Draft version May 6, 2019
Typeset using LaTeX default style in AASTeX62

`HQVseg`: A python module for discretizing coronal magnetic domains.

Roger B. Scott[1]

[1] *School of Science and Engineering, University of Dundee, Dundee DD1 4HN, UK*

(Received May, 2018; Revised May 6, 2019)

ABSTRACT

In order to understand the topology of coronal magnetic field models it is useful to have a method for partitioning the coronal volume into discrete magnetic flux domains. This can be accomplished by finding all of the magnetic nulls in the volume and then tracing their fan and spine field lines; however, this process is quite laborious and computationally intensive. Additionally, while the magnetic skeleton is, in principle, a complete representation of all topological features, there are morphological features such as hyperbolic flux-tubes, which can be important for magnetic reconnection, that are not captured by this method. An alternative approach is to calculate the magnetic squashing factor and use this to identify so-called quasi-separatrix layers, which can be sites of 3D magnetic reconnection without the presence of magnetic nulls. Because the magnetic squashing factor is scalar quantity that is conserved along a given field line, its isosurfaces are equivalent to magnetic flux tubes, and can therefore be used is indicators of magnetic flux domains. Here we describe a domain segmentation routine that discretises the coronal volume using the magnetic squashing factor as a proxy for topological boundaries. This method draws heavily on methodology developed for image processing and 3D medical imaging, but is entirely open source and depends only on publicly available image analysis libraries.

1. CODE SUMMARY

The strategy of this routine is as follows:

1 The magnetic field model and calculated values for the magnetic squashing factor are read in from disc, and their numerical representations are reconciled to a single grid.

2 A mask is created based on thresholding of the magnetic squashing factor and its gradient. The mask is then padded and holes are removed to ensure that it is 'water tight'.

3 The unmasked volume is partitioned into discrete domains, each with a unique label – the discretisation is performed separately for open and closed field regions to ensure that the open and closed boundary is not spanned by any single domain.

4 The masked region is then labeled using a watershed segmentation, which allows each region to compete for the volume previously occupied by the mask, so that the various regions collectively span the entire coronal volume.

5 The masked volume is then sorted based on proximity to the various discrete domains, with the interfaces between these domains indicating the location of topological and morphological features.

6 Finally, the various subsets of the masked volume are tagged and sorted based on their proximity to each other and to nulls in the magnetic field, so that the inferences can be made regarding the typical rates of occurrence of the various types.

Corresponding author: Roger B. Scott
roger.b.scott@gmail.com

In order to execute this strategy we have created a self-contained `Python` module `HQVseg`, whose methods and classes are described in detail below. This module depends on a number of other publicly available `Python` modules, such as `NumPy`, `SciPy` (Jones et al. 2001), `Scikit-Image` (van der Walt et al. 2014), `ndimage`, to name a few. The hierarchy of classes and their methods is given below, with more detailed descriptions in subsequent sections.

- module: `HQVseg`
  - class: `Inputs()`
  - class: `Source()`
  - class: `Result()`
  - class: `Model()`
    - subclasses: `Inputs(),Source(),Result()`
    - setup methods: `build_grid(),import_squash_data(),import_bfield_data()`
    - segmentation methods: `build_masks(),segment_volume()`
    - metadata methods: `determine_adjacency(),find_interior_HQVs(),find_exterior_HQVs(),get_null_region_dist(),get_null_intHQV_dist(),associate_structures(),categorize_structures()`
    - support methods: `global_reduce(),global_expand(),associate_labels(),get_reg_hqv(),get_reg_bnd(),make_null_box(),get_null_regs()`
    - data i/o methods: `cloan(),save_data(),load_data(),export_vtk()`
    - visualization methods: `gauss_slice(),segcmap(),mskcmap(),visualize()`
  - class: `Foo()`
  - class: `Bytefile()`
  - method: `portattr()`
  - method: `get_mask_boundary()`

The primary wrapper for handling data and related calculations is the `Model()` object class, whose methods and attributes define the main scope of the module. Within this main class there are three additional object classes that contain the data, metadata, and results from the segmentation calculation. These are classes of the module at large but their primary functionality is as attributes of the larger model class. The contents of these data objects is as follows:

- `Model().Inputs()`:
  - Object class containing default values, filenames, and other initialization data.
  - Updated dynamically as various routines check for and subsequently set default values.
  - Attributes: `nrr,nth,nph,phmin,phmax,thmin,thmax,rrmin,rrmax,r_samp,solrad,q_dir,b_dir,glbl_mdl,ss_eof,sbn_thrsh,ltq_thrsh,adj_thrsh,pad_ratio,bot_rad,auto_imp,auto_seg,protocol,vis_title,initialization`
- `Model().Source()`:
  - Object class containing source magnetic field data and numerical grid.
  - Populated by execution of setup methods.
  - Attributes: `crr,cth,cph,brr,bth,bph,slog10q,metrr,metth,metph,null_locs`.
- `Model().Result()`:
  - Object class containing data generated by segmentation routines and subsequent post-processing.
  - Populated by execution of segmentation methods and metadata methods.
  - Attributes: `hqv_msk,PIL_msk,GlnQp,reg_width,hqv_width,pad_msk,vol_seg,seg_msk,labels,open_labels,clsd_labels,opos_labels,oneg_labels,adj_msk,adj_msk_shape,adj_msk_boolsize,exterior_HQVs,interior_HQVs,null_info,null_to_region_dist,null_to_intHQV_dist`.

