields have default values.

4.3. Query and mutation methods

Already constructed structures can be queried and modified in a plethora of ways. Most methods return `this` at the end of execution for convenience.

4.3.1. smooth objects

```
real xsize () { return xsize(this.contour); }
real ysize () { return ysize(this.contour); }
```

Calculate the vertical and horizontal size of `this`.

```
bool inside (pair x)
```

Check if `x` lies inside the contour of `this`, but not inside any of its `holes`.

```
smooth move (
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = this.center,
    bool readjust = true,
    bool drag = true
)
```

```
smooth shift (explicit pair shift)

smooth shift
(real xshift, real yshift = 0)

smooth scale (real scale)

smooth rotate (real rotate)
```

Scale `this` by `scale` (with center at `point`), rotate by `rotate` around `point`, and then shift by `shift`. If `readjust` is `true`, also recalculate the `unitadjust` field. If `drag` is `true`, also apply the `move` to all smooth objects `attached` to `this`. In the end return `this`. The `shift`, `scale` and `rotate` methods on the right are all specializations of the `move` method.

```
void xscale (real s)
```

Scale `this` by `s` along the x-axis.

```
smooth dirscale (pair dir, real s)
```

Scale `this` by `s` in the direction `dir`. Return `this`.

```
smooth setcenter (
    int index = -1,
    pair center = config.system.dummypair,
    bool unit = config.smooth.unit
)
```

Set the center of `this` if `index == -1`, and the center of `this.subsets[index]` otherwise. If `unit` is `true`, interpret `center` in the unit coordinates of `this` (i.e. apply `this.unitadjust` to `center`). For the definition of `config.system.dummypair`, see Section 5.1.

```
smooth setlabel (
    int index = -1,
    string label = config.system.dummystring,
    pair dir = config.system.dummypair,
    pair align = config.system.dummypair
)
```

Set the `label`, `labeldir` and `labelalign` of `this` if `index == -1`, and set these fields of `this.subsets[index]` otherwise. For the definition of `config.system.dummystring`, see Section 5.1.

```
smooth addelement (
    pair pos,
    string label = "",
    pair align = 1.5*S,
    int index = -1,
    bool unit = config.smooth.unit
)
```

```
smooth addelement (
    element elt,
    int index = -1,
    bool unit = config.smooth.unit
)
```

Add a new element to `this`. Return `this`. If `index >= 0`, then the function will additionally check if the `element` is contained within the contour of `this.subsets[index]`. If `unit` is true, then the position of the `element` is interpreted in `this` object's unit coordinates.

```
smooth setelement (
    int index,
    pair pos = config.system.dummypair,
    string label = config.system.dummystring,
    pair labelalign = config.system.dummypair,
    int sbindex = -1,
    bool unit = config.smooth.unit
)
```

Change the fields of the element of `this` at index `index`. Return `this`. Slightly different versions of this routine are also defined in the source code, feel free to peruse it.

```
smooth rmelement (int index)
```

Delete an element at index `index`. Return `this`.

```
smooth movelement (int index, pair shift)
```

Move the element at index `index` by `shift`. Return `this`.

```
smooth addhole (
    path contour,
    pair center = config.system.dummypair,
    real[][] sections = {},
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = center(contour),
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
```

```
smooth addhole (
    hole hl,
    int insertindex = this.holes.length,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
```

Add a new hole to `this`. If `clip` is `true` and the hole's contour is *not* contained within `this` object's contour, then clip `this` smooth object's contour using `difference` [page 7], like so:

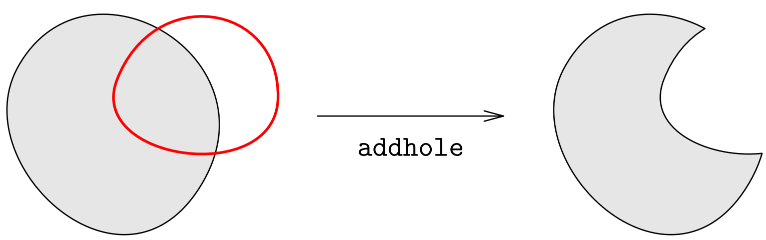


Figure 13: A showcase of the `clip` parameter

If `unit` is `true`, then the hole contour will be treated in the unit coordinates, i.e. the `unitadjust` transform will be applied to it. See Section 4.5 for more details.

```
smooth addholes (
    hole[] holes,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
smooth addholes (
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
    ... hole[] holes
)
```

```
smooth addholes (
    path[] contours,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
smooth addholes (
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
    ... path[] contours
)
```

Auxiliary routines made for conveniently adding multiple holes at the same time.

```
smooth rmhole (int index = this.holes.length-1)
```

Delete the hole under index `index` from `this`.

```
smooth rmholes (int[] indices)
smooth rmholes (... int[] indices)
```

Delete a number of holes from `this`.

```
smooth movehole (
    int index,
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = this.holes[index].center,
    bool movesections = false
)
```

Move the hole at index `index` by scaling and rotating it around `point`, and shifting it by `shift`. If `movesections` is `true`, the `sections` values of the hole are updated accordingly with the transform. This is only really necessary when `rotate` is non-zero.

```
smooth addsection (
    int index,
    real[] section = {}
)
```

```
smooth setsection (
    int index,
    int scindex = 0,
    real[] section = {}
)
```

```
smooth rmsection (
    int index =
      this.holes.length-1,
    int scindex = 0
)
```

Add, set or remove a section under `scindex` in the hole under `index`. When adding, the section will have to pass the

```
bool checksection (real[] section)
```

check.

```
smooth addsubset (
    subset sb,
    int index = -1,
    bool inferlabels = config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit,
    bool checkintersection = true
)
```

Add a new subset to `this`. The meaning of the arguments is as follows:

- `sb` — the subset to add;
- `index` — the index of another, preexisting subset of `this`, such that `sb` should be made a subset of `this.subsets[index]`. If `index` is `-1`, then `sb` is considered a toplevel subset until found otherwise by

containment checks. Essentially, the `index` parameter saves the algorithm some work figuring out where to fit `sb` in the subset hierarchy. See Section 4.4 for more explanation;

- `inferlabels` — if set to `true`, the intersection subsets arising from the addition of `sb` will be given labels like `"$A \cap B$"`, given that some subsets have labels `"$A$"` and `"$B$"`;
- `clip` — if set to `false`, this leads to an error whenever `sb`'s contour is out of bounds with `this` objects's contour. If `clip` is set to true, then `sb`'s contour is clipped instead, and its `isonboundary` field is set to `true`;
- `unit` — as usual, setting this to `true` leads to `sb` being interpreted in `this` objects's unit coordinates, see Section 4.5;
- `checkintersection` — if set to `false`, the routine will *not* perform out-of-bounds checks. This can significantly increase efficiency when the user is confident in the correctness of the call. But then you only have yourself to blame when your subsets are sticking out of your smooth objects!

```
smooth addsubset (
    int index = -1,
    path contour,
    pair center = config.system.dummypair,
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = center(contour),
    string label = "",
    pair dir = config.system.dummypair,
    pair align = config.system.dummypair,
    bool inferlabels = config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
```

