SelaV$_{1D}$ Manual

Philipp Amstutz, Mathias Vogt
Deutsches Elektronen-Synchrotron DESY

2022-03-26
version 0.2.0

Contents

1 Introduction

SelaV$_{1D}$ is a semi-Lagrangian Vlasov simulation code in 1 degree of freedom, which is especially suited for simulating sparse phase-space densities (PSDs). Its primary intended area of application is the investigation of collective effects in the longitudinal phase-space of electron bunches in Free-Electron Lasers (FELs). Efficient treatment of sparse PSDs is achieved by tree-based domain decomposition, which allows to sample a PSD only in populated areas of the phase-space. To this end, SelaV$_{1D}$ utilizes the `libselav` library which implements arbitrary dimensional PSD-Trees.

2 Installation

In order to compile SelaV$_{1D}$ the following tools and libraries are required

- `GNU make`
- `libmatheval`
- `fftw`
- `X11 (optional)`

All of them should be available in the repositories of any major Linux distribution. The source code of SelaV$_{1D}$ is available as a tar-ball from `www.desy.de/~amstutz/selav/` . After downloading, decompress the archive and run `make` in the resulting directory:

```
tar xzf selav-0.0.2.tar.gz
cd selav-0.0.2/
make
```

Then, optionally, move or link the binary `selav1d` to a directory in your `PATH`.

3 Command Language

The input format for `selav1d` is a simple interpreted language with a python-influenced, C-style syntax. It supports floating point arithmetic, variable assignment and macros. Every command line ends needs to end with a semi-colon (`;`). White spaces (i.e. tabs, carriage returns, spaces, etc.) are ignored. `selav1d` expects input on standard input. For testing purposes the interpreter can be used interactively. To run commands stored in a file (e.g. `example.inp`) use the input redirection of your shell to feed them to SelaV$_{1D}$:

```
selav1d < example.inp
```

3.1 Comments

Both, C and C++ style comments are supported. Anything between a pair of `/*` and `*/` is ignored, as well as anything after a `//` until the end of the line.

3.2 Data types

SelaV$_{1D}$ manages objects of four data types. FLT for floating point numeric values, STR for strings, PSD for tree-PSDs, and MAP for maps.
Numeric values can be specified in the usual decimal or exponential notation, e.g. `42`, `13.37e2`, or `5.0e-3`. Arrays of FLTs are specifed by enclosing them in brackets, e.g. `[1, 2e3, 42]`.

Strings are specified by enclosing them in either double quotation marks (`"`) or a pair of curly braces ({ , }), where the latter syntax is intended for the definition of macros, see `eval` PSD-types and MAP-types occur only as return values of functions and can not be specified directly.

3.3 Variables

Variables can be assigned using the `=` operator. A valid variable name is any combination of the letters in the english alphabet (upper and lower case), the numbers 0-9, and an underscore (_).
The `who` command lists all currently defined variables.

3.4 Arithmetic

Elementary arithmetic expressions are supported in infix notation using the standard operators `+`,`-`, `/`, and `*`. Trigonometric functions `sin`, `cos`, and `tan` take their arguments in radians. Parentheses can be used for grouping expressions. Arithmetic expressions may appear anywhere a numeric value is expected. The constant `pi` is defined.

```
my_2pi = 2*pi;
a = 42;
print( sin(my_2pi * (a+1.2)) );

>>> 9.5105651629515031e-01
```

3.5 Keyword Arguments

Some functions take optional arguments in the form of keywords; for example

```
psd = psd_gauss(sig_q=0.5, sig_p=0.5);
```

Keywords are always optional; if a keyword is not specified it is assigned a default value. See Section 6 for a list of all functions with their keyword arguments and their default values.

3.6 Control Flow

Conditiontioal execution is supported via `if` (`else`) statements. The condition is supplied in the form of a FLT object; values > 0 are interpreted as "true" and values ≤ 0 as "false".

```
if(2 - 4) {
    print("Two is strictly greater than four.");
} else {
    print("Four is strictly greater than two.");
};

>>> Four is strictly greater than two.
```

The `do` statement provides a simple looping construct. It unconditionally executes a block of commands a given number of times.

```
n=0; do(3) {
    print(n);
    n=n+1;
};

>>> 0.0000000000000000e+00
>>> 1.0000000000000000e+00
>>> 2.0000000000000000e+00
```

4 Usage

This section will give a quick introduction on how to use SelaV$_{1D}$.

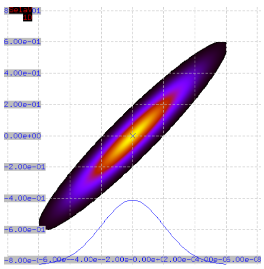
4.1 Initializing PSDs

Typically, the first step in any SelaV$_{1D}$ run is to initialize a phase-space density. There are a number of functions available (see ??), which produce either analytically defined PSDs or generate them from particle distributions. The function `psd_gauss`, for instance, produces a bivariate Gaussian distribution.

```
psd = psd_gauss();
```

It returns a PSD object that needs to be assigned to a variable (here `psd`), so that it can be referenced later on.
Especially when setting up a simulation for the first time, it can be helpful to visualize the phase-space densities at different steps of the simulation. The command `show` starts an X window that lets the user explore the PSD interactively. See the entry in the Reference section for a complete list of its capabilities. If the `file` keword is supplied to `show`, instead of starting an interactive window it writes an image in PPM format to the specified file. The PPM format is a straight-forward ASCII image format and the only format the author saw himself able to implement without the use of an external library. PPM images can be easily converted to more common formats with image manipulation programs such as the ImageMagick suite or the GIMP.

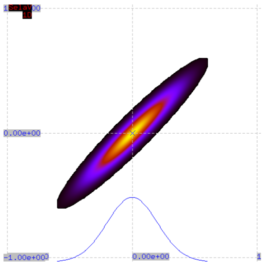
```
psd = psd_gauss(correlation=0.95);
show(psd,file="gauss.ppm");
```

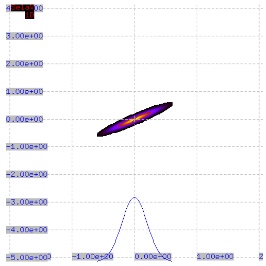


gauss.ppm

All functions that initialize a PSD allow the keywords `limits`, `nexp`, `depth`, `weight`. With `limits` the size of the simulation window is specified. This need to be larger that the support of the phase-space density.

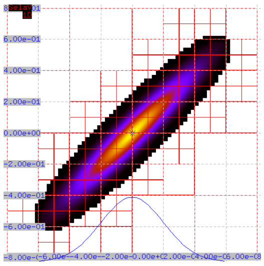
```
psd = psd_gauss(correlation=0.95,
                limits=[-1,-1,1,1]);
show(psd,file="limits_a.ppm");

psd = psd_gauss(correlation=0.95,
                limits=[-2,-5,2,4]);
show(psd,file="limits_b.ppm");
```

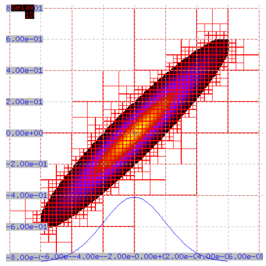

limits_a.ppm


limits_b.ppm

The depth of the tree structure used for the domain decomposition of the PSD can be selected with the `depth` keyword.