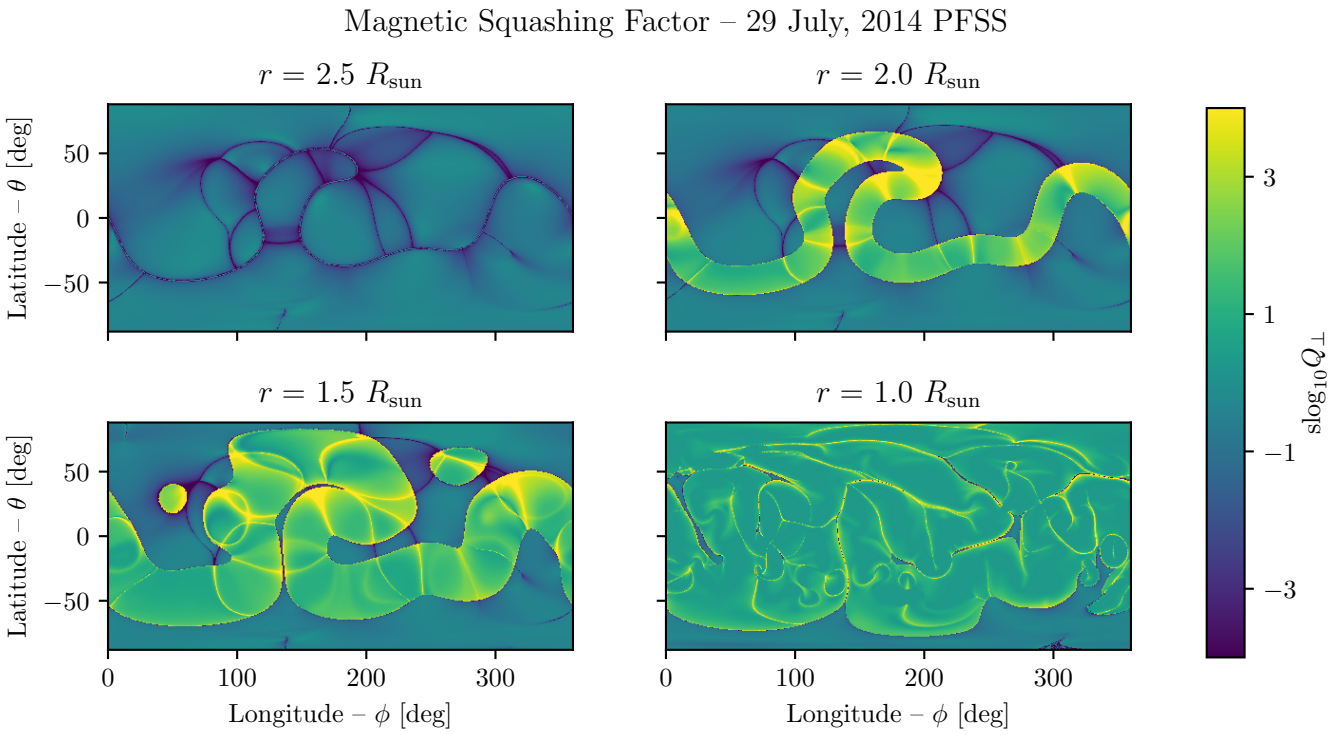

Figure 1. Sign $\log_{10} Q$ at radial heights of $r = \{1.0, 1.5, 2.0, 2.5\} R_{\rm sun}$ for 29 July, 2014 PFSS. Dark colors indicate open flux, corresponding to coronal holes at the lower boundary. Dark bands at the upper boundary indicate the intersections of QSLs with the source surface. The yellow curve in the top panel shows the apex of the helmet streamer, just below the heliospheric current sheet.

2. SETUP

Setting up the model consists of reading in data from disc, constructing the numerical grid, and reconciling the magnetic field data to the finer resolution grid of the squashing factor data. Squashing factor data is output from the post-processed `qslSquasher` code, for which the data is stored in a flattened datafile with no coordinate information. As such, the grid is first built (`Model().build_grid()`) by assuming typical limits and resolution to the coordinate domain, consistent with the definition used in the `qslSquasher` routines. The squashing factor (Q) data is then read in (`Model().import_squash_data()`) and reshaped to match the grid dimensions. The source magnetic field ($\boldsymbol{B}$) data is then read in (`Model().import_bfield_data()`) along with the associated coordinate axes, and then interpolated onto the appropriate grid.

