QSL Squasher Documentation

Release 1.1

Svetlin Tassev

Jun 26, 2017

CONTENTS

Author Svetlin Tassev

Version 1.1

Date June 26, 2017

Homepage QSL Squasher Homepage

Documentation PDF Documentation

License GPLv3+ License

CHAPTER ONE

OVERVIEW

QSL Squasher is an OpenCL code for calculating the squashing factor, Q, of a vector field specified within a finite volume. Its description below focuses on its application to solar magnetic fields, but the code itself is completely general.

QSL Squasher is based on the following paper: *[QSL3d]*. We kindly ask you[1] to acknowledge it and its authors in any program or publication in which you use QSL Squasher or a derivative of it.

1.1 Compiling

QSL Squasher requires Boost, VexCL, an OpenCL implementation, as well as their respective dependencies. The visualization scripts require Python with SciPy and PyEVTK. The resulting 3d data cubes are exported to VTK format, which can then be visualized using Paraview, VisIt or Mayavi among many.

The code has been mostly tested on an Arch Linux server with an AMD FirePro W8100 GPU, and it has been successfully run on laptops with subpar hardware (the example included in this documentation was run on such a laptop). The code is memory hungry when performing large refinements in 3D, so we use a swap of 256GB on an SSD on the server. As a reference, the following relevant packages were installed on the server, which may or may not be required depending on your particular hardware configuration:

Package	Arch Linux Version
amd-adl-sdk	6.0-1
amdapp-aparapi	20130123-1
amdapp-codexl	1.6-7247
amdapp-sdk	2.9.1-1
amdapp-sdk-aparapi	2.9.1-1
amdapp-sdk-opencv	2.9.1-1
boost-compute-git	0.4-2
catalyst-firepro	14.502.1040-1
clang	3.6.2-2
intel-opencl-sdk	2014_R2-2
linux	4.1.5-1
pocl	0.11-1
vexcl-git	20150710-4
xorg-server	1.16.4-1

A compile script, `compile.sh`, is included with the source code which compiles the two main programs: the main calculation code, `qslSquasher.cpp`, as well the post-processing code `snapshot.cpp`. You need to edit the

[1] We cannot *require* you, however, as we want QSL Squasher to be GPLv3+ compatible.

script by hand to make it consistent with your configuration, especially since different OpenCL implementations can co-exist on the same hardware on different paths.

As an example, for the AMD GPU on our dedicated server, `qslSquasher.cpp` is compiled with:

```
$ clang -I/opt/AMDAPP/SDK/include qslSquasher.cpp -std=c++11 \
> -lstdc++ -lm -I/usr/local/include/vexcl -lOpenCL \
> -lboost_system -O3 -march=native -mcpu=native -o qslSquasher
```

If you want to test the code on your CPU, you would need the POCL OpenCL implementation to be installed on your computer. Then you need to define `OpenCL_DEVICE_TYPE` as `CL_DEVICE_TYPE_CPU` in `options.hpp`. You can then compile the code with the following command (note that some paths may need to be adjusted to your configuration):

```
$ clang -I/opt/intel/opencl-sdk/include qslSquasher.cpp \
> -std=c++11 -lstdc++ -lm -I/usr/local/include/vexcl \
> -lOpenCL -lboost_system -O3 -march=native -mcpu=native \
> -o qslSquasher
```

Warning: Some of the options for the OpenCL kernels need to be specified at compile time. Therefore, you need to recompile the code every time you change the hard-coded options in `options.hpp`. For instance, by default, the code runs on the CPU, not on the GPU. To run on the GPU, you need to set `OpenCL_DEVICE_TYPE` to `CL_DEVICE_TYPE_GPU`.

Warning: Make sure you optimize the `CHUNKSIZE` in `options.hpp` before using this code for production purposes. If `CHUNKSIZE` is set too high, you may run out of GPU memory, and get curious error messages…

1.2 Input

The code is configured by adjusting the hard-coded values in `options.hpp`. Those are described in detail here.

