AMERICAN UNIVERSITY

WASHINGTON, DC

Sheaf Invariants for Information Systems

Michael Robinson

2012

Technical Report No. 2014-2

https://doi.org/10.17606/m0nx-h357

Final Report for AFOSR MURI on Complex Networks

Federal Contract No. FA9550-09-1-0643

Sheaf Invariants for Information Systems

Principal Investigator:
Prof. Michael Robinson
Assistant Professor
Mathematics and Statistics
American University
Office: (202)885-3681
Mobile: (484)477-3345
michaelr@american.edu

Program Manager:
Dr. Robert Bonneau
AFOSR/RSL
Office: (703) 696-9545
Fax: (703) 696-7360
Secure: 426-9545
robert.bonneau@afosr.af.mil

American University

Contents

1. Executive summary

The primary objective of this project was to *construct, classify, and exploit invariants for discriminating information systems* that are based on abstracted structural descriptions. This main objective was split into three smaller objectives:

(1) Construct invariants for information systems that exploit coarse and multiscale structural specifications about their underlying network or communication topology,
(2) Classify the semantic and dynamic features of the systems that these invariants consider, and
(3) Exploit the classification results to provide actionable design and analysis rules that can be incorporated into experimental and simulation workflows.

All of these objectives were met. Several interesting (and potentially important) discoveries were made as a result of the project. These discoveries have been reported to the scientific community, and they are being written as articles for archival journals. In addition, the Principal Investigator (PI), Prof. Michael Robinson, completed a draft of a manuscript entitled *Topological Signal Processing*

that can be used to teach the techniques discovered on this project to beginning graduate student researchers. Finally, Prof. Robinson's research group grew from one student at the start of the project to five students (partially funded by this project), partially as a means to apply the new algorithmic techniques discovered on this program.

2. Research progress

This project discovered several new classes of sheaves and morphisms that are relevant to information systems, and developed algorithms that can be applied to simulated data or data collected by laboratory systems.

2.1. **Objective 1: Novel invariants. Status: Success!**

The project began by revisiting transmission line sheaves [9] and switching sheaves [8], since these classes of sheaves are concrete and have proven value in applications. By studying how these classes of sheaves are used as models of physical phenomena, we discovered new sheaf models for other applications. Over the course of the project, we discovered

(1) Sheaves of timeseries, images, and video streams [7],
(2) Flow and concentration sheaves [7],
(3) Sheaves of bandlimited signals [6],
(4) Sheaves of piecewise linear functions and Taylor series [6].

2.2. **Objective 2: Semantic and dynamic features. Status: Success! As a result, many new avenues of research have been opened.**

Since *sheaf morphisms* are the primary tool for studying the relationships between sheaves, we began to study how sheaf models for information systems could be related to one another.

We discovered three classes of engineering-relevant morphisms between sheaves:

(1) Pairs of morphisms that describe discrete, linear translation-invariant systems [7],
(2) Sampling morphisms that describe measurement procedures [6], and
(3) The number of morphisms between pairs of switching sheaves can detect semantic differences between logically-equivalent circuits.

Because discrete, linear translation-invariant systems have now been placed on a sheaf-theoretic footing, the greater generality afforded by sheaves allows us to consider substantially broaders classes of filters. We have uncovered a number of interesting directions for future study, including a class of nonlinear, angle-sensitive filters.

The discovery of the correct definition of sampling in terms of sheaf morphisms opens up the possibility of a general sheaf-theoretic treatment of measurement processes and inference algorithms. Although continuing research in this direction is ambitious, we recommend studying sampling in general sheaf contexts. Success in this direction could lead to a Shannon-Nyquist-like theorem for networks of sensors collecting multi-modal data.

We found that switching sheaf morphisms characterize semantic similarity, and our discovery initiated the study of the combinatorics of switching sheaf morphisms. In particular, we believe enumeration and interpretation of sheaf morphisms is now an interesting open problem about which very little is known.

2.3. **Objective 3: Exploitation of the theory. Status: Success! As a result, many new avenues of research have been opened.**

The theoretical progress made on this project directly led to the development of novel algorithms for processing data. We tested these algorithms on a mixture of simulation and laboratory data. Specifically, we

(1) Invented a class of novel filtering algorithms, and performed initial tests on image datasets,
(2) Developed a library (written in the Python language; listing included as an appendix of this report) for computing sheaf theoretic invariants, such as cohomology and induced maps on global sections,
(3) Enumerated the complete list of morphisms between two switching sheaves,
(4) Implemented and tested a geometry extraction algorithm (developed on a previous effort) for transmission line sheaves,
(5) Developed and tested a new differential-topological algorithm to analyze the sonar echos of a rotating target [5], and compared its output to a prexisting persistent cohomology algorithm [3].

3. Research narrative

3.1. **Background material.** We include here a brief introduction to the relevant sheaf theory necessary to understand our progress on this project. The reader is encouraged to see [2] for a more extensive treatment.

3.1.1. *Sheaves.* A sheaf is a mathematical object that stores locally-defined data over a space. In order to formalize this concept, we need a concept of space that is convenient for computations. The most efficient such definition is that of a simplicial complex.

Definition 1. An *abstract simplicial complex* over a set A is a collection X of (possibly ordered) subsets of A, for which $x \in X$ implies that every subset of x is also in X. We call each $x \in X$ with $k+1$ elements a *k-face* of X, referring to the number k as its *dimension*. Zero dimensional faces (singleton subsets of A) are called *vertices*, and one dimensional faces are called *edges*. We say that a face a *includes into* a face b (written $a \rightsquigarrow b$) whenever a is a proper subset of b.

Although sheaves have been extensively studied over topological spaces (see [1] or the appendix of [4]), the resulting definition is ill-suited for application to sampling. Instead, we follow a substantially more combinatorial approach introduced in the 1985 thesis of Shepard [11].

Definition 2. A *sheaf* $\mathcal{F}$ on an abstract simplicial complex X is a covariant functor from the face category of X to the category of vector spaces. Explicitly,

- for each element a of X, $\mathcal{F}(a)$ is a vector space, called the *stalk at a*,
- for each inclusion of two faces $a \rightsquigarrow b$ of X, $\mathcal{F}(a \rightsquigarrow b)$ is a linear function from $\mathcal{F}(a) \to \mathcal{F}(b)$ called a *restriction*, and
- for every composition of inclusions $a \rightsquigarrow b \rightsquigarrow c$, $\mathcal{F}(b \rightsquigarrow c) \circ \mathcal{F}(a \rightsquigarrow b) = \mathcal{F}(a \rightsquigarrow b \rightsquigarrow c)$.

Definition 3. Suppose $\mathcal{F}$ is a sheaf on an abstract simplicial complex X and that $\mathcal{U}$ is a collection of faces of X. An assignment s which assigns an element of $\mathcal{F}(u)$ to each face $u \in \mathcal{U}$ is called a *section* supported on $\mathcal{U}$ when for each inclusion $a \rightsquigarrow b$ (in

X) of objects in $\mathcal{U}$, $\mathcal{F}(a \rightsquigarrow b)s(a) = s(b)$. A *global section* is a section supported on X. If r and s are sections supported on $\mathcal{U} \subset \mathcal{V}$, respectively, in which $r(a) = s(a)$ for each $a \in \mathcal{U}$ we say that s *extends* r. The collection of sections supported on a given set forms a vector space.

Definition 4. A *sheaf morphism* is a natural transformation between sheaves. Explicitly, a morphism $f : \mathcal{F} \to \mathcal{G}$ of sheaves on an abstract simplicial complex X assigns a linear map $f_a : \mathcal{F}(a) \to \mathcal{G}(a)$ to each face a so that for every inclusion $a \rightsquigarrow b$ in the face category of X, $f_b \circ \mathcal{F}(a \rightsquigarrow b) = \mathcal{G}(a \rightsquigarrow b) \circ f_a$.

3.2. **Sheaf cohomology.** Much of the theory of sheaves is concerned with computing spaces of sections and identifying obstructions to extending sections. The machinery of cohomology systematizes the computation of the space of global sections for a sheaf.