The routines used assume by default that the data is in a particular format; however, these should be readily extensible to variations in formatting, either through changes to keywords or restructuring of the code. The default functionality of these routines is given below.

The `Model().build_grid()` method assumes a default configuration of `nrr` = 120, `nth` = 4 `nrr`, `nph` = 8 `nrr`, with coordinates spaced evenly in $0 \leq \ln(r/R_\odot) \leq \ln(2.5)$, $-88° \leq \theta \leq 88°$, $0 \leq \phi \leq 360°$. The value of $R_\odot$ is set explicitly to `inputs.solrad`=696 and the resulting coordinate grid is a finite difference mesh with coordinates written to the variables `source.crr`, `source.cth`, `source.cph`. Coordinate gradients (diagonal metric entries) are also calculated and written to `source.metrr`, `source.metth`, `source.metph`.

With the grid in place, the `Model().import_squash_data()` method assumes that the squashing factor data are stored in a file called `q_dir/grid3d.dat`, where `q_dir` is an attribute of `Model().inputs` and is set during instantiation. The data is assumed to have the dimensions defined in `Model().build_grid()`, and is written as a nested loop of the form given in Tassev & Savcheva (2017). If `inputs.q_dir` is not set explicitly, the current working directory is inspected for the appropriate file. Once the data are read in to memory they are stored in

the variable `source.slog10q`, which is assumed to correspond to the $\text{slog}_{10}Q_\perp$ representation from Scott et al. (2018)

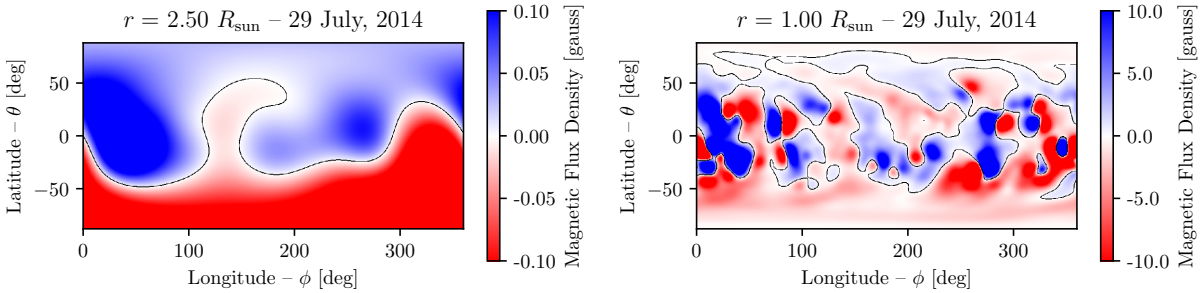

Figure 2. Radial magnetic field $\mathbf{B}^{(8)}$ from 29 July 2014. The top and bottom panels represent the source surface ($r = 2.5R_{\rm sun}$) and photospheric ($r = 1.0R_{\rm sun}$) boundaries. Data is smoothed with a pseudo-gaussian filter of the form $f = exp(-l(l+1)k)$, with $k = 0.002$. The field is then interpolated onto a grid with the same dimensions as Q (see section **??**) for consistency.

The `Model().import_bfield_data()` method assumes that the magnetic field data are stored in a collection of files `b_dir/bx0.dat`, `b_dir/by0.dat`, `b_dir/bz0.dat`, with corresponding coordinate axes in `b_dir/xs0.dat`, `b_dir/ys0.dat`, `b_dir/zs0.dat`. As above, `b_dir` is an option stored in `inputs.b_dir` and is set, by default, to be `b_dir=q_dir+bfield_data/`. Consistent with the convention used in `qslSquasher`, the x, y, z variable are taken as proxies for ϕ, θ, r in spherical coordinates. The data are assumed to be unrolled in the same fashion as `slog10q`. Interpolating functions are then generated assuming a regular grid and the data are then interpolated onto the same grid as `slog10q`, and the original data are discarded. The interpolated values of the magnetic field are stored at `source.brr`, `source.bth`, `source.bph`. Additionally, data indicating the number and location of magnetic nulls, which are calculated separately and stored in `b_dir/nullpositions.dat`, are imported at this step if available.

3. SEGMETATION

The actual volume segmentation is performed in combination between two methods of the `Model()` class, `Model().build_hqv_msk()` and `Model().segment_volume()`. The first method is a standalone routine that constructs a mask for identifying so-called High Q Volumes (HQVs). The routine first recovers the perpendicular squashing factor from the array of stored values as $Q_\perp = 10\hat{}|\text{slog10q}|$. The (radius-normalized) squared Sobolev norm is then constructed as

$$N_S^2 = Q_\perp + \frac{r}{R_\odot}\boldsymbol{\nabla}_r \ln(Q_\perp), \tag{1}$$

where r and $R_\odot$ are the local radius and solar radii, respectively, and $\boldsymbol{\nabla}_r$ is the numerical gradient vector, normalized to the local value of r.