A convenience routine that creates the subset from given parameters and calls `addsubset` [page 16] on it.

```
smooth addsubsets (
    subset[] sbs,
    int index = -1,
    bool inferlabels =
config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
smooth addsubsets (
    int index = -1,
    bool inferlabels =
config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
    ... subset[] sbs
)
```

```
smooth addsubsets (
    path[] contours,
    int index = -1,
    bool inferlabels =
config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
smooth addsubsets (
    int index = -1,
    bool inferlabels =
config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
    ... path[] contours
)
```

Further convenience routines that allow adding multiple subsets.

```
smooth rmsubset (
    int index = this.subsets.length-1,
    bool recursive = true
)
```

```
smooth rmsubsets (
    int[] indices,
    bool recursive = true
)
// (... indices)
```

Remove one or more subsets from `this`. If `recursive` is set to `true`, then all subsets of the removed subset shall also be removed.

```
smooth movesubset (
    int index = this.subsets.length-1,
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = config.system.dummypair,
    bool movelabel = false,
    bool recursive = true,
    bool bounded = true,
    bool clip = config.smooth.clip,
    bool inferlabels = config.smooth.inferlabels,
    bool keepview = true
)
```

Move the subset at index `index` (say, `sb`), scaling and rotating it around `point`, and then shifting it by `shift`. The meaning of the `bool` parameters is as follows:

- `movelabel` — if set to `true`, the `labeldir` of `sb` is rotated with the subset itself;
- `recursive` — if set to `true`, all subsets of `sb` are moved as well;
- `bounded` — if set to `true`, the movement of `sb` is restricted by its subsets and supersets;
- `clip` — if set to `true`, the contour of `sb` will be clipped if it becomes out-of-bounds as a result of the movement;
- `inferlabels` — same as the corresponding parameter of the `addsubset` [page 16] function.

```
smooth attach (smooth sm)
```

Add the smooth object `sm` to the `attached` field of `this`. Return `this`.

```
smooth fit (
    int index = -1,
    picture pic = currentpicture,
    picture addpic,
    pair shift = (0,0)
)
```

Fit an entire picture `pic` into one of the subsets of `this`, namely one under index `index`. If `index` is `-1`, the picture is fit inside the contour of `this`.

```
smooth copy ()
```

```
smooth replicate (smooth sm)
```

Perform a deep copy of `this` and return this copy, or deeply copy all fields from another smooth object `sm`, returning `this`.

4.3.2. `subset` objects

```
real xsize () { return xsize(this.contour); }
real ysize () { return ysize(this.contour); }
```

Calculate the vertical and horizontal size of `this`.

```
subset move (transform move)
```

Move `this` by applying `move` to its `contour`.

```
subset move (pair shift, real scale, real rotate, pair point, bool movelabel)
```

A more sophisticated version of `move` [page 18], which accepts the usual `shift`, `scale`, `rotate`, `point` arguments (see `smooth.move` [page 14]), and moves the subset's `labeldir` based on the `movelabel` flag.

```
subset copy ()
```

```
subset replicate (subset s)
```

Perform a deep copy of `this` and return the copy, or deeply copy all fields of another subset, `s`, into `this`, returning `this`.

4.3.3. hole objects

```
hole move (transform move)
```

Move `this` by applying `move` to the hole's `contour`.

```
hole move (pair shift, real scale, real rotate, pair point, bool movesections)
```

A more sophisticated version of `move` [page 19], which accepts the usual `shift`, `scale`, `rotate`, `point` arguments (see `smooth.move` [page 14]), and rotates the sections of `this` (see Section 4.7.2) if the `movesections` flag is set.

```
hole copy ()
```

```
hole replicate (hole h)
```

Perform a deep copy of `this` and return the copy, or deeply copy all fields of another hole, `h`, into `this`, returning `this`.

4.3.4. element objects

```
element move (transform move)
```

Move `this` by applying `move` to the element's `contour`.

```
element move (pair shift, real scale, real rotate, pair point, bool movelabel)
```

A more sophisticated version of `move` [page 19], which accepts the usual `shift`, `scale`, `rotate`, `point` arguments (see `smooth.move` [page 14]), and rotates this element's `labeldir` if the `movelabel` flag is set.

```
element copy ()
```

```
element replicate (element elt)
```

Perform a deep copy of `this` and return the copy, or deeply copy all fields of another element, `elt`, into `this`, returning `this`.

4.4. The subset hierarchy

In general, a system of subsets of a set can form a complicated network of intersections and inclusions. Consider the following example:

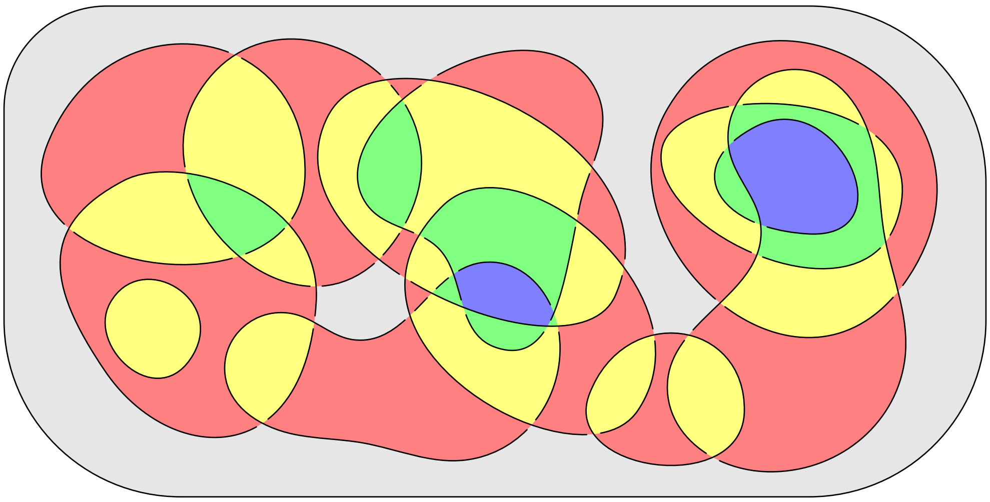


Figure 14: An example of many intersecting subsets

To manage this mess (and to be able to draw the diagram above in under 30 lines of code), `smoothmanifold` employs a *hierarchy of subsets.* This is achieved through every subset having a `layer` field which specifies how deep in the hierarchy it lies (in Figure 14 different layers are colored with different colors). Some points to note:

- The `int[] subsets` field of a subset `sb` structure contains all indices of subsets in the parent `smooth` object, that are direct subsets of `sb`;

- When adding a subset sb with smooth.addsubset [page 16], the algorithm automatically checks its intersections with all other subsets of the smooth object, creating additional subsets, and fits sb on the appropriate layer.