The code takes as input 6 ASCII files with the following naming conventions:

```
in_dir+'bx0'+in_filename+'.dat'
in_dir+'by0'+in_filename+'.dat'
in_dir+'bz0'+in_filename+'.dat'

in_dir+'xs0'+in_filename+'.dat'
in_dir+'ys0'+in_filename+'.dat'
in_dir+'zs0'+in_filename+'.dat'
```

The files `b(x|y|z)*.dat` contain the 3d arrays for the 3 components of the magnetic field as a flattened list of numbers. The units of the magnetic field can be arbitrary as only the tangent unit vectors are used to calculate the Q values. The 3d arrays are of dimensions (`NX, NY, NZ`) and are read inside the following nested for loops:

```
for (size_t k = 0; k < NZ; ++k)
        for (size_t j= 0; j < NY; ++j)
                for (size_t i = 0; i < NX; ++i)
```

So, take this ordering into account when writing inputs for this code.

The dimensions (`NX, NY, NZ`) need to be set in `options.hpp` at compile time for the OpenCL kernels, which means that you need to recompile the code for each new box.

The code assumes that the magnetic field components are sampled on a rectilinear grid in either spherical or cartesian coordinates. The grid point coordinates are specified by the files `xs*.dat`, `ys*.dat`, `zs*.dat`. Those samples should be in increasing order.

Depending on whether `GEOMETRY` is set to `CARTESIAN` or `SPHERICAL`, the magnetic field and the grid point coordinates are given as follows:

For the `CARTESIAN` setting, the files `b(x|y|z)*.dat` contain the components of the magnetic field in the usual orthonormal $\hat{x},\ \hat{y},\ \hat{z}$ cartesian basis. For the `SPHERICAL` setting those files contain the magnetic field components in the orthonormal spherical basis $\hat{\phi},\ \hat{\theta},\ \hat{r}$ (i.e. longitude, latitude, radius). Therefore, in that case `bx*.dat` contains the magnetic field at each grid point in the longitudinal direction, `by*.dat` gives the latitudinal component, and `bz*.dat` – the radial.

For the `CARTESIAN` setting, the grid coordinate files `xs*.dat`, `ys*.dat`, `zs*.dat` contain `NX, NY, NZ` numbers specifying the respective $x,\ y,\ z$ coordinates of the grid points in Mm.

For the `SPHERICAL` setting, `xs*.dat` contains `NX` numbers specifying the longitudes of the grid points in *degrees*, while the file `ys*.dat` contains `NY` numbers specifying the latitudes of the grid points in *degrees*. The file `zs*.dat` contains `NZ` numbers specifying the radial coordinates of the grid points in units of *solar radii*. In other words, the photosphere of the sun should be at $r = 1$ in this file when using spherical coordinates.

1.3 Output

1.3.1 Output from qslSquasher

The code calculates the squashing Q values for the input magnetic field on either a 2d slice or a 3d cube, depending on whether `QSL_DIM` is set to `2` or `3`, respectively.

Progress and debugging information is output to `stderr`, while the calculation results are output to `stdout` after the Q values are calculated for the initial grid, and then after each successive mesh refinement.

In the code, the slice/cube for which the Q values are calculated is indexed with a Hilbert curve that fills the region of interest. The output from the `qslSquasher.cpp` code is printed to stdout in five columns:

- The first column corresponds to the Hilbert key of the point for which the Q value is calculated. This key is used by the next post-processing step described below.
- The second, third and fourth columns give the coordinates of the grid point for which the Q value is calculated. For cartesian coordinates, those correspond to the $\hat{x}, \hat{y}, \hat{z}$ coordinates in Mm; while for spherical coordinates, those are the $\hat{\phi}, \hat{\theta}, \hat{r}$ coordinates in units of deprees, degrees and solar radii, respectively.
- The fifth columns returns the Q value for that grid point.

The output after the initial calculation on a grid and after each mesh refinement is sorted according to Hilbert key values. For multiple refinements, the output can easily reach more than a billion Q values sampled on an irregular grid. Thus, for convenience, we provide a series of post-processing routines, which allow for easier vizualization of the results. The post-processing is performed by `snapshot.cpp` and the Python visualization scripts described below.

1.3.2 First post-processing step with `snapshot.cpp`

This code assumes that the output from `qslSquasher` is saved in the current directory as `raw.dat`. Then, `snapshot` parses that file and returns to `stdout` a list of $log_{10}(Q)$ values on a rectilinear grid spanning the 2d/3d region of interest for which the Q values were calculated in `qslSquasher`. The grid is of size `nx_out`, `ny_out` (and `nz_out` when working with a 3d cube), which are specified at the top of `snapshot.cpp` at compile time.

