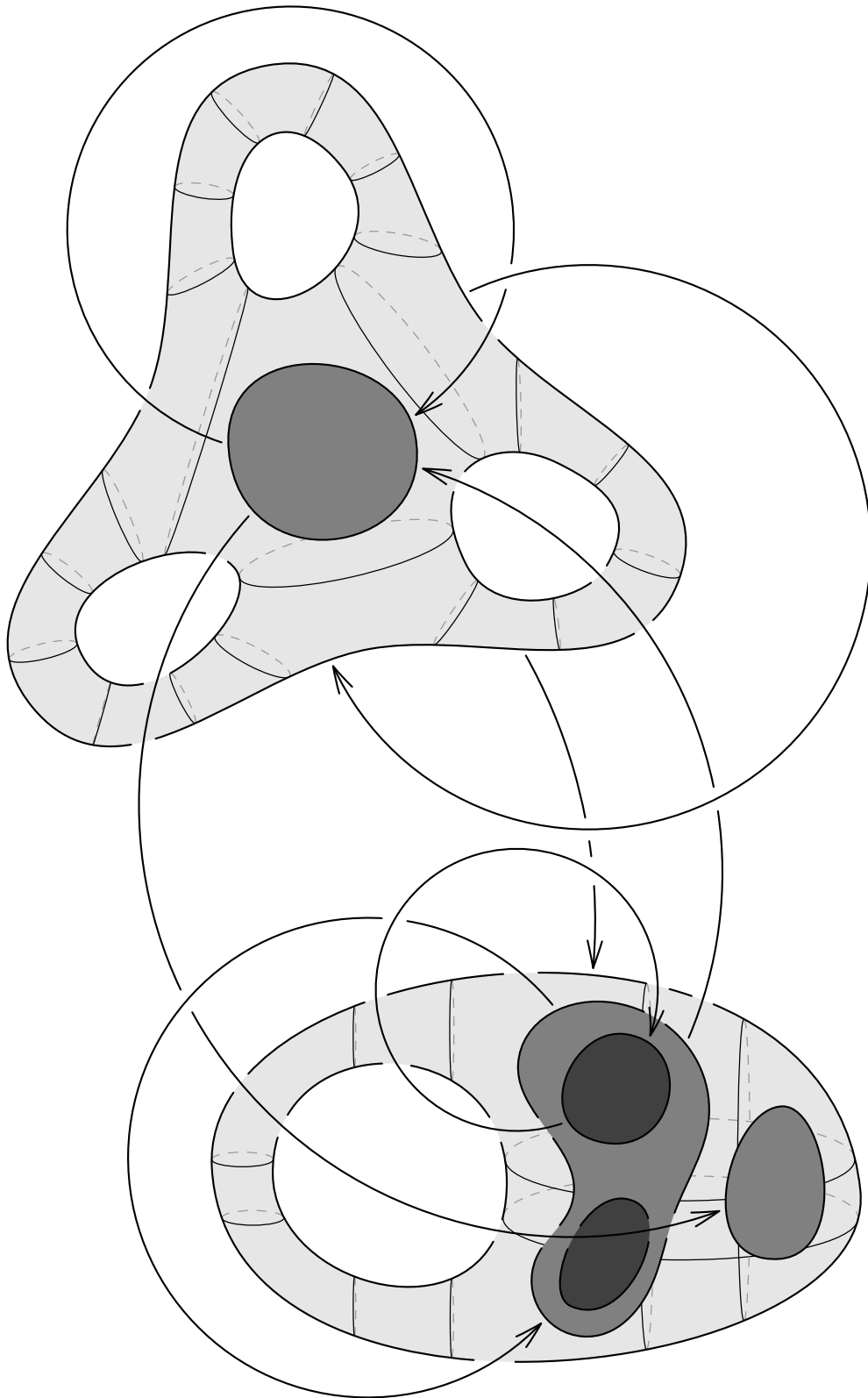


Diagrams in higher mathematics with Asymptote

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

Abstract	2
1. Introduction	3
2. Deferred drawing and path overlapping	4
2.1. The general mechanism	4
2.2. The <code>tarrow</code> and <code>tbar</code> structures	4
2.3. The <code>fitpath</code> function	5
2.4. Other related routines	6
3. Operations on paths	7
3.1. Set operations on bounded regions	7
3.2. Other path utilities	9
4. Smooth objects	12
4.1. Definition of the <code>smooth</code> , <code>hole</code> , <code>subset</code> , and <code>element</code> structures	12
4.2. Construction	14
4.3. Query and mutation methods	14
4.3.1. <code>smooth</code> objects	14
4.3.2. <code>subset</code> objects	18
4.3.3. <code>hole</code> objects	19
4.3.4. <code>element</code> objects	19
4.4. The subset hierarchy	19
4.5. Unit coordinates in a <code>smooth</code> object	20
4.6. Reference by label	20
4.7. The modes of cross section drawing	22