There are two internally important methods of smooth, related to the subset hierarchy:

```
bool onlyprimary (int index)
```

Determine if the subset of this at index index only contains "proper" subsets (that is, those that are not intersections of other subsets).

```
bool onlysecondary (int index)
```

Same as onlyprimary [page 20], but now check if all subsets of this.subsets[index] are intersections of other subsets.

You will not be able to move [page 18] a subset, if it contains both primary ("proper") and secondary subsets, since the situation becomes too complicated and I don't know how to resolve it algorithmically. This is the case because moving a subset should trigger a recalculation of the entire subset hierarchy, which is a generally difficult task.

4.5. Unit coordinates in a smooth object

Sometimes, when working with a smooth object, it is easier not to think about where it is located in user coordinates, and assume that its center [page 12] is at (0,0), and its radius [page 9] is approximately 1. Module smoothmanifold supports a mechanism to allow this, namely the unitadjust [page 12] field of the smooth structure. It establishes a bridge between the object's "unit coordinates" and the global user coordinates. This field is calculated via

```
transform selfadjust ()
{ return shift(this.center)*scale(radius(this.contour)); }
```

Calculate the unit coordinates of this. See unitadjust [page 12] for reference.

The unit coordinates of a subset can also be obtained:

```
transform adjust (int index)
```

Calculate the unit coordinates of the subset of this at index index. If index is set to -1, the unitadjust field of this is used instead.

Now, any method that accepts a bool unit parameter, can accept pairs/paths in the parent object's unit coordinates, since it will convert them to global coordinates by applying unitadjust.

Another unit-related method of the smooth structure is

```
pair relative (pair point)
```

Convert point (given in unit coordinates) to a point in global coordinates.

4.6. Reference by label

The global array smooth.cache [page 12] gives many opportunities, and one of them is *reference by label*. Given a string label, one can loop over the smooth.cache array and search for a smooth object with this label. Moreover, one can inspect the subsets and elements of these smooth objects, and compare their labels to label. In this way, one can obtain a smooth, subset, or element from their label. This gives rise to the following versions of the already familiar smooth methods:

```
smooth setcenter (
    string destlabel,
    pair center,
    bool unit = config.smooth.unit
) { return this.setcenter(findlocalsubsetindex(destlabel), center, unit); }
```

An alternative to setcenter [page 14], but finds the subset by label.

```
smooth setlabel (
    string destlabel,
    string label,
    pair dir = config.system.dummypair,
    pair align = config.system.dummypair
) { return this.setlabel(findlocalsubsetindex(destlabel), label, dir, align); }
```

An alternative to `setlabel` [page 14], but finds the subset by label.

```
smooth setelement (
    string destlabel,
    pair pos = config.system.dummypair,
    string label = config.system.dummystring,
    pair labelalign = config.system.dummypair,
    bool unit = config.smooth.unit
) { return this.setelement(findlocalelementindex(destlabel), pos, label, labelalign,
unit); }
```

An alternative to `setelement` [page 15], but finds the element by label.

```
smooth rmelement (string destlabel)
```

An alternative to `rmelement` [page 15], but finds the element by label.

```
smooth movelement (string destlabel, pair shift)
```

An alternative to `movelement` [page 15], but finds the element by label.

```
smooth addsubset (
    string destlabel,
    subset sb,
    bool inferlabels = config.smooth.inferlabels,
    bool clip = config.smooth.clip,
    bool unit = config.smooth.unit
)
```

An alternative to `addsubset` [page 16], but finds the destination subset by label. The specialized versions of `addsubset` also have a label version.

```
smooth rmsubset (
    string destlabel,
    bool recursive = true
)
```

```
smooth rmsubsets (
    string[] destlabels,
    bool recursive = true
)
// (... destlabels)
```

Label-based alternatives to `rmsubset` [page 17] and `rmsubsets` [page 17].

```
smooth movesubset (
    string destlabel,
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = config.system.dummypair,
    bool movelabel = false,
    bool recursive = true,
    bool bounded = true,
    bool clip = config.smooth.clip,
    bool inferlabels = config.smooth.inferlabels,
    bool keepview = true
)
```

A label-based alternative of `movesubset` [page 18].

These are methods that facilitate the correct finding of objects by label:

```
static bool repeats (string label)
```

Check if `label` already exists as a label of some `smooth`, `subset`, or `element` object.

```
int findlocalsubsetindex (string label)
```

Locate a subset of `this` by its label and return its index.

As a final step, module `smoothmanifold` defines a number of conversion routines to find objects by label:

```
smooth findsm (string label)
```

Find a `smooth` object by its `label`.

```
subset findsb (string label)
```

Find a `subset` by its `label`.

```
element findelt (string label)
```

Find an `element` by its `label`.

These routines rely on the following auxiliary internal functions:

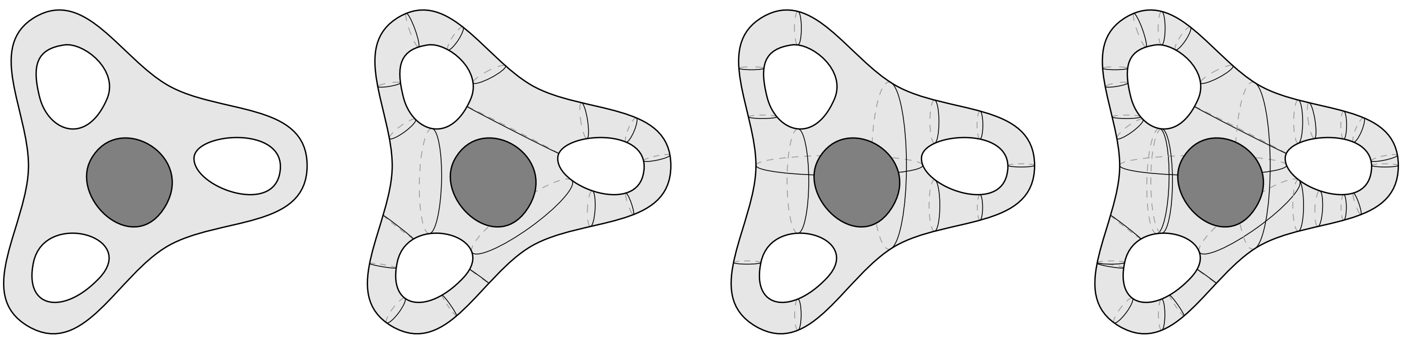
```
int findsmoothindex (string label)
int[] findsubsetindex (string label)
int[] findelementindex (string label)
```

Inspect the `smooth.cache` array for `label` and return the relevant index/indices.

Up until recently module `smoothmanifold` supported three `operator cast` routines that could cast strings to `smooth`, `subset`, and `element` structures. However, I have recently realized that this may easily lead to ambiguity in function calls, and so now the module encourages the direct use of the `findsm`, `findsb` and `findelt` routines.