Since $Q_\perp$ is formally infinite at various places in the domain, N_S^2 has no well defined mean; however, we can compute a proxy for the typical value of N_S^2 as

$$\tilde{N}_S^2 = \sqrt{\left\langle \left(2 + \frac{r}{R_\odot}\boldsymbol{\nabla}_r \ln Q_\perp\right)^2 \right\rangle}, \tag{2}$$

where $Q_\perp$ has been replaced by its theoretical minimum value of 2. With this characteristic value we can then define the High Q Volume as a subset of the larger domain given by

$$\Omega_{\rm HQV} = \left\{ \mathbf{x} \in \Omega \quad | \quad \left(N_S^2(\mathbf{x}) > 1.5\,\tilde{N}_S^2 \quad \text{or} \quad \log_{10} Q_\perp > 3.7\right) \right\}, \tag{3}$$

which is stored as a mask according to the boolean

$$\texttt{hqv\_msk}_{ij} = (1 \quad \text{if} \quad \mathbf{x}_{ij} \in \Omega_{\rm HQV} \quad , \qquad 0 \quad \text{else}). \tag{4}$$

The `Model().build_hqv_msk()` method makes no strict assumptions about the grid but does assume that positive (negative) values of `slog10q` correspond to closed (open) magnetic flux domains. The gradient thresholding described above is performed using built-in `numpy` routines and the mask is smoothed using morphological opening and closing from the `scikit-image.morphology` package. Defaults for the thresholding are stored in `inputs.ltq_thrsh` and `inputs.sbn_thrsh`, and can be specified separately prior to execution. The resultant mask is stored in `result.hqv_msk` and $\boldsymbol{\nabla}_r \ln Q_\perp$ is stored at `inputs.GlnQp` for diagnostic purposes.

Following the creation of the `hqv_msk`, the volume is then discretized using the `Model().segment_volume()` routine, which is built on the operating assumption that domain boundaries should lie within $\Omega_{\rm HQV}$ and be locally perpendicular to $\boldsymbol{\nabla} Q_\perp$. The segmentation is performed in a number of steps that are designed to minimize the possibility of separate flux domains having a common label, especially across the open-closed boundary. Note that while the visuals provided here are indicative of the behavior at the top boundary, the segmentation is fully-3D and includes open and closed flux, throughout the volume.

Prior to segmenting the volume, we first consider whether the azimuthal boundary is meant to be periodic, in the case of a global model (`inputs.glb_mdl`). If it is, we first expand the `hqv_msk` and `slog10q` arrays to double their azimuthal extent, by stacking to adjacent versions of these, and then sliding by 180° in ϕ. This is done to place the periodic, azimuthal boundary, within the interior of the numerical domain, so that it will be transparent to the subsequent segmentation. A future improvement to this method would simplify this process by incorporating periodicity directly into the underlying segmentation routines, which do not support periodicity in their standard form.

The first step in the segmentation is to pad the mask to ensure that there are no 'leaks' from numerical artefacts. This is accomplished by calculating the euclidean distance transform (`scipy.ndimage.distance_transform_edt`) to determine the distance of all non-HQV points to the nearest element of $\Omega_{\rm HQV}$. This value is stored in `hqv_dist` and a new mask, `pad_msk`, is then constructed to reflect all points with `hqv_dist` greater than a threshold value. The threshold is set by calculating the distance transform for the compliment of $\Omega_{\rm HQV}$, from which we get the mean linear width of a typical HQV element, which is stored at `hqv_width`. The `pad_msk` variable is then used to discretely label all unique, simply connected domains, using the `skimage.measure.label` method, which assigns a unique integer label to all pixels within each unique domain. This labling is performed separately for open and closed domains, which are assigned positive (negative) labels consistent with the convention for `slog10q`. The combined label array is stored as `vol_seg`, which is an integer array of the same size as `slog10q`.

Following the initial labeling, we must still assign domain labels to the HQV volume, as we intend for the various domains to completely span the numerical domain eventually. The intention is that the HQV regions should be assigned labels from adjacent low-Q regions so that eventually each label corresponds to a region with low-Q in its interior and high-Q only and exactly at the interfaces between adjacent regions. To accomplish this, we use the `skimage.morphology.watershed` routine, which takes a collection of seed values, and populates unmasked pixels with seed values consistent with a watershed-basin model.