Recall that an abstract simplicial complex X consists of *ordered* sets. For a a k-face and b a $k+1$-face, define

$$[b : a] = \begin{cases} +1 & \text{if the order of elements in } a \text{ and } b \text{ agrees,} \\ -1 & \text{if it disagrees, or} \\ 0 & \text{if } a \text{ is not a face of } b. \end{cases}$$

Define the following formal *cochain* vector spaces $C^k(X;\mathcal{F}) = \bigoplus_{a \text{ a } k\text{-face of } X} \mathcal{F}(a)$. The *coboundary map* $d^k : C^k(X;\mathcal{F}) \to C^{k+1}(X;\mathcal{F})$ takes an assignment s from the k faces to an assignment $d^k s$ whose value at a $k+1$ face b is

$$(d^k s)(b) = \sum_{a \text{ a } k\text{-face of} X} [b : a]\mathcal{F}(a \rightsquigarrow b)s(a).$$

It can be shown that $d^k \circ d^{k-1} = 0$, so that the image of d^{k-1} is a subspace of the kernel of d^k.

Definition 5. The k-th *sheaf cohomology* of $\mathcal{F}$ on an abstract simplicial complex X is

$$H^k(X;\mathcal{F}) = \ker d^k / \text{image } d^{k-1}.$$

Observe that $H^0(X;\mathcal{F}) = \ker d^0$ consists precisely of those assignments s which are global sections. Cohomology is also a functor: sheaf morphisms induce linear functions between cohomologies. This indicates that cohomology preserves and reflects the underlying relationships between sheaves.

3.2.1. *Transmission line sheaves.* One of the first places where signal processing got its start was in the transmission of signals along telegraph wires. In the context of wave propagation on a graph, the space of solutions forms a sheaf. This even holds if the propagation is lossy, or if we instead consider fundamental solutions, in which there are a number of sources [9]. This sheaf has a convenient cellular structure, as explained in the following definition.

Definition 6. Suppose X is a 1-dimensional cell complex X whose edges e are labeled by (1) a nonnegative real function L (called the *length*) and (2) an arbitrary direction. The *transmission line sheaf* $\mathcal{T}$ with wavenumber k is given by

(1) $\mathcal{T}(v) = \mathbb{C}^{\deg v}$ for all vertices v,
(2) $\mathcal{T}(e) = \mathbb{C}^2$ for all edges e, and

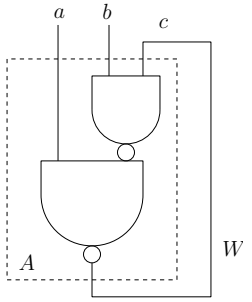


FIGURE 1. An R-S flip-flop circuit

(3) If e_m is the m-th edge attached to a degree n vertex v,

$$\mathcal{T}(v \rightsquigarrow e_m)(u_1, \ldots, u_n) = \begin{cases} \left(u_m, e^{-ikL(e_m)} \left(\frac{2}{n} \sum_{j=1}^{n} u_j - u_m\right)\right) & \text{if } e_m \text{ is inward at } v \\ \left(e^{ikL(e_m)} \left(\frac{2}{n} \sum_{j=1}^{n} u_j - u_m\right), u_m\right) & \text{if } e_m \text{ is outward at } v \end{cases}$$

3.2.2. *Switching sheaves.* Switching sheaves provided the starting point for this project, as a concrete example of a sheaf for an information system.

A switching sheaf is an algebraic-topological model of an asynchronous circuit. In order to define what a switching sheaf is, we need some preliminary definitions. Suppose that X is an oriented CW complex. If c, d are cells of X such that $c \in \partial d$ and $\dim d = \dim c + 1$, then c is called a *face* of d, written $c \rightsquigarrow d$. We call d a *coface* of c. If c and d agree about orientation, we call them *co-oriented*.

Suppose $\mathcal{Q}$ is a sheaf on an oriented cell complex X. We will say that $\mathcal{Q}$ is a *quiescent switching sheaf* based on a set A if for all cells $c \in X$,

(1) the stalk $\mathcal{Q}(c) = A$ if c has no co-oriented cofaces,
(2) otherwise, the stalk $\mathcal{Q}(c) = \prod_d \mathcal{Q}(d)$ where d ranges over the co-oriented cofaces of c, and
(3) the restriction $\mathcal{Q}(c) \to \mathcal{Q}(e)$ function is the projection onto the factor in the product $\mathcal{Q}(c) = \prod_d \mathcal{Q}(d)$ associated to e.

Unfortunately, quiescent switching sheaves are not generally sheaves of abelian groups. As a result, we cannot compute their cohomology and their analysis is subject to a state explosion. We correct for both problems by encoding the values of A in an $\mathbb{F}$-vector space. Consider the function $T : A \to \mathbb{F} \otimes A$, given by the inclusion $x \mapsto 1 \otimes x$. This T lifts the restrictions to $\mathbb{F}$-linear maps. Applying this idea in our definition of quiescent switching sheaves corresponds to a particular *categorification*.

We therefore define a *switching sheaf* to be a cellular sheaf that is the categorification via T of a quiescent switching sheaf. This essentially amounts to rewriting the definition of quiescent switching sheaves by replacing the Cartesian product with a tensor product, and the restriction maps become contractions instead of projections.

As an example, consider the circuit X shown in Figure 1, which is a basic memory element. Computation of the cohomology of the associated switching sheaf $\mathcal{S}$ shows that $H^0(X; \mathcal{S})$ has dimension 7 and $H^1(X; \mathcal{S})$ has dimension 1. Here is a basis for $H^0(X; \mathcal{S})$:

Element of $H^0(X;\mathcal{S})$	Description
$\overline{a}\otimes\overline{b}\otimes c$	Danger
$\overline{a}\otimes b\otimes c$	Set
$a\otimes\overline{b}\otimes\overline{c}$	Reset
$a\otimes b\otimes\overline{c}$	Hold zero
$a\otimes b\otimes c$	Hold one
$\overline{a}\otimes\overline{b}\otimes\overline{c}+a\otimes\overline{b}\otimes c$	Transition Danger to Reset
$\overline{a}\otimes\overline{b}\otimes\overline{c}+\overline{a}\otimes b\otimes\overline{c}$	Transition Danger to Set

The first five basis elements should be familiar to any electrical engineer, since they are the quiescient logic states of the circuit [10]. Therefore, the last two basis elements are of most interest. These are linear combinations of two terms, neither of which is a T-lift of a section of $\mathcal{Q}$. The most suggestive interpretation is that they imply an uncertainty when exiting the Danger state. As the inputs a and b transition from both logic 0 to both logic 1, there is a race condition. Only one of them transitions first, so there is a brief transition into the Set or Reset states before entering a Hold state. If we add the last two basis elements, we obtain $a\otimes\overline{b}\otimes c+\overline{a}\otimes b\otimes\overline{c}$ which indicates that an uncertainty about which of a or b transitions has occured results in uncertainty in the signal c.

It is clear from the example that *switching sheaves contain strictly more information than simply the logic states of a circuit*, which are encoded as sections of $\mathcal{Q}$. Using the Mayer-Vietoris principle [1] for cellular sheaves, it is possible to relate the cohomology of a circuit composed of simpler circuits to the cohomology of each of these subcircuits.

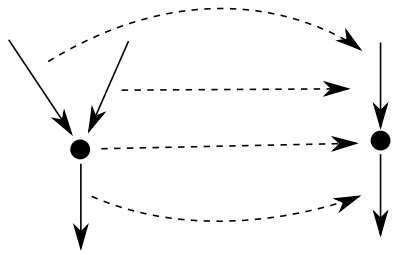
Theorem 7. [8] *The effect of attaching a wire is best described by the following slogan:*

- *Attaching a wire that does not participate in feedback suppresses logic states and leaves H^1 unchanged.*
- *Attaching a wire that participates in feedback leaves logic states unchanged and adds to the dimension of H^1.*

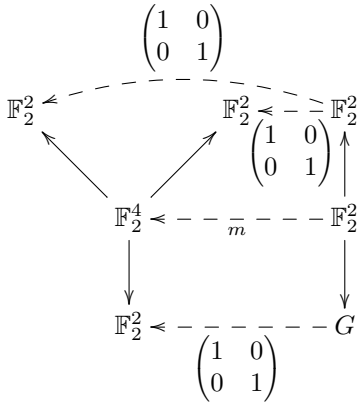
As an immediate corollary, *nontrivial H^1 of a switching sheaf detects race conditions.*

3.3. **Discoveries related to switching sheaves.** This section addresses Objective 2, because the morphisms discovered capture semantic relations between logic circuits.