4.7. The modes of cross section drawing

Cross sections (as seen in Figure 2) are meant to create the illusion of 3D in a 2D diagram. They typically connect the contours of an object's holes with the contour of the object itself. These sections can be drawn in various modes. Compare:



`plain` mode | `free` mode | `cartesian` mode | `combined` mode

Figure 15: Different modes of drawing cross sections

We will now explain how each mode is implemented.

4.7.1. The `plain` mode

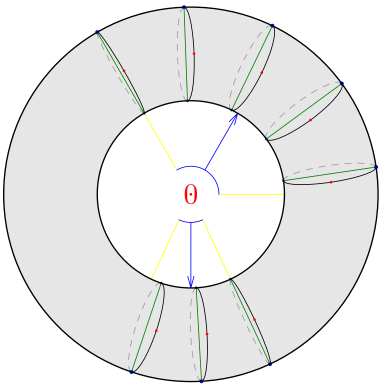
This is the simplest mode — no sections at all. It is the default mode, since most of `smoothmanifold` diagrams (despite the name of the module) are purely 2D.

4.7.2. The `free` mode

This mode makes use of the `real[][] sections` field of the `hole` [page 12] structure. Each member of this two-dimensional array is an array `{d, a, n}` of three `real` numbers, with the following meaning:

- `d` and `a` — these numbers determine the region where the cross sections are to be drawn. This is done by calling `range(g, ctr, dir(d), a)` [page 10], where `g` is the contour of either the hole or the parent object, and `ctr` is the `center` of the hole;
- `n` — the number of cross sections to draw. This is supposed to be an integer, and is converted to one with `floor(n)`.

Consider the following example:




```
smooth sm = smooth(
    contour = ucircle
).addhole(
    contour = ucircle,
    scale = .5,
    sections = new real[][]{
        {60, 120, 5}, {-90, 50, 3}
    }
);

draw(sm, dpar(help = true, mode = free, viewdir = dir(-40)));
```

Figure 16: A showcase of the `sections` field of the `hole` structure and how it is used

(for an explanation of the `dpar` structure seen in the code in Figure 16, see Section 4.8).
Now, as seen on the second picture in Figure 15, cross sections in `free` mode can connect not only the parent object's contour with the hole's contour, but also the contours of two holes. This is achieved through the `int scnumber` field of the `hole` [page 12] structure. It determines how many "inter-hole" sections a hole is willing to support with any other hole. If `hl1` has `hl1.scnumber = 4` and `hl2` has `hl2.scnumber = 2`, then there will be `2` cross sections drawn between `hl1` and `hl2`. In fact, there is a very neat trick. The expression to get the resulting number of holes is

```
abs(min(hl1.scnumber, hl2.scnumber))
```

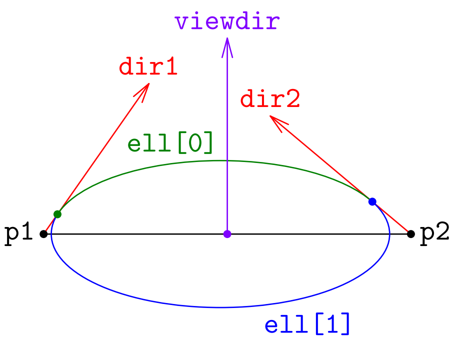
meaning that you can set, for example, `hl1.scnumber = -3`, and it will *force* the number of sections to be `3`, *regardless* of the value of `hl2.scnumber` (unless it is also negative). In other words,

- setting `hl1.scnumber = -n` guarantees that the number of sections is `n` *or more;*
- setting `hl1.scnumber = n` guarantees that the number of sections is `n` *or less;*

The `free` mode is usually the preferred mode for three-dimensional drawing. Its implementation relies heavily on the following technical routines:

```
path[] sectionellipse (pair p1, pair p2, pair dir1, pair dir2, pair viewdir)
```

Return an array of two paths, together composing an ellipse whose center lies on `p1 -- p2`, such that both vectors `dir1`, `dir2`, when starting from `p1` and `p2` respectively, are tangent to the ellipse.




```
pair p1 = (1,0), p2 = (4,0);
pair dir1 = dir(55), dir2 = dir(140);
pair viewdir = .2*dir(90);

path[] ell = sectionellipse(
  p1, p2, dir1, dir2, viewdir
);

// Drawing it in pretty colors
```

Figure 17: An illustration of the output of the `sectionellipse` function

The `viewdir` parameter represents "direction of view", it helps coordinate the tilt angles of all section ellipses in a picture to maintain the illusion of 3D.
This algorithm uses either an $O(1)$ formula, if `config.section.elprecision` (see Section 5.3) is less than zero (which is true by default), or an $O(\log n)$ binary search procedure, otherwise.

```
pair[][] sectionparams (path g, path h, int n, real r, int p)
```

Search for potential section positions between paths `g` and `h`, aiming to construct `n` sections. The meanings of `r` and `p` are:

- `r` — ranges from `0` to `1`, and can be interpreted as "freedom": when small, it restricts the section positions, but when large, the algorithm has more choice. The default value for `r` is captured by `config.section.freedom` (see Section 5.3);
- `p` — controls precision. The bigger the value of `p`, the more precise the search, but the longer it takes.

The algorithm runs in $O(n+p)$ time and produces an array of `pair` arrays, whose values can then be plugged into the `sectionellipse` [page 23] function.

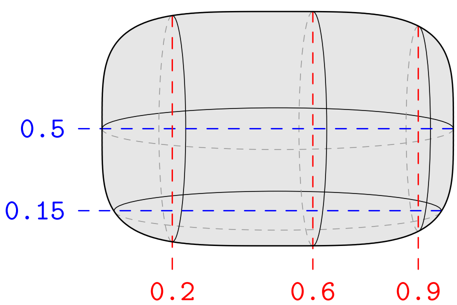
```
void drawsections (picture pic, pair[][] sections, pair viewdir, bool dash, bool help,
bool shade, real scale, pen sectionpen, pen dashpen, pen shadepen)
```

Draw the cross sections specified in `sections` by calling the `sectionellipse` [page 23] function. The meanings of the rest of the parameters is as follows:

- `viewdir` — passed to `sectionellipse` [page 23];
- `dash`, `dashpen` — after obtaining a `path[]` array from `sectionellipse`, whether to draw its second member with `dashpen`;
- `shade`, `shadepen` — whether to shade the region bounded by each ellipse with `shadepen`;
- `help` — whether to draw auxiliary help information, e.g. mark all the parameters;
- `scale` — an internal parameter that only matters when `help` is `true`;
- `sectionpen` — the pen to draw the first member of the section ellipse with.