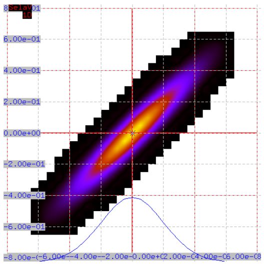
```
psd = psd_gauss(correlation=0.95,depth=4);
show(psd,file="depth_4.ppm",cells=1);

psd = psd_gauss(correlation=0.95,depth=6);
show(psd,file="depth_6.ppm",cells=1);
```

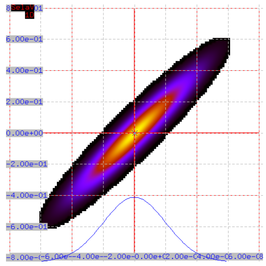

depth_4.ppm


depth_6.ppm

Only on the smallest cells, the so-called leafs, values of the PSD is stored in memory. The number of sample points per leaf is controlled by the keyword `nexp`, which is the $\log_2$ of the number of points per dimension. For example, `nexp`$= 3$ means $2^3 = 8$ points per dimension, resulting in a total of 64 sample points per leaf.

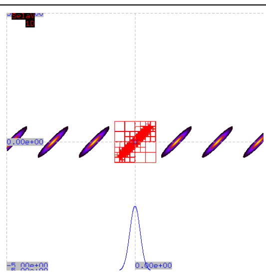
```
psd = psd_gauss(correlation=0.95,
                depth=2,nexp=3);
show(psd,file="nexp_3.ppm",cells=1);

psd = psd_gauss(correlation=0.95,
                depth=2,nexp=5);
show(psd,file="nexp_5.ppm",cells=1);
```

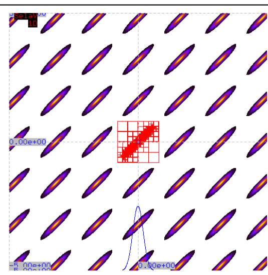

nexp_3.ppm


nexp_5.ppm

The total resolution of the PSD is therefore given by $2^{-(nexp+depth)}$ relative to the size of the simulation window. Hence, to get higher resolution either *nexp* or *depth* can be increased. Larger values for *depth* lead to a larger tree structure and therefore to more computational overhead. Choosing *depth* too low will lead to the tree covering more phase-space than necessary, which is memory efficient. Hence, *depth* should be chosen as small as possible but large enough so that the support of the PSD is well approximated by the tree. After *depth* is fixed, *nexp* can increased to achieve the required final resolution.
PSDs can be created on different topologies ($\mathbb{R}^2$, $S^1 \times \mathbb{R}^1$, $\mathbb{R}^1 \times S^1$, and S^2) using the keyword `topology`.

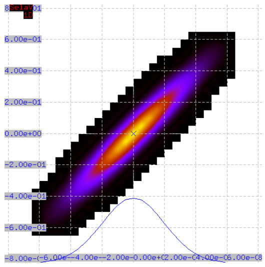
```
psd = psd_gauss(correlation=0.95,
                topology=1);
show(psd,file="topo_1.ppm",
         limits=[-5,-5,5,5],cells=1);
psd = psd_gauss(correlation=0.95,
                topology=3);
show(psd,file="topo_3.ppm",
         limits=[-5,-5,5,5],cells=1);
```


topo_1.ppm

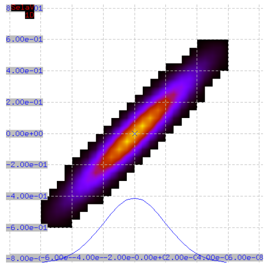

topo_3.ppm

Multiple options are available for the interpolation method used to evaluate the phase-space density. They can be selected via the `interpolation` keyword. Currently implemented methods are nearest-neighbor, bilinear and bicubic interpolation.

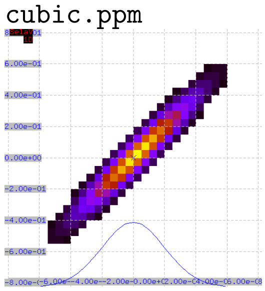
```
psd = psd_gauss(correlation=0.95,
                depth=2,nexp=3,
                interpolation=0);
show(psd,file="nearest.ppm");
psd = psd_gauss(correlation=0.95,
                depth=2,nexp=3,
                interpolation=1);
show(psd,file="linear.ppm");
psd = psd_gauss(correlation=0.95,
                depth=2,nexp=3,
                interpolation=2 );
show(psd,file="cubic.ppm");
```



cubic.ppm



linear.ppm



nearest.ppm

4.2 Propagating PSDs

After a PSD is initialized, the next step is to execute one or more propagation steps. Given an initial PSD Ψ_0 and a *map* $f\colon \mathbb{R}^2 \to \mathbb{R}^2$, a propagation step will return a new PSD Ψ_1 given by

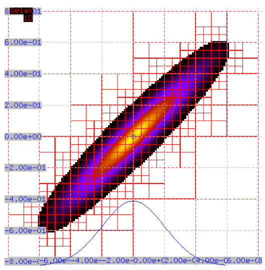
$$\Psi_1(\cdot) = \Psi_0(f^{-1}(\cdot)). \tag{1}$$

In short, if f is the solution of the single particle equations of motion, then $\Psi_0(f^{-1}(\cdot))$ is the solution of the Vlasov equation, i.e. the equation of motion of the phase space density with the initial condtion given by Ψ_0.
SelaV$_{1D}$ handles maps in the form of MAP type objects. There a number of functions returning MAP type objects, see Reference for a complete list. For example `map_kickpoly()` produces a kick map with polynomial kick function.

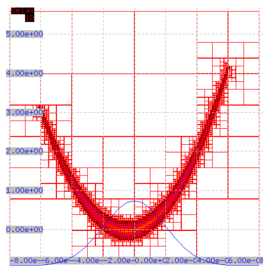
```
f = map_kickpoly([0,0,10]);
```

A propagation step is executed by calling the funtion `propagate`. It takes the initial PSD and the map as arguments and returns the new PSD. A `propagate` call is the direct equivalent of Equation 1.

```
psd0 = psd_gauss(correlation=0.95,
                 depth=5);
f = map_kickpoly([0,0,10.]);
psd1 = propagate(psd0,f);
show(psd0,file="psd0.ppm",cells=1);
show(psd1,file="psd1.ppm",cells=1);
```



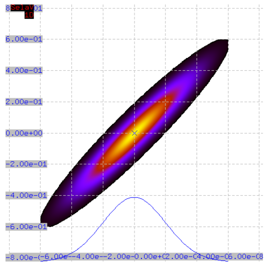
psd0.ppm

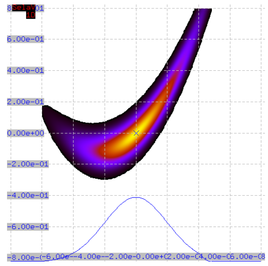


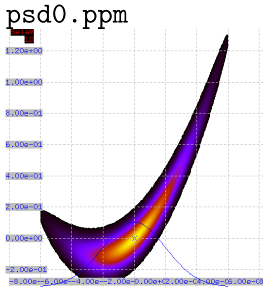
psd1.ppm

Note that the outer rectangle of the new tree has automatically adapted so that it can fit the support of the new PSD. Further, the recursion depth has also been increased automatically to keep the sampling resolution constant. This behaviour can be controlled with the `box` and `center` keywords. Setting them to `"KEEP"` will keep the old box dimensions.

```
psd0 = psd_gauss(correlation=0.95);
f = map_kickpoly([0,0,2]);
psd1 = propagate(psd0,f,box="KEEP",
                  center="KEEP");
psd2 = propagate(psd0,f,box="KEEP");
show(psd0,file="psd0.ppm");
show(psd1,file="psd1.ppm");
show(psd2,file="psd2.ppm");
```

