The Plasma Simulation Code (PSC) Documentation

Release 2.99

Kai Germaschewski

Aug 08, 2023

Contents

CHAPTER 1

Installation

1.1 Introduction

PSC is available from github at https:://github.com/psc-code/psc.git . The recommended build method uses Spack package manager. Spack is a package manager that targets HPC use cases. Installing with spack will automate the installation and linking of dependencies. The Legacy build method uses cmake, but the dependencies must be installed manually.

1.2 Dependencies

PSC has the following dependencies:

- (required) cmake is used as the build system
- (required) MPI, the Message Passing Interface library used for running the code in parallel
- (included) googletest for testing
- (required) HDF5, currently still the only real option for regular field / particle output
- (optional) ADIOS2, currently the only option for checkpoint I/O
- (optional) viscid is useful for analyzing / visualizing PSC data
- (optional) rmm, Performance optimization, provides custom cuda allocator to reduce cudaMalloc/cudaFree overhead.
- (optional) nvtx_pmpi MPI hooks so that all MPI calls appear on nsight systems profiler

1.3 Spack Build Instructions

- Clone Spack

```
$ git clone -b releases/v0.15 https://github.com/spack/spack.git
```

- Enable shell support for Spack.

```
# For bash/zsh users
$ export SPACK_ROOT=/path/to/spack
$ . $SPACK_ROOT/share/spack/setup-env.sh
```

- Set up spack for your machine

Generally speaking, `spack` should be configured for your particular system, so that it uses preinstalled system software, like MPI and compilers rather than building everything from scratch.

We do provide a config for Summit, which you can use by doing

```
$ mkdir -p ~/.spack/linux
$ ln -s path/to/psc/spack/configs/summit/*.yaml ~/.spack/linux
```

Note: It'd be nice to add configurations for more systems, including an Ubuntu docker image.

- Add the PSC package repo to Spack

```
$ spack repo add path/to/psc/spack/psc
==> Added repo with namespace 'psc'.
```

- And finally,

```
$ spack install psc
```

1.4 Legacy Build Instructions

Warning: This instructions are likely quite out of date, try to follow the spack approach instead.

1.4.1 Generic Build Instructions

Getting the code from github

PSC is available from github at https:://github.com/psc-code/psc.git. To use it, make a local clone using `git`:

```
[kai@mbpro ~]$ git clone https://github.com/psc-code/psc.git
```

Building

PSC uses `cmake` as a build system. The build happens in a separate build directory, e.g. called `build`:

```
[kai@macbook ~]$ cd psc
[kai@macbook ~]$ mkdir build
[kai@macbook ~]$ cd build
[kai@macbook ~]$ cmake -DCMAKE_BUILD_TYPE=Release ..
[...]
```

Todo: add description of cmake options

Hopefully, the cmake step will succeed without error, at which point we're ready to actually compile the code:

```
[kai@macbook ~]$ make
[...]
```

From now on, after making changes, one should only ever need to rebuild the code using the above `make` command.

Running the tests

The PSC code base includes a bunch of unit tests, though coverage is still far from complete. To run these tests, use `ctest`:

```
[kaig1@login3 build-summit]$ ctest .
[...]

100% tests passed, 0 tests failed out of 233

Total Test time (real) = 120.68 sec
```

1.4.2 Building and Running on Summit

Setting up the environment

I have a little script that I source after logging in:

```
[kaig1@login3 ~]$ cat setup-psc-env-summit.sh
module load cmake/3.14.2
module load gcc/7.4.0
module load hdf5/1.10.3
module load cuda/10.1.168

[kaig1@login3 ~]$ source setup-psc-env-summit.sh
```

It might be even better to load those modules from your `.bashrc`, so they'll be available automatically every time you log in.

Getting the code from github

I keep my codes in the `~/src` directory, so I'm using this in the following example.

