Efficient Software for Programmable Visual Analysis using Morse-Smale complexes

Nithin Shivashankar and Vijay Natarajan

Abstract The Morse-Smale complex is a topological data structure that represents the behavior of the gradient of an input scalar field. Recent years have witnessed a significant number of applications that use this data structure for visualization and analysis of data from various scientific domains. However, these applications have required significant expertise in the implementation of algorithms. This potentially makes such analysis inaccessible to a large audience. In this paper we present open source software modules for the computation, analysis, and visualization of scientific data using the Morse-Smale complex. The modules, named *pymstri* and *pyms3d*, are intended for domains represented using 2D triangle meshes and 3D structured grids respectively. The software is designed to significantly reduce the effort required to use Morse-Smale complex based analysis. Also, the software leverages modern multi-core CPU and GPU architectures for computational efficiency. We demonstrate the usefulness via a case study to visually analyze and interactively segment the eye of the Hurricane Isabel simulation dataset. In particular, we highlight the ability to couple the visual analysis and the computation with ParaView, a popular general purpose visualization tool. The code is is available at the project website http://vgl.serc.iisc.ernet.in/mscomplex/.

Nithin Shivashankar
Department of Computer Science and Automation, Indian Institute of Science, CV Raman Road, Bangalore-12, e-mail: nithin@csa.iisc.ernet.in

Vijay Natarajan
Department of Computer Science and Automation, and Department of Computational and Data Sciences, Indian Institute of Science, CV Raman Road, Bangalore-12, e-mail: vijayn@csa.iisc.ernet.in

1 Introduction

In recent years, topological methods have gained wide popularity for scientific data analysis. In particular, gradient based analysis and the Morse-Smale complexes have been extensively applied for a multitude of data-analysis and visualization tasks [9, 15, 16, 20, 29, 31, 41]. A key reason for the success of Morse-Smale complex based analysis is that it enables a translation from scalar function data into topology and gradient based features. It is therefore no surprise that a lot of recent work has focused on various aspects of the Morse-Smale complex such as its efficient computation [17, 19, 32, 34, 38, 39], feature directed visualization [22], user defined feature preservation [18, 21], etc. In most of the above applications of the Morse-Smale complex, the software implementations are not open source. Also, the software is often developed as standalone a executable that does not easily interface with other large software tools.

Related Work: complexes were first introduced for the analysis of dynamical systems by Smale [40]. The first computational algorithms were described by Edelsbrunner *et al.* [12] and Bremer *et al.* [7], where the definition of the 2D Morse-Smale was extended to triangulated domains resulting in the *Quasi-Morse-Smale* complex. Other methods that develop this notion for 3D are available in the literature [11, 23, 24, 25]. In recent years, many algorithms based on Forman's [14] discrete Morse theory have become popular [17, 19, 34, 38, 39]. A primary reason for this is the combinatorial robustness of these algorithms, which greatly simplifies implementation effort. However, to the best of our knowledge, source code implementation of these methods are only available for Sousbie *et al.* [41]. A crucial aspect in Morse-Smale complex based analysis is in its toplogical simplification and subsequent analysis. Edelsbrunner *et al.* describe the notion of topological persistence [13] and its application to Morse-Smale complex simplification in 2D [12]. Most of the implementations described in the above literature describe some form of simplification using persistence. However, in many applications [15, 20, 41] successful feature identification requires simplification using other sources of data as well as domain specific criteria. In these cases the effort needed to apply the analysis is dependent on the ease with which one interfaces with the Morse-Smale complex datastructure. More generally, source code implementation for Reeb graphs [10] contour trees [8] and persistent homology [5, 4] have become available over the years. These packages are most readily usable as standalone packages, though most offer access to their internal data structures only in their native programming language. Python [35], being particularly suited for high level scripting operations, is very often available as the de-facto scripting interface in a many large software tools. A primary reason for this is that Python is a mature interpreted language which allows for easy runtime loading/unloading of modules. A few examples of tools that offer Python interfaces include Pymol [36], VMD [27], Chimera [33], ParaView [26], VTK [37], Blender [6].

Contributions: In this paper, we describe a Python-scriptable Morse-Smale complex computation and analysis package. The package consists of two modules, named and , which contain implementations for 3D structured grids and 2D trian-

gle meshes. The modules implement efficient algorithms for Morse-Smale complex computation [39, 38, 34] and hierarchical feature analysis [7, 22]. Furthermore, the implementations leverage the OpenCL and OpenMP frameworks for parallel computation. Access to the combinatorial structure and geometric elements of the Morse-Smale complex are granted through a Python interface. We demonstrate the usefulness of the software via a case study. We interactively segment the Hurricane Isabel Dataset [42] to highlight the interactivity and the ease of use. The *pyms3d* module is used to develop a programmable filter which is loadable in ParaView [26]. The extracted features may be changed and updated from within the ParaView runtime environment to drive interactive feature based visual analysis. This case study is demonstrated via a video hosted at https://youtu.be/UX1q9gI2DEk.

2 Background

In this section, we introduce the necessary background relevant to the definition of the Morse-Smale (MS) complex.

