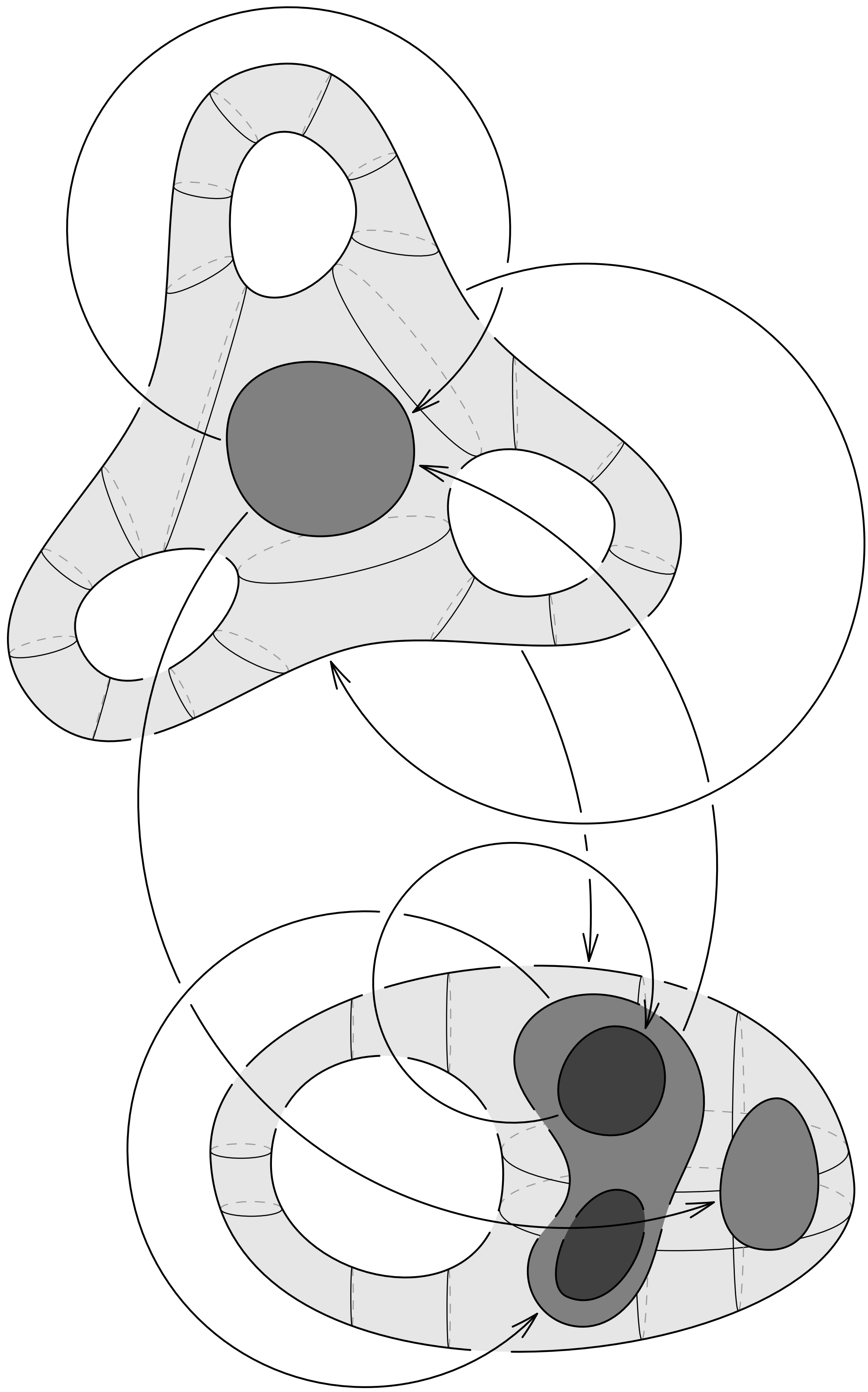
smoothmanifold.asy - v6.3.0

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, 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`:

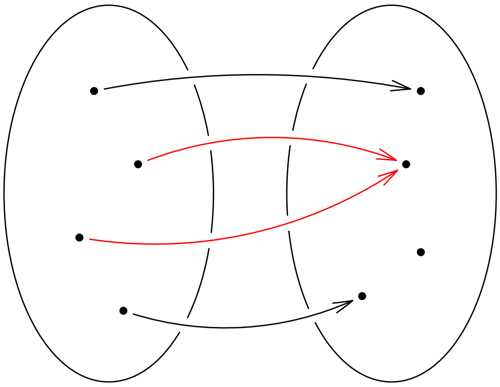


Figure 1: An illustration of non-injectivity (set theory)