PSC is available from github at https://github.com/psc-code/psc.git. To use it, make a local clone using `git`:

```
[kaig1@login3 ~]$ cd src
[kaig1@login3 src]$ git clone https://github.com/psc-code/psc.git
Cloning into 'psc'...
remote: Enumerating objects: 1245, done.
remote: Counting objects: 100% (1245/1245), done.
remote: Compressing objects: 100% (387/387), done.
remote: Total 102767 (delta 885), reused 1111 (delta 838), pack-reused 101522
Receiving objects: 100% (102767/102767), 39.34 MiB | 24.27 MiB/s, done.
```

(continues on next page)

(continued from previous page)

```
Resolving deltas: 100% (84389/84389), done.
Checking out files: 100% (1467/1467), done.
```

Building

PSC uses `cmake` as a build system. The build happens in a separate build directory, e.g. called `build-summit`:

```
[kaig1@login3 src]$ cd psc
[kaig1@login3 psc]$ mkdir build-summit
[kaig1@login3 psc]$ cd build-summit
```

I create another little script file `cmake.sh` in the `build-summit/` directory, so that I know how I invoked `cmake` if I need to do it again in the future:

```
#! /bin/bash

cmake \
   -DCMAKE_BUILD_TYPE=Release \
   -DCMAKE_C_COMPILER=mpicc \
   -DCMAKE_CXX_COMPILER=mpicxx \
   -DADIOS2_ROOT=/ccs/home/kaig1/build/ADIOS2/build-summit/install \
   -DUSE_CUDA=ON \
..
```

Now, invoke cmake:

```
[kaig1@login3 build-summit]$ ./cmake.sh
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
[...]
```

Hopefully, the cmake step will succeed without error, at which point we're ready to actually compile the code:

```
[kaig1@login3 build-summit]$ make
[...]
```

From now on, after making changes, one should only ever need to rebuild the code using the above `make` command.

Note: On Summit, building the tests creates a bunch of annoying warnings since cmake runs those executables to discover the tests, and on Summit, running those executables gives warnings. The following helps to quiet those down somewhat (but don't use this when actually running the code with mpirun).

```
[kaig1@login3 build-summit]$ export OMPI_MCA_btl=tcp,self
```

Running the tests

The PSC code base includes a bunch of unit tests, though coverage is still far from complete. To run these tests, use `ctest`:

```
[kaig1@login3 build-summit]$ ctest .
[...]
```

(continues on next page)

(continued from previous page)

```
100% tests passed, 0 tests failed out of 233

Total Test time (real) = 120.68 sec
```

Running a job

Here is a job script `flatfoil.sh` to run the small sample 2-d flatfoil case on Summit:

```
#! /bin/bash
#BSUB -P AST147
#BSUB -W 00:10
#BSUB -nnodes 1
#BSUB -J flatfoil_summit004

DIR=$PROJWORK/ast147/kaig1/flatfoil-summit004
mkdir -p $DIR
cd $DIR

jsrun -n 4 -a 1 -c 1 -g 1 ~/src/psc/build-summit-gpu/src/psc_flatfoil_yz
```

Submit as usual:

```
[kaig1@login3 build-summit-gpu]$ bsub flatfoil.sh
Job <523811> is submitted to default queue <batch>.
[kaig1@login3 build-summit-gpu]$ bjobs
JOBID   USER       STAT   SLOTS    QUEUE       START_TIME    FINISH_TIME   JOB_NAME
523811  kaig1      PEND       -    batch            -             -        flatfoil_
↪summit004
```

1.4.3 Building and Running on Trillian (at UNH)

Setting up the environment

I have a little script that I source after logging in:

```
[kaig1@login3 ~]$ cat setup-psc-env-trillian.sh
PATH=/home/space/kai/bin:$PATH
module load gcc/7.3.0

[kaig1@login3 ~]$ source setup-psc-env-trillian.sh
```

It might be even better to do these things from your `.bashrc`, so they'll be available automatically every time you log in.

Getting the code from github

I keep my codes in the `~/src` directory, so I'm using this in the following example.

PSC is available from github at https:://github.com/psc-code/psc.git. To use it, first make a fork on github into your own account. (Click the `Fork` button in the upper right corner). You then should make a local clone using `git`:

Todo: Unify fork / clone instructions.

```
[kai@trillian ~]$ cd src
[kai@trillian src]$ git clone git@github.com:germasch/psc.git
Cloning into 'psc'...
```

Building

PSC uses `cmake` as a build system. The build happens in a separate build directory, e.g. called `build-trillian`:

```
[kaig1@login3 src]$ cd psc
[kaig1@login3 psc]$ mkdir build-trillian
[kaig1@login3 psc]$ cd build-trillian

I create another little script file ``cmake.sh`` in the ``build-trillian/`` directory,
↪ so that I know how I invoked ``cmake`` if I need to do it again in the future:
```

```
#! /bin/bash

cmake \
   -DCMAKE_BUILD_TYPE=Release \
   -DCMAKE_CXX_FLAGS="-Wno-undefined-var-template" \
..
```

Now, invoke cmake:

```
[kaig1@trillian build-trillian]$ ./cmake.sh
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
[...]
```

Hopefully, the cmake step will succeed without error, at which point we're ready to actually compile the code:

```
[kaig1@trillian build-trillian]$ make
[...]
```

From now on, after making changes, one should only ever need to rebuild the code using the above `make` command.