Early in the project, we discovered examples of sheaf morphisms between switching sheaves. For instance, consider the cellular map of two directed graphs, given by the dashed arrows in the diagram



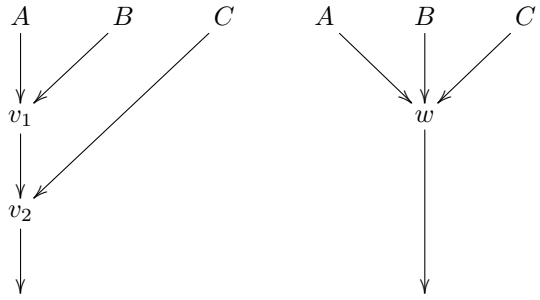
The following diagram shows a switching sheaf morphism over this map



where m is either $\begin{pmatrix}1 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 1\end{pmatrix}$ or $\begin{pmatrix}0 & 1\\ 1 & 1\\ 1 & 1\\ 1 & 0\end{pmatrix}$. Observe that each square in the diagram above commutes, and that the arrows are reversed with respect to the cellular map. Both of these are defining features of a sheaf morphism.

This morphism induces a homomorphism on the sheaf cohomologies, which explains how consistent logic states of the simplified circuit diagram (on the right) are represented in the diagram with more detail (on the left).

As a step in the direction of understanding and generalizing this result, we tabulated the number of sheaf morphisms between two switching sheaves. For concreteness, we considered the case of switching sheaves over the connection diagrams



At each of the vertices in the connection graphs, a logic gate must be specified. To ensure consistency, we constructed switching sheaves over each graph so that the two logic circuits had the same truth table. As a result, the quiescent switching sheaves were isomorphic, though the switchig sheaves are typically not isomorphic. We iterated over each of the 256 ($2 \times 2 \times 2$) possible pairs of 2-input logic gates and computed the number of morphisms that connect the two resultant switching sheaves. Table 1 lists the frequencies of morphism counts we found.

We found that the like the cohomology of the switching sheaf, the number of switching sheaf morphisms is more sensitive to semantic differences than the truth table of a logic circuit. Specifically, *there exist pairs of circuits all with the same truth table that do not have the same number of switching sheaf morphisms between*

TABLE 1. Number of morphisms between switching sheaves over chained 2-input gates to a single 3-input gate

Number of morphisms	Frequency
0	96
1	96
2	60
3	0
4	4

them. For instance, consider the case where v_2 is the 2-input zero function. There are 2 morphisms when v_1 is also 2-input zero function and w is the 3-input zero function. However, if v_1 is changed to be an AND gate, then there are no morphisms between the two circuits.

At the time of writing, the number of morphisms between pairs of logically equivalent switching sheaves remains rather mysterious. It remains an open problem to characterize the number of morphisms between two switching sheaves in terms of their semantic properties. In the case above, there are four switching sheaf pairs that have 4 morphisms between them, all of which are insensitive to inputs A and B. However, many other circuit pairs that are insensitive to A and B have a different number of sheaf morphisms between them.

Although apparently plentiful, it appears that sheaf morphisms of the kind just exhibited are perhaps not general enough. Instead one could require that a diagram like

$$\mathcal{R} \longleftarrow \mathcal{S} \longrightarrow \mathcal{T}$$

be used, in which one of the two morphisms induces isomorphisms on cohomology. Although abstract, this *derived category* construction can be used to realize all discrete linear translation invariant systems as discussed in the next section.

3.4. **Discoveries related to filter theory.** Our discoveries in filter theory address

- Objective 1: By providing novel models of signals within information systems, and
- Objective 2: By modeling the dynamics of these signals within signal processing hardware and software.

The primary finding was a constructive theorem [7] that encodes discrete, linear translation invariant filters with finite impulse response (FIR LTI) as a pair of sheaf morphisms.

Theorem 8. *Every FIR LTI filter F arises as the composition of linear maps $F = \lambda_* \circ p_*^{-1} : \Gamma\mathcal{S}_1 \to \Gamma\mathcal{S}_3$ induced on global sections by a pair of sheaf morphisms*

$$\mathcal{S}_1 \xleftarrow{p} \mathcal{S}_2 \xrightarrow{\lambda} \mathcal{S}_3.$$

In this diagram, the invertible linear map $p_* : \Gamma\mathcal{S}_2 \to \Gamma\mathcal{S}_1$ is induced by p, and the map induced by λ is $\lambda_* : \Gamma\mathcal{S}_2 \to \Gamma\mathcal{S}_3$.

This theorem has a clear interpretation in terms of the typical implementation of a filter, either in hardware or software. The global sections of $\mathcal{S}_1$ are precisely

the possible input sequences, the global sections of $\mathcal{S}_3$ correspond to the output sequences, and the global sections of $\mathcal{S}_2$ correspond to the contents of the internal storage of the filter. The proof of the theorem is a rather explicit construction, which outlines the evolution of these three timeseries.

The benefit of using sheaf morphisms to describe filters is that they can treat a number of additional cases. In addition to the novel algorithms we discovered, each of the following are straightforward generalizations, requiring no additional theoretical work to construct:

(1) Infinite impulse response filters can be constructed simply by extending the definition of $\mathcal{V}^{(N)}$ to treat spaces of sequences instead of finite-dimensional vectors.
(2) Nonlinear, block processing filters can be constructed by modifying the component maps of the morphism λ to be nonlinear functions. For instance, constant false alarm rate (CFAR) detectors can be encoded in this way. We discovered a nonlinear, angle-sensitive filter using this framework (Section 3.6.1).
(3) If G is a finitely-generated group that acts on X, then G-equivariant simplicial maps can be used to generalize $\mathcal{V}^{(N)}$ to other simplicial complexes X. This permits extensions of Theorem 8 to treat images, video, and more complex discrete datasets.

3.5. **Discoveries related to sampling theory.** Our sampling theory discoveries address

- Objective 1: By providing a concrete model of the unknown quantity being measured and the measurements obtained and relating the two through a *sampling morphism*, and
- Objective 2: By way of a theorem that explains whether inferences drawn from the measurements will suceed.

Suppose that $\mathcal{F}$ is a sheaf on an abstract simplicial complex X, and that $\mathcal{S}$ is the grouping sheaf $\mathcal{V}^{(1)}$ on X supported on a closed subcomplex Y. A *sampling* of $\mathcal{F}$ is a morphism $s : \mathcal{F} \to \mathcal{S}$ that is surjective on every stalk. Given a sampling, we can construct the *ambiguity sheaf* $\mathcal{A}$ in which the stalk $\mathcal{A}(a)$ for a face $a \in X$ is given by the kernel of the map $\mathcal{F}(a) \to \mathcal{S}(a)$. If $a \rightsquigarrow b$ is an inclusion of faces in X, then $\mathcal{A}(a \rightsquigarrow b)$ is $\mathcal{F}(a \rightsquigarrow b)$ restricted to $\mathcal{A}(a)$. This implies that

$$0 \to \mathcal{A} \hookrightarrow \mathcal{F} \xrightarrow{s} \mathcal{S} \to 0$$

is an exact sequence, which induces the long exact sequence (via the Snake lemma)

$$0 \to H^0(X;\mathcal{A}) \to H^0(X;\mathcal{F}) \to H^0(X;\mathcal{S}) \to H^1(X;\mathcal{A}) \to \cdots$$

An immediate consequence is therefore

Theorem 9. *(Sheaf-theoretic Nyquist theorem) The global sections of $\mathcal{F}$ are identical with the global sections of $\mathcal{S}$ if and only if $H^k(X;\mathcal{A}) = 0$ for $k = 0$ and 1.*

Several applications to specific sheaves were constructed during the project:

(1) The usual proof the Shannon-Nyquist sampling theorem can be reinterpreted as a special case [6],

FIGURE 2. An image (482×653 pixels) with curved striations (left) and its LSRA filtered image (right), in which the colors represent angle in degrees. The filter used a block size of 30 pixels and a spectral radius between 3 and 8 pixels.

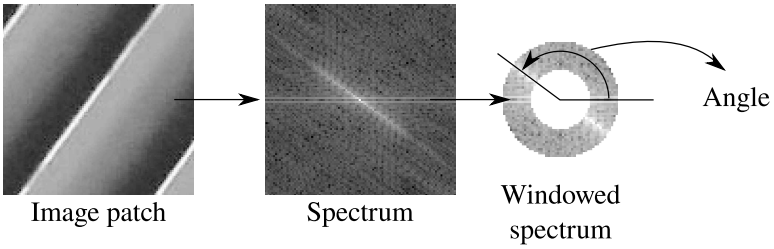


FIGURE 3. Schematic of the local spectral angle calculation

(2) A novel sampling reconstruction condition was obtained for sheaves of piecewise linear functions [6], which are not bandlimited, and
(3) Several examples of the ambiguity sheaves associated to sheaves describing the diffusion of contaminants through a network of channels were described in [7].

