QSL Squasher – Transverse Version Documentation

Release 2.0

Svetlin Tassev

Jan 08, 2019

CONTENTS

Author	Svetlin Tassev
Version	2.0
Date	Jan 1, 2019
Homepage	QSL Squasher – Transverse Version Homepage
Documentation	PDF Documentation
License	GPLv3+ License

CHAPTER

ONE

OVERVIEW

QSL Squasher is an OpenCL code for calculating the squashing (Q) and squeezing (Z) factors as well as twist and coiling numbers 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.

Note that this is the manual for the QSL Squasher – Transverse Version (corresponding to QSL Squasher version >=2.0). This version of the code is restricted to Euler integration scheme and trilinear interpolation in calculating Q. For higher-order integration and interpolation schemes in the calculation of Q, use the original version of QSL Squasher (version <2.0).

QSL Squasher is based on the following papers: [QSL3d] and [Coiling]. We kindly ask you[1] to acknowledge both papers and its authors in any program or publication in which you use QSL Squasher (or QSL Squasher – Transverse Version) 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

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

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 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/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
```

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`.

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.

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 values for the quantities of interest on a rectilinear grid spanning the 2d/3d region of interest, which was specified for `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 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
```

The output is parsed with the Python script described in the next section. See that section for a description of the output.

1.3.3 Second post-processing step with Python

The script `viz3d.py` shows examples of post-processing the 3d output from `snapshot`.

The 3d post-processing script, `viz3d.py`, outputs two VTK files containing the global and local quantities in the 3d cube sampled by qslSquasher. Those VTK files can then be visualized by ParaView as in the *example* included with this documentation.

The quantities saved in the VTK files by the Python script are as follows:

"Type": 0 for transverse saddle flow; 1 for node; 2 for LH spiral; 3 for RH spiral; 4 for LH center; 5 for RH center; 6 for improper node (one repeated eigenvector); 7 for star node (two distinct eigenvectors).

"rho_Z" = $\rho_{\mathcal{Z}}$, the squeezing rate (units of 1/Mm)

"omega_c" = ω_c, the coiling rate (units of 1/Mm)

"Trace" = sum of transverse eigenvalues of the gradient of the normalized magnetic field.

"J" = current (defined as $\nabla \times \vec{B}$; has units of [B]/Mm.)

"Alpha" = generalized force-free parameter (units of 1/Mm)

"Alpha_im" = generalized force-free parameter, which is set to zero in regions with real tranverse eigenvalues (units of 1/Mm)

"FLL" = field-line length (in Mm)

"open" = 1 for open field lines; 0 for closed field lines; otherwise, for missing data

"N_c" = coiling number

"log10(Z)" = logarithm of the squeeze factor

"log10(Q)" = logarithm of the squashing factor

"N_t" = standard twist number

"N_t_im" = twist number after dropping saddle-flow contributions (see Section 5.3 of [Coiling])

"FLEDGE" = FLEDGE map

"open_dilat" = the "open" array with removed boundary pixels (between open and closed field-line regions); useful for filtering only closed field lines of the FLEDGE map, without including the large-FLEDGE boundary pixels.

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.

PERIODIC_XY
If set, then use periodic boundary conditions in X and Y.

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

BOX_SIZE
The typical size of the box in Mm. Used to set sane default values for some of the other parameters of the code.

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 local and global quantities associated with the transverse eigenvalues of the gradient of the normalized magnetic field when CALCULATE is set to TRANSVERSE_EIGENVALUES (default). As part of the output, the code will generate numerous .dat files containing the local quantities associated with the transverse eigenvalues. Those are post-processed by the python script and are output into the VTK files. When set to QSL, the code calculates the squashing factor of the field lines passing through each sampled point. One has to go through the same post-processing pipeline, irrespective of the option set by CALCULATE.

RHO_Z
When CALCULATE is set to TRANSVERSE_EIGENVALUES, one can set several options for how the squeezing rate is calculated by setting RHO_Z to one of OPT1_LAMBDA, OPT2_LAMBDA and SYMM_LAMBDA. For the first two, see the equations for options 1 and 2 for ρ_Z in Section 2.2.3 of [Coiling]. For the third (symmetrized) option, see Section 4.5 of the same paper.

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.

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.

MAX_BATCHSIZE

Specifies the maximum number of field lines to be integrated in each refinement before the code exits.

