

---

# WEIR-A System for Making Generative Systems

## About

This library is specifically written to be useful for a broad range of ways in which I create art using various generative algorithms. It is the next iteration of snek. I made a new version because I wanted to make some significant changes.

Main components:

1. 2d and 3d vectors with corresponding operations:

```
1 ; create a 2d vector
2 (vec:vec 1d0 3d0)
3
4 ; create a 3d vector
5 (vec:3vec 1d0 2d0 3d0)
```

`vec` supports common vector operations such as `add`, `mult`, `div`, `sub`, as well as cross products `cross`, dot products, `dot`.

*Note: Functions that operate on 3d vectors are prefixed with 3.*

Furthermore there are corresponding functions for scalars, lists of vectors, and broadcasting. They are indicated by prefix `l` and `s`:

```
1 ; add scalar s to a, return result
2 (vec:sadd va s)
3
4 ; add vectors in two lists, returns list of new vectors
5 (vec:ladd aa bb)
6
7 ; add b to elements of aa, return list of new vectors
8 (vec:ladd* aa vb)
```

Most of these functions also have a corresponding function postfix with `!`, which performs the same operation, but assigns the result to the first argument. This is faster since there is no need to create a new `vec` struct.

*Note: This can cause strange behaviour since you can inadvertently change the internal state of a struct. Use with caution.*

Example:

```
1 ; add b to a, store result in a; also returns a
2 (vec:add! va vb)
```

---

There are also some common geometric operations, such as line interpolation, line-line intersection, angles, and line-plane intersection.

```
1 ; find the position s between a and b. s should be between 0 and 1
2 (vec:on-line s va vb)
3 ; find intersection of two lines, returns
4 ; a bool, and interpolation value along line 1, line 2
5 (vec:segx line1 line2)
```

See the code in the package for more details.

## 2. Random numbers, some examples:

```
1 ; random double 0d0 and a (default: 1d0)
2 (rnd:rnd a)
3
4 ; ditto, between -a and a
5 (rnd:rnd* a)
6
7 ; between a and b
8 (rnd:rndrng a b)
9
10 ; random fixnum
11 (rnd:rndi 10)
12
13 ; return n random numbers between a and b
14 (rnd:rndspace n a b)
15
16 ; random number from normal distribution
17 (rnd:norm :mu 0d0 :sigma 1d0)
18
19 ; uniform in circle of radius a at centered at p
20 (rnd:in-circ a :xy p)
21
22 ; uniform numbers in a rectangle
23 (rnd:in-rect w h :xy p)
24
25 ; pick a random point between points va and vb
26 (rnd:on-line va vb)
27
28 ; execute a form with a certain probability
29 ; second form is optional
30 (rnd:prob 0.1d0 (print "10% hi") (print "90% oh no"))
31
32 ; perform either form 1 or 2
33 (rnd:either (print "form 1") (print "form 2"))
34
35 ; repeat the form at most n-1 times
36 (rnd:rep n (print 1))
```

There are also some utils for random 3d vector numbers, see [src/rnd/3rnd.lisp](#).

- 
3. A simple graph data structure, `weir`, for working with vertices and edges. The structure is combined with a programming pattern for applying changes to the structure. The pattern relies on [alterations](#), see below. You can also manipulate the structure directly.

Below is a small example:

```
1 (let ((wer (weir:make))) ; use :dim 3 for 3d
2   ; add three edges
3   (loop repeat 3
4     do (weir:add-edge! wer
5         (weir:add-vert! wer (rnd:in-circ 200d0))
6         (weir:add-vert! wer (rnd:in-circ 200d0
7           :xy (vec:rep 500d0))))))
8   ; iterate verts
9   (weir:itr-verts (wer v) (print (weir:get-vert wer v)))
10
11  ; move a vert relativ to current position:
12  (weir:move-vert! wer 0 (vec:vec 1d0 2d0))
13  ; move a vert to an absolute position
14  (weir:move-vert! wer 1 (vec:vec 1d0 2d0) :rel nil)
15
16  ; iterate edges
17  (weir:itr-edges (wer vv) (print (weir:get-verts wer vv)))
18
19  ; edges are represented as lists of verts, and they are always
20  ; sorted with the smallest vert index first, so both
21  (weir:edge-exists wer '(0 1)) ; and
22  (weir:edge-exists wer '(1 0)) ; returns t
23
24  ; get edges incident to vert 0
25  (weir:get-incident-edges wer 0))
```

See the [examples](#) folder for more.