3.6. **Novel algorithms developed.** In order to address Objective 3, we developed several novel algorithms and tested them on various datasets.

3.6.1. *Angle-sensitive filters.* Curved striations are sometimes an important feature to be detected in an image. For instance, the left panel of Figure 2 shows a photograph of a stack of dishes. The collection of edges of the dishes forms a striated feature in the image. It is therefore useful to have a filter that measures the orientation of striated features from an image. It is most effective to describe this orientation by an angle.

We developed a topological filter called the local spectral rotation angle (LSRA) which takes an intensity-valued image to an angle-valued image. Of necessity, this filter is not linear, since the space of angles is not a vector space. On the other hand, it is local since the orientation of striations can change across the image.

Topological filters provide a solid foundation on which to construct a method for making local angular measurements of striations in an image. Essentially, the desired filter should compute the angle of any striations in small patches of an image, and then assemble the resulting computations into an angle-valued image,

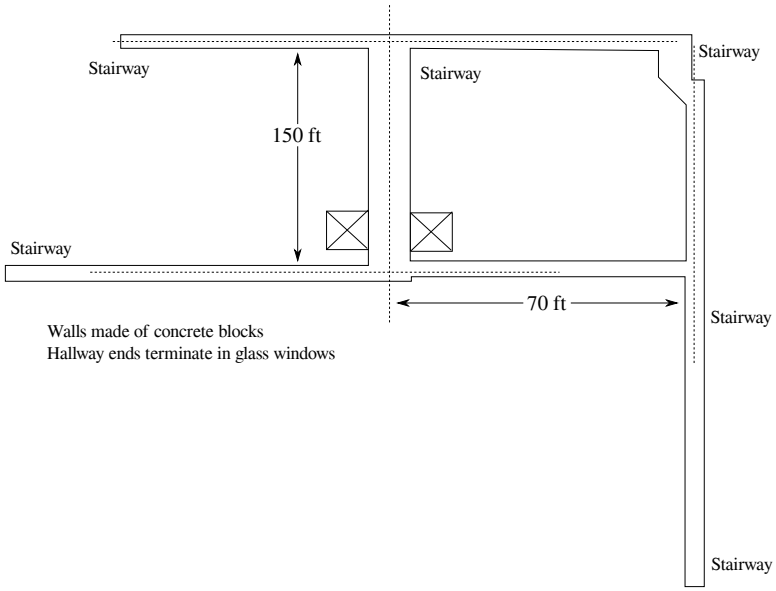


FIGURE 4. Dimensioned floorplan of the third floor of David Rittenhouse Laboratory

as shown in Figure 3. The topological filter described here uses (1) the 2d-Fast Fourier Transform of a small patch followed by (2) a peak detection on an annular window to determine the dominant angle.

This filter can be constructed using the same kind of sequence of sheaves

$$\mathcal{S}_1 \xleftarrow{p} \mathcal{S}_2 \xrightarrow{\lambda} \mathcal{S}_3.$$

as occurs with FIR LTI filters. Again, $\mathcal{S}_1$ represents the input and $\mathcal{S}_3$ represents the output, though the sections are images in the case of an angle-sensitive filter. The sheaf $\mathcal{S}_2$ models the internal state of the filter, in this case circular regions of pixels centered on each point. The morphism p performs the same role as before, by loading input pixels into the state buffer. However, λ is now no longer a linear operator. Instead it is an implementation of the procedure shown in Figure 3.

3.6.2. *Geometry extraction for transmission lines.* A global section of a sheaf is uniquely specified by its value on all vertices. In the case of a transmission line sheaf, a global section constrains the geometry of the graph. The calculation of the values of a section along a graph is a useful tool for "sounding" the length of edges in a graph, described in detail in [9]. The collection procedure is straightforward: place a directional sensor at each vertex in the graph and measure the incoming wave amplitudes. For instance, a horn antenna could be placed at each intersection of hallways in a building, and oriented in the direction of each hallway for each measurement. As is already well-known, it is necessary to select several algebraically independent operating wavenumbers, as the next example shows.

Example 10. Consider the case of attempting to measure the geometry of Figure 4 from the graph model in Figure 5. Suppose that three sensors are placed at

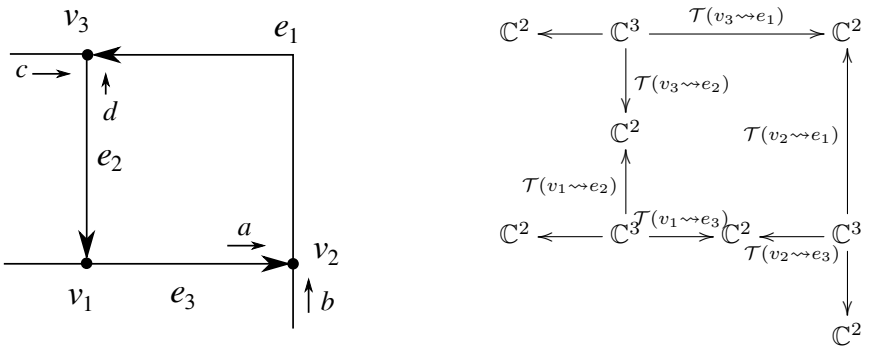


FIGURE 5. Directed graph (left) and transmission line sheaf (right) for the third floor of David Rittenhouse Laboratory

TABLE 2. Simulated magnitude and phase measurements for Example 10

Vertex	Hallway	Mag at 905 MHz	Phase at 905 MHz	Mag at 2451 MHz	Phase at 2451 MHz
v_1	External	1.4 dB	$-8.6°$	0.44 dB	$-14°$
v_1	e_2	-5.5 dB	$-1.5°$	-0.82 dB	$-28°$
v_1	e_3	-6.6 dB	$82°$	-4.9 dB	$83°$
v_2	External	1.9 dB	$-78°$	0.57 dB	$-85°$
v_2	e_1	-3.4 dB	$156°$	-1.7 dB	$108°$
v_2	e_3	-6.6 dB	$131°$	-4.9 dB	$150°$
v_3	External	0 dB	$0°$	0 dB	$0°$
v_3	e_1	-3.4 dB	$-126°$	-1.7 dB	$-17°$
v_3	e_2	-5.5 dB	$150°$	-0.82 dB	$-16°$

each of v_1, v_2, and v_3, for which simulated magnitude and phase measurements shown in Table 2. These measurements correspond to two operating frequencies, one at 905 MHz and one at 2.451 GHz (typical wireless network frequencies), and were simulated by solving the lossless Helmholtz equation on a graph in which the edge lengths were as shown in Figure 4, namely $L(e_1) = 220$ ft, $L(e_2) = 150$ ft, and $L(e_3) = 70$ ft. The choice of frequences is important; frequencies with small common factors make accurate measurements of edge lengths difficult.

The algorithm described in [9] allows us to estimate the lengths of e_1, e_2, and e_3. For instance, using the operating frequency of 905 MHz yields the following estimates:

- $L(e_1) \approx -0.233$ ft, which is too small by exactly 405 half-wavelengths,
- $L(e_2) \approx 0.459$ ft, which is too small by exactly 275 half-wavelengths, and
- $L(e_3) \approx -0.148$ ft, which is too small by exactly 129 half-wavelength.

Using 2.451 GHz alone isn't more accurate, since it yields the following estimates:

- $L(e_1) \approx 0.140$ ft, which is too small by exactly 1095 half-wavelengths,
- $L(e_2) \approx 0.0130$ ft, which is too small by exactly 747 half-wavelengths, and
- $L(e_3) \approx -0.0742$ ft, which is too small by exactly 349 half-wavelengths.

However, combining the two frequencies alleviates the difficulty, in the following way. Suppose we have two estimates L and L' for an edge length, associated to wavelengths λ and λ', we then search for the smallest such value that satisfies

$$L + m\frac{\lambda}{2} = L' + n\frac{\lambda'}{2},$$

where m and n are integers. Performing this search on each of the edges in our graph yields the correct lengths, namely $L(e_1) \approx 220$ ft, $L(e_2) \approx 150$ ft, and $L(e_3) \approx 70$ ft.