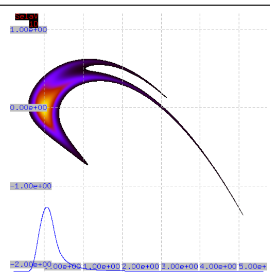

psd0.ppm


psd1.ppm


psd2.ppm

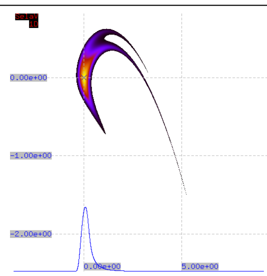
Similarly, the resolution of the new PSD can be controlled with the keywords `depth` and `nexp`. Typically, multiple maps will need to be applied to an initial PSD before the final result is attained. Of course, this can be achieved by applying the maps successively by calling `propagate` multiple times.

```
psd = psd_gauss(correlation=0.1);
m1 = map_kickpoly([0,0,3]);
m2 = map_driftpoly([0,0,3]);
m3 = map_kickpoly([0,0,-0.1]);
psd = propagate(psd,m1);
psd = propagate(psd,m2);
psd = propagate(psd,m3);
show(psd,file="psd.ppm");
```


psd.ppm

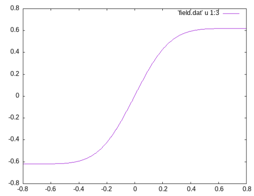
Executing a simulation steps is a computationally intensive operation. But oftentimes some of the intermediate PSDs are of no particular interest. In that case it is advisable to compose multiple maps into one MAP object. This allows to do the same calculation with only a single `propagate` call.

```
psd = psd_gauss(correlation=0.1);
m1 = map_kickpoly([0,0,3]);
m2 = map_driftpoly([0,0,3]);
m3 = map_kickpoly([0,0,-0.1]);
M = map_compose(m1,m2,m3);
psd = propagate(psd,M);
show(psd,file="psd.ppm");
```

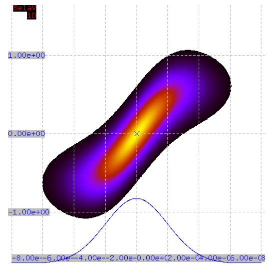

psd.ppm

An especially interesting type of maps are *collective maps*, such as for instance `map_poisson` and `map_spacecharge`. These map functions take a PSD as one of there arguments and return a map calculated based on that PSD. It is to be noted that in SelaV$_{1D}$ once a MAP object is generated in this way it remains independent on the PSD it was calculated from. The PSD object can be modified or deleted afterwards, without affecting the previously calculated map.

```
psd = psd_gauss();
m = map_poisson(psd, "-2*(step(q)-0.5)",
   file="field.dat",npad=2);
psd = propagate(psd,m);
show(psd,file="psd.ppm" );
```



field.png



psd.ppm

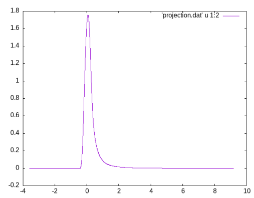
4.3 Generating Output

After executing all desired propagation steps, information about the resulting PSD can be gathered and output for further evaluation outside of SelaV$_{1D}$.
Generally all output that would be written to the standard output can be redirected using the > and >> operators. They redirect the output of a preceding command block to a specified file. Both operators create the file if it does not exist. If the file already exists, the > will delete its content, while the >> operator will append to it.

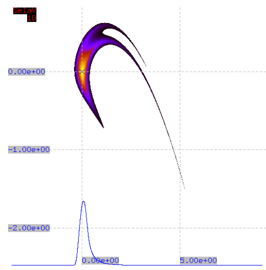
```
{
print("This text will be written into a file.");
print("More text.");
} > "output.dat" ;
```

Further, there are a number of functions to generate output from PSD objects. The function `write_projection` calculates the projection of the PSD along an axis and writes the result to a file.

```
psd = psd_gauss(correlation=0.1);
m1 = map_kickpoly([0,0,3]);
m2 = map_driftpoly([0,0,3]);
m3 = map_kickpoly([0,0,-0.1]);
M = map_compose(m1,m2,m3);
psd = propagate(psd,M);
show(psd,file="psd.ppm");
write_projection(psd,"projection.dat",2);
```



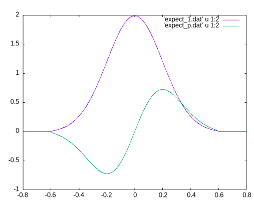
projection.png



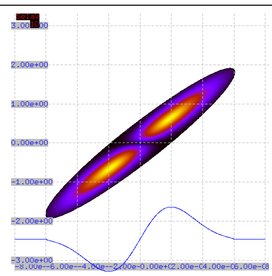
psd.ppm

In conjunction with the `modify` it is for instance possible to calculate local, unnormalized expected values of arbitrary functions.

```
psd = psd_gauss();
M = map_kickpoly([0,3]);
psd = propagate(psd,M);
write_projection(psd,"expect_1.dat",2);
modify(psd,"psi*p");
show(psd,file="psd.ppm");
write_projection(psd,"expect_p.dat",2);
```

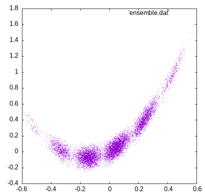


projection.png

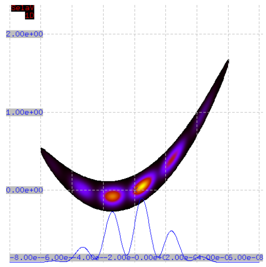


psd.ppm

```
psd0 = psd_gauss(correlation=0.95);
m = map_kickpoly([0,0,3]);
psd1 = propagate(psd0, m);
modify(psd1, "psi*(1+0.8*sin(2*pi*5*q))");
show(psd1, file="psd1.ppm");
write_ensemble(psd1, "ensemble.dat", n=1e4);
```



`ensemble.png`



`psd1.ppm`

5 Application to Beam Dynamics

TODO

6 Reference

average – Function average from discrete values

Synopsis

FLT average(FLT a, FLT b, FLT[n] x, FLT[n] y);

Description

Calculates the average function value

$$\frac{\int_a^b f(x)\mathrm{d}x}{b-a} \tag{2}$$

where $f(x)$ is the linear interpolant with $f(x_i) = y_i$.

Return Value

Average function value.

centroid – Calculate PSD centroid

Synopsis

FLT[2] centroid(PSD Ψ);

Description

Calculate the centroid $\int_{\mathbb{R}^2} \Psi(z)\, z\, \mathrm{d}z$.

Return Value

Array of size 2 containing the centroid of Ψ.

Example

```
psi = psd_test();
centroid(psi);

>>> [-4.844542e-02, -3.541503e-02]
```

charge – Charge (or Weight) of a PSD

Synopsis

FLT charge(PSD Ψ);

Description

PSD objects carry an additional value, the “weight”. It takes the function in beam dynamics calculation it takes the role of the bunch charge in Coulomb.

Return Value

Returns the charge (“weight”) associated with Ψ.

chdir – Change working directory

Synopsis

example(STR *dir*);

Description

Changes working directory to *dir*, which can be an absolute or relative path name.

chicanecoef – Taylor coefficients of C-shape chicane

Synopsis

FLT[n] chicanecoef(FLT n, FLT ϕ, FLT l_B, FLT l_D);

Description

Calculates the first n Taylor coefficients of the drift map of a symmetric C-shape chicane with bending angle ϕ (in rad), dipole length l_B, and dipole distance l_D (both in meter).