4.7.1. The <code>plain</code> mode	22
4.7.2. The <code>free</code> mode	22
4.7.3. The <code>cartesian</code> mode	23
4.7.4. The <code>combined</code> mode	23
4.8. The <code>dpar</code> drawing configuration structure	23
4.9. The <code>draw</code> function	23
5. Global configuration	23
5.1. System variables	23
5.2. Path variables	23
5.3. Cross section variables	23
5.4. Smooth object variables	23
5.5. Drawing-related variables	23
5.6. Help-related variables	23
5.7. Arrow variables	23
6. Debugging capabilities	23
6.1. Errors	23
6.2. Warnings	23
7. Miscellaneous auxiliary routines	23
8. The <code>export.asy</code> auxiliary module	24
8.1. The <code>export</code> routine	24
8.2. Animations	24
8.3. Configuration	24
9. Index	24

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`:

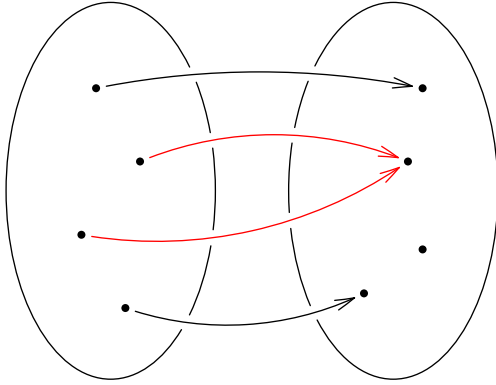


Figure 1: An illustration of non-injectivity (set theory)

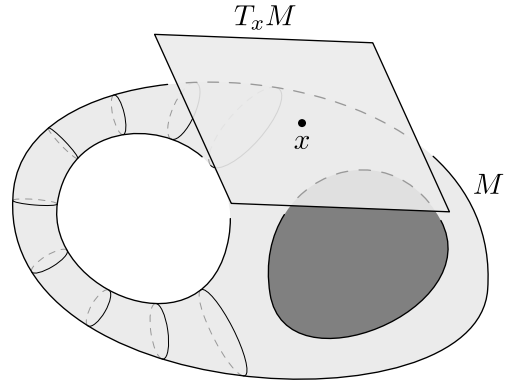


Figure 2: Tangent space at a point on a manifold (diff. geometry)