The output is a column of $log_{10}(Q)$ values printed by the following nested for-loops:

```
for (size_t i = 0; i < nx_out; ++i)
        for (size_t j= 0; j < ny_out; ++j)
                for (size_t k = 0; k < nz_out; ++k) # for 3d cube
```

If several Q values are found within a cell of the grid (as defined by the neighborhood of the point along the Hilbert curve), then the code takes the maximum of those. Otherwise, the values are interpolated along the Hilbert curve filling the cube/slice.

The definition of Q is such that $log_{10}(Q) \geq log_{10}(2)$. Junk values are returned as *-1000*. We recommend that you parse the output of `snapshot` with the Python scripts described in the next section.

Note that if `CALCULATE` is set to `FIELD_LINE_LENGTH` instead of `QSL`, then the output from this post-processing step contains the values of the length of the fields line passing through each sampled point, and not the values of $log_{10}(Q)$.

1.3.3 Second post-processing step with Python

The scripts `viz2d.py` and `viz3d.py` show examples of post-processing the 2d/3d output from `snapshot`. The 2d post-processing script, `viz2d.py`, outputs a png image file containing the slice produced by qslSquasher.

The 3d post-processing script, `viz2d.py`, outputs two VTK files containing the $log_{10}(Q)$ and magnetic field values in the 3d cube sampled by qslSquasher. Those VTK files can then be visualized by ParaView as in the *example* included with this documentation.

Note that if `CALCULATE` is set to `FIELD_LINE_LENGTH` instead of `QSL`, then the output from this post-processing step contains the gradient magnitude from the Sobel operator applied to the field-line length map.

CHAPTER

TWO

OPTIONS

The options for `qslSquasher` below are configurable in `options.hpp`. This means that the code needs to be recompiled after each modification to the options below.

std::string **`in_dir`**
The directory name for the input files. See Input.

std::string **`in_filename`**
The filename suffix for the input files. See Input.

`N`(X|Y|Z)
The size of the input arrays in the x/longitudinal (`NX`), y/latitudinal(`NY`) and z/radial (`NZ`) direction.

`GEOMETRY`
Pick one type of geometry for your input box. Possible values are: `SPHERICAL` (default) or `CARTESIAN`. When using spherical geometry, the poles, as well as the periodicity in longitude, are treated correctly.

`GLOBAL_MODEL`
If defined (only for `SPHERICAL` geometry), the code assumes that the input magnetic field covers the whole sun. Longitude samples should start at >=0 degrees, and end at <360 degrees. Latitude samples should start >-90 degrees, and end at <90 degrees. The code currently supports only trilinear interpolation when this options is set. The poles, as well as the periodicity in longitude, are treated correctly.

`SOLAR_RADIUS`
The solar radius in Mm (default: `696.`). Needed only for spherical geometry.

`QSL_DIM`
Tells the code whether you want a 2d slice (`QSL_DIM=2`) of Q values, or a 3d cube (`QSL_DIM=3`)?

If `QSL_DIM=2`, then one needs to specify the parameters controlling the size, location and orientation of the slice one wants computed. Here is the set of relevant options that need to be set:

`SLICE_TYPE`
Specifies the type of slice. For cartesian geometry, the only available option is `CARTESIAN`. For spherical geometry, one can pick a `CARTESIAN` or a `SPHERICAL` slice. When set to `CARTESIAN`, the slice is an intersection the volume with a plane of position, size and orientation specified by the options below. When set to `SPHERICAL` (default), the slice is a curved 2d surface at fixed radius.

`double SLICE_NORMAL[]`
Vector normal to slice. Need not be normalized. Used only for cartesian slices.

`double SLICE_UP[]`
`SLICE_UP` gives the general "up" direction along the slice. We take only the component of `SLICE_UP` that lies in the plane of the slice to construct the `y` direction in the plane of the slice. So, need not be orthonormal to `SLICE_NORMAL`. Note that the `x` direction in the plane of the slice is given by the cross product `SLICE_UP` $\times$ `SLICE_NORMAL`. So, be careful with the overall sign of `SLICE_UP`, or you may end up with a flipped image. Used only for cartesian slices.

double SLICE_CENTER[]
: SLICE_CENTER gives the coordinates of the center of the slice. The coordinates are in units of (Mm, Mm, Mm) for cartesian geometry, or in units of (degrees, degrees, Mm above the photosphere) for spherical geometry. The slice will pass through this point.

double **SLICE_L**(X|Y)
: SLICE_LX and SLICE_LY give the size of the slice in Mm for cartesian slices. For spherical slices, the units are in degrees.