4. A series of other useful data structures and tools. E.g. a package for handling colors (`pigment`), splines (`bzspl`), and various vector and path functionality. Eg. `math`, `lin-path` and `simplify-path`.

```
1 (let* ((points (rnd:nin-circ 5 400d0))
2        (bz (bzspl:make points))
3        (lp (lin-path:make points)))
4   ; sample a point on the spline
5   (bzspl:pos bz (rnd:rnd))
6   ; sample a point on path
7   (lin-path:pos lp (rnd:rnd))
8
9   ; represent the spline with a limited number of points
10  (bzspl:adaptive-pos bz :lim 1d0))
11
12 ; return n numbers evenly spaced between a and b, inclusive
```

---

```

13 (math:linspace n a b :end t)
14
15 ; all fixnums from a to b-1
16 (math:range a b)
17
18 ; repeat the form n times
19 (math:nrep n (rnd:rnd))

```

#### 5. Orthogonal projection `ortho`:

```

1 (let ((proj (ortho:make :s 1d0
2                  :xy (vec:rep 500d0)
3                  :cam (vec:3vec 1000d0 1000d0 0d0)
4                  :look (vec:3zero))))
5   (multiple-value-bind (v d)
6     (ortho:project proj (rnd:3in-sphere :rad 300d0))
7     ; point in 2d
8     (print v)
9     ; distance from 3d point to camera plane
10    (print d))
11
12    ; update cam position and look at something else
13    (ortho:update proj :cam (vec:3vec 3d0 4d0 1d0)
14                      :look (vec:3rep 79d0)))

```

#### 6. A tool for drawing `svg` files: `draw-svg`. Mainly files that are good for plotting.

```

1 (let ((psvg (draw-svg:make :stroke "black"))
2       (pts (list (vec:vec 10d0 20d0) (vec:vec 20d0 30d0)
3                  (vec:vec 10d0 50d0))))
4   ; sw is the stroke width
5   (draw-svg:path psvg pts :sw 3d0)
6   (draw-svg:bzspl psvg pts :sw 3d0 :so 0.5d0)
7   (draw-svg:circ psvg (vec:rep 30d0) 10d0 :sw 3d0 :stroke "red")
8   (draw-svg:save psvg "filename"))

```

#### 7. A tool for drawing `pngs` called `sandpaint`. This package uses random sampling to draw. This creates a fairly distinct and gritty look in many cases. Also supports direct pixel manipulations and a few filters.

## Weir Graphs and Alterations

An `alteration` is a change that will be applied to the structure at the end of a given context. In practical terms, an alteration is a function that returns a `lambda` (or just a `lambda`).

The main motivation behind this is that this makes it possible to “queue” up a number of changes that will be applied at a later time. This makes it possible to access the state in the `weir` instance while you

---

are creating the alterations. Without there being any changes made to the state of the `weir` instance while the alterations are being created. Once all alterations are created, they will be applied.

Existing alterations in `weir` are postfixed `?`  by convention, and it might look like this:

```
1 (weir:with (wer %)
2   ( ; some code
3     (% (weir:add-vert? ...))
4       ; more code
5       (% (weir:add-edge? ...))))
```

all `(% ...)` forms inside the `weir` context will cause the alteration inside to be created and collected. They will be executed at the end of the context. if an alteration evaluates to nil, nothing will happen.

## Names and Args

You can assign a name (`:res`) to the result of an alteration. This makes it possible to create alterations that depend (`:arg`) on the result of other alterations:

```
1 (weir:with (wer %)
2   (let ((pt (...)))
3     (% (weir:add-vert? pt) :res :a) ; alteration result is named :a
4     (% (weir:add-vert? (vec:vec 1d0 2d0)) :res 'b) ; result is named 'b
5     (% (weir:add-edge? :a 'b) :arg (:a 'b)))) ; uses :a and 'b
```

Note that `res` must be a keyword, symbol, or a variable with keyword or symbol value. Similarly, `arg` must be a list of `res` elements that exist inside the same context. make sure that all elements in the `:arg` are present in the context, or the code will loop infinitely.

it is always possible to both reference future results, and assign the result a name. The order the order of `:res` and `:arg` does not matter:

```
1 (% (some-alteration? :a :b) :res :x :arg (:a :b)) ; is equivalent to
2 (% (some-alteration? :a :b) :arg (:a :b) :res :x)
```

Results will be available after the `(with:weir ...)` context. See `(get-alteration-result-list)` or `(get-alteration-result-map)`. Also, note that using the same name for multiple alterations *will* result in undefined behaviour.

## Dependency and Futures

You can consider a named alteration as something akin to a *future*; the value of `:res` is a reference to a value that does not yet exist. For this to work, any alteration that depends on a future that fails to be fulfilled will be skipped.