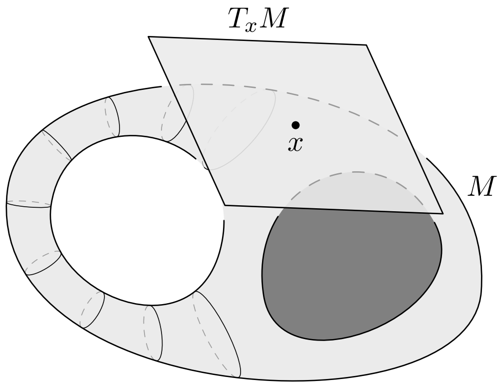


Figure 2: Tangent space at a point on a manifold (diff. geometry)

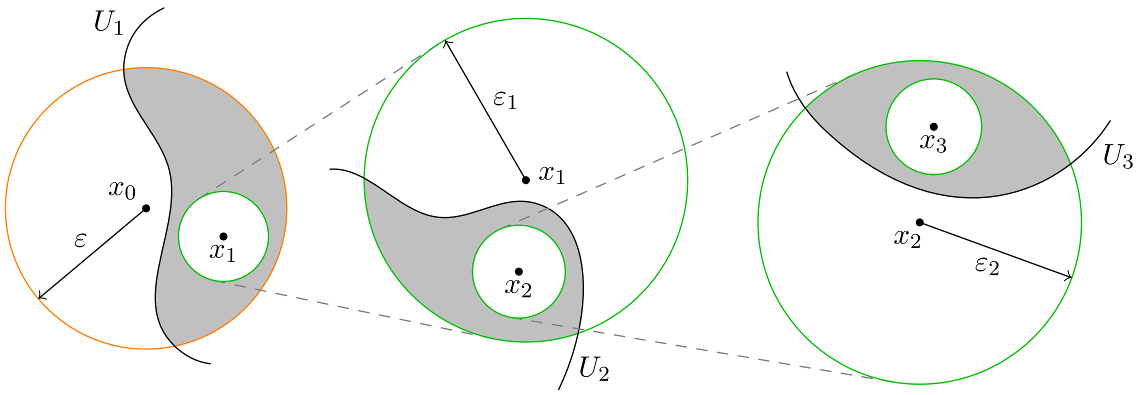


Figure 3: The proof of the Baire category theorem (topology)

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` struct, the paths drawn to 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. Construct a `deferredPath` based on `p`, say `dp`;
2. Exploit the `nodes` field of the `picture` struct to store an integer. Retrieve this integer, say `n`, from `pic`.
3. Store the deferred path `dp` in a global two-dimensional array, under index `n`;
4. Modify the deferred path `dp` as needed, e.g. add gaps;
5. At shipout time, when processing the picture `pic`, retrieve the index `n` from its `nodes` field and draw all `deferredPath` objects in the two-dimensional array at index `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