3.6.3. *Tracking of rotating targets.* As one of the first examples of topologically-motivated filtering and detection process, we developed a method for tracking a rotating target. In [5], we presented a novel angular fingerprinting algorithm for detecting changes in the direction of rotation of a target with a monostatic, stationary sonar platform. Unlike other approaches, we assumed that the target's centroid is stationary, and exploited doppler multipath signals to resolve the otherwise unavoidable ambiguities that arise. Since the algorithm is based on an underlying differential topological theory, it is highly robust to distortions in the collected data. We demonstrated performance of this algorithm experimentally, by exhibiting a pulsed doppler sonar collection system that runs on a smartphone. The performance of this system is sufficiently good to both detect changes in target rotation direction using angular fingerprints.

Operationally, the process consists of the following steps:

(1) Acquisition of the reference collection.
(2) Measurement of the rotation period. We used a range-doppler filter to identify the target's signature, from which the rotation period can be deduced as the largest doppler component at the target's range.
(3) Range gating. Since the target's range is known, all range bins before this range are removed. This is important since there were substantial range and doppler sidelobes present in the waveform. (These are largely due to receiver desense and overload effects as no time sensitivity control was applied to the receiver.)
(4) Storage of a contiguous block of pulses that correspond to precisely one period. Assuming the fan is rotating at a constant speed, the pulse number corresponds linearly with the angular coordinate of the fan's rotation.
(5) Acquisition of the second collection.
(6) Application of the same range gate filtering as applied to the reference collection.
(7) For each pulse of the second collection, the nearest pulse (in the energy norm) of the first collection is computed. This sets up a function from the pulses of the second collection to the angular coordinate of the first.
(8) Since this function is subject to noise, we Kalman filtered it, resulting in an approximation of the angular fingerprint function F is the output of the procedure.

By examining the slope of the angular fingerprint, changes in rotation direction speed can be deduced. If the slope is negative, then the rotation directions in the first and second collections differ; if positive, then the rotation directions agree.

We demonstrated the Algorithm by measuring the rotation rate of a ceiling fan using a simple sonar setup. Consider the collection geometry indicated in Figure

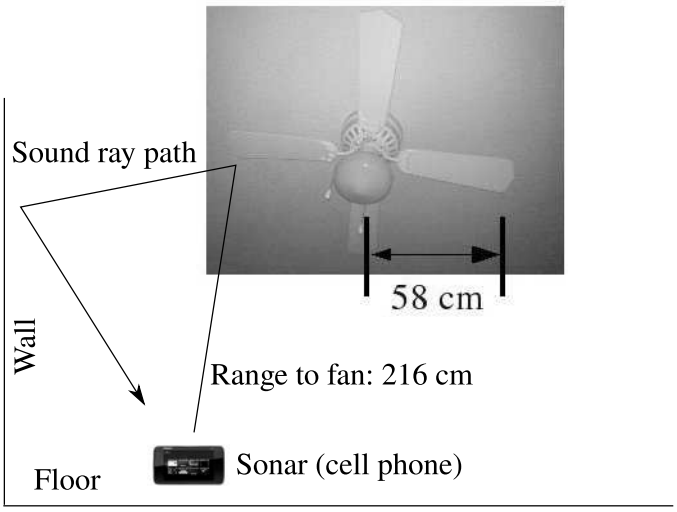


FIGURE 6. Experimental setup: fan on ceiling, sensor on floor

6. In this scenario, a rotating fan was located on the ceiling of a room with walls made of drywall and a sonar platform was placed on the floor. The sonar platform consisted of the speaker and microphone of a cell phone (Prof. Robinson's Nokia n900). The transmitted waveform was an impulse train with a bandwidth of 7 kHz, range resolution of 5 cm, and pulse repetition rate of 34 Hz. In all, 175 pulses were collected.

A typical final output fingerprint function (after Kalman filtering) is displayed in Figure 7. The negative slope of the graph indicates that the fan was spinning different directions during each of the two collections.

4. Dissemination activities

4.1. **Monograph on Topological Signal Processing.** One of Prof. Robinson's primary intellectual outputs from this project was a monograph entitled *Topological Signal Processing*. Prof. Robinson has already used portions of the book for training the students in his research group. Prof. Robinson has a contract with Springer for publishing the completed manuscript both in print and online, and a completed draft of it was circulated for comments from trusted members of the Applied Topology community at the beginning of Summer 2013.

This book takes the perspective that signal processing has much to gain by taking a more *local* approach; consistency between nearby sensors or measurements is expected, but is not expected between sensors that are far apart. But how does one measure *distance* without explicitly invoking geometry, which is potentially very uncertain? This is the purview of *topology*; the lesson is that nearness can be studied implicitly and local signals can be studied through the theory of *sheaves*.

4.2. **Papers and preprints written.**

(1) [5] "Multipath-dominant analysis of rotating blades using a pulsed waveform," *IET Radar Sonar and Navigation*, Volume 7, Issue 3, March 2013,

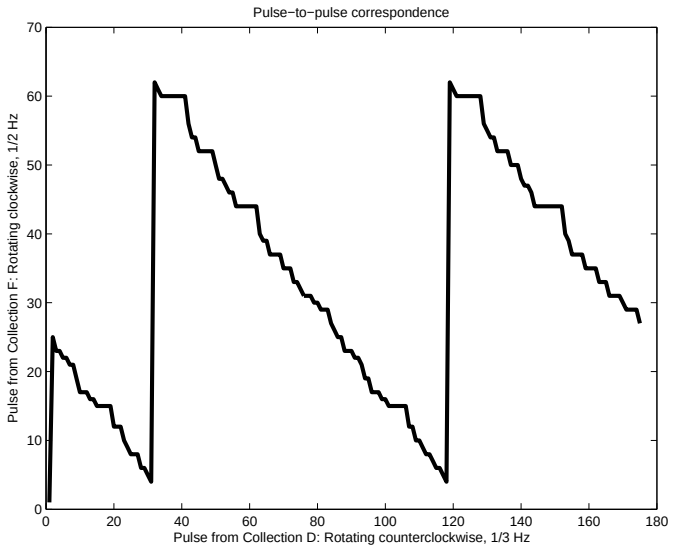


FIGURE 7. Angular fingerprint of two experiments with different rotation directions and speeds

pp. 217-224. Wind turbines present an opportunity for real-time, clear weather wind observations. By remotely measuring the speed and direction of turbine rotation, one could opportunistically measure wind speed and direction. However, current methods do not allow for rotation direction measurement if one uses a stationary monostatic radar platform. We presented a novel angular fingerprinting algorithm that fills this gap. Unlike other approaches, the target's centroid is stationary and, so the algorithm exploits doppler multipath signals to resolve the otherwise unavoidable ambiguities. Since the algorithm is based on an underlying differential topological theory, it is robust to distortions in the collected data. The authors demonstrate performance of this algorithm experimentally, by exhibiting a pulsed doppler sonar collection system that runs on a smartphone.

(2) [6] "The Nyquist theorem for cellular sheaves," *Sampling Theory and Applications (SampTA) 2013*, Bremen, Germany, arXiv:1307.7212. We develop a unified sampling theory based on sheaves and show that the Shannon-Nyquist theorem is a cohomological consequence of an exact sequence of sheaves. Our theory indicates that there are additional cohomological obstructions for higher-dimensional sampling problems. Using these obstructions, we also present conditions for perfect reconstruction of piecewise linear functions on graphs, a collection of non-bandlimited functions on topologically nontrivial domains.

(3) [7] "Understanding networks and their behaviors using sheaf theory," to appear in *IEEE Global Conference on Signal and Information Processing (GlobalSIP) 2013*, Austin, Texas, arXiv:1308.4621. Many complicated network problems can be easily understood on small networks. Difficulties arise when small networks are combined into larger ones. Fortunately, the

mathematical theory of sheaves was constructed to address just this kind of situation; it extends locally-defined structures to globally valid inferences by way of consistency relations. This paper exhibits examples in network monitoring and filter hardware where sheaves have useful descriptive power.

4.3. **Conference activity.** In Prof. Robinson's role as a mediator between mathematics and engineering, he traveled on this project to foster cross-discipline collaboration between engineers and mathematicians. Specifically, his conference participation during Summer 2013 on this project brought visibility to the message that *sheaf theory is a valuable engineering tool.* He made two international trips, which presented results on this project. The first of these two conferences (SampTA in Bremen, Germany (July 1-5, 2013)) is populated by half mathematicians and half engineers, while the second (Applied Topology in Bedlewo, Poland (July 21-27, 2013)) attracts exclusively mathematicians.

Prof. Robinson served as an organizer of the invited special session on Sampling and Geometry at SampTA, and was a referee for papers submitted to the session. As an invited speaker at the conference, he presented the results outlined above on sampling theory, which served to convince engineers that they have something to learn from sheaf theory. This conference was populated potential *users* of the theory developed on this project. The wider SampTA conference attracts very high-profile attention; Emmanuel Candes (Stanford), one of the inventors of compressive sensing, gave a plenary address.