Return Value

Array of size n containing the chicane drift coefficients.

defined – Check whether symbol is defined

Synopsis

FLT example(*symbol*);

Description

Check whether *symbol* is already defined.

Return Value

Returns 1 if *symbol* is defined, and 0 otherwise.

Example

```
a=2;
defined(a);
defined(b);

>>> 1.0000000000000000e+00
>>> 0.0000000000000000e+00
```

do – Execute commands multiple times

Synopsis

do(FLT *n*) { *commands* };

Description

Executes *commands* *n* times.

Example

```
i=1;
do(4) {
    i=i*2;
    print(i);
};

>>> 2.0000000000000000e+00
>>> 4.0000000000000000e+00
>>> 8.0000000000000000e+00
>>> 1.6000000000000000e+01
```

eval – Evaluate commands in a string

Synopsis

eval(STR *string*);

Description

Evaluate the commands in *string*.

Example

```
cmd = {
    print("Hello!");
    a = 3;
};
eval(cmd);
print(a);

>>> Hello!
>>> 3.0000000000000000e+00
```

format – Format numbers into a string

Synopsis

format(STR *format*, FLT f_0, ..., FLT f_n);

Description

Formats numbers $f_0, \ldots, f_n$ into a string according to the `printf`-style format *format*. Only conversion specifiers that accept double arguments are allowed in *format*.

Example

```
a=1; b=32; c=0;
numbers = format("%03g -- %e -- %g",a,b,c);
print(numbers);

>>> 001 -- 3.200000e+01 -- 0
```

for – Loop over array

Synopsis

for(*symbol* in *array*) { *commands* };

Description

Executes *commands* multiple times with *symbol* successively taking each value in *array*.

Example

```
a = [1,5,32,2];
for x in a {
    print(a[2]* x);
};

>>> 3.2000000000000000e+01
>>> 1.6000000000000000e+02
>>> 1.0240000000000000e+03
>>> 6.4000000000000000e+01
```

getcwd – Name of current working directory

Synopsis

STR getcwd();

Return Value

String containing the name of the current working directory.

Example

```
chdir("/tmp");
getcwd();

>>> '/tmp'
```

getenv – Read environment variable

Synopsis

STR getenv(STR *name*);

Description

Reads the environment variable *name*.

Return Value

String containing the value of the environment variable.

Example

```
getenv("TERM");

>>> 'xterm'
```

if – Conditional execution

Synopsis

if(FLT x) { *commands* } [else { $commands_b$ }];

Description

If x is larger than 0 executes *commands*. If x is smaller or equal 0 and the else clause is given executes $commands_b$.

Example

```
if(2) {
    print("true");
} else {
    print("false");
};

>>> true
```

include – Evaluate commands in a file

Synopsis

include(STR *file*);

Description

Reads and evaluates commands from *file*.

Example

```
// Content of file example.inp:
//
//    a = 42;
//    print("Hello!");
//

include("example.inp");
print(a);

>>> Hello!
>>> 4.2000000000000000e+01
```

integral – Calculate expectation values

Synopsis

FLT integral(PSD Ψ[, STR f]);

Description

If f is given, calculate

$$\int_{\mathbb{R}^2} \Psi(z) f(z) \mathrm{d}z, \tag{3}$$

otherwise calculate

$$\int_{\mathbb{R}^2} \Psi(z) \mathrm{d}z. \tag{4}$$

In the function string f, use `q` and `p` as the phase-space coordinates.

Return Value

Expectation value of 1, or f respectively.

Example

```
psi = psd_test();
integral(psi);
integral(psi,"q*p");

>>> 4.0852578184938004e-01
>>> -1.5133745968341827e-02
```

linspace – Create array of equally spaced values

Synopsis

FLT linspace(FLT *start*, FLT *end*, FLT n);

Return Value

Array of size n containing equally spaced values between *start* (inclusive) and *end* (inclusive).

Example

```
linspace(0,1,5);

>>> [0.000000e+00, 2.500000e-01, 5.000000e-01, 7.500000e-01, 1.000000e+00]
```

load – Load a PSD from a file

Synopsis

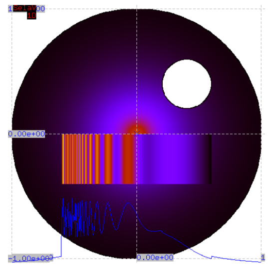
PSD load(STR $file$);

Description

Load a PSD from a file $file$ that has been created by the `save` function.

Example

```
filename="psd.dat";
psd0 = psd_test();
save(psd0,filename);
psd1 = load(filename);
show(psd1,file="psd1.ppm");
```



psd1.ppm

map_analytic – Map defined by analytic expressions

Synopsis

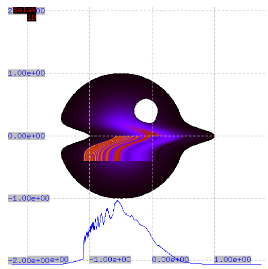
MAP map_analytic(STR f_q, STR f_p);

Description

This function constructs the time-one map of the flow $\phi\colon t, z \mapsto (f_q(t,z), f_p(t,z))$.
The string representation of f_q and f_p use the symbols `t`, `q`, and `p` to refer to the independent variable t, and the phase-space coordinates $(q,p) \equiv z$ respectively.
No checks are conducted whether ϕ fulfills the flow properties, nor whether the resulting map is symplectic. Specifying a flow for which $\phi(t,\cdot) \circ \phi(-t,\cdot) \equiv \mathrm{Id}$ does not hold, can lead to unexpected behaviour.
The expression is evaluated using library "GNU libmatheval". See its documentation for a list of all supported features. Due to technical issues with the "GNU libmatheval" library, this function currently is not as computationally efficient as the others. Using it can slow down the simulation.

Example

```
psd0 = psd_test();
m = map_analytic(
        "q+t*0.5*(exp(-0.5*p*p/0.1^2)-exp(0))",
        "p");
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd1.ppm");
```



psd1.ppm

map_cavity – Cavity map

Synopsis

MAP map_cavity(*keywords*);

Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + k(q) - k(0) \end{pmatrix},$$

with

$$k(q) = \begin{cases} A\,\cos(2\pi\, f/c\, q + \phi) & mode = 0 \\ -A\, 2\pi\, f/c\, \sin(\phi) & mode = 1 \end{cases}$$

Keywords

Keyword	Type	Default	Unit	Description
`freq`	FLT	10^9	Hz	Frequency f.
`phase`	FLT	0	rad	Phase ϕ.
`ampl`	FLT	10^6	V	Amplitude A.
`mode`	FLT	0		Selects cavity model.

map_chicanec – C-shape Chicane map

Synopsis

MAP map_chicanec(*keywords*);

Description

Returns a kick-map corresponding to a C-shape chicane.