---

As an example, we can make the alteration `prob-add-edge?` like this:

```
1 (defun prob-add-edge? (l a b)
2   'add edge (a b) with probability p'
3   (lambda (w) (when (< (rnd:rnd 100d0) l)
4                     (add-edge! w a b))))
```

This will attempt to create edge `(a b)`, but only if the random number is less than `l`. This is to illustrate that the alteration may or may not attempt to `weir` instance. If no edge is created, the above lambda will return `nil`.

Here is an example of use:

```
1 ; context start
2 (let (wer (weir:make))
3
4   ; add some data to wer here ...
5
6   ; (% ...) is used to accumulate alterations
7   ; alterations are applied at the end of (weir:with ...)
8   (weir:with (wer %)
9     ; iterate all vertices in wer
10    (weir:itr-verts (wer v)
11      (% (move-vert? v (rnd:in-circ 10d0)))
12      ; w will be an arbitrary vertex in wer
13      (weir:with-rnd-vert (wer w)
14        (% (prob-add-edge? (weir:edge-length wer v w) v w))))))
```

The important thing to note here is that it is *crucial* that the length of edge `(v w)` is calculated outside `(defun prob-add-edge? ...)`. This ensures that `(weir:edge-length wer v w)` is the length of the edge *before* the calls to `(move-vert? ...)` have a chance to move either `v` or `w`.

Naturally, you can construct an alteration that checks the length of the edge inside the `lambda` in `(prob-add-edge? ...)`, but this will result in different behaviour. In this case, any edge length will be calculated while all the other vertices are moving around. Thus resulting in what can be considered “unexpected side effects”.

This becomes more clear if you consider an n-body simulation. If any single body in the simulation moves before you have calculated the force between each pair of bodies, you will get an incorrect result.

## Shadowing

As we have mentioned, arguments to an alteration may, or may not, exist right away. To handle this, arguments to an alteration will be shadowed before the alteration is collected. This applies to arguments that are atoms, or forms that do not contain a reference to a future alteration result. This is the

---

behaviour you will usually want in an example such as the one above. But it might cause unexpected behaviour.

As an example of what happens, consider the alteration:

```
1 (% (my-alteration? (first var-1)
2                      :a
3                      (my-function (rnd:rnd) :b (second var-2)))
4    :arg (:a :b))
```

This will be expanded by the macro to something similar to:

```
1 (LET ((#:|non-atom:91| (FIRST VAR-1)) ; values are evaluated at time of
      collection
2       (#:|non-atom:92| (RND:RND))
3       (#:|non-atom:93| (SECOND VAR-2)))
4  (LAMBDA (#:WNAME90)
5    (CASE (WEIR:::IF-ALL-RESOLVED #:ALT-RES88 (LIST :A :B))
6      (:OK ; :A and :B both have a value
7        (VALUES T
8          (FUNCALL
9            (THE FUNCTION
10             (MY-ALTERATION? #:|non-atom:91| (GETHASH :A #:
11              ALT-RES88)
12              (MY-FUNCTION #:|non-atom:92| (GETHASH :B #:
13              ALT-RES88)
14              #:|non-atom:93|)))
15             #:WNAME90)))
16      (:BAIL (VALUES T NIL)) ; either :A or :B returned nil. skip
17      alteration
18      (:WAIT (VALUES NIL NIL)))) ; :A or :B does not yet exist
```

As you can see, the variables/forms that will be shadowed here are: (`first var-1`), (`second var-2`) and (`rnd:rnd`).

You can use (`weir:with ... :bd t`) to see how an alteration is expanded. This might make it easier to find issues with shadowed/non-shadowed variables. Also, you can usually solve some problems you might encounter by defining custom alterations locally (but outside `weir:with`) using (`labels ()`).

## Looping

It is possible to use `:ref` and `:arg` inside loops as well. but it requires a bit more careful consideration. Here is an example:

```
1 (weir:with (wer % :db t)
2   (loop for x in (math:linspace 20 -20d0 20d0)
```

---

```

3      do (loop for z in (list 1d0 2d0)
4          do (let ((xy (vec:vec x y z))
5                  (s (vec:vec 1d0 80d0))
6                  (g (gensym "g"))) ; create a distinct name
7              (% (weir:add-grp? :name (gensym "line")) :res g)
8              (% (weir:add-path? (list (vec:sub xy s) (vec:add
9                                      xy s))
10                                     :g g)
11                                     :arg (g))))))