The second conference was the annual meeting of Applied Topologists. This conference was organized by the top researchers in the field, indeed its initial proponents, including Frederick Cohen (University of Rochester), Michael Farber (University of Warwick), Robert Ghrist (University of Pennsylvania), and Marian Mrozek (Jagiellonian University). There are no contributed paper sessions at this conference; one speaks by invitation only. Prof. Robinson gave a plenary address at this conference, and presented the other half of the story about applied sheaf theory, that new mathematical aspects to the theory must be developed to address engineering problems.

5. Student participation

The mathematics master's degree program at American University allows students to pursue a degree in pure and applied mathematics. American University provides an excellent combination of resources for advanced education in the mathematical sciences. Through small class sizes and attentive faculty, students receive personalized attention throughout the program. This project nutured the development of two talented students, who in turn extended the reach of the project's technical results.

5.1. **Graduate student: Morgan DeHart.** The project funded Morgan DeHart during the Fall 2012 and Spring 2013 terms, which was the first academic year of his Master's degree program. Mr. DeHart assisted the analysis of switch sheaf cohomology, and was instrumental in the study of morphisms between switching sheaves. The software that he wrote systematically counted the possible morphisms between switching sheaves along a cellular map, and tabulated these results over all possible logic circuits of a given topology.

Mr. DeHart started his study of sheaf theory during the Fall 2012 semester, and was therefore an ideal candidate to proofread the Topological Signal Processing book being written by Prof. Robinson. He made numerous useful comments during this process and because of this, developed a solid understanding of sheaf theoretic techniques.

At the completion of the Spring 2013 semester, Mr. DeHart took a summer internship with United Technologies Research Center under the direction of Dr. Alberto Speranzon, studying the application of topology to network cyber/physical security.

5.2. **Graduate student: Matthew Hubler.** The project funded Matthew Hubler during the summer of 2013, during which he was working on his Master's degree. Mr. Hubler wrote software in support of the application of novel topological filters (as described above) to image analysis tasks. Mr. Hubler also developed a good working knowledge of the operating parameters for the TerraSAR-X satellite radar platform, which was the source of some of the imagery he helped to analyze. His work was (and continues to be) valuable by providing a fielded testbed for testing the novel algorithms developed on this project.

6. Recommendations for future work

This project uncovered several different kinds of sheaves for signal processing and network problems. These sheaves appear to arise from related constructions, and have related properties, but no explicit common framework has been discovered. A good next step would be to develop a general classification of information system-relevant sheaves that includes existing sheaf models as special cases. This model would highlight what aspects of an information system are best captured by a sheaf theoretic model, and would suggest ways to avoid ambiguities when using the model to make inferences.

Specific recommendations for future work include

(1) *Unify* the construction of sheaf models for information systems, so that the different examples discovered on this project are special cases,
(2) *Analyze* the persistent cohomological features arising from these models, and
(3) *Discover* plausible inference patterns from these features.

6.1. **Recommendation 1: Unify the constructions of sheaf models for information systems.** Given a collection of sheaf models of an information system, it would be useful to produce a systematic, canonical construction of morphisms between them. For instance, the specification of a specific switching sheaf morphism is rather constrained. This indicates that given a particular category of sheaf models, the morphisms may be determined by a smaller, "engineering-friendly" specification. With such a specification, and it should be possible to understand the resulting cohomology of sheaf models. For instance, this would produce an explanation of the meaning of H^0 for switching sheaves; an unresolved mystery at present. More aggressively, we may discover a duality theorem (probably based on Verdier duality) that relates sheaves describing network nodes and sheaves describing network connections.

6.2. **Recommendation 2: Analyze persistent cohomological features.** Sequences of sheaves support the definition of *persistence sheaves*, a (dual) generalization of the increasingly popular *persistent homology* used in data analysis. We should aim to understand the meaning of "persistent topological feature" in this context, with a careful emphasis on the meaning of the complexity scale. Indeed, while zig-zag persistence works formally in this context, it is unclear how to interpret the persistence scale. The usual theorems about robustness of persistence diagrams are of no use. We should therefore aim to prove theorems that explain the robustness of persistent cohomology in this new context. Finally, adoption by the engineering community would require developing a software tool to aid the study of persistent sheaf cohomology in applications.

6.3. **Recommendation 3: Discover patterns of inference from cohomological features.** The most ambitious research direction would be an assault on a proof of a multiscale, general sampling theorem. This theorem could have wide-ranging implications, including special cases that indicate bounds on sampling requirements for network and software validation, sensor planning, and beyond. As a mitigation of the associated technical risk, we could enhance the software developed in support of Recommendation 2 to compute cohomology of sequences of persistence sheaves. This would enable the *ad hoc* study of sampling in particular systems, even in the absence of a theoretical result.

References

[1] Glen Bredon. *Sheaf theory*. Springer, 1997.

[2] J. Curry. Sheaves, cosheaves and applications, `arxiv:1303.3255`. 2013.

[3] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Persistent cohomology and circular coordinates. *Discrete & Computational Geometry*, 45(4):737–759, 2011.

[4] John H. Hubbard. *Teichmüller Theory, volume 1.* Matrix Editions, 2006.

[5] M. Robinson. Multipath-dominant analysis of rotating blades using a pulsed waveform. *IET Radar Sonar and Navigation*, 7(3):217–224, March 2013.

[6] M. Robinson. The Nyquist theorem for cellular sheaves. In *Sampling Theory and Applications (SampTA)*, 2013.

[7] M. Robinson. Understanding networks and their behaviors using sheaf theory. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2013.

[8] Michael Robinson. Asynchronous logic circuits and sheaf obstructions. In *GETCO 2010*, January 2010.

[9] Michael Robinson. Inverse problems in geometric graphs using internal measurements, `arxiv:1008.2933`, 2010.

[10] Claude Shannon. A symbolic analysis of relay and switching circuits. Master's thesis, MIT, 1940.

[11] A. Shepard. *A cellular description of the derived category of a stratified space*. PhD thesis, Brown University, 1985.