Keywords

Keyword	Type	Default	Unit	Description
`alpha`	FLT	10	rad	Bending angle.
`lD`	FLT	0.5	m	Drift length.
`lB`	FLT	0.5	m	Magnet length.
`energy`	FLT	0	eV	Beam energy.
`mode`	FLT	0		Selects chicane model.

map_compose – Compose multiple maps

Synopsis

MAP map_compose(MAP f_n, ..., MAP f_0);

Description

Returns the map

$$\bigcirc_{i=0}^{N} f_i \equiv f_n \circ \cdots \circ f_0.$$

map_csr – CSR kick map

Synopsis

MAP map_csr(PSD Ψ, *keywords*);

Description

TODO

Keywords

Keyword	Type	Default	Unit	Description
`angle`	FLT	1	rad	bending angle of the dipole
`length`	FLT	1	m	effective length of the dipole
`energy`	FLT	$m_e\, c^2$	eV	total particle energy
`s`	FLT	length	m	position inside the dipole (arclength)
`ds`	FLT	length	m	propagation length ("time step")
`fudge`	FLT	1		artificial factor to scale fields with
`npad`	FLT	2		FFT padding factor
`mode`	FLT	2		Select CSR model
`filter`	FLT	0		Selects smoothing filter type
`filterwidth`	FLT	0.1	f_{nyquist}	Width of the smoothing filter
`large_dist_cutoff`	FLT	0	R_{bend}/γ^3	Truncate CSR kernel for $(s - s')$ smaller than this value
`transient`	FLT	1		0: no transient terms, 1: both terms, 2: first term only
`file`	STR			Write field to this file
`debug`	STR			Write details of field calculation to this file
`verbose`	FLT	0		If true, print additional information

map_driftl – Linear drift map

Synopsis

MAP map_driftl(FLT l);

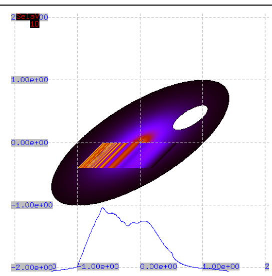
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q + l\, p \\ p \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_driftl(1);
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_driftpoly – Polynomial drift map

Synopsis

MAP map_driftpoly(FLT[n] a);

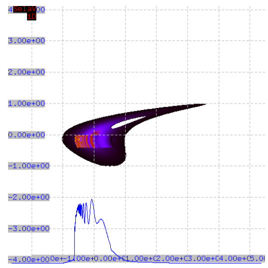
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q + \sum_{i=0}^{n} a_i\, p^i \\ p \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_driftpoly([0,0.5,2,1]);
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_driftsine – Sinusoidal drift map

Synopsis

MAP map_driftsine(FLT a, FLT k, FLT ϕ);

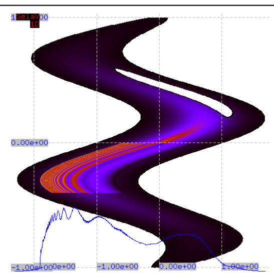
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q + a\,\sin(k\,p + \phi) \\ p \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_driftsine(1, 2*pi, 0.1*pi);
psd1 = propagate(psd0,m);
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_hyperbolic – Hyperbolic map

Synopsis

MAP map_hyperbolic(FLT a);

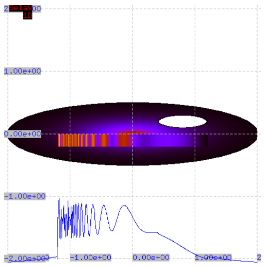
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q\,a \\ p\,a^{-1} \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_hyperbolic(2.0);
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_identity – Identity map

Synopsis

MAP map_identity();

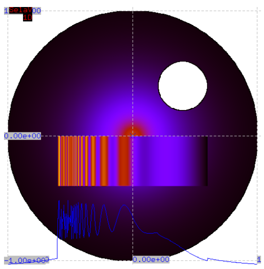
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_identity();
psd1 = propagate(psd0,m);
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_kickl – Linear kick map

Synopsis

MAP map_kickl(FLT k);

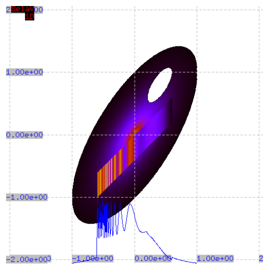
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + k\,q \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_kickl(1);
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_kickpoly – Polynomial kick map

Synopsis

MAP map_kickpoly(FLT[n] a);

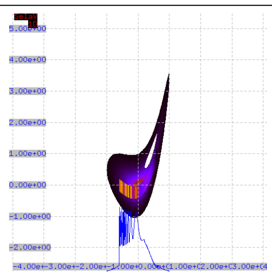
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + \sum_{i=0}^{n} a_i\, q^i \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_kickpoly([0,0.5,2,1]);
psd1 = propagate(psd0,m,box="EQUAL");
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_kicksine – Sinusoidal kick map

Synopsis

MAP map_kicksine(FLT a, FLT k, FLT ϕ);

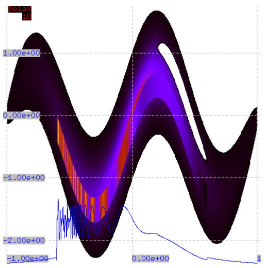
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + a\,\sin(k\,q + \phi) \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_kicksine(1, 2*pi, 0.1*pi);
psd1 = propagate(psd0,m);
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_poisson1d – Solve 1D Poisson's equation

Description

TODO

map_poisson – Poisson-type collective kick map

Synopsis

MAP map_poisson(PSD Ψ, STR g, *keywords*);

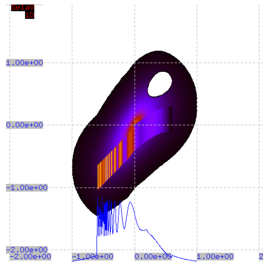
Description

Returns the Poisson-type kick map given by the convolution of the Greens function g with the spatial density

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + \left[g(\cdot) * \int_{\mathbb{R}} \Psi(\cdot, p)\mathrm{d}p \right](q) \end{pmatrix}.$$

Example

```
psd = psd_test(); normalize(psd);
m = map_poisson(psd, "2*(0.5-step(q))");
psd = propagate(psd,m,box="EQUAL");
show(psd,file="psd.ppm");
```



psd.ppm

map_rotate – Rotation map

Synopsis

MAP map_rotate(FLT α);

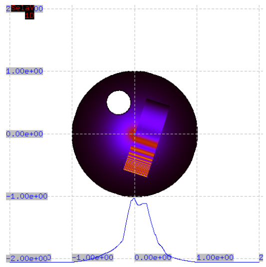
Description

Returns the map

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} q \\ p \end{pmatrix}.$$

Example

```
psd0 = psd_test();
m = map_rotate(2*pi*0.2);
psd1 = propagate(psd0,m);
show(psd1,file="psd0.ppm");
```



psd0.ppm

map_spacecharge – Spacecharge kick map

Synopsis

MAP map_spacecharge(PSD Ψ, *keywords*);

Description

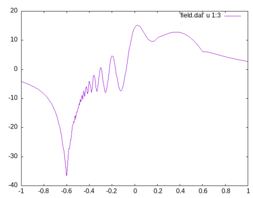
Returns the kick-map generated by the spacecharge fields of the phase-space density Ψ.

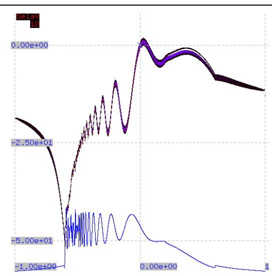
Keywords

Keyword	Type	Default	Unit	Description
`beamsize`	FLT	1	m	Average transverse beamsize.
`length`	FLT	1	m	Length of the drift space.
`energy`	FLT	$m_e\,c^2$	eV	Total particle energy.
`beamsize_factor`	FLT	1.747		Factor to multiply beamsize with.
`file`	STR			File name to write field data to.