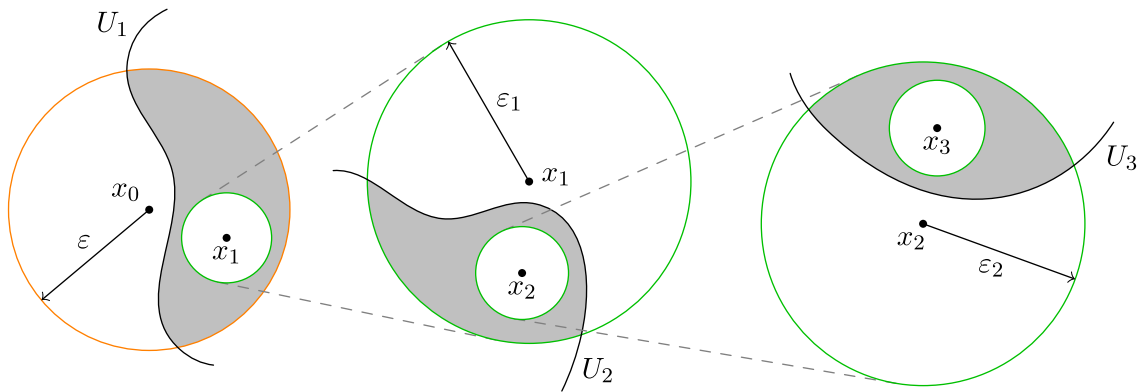


Figure 3: The proof of the Baire category theorem (topology)

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 it 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)

```

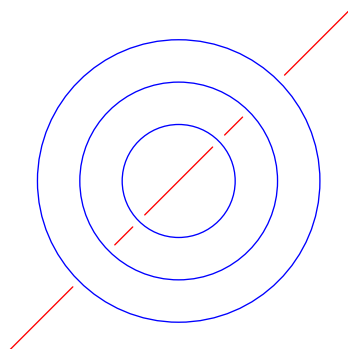
Arguments:

- `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

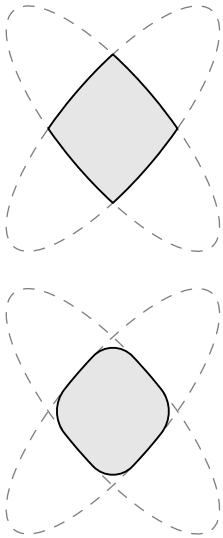
3.1. Set operations on bounded regions

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:

<pre>path[] difference (path p, path q, bool correct = true, bool round = false, real roundcoeff = config.paths.roundcoeff)</pre>	<pre>path[] operator - (path p, path q) { return difference(p, q); }</pre>
---	--

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:

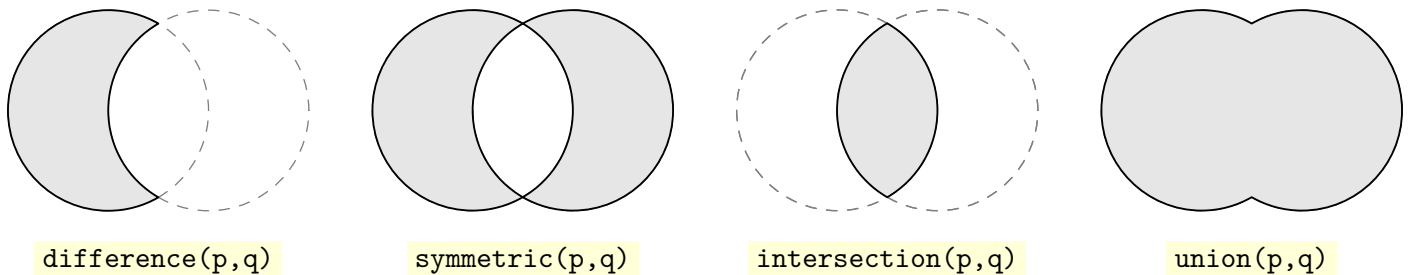


Figure 6: Different specializations of the combination function

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))*0.25; }
```

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:

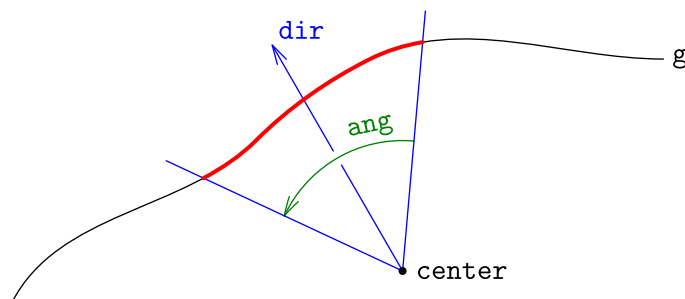


Figure 7: An illustration of the `range` function

```
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)
```

Construct 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.