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

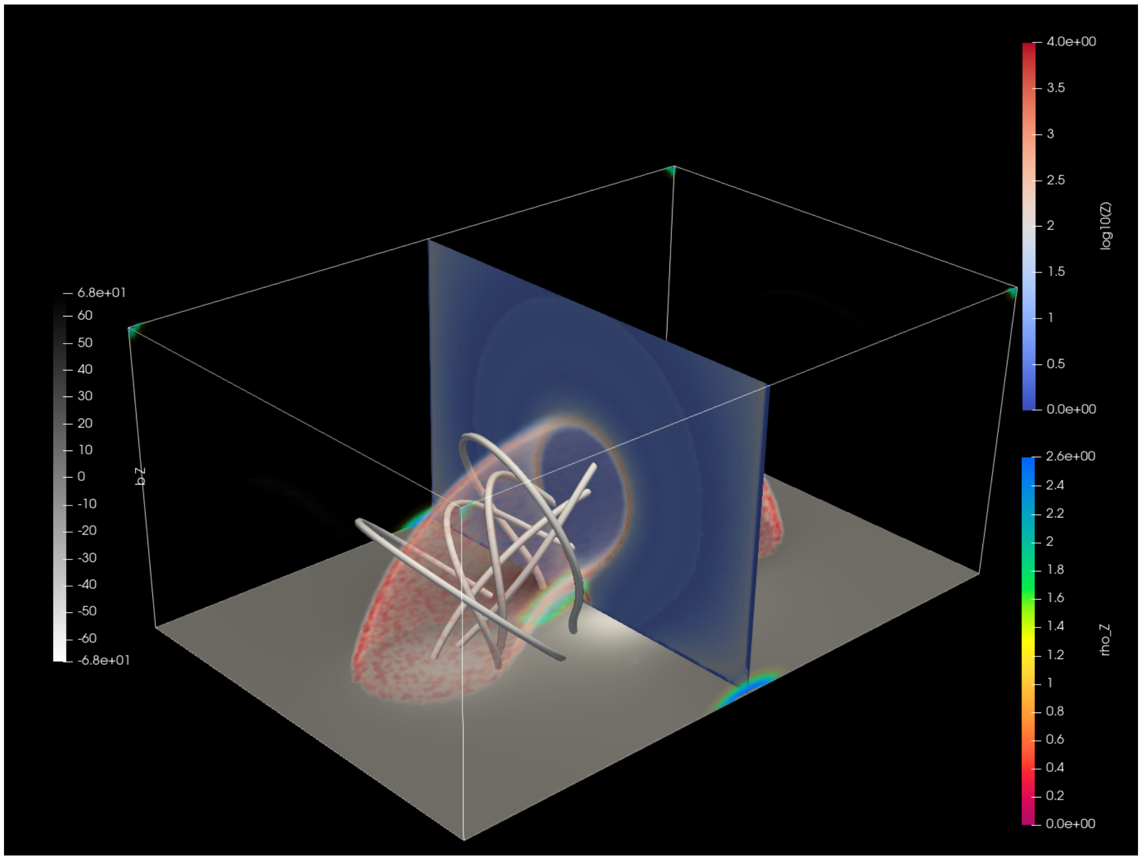
INTEGRATION_STEPS_PER_CELL is used to calculate the step size for the field line integrators. The step size is such that there are roughly INTEGRATION_STEPS_PER_CELL steps in each cell in the input grids. The resulting step size (printed to stderr) is the integration step for the Euler integration scheme.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

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  0m20.256s
  user  0m19.788s
```

(continues on next page)

(continued from previous page)

```
  sys   0m0.393s

$ time ./qslSquasher > raw.dat
  1. pthread-Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz (Portable Computing Language)

  Calculating current ...
  ... done.
  Analyzing transverse FL motions ...
  ... done.
  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 242172 in mesh refinement: 0
     ... skipping lines ...
  # of computed field lines = 242166 out of 242172 in mesh refinement: 0
  Beginning BACKWARD integration along field lines ...
  # of computed field lines = 0 out of 242172 in mesh refinement: 0
     ... skipping lines ...
  # of computed field lines = 242166 out of 242172 in mesh refinement: 0
  Quantities calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 265381
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 265223 in mesh refinement: 1
     ... skipping lines ...
  # of computed field lines = 265221 out of 265223 in mesh refinement: 1
  Beginning BACKWARD integration along field lines ...
  # of computed field lines = 0 out of 265223 in mesh refinement: 1
     ... skipping lines ...
  # of computed field lines = 265222 out of 265223 in mesh refinement: 1
  Quantities calculated successfully.
  Starting sort...
  ... done sorting.
  Starting sort...
  ... done sorting.
  Number of field lines to be integrated in this mesh refinement step: 557436
  Beginning FORWARD integration along field lines ...
  # of computed field lines = 0 out of 556914 in mesh refinement: 2
     ... skipping lines ...
  # of computed field lines = 3897603 out of 3897606 in mesh refinement: 5
  Quantities calculated successfully.
  ^C

  real  19m46.221s
  user  119m21.692s
  sys   2m16.436s
$ time ./snapshot > grid3d.dat

  real  0m12.897s
  user  0m12.611s
  sys   0m0.260s
$ time python viz3d.py
```

(continues on next page)

(continued from previous page)

```

  real  0m19.614s
  user  0m18.337s
  sys   0m1.218s
$ 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`.
- `Global_Quantities.vtr` is a VTK file, containing the rectilinear grid of global quantities in cartesian coordinates.
- `Local_Quantities.vtr` is a VTK file, containing the rectilinear grid of local quantities 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

[Coiling] ‘ Coiling and Squeezing: Properties of the Local Transverse Deviations of Magnetic Field Lines‘, S. Tassev and A. Savcheva (2019), arXiv:???

INDEX