Example

```
psd0 = psd_test(weight=500e-12);
m = map_spacecharge(psd0,
        beamsize=0.05,
        file="field.dat");
psd1 = propagate(psd0,m);
show(psd1,file="psd1.ppm");
```


`field.png`


`psd1.ppm`

maximum – Maximum of a PSD

Synopsis

FLT maximum(PSD Ψ);

Description

Returns value of the largest sample of Ψ.

Return Value

Maximum of Ψ.

mkdir – Creates new directory

Synopsis

mkdir(STR *dir*);

Description

Creates new directory named *dir* in the current working directory.

modify – Modify a PSD

Synopsis

modify(PSD Ψ, STR fnc);

Description

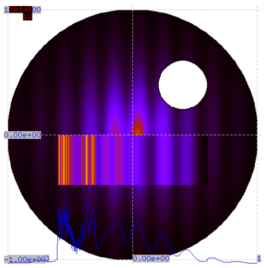
Modify a PSD with the function f given in the string fnc in the following way

$$\Psi(z) \mapsto \begin{cases} f(\Psi(z), z) & z \in \mathrm{supp}\Psi \\ 0 & \text{else} \end{cases}.$$

In fnc the symbols `psi`, `q`, and `p` refer to the local value of the PSD, and the phase-space coordinates $(q, p) \equiv z$ respectively. The expression fnc is evaluated using the library “GNU libmatheval”. See its documentation for a list of all supported features.

Example

```
psd = psd_test();
modify(psd,"psi*(1+0.5*sin(2*pi*5*q))");
show(psd,file="psd.ppm");
```



psd.ppm

multiply – Multiply PSD with a constant

Synopsis

multiply(PSD Ψ, FLT a);

Description

Multiplies the PSD Ψ with a constant factor

$$\Psi \mapsto a\,\Psi.$$

Example

```
psd = psd_test(); normalize(psd);
print(integral(psd));
multiply(psd,3.2);
print(integral(psd));

>>> 9.9999999999998146e-01
>>> 3.1999999999999846e+00
```

noise – Add noise to PSD

Synopsis

noise(PSD Ψ, FLT a, *keywords*);

Description

Scales all values of Ψ by a random value. If `type` is 0, then the values are scaled according to

$$\Psi_{ij} \mapsto (1 + a\, x_{ij})\Psi_{ij} \tag{5}$$

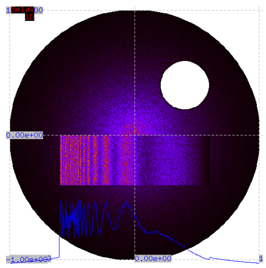
where $x_{ij} \in [-1, 1]$ is sampled from a uniform distribution. If `type` is 1, then a is interpreted as the total number of particles and the values are scaled according to a Poisson distribution with a mean of the local expected value of the number of particles.

Keywords

Keyword	Type	Default	Unit	Description
`seed`	FLT	0		seed value for the random number generator
`type`	FLT	0		0: uniform noise, 1: Poisson noise

Example

```
psi = psd_test();
noise(psi, 0.8);
show(psi, file="noise.ppm");
```



`noise.ppm`

normalize – Normalize integral of PSD

Synopsis

normalize(PSD Ψ)

Description

Normalizes the integral of Ψ to unity

$$\Psi \mapsto \Psi / \int_{\mathbb{R}^2} \Psi(z) \mathrm{d}z.$$

Example

```
psd = psd_test();
print(integral(psd));
normalize(psd);
print(integral(psd));

>>> 4.0852578184938004e-01
>>> 9.9999999999998146e-01
```

plot – Save a grayscale image of a PSD

Synopsis

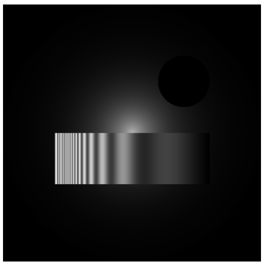
plot(PSD Ψ, STR $fname$);

Description

Saves an image of the PSD Ψ in the pgm format to the file $fname$. The image is not downsampled, i.e. if the PSD has $nexp = a$ and $depth = b$, the resulting image will have the dimensions $2^{(a+b)} \times 2^{(a+b)}$.

Example

```
psd = psd_test();
plot(psd, "plot.pgm");
```



plot.pgm

print – Print objects

Synopsis

print(PSD/STR/FLT obj_0,...,PSD/STR/FLT obj_n);

Description

Prints objects to standard output. If the object is of type PSD information about its tree-structure is printed. For MAP type objects no output is produced.

Example

```
mystr = "Hello!";
myflt = 12;
myarr = [1,2,3,5];
mypsd = psd_gauss(); normalize(mypsd);
mymap = map_kickl(2);
print("mystr is: ", mystr);
print("myflt is: ", myflt);
print("myarr is: ", myarr);
print("mypsd is: ", mypsd);
print("mymap is: ", mymap);

>>> mystr is:  Hello!
>>> myflt is:  1.2000000000000000e+01
>>> myarr is:  1.0000000000000000e+00 2.0000000000000000e+00 3.0000000000000000e+00 5.0000000
>>> mypsd is:
>>> Center:  0.000000e+00 0.000000e+00
>>> Width:   8.000000e-01 8.000000e-01
>>> Limits:  -8.000000e-01 -8.000000e-01 8.000000e-01 8.000000e-01
>>> Ref.Pt:  0.000000e+00 0.000000e+00
>>> Depth:   7
>>> nexp:    2
>>> Weight:  1.000000e+00
>>> Ipol:    2
>>> Topo::   0
>>> Leafs:   7896
>>> Integrl: 1.000000e+00
>>>
>>> mymap is:
```

propagate – Propagate a PSD according to a map

Synopsis

PSD propagate(PSD Ψ, MAP f, *keywords*);

Description

Executes a Perron-Frobenius step, i.e. returns the phase-space density $\Psi \circ f^{-1}$.
The `box` keyword determines how the bounding box of the new PSD is chosen. Possible values are `"KEEP"` (new box is equal to the initial), `"AUTO"` (each axis is scaled independently in powers of 2 to fit the new PSD), and `"EQUAL"` (both axes are scaled by the same power of 2 to fit the new PSD).
The `center` keyword determines how the center point of the new PSD is chosen. Possible values are `"KEEP"` (new center is equal to the inital), `"AUTO"` (new center is the center of the minimum bounding box of the support of the new PSD).
nexp and *depth* choose the resoltution parameters of the new PSD. Setting *nexp* to 0 will keep the value of the inital PSD. Setting *depth* to zero will keep the depth of the inital PSD plus the $\log_2$ of the largest scaling factor of the axes.

Keywords

Keyword	Type	Default	Unit	Description
`center`	STR	`"AUTO"`		Method to determine the center point of the new tree.
`box`	STR	`"AUTO"`		Method to determine the width of the new tree.
`nexp`	FLT	0		New sample rate.
`depth`	FLT	0		New recursion depth.
`t`	FLT	1		Independent variable.

psd_analytic – Initialize a PSD from analytic expression

Synopsis

PSD psd_analytic(STR *str*, *keywords*);

Description

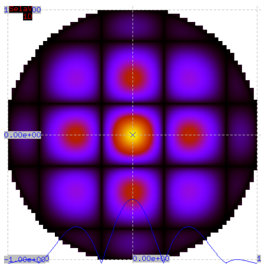
Initialize a PSD from the analytic expression given in *str*. In *str* the symbols `q` and `p` are used to refer to the phase-space coordinates. The expression is evaluated using library "GNU libmatheval". See its documentation for a list of all supported features.