4.7.3. The `cartesian` mode

This is the alternative mode of drawing cross sections, mainly implemented to draw three-dimensional smooth objects without any holes (note that the `free` mode relies on the presence of holes). For the `cartesian` mode, the `real[] hratios` and `real[] vratios` fields of the `smooth` [page 12] structure are used. These fields are completely similar, the only difference being that `hratios` deals with horizontal sections, where `vratios` deals with vertical. Both these arrays contain numbers ranging from `0` to `1`, and are used as follows:




```
smooth sm = smooth(
    contour = contour(<...>),
    hratios = new real[] {0.15, 0.5},
    vratios = new real[] {0.2, 0.6, 0.9}
);

draw(sm, dpar(mode = cartesian, viewdir = .7*dir(45)));
```

Figure 18: A showcase of the way the `hratios` and `vratios` fields are used

In other words, horizontal sections are drawn `r` of the way through `sm`'s *height* for every `r` in `sm.hratios`, and vertical sections are drawn `s` of the way through `sm`'s *width* for every `s` in `sm.vratios`. The mode is enabled by writing `mode = cartesian` in the `dpar` [page 25] structure.

The `cartesian` mode (so called because the sections are only vertical and horizontal) is implemented by means of the following technical routines:

```
real getyratio (real y)
real getxratio (real x)
real getypoint (real y)
real getxpoint (real x)
```

Convert to and from relative lengths.

```
smooth smooth.setratios (real[] ratios, bool horiz)
```

Set the horizontal/vertical cartesian ratios of `this` smooth object.

```
pair[][] cartsectionpoints (path[] g, real r, bool horiz)
```

Construct an array of section points in `g` (which represents a contour and holes in it) at relative length `r`, either vertically or horizontally, depending on `horiz`.

```
pair[][] cartsections (path[] g, path[] avoid, real r, bool horiz)
```

A more refined version of `cartsectionpoints` [page 25], which performs additional tests and selects suitable sections.

```
void drawcartsections (picture pic, path[] g, path[] avoid, real y, bool horiz, pair
viewdir, bool dash, bool help, bool shade, real scale, pen sectionpen, pen dashpen, pen
shadepen)
```

A wrapper drawing function for the `cartesian` mode, which calls `cartsections` [page 25] and passes the result to `drawsections` [page 24], along with other arguments.

Besides, the final section ellipses are, of course, still calculated via the `sectionellipse` [page 23] function.

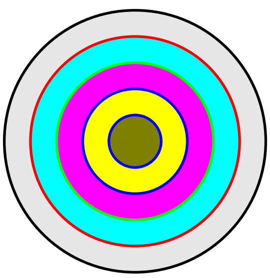
4.7.4. The `combined` mode

As seen in Figure 15, the `combined` mode combines both `free` and `cartesian` modes together, drawing all sections at once. Maybe, sometimes this is useful.

4.8. The `dpar` drawing configuration structure

You may have noticed that the `smooth` [page 12] structure contains no information about *how to draw* a smooth object (although historically it did). Now, all of this information is isolated into a separate structure called `dpar` (short for "drawing parameters"), which has the following fields:

- `pen contourpen` — the pen used to draw the contour of the smooth object, as well as those of its holes and subsets;
- `pen smoothfill` — the pen used to fill the smooth object's interior;
- `pen[] subsetcontourpens` — a list of pens to draw subset contours with (by layer). Subsets on layer `0` will be drawn with `subsetcontourpens[0]`, etc. If the array is empty, then `contourpen` will be used instead for all layers. If the number or layers is larger than the length of `subsetcontourpens`, then the last member of the array will be used for all subsequent layers that are not covered;
- `pen[] subsetfill` — similarly, a list of pens to use to fill different layers of subsets. If the array is empty, then lightened versions of the corresponding `subsetcontourpens` pens are used instead. If some layers are not covered by `subsetfill`, then the last member of the array is used, getting progressively darker. Consider the following example:



```
smooth sm = smooth(
    contour = ucircle
).addsubsets(<...>);

draw(sm, dpar(
    subsetcontourpens = new pen[] {red, green, blue},
    subsetfill = new pen[] {cyan, magenta, yellow}
));
```

Figure 19: A showcase of the interpretation of `subsetcontourpens` and `subsetfill`

- `pen sectionpen` — the pen used to draw cross sections (their visible parts);
- `pen dashpen` — the pen used to draw the “invisible” (dashed) parts of cross sections;
- `pen shadepen` — the pen used to fill the section ellipses;
- `pen elementpen` — the pen used to `dot` the elements of the smooth object;
- `pen labelpen` — the pen used to `label` the label of the smooth object;
- `pen[] elementlabelpens` — pens used to `label` the labels of the smooth object’s elements. If the array is empty, then `labelpen` is used for all elements. If there are more elements than the length of `elementlabelpens`, then the last member of the array is used for all elements not covered;
- `pen[] subsetlabelpens` — pens used to `label` the labels of the smooth object’s subsets. `labelpen` is used in case `subsetlabelpens` is empty. All subsets not covered by the array use the last member thereof;
- `int mode` — the drawing mode. Can be one of the four: `plain`, `free`, `cartesian`, `combined` (which have values of `0`, `1`, `2`, `3` respectively). Any other integer value would be accepted, but the consequences may be unpredictable;
- `pair viewdir` — the `viewdir` parameter to pass to the `sectionellipse` [page 23] function;
- `bool drawlabels` — whether to draw the label of the smooth object as well as those of its subsets and elements;
- `bool fill` — whether to fill the region bounded by the contour of the smooth object;
- `bool fillsubsets` — whether to fill the subsets of the smooth object;
- `bool drawcontour` — whether to draw the smooth object’s contour;
- `bool drawsubsetcontour` — whether to draw the contours of subsets;
- `int subsetcovermode` — the `covermode` to pass to `fitpath` [page 5] when drawing the contours of subsets;
- `bool help` — whether to enable additional information being drawn with the smooth object. Useful for debugging;
- `bool dash` — the `dash` parameter to pass to `drawsections` [page 24] or `drawcartsections` [page 25];
- `bool shade` — the `shade` parameter to pass to `drawsections` [page 24] or `drawcartsections` [page 25];
- `bool avoidsubsets` — whether to avoid drawing cross sections that intersect with the smooth object’s subsets. You can see that on the diagram on the title page of this document, this option was disabled;
- `bool overlap` — the `overlap` parameter to pass to `fitpath` [page 5];
- `bool drawnow` — the `drawnow` parameter to pass to `fitpath` [page 5];
- `bool drawextraover` — whether to apply the `drawextra` function of the smooth object “over” everything else (that is, after drawing the object itself). In other words, if `drawextraover` is `false`, then the `drawextra` will be called in the beginning of drawing the smooth object, otherwise in the end.

These are all the fields of `dpar`. The structure has a comprehensive `void operator init` constructor where all fields are given default values from the `config.drawing` [page 33] global configuration structure. The `dpar` structure also supports a method

```
dpar subs (
    pen contourpen = this.contourpen,
    pen smoothfill = this.smoothfill,
    <...>
)
```

which replaces some of the fields of `this` with new values.

Besides, there are two conveniently defined `dpar` instances:

```
dpar ghostpar (pen contourpen = currentpen)
```

A `dpar` to draw a faint outline of a smooth object.

```
dpar emptypar ()
```

A `dpar` that doesn’t draw any contours or fill any regions.