Running the tests

Running the tests on trillian is kinda non-trivial, since they are using MPI, so they require to be run with *aprun*, but that only works if you're inside of a batch job.

Running a job

Here is a job script `harris.sh` to run the small sample 2-d flatfoil case on Trillian:

```
#! /bin/bash
#PBS -l nodes=1:ppn=32
#PBS -l walltime=00:10:00

```

(continues on next page)

(continued from previous page)

```
DIR=~/scratch/harris/harris_001
mkdir -p $DIR
cd $DIR

cp ~/src/psc/src/psc_harris_xz.cxx .

aprun -n 4 ~/src/psc/build-trillian/src/psc_harris_xz \
  2>&1 | tee log
```

Submit as usual:

```
[kaig1@trillian build-trillian]$ qsub harris.sh
[kai@trillian build-trillian]$ qstat -u kai

sdb:
                                                            Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
---------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
57623.sdb       kai      workq    run.sh      22952   1  32    --  00:10 R 00:04
```

1.4.4 Installing Viscid

Viscid is a visualization suite that was developed by a graduate student at UNH. It understands PSC HDF5 field data and is the easiest way to read and plot data.

For installation instructions, please look here: https://viscid-hub.github.io/Viscid-docs/docs/dev/installation.html

Note: The stable (`master`) version of `Viscid` gives errors after recent updates to *matplotlib*. Therefore, I had to use the development version of `Viscid` which fixes those issues. Unfortunately, that meant I had to install from source.

CHAPTER 2

PSC Cases

A PSC case is a source file that defines a particular simulation run. This corresponds to what otherwise might be refered to as a parameter file or input deck. It is, however, actual C++ code that needs to be compiled when changes are made to it.

The goal is to have PSC cases to be self-contained, ie., everything related to particular run should be in this one file, including plasma parameters, control parameters, boundary conditions, initial conditions, etc. Having cases written in an actual programming language has the advantage of being fully flexible and powerful. There are drawbacks, though:

- In particular, those parameter files are subject to break as PSC evolves. This is currently likely to happen frequently, as major work is still going on with the code.
- One cannot just change a parameter in the case and run it – the code needs to be recompiled first.

2.1 Changing / adding a case

The eventual goal is for PSC's API to settle down to a stable state. At that point, the idea is that cases can be maintained externally (e.g., in their own repository), and that an easy way to build those against an installed PSC (library) will be provided. However, as the code is currently far from that goal, one should just adapt cases as part of the main repository. The natural starting point is the existing `src/psc_flatfoil_yz.cxx` case. One can either modify that directly, or, if larger changes are to be made copy this file to `src/psc_my_case.cxx` and work with that. To add the new `psc_my_case` to the build, simply add a line `add_psc_executable(psc_my_case)` to `src/CMakeLists.txt`.

2.2 Anatomy of a case

src/psc_flatfoil_yz.cxx has been commented to help understand what goes into a case.

Todo: There really should be a description here, too…

CHAPTER 3

Analyzing and Visualizing PSC data using Python

3.1 Basics of Analyzing PSC Data Using Python/Jupyter/Viscid

3.1.1 Tools

One way to analyze the data used by PSC is using Python, Jupyter and Viscid. Python is hopefully well known. Project Jupyter let's you interact with Python (and other languages) through your web browser. With some setup, it is possible to run Jupyter remotely and interact with it through a local web broswer, which has the advantage that the data can remain in the remote location, only plots / movies are sent across the network.

Viscid is Python viz library developed by a graduate student at UNH. It is built on top of **matplotlib** and helps to seamlessly read various data formats.

3.1.2 An example

Let's presume you've run the small 2-d flatfoil sample case. The basics of making some plots based on the data generated are shown below.

Importing modules

The below imports the modules needed for reading the PSC data and making some plots. Other than **viscid**, they are pretty standard.