If unchecked, the method will populate the entire domain; however, we wish to enforce certain constraints on how the regions are grown, so the watershed is performed in a series of individual steps, with the mask peeled away in stages. At the outset, we remove any domains with a volume below a threshold value, as these tend to be single-pixel domains and are taken to be artefacts. We then proceed with the expansion of remaining regions, and in the following, each step is always performed twice, once for open domain, and once for closed domains, so as to ensure that open flux is never assigned to a closed flux domain, and visa-versa. First, the padded buffer is backfilled with a watershed growth into the unlabled pixels in `pad_msk` but not in `hqv_msk` and a same-type-only (e.g., open or closed) mask is used to ensure contiguous growth. Second, the watershed is performed on unlabled pixels within the `hqv_msk`, again with a same-type-only contiguous growth mask, to fill smooth, non-pathological elements of the HQV. Third, we repeat the watershed on unlabled pixels in the `hqv_msk` but now without contiguous growth, as we recognize that domains can appear to be pinched off when they are in fact connected below the grid resolution. And finally, we remove all masks and label as-yet unlabled pixels, to allow for population of junk data-values where the flux could not be accurately reported as strictly open or closed.

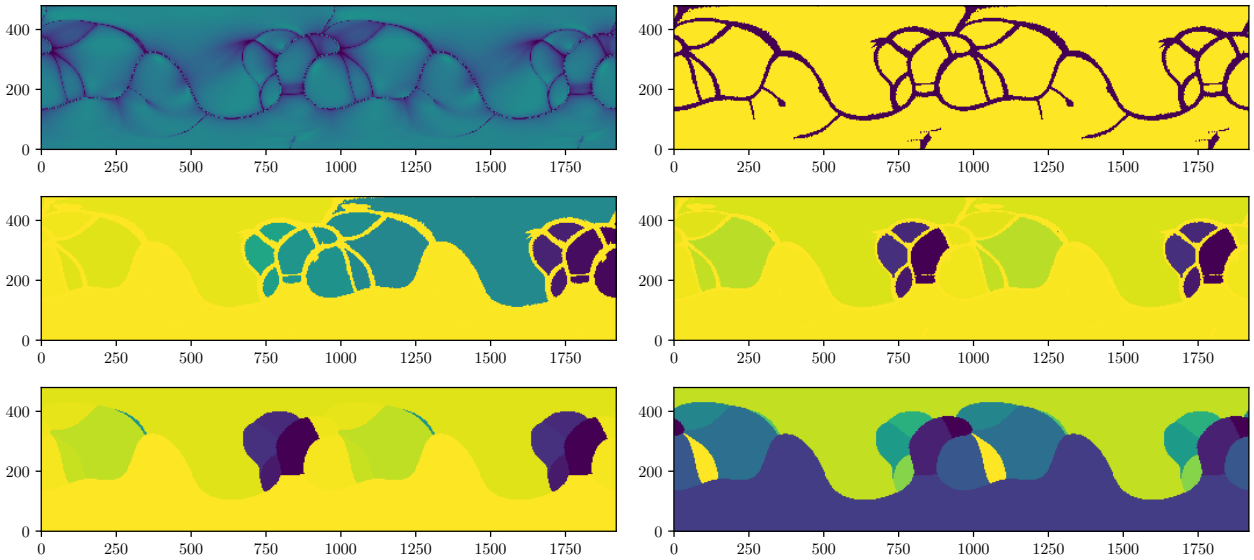

Figure 3. Stages of volume segmentation: raw $Q_\perp$ map (tl), HQV mask (tr), discrete labeling (cl), periodic relabeling (cr), backfilling (bl), label reduction (br).

After segmenting the volume, we must still reconcile the periodicity of the domains in the case of a global model. As the labeling was performed with an artificially enlarged domain, the azimuthal boundaries are placed within the interior of the numerical domain during segmetnation, so there are no discontinuities at these locations; however, it remains that the labels on the left side of the domain are different than their counterparts on the right side. To rectify this we use another routine, `Model().associate_labels()`, which breaks to domain into its two adjacent copies, and then compares their labels pairwise to look for overlaps. Once the overlaps are identified, the domains are iteratively renamed and the pairwise matches updated, until all overlaps have been reconciled. The domain labels are then reduced to a compact, monotonic set, with missing elements removed, and are classed based on whether the flux is open or closed, with open flux further subdivided into two groups based on the sign of the magnetic field at the top boundary. Once the domains have been relabeled, the domain extent is reduced to the original grid and the label map is written to `result.vol_seg` with the various lists of different typed labels recorded as `result.labels`,`result.clsd_labels`,`result.open_labels`,`result.opos_labels`,`result.oneg_labels`.