4.9. The draw function

One of the central functions of module `smoothmanifold` is the `draw` function which allows one to draw a `smooth` object:

```
void draw (
    picture pic = currentpicture,
    smooth sm,
    dpar dspec = null
)
```

Draw `sm` on picture `pic`, with drawing configuration `dspec`. This function follows all the drawing protocols described previously in Section 4.4, Section 4.7 and Section 4.8.

Moreover, there are overloaded versions of `draw`:

```
void draw (
    picture pic = currentpicture,
    smooth[] sms,
    dpar dspec = null
)
```

```
void draw (
    picture pic = currentpicture,
    dpar dspec = null
    ... smooth[] sms
)
```

4.10. Set operations on smooth objects

Nobody asked, but module `smoothmanifold` implements a few very complicated functions dedicated to calculating unions and intersections of `smooth` objects.

```
smooth[] intersection (
    smooth sm1,
    smooth sm2,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff,
    bool addsubsets = config.smooth.addsubsets
)
smooth[] intersection (
    smooth[] sms,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff,
    bool addsubsets = config.smooth.addsubsets
)
// (... sms)
```

Calculate the intersection of two or more `smooth` objects. The `round` and `roundcoeff` parameters are passed to the `intersection` [page 9] function. If `keepdata` is set to `true`, then references to old `hole` and `subset` objects will be used in the construction of the new `smooth` object. If `addsubsets` is set to `true`, the function will try to move the subsets of the given `smooth` objects to their intersection.

For convenience, `intersection` is available as an operator:

```
smooth[] operator ^^ (smooth sm1, smooth sm2)
{ return intersection(sm1, sm2); }
```

Furthermore, there is a specialized routine which only return one smooth object:

```
smooth intersect (
    smooth[] sms,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff,
    bool addsubsets = config.smooth.addsubsets
)
// (... sms)
```

Apply `intersection` [page 27] and return the 0-th element of the resulting `smooth[]` array. If the array is empty, raise an error. If the array contains more than one object, give a warning. See Section 6 for details on errors and warnings.

For `intersect` there is also an operator version:

```
smooth operator ^ (smooth sm1, smooth sm2)
{ return intersect(sm1, sm2); }
```

Likewise, there are similarly defined `union` and `unite` routines:

```
smooth[] union (
    smooth sm1,
    smooth sm2,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff
)
smooth[] union (
    smooth[] sms,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff
)
// (... sms)
```

Calculate the union of one or more `smooth` objects. The meanings of `round`, `roundcoeff` and `keepdata` are as in `intersection` [page 27].

```
smooth[] operator ++ (smooth sm1, smooth sm2)
{ return union(sm1, sm2); }
```

An operator version of `union`.

```
smooth unite (
    smooth[] sms,
    bool keepdata = true,
    bool round = false,
    real roundcoeff = config.paths.roundcoeff
)
// (... sms)
```

A specialized version of `union` which returns only one `smooth` object. An error/warning is raised if the resulting `smooth[]` array contains any number of elements other than `1`.

```
smooth operator + (smooth sm1, smooth sm2)
{ return unite(sm1, sm2); }
```

An operator version of `unite`.

4.11. Drawing arrows and paths

One of many crucial features of module `smoothmanifold` is the ease with which arrows can be drawn between smooth objects, their subsets or elements. In plain Asymptote one can easily draw an arrow between two points, but not between two *areas.* This is mainly what the `drawarrow` routine takes care of.

```
void drawarrow (
    picture pic = currentpicture,
    smooth sm1 = null,
    int index1 = config.system.dummynumber,
    pair start = config.system.dummypair,
    smooth sm2 = sm1,
    int index2 = config.system.dummynumber,
    pair finish = config.system.dummypair,
    bool elements = false,
    real curve = 0,
    real angle = 0,
    real radius = config.system.dummynumber,
    bool reverse = false,
    pair[] points = {},
    Label L = "",
    pen p = currentpen,
    tarrow arrow = config.arrow.currentarrow,
    tbar bar = config.arrow.currentbar,
    bool help = config.help.enable,
    bool overlap = config.drawing.overlap,
    bool drawnow = config.drawing.drawnow,
    real beginmargin = config.arrow.mar,
    real endmargin = config.system.dummynumber
)
```

Draw an arrow. The routine is quite sophisticated in that it accepts many different types of arguments. You can start the arrow from:

- The `sm1` object;
- A subset of the `sm1` object, under index `index1`;
- An arbitrary point `start`.

Likewise, you can finish the arrow at:

- The same object `sm1` (by default), getting a cyclic arrow;
- The same `sm1`, but different subset, under index `index2`;
- A different smooth object, `sm2`, or its subset under `index2`;
- An arbitrary point `finish`.

The meaning of the remaining arguments is as follows:

- `elements` — if set to `true`, then `index1` and `index2` will be treated as element indices instead of subset indices;
- `curve` — for non-cyclic arrows, this argument is passed to the `curvedpath` [page 10] function;
- `angle` and `radius` — for cyclic arrows, these arguments are passed to the `cyclepath` [page 10] function;
- `reverse` — whether the arrow should be reversed;
- `points` — an array of arbitrary points that the arrow should pass through. If this array is non-empty, the `connect` [page 11] function is used instead of `curvedpath` [page 10] or `cyclepath` [page 10];
- `L` — the label to attach to the arrow;
- `p` — the pen to draw the arrow with;
- `arrow` and `bar` — the arrow and bar to use with the arrow. Since `drawarrow` is integrated with `smoothmanifold`'s deferred drawing system, the custom `tarrow` [page 4] and `tbar` [page 4] structures are used here;
- `help` — if set to `true`, addition help/debug information will be drawn;
- `overlap` and `drawnow` — these arguments are passed to `fitpath` [page 5] together with `arrow` and `bar`;

- `beginmargin` and `endmargin` — the margins left at the beginning and end. If `endmargin` is not specified, `beginmargin` is used instead.

For the definitions of the default values of `drawarrow`'s parameters, see Section 5.

There is a specialized overloaded version of `drawarrow`:

```
void drawarrow (
    picture pic = currentpicture,
    string destlabel1,
    string destlabel2 = destlabel1,
    real curve = 0,
    <...>
)
```

A label-based version of `drawarrow`. It automatically detects if the labels belong to `smooth` objects/subsets/elements, and passes appropriate arguments to `drawarrow` [page 29].

Now, apart from arrows, one may want to draw a path connecting two points. This may be done with the `drawpath` routine:

```
void drawpath (
    picture pic = currentpicture,
    smooth sm1,
    int index1,
    smooth sm2 = sm1,
    int index2 = index1,
    real range = config.paths.range,
    real angle = config.system.dummynumber,
    real radius = config.system.dummynumber,
    bool reverse = false,
    pair[] points = {},
    Label L = "",
    pen p = currentpen,
    bool help = config.help.enable,
    bool random = config.drawing.pathrandom,
    bool overlap = config.drawing.overlap,
    bool drawnow = config.drawing.drawnow
)
```

Draw a surface path connecting `sm1.elements[index1]` and `sm2.elements[index2]`. By default, `sm1` and `sm2` coincide. All remaining arguments have the same meaning as in `drawarrow` [page 29], except for two:

- `range` — this value is passed to `randomdir` [page 11] as `angle`;
- `random` — if set to `true`, then the `randompath` [page 11] routine will be used for the path, instead of `connect` [page 11] or `cyclepath` [page 10].