[1]:
```
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import viscid
viscid.calculator.evaluator.enabled = True
from viscid.plot import vpyplot as vlt
```

(continues on next page)

(continued from previous page)

```
%config InlineBackend.figure_format = 'retina'
```

Open the data files

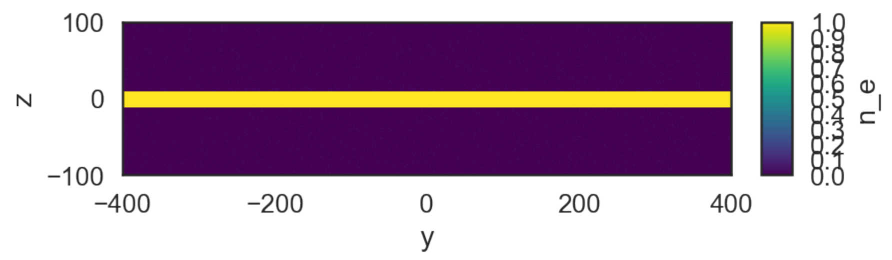
The path to the data will vary for your own run. You usually want to open the top-level `pfd.xdmf` file that encompasses all the individual output steps.

```
[2]: run = "/Users/kai/src/pppl-project/flatfoil/flatfoil_702/pfd.xdmf"
     vf = viscid.load_file(run, force_reload=True)
```

Make a plot

Select a particular output step (they are simply numbered consecutively). You can then plot a particular field by name.

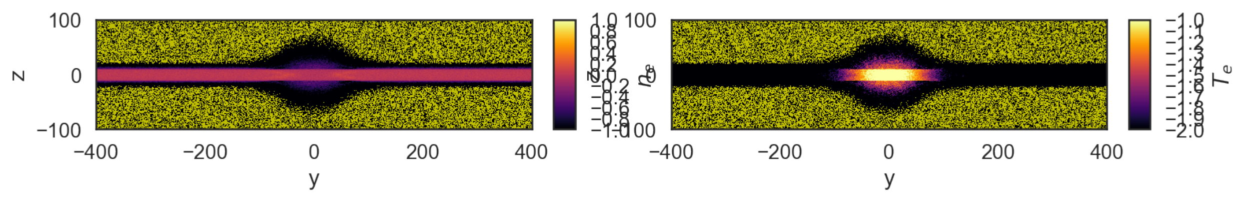
```
[3]: vf.activate_time(0)
     vlt.plot(vf["n_e"]);
```



Make another plot

Here, I select a later output and plot two quantities, which shows how some simple computations can be done.

```
[4]: vf.activate_time(10)
     plt.figure(figsize=(12, 6))
     plt.subplot(1, 2, 1)
     vlt.plot(vf["$n_e$=log10(n_e)"], clim=(-1,1), cmap="inferno");
     plt.subplot(1, 2, 2)
     vlt.plot(vf["$T_e$=log10(Txx_e+Tyy_e+Tzz_e)"], clim=(-2,-1), cmap="inferno");
```



```
[ ]:
```

3.2 Reading and plotting of PSC particle data

This notebook demonstrates reading of PSC particle data and some simple analysis that can be done using Python / pandas / matplotlib.

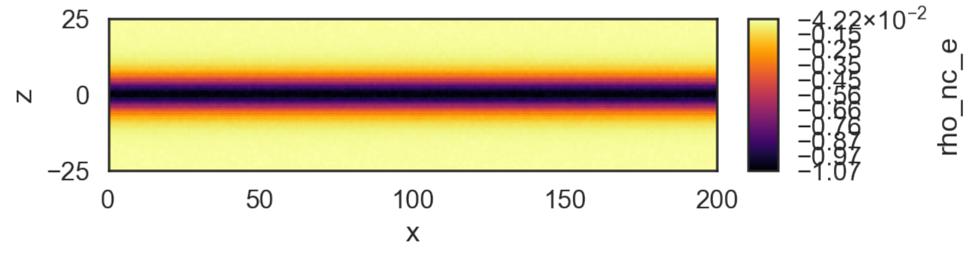
First usual set up, import modules, etc

```
[1]: %matplotlib inline

     import matplotlib as mpl
     import matplotlib.pyplot as plt
     import numpy as np
     import viscid
     from viscid.plot import vpyplot as vlt
     import h5py
     import pandas as pd
     viscid.calculator.evaluator.enabled = True

     %config InlineBackend.figure_format = 'retina'
```

```
[2]: project_dir = "/Users/kai/src/pppl-project/flatfoil/harris_700"
```

3.2.1 Make 2-d plot first of density moment first

```
[3]: f = viscid.load_file(project_dir + "/pfd.xdmf", force_reload=True)
     f.activate_time(0)
     vlt.plot(f["rho_nc_e"], cmap="inferno");
```



3.2.2 ParticleReader

This class really should be provided by PSC in a module (or as part of viscid). It also still needs work, in particular:

- Support specifying regions by actual coordinates, rather than grid points
- Handle the case where only a subset of particles were written

```
[4]: class ParticleReader(object):
         def __init__(self, filename):
             file = h5py.File(filename, "r")
             self._particles = file["particles/p0/1d"]
             self._idx_begin = file["particles/p0/idx_begin"]
             self._idx_end = file["particles/p0/idx_end"]
             self.selection_lo = file["particles"].attrs["lo"]
             self.selection_hi = file["particles"].attrs["hi"]
```

(continues on next page)

(continued from previous page)

```
        self.gdims = np.array(self._idx_begin.shape[3:0:-1])
        self.n_kinds = self._idx_begin.shape[0]

    def __xinit__(self, particles, idx_begin, idx_end, selection_lo, selection_hi):
        self._particles = particles
        self._idx_begin = idx_begin
        self._idx_end = idx_end
        self.selection_lo = selection_lo
        self.selection_hi = selection_hi
        self.gdims = np.array(idx_begin.shape[3:0:-1])
        self.n_kinds = idx_begin.shape[0]

    def size(self):
        return self.particles.shape[0]

    def inCell(self, kind, idx):
        loc = (kind, idx[2] - selection_lo[2], idx[1] - selection_lo[1], idx[0] -␣
↪selection_lo[0])
        return slice(self._idx_begin[loc], self._idx_end[loc])

    def __getitem__(self, idx):
        if isinstance(idx, slice) or isinstance(idx, int):
            return self._particles[idx]
        elif isinstance(idx, tuple):
            return self._particleSelection(idx)
        else:
            raise "don't know how to handle this index"

    def _particleSelection(self, idx):
        indices = self._indices(idx)
        n = np.sum(indices[:,1] - indices[:,0])
        prts = np.zeros(n, dtype=self._particles[0].dtype)
        cnt = 0
        for r in indices:
            l = int(r[1] - r[0])
            prts[cnt:cnt+l] = self._particles[r[0]:r[1]]
            cnt += l
        return pd.DataFrame(prts)

    def _indices(self, idx_):
        # reorder index
        idx = (idx_[0], idx_[3], idx_[2], idx_[1])
        begin = self._idx_begin[idx].flatten()
        end = self._idx_end[idx].flatten()
        indices = [sl for sl in zip(begin, end)]
        indices = np.sort(indices, axis=0)
        # this index range could be merged where contiguous, or maybe even where
        # not contiguous but close
        return indices

    def __repr__(self):
        return ("ParticleReader(n={} gdims={} n_kinds={} selection=({}, {})"
                .format(self.size(), self.gdims, self.n_kinds, self.selection_lo,␣
↪self.selection_hi))
```

3.2.3 Read particles

In this particular case, read all species (`:`), and in cells `128 <= i < 130`, `j = 0`, `k = 32`, which is the center of the domain.

Particles are returned as a pandas data frame, so they can be displayed easily as a table, selections can be made, and simple plotting is provided, too.

```
[5]: p = ParticleReader(project_dir + "/prt.000000_p000000.h5")
     df = p[:, 128:130, 0, 32]
     df.head()
```

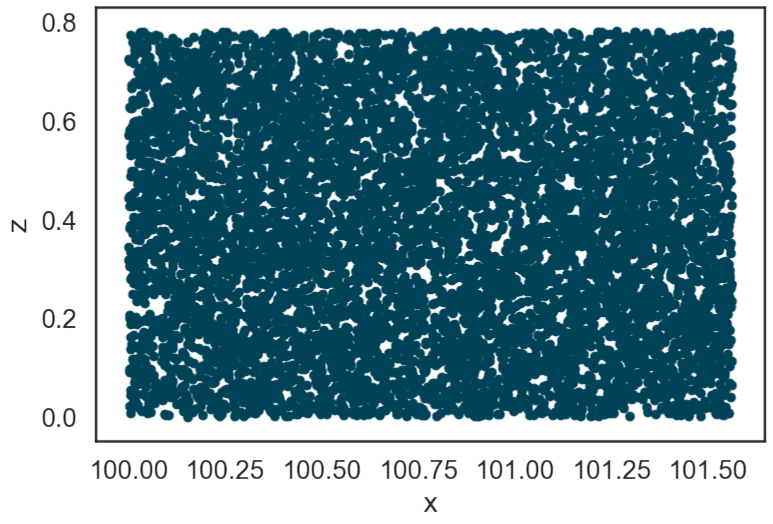
```
[5]:             x         y         z        px        py        pz    q    m  \
     0  100.209442  1.414471  0.547419 -0.191176 -0.046516  0.165849 -1.0  1.0
     1  100.135612 -1.425988  0.714132  0.094603 -0.079601 -0.259543 -1.0  1.0
     2  100.383514 -1.144820  0.228649 -0.101998 -0.148720  0.001895 -1.0  1.0
     3  100.745880  2.161036  0.686588  0.096610 -0.129796 -0.142294 -1.0  1.0
     4  100.340401 -1.642766  0.198549  0.118109  0.129963  0.145169 -1.0  1.0

              w
     0  0.00025
     1  0.00025
     2  0.00025
     3  0.00025
     4  0.00025
```