4. HQV IDENTIFICATION

In order to identify which portions of the high-Q volume lie at the interfaces of specific domains, we must find the distance from each point in the volume to nearest portion of given domain. This is accomplished by the routine `Model().determine_adjacency()`, which loops over all possible domain labels and calculates the distance from every point in the volume to the nearest point in a given domain according the the `scipy.ndimage.distance_transform_edt()` routine. The resultant value is compared to a threshold value, which is set by default to be equal to `result.hqv_width` in the middle of the domain, and scales with the local radius. The collection of all points whose distance to a given domain is less than the threshold value are said to be adjacent to the domain, and this information is stored as a boolean map. The ensemble information for all domains is boolean array called `result.adj_msk`, with dimension (`nrr` $\times$ `nth` $\times$ `nph` $\times$ `n_regs`), where `n_regs` is the total number of discrete domains, stored at `result.n_regs`.

Because `Python` stores boolean as an integer type, `result.adj_msk` is quite large in practice, so for practical reasons it is compressed to a bitwise array when storing to disc. The adjacency mask is used directly by the `Model().get_reg_hqv()` routine, which returns a mask corresponding to all pixels that lie within the larger `hqv_msk` and also identify as adjacent to whichever domain is specified. If more than one domain is specified the result is either the intersection (default) or union of the individual entries. By testing for the existence of non-empty intersection groups with various domain pairings, a list of all non-trivial interface HQVs is formed using the

routine `Model().find_exteriorHQVs()`. The result of this testing is an attribute called `result.exterior_HQVs`, which is a list whose entries are objects containing the various metadatum about each pairing.

The collection of `exterior_HQVs` necessarily omits any high-Q volume that is contained entirely within a single domain, so an additional routine is required to identify such structures. This is accomplished by a routine called `model.find_interiorHQVs()`, which loops through each domain and identifies portions of the `hqv_msk` that associates with that domain and no other. These sub-volumes are then inspected for contiguous structures, which are labeled and filtered against a size threshold, after which surviving elements are allowed to expand back to the domain boundary using a simplified version of the same watershed technique used for the larger segmentation. The result of this search is another attribute called `result.interior_HQVs`, which again is a list whose entries are objects with attributes containing metadatum about the various unique, non-interface high-Q volumes.

The final result of this sorting of high-Q sub-volumes is a pair of lists containing masks (or pointers to masks) and metadata for every non-trivial high-Q structure, catalogued according to the specific flux-domains to which they are adjacent. In the case of `result.exterior_HQVs`, each entry contains a list of labels, and the corresponding high-Q volume is constructed by passing this list as an argument to `Model().get_reg_hqv()`. In the case of `result.interior_HQVs`, since a given domain can contain multiple separate high-Q volumes within it's interior, each entry contains a label number and a sub-label number, along with a boolean mask for that specific high-Q structure.

5. HQV INSPECTION

Following the initial generation of the `interior_HQV` and `exterior_HQV` attribute lists, we use a combination of routines to determine how each entries associates with other relevant features. We begin with the `Model().get_null_regions_dist()` and `Model().get_null_intHQV_dist()` methods, which tabulate distance from every magnetic null to the nearest point in a given domain, or subdomain in the case of interior HQVs. After determining the distances to the nulls, the `Model().associate_structures()` routine iterates through each entry in `interior_HQVs` and `exterior_HQVs` and tests to see if either list has entries that are co-spatial with entries from the other list, in which case such associated structures are appended to each others metadata. The magnetic nulls are then reordered according to distance from each HQV entry, and vice versa, so that every HQV object contains its own list of magnetic nulls, ordered by distance, and every null contains its own lists of interior and exterior HQVs, again ordered by distance. Finally, for each element in `exterior_HQVs`, the properties of its various parent and child entries and associated interior HQVs are inspected in detail to determine whether a given HQV structure is to be considered a simple layer, a branching layer, a vertex, or an element of the heliospheric current sheet (HCS) or open-closed boundary (OCB), according to logic developed for consistency with Scott et al. (2018).

As before, we take vertices, simple arc segments, branching arc segments, and detached arc segments, to refer to specific morphological features at the outermost boundary. Here these structures are generalized to 3-dimensions and we consider their extension into the coronal volume. The details of this determination can be rather subtle in some cases, but broadly speaking the logic is as follows:

- An exterior HQV is considered part of the HCS if the associated label list contains one or more entries from each of `result.opos_labels` and `result.oneg_labels`.
- An exterior HQV is considered part of the OCB if the associated label list contains one or more entries from each of `result.clsd_labels` and either `result.opos_labels` or `result.oneg_labels`.
- An exterior HQV is considered a 'vertex line' if is derived from the interface of three or more domains, which merge at a point away from the HCS and OCB, so that it's imprint at the outer boundary is consistent with a 'vertex'.
- An exterior HQV is considered a 'branching layer' if it is derived from the interface of a pair of domains and is partially co-spatial with a 'vertex line'.
- An exterior HQV is considered a 'simple layer' if it is derived from the interface of a pair of domains and is not at least partially co-spatial with a 'vertex line'.