Morse theory and the MS complex: Morse theory studies critical points of smooth scalar functions defined on manifolds [30]. Given a smooth scalar function $f : \mathbb{R}^n \to \mathbb{R}$, its *critical points* are points where the *gradient*, the vector of first order partial derivatives $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)\right)$, is identical to zero. A critical point is non-degenerate if the *Hessian*, the matrix of second order partial derivatives, is non-singular. The function f is said to be *Morse* if all its critical points are non-degenerate. The *index* of a critical point is the number of negative eigenvalues of the Hessian matrix. An *integral line* is a maximal curve in $\mathbb{R}^n$ whose tangent at every point is collinear with the gradient of f at that point. The limit points of integral lines, $t \to \pm\infty$, are the critical points of f.

The set of all integral lines that share a common source (destination) p, is called the *ascending manifold* (*descending manifold*) of p. The *Morse-Smale (MS) complex* is a partition of the domain into cells formed by the collection of integral lines that share a common source and a common destination. The combinatorial structure is a graph, where the nodes are critical points and edges are arcs between them if there is an integral line that connects them and their indices differ by one.

Discrete Morse theory: Forman [14] introduced discrete Morse theory to study the topology of cell complexes. A d-cell α^d is a topological space homeomorphic to a d-ball $B^d = \{x \in \mathbb{R}^d : |x| \leq 1\}$. Lower dimensional d-cells include vertices, edges, triangles/quads, and tetrahera/cubes. A *cell complex* K is a collection of cells where the set of cells incident on the boundary of a cell are also in K, and two cells intersect only along a single common boundary cell. Examples of cell complexes include 2D triangle meshes and 3D cubical complexes (see Figures 2(a) and 2(b)). A function $f : K \to \mathbb{R}$ is said to be a *discrete Morse function* if for all d-cells α in K, there exists no more than one incident higher dimensional cell β so that $f(\alpha) \leq f(\beta)$, and no more than one incident lower dimensional cell γ exists so that $f(\gamma) \leq f(\alpha)$. A pairing between two incident d-$d+1$ cells, α-β, so that $f(\alpha) \geq f(\beta)$ is

called a *discrete vector*. A *V-path* is a sequence of unique d-$d+1$ discrete vectors, $\alpha_0^d, \beta_0^{d+1}, \alpha_1^d, \beta_1^{d+1}, \ldots, \alpha_r^d, \beta_r^{d+1}, \alpha_{r+1}^d$, so that every $d+1$ cell is incident on the next d-cell. A *discrete gradient field* is a collection of V-paths without any non-trivial loops. Acyclic V-paths correspond to the notion of integral lines of Morse functions. A cell that is not paired is called a *critical cell* and is analogous to the notion of a critical point. Ascending / descending manifolds and the combinatorial structure are similarly defined for discrete Morse functions. Figure 2(c) shows an example of the combinatorial structure of the Morse-Smale complex defined on a cell complex using discrete Morse theory.

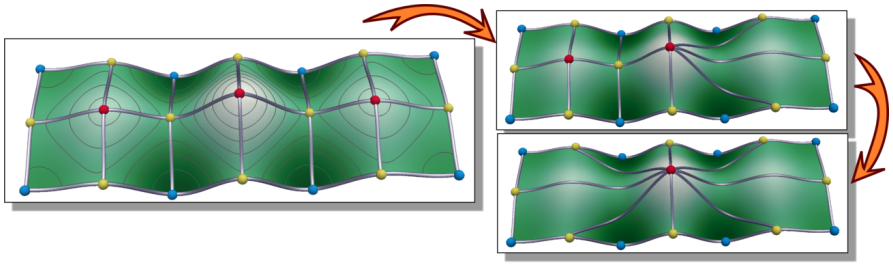


Fig. 1 *(left)* The combinatorial structure of the Morse-Smale complex of a function with three maxima. The critical points are shown as spheres, blue for minima, yellow for saddles, and red for maxima. The arcs are shown as gray tubes. *(right)* Two cancellation operations applied to the Morse-Smale complex eliminate two maximum saddle pairs connected to the central maximum. Each cancellation operations re-routes arcs from the maxima connected to the canceled saddle to the saddles connected to the canceled maximum. The two cancellations result in two successive versions of the Morse-Smale. This sequence of versions is referred to as the hierarchical Morse-Smale complex.