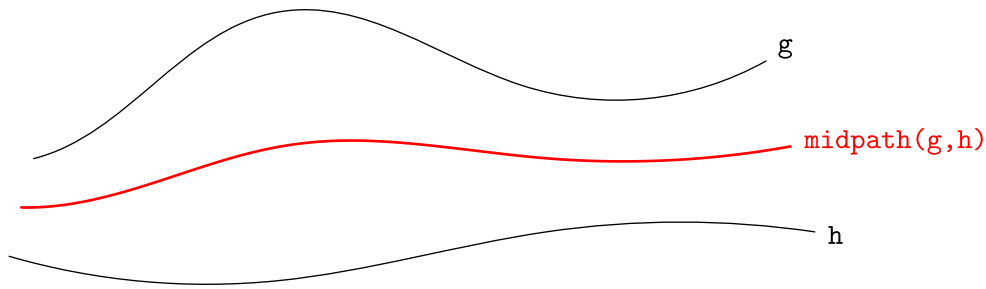


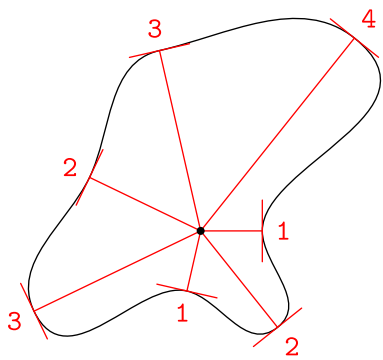
Figure 8: An illustration of the midpath function

```
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:

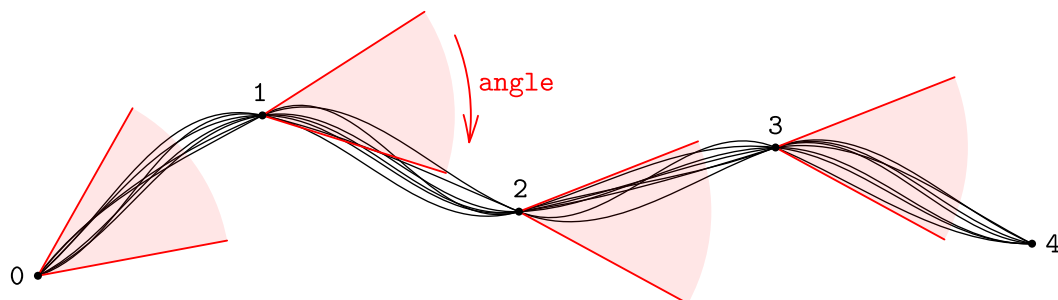


Figure 10: A showcase of the randompath function

```

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:

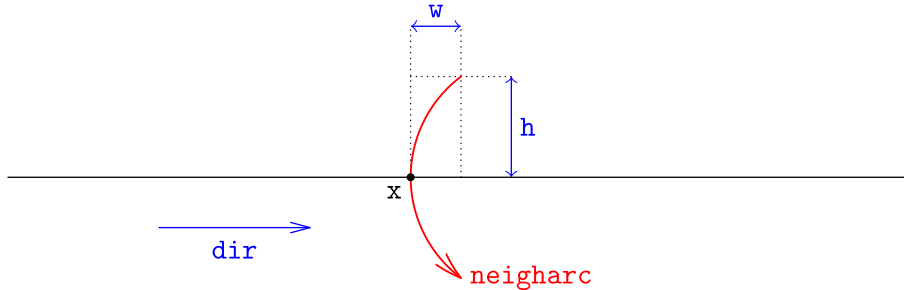


Figure 11: A showcase of the `neigharc` function

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:

<pre> 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 postdraw (dpar, smooth); static smooth[] cache; } </pre>	<pre> 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; } </pre>
--	--

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.