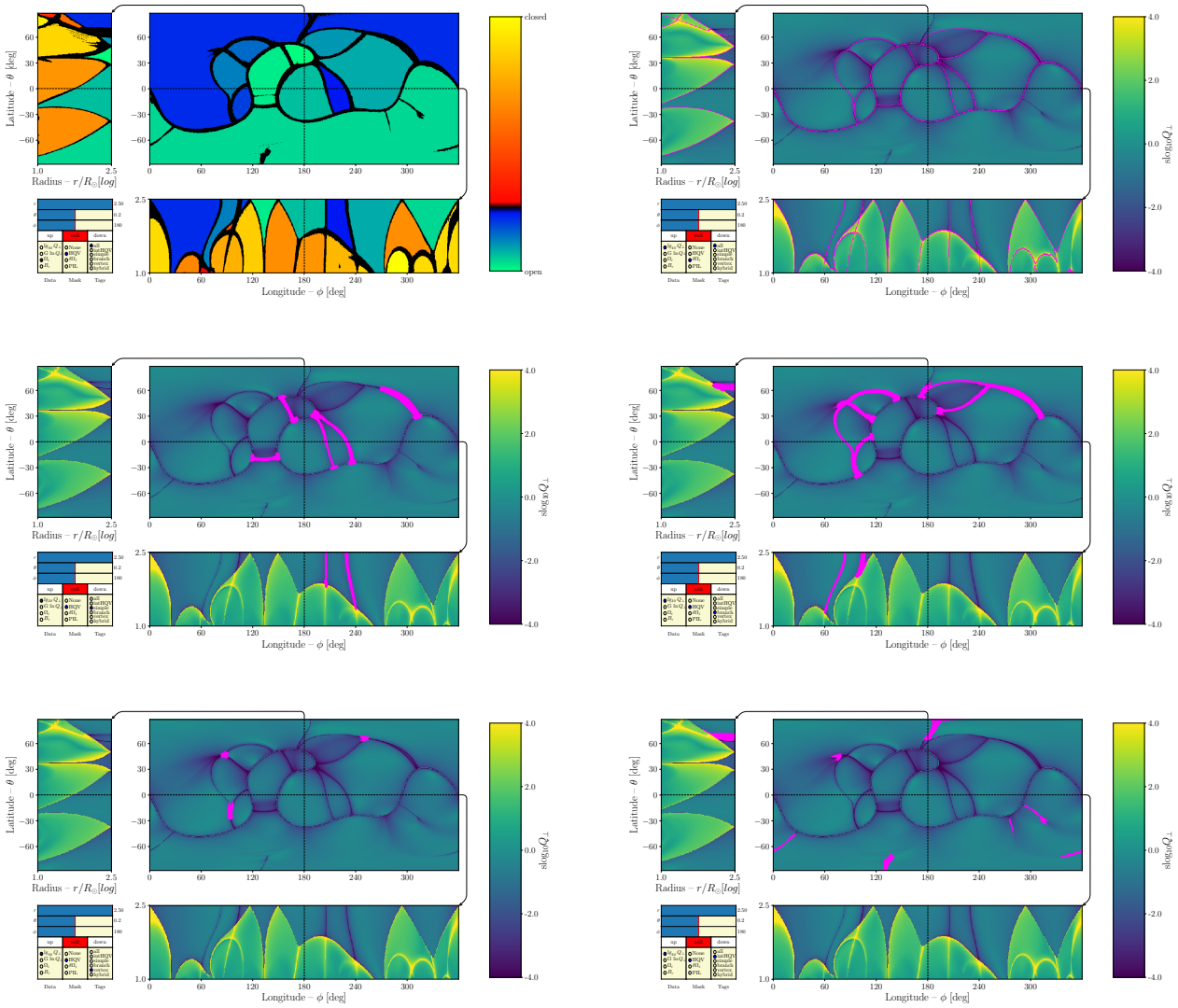

Figure 4. Visualization of catalogued structures: Segmentation map (tl), $Q_\perp$ map with domain boundaries (tr), $Q_\perp$ map with 'simple layers' (cl) $Q_\perp$ map with 'branching layers' (cr), $Q_\perp$ map with 'vertex lines' (bl), $Q_\perp$ map with interior HQVs ('br').

– Interior HQVs, which do not partition discrete flux domains, form the collection of fully and partially detached high-Q layers, which are distinguished by the properties of their `associated_extHQVs`.

The above definitions are a summary of relevant properties of each type; however the exact logic is significantly more complex as it deals with a variety of possible counter-examples, such as, e.g., the intersection of a pair of domains that exists only and entirely within a vertex formed of four or more groups.