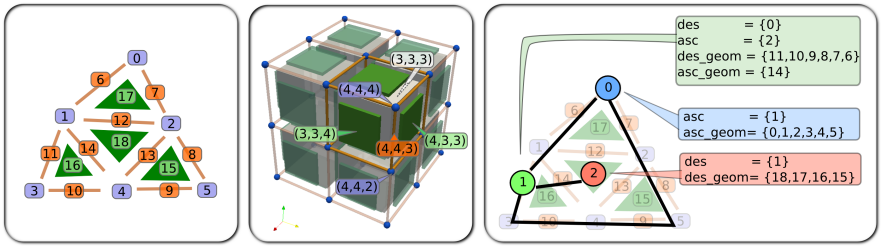
Topological Cancellation and the Hierarchical MS complex: A topological cancellation is a process of removal of a pair of index i-$i+1$ critical points p-q that are singularly connected in the combinatorial structure [12]. The combinatorial structure is modified so that all other index $i+1$ critical points connected to p are connected to all other index i critical points connected to q. The ascending (descending) manifold of p (q) is merged with the ascending (descending) manifolds of all other i ($i+1$) critical points connected to q (p). *Topological persistence* [12] measures the importance of a pair of critical points. Pairs of critical points are canceled in increasing order of its *persistence*. As one iteratively applies the above operations to simplify the MS complex, each application results in a new combinatorial and geometric version of the MS complex. This sequence of MS complex versions is referred to as the hierarchical MS complex, where each version is indexed by its position in the sequence. Selecting appropriate versions for feature analysis is often challenging and thus it is desirable to try multiple versions before selecting one. Figure 1 shows an example of a Morse-Smale complex along with two cancellations applied to it to generate a hierarchical MS complex with three versions in the sequence.

3 Data representation and Algorithms

We now describe the data structures and algorithms that are implemented in the software package. We first describe how the data over 2D surfaces and 3D grids are represented. Next, we describe the data structure used for the Morse-Smale complex. Finally, we briefly describe the discrete Morse-Smale complex computation algorithm and the construction of the Hierarchical Morse-Smale complexes.

Data Representation

Cells of the underlying domain are represented using unique identifiers. This is relevant when querying the geometry of the Morse-Smale complex. The data representation of the Morse-Smale complex is common to both *pymstri* and *pyms3d*. This is relevant for analysis of the combinatorial structure of the Morse-Smale complex.



(a) (b) (c)

Fig. 2 Data representation. (a) In 2D triangle complexes, each cell in the complex is identified by a unique integer ID. (b) A 3D structured grid is interpreted as a cubical complex. Each cell in the complex is uniquely identified by the centroid of its Cartesian coordinate scaled by two. (c) The Morse-Smale complex is represented as a graph whose nodes are critical points and arcs are edges between them. Each critical point is given a unique integer identifier. For each critical point, the lists *asc* and *des* contain the ID's of the ascending and descending critical points connected to it. Similarly, for each critical point, the lists *asc_geom* and *des_geom* contain the cell identifiers of ascending and descending cells from the underlying domain, (shown in (a)).

2D surfaces in pymstri: In the implementation of pymstri, a triangle mesh representing the surface is stored using the edge-facet data structure [28]. Triangle meshes are assumed to be available as a set of vertices and a set of triangles where each triangle specifies three indices into the list of vertices.

3D structured grids: The structured 3D grid domain is interpreted as a cubical cell complex whose cells are vertices, edges, quads, and cubes. The cells of the domain are implicitly represented using the Cartesian coordinates of their centroids as identifiers. Each cell is uniquely identified using a tuple with three integers. We scale the coordinates by two so that the interleaving cells, namely edges, faces, and cubes, also obtain integral coordinate values at their centroids. Queries for facets / cofacets

are therefore implicitly computed taking into consideration the boundary conditions imposed by the grid. These queries use integer arithmetic instead of floating point arithmetic as a consequence of the scaling described above. The center panel in Figure 2(b) shows a simple structured grid interpreted as a cubical complex.

Morse-Smale complex representation: The Morse-Smale complex is represented as a graph whose nodes represent critical points of the Morse-Smale complex and edges represent arcs. Let M denote the Morse-Smale complex. Each critical point is identified by a unique integer identifier. M stores per-vertex information such as the function and index of the critical point. The adjacencies between critical points are maintained in a pair of lists, one for the ascending and one for the descending adjacencies. In 2D, each list consists of a list critical point identifiers. In 2D, each critical point may be incident upon another via at most two arcs. In 3D, however, there may be arbitrarily more. Hence, each entry in the list comprises of a tuple, where the first value identifies the incidence relation, and the second value identifies the multiplicity. The ascending and descending geometry of each critical point is maintained in a pair of lists. Each list consists of a list of cell identifiers which identify cells of the underlying domain. The right panel in Figure 2(c) shows a simple Morse-Smale complex representation with the adjacency and geometry data.

Algorithms

For computing the discrete gradient field, we use the algorithm by Robins *et al.* for *pymstri* as it results in the fewest spurious critical points. For *pyms3d*, we use the algorithm by Shivashankar *et al.* [38] as it is implementable in massively parallel architectures using GPUs. Though it results in more spurious critical points, the trade-off is acceptable in terms of the efficiency offered by GPUs. We use the algorithms described by Shivashankar *et al.* [38] for efficient traversal of the gradient field on the CPU and GPU. A full performance evaluation of the above algorithm for 2D and 3D structured grids is available by Shivashankar *et al.* [39, 38]. For the hierarchical Morse-Smale complex, we directly implement the cancellation and anti-cancellation operations described by Bremer *et al.* [7]. Cancellations may be scheduled based on topological persistence [13]. Alternatively, cancellations may be specified explicitly and performed sequentially via the Python interface. To traverse the combinatorial structure of the hierarchical Morse-Smale complex, the list of cancellation pairs are stored and the cancellation / anti-cancellation operations are repeatedly applied using this list to obtain the desired level in the hierarchy. We employ the *merge_dag* [22] data structure for efficient geometry queries from the hierarchical Morse-Smale complex.