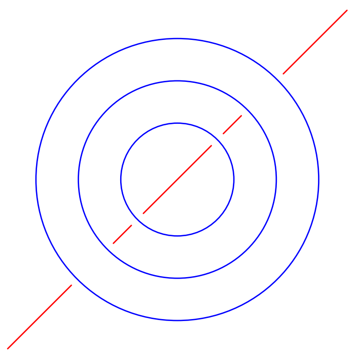
This is a substitute for the plain `draw` function. The `fitpath` function implements steps 1-4 of the deferred drawing system describes above.

```
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 TODO. 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 TODO.

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

3. Operations on paths

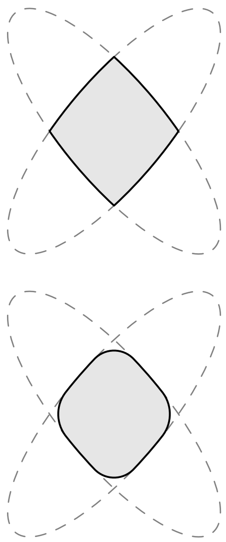
3.1. Combination of bounded regions

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:

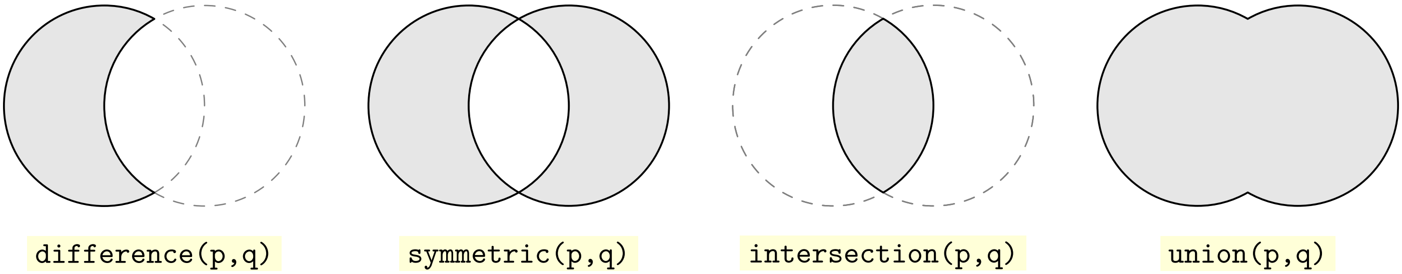


Figure 6: Different specializations of the `combination` function

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:

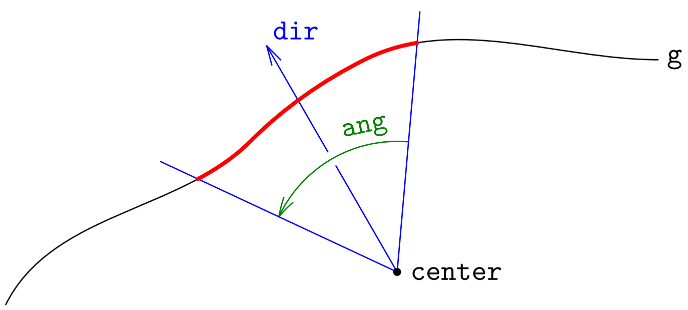


Figure 7: An illustration of the `range` function

```
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.

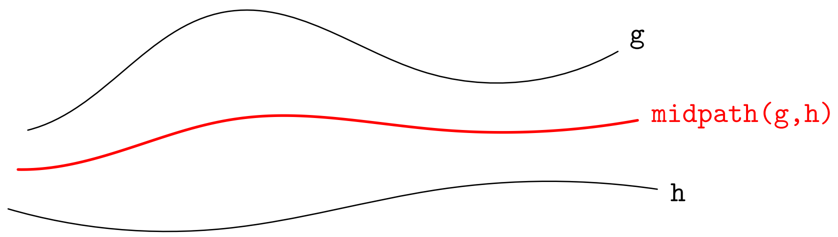


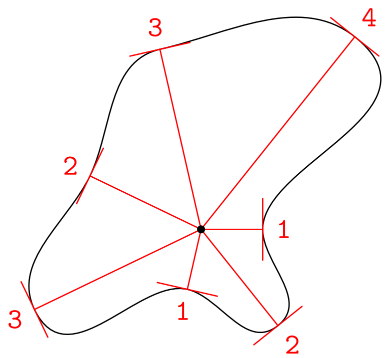
Figure 8: An illustration of the `midpath` function

```
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` (see [page 9]) is `(0,0)`, and its `radius` (see [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:

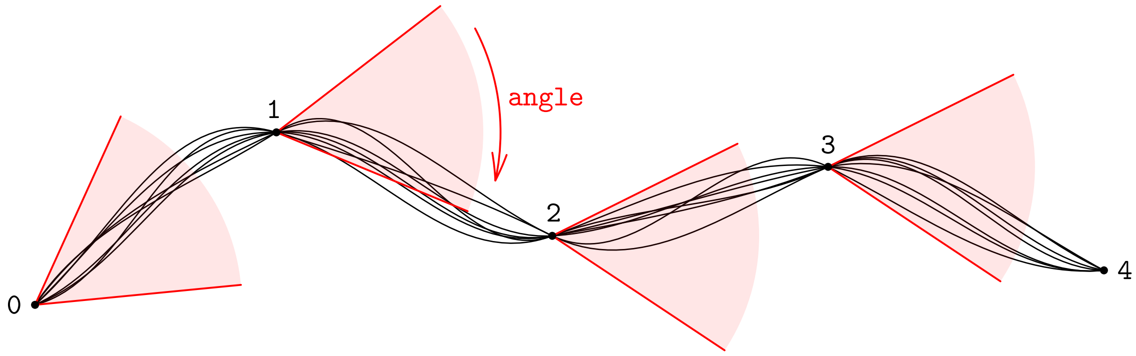


Figure 10: A showcase of the `randompath` function