Selections

A simple example of making a selection: `electrons` are those particles with negative charge.

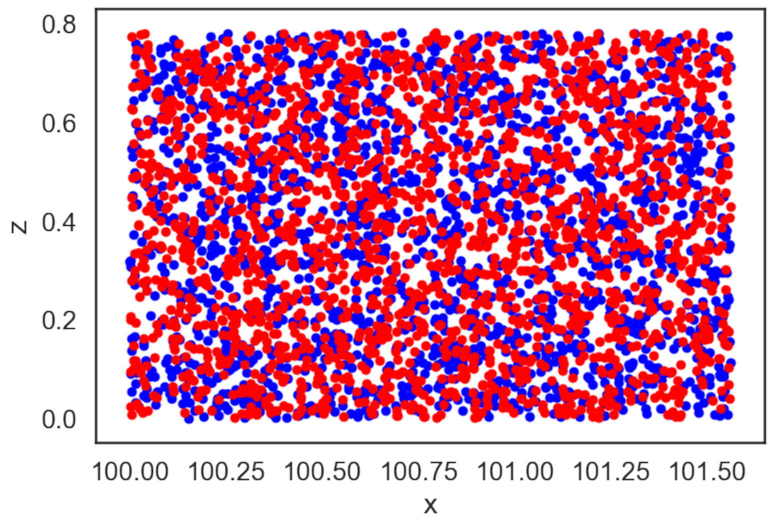
```
[6]: electrons = df[df.q < 0]
```

```
[7]: electrons.plot.scatter('x', 'z');
```



A bit more complex if contrived, example: Plot electrons moving quickly right (red) and quickly left (blue).

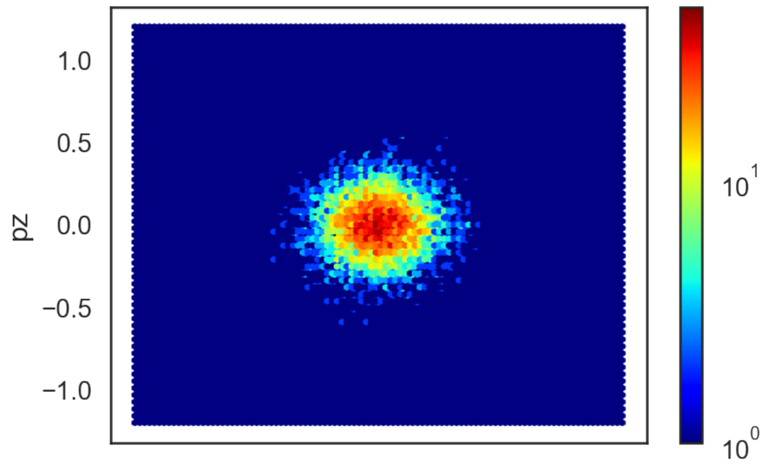
```
[8]: electrons[electrons.px < -.1].plot.scatter('x', 'z', color='b', ax=plt.gca())
     electrons[electrons.px > .1].plot.scatter('x', 'z', color='r', ax=plt.gca());
```



3.2.4 Plot a distribution function

Pandas takes care of the binning, so that's simple, too.

```
[9]: kwargs = {"gridsize": 100, "extent": (-1.2, 1.2, -1.2, 1.2), "cmap": 'jet'}
electrons.plot.hexbin(x='px', y='pz', bins="log", **kwargs);
```



Todo: Explain output quantities / format / list available

Todo: Particles, tracing, etc

Todo: put links to the actual jupyter notebooks used in docs

CHAPTER 4

Visualization using Paraview

The default output format for PSC is HDF5 augmented with XDMF descriptor files, so output data can be directly loaded and visualized in Paraview.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search