4 Design and implementation

In this section, we briefly discuss design and implementation issues of the above data structures. The above described algorithms are implemented using C++. The code is extensively parallelized to exploit multi-core CPUs when available. This is done using the OpenMP framework, which has the advantage of requiring only compiler directives to concisely indicate shared memory parallel loops and sections. The discrete gradient algorithm as well as gradient field traversals for extrema in *pyms3d* are implemented using the OpenCL framework. The implementation selects the GPU when available. If a GPU is unavailable, it runs the OpenCL code on the CPU.

The C++ implementations are made available as Python modules using the Boost Python framework [3]. Both modules expose a Python class representing the Morse-Smale complex. The implementation makes extensive use of the NumPy[2] module to enable efficient passing of data arrays from C++ runtime objects to the Python environment. This module is easily available in most Python installations and its C++ bindings are made available by the Boost NumPy project [1]. Table 1 shows a subset of methods that are exposed to the Python Interface along with a brief description.

The implementations allow for computation of the Morse-Smale complex, querying for the combinatorial structure, querying for the ascending and descending manifolds of the critical points, generating a hierarchy (either by persistence or by a user defined sequence using the *cancel_pair* call), and querying the combinatorial and geometric structures at different levels of the hierarchy.

We now discuss the functions listed in Table 1. The interface can be broadly classified into three groups, namely computation functions, simplification, and query functions. The first two groups primarily alter the state of the Morse-Smale complex. Therefore, they involve minimal overhead in terms of interfacing with Python as they simply reroute the calls to C++ implementation. The third group generally involves data copying overheads via the Boost NumPy [2] array API. The first group comprises of the computation functions that compute the Morse-Smale complex and collect the geometry associated with its critical points. The calls to *compute_bin*, *compute_arr* and *compute_off*, detailed in Table 1, compute the combinatorial Morse-Smale complex. In many applications, either the combinatorial structure suffices or the geometry is desired after some pre-simplification. The call to *collect_geom* collects the Morse-Smale complex geometry at the current hierarchical version. Optionally, the user may choose to only collect the ascending or descending geometry of critical points with a given index.

The second group of functions may be used to simplify the Morse-Smale complex. The primary function here is the *cancel_pair* routine which cancels a given pair of critical points as long as the cancellation is permissible. Each cancellation results in a new hierarchical version of the Morse-Smale complex. Due to the popularity of the persistence hierarchy, the call to *simplify_pers* is provided to generate a persistence hierarchy so that arcs with a specified threshold are simplified. This call may also be supplied with a number of extrema that are desired to be retained. After some

simplification, one may obtain the earlier versions of the Morse-Smale complex using *set_hversion*. For instance, one may obtain the Morse-Smale complex prior to two simplifications by using the call *msc.set_hversion(msc. get_hversion()-2)*.

The third group of functions involve providing data pertaining to the Morse-Smale complex via the NumPy [2] array module. This is designed as a copy only mechanism for two reasons. First, NumPy is a very popular scientific computation API which eases further computations and interfacing with other tools, such as ParaView's Python API (see Listing 1 for example). Second, the memory management is eased by relinquishing the returned array objects to the Python runtime as opposed to maintaining the same in the C++ runtime. The list of critical points in the current hierarchical version, optionally of a given index d, may be obtained as an array of integer identifiers via the call to *cps([d])*. The list of ascending/descending critical points connected to a given critical point i may be obtained by the *asc(i)/des(i)* calls. In 3D, each entry in the returned list is a pair, where the first is the ID and the second is the multiplicity of the connection. In 2D, multiple entries are simply repeated as the multiplicity is at most two. The ascending/descending manifold of a given critical point i may be obtained using the *asc_geom(i)/des_geom(i)* calls. For a given critical point of index d, these methods return an array containing d-cells of gradient pairs that originate/terminate at i. By default, this set is constructed for the current hierarchical version of the Morse-Smale complex object. The ascending/descending manifold may also be obtained at a given hierarchical version n without altering the current hierarchical version by using the *merge_dag* data-structure. Note that non-empty geometry can only be returned if a prior call to *collect_geom* was made at a hierarchical version lower than n. For *pyms3d*, each cell is represented by a tuple of indices. In the case of ascending manifolds of 1-saddles and 2-saddles, the indices index into a list of vertex coordinates, which may be obtained by the call to *get_primal_points*. Analogously, for ascending manifolds of saddles, the indices index into a list of cube coordinates obtained via the call to *get_dual_points*. For minima, as their ascending manifolds form a partition on the set of vertices, the indices index into the list of vertex coordinates. Analogously, for maxima, the indices index into a list of cube coordinate, *i.e.* the cube centroids. For *pymstri*, an analogous output form is supported. Additionally, the indices discussed above respect the ordering of triangles and vertices given as the input to the computation. Other utility functions to query per-critical point information, such as the index and function value of critical points, as well as save/load the computed data from/to the file-system are also available.