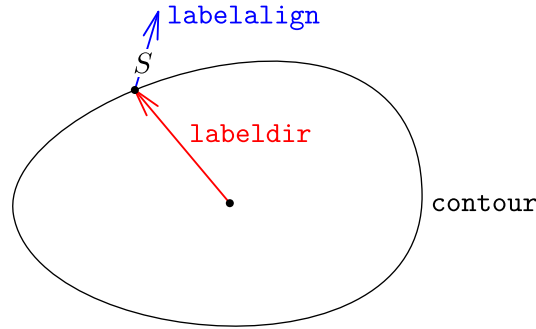


Figure 12: A showcase of the `labeldir` and `labelalign` fields

- `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;
- `postdraw` — 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.

<pre>smooth move (pair shift = (0,0), real scale = 1, real rotate = 0, pair point = this.center, bool readjust = true, bool drag = true)</pre>	<pre>smooth shift (explicit pair shift) smooth shift (real xshift, real yshift = 0) smooth scale (real scale) smooth rotate (real rotate)</pre>
--	--

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 dirscales (pair dir, real s)
```

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

```
smooth setcenter (
    int index = -1,
    pair center = config.system.dummyspair,
    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.dummyspair`, see Section 5.1.

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

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.dummyspair,
    string label = config.system.dummystring,
    pair labelalign = config.system.dummyspair,
    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.dummyspair,
    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:

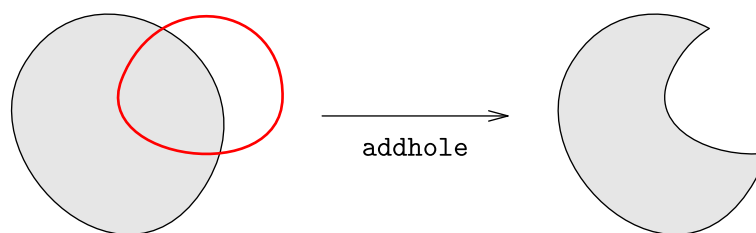


Figure 13: A showcase of the `clip` parameter

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`.

```
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' 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' 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.dummyspair,
    pair shift = (0,0),
    real scale = 1,
    real rotate = 0,
    pair point = center(contour),
    string label = "",
    pair dir = config.system.dummyspair,
    pair align = config.system.dummyspair,
    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.dummyspair,
    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:

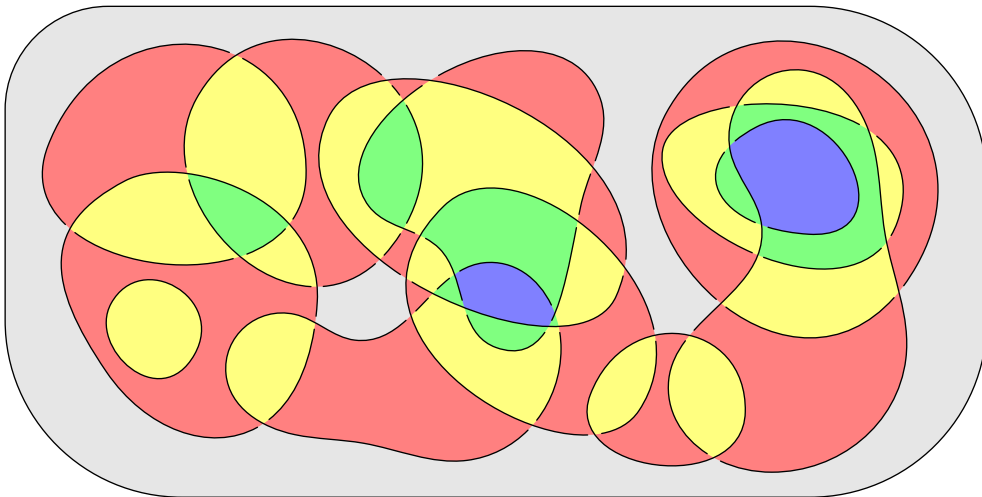


Figure 14: An example of many intersecting subsets

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.dummyspair,
    pair align = config.system.dummyspair
) { 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.dummyspair,
    string label = config.system.dummysstring,
    pair labelalign = config.system.dummyspair,
    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.

<pre>smooth rmsubset (string destlabel, bool recursive = true)</pre>	<pre>smooth rmsubsets (string[] destlabels, bool recursive = true) // (... destlabels)</pre>
--	--

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.dummyspair,
    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 operator `cast` functions for full label integration:

```
smooth operator cast (string label)
smooth[] operator cast (string[] labels)
```

Cast a string to a smooth object.

```
subset operator cast (string label)
subset[] operator cast (string[] labels)
```

Cast a string to a subset object.

```
element operator cast (string label)
element[] operator cast (string[] labels)
```

Cast a string to an element object.

These casts rely on the following auxiliary internal functions:

<pre>int findsmoothindex (string label) int[] findsubsetindex (string label) int[] findelementindex (string label)</pre>	<pre>smooth findsm (string label) subset findsb (string label) element findelt (string label)</pre>
--	---

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:

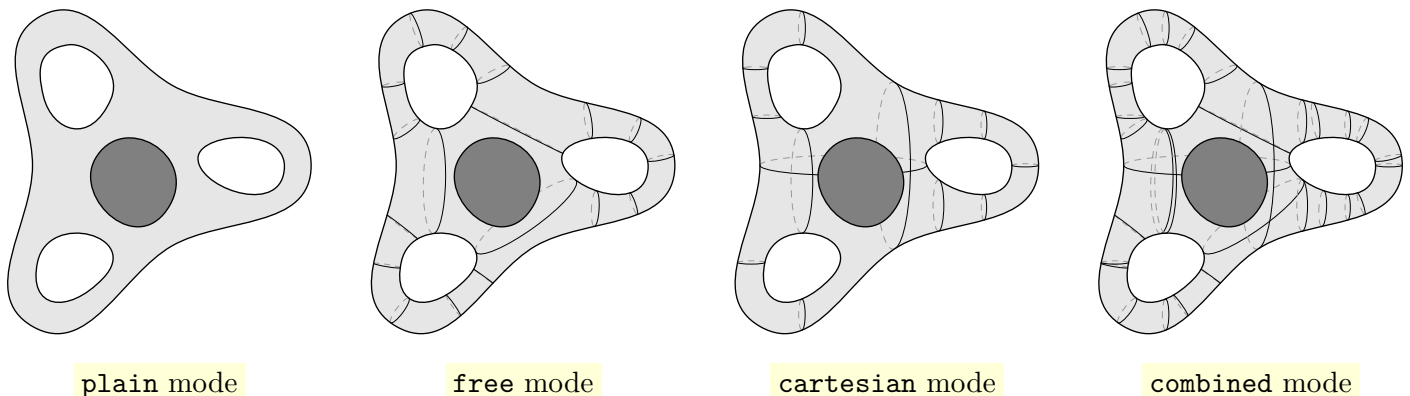


Figure 15: Different modes of drawing cross sections

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 `sections` field of the hole [\[page 12\]](#) structure.

4.7.3. The cartesian mode

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

Convert to and from relative lengths.

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

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

4.7.4. The combined mode

4.8. The dpar drawing configuration structure

4.9. The draw function

5. Global configuration

5.1. System variables

5.2. Path variables

5.3. Cross section variables

5.4. Smooth object variables

5.5. Drawing-related variables

5.6. Help-related variables

5.7. Arrow variables

6. Debugging capabilities

6.1. Errors

Should you perform an erroneous calculation step (like adding an out-of-bounds subset to a smooth object, or referring to a non-existent label), `smoothmanifold` will crash with an error message.

6.2. Warnings

7. Miscellaneous auxiliary routines

```
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.

8. The `export.asy` auxiliary module

8.1. The export routine

8.2. Animations

8.3. Configuration

9. Index