ZMIN
: ZMIN forces field lines to be terminated at that height above the photosphere/bottom of the box for spherical/cartesian coordinates. This is useful for eliminating photospheric "noise".

If QSL_DIM=3, then one needs to specify the size and location of the 3d cube for which the Q values are to be computed. Here is the set of relevant options that need to be set:

(X|Y|Z)(MIN|MAX)
: These six parameters give the boundaries of the cube for the 3d Q calculation. For cartesian geometry, all are in Mm. For spherical geometry, the X and Y limits are set in degrees along the longitudinal and latitudinal directions, respectively. In that case, ZMIN and ZMAX are measured in Mm above the photosphere. ZMIN also forces the calculation of the field lines to terminate at that height above the solar photosphere/bottom of the input box for spherical/cartesian geometries. This is useful for eliminating photospheric "noise". Note that apart from the ZMIN limit, the field lines are followed to the boundaries of the data cube spanned by the *input* files.

z_sampler(z)
: A function specifying how to sample the 3d cube in the radial/z direction for spherical/cartesian geometries. Its argument is assumed normalized between 0 (corresponding to bottom index of the cube) and 1 (corresponding to top index of the cube). Its output must span the physical size of the box in Mm, i.e. it should run between ZMIN and ZMAX.

CALCULATE
: The code calculates the squashing factor values when CALCULATE is set to QSL (default). When set to FIELD_LINE_LENGTH, it calculates the length of the field lines passing through each sampled point. The code does not do refinements in the latter case, as those are unnecessary for the field-line length map (as long as the initial grid sampling is fine enough to resolve the connectivity domains of interest). When calculating field-line lengths, the code reuses the same infrastructure as when calculating the squashing factor values. Thus, one has to go through the same post-processing pipeline, irrespective of the option set by CALCULATE.

n(x|y|z)_init
: The size of the initial grid (before mesh refinement) for which the Q values are to be computed. nz_init is not needed if QSL_DIM=2.

OpenCL_DEVICE_TYPE
: Tells VexCL whether to use the CPU when defined as CL_DEVICE_TYPE_CPU (default), or the GPU when defined as CL_DEVICE_TYPE_GPU.

NGPU
: NGPU (default: 0) tells VexCL on which GPU you want to do the computation. In case you want to specify the GPU in other ways, consider changing the GPU filter specified by the following line in qslSquasher.cpp:

```
vex::Context ctx(   vex::Filter::Type(OpenCL_DEVICE_TYPE)
                 && vex::Filter::Position(NGPU) );
```

const size_t **CHUNKSIZE**
: The CHUNKSIZE sets how many Q value calculations are to be dispatched to the GPU in one go. Set CHUNKSIZE too high and you'll run out of GPU memory. Set it too low, and you'll find performance being degraded. The proper value will depend mostly on your hardware and on your choice for integration sheme,

so experiment until you find the sweet spot for your configuration. The default value (2^{19}) is optimized for the `EULER` scheme on AMD FirePro W8100, which has 8GB memory.

INTERPOLATION_TYPE
Pick one interpolation algorithm used for interpolating the B-field values. Possible values are: `TRILINEAR` (default), `TRIQUADRATIC`, `TRICUBIC`.

LENGTH_JUMP_REFINEMENT_THRESHOLD
Specifies the threshold (default: 1Mm) for the change in field-line length between two neighbouring points on the Hilbert curve. If that threshold is exceeded, then the code makes a refinement by sampling the point lying half-way on the Hilbert curve between those two points.

MAX_REFINEMENTS
Specifies the maximum number of refinements the code will make before exiting.

INTEGRATION_SCHEME
Specifies the integration scheme. One can set this to `EULER` (default) for an explicit Euler scheme, or to `ADAPTIVE` for adaptive stepping. The default adaptive stepper is Boost's 5-th order runge_kutta_cash_karp54. You can always experiment with others by changing the corresponding line in `qslSquasher.cpp`.

eps_rel, eps_abs, **DISPLACEMENT_WEIGHT**
Have an effect only when one uses the `ADAPTIVE` integration scheme. The first two specify the relative and absolute error (defaults: `1e-2`) for the adaptive stepper. Those bounds are both for the field line positions, as well as for the perturbations to the field lines that are needed for the squashing factor calculation. The `DISPLACEMENT_WEIGHT` (default: 10) boosts the weight of those perturbations, since their errors will otherwise be swamped by the errors in the positions.