6. MISCELLANEOUS

Execution of the entire model generation begins with first loading the module, and then instantiating a model object as `model=HQVseg.Model()`. The working directory can then be set by `Model().inputs.q_dir='your_path'`, or by passing the argument `'your_path'` as an option to `Model().inputs=HQVseg.Inputs(q_dir='your_path')`. In either case, once the model object has a valid entry for `inputs.q_dir`, the various routines can be executed by calling their wrappers: `Model().do_import()`,`Model().do_segment()`,`Model().do_itemize()`,`Model().do_inspect()` or with the global wrapper `Model().do_all()`. Individual models can be written to or read from disc using the `Model().save_data()` and `Model().load_data()` routines, with either

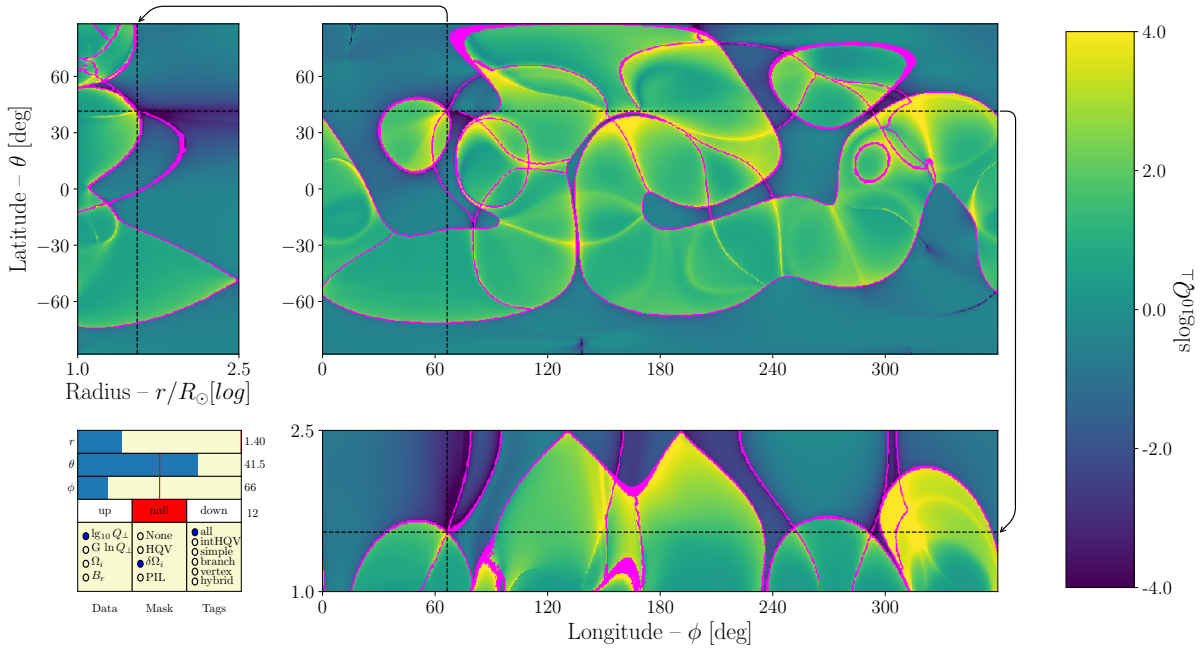

Figure 5. GUI visualization demonstrating slices, masks, and null stepping.

explicit or default filenames. The save format is handled by `Numpy's pickle` method, using a custom file type (`Bytefile()`) which executes the write in batches to avoid buffer issues.

Visualization is handled through a GUI that is launched with the command `Model().visualize()`. The GUI shows slices through three different coordinates, set using sliders, and allows stepping through the locations of the magnetic nulls. Displayed data can be set to $\slog_{10}\mathrm{Q}_\perp$, $\boldsymbol{\nabla}\ln Q_\perp$, `vol_seg`, or B_r, while the mask can be set to 'None', `hqv_msk`, $\delta\Omega_i$, or 'PIL'. The mask can be further reduced with tags that select the subsets for 'all', 'interior HQVs', 'simple layers', 'branching layers', 'vertex lines', or 'hybrid', which shows layers and interior HQVS that are mutually co-spatial. Additional visualization can be achieved manually by plotting the various arrays directly, or by calling the `Model().export_vtk` method, which accepts various keywords for writing either the entire model or subsets of the model to VTK files, which can be rendered using various routines such as ParaView.

REFERENCES

Jones, E., Oliphant, T., Peterson, P., et al. 2001, SciPy: Open source scientific tools for Python, http://www.scipy.org/, [Online; accessed 2018-01-15]

Scott, R. B., Pontin, D. I., Yeates, A. R., Wyper, P. F., & Higginson, A. K. 2018, ApJ, 869, 60

Tassev, S., & Savcheva, A. 2017, ApJ, 840, 0

van der Walt, S. t. e., S ch onberger, J. o. L., Nunez-Iglesias, J. u., et al. 2014, PeerJ, 2, e453