5 Case study: Interactive Visual Feature analysis

In this section, we present a case study, where the Morse-Smale complex based analysis is coupled with the ParaView [26] visualization package. ParaView offers a Python programmable interface to its visualization pipeline. We apply the Morse-Smale complex to perform hierarchical segmentation of a simulation of the

Method	**Brief description**
compute_off(f)	Computes the mscomplex of a triangle mesh and scalar function given in a file f
compute_bin(f,s)	Computes the mscomplex of a structured grid. Scalar function is given as 32-bit floating point binary file f in fortran order. Grid size is given in the tuple s.
compute_arr(a)	Compute the mscomplex of a structured grid. Scalar function is given as 3d NumPy array a. Internally, data is converted to single precision (32-bit) floating point values.
collect_geom([d,dir])	Collects the [(dir =) Ascending/Descending] geometry of all [d-] critical points in the current hierarchical version.
get_hversion(n) / set_hversion(n)	Gets/Sets the hierarchical version of the Morse-Smale complex.
cancel_pair(p,q)	Cancels a pair of singularly connected unpaired critical points p and q.
simplify_pers(t)	Simplifies the Morse-Smale complex upto persistence threshold t.
cps([d])	Returns a list of ids of active critical points [with index d].
des(i) /asc(i)	Returns a list of descending/ascending critical points connected to i.
des_geom(i,[n]) / asc_geom(i,[n])	Returns descending/ascending manifold of the critical point i [in the nth hierarchical version].
get_primal_points(), get_dual_points()	Returns the coordinates of 0-dimensional cells of the primal/dual cell complex
cp_func(i), cp_index(i), *etc.*	Returns the per-critical point information such as the function value, pair
get_hversion_pers(t)	Returns hierarchy version number where pairs with persistence t are eliminated.
save(f)/load(f)	Save/load all data to/from file f

Table 1 A subset of the methods available to an **mscomplex** object created by the modules *pyms3d* and *pymstri*. Optional arguments to the methods are shown indicated in square braces ([...]).

hurricane Isabel. Hurricane Isabel was a hurricane that struck the coast of Florida, USA, in September 2003. The simulation dataset was made available by Wang *et al.*, for the 2004 IEEE Visalization contest [42]. The simulation is available as 32-bit floating point values on a $500 \times 500 \times 100$ 3D structured grid. This dataset is well understood in visualization literature and therefore helps illustrate the ease of feature analysis and visualization using the software. We begin by simply computing the Morse-Smale complex of the wind-speed field. Listing 1 shows code to generate the segmentation data using *pyms3d*. Figure 3 shows the volume segmentation using a persistence threshold of 0.05.

Next, we identify the highest finite-persistent minimum and segment its corresponding ascending manifold. This is done by simplifying using persistence till only two minima remain. Further simplification eliminates the desired minimum. Since the object is cached during runtime, it may be retrieved without re-computation for these operations as shown in Listing 2. Using the minimum representing the eye, the function value restricted to the ascending manifold of the desired minimum is generated.

The above listings are demonstrated in a video hosted at https://youtu.be/UX1q9gI2DEk, where they are plugged into ParaView's programmable filter. In particular, changes to the persistence threshold to visualize the eye at differ-

```
import pyms3d, numpy

X,Y,Z     = (500,500,100)
msc       = pyms3d.mscomplex()           # Create the object.
msc.compute_bin("Speed03.bin",(X,Y,Z)) # Compute from bin file.
msc.simplify_pers(thresh=0.05)         # Simplify.
msc.collect_geom(1,0)                  # Collect the asc geom
                                       # of 0 index cps.
ids = numpy.empty([X*Y*Z],numpy.int32) # Array to hold ids.
for m in msc.cps(0):                   # For each minima m,
  ids[msc.asc_geom(m)] = m             # set id's of vert's
                                       # in asc of m to m.
setattr(pyms3d,"msc",msc)              # cache the object.

# pass the ids array to ParaView (Code omitted for brevity).
```

Listing 1: Code for segmenting the Isabel dataset using the *pyms3d* module.