LOCAL_Q
By default, the Q value of a field line is obtained by calculating the squashing factor between the two ends of a field line. An end of a field line is considered the point where the field line intersects the surface of the *input* b-field box, or where it hits a null.

However, you can calculate a more localized value of Q by measuring the squashing factor over a specified length (in Mm) up and down each field line. To do that, uncomment the `LOCAL_Q` line in `options.hpp`. You'd need to specify the length over which you want the local Q to be calculated. That is given by `INTEGRATION_RANGE` in Mm.

INTEGRATION_RANGE
If `LOCAL_Q` is not defined, then the global Q values are computed. In that case, field line integration is done in chunks until the field line terminates at the box boundaries, or $\boldsymbol{B}$ gets very close to zero (e.g. near nulls). The length of each chunk is specified by `INTEGRATION_RANGE`. After each chunk, the field lines are checked for whether they have terminated. If left undefined, a sane value for `INTEGRATION_RANGE` is picked in `qslSquasher.cpp`.

INTEGRATION_STEPS_PER_CELL
`INTEGRATION_STEPS_PER_CELL` is used to calculate the step size for the field line integrators. The step size is such that there are roughly `INTEGRATION_STEPS_PER_CELL` steps in each cell in the input grids. The resulting step size (printed to `stderr`) is the integration step for the Euler integration scheme, or is the initial step for the adaptive stepper. If left undefined, sane defaults are set in `qslSquasher.cpp`.

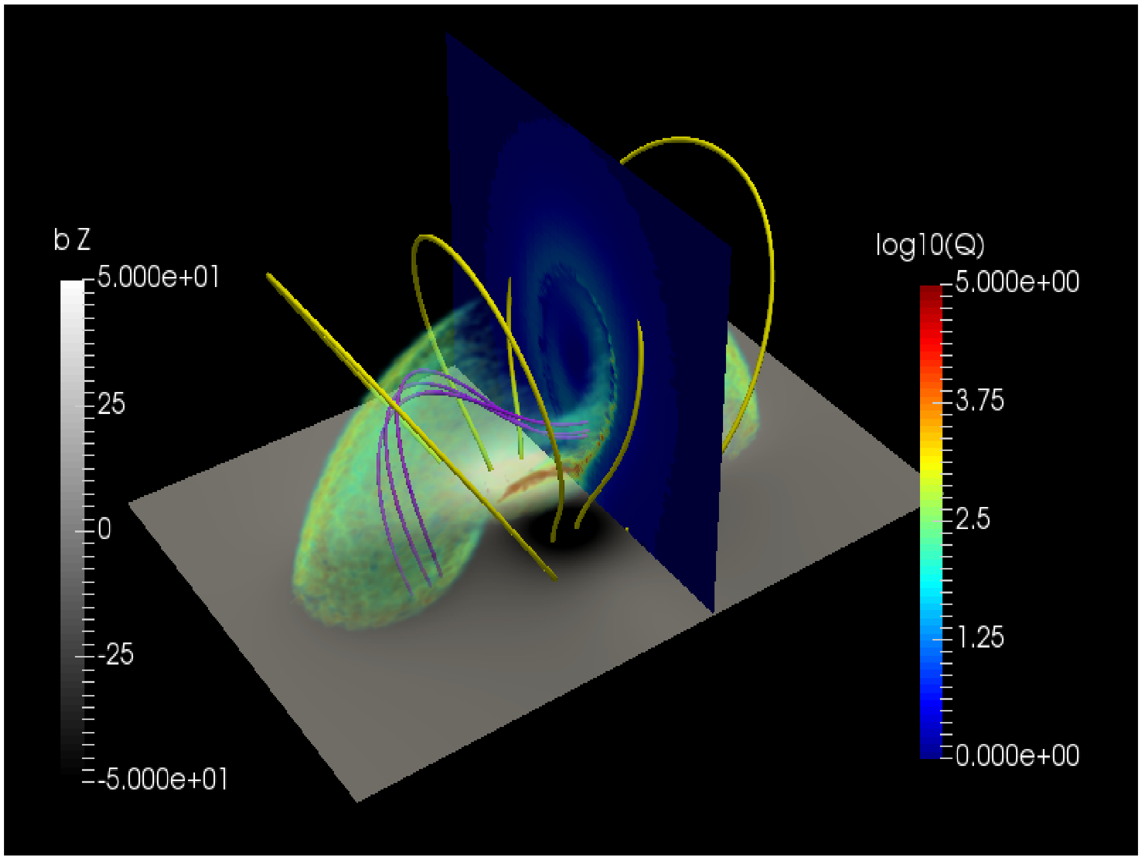
MARK_OPEN_FIELD_LINES
When `MARK_OPEN_FIELD_LINES` is defined (default), then the code calculates Q values only for field lines which begin and end at the bottom surface of the volume, corresponding to z or height above the photosphere equal to *ZMIN* for cartesian or spherical geometry, respectively. Open field lines are marked with the generic value of *-1000*, which is used for any junk values encountered by the code. If this keyword is left undefined, then the code calculates the Q value for all points in the volume, irrespective of whether they belong to open field lines or not. For open field lines, the Q value is calculated between the two endpoints of the field lines, independent of whether those occur at the bottom boundary or not.