Similarly to `drawarrow`, there is a label-based version of `drawpath`:

```
void drawpath (
    picture pic = currentpicture,
    string destlabel1,
    string destlabel2 = destlabel1,
    real range = config.paths.range,
    <...>
)
```

Draw a surface path between elements with labels `destlabel1` and `destlabel2`.

5. Global configuration and the `config` structure

Throughout the document, we have seen references to a global `config` structure instance. It is defined as follows:

```
struct globalconfig {
    systemconfig system;
    pathconfig paths;
    sectionconfig section;
    smoothconfig smooth;
    drawingconfig drawing;
    helpconfig help;
    arrowconfig arrow;
}

private globalconfig defaultconfig;
globalconfig config;
```

The `defaultconfig` instance contains all default values, while `config` contains all *current* values, which can be changed directly at any time in the Asymptote code. Functions and methods feature default argument values that borrow from `config`, never `defaultconfig`. This way, a convenient configuration system is created, where all functions are aware of the flexible current `config`, and the defaults can be easily restored by deeply copying `defaultconfig` to `config`. We will now define in detail the member structures of `config`.

5.1. System variables

```
struct systemconfig {
    string version = "v6.3.1";
    int dummynumber = -10000;
    string dummystring = (string) dummynumber;
    pair dummypair = (dummynumber, dummynumber);
    bool repeatlabels = false;
    bool insertdollars = false;
}
```

A structure containing system configuration variables. Their meanings are as follows:

- `version` — the version of `smoothmanifold`, currently `6.3.1`;
- `dummynumber` — the number that "the program knows what to do with". It is often given as a default value in many functions, when the true default value requires additional computations that are better done in the body of the function;
- `dummypair` and `dummystring` — versions of `dummynumber` with different types. Module `smoothmanifold` features four helpher functions to check if a value is "dummy":

```
bool dummy (int n)    bool dummy (real r)    bool dummy (pair p)    bool dummy(string s)
```

- `repeatlabel` — whether to allow different objects to share the same label. If set to `false`, an error (see Section 6) will be raised on attempt to bestow an already existent label;
- `insertdollars` — whether to automatically insert `$` characters around all labels (including those passed to `drawarrow` [page 29] and `drawpath` [page 30]) right before drawing them. This is useful, but disabled by default to avoid confusion.

5.2. Path variables

```
struct pathconfig {
    real roundcoeff = .03;
    real range = 30;
    real neighheight = .05;
    real neighwidth = .01;
}
```

A structure containing global path-related variables. The meaning of the fields is as follows:

- `roundcoeff` — default value to pass to `combination` [page 7] and its derivatives;
- `range` — default value to pass to `drawpath` [page 30];
- `neighheight` — default value to pass to `neigharc` [page 12] as `h`;

- `neighwidth` — default value to pass to `neigharc` [page 12] as `w`;

5.3. Cross section variables

```
struct sectionconfig {
    real maxbreadth = .65;
    real freedom = .3;
    int precision = 20;
    real elprecision = -1;
    bool avoidsubsets = false;
    real[] default = new real[] {-10000,235,5};
}
```

A structure to hold cross section-related configuration. The meanings of the fields are as follows:

- `maxbreadth` — bound on how wide `dir1` and `dir2` can be spread in `sectionellipse` [page 23] (see Figure 17). There are two technical routines dedicated to controlling the “quality” of cross sections:

```
real sectionsymmetryrating (pair p1p2, pair dir1, pair dir2)
bool sectiontoobroad (pair p1, pair p2, pair dir1, pair dir2)
```

- `freedom` — the default value to pass to the `sectionparams` [page 24] function as `r`;
- `precision` — the default value to pass to `sectionparams` [page 24] as `p`;
- `elprecision` — the default precision for the `ellipsepath` [page 10] binary search algorithm. Set to `-1` by default, to use the $O(1)$ formula;
- `avoidsubsets` — the default value for the `avoidsubsets` field of the `dpar` [page 25] structure;
- `default` — the default section array used to substitute missing values in `addsection` [page 16] and its related methods.

5.4. Smooth object variables

```
struct smoothconfig {
    int interholenumber = 1;
    real interholeangle = 25;
    real maxsectionlength = -1;
    real rejectcurve = .15;
    real edgemargin = .07;
    real stepdistance = .15;
    real nodesize = 1;
    real maxlength;
    bool inferlabels = true;
    bool addsubsets = true;
    bool correct = true;
    bool clip = false;
    bool unit = false;
    bool setcenter = true;
}
```

A structure to hold all smooth object-related variables, whose meanings are as follows:

- `interholenumber` — the default value for the `scnumber` field in the `hole` [page 12] structure;
- `interholeangle` — this value gets passed to the `range` [page 10] function as `ang`, when drawing sections between holes;
- `maxsectionlength` — the maximum length of a section. A value of `-1` means no restriction. Setting this value may help get rid of some annoying sections that look bad because they are too big;
- `rejectcurve` — this value is passed to `curvedpath` [page 10] when constructing curves to determine which inter-hole sections to draw and which not to. This is done by checking if the curves intersect with any holes or subsets;
- `edgemargin` — no point in explaining, let this one remain a mystery. If you REALLY need to know what this does, go see the source code...
- `stepdistance` — same;
- `nodesize` — the default value of the `size` parameter in the `node` TODO function;

- `maxlength` — a parameter dependent on `maxsectionlength`, should not be set manually;
- `inferlabels` — the default value of the `inferlabels` parameter passed to `addsubset` [page 16] and its derivatives;
- `addsubsets` — the default value to pass to the `intersection` [page 27] function and its derivatives;
- `correct` — whether non-clockwise paths should be reversed to clockwise. This value is passed as default in set operations with paths (see Section 3.1);
- `clip` — the default value passed to the `addsubset` [page 16], `addhole` [page 15], `movesubset` [page 18] and their derivatives, as the `clip` parameter;
- `unit` — whether to use unit coordinates (see Section 4.5). This is the default value of the `unit` parameter in all methods that accept it;
- `setcenter` — whether to automatically set the centers of various objects in various situations by calling `center` [page 9] on their `contour`.