Keywords

Keyword	Type	Default	Unit	Description
`depth`	FLT	7		Refinement depth of the tree.
`nexp`	FLT	2		$\log_2$(sample points / dimension).
`weight`	FLT	1.0		Weight of the distribution.
`limits`	FLT[4]	$[-1, -1, 1, 1]$		bounding box limits $[q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.
`interpolation`	FLT	2		Interpolation method (nearest, linear, cubic).
`topology`	FLT	0		Topology ($\mathbb{R}^2$, $S^1 \times \mathbb{R}^1$, $\mathbb{R}^1 \times S^1$, and S^2).

Example

```
psd = psd_analytic("abs(cos(2*pi*q)*cos(2*pi*p))
                    *(1-sqrt(q^2+p^2))");
show(psd,file="psd.ppm");
```



`psd.ppm`

psd_ensemble – initialize a PSD from an particle ensemble

Synopsis

PSD psd_ensemble(STR *file*, *keywords*);

Description

Returns a PSD constructed from an particle distribution, read from the file *file*. $file$ is expected to contain phase-space coodinates in ASCII representation in the order $q_1, p_1, \ldots, q_N, p_N$. Additional white-space (apart from that needed to separate the values) is allowed but not required. The distribution is binned into *nslices* bins along the q-axis. The resulting PSD is of the form

$$\Psi(q,p) = \begin{cases} \lambda(q)\,\xi_{\mu(q),\sigma(q)+\Delta\sigma}(p) & |p-\mu(q)| < (\sigma(q)+\Delta\sigma)\,a \\ 0 & \text{else} \end{cases}$$

where $\xi_{\mu,\sigma}$ denotes the one-dimensional normal distribution with mean μ and standard deviation σ. $\lambda(q)$, $\mu(q)$, and $\sigma(q)$ are functions interpolating the density, centroid, and standard deviation in p respectively, where the data points are determined from the binned particles. The distribution is truncated at $(\sigma(q)+\Delta\sigma)\,a$. The value of *spread* is added to the local standard deviation.
If the *type* keyword is set to `"astra"` the $file$ assumed to be a particle distribution file in the format used by ASTRA. The weight of the resulting PSD is set to the total bunch charge.

Keywords

Keyword	Type	Default	Unit	Description
`nslices`	FLT	32		Number of slices.
`cutoff`	FLT	3		a, cutoff in sigma.
`spread`	FLT	0	eV	$\Delta\sigma$, Additional standard deviation
`type`	STR	`"plain"`		Selects file format.
`depth`	FLT	7		Refinement depth of the tree.
`nexp`	FLT	2		$\log_2$(sample points / dimension).
`weight`	FLT	1.0		Weight of the distribution.
`limits`	FLT[4]	$[-1,-1,1,1]$		bounding box limits $[q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.
`interpolation`	FLT	2		Interpolation method (nearest, linear, cubic).
`topology`	FLT	0		Topology ($\mathbb{R}^2$, $S^1\times\mathbb{R}^1$, $\mathbb{R}^1\times S^1$, and S^2).

psd_gauss – initialize a Gaussian PSD

Synopsis

PSD psd_gauss(*keywords*);

Description

Returns the truncated bivariate Gaussian distribution

$$z \mapsto \begin{cases} 0 & \sqrt{r} > a \\ \exp\left(-r/2\right)/(2\pi \;\det \Sigma) & \text{else} \end{cases}$$

with $r = z^T \Sigma^{-1} z$ and the covariance matrix

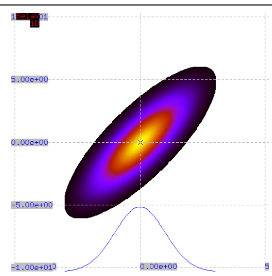
$$\Sigma = \begin{pmatrix} \sigma_q^2 & \rho\sigma_q\sigma_p \\ \rho\sigma_q\sigma_p & \sigma_p^2 \end{pmatrix}.$$

Keywords

Keyword	Type	Default	Unit	Description
sig_q	FLT	0.2	[q]	σ_q, standard deviation in q.
sig_p	FLT	0.2	[p]	σ_p, standard deviation in p.
correlation	FLT	0		ρ, correlation parameter.
cutoff	FLT	3		a, cutoff parameter.
depth	FLT	7		Refinement depth of the tree.
nexp	FLT	2		$\log_2$(sample points / dimension).
weight	FLT	1.0		Weight of the distribution.
limits	FLT[4]	$[-1, -1, 1, 1]$		bounding box limits $[q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.
interpolation	FLT	2		Interpolation method (nearest, linear, cubic).
topology	FLT	0		Topology ($\mathbb{R}^2$, $S^1 \times \mathbb{R}^1$, $\mathbb{R}^1 \times S^1$, and S^2).

Example

```
psd=psd_gauss(correlation=0.8,
              sig_q=1,
              sig_p=2,
              limits=[-5,-10,5,10]);
show(psd,file="psd.ppm");
```



psd.ppm

psd_rectangle – initialize a rectangular PSD

Synopsis

PSD psd_rectangle(FLT[4] *bbox*, *keywords*);

Description

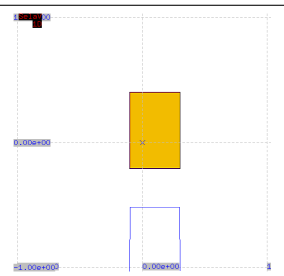
Returns a PSD with rectangular support given by $bbox = [q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.

Keywords

Keyword	Type	Default	Unit	Description
`depth`	FLT	7		Refinement depth of the tree.
`nexp`	FLT	2		$\log_2$(sample points / dimension).
`weight`	FLT	1.0		Weight of the distribution.
`limits`	FLT[4]	$[-1, -1, 1, 1]$		bounding box limits $[q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.
`interpolation`	FLT	2		Interpolation method (nearest, linear, cubic).
`topology`	FLT	0		Topology ($\mathbb{R}^2$, $S^1 \times \mathbb{R}^1$, $\mathbb{R}^1 \times S^1$, and S^2).

Example

```
psd=psd_rectangle([-0.1,-0.2,0.3,0.4]);
show(psd,file="psd.ppm");
```



psd.ppm

psd_test – initialize a test PSD

Synopsis

PSD psd_test(*keywords*);

Description

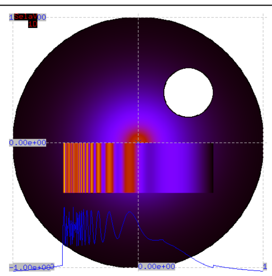
Returns a PSD with clear visual features for testing purposes.

Keywords

Keyword	Type	Default	Unit	Description
`depth`	FLT	7		Refinement depth of the tree.
`nexp`	FLT	2		$\log_2$(sample points / dimension).
`weight`	FLT	1.0		Weight of the distribution.
`limits`	FLT[4]	$[-1, -1, 1, 1]$		bounding box limits $[q_{\min}, p_{\min}, q_{\max}, p_{\max}]$.
`interpolation`	FLT	2		Interpolation method (nearest, linear, cubic).
`topology`	FLT	0		Topology ($\mathbb{R}^2$, $S^1 \times \mathbb{R}^1$, $\mathbb{R}^1 \times S^1$, and S^2).

Example

```
psd=psd_test();
show(psd,file="psd.ppm");
```



`psd.ppm`

save – Save a PSD to a file

Synopsis

save(PSD Ψ, STR *file*);

Description

Save PSD Ψ in a lossless binary format to *file*. A PSD saved in this way can be restored using the `load` function.