CHAPTER

THREE

WORKED-OUT EXAMPLE

The default options in the code generate a *very low-resolution* 3d cube of Q values with the sample dataset included with the code. Running the code with those options using the commands listed below, generates the following 3d view in `ParaView`, which includes a magnetogram, volumetric rendering of Q values, as well as traced field lines:



The command lines below show a typical sequence of commands to generate a 3d data cube of Q values, and then visualize the result. The example uses the data files distributed with the code and was run on the *CPU* of a low-end laptop. The calculation of about half a million Q values, post-processing and rendering took less than 5 minutes with the default settings. Note that in the session below, qslSquasher was killed after two mesh refinements.

```
$ time ./compile.sh

  real   0m28.613s
  user   0m27.370s
  sys   0m0.417s
$ time ./qslSquasher > raw.dat
  1. Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz (Intel(R) OpenCL)

  Reading successful.
  Integration step set at 0.022446 Mm
  Initialization successful.
  Number of field lines to be integrated in this mesh refinement step: 262144
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 262144 in mesh refinement: 0
  # of computed field lines = 103793 out of 262144 in mesh refinement: 0
         ... skipping lines ...
  # of computed field lines = 262118 out of 262144 in mesh refinement: 0
  Beginning BACKWARD integration along field lines ...
  # of computed field lines = 0 out of 262144 in mesh refinement: 0
         ... skipping lines ...
  # of computed field lines = 262118 out of 262144 in mesh refinement: 0
  Q values calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 34352
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 34352 in mesh refinement: 1
         ... skipping lines ...
  # of computed field lines = 34250 out of 34352 in mesh refinement: 1
  Q values calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 34609
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 34609 in mesh refinement: 2
         ... skipping lines ...
  # of computed field lines = 34597 out of 34609 in mesh refinement: 2
  Q values calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 41122
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 41122 in mesh refinement: 3
         ... skipping lines ...
  # of computed field lines = 41018 out of 41122 in mesh refinement: 3
  Q values calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 48880
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 48880 in mesh refinement: 4
         ... skipping lines ...
  # of computed field lines = 48866 out of 48880 in mesh refinement: 4
  Q values calculated successfully.
  ^C
  real  2m23.511s
  user  8m17.523s
  sys   0m25.037s
$ time ./snapshot > grid3d.dat

  real  0m1.999s
  user  0m1.937s
  sys   0m0.060s
$ time python2 viz3d.py

  real  0m7.020s
  user  0m6.450s
  sys   0m0.317s
$ paraview --state=viz3d_paraview.pvsm
```

The example above is for input in cartesian coordinates. It generates several output files:

- `raw.dat` contains the raw output from `qslSquasher.cpp`.
- `grid3d.dat` is the result of the first post-processing step done by `snapshot.cpp`.
- `SquashingFactor_CartCoo.vtr` is a VTK file, containing the rectilinear grid of Q values in cartesian coordinates.
- `MagneticField_CartCoo.vtr` is a VTK file, containing the rectilinear grid of magnetic field component values in cartesian coordinates. This file is generated from the input files used by `qslSquasher`.

The last two files are used by the included `ParaView session file` to generate the figure shown in the beginning of this section.

BIBLIOGRAPHY

[QSL3d] ‘ QSL Squasher: A Fast Quasi-Separatrix Layer Map Calculator‘, S. Tassev and A. Savcheva (2016), arXiv:1609.00724

INDEX