5.5. Drawing-related variables

```
struct drawingconfig {
    pair viewdir = (0,0);
    real viewscale = 0.12;
    real gaplength = .05;
    pen smoothfill = lightgrey;
    pen[] subsetfill = {};
    real sectpenscale = .6;
    real elpenwidth = 3.0;
    real shadescale = .85;
    real dashpenscale = .4;
    real dashopacity = .4;
    real attachedopacity = .8;
    real subpenfactor = .5;
    real subpenbrighten = .5;
    pen sectionpen = nullpen;
    real lineshadeangle = 45;
    real lineshadedensity = 0.15;
    real lineshademargin = 0.1;
    pen lineshadepen = lightgrey;
    int mode = 0;
    bool useopacity = false;
    bool dash = true;
    bool underdashes = false;
    bool shade = false;
    bool drawlabels = true;
    bool fill = true;
    bool fillsubsets = true;
    bool drawcontour = true;
    bool drawsubsetcontour = true;
    int subsetcovermode = 0;
    bool pathrandom = false;
    bool overlap = false;
    bool drawnow = false;
    bool drawextraover = false;
    bool subsetoverlap = false;
    real elementcirclerad = -1;
}
```

A structure to hold drawing-related configuration, encoded in the following variables:

- `viewdir`, `smoothfill`, `subsetfill`, `sectionpen`, `mode`, `dash`, `shade`, `drawlabels`, `fill`, `fillsubsets`, `drawcontour`, `drawsubsetcontour`, `subsetcovermode`, `overlap`, `drawnow`, `drawextraover` — default values of the corresponding `dpar` [page 25] fields;
- `viewscale` — a value by which the `viewdir` vector is scale on call of the `draw` [page 27] function;

- gaplength — the default length of the gaps left in deferredPath [page 4] objects when the fitpath [page 5] function is called;
- sectpenscale — determines how thinner the section pen is compared to contourpen;
- elpenwidth — the width of the dot's representing a smooth object's elements, used when calling draw [page 27]. The relevant pen creation utility is

```
pen elementpen (pen p) { return p + linewidth(config.drawing.elpenwidth); }
```

- shadescale — how darker shaded section ellipses are compared to object filling color. This is relevant if the shade flag is set to true;
- dashpenscale — how lighter dashed lines (in cross section drawing) are compared to regular lines;
- dashopacity — default opacity of dashed pens;
- attachedopacity — the opacity used to draw smooth objects attached [page 12] to the one currently being drawn;
- subpenfactor — how darker the fill color for subsets get with each layer. See Section 4.8 and Figure 19;
- subpenbrighen — how brighter to make the fill color of a subset than its contour color. The relevant pen creation routines are

```
pen brighten (pen p, real coeff) { return inverse(coeff * inverse(p)); }
pen nextsubsetpen (pen p, real scale) { return scale * p; }
```

- lineshadeangle, lineshadedensity, lineshademargin, lineshadepen — default values passed to the shaderegion TODO function;
- useopacity — whether to use Asymptote's opacity function for generating pens (primarily dashed section pens). Since some image formats do not support opacity, this is disabled by default, and so the dashpenscale field is used for dashed pen creation. Otherwise, the dashopacity field is used. The relevant pen creation routines are

```
pen sectionpen (pen p)
pen dashpenscale (pen p)
    { return inverse(config.drawing.dashpenscale*inverse(p))+dashed; }
pen dashopacity (pen p) { return p+dashed+opacity(config.drawing.dashopacity); }
pen dashpen (pen p)
pen shadepen (pen p) { return config.drawing.shadescale*p; }
pen underpen (pen p) { return dashpen(p); }
```

- underdashes — whether to draw the "demoted" parts of a deferredPath [page 4] in a dashed line, instead of erasing them;
- pathrandom — the default value passed to the drawpath [page 30] function as random;
- elementcirclerad — the radius to use when drawing elements as circle's instead of dot's. This exists because sometimes dots are not showing up in SVG output, in which case the simplest thing to do is say screw it and fill a circle instead of calling the dot function.

5.6. Help-related variables

```
struct helpconfig {
    bool enable = false;
    real arcratio = 0.2;
    real arrowlength = .2;
    pen linewidth = linewidth(.3);
}
```

A structure to hold the few help-related variables, namely
- enable — whether to enable auxiliary drawing by default;
- arcratio — the relative radius of the blue arcs seen near the center of the hole on Figure 16, it may be useful to tweak if the arc covers the number in the center of the hole;
- arrowlength — the length of auxiliary arrows drawn by the sides of cross sections when help = true. A value of -1 will disable these arrows (like on Figure 16);
- linewidth – the pen containing the line width to draw help lines with.

5.7. Arrow variables

```
struct arrowconfig {
    real mar = 0.03;
    tarrow currentarrow = null;
    tbar currentbar = null;
    bool absmargins = true;
}
```

A structure to hold variables related to arrows, namely the following:

- `mar` — the default arrow margin that is passed as the default value to the `beginmargin` and `endmargin` parameters of the `drawarrow` [page 29] function;
- `currentarrow`, `currentbar` — the default values of the `arrow` and `bar` parameters of `drawarrow` [page 29]; The `currentarrow` value is updated after the definition of `DeferredArrow` [page 5]:

  ```
  config.arrow.currentarrow = DeferredArrow(SimpleHead);
  ```

- `absmargins` — whether arrow margins should be absolute, as opposed to being relative to the length of the arrow.

6. Debugging capabilities

In case an algorithm runs into an incorrect situation or is given incorrect input, mechanisms are put in place in module `smoothmanifold` to trigger an *error* (subsequently stopping execution) or a *warning* (while continuing execution). For errors, the relevant routine is

```
void halt (string msg) {
    write();
    write("> ! " + msg);
    abort("");
}
```

Warnings are simply written with a direct call to `write`.

7. Miscellaneous auxiliary routines

7.1. `smooth`-related functions

```
smooth[] concat (smooth[][] smss)
```

Concatenate an array of `smooth` objects. In Asymptote, it is difficult to write a polymorphic `concat` function.

```
void print (smooth sm)
```

Print various information about the smooth object `sm` in the console.

```
void printall ()
```

Print all `smooth` objects in the global `smooth.cache` array.

```
void smooth.checksubsetindex (int index, string fname)
```

Check if `this` has a subset at index `index`. The `fname` parameter is used for debugging purposes – it contains the name of the function that is calling `checksubsetindex`.

```
void smooth.checkelementindex (int index, string fname)
```

Check if `this` has a element at index `index`. The `fname` parameter is used for debugging purposes – it contains the name of the function that is calling `checkelementindex`.

7.2. Array utilities

```
path[] c (... path[] source) { return source; }
```

An R-style convenience function, for those who are tired of writing `new path[]{}`.

```
path[][] cc (... path[] source) { return new path[][]{source}; }
```

A convenient replacement of `new path[][]{{}}`.

```
pair[] concat (pair[][] a)
```

Concatenate an array of `pair`.

```
path[] concat (path[][] a)
```

Concatenate an array of `path`.

```
bool contains (int[] source, int a)
```

Check if `source` contains `a`.

7.3. Other routines

```
pair comb (pair a, pair b, real t) { return t*b + (1-t)*a;}
```

A convex combination between `a` and `b` with time `t`.

8. Index

A

B

C