```

The second alteration will be expanded to:

```

1  (LET ((#:|non-atom:8| (LIST (VEC:SUB XY S) (VEC:ADD XY S))))
2    (LAMBDA (#:WNAME7)
3      ; every G is now a distinct future
4      (CASE (WEIR::-IF-ALL-RESOLVED #:ALT-RES3 (LIST G))
5        (:OK
6          (VALUES T
7            (FUNCALL
8              (THE FUNCTION
9                (WEIR:ADD-PATH? #:|non-atom:8| :G
10                 (GETHASH G #:ALT-RES3)))
11              #:WNAME7)))
12        (:BAIL (VALUES T NIL))
13        (:WAIT (VALUES NIL NIL))))

```

## Custom Alterations

You can define your own arbitrary alterations. There is an example of custom alterations and references in [examples/custom-alt.lisp](#).

## Final Note

In the previous implementations of the `(weir:with ...)` context, `(% ...)` was a function. This ensured that the arguments to the alteration, and indeed the lambda inside the alteration, was created before the end of the context. This eradicated the need for these complex shadowing rules. I'm not currently sure whether it is possible to avoid shadowing as long as some arguments do not exist at the time the alteration is collected.

Also, I use the term "shadowing" above. Not sure if this is really appropriate, but I failed to think of a better term



---

## Use

I use `weir` for most of the work that I post online (<https://inconvergent.net/>, <https://img.inconvergent.net/>, <https://twitter.com/inconvergent>). Both for raster images, as well as vector images for plotter drawings.

Here are some plotted examples:

- <https://inconvergent.net/2017/spline-script-plots/>
- <https://inconvergent.net/mechanical-plotter-drawings/>
- <https://inconvergent.net/mechanical-plotter-drawings/3/>
- <https://inconvergent.net/mechanical-plotter-drawings/5/>

## Writing

I have written about things related to this code (when it was called `snek`) at:

- <https://inconvergent.net/2017/snek-is-not-an-acronym/>
- <https://inconvergent.net/2017/a-method-for-mistakes/>
- <https://inconvergent.net/2017/arbitrary-alterations/>
- <https://inconvergent.net/2017/grains-of-sand/>
- <https://inconvergent.net/2017/a-propensity-for-mistakes/>

And recently at:

- <https://inconvergent.net/2020/future-alterations/>

## On Use and Contributions

This code is written for my personal use, and parts of it is rather experimental. Also, it is likely to change at my whim. For this reason I don't recommend depending on this library for anything.

I release it publicly in case people find it useful or interesting. It is not, however, intended as a collaboration/Open Source project. As such I am unlikely to accept PRs, reply to issues, or take requests.

## Installation and Dependencies

This code requires Quicklisp to install dependencies (which are listed in `weir.asd`). To install and load Weir, do:

```
1 (ql:quickload :weir)
```

---

If this does not work, Weir may not be in a place Quicklisp or ASDF can see them. To fix this, either

```
1 (load "weir.asd")
```

or, for a long term solution, push the directory in which Weir sits to the variable `quicklisp:*local-project-directories*`:

```
1 ; in your .sbclrc, for example:  
2 #+quicklisp  
3 (push "/path/to/dir/containing/weir" ql:*local-project-directories*)
```

The `fn` package (for generating file names) depends on the `fn` command from <https://github.com/inconvergent/fn>, but this is not necessary to use any of the other packages.

The code has only been tested in `Ubuntu 18.04 LTS` with `SBCL 2.0.1`. I've been told that examples work with `SBCL` in `macOS`.

## Tests

Run:

```
1 (asdf:test-system :weir)
```

## In Case of QL Version Issues

See <http://blog.quicklisp.org/2011/08/going-back-in-dist-time.html>

Summary:

```
1 (use-package :ql-dist)  
2 ; see versions  
3 (available-versions (dist "quicklisp"))  
4 ; select a dist version  
5 (install-dist  
6   "http://beta.quicklisp.org/dist/quicklisp/2019-03-07/distinfo.txt"  
7   :replace t)
```

## Thanks

I would like to thank:

- <https://twitter.com/RainerJoswig>
- <https://twitter.com/jackrusher>

- 
- <https://twitter.com/paulg>
  - <https://twitter.com/porglezomp>
  - <https://twitter.com/stylewarning>
  - <https://github.com/Hellseher>

Who have provided me with useful hints and code feedback.

The ASDF config and test setup was kindly suggested and implemented by Robert Smith (<https://twitter.com/stylewarning>). The remaining weirdness in the test system is my fault. Hope to fix it properly later.

Also, many thanks to <https://twitter.com/xach> for making Quicklisp.