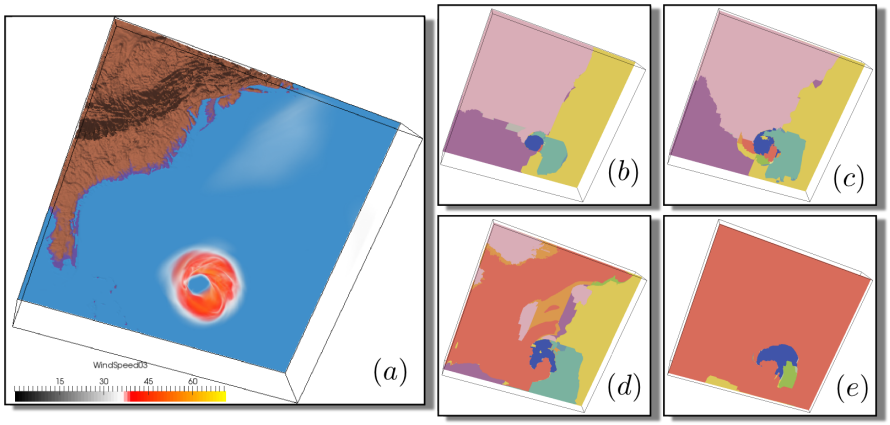


Fig. 3 Segmenting the wind speed field in 3^{rd} time-step of the Isabel simulation. (a) A volume visualization of the wind speed field. The eye is distinctly discernible as a low wind speed region enveloped by high wind speed regions. The height field representing the land and sea regions with appropriate colors is shown. (b),(c),(d), and (e) Segmentation of the scalar field at four equally spaced z-slices using a persistence threshold of 0.05, generated using Listing 1. The distinctive structure of the eye is retained in the lower z-slices (b) and (c), which is less discernible in the higher slices (d) and (e).

ent hierarchical versions is demonstrated. These modifications are made at runtime and the visualization updates occur interactively. The video demonstrates the execution of the above listings on a HP xw8600 workstation with 8 CPU cores, 8GB RAM, and Nvidia 260 GPU that has 895MB VRAM. The time taken to compute the Morse-Smale complex and generate the visualization using Listing 1 is approximately 30 seconds. The time to update the visualizations using Listing 2 is under

```
import pyms3d, numpy

msc = getattr(pyms3d,"msc")    # get the cached msc object.

if not hasattr(pyms3d,"eye"): # if 'eye' is not in cache
  msc.simplify_pers(nmin=2)   # simplify until 2 minima survive
  m1,m2 = msc.cps(0)          # get the 2 minima
  msc.simplifiy_pers(nmin=1)  # simplify the penultimate minimum
  m,    = msc.cps(0)          # get the surviving minimum
  e = (m1 if m == m2 else m2) # the required minimum is the other
  setattr(pyms3d,"eye",e)     # cache the crit. pt. id of eye.

e = getattr(pyms3d,"eye")     # get crit. pt. id of eye

v = msc.get_hversion_pers(thresh=0.018)  # change hier. version
msc.set_hversion(v)

ag = msc.asc_geom(e)          # id's of verts in e's asc mfold

fns = numpy.fromfile("Speed03.bin",numpy.float32) #read scalars

ag_fn  = fns[ag]              # save func. values at eye
fns[:] = -1                   # set value everywhere to -1
fns[ag] = ag_fn               # set scalar value only at eye.

# pass the fns array to ParaView (Code omitted for brevity).
```

Listing 2: Code for extracting the id of the minimum representing the eye of the hurricane from the Morse-Smale complex object computed and cached in Listing 1. The corresponding ascending manifold region is segmented and the scalar values within this region is volume rendered in Figure 3.

1 second. A more detailed study of the performance of the efficient computation algorithms is presented by Shivashankar *et al.*[39, 38].

Due to the efficient computation, as well as the interactive visual analysis, Listing 2 was used for multiple time steps. Figure 4 shows the wind-speed restricted to the eye using this simultaneous visualization setup. The transfer function is chosen to highlight the low speeds within the eye.

We observe the performance of a few of the crucial methods over 50 executions of Listing 1. The mean execution time of *compute_bin* with OpenCL on the GPU is 12.99s (std[1]=0.1s). The same method deployed on the 8-core CPU has a mean execution time of 52.31s (std=0.916s). For both invocations time taken for the call to transition from Python to the C++ runtime had a mean of $2.28 \times 10^{-3}s$ (std=$1.17 \times 10^{-4}s$). The transition time for other function calls had similar timings and hence we do not consider it to be a performance issue. The call to *collect_geom* has a mean time of 1.45s (std=0.082s) with the GPU deployment and

[1] The standard deviation is abbreviated as *std*

3.06 (std=0.052s) with the CPU deployment. The total time taken for all calls to *asc_geom* in Listing 1 had a mean of 0.46s (std=0.013s) over 50 executions. The method has three steps. First, the geometry representing the ascending manifold at a given hierarchy is computed using the *merge_dag*. Second, the data is converted from cell identifiers to vertex indices. Third, a NumPy array is allocated, the data is copied to it, and returned. The mean (and variance) for each of the above steps is 0.34s (std=0.0027), 0.12s (std=0.0113), and 0.0013s (std= 4.65×10^{-5}). From the above timings, we conclude that both the Python call transfer overhead as well as data copy overheads are negligible compared to the method timings. Also, the comparison of the CPU and GPU timings reaffirm the earlier experimental results [39, 38].

We also profile the memory usage of Listing 1 using both the CPU and GPU deployments. We super-sample the dataset using bilinear interpolation to generate a sequence of datasets, where the number of points along each dimension are increased by 10%, 20%, upto 100%. In the CPU deployment on the machine described above, we observe that the maximum physical memory used is 6 GB for a $1000 \times 1000 \times 200$ sized version of the dataset. In the GPU deployment, as there is no virtual memory provision, we observe failures in memory allocation for a dataset sized $550 \times 550 \times 110$. For this dataset, we require a byte buffer of size $1099 \times 1099 \times 219$ (252MB) to store the gradient information of all cells. The GPU device used does not allow individual buffer sizes beyond 224MB, even though the GPU has a 895MB VRAM. A detailed analysis of this experiment is available in the project website.