Example

```
psd = psd_test();
save(psd,"psd.dat");
```

show – Visualize PSD

Synopsis

show(PSD Ψ);

Description

Start an interactive visualization of the phase-space density Ψ.
If the `file` keyword is supplied, an image in PPM format is written to the specified file. PPM images can be easily converted to more common formats with image manipulation programs such as the ImageMagick suite or the GIMP.

Key	Function
←, ↑, →, ↓	Scroll the window
`o`	Zoom out
`p`	Zoom in
`s`	Save a screenshot to `test.ppm`
`u`	Unzoom
`l`	Toggle cell drawing
`n`	Toggle how to draw negative numbers
`g`	Toggle grid drawing
`r`	Rescale colormap
`q`	Exit
`LMB`	Print value of PSD to stdout
`RMB + drag`	Zoom in to region (click lower left, release upper right)

Keywords

Keyword	Type	Default	Unit	Description
`file`	STR			Write image in PPM format to a file.

strcat – Concatenate strings

Synopsis

FLT strcat(STR a, STR b);

Description

Concatenates two strings a and b to a single string.

Return Value

Concatenated string ab.

Example

```
strcat("conc","atenated");

>>> 'concatenated'
```

strcmp – description

Synopsis

FLT strcmp(STR *a*, STR *b*);

Description

Compares two strings.

Return Value

If the strings are equal 0 is returned. If *a* is less than *b*, a negative value is returned. If *a* is greater than *b*, a positive value is returned.

Example

```
strcmp("test","test");
strcmp("test","tea");

>>> 0.0000000000000000e+00
>>> 1.8000000000000000e+01
```

strtod – Convert string to number

Synopsis

FLT strtod(STR *s*);

Description

Converts the string *s* containing a representation of a floating point number to a floating point number object.

Return Value

Floating point number represented by *s*.

Example

```
strtod("1e-3");

>>> 1.0000000000000000e-03
```

transfer4d – Propagate a 4D transfer matrix

Synopsis

FLT[16] transfer4d(FLT[16] R, FLT l, FLT ϕ_f, FLT k_0, FLT k_1);

Description

Calculates the propagated 4-dimensional transfer matrix

$$\left(\begin{cases} F(\phi_f, k_0) & \text{if } \phi_f \neq 0 \\ B(l, k_0) & \text{else and } k_0 \neq 0 \\ Q(l, k_1) & \text{else and } k_1 \neq 0 \\ D(l) & \text{else and } l \neq 0 \end{cases}\right) \cdot R, \tag{6}$$

where

$$F(\phi_f, k_0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ |\tan(\phi_f)\, k_0| & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7}$$

and

$$B(l, k_0) = \begin{pmatrix} C & S/k_0 & 0 & (1-C)/k_0 \\ -S\, k_0 & C & 0 & S \\ -S & -(1-C)/k_0 & 1 & -(a-S)/k_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

with $a = k0\, l$, $S = \sin(a)$, $C = \cos(a)$

$$Q(l, k_1) = \begin{pmatrix} C & S/|a| & 0 & 0 \\ -|a|\, S & C & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9}$$

with $a = \sqrt{k_1}$, $S = \mathrm{Re}(\sin(a))$, $C = \mathrm{Re}(\cos(a))$

$$D(l) = \begin{pmatrix} 1 & l & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{10}$$

Return Value

Array of size 16 containing the new transfer matrix.

variance – Calculate covariance matrix

Synopsis

FLT[4] variance(PSD Ψ);

Description

Calcuates the covariance matrix of Ψ

$$\int_{\mathbb{R}^2} \Psi(z) \begin{pmatrix} q^2 & q\,p \\ p\,q & p^2 \end{pmatrix} \,\mathrm{d}z. \tag{11}$$

Return Value

Array of size 4 containing the covariance matrix in row-major order.

Example

```
psi = psd_test();
variance(psi);

>>> [4.608020e-02, 4.922162e-03, 4.922162e-03, 3.410189e-02]
```

who – List all defined variables

Synopsis

who();

Description

List all defined variables together with their type and value.

Example

```
mystr = "Hello!";
myflt = 12;
myarr = [1,2,3,5];
mypsd = psd_gauss(); normalize(mypsd);
mymap = map_kickl(2);
who();

>>> myarr = [1.000000e+00, 2.000000e+00, 3.000000e+00, 5.000000e+00];
>>> myflt = 1.2000000000000000e+01;
>>> mymap ;
>>> mypsd =
>>> Center:  0.000000e+00 0.000000e+00
>>> Width:   8.000000e-01 8.000000e-01
>>> Limits:  -8.000000e-01 -8.000000e-01 8.000000e-01 8.000000e-01
>>> Ref.Pt:  0.000000e+00 0.000000e+00
>>> Depth:   7
>>> nexp:    2
>>> Weight:  1.000000e+00
>>> Ipol:    2
>>> Topo::   0
>>> Leafs:   7896
>>> Integrl: 1.000000e+00
>>> ;
>>> mystr = 'Hello!';
```

write_ensemble – Write ensemble to file

Synopsis

write_ensemble(PSD Ψ, STR *file*, FLT *npts*);

Description

Write an ensemble of *npts* points distributed according to Ψ to *file*.

write_grid – Write grid to file

Synopsis

write_grid(PSD Ψ, STR *file*, *keywords*);

Description

Writes values of the PSD along an equidistant grid to *file*.

Keywords

Keyword	Type	Default	Unit	Description
`npts`	FLT[2]	[128,128]		Number of sample points in q and p respectively.
`limits`	FLT[4]	Limits of Ψ		Sampling area $[q_{min}, p_{min}, q_{max}, p_{max}]$.

write_localmoments – Write moments of marginal distribution to file

Synopsis

write_localmoments(PSD Ψ, STR *fname*, FLT *i*);

Description

Writes the local projected density ρ, centroid μ, and variance σ^2 of Ψ along the dimension i to *fname*, where

$$\rho(z_j) = \int_{\mathbb{R}} \Psi(z)\,\mathrm{d}z_i \tag{12}$$

$$\mu(z_j) = \int_{\mathbb{R}} \Psi(z)\,z_i\,\mathrm{d}z_i \tag{13}$$

$$\sigma(z_j) = \int_{\mathbb{R}} \Psi(z)\,(z_i - \mu(z_j))^2\,\mathrm{d}z_i \tag{14}$$

$$\tag{15}$$

and z_j is the remaining phase-space coordinate. Indexing of the dimension starts at 0, so that $z_0 = q$ and $z_1 = p$.

write_projection – Write projection to file

Synopsis

write_projection(PSD Ψ, STR $file$, FLT $axes$);

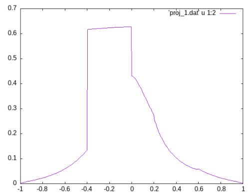
Description

Write the projection along the axes specified by $axes$ to $file$. $axes$ is cast into an integer and interpreted as a bitfield; if the i-th bit is set, the i-th axis will be projected along.

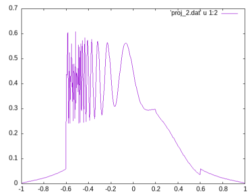
$axes$	Effect
$0 = 00_2$	No projection; the 2D PSD will be written to the file.
$1 = 01_2$	Projection along q.
$2 = 10_2$	Projection along p.
$3 = 11_2$	Projection along p and q. Currently not supported.

Example

```
psd = psd_test();
write_projection(psd,"proj_1.dat",1);
write_projection(psd,"proj_2.dat",2);
```



proj_1.png



proj_2.png