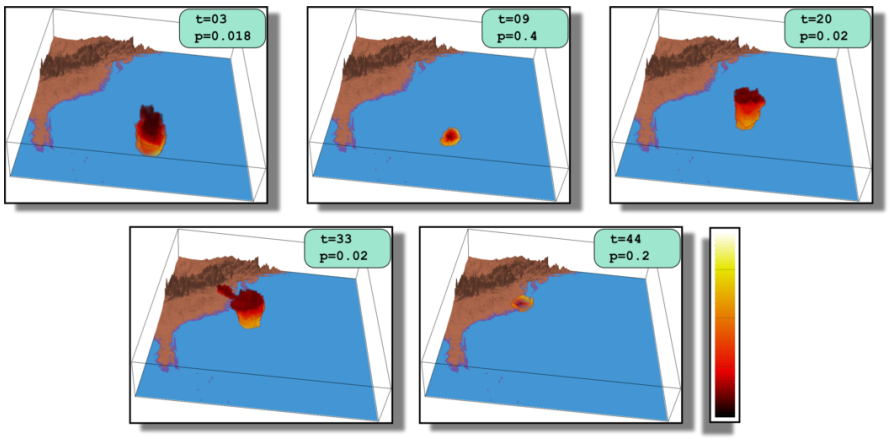


Fig. 4 Visualizations of multiple time-steps of the eye of the Isabel simulation segmented using the Listing 2. Due to the efficient implementation of computation and filtering, the above Listings can be used for multiple time-steps in the same instance of ParaView for interactive visual identification of appropriate persistence thresholds.

6 Conclusions

In this paper, we have presented an efficient implementation of Morse-Smale complex based algorithms for visual analysis. We demonstrated its versatility as a computation and a visual analysis tool by plugging into ParaView to generate visualizations of the Hurricane Isabel dataset. This implementation includes state of the art algorithms for computation of Morse-Smale complexes as well as implementations of algorithms for hierarchical analysis.

Acknowledgements: This work was partially supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/0086/2012 and by the Robert Bosch Centre for Cyber Physical Systems, Indian Institute of Science.

References

1. Boost Python interface for NumPy. https://github.com/ndarray/Boost.NumPy
2. NumPy & SciPy: Scientific computing in python. http://www.numpy.org/
3. Abrahams, D., Grosse-Kunstleve, R.W.: Building Hybrid Systems with Boost.Python (2003)
4. Bauer, U., Kerber, M., Reininghaus, J.: Distributed Computation of Persistent Homology. In: Proc. Algorithm Engineering and Experiments (ALENEX), pp. 31–38 (2014)
5. Bauer, U., Kerber, M., Reininghaus, J., Wagner, H.: Phat persistent homology algorithms toolbox. In: H. Hong, C. Yap (eds.) Mathematical Software ICMS 2014, *Lecture Notes in Computer Science*, vol. 8592, pp. 137–143. Springer Berlin Heidelberg (2014)
6. Blender Online Community: Blender - a 3D modelling and rendering package. Blender Foundation, Blender Institute, Amsterdam (2014)
7. Bremer, P.T., Edelsbrunner, H., Hamann, B., Pascucci, V.: A topological hierarchy for functions on triangulated surfaces. IEEE Trans. Vis. Comp. Graphics **10**(4), 385–396 (2004)
8. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. Computational Geometry **24**(2), 75 – 94 (2003)
9. Cazals, F., Chazal, F., Lewiner, T.: Molecular shape analysis based upon the Morse-Smale complex and the Connolly function. In: Proc. 19th Ann. ACM Sympos. Comput. Geom., pp. 351–360 (2003)
10. Doraiswamy, H., Natarajan, V.: Computing Reeb graphs as a union of contour trees. IEEE Trans. Vis. Comp. Graphics **19** (2013)
11. Edelsbrunner, H., Harer, J., Natarajan, V., Pascucci., V.: Morse-Smale complexes for piecewise linear 3-manifolds. In: Proc. 19th Ann. Sympos. Comput. Geom., pp. 361–370 (2003)
12. Edelsbrunner, H., Harer, J., Zomorodian., A.: Hierarchical Morse-Smale complexes for piecewise linear 2-manifolds. Discrete and Computational Geometry **30**(1), 87–107 (2003)
13. Edelsbrunner, H., Letscher, D., Zomorodian., A.: Topological persistence and simplification. Discrete and Computational Geometry **28**(4), 511–533 (2002)
14. Forman, R.: A user's guide to discrete Morse theory. Séminaire Lotharingien de Combinatoire **48** (2002)
15. Günther, D., Boto, R.A., Contreras-Garcia, J., Piquemal, J.P., Tierny, J.: Characterizing Molecular Interactions in Chemical Systems. IEEE Trans. Vis. Comp. Graphics **20**(12) (2014)
16. Günther, D., Jacobson, A., Reininghaus, J., Seidel, H.P., Sorkine-Hornung, O., Weinkauf, T.: Fast and Memory-Efficient Topological Denoising of 2D and 3D Scalar Fields. IEEE Trans. Vis. Comp. Graphics **20**(12) (2014)
17. Günther, D., Reininghaus, J., Wagner, H., Hotz, I.: Memory Efficient Computation of Persistent Homology for 3D Image Data using Discrete Morse Theory. In: Conference on Graphics, Patterns and Images, 24. (SIBGRAPI), pp. 25–32 (2011)

18. Gyulassy, A., Bremer, P., Pascucci, V.: Computing Morse-Smale Complexes with Accurate Geometry. IEEE Trans. Vis. Comp. Graphics **18**(12), 2014–2022 (2012)
19. Gyulassy, A., Bremer, P.T., Pascucci, V., Hamann, B.: A practical approach to Morse-Smale complex computation: scalability and generality. IEEE Trans. Vis. Comp. Graphics **14**(6), 1619–1626 (2008)
20. Gyulassy, A., Duchaineau, M., Natarajan, V., Pascucci, V., Bringa, E., Higginbotham, A., Hamann, B.: Topologically clean distance fields. IEEE Trans. Vis. Comp. Graphics **13**(6), 1432–1439 (2007)
21. Gyulassy, A., Günther, D., Levine, J.A., Tierny, J., Pascucci, V.: Conforming Morse-Smale complexes. IEEE Trans. Vis. Comp. Graphics **20**(12) (2014)
22. Gyulassy, A., Kotava, N., Kim, M., Hansen, C., Hagen, H., Bremer, P.T.: Direct feature visualization using Morse-Smale complexes. IEEE Trans. Vis. & Comp. Graphics **18**(9), 1549–1562 (2012)
23. Gyulassy, A., Natarajan, V., Pascucci, V., Bremer, P.T., Hamann, B.: Topology-based simplification for feature extraction from 3d scalar fields. In: Proc. IEEE Conf. Visualization, pp. 535–542 (2005)
24. Gyulassy, A., Natarajan, V., Pascucci, V., Bremer, P.T., Hamann, B.: A topological approach to simplification of three-dimensional scalar fields. IEEE Trans. Vis. Comp. Graphics **12**(4), 474–484 (2006)
25. Gyulassy, A., Natarajan, V., Pascucci, V., Hamann, B.: Efficient computation of Morse-Smale complexes for three-dimensional scalar functions. IEEE Trans. Vis. Comp. Graphics **13**(6), 1440–1447 (2007)
26. Henderson, A.: ParaView Guide, A Parallel Visualization Application (2007)
27. Humphrey, W., Dalke, A., Schulten, K.: VMD – Visual Molecular Dynamics. Journal of Molecular Graphics **14**, 33–38 (1996)
28. Kettner, L.: CGAL HalfEdge Data-Structure: User Manuel. http://doc.cgal.org/latest/HalfedgeDS/index.html
29. Laney, D., Bremer, P.T., Mascarenhas, A., Miller, P., Pascucci, V.: Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. IEEE Trans. Vis. Comp. Graphics **12**(5), 1053–1060 (2006)
30. Matsumoto, Y.: An Introduction to Morse Theory. Amer. Math. Soc. (2002). Translated from Japanese by K. Hudson and M. Saito
31. Natarajan, V., Wang, Y., Bremer, P.T., Pascucci, V., Hamann, B.: Segmenting molecular surfaces. Comput. Aided Geom. Des. **23**(6) (2006)
32. Peterka, T., Ross, R., Gyulassy, A., Pascucci, V., Kendall, W., Shen, H.W., Lee, T.Y., Chaudhuri, A.: Scalable parallel building blocks for custom data analysis. In: Proc. IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV) (2011)
33. Pettersen, E.F., et al.: UCSF Chimera–a visualization system for exploratory research and analysis. J. Comp. Chem. (13), 1605–1612 (2004)
34. Robins, V., Wood, P.J., Sheppard, A.P.: Theory and algorithms for constructing discrete Morse complexes from grayscale digital images. IEEE Trans. Pattern Analysis and Machine Intelligence **99** (2011)
35. van Rossum, G.: Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1995)
36. Schrödinger, LLC: The PyMOL molecular graphics system, version 1.3r1 (2010)
37. Schroeder, W., et al.: The visualization toolkit, 3rd edition (2003)
38. Shivashankar, N., Natarajan, V.: Parallel computation of 3D Morse-Smale complexes. Computer Graphics Forum **31**(3pt1), 965–974 (2012)
39. Shivashankar, N., Senthilnathan, M., Natarajan, V.: Parallel computation of 2D Morse-Smale complexes. IEEE Trans. Vis. & Comp. Graphics **18**(10), 1757–1770 (2012)
40. Smale, S.: On gradient dynamical systems. Ann. of Math. **74**, 199–206 (1961)
41. Sousbie, T.: The persistent cosmic web and its filamentary structure I. Theory and implementation. MNRAS **414**(1), 350–383 (2011)
42. Wang, W., Bruyere, C., Kuo, B.: Competition data set and description in 2004 IEEE Visualization design contest. http://vis.computer.org/vis2004contest/data.html (2004)