Python sheaf library

```
# Persistence-capable sheaf manipulation library
#
# Copyright (c) 2013, Michael Robinson
# Distribution of unaltered copies permitted for noncommercial use only
# All other uses require express permission of the author
# This software comes with no warrantees express or implied
```

```
import numpy as np
import random

## Data structures
class Coface:
    """A coface relation"""
    def __init__(self,index,orientation):
        self.index=index
        self.orientation=orientation

class Cell:
    """A cell in a cell complex"""
    def __init__(self,dimension,compactClosure=True,cofaces=[]):
        self.dimension=dimension
        self.compactClosure=compactClosure
        self.cofaces=cofaces

    def cofaceList(self):
        return [cf.index for cf in self.cofaces]

    def isCoface(self,index,orientation=None):
        if orientation==None:
            return index in [cf.index for cf in self.cofaces]
        else:
            return (index,orientation) in [(cf.index,cf.orientation) for cf in self.cofaces]

class CellComplex:
    def __init__(self,cells):
        """Construct a cell complex from its cells"""
        self.cells=cells

    def isFaceOf(self,c,cells=[]):
        """Construct a list of all cells that a given cell is a face of"""
        if cells:
            cl=cells
        else:
            cl=range(len(self.cells))
        return [i for i in cl if self.cells[i].isCoface(c)]

    def skeleton(self,k,compactSupport=False):
        return [i for i in range(len(self.cells))
                if ((compactSupport or self.cells[i].compactClosure) and self.cells[i].dimens

    def faces(self,c):
        """Compute a list of all faces of a cell"""
        return [i for i in range(len(self.cells)) if self.cells[i].isCoface(c)]

    def cofaces(self,c,cells=[]):
```

```
        """Iterate over cofaces (of all dimensions) of a given cell; optional argument specif
        for cf in self.cells[c].cofaces:
            if cf.index in cells or not cells:
                yield cf

        for cf in self.cells[c].cofaces:
            if cf.index in cells or not cells:
                self.cofaces(cf.index,cells)

    def components(self,cells=[]):
        """Compute connected components; optional argument specifies permissible cells"""
        if not cells:
            cellsleft=range(len(self.cells))
        else:
            cellsleft=cells

        cpts=[]
        cpt=[]
        while cellsleft:
            cpt=self.expandComponent(cellsleft[0],cells,[cellsleft[0]])
            cpts+=[list(set(cpt))]
            cellsleft=list(set(cellsleft).difference(cpt))
        return cpts

    def expandComponent(self,start,cells=[],current_cpt=[]):
        """Compute the connected component started from a given cell.  Optional argument spec
        if not cells:
            cellsleft=list(set(range(len(self.cells))).difference(current_cpt))
        else:
            cellsleft=list(set(cells).difference(current_cpt))
if not cellsleft:
            return current_cpt

        neighbors=self.connectedTo(start,cellsleft)
        for c in neighbors:
            current_cpt+=self.expandComponent(c,cellsleft,list(set(current_cpt+neighbors)))
            current_cpt=list(set(current_cpt))
        return current_cpt

    def connectedTo(self,start,cells=[]):
        """Which cells is a cell connected to? Optional argument specifies permissible cells"
        if not cells:
            return list(set(self.faces(start) + self.cells[start].cofaceList()))
        else:
    return list(set.intersection(set(self.faces(start) + self.cells[start].cofaceList()),cell

    def starCells(self,cells):
        """Cells in star over a subset of a cell complex"""
```

```
        return list(set(cells+[cf.index for c in cells for cf in self.cofaces(c)]))

class SheafCoface(Coface):
    """A coface relation"""
    def __init__(self,index,orientation,corestriction):
        self.index=index
        self.orientation=orientation
        self.corestriction=corestriction

    def __repr__(self):
        return "(index=" + str(self.index) + ",orientation="+str(self.orientation)+",corestri

class SheafCell(Cell):
    """A cell in a cell complex with a sheaf over it"""
    def __init__(self,dimension,cofaces=[],compactClosure=True,stalkDim=1):
        if cofaces:
            try:
                self.stalkDim=cofaces[0].corestriction.shape[1]
            except AttributeError:
                self.stalkDim=0
        else:
            self.stalkDim=stalkDim
        Cell.__init__(self,dimension,compactClosure,cofaces)

    def __repr__(self):
        string="(dimension="+str(self.dimension)+",compactClosure="+str(self.compactClosure)
        if self.cofaces:
            for cf in self.cofaces:
                string+="," + cf.__repr__()
            return string+")"
        else:
            return string+",stalkdim="+str(self.stalkDim)+")"

# Sheaf class
class Sheaf(CellComplex):
    def cofaces(self,c,cells=[],currentcf=[]):
        """Iterate over cofaces (of all dimensions) of a given cell; optional argument specif

        for cf in self.cells[c].cofaces:
            if cf.index in cells or not cells:
                if currentcf:
                    cfp=SheafCoface(cf.index,
                        currentcf.orientation*cf.orientation,
                        np.dot(currentcf.corestriction,corestriction))
                else:
                    cfp=cf
                yield cfp
```

```
        for cf in self.cells[c].cofaces:
            if cf.index in cells or not cells:
                if currentcf:
                    cfp=SheafCoface(cf.index,
                        currentcf.orientation*cf.orientation,
                        np.dot(currentcf.corestriction,corestriction))
                else:
                    cfp=cf
                self.cofaces(cf.index,cells,cfp)

    def star(self,cells):
        """Restrict a sheaf to the star over a subset of the base space"""

        # Extract a list of all relevant cells in the star
        cells=CellComplex.starCells(self,cells)

        return Sheaf([SheafCell(dimension=self.cells[i].dimension,
                                stalkDim=self.cells[i].stalkDim,
                                compactClosure=self.cells[i].compactClosure and (not set(self
                                cofaces=[SheafCoface(cells.index(cf.index),cf.orientation,cf.

    def kcells(self,k,compactSupport=False):
        """Extract the compact k-cells and associated components of coboundary matrix"""
        k=CellComplex.skeleton(self,k,compactSupport)
        ksizes=[self.cells[i].stalkDim for i in k]
        kidx=list(cumulative_sum(ksizes))
        return k,ksizes,kidx

    def localSectional(self,cells=[]):
        """Construct a new sheaf whose global sections are the local sections of the current

        if not cells:
            cells=range(len(self.cells))

        # Edges of new sheaf = elements of S with at least one face in the list of cells
        edges=[i for i in cells if self.isFaceOf(i,cells)]

        morphism=[]
        newcells=[]
        for i in edges:
            newcells.append(SheafCell(1,
                compactClosure=self.cells[i].compactClosure and (not set(self.faces(i)).diffe
                stalkDim=self.cells[i].stalkDim))
            morphism.append(SheafMorphismCell([i],[np.eye(self.cells[i].stalkDim)]))

        # Vertices of new sheaf = elements of S with no faces
        vert=list(set(cells).difference(edges))
```

```
        # Corestrictions of new sheaf = compositions of corestrictions
        for i in vert:
            # starting at this vertex, do a depth-first search for the edges in the new sheaf
            cofaces=list(self.cofaces(i,cells))
            newcofaces=[]
            for cf in cofaces:
                newcofaces.append(SheafCoface(edges.index(cf.index),
                    cf.orientation,
                    cf.corestriction))

            if cofaces:
                newcells.append(SheafCell(0,compactClosure=True,cofaces=newcofaces))
            else:
                newcells.append(SheafCell(0,compactClosure=True,stalkDim=self.cells[i].stalkD

            morphism.append(SheafMorphismCell([i],[np.eye(self.cells[i].stalkDim)]))

        return Sheaf(newcells),morphism

    def localRestriction(self,cells_1,cells_2):
        """Compute the map induced on local sections by restricting from a larger set to a sm

        # Obtain sheaves and morphisms of local sections for both sets
        sheaf_1,mor_1=self.localSectional(cells_1)
        sheaf_2,mor_2=self.localSectional(cells_2)

        # Compute global sections of each sheaf in terms of cohomology
        H0_1=sheaf_1.cohomology(0)
        if not np.all(H0_1.shape):
            return np.zeros(H0_1.shape)

        # Extend sections of sheaf 1 to vertices of sheaf 2, if needed
        # Observe that value at each vertex of sheaf 2 either
        #  (1) comes from value at a vertex of sheaf 1 or
        #  (2) comes from value at an edge of sheaf 1,
        #      in which case a single corestriction map obtains it
        #      from the value at a vertex of sheaf 1
        k_1,ksizes_1,kidx_1=sheaf_1.kcells(0)
        k_2,ksizes_2,kidx_2=sheaf_2.kcells(0)
        rows=sum(ksizes_2)
        sections=np.zeros((rows,H0_1.shape[1]))
        for ss in range(H0_1.shape[1]): # Looping over sections in sheaf 1
            for i in range(len(k_2)): # Looping over vertices in sheaf 2
                # Compute compute preimages of this sheaf 2 vertex
                ms=[k for k in range(len(mor_1)) if
                    set(mor_1[k].destinations).intersection(mor_2[k_2[i]].destinations)]
                if ms:
                    if sheaf_1.cells[ms[0]].dimension==0:
```

```
                    ii=ms[0]
                    idx=k_1.index(ii)
                    map,j1,j2,j3=np.linalg.lstsq(mor_2[i].maps[0],mor_1[ii].maps[0])
                    A=np.dot(map,H0_1[kidx_1[idx]:kidx_1[idx+1],ss])
                    sections[kidx_2[i]:kidx_2[i+1],ss]=A
                else:
                    ii=sheaf_1.faces(ms[0])[0] # parent cells
                    idx=k_1.index(ii)
                    for cf in sheaf_1.cells[ii].cofaces:
                        if cf.index==ms[0]:
                            cr=cf.corestriction
                            break
                    A=np.dot(cr,mor_1[ii].maps[0])

                    map,j1,j2,j3=np.linalg.lstsq(mor_2[i].maps[0],A)
                    sections[kidx_2[i]:kidx_2[i+1],ss]=np.dot(map,H0_1[kidx_1[idx]:kidx_1

    # Rewrite sections over sheaf 2 in terms of 0-cohomology basis

    map,j1,j2,j3 = np.linalg.lstsq(sections,sheaf_2.cohomology(0))
    return map.conj().T

# Input: k = degree of cohomology to compute
# Output: matrix
def coboundary(self,k,compactSupport=False):
    """Compute k-th coboundary matrix"""
    # Collect the k-cells and k+1-cells
    ks,ksizes,kidx=self.kcells(k,compactSupport)
    kp1,kp1sizes,kp1idx=self.kcells(k+1,compactSupport)

    # Allocate output matrix
    rows=sum(kp1sizes)
    cols=sum(ksizes)
    d=np.zeros((rows,cols),dtype=np.complex)
    if rows and cols:
        # Loop over all k-cells, writing their matrices into the output matrix
        for i in range(len(ks)):
            # Loop over cofaces with compact closure
            for cf in self.cells[ks[i]].cofaces:
                if self.cells[cf.index].compactClosure or compactSupport:
                    ridx=kp1.index(cf.index)
                    block=np.matrix(cf.orientation*cf.corestriction)
                    d[kp1idx[ridx]:kp1idx[ridx+1],kidx[i]:kidx[i+1]]+=block
        return d
    else:
        return d

def cohomology(self,k,compactSupport=False,tol=1e-5):
```

```
        """Compute basis for k-th cohomology of the sheaf"""

        # Obtain coboundary matrices for the sheaf
        dm1=self.coboundary(k-1,compactSupport)
        dm1=np.compress(np.any(abs(dm1)>tol,axis=0),dm1,axis=1)
        d=self.coboundary(k,compactSupport)

        # Compute kernel
        if d.size:
            ker=kernel(d,tol);
        else:
            ker=np.eye(d.shape[1])

        # Remove image
        if k > 0 and dm1.any():
            map,j1,j2,j3=np.linalg.lstsq(ker,dm1)
            Hk=np.dot(ker,cokernel(map,tol));
        else:
            Hk=ker

        return Hk

    def betti(self,k,compactSupport=False,tol=1e-5):
        """Compute the k-th Betti number of the sheaf"""
        return self.cohomology(k,compactSupport).shape[1]

    def pushForward(self,targetComplex,map):
        """Compute the pushforward sheaf and morphism along a map"""

        sheafCells=[]
        mor=[]
        # Loop over cells in the target cell complex
        for cidx in range(len(targetComplex.cells)):
            c=targetComplex.cells[cidx]

            # Compute which cells are in the preimage of this cell
            bigPreimage=[d for d,r in map if r==cidx]

            # For each cell, compute map on global sections over the star
            # along each attachment
            cfs=[]
            for cf in c.cofaces:
                smallPreimage=[d for d,r in map if r==cf.index]
                corest=self.localRestriction(self.starCells(bigPreimage),
                    self.starCells(smallPreimage))
                cfs.append(SheafCoface(cf.index,
                    cf.orientation,corest))
```

```
            mor.append(SheafMorphismCell(bigPreimage,
                [self.localRestriction(self.starCells(bigPreimage),[d]) for d in bigPreimage]
            if cfs:
                sheafCells.append(SheafCell(c.dimension,cfs,c.compactClosure))
            else:
                ls,m=self.localSectional(self.starCells(bigPreimage))
                sheafCells.append(SheafCell(c.dimension,[],c.compactClosure,stalkDim=ls.betti

        return Sheaf(sheafCells),mor

    def flowCollapse(self):
        """Compute the sheaf morphism to collapse a sheaf to a flow sheaf over the same space

        # Generate the flow sheaf
        fs=FlowSheaf(self)

        mor=[]
        for i in range(len(self.cells)):
            c=self.cells[i]

            # If a vertex, collapse by composing edge morphism with corestrictions
            if c.dimension==0:
                map=np.zeros((0,c.stalkDim))
                for j in range(len(c.cofaces)-1):
                    cf=c.cofaces[j]
                    map=np.vstack((map,np.sum(cf.corestriction,axis=0)))

                mor.append(SheafMorphismCell([i],[map]))
            else:
                # If an edge, collapse by summing
                mor.append(SheafMorphismCell([i],[np.ones((1,c.stalkDim))]))

        return fs,mor

class AmbiguitySheaf(Sheaf):
    def __init__(self,shf1,mor):
        """Construct an ambiguity sheaf from two sheaves (over the same base) and a morphism

        cellsnew=[]
        for i in range(len(shf1.cells)):
            c=shf1.cells[i]

            # New cell has same dimension, compactness,
            # Stalk is the kernel of the component map there
            # Corestrictions come from basis change on each corestriction
            K=kernel(mor[i].map[0])
            stalkDim=K.shape[0]
            cfnew=[]
```

```
            for cf in shf1.cells[i].cofaces:
                S=cf.corestriction
                L=kernel(mor[cf.index].map[0])
                R=np.linalg.lstsq(L,np.dot(S,K))
                cfnew.append(SheafCoface(index=cf.index,
                    orientation=cf.orientation,
                    corestriction=R))

            cellsnew.append(SheafCell(dimension=c.dimension,
                compactClosure=c.compactClosure,
                stalkDim=stalkDim,
                cofaces=cfnew))

        Sheaf.__init__(self,cellsnew)

# Flow sheaves
class DirectedGraph(CellComplex):
    def __init__(self,graph,vertex_capacity=-1):
        """Create a cell complex from a directed graph description, which is a list of pairs
        The vertex labeled None is an external connection
        Cells are labeled as follows:
         First all of the edges (in the order given),
         then all vertices (in the order they are given; not by their numerical
         values)"""

        # Construct list of vertices
        verts=[]
        for ed in graph:
            s=ed[0]
            d=ed[1]
            if s != None:
                verts.append(s)
            if d != None:
                verts.append(d)
        verts=list(set(verts))

        # Loop over edges, creating cells for each
        compcells=[]
        for i in range(len(graph)):
            compcells.append(Cell(dimension=1,
                                  compactClosure=(graph[i][0]!=None) and (graph[i][1]!=None))
            compcells[-1].vertex_label=None
            try: # Add capacity if specified
                compcells[-1].capacity = graph[i][2]
            except:
                pass

        # Loop over vertices, creating cells for each
```

```
        for i in verts:
            # Collect cofaces
            cfs=[j for j in range(len(graph)) if graph[j][0]==i or graph[j][1]==i]
            # Compute orientations of each attachment
            orient=[]
            cofaces=[]
            for j in range(len(cfs)):
                if graph[cfs[j]][0]==i:
                    orient.append(-1)
                else:
                    orient.append(1)
                cofaces.append(Coface(cfs[j],orient[j]))

            compcells.append(Cell(dimension=0,
                                  compactClosure=True,
                                  cofaces=cofaces))
            compcells[-1].vertex_label=i
            compcells[-1].capacity=vertex_capacity

        CellComplex.__init__(self,compcells)

    def findPath(self,start,end,history=[]):
        """Find a path from specified start cell to end cell
Cell attribute .capacity_left specifes whether the cell can be used"""
        if start == end:
            return history+[end]

        # Initialize the capacities used, if unavailable
        for c in self.cells:
            if not hasattr(c,'capacity_left'):
                c.capacity_left = 1

        if self.cells[start].dimension == 0:
            # Compute list of outgoing edges
            for cf in self.cells[start].cofaces:
                if cf.orientation == -1 and cf.index not in history and self.cells[cf.index].
                   ch=self.findPath(cf.index,end,history+[start])

                   if ch:
                       return ch
            return None
        else:
            # Locate vertices which this edge is pointing into
            fs=[i for i in range(len(self.cells)) if self.cells[i].isCoface(start,1)]
            # Is there is a vertex with remaining capcity?
            if fs and fs[0] not in history and self.cells[fs[0]].capacity_left:
                return self.findPath(fs[0],end,history+[start])
            else: # No such vertex
```

```
            return None

    def maxFlow(self,start,end):
        """Compute the maximal flow through a graph using Ford-Fulkerson algorithm.  Cell att
        # Initialize the capacities on intermediate cells
        for c in self.cells:
            try: # ... to use requested capacities
                c.capacity_left = c.capacity
            except: # if no capacity specified
                if c.dimension == 0: # Vertices get infinite capacity
                    c.capacity_left=-1
                else: # Edges get capacity 1
                    c.capacity_left = 1

        # Initialize start/end cell capacities to be infinite
        self.cells[start].capacity_left=-1
        self.cells[end].capacity_left=-1

        # Search for paths
        chains=[]
        ch=self.findPath(start,end)
        while ch:
            # Add list of chains
            chains+=[ch]

            # Delete capacities from the cells in this chain
            for i in ch:
                self.cells[i].capacity_left -= 1

            # Find the next chain
            ch=self.findPath(start,end)

        return chains

    def maximalChains(self,start,history=[]):
        """Compute a list of maximal chains beginning at a cell"""

        if self.cells[start].dimension == 0:
            # Compute list of outgoing edges
            cfs=[cf for cf in self.cells[start].cofaces
                if cf.orientation == -1 and not cf.index in history]
            if cfs: # There are outgoing edges, loop over them
                chains=[self.maximalChains(cf.index,history+[start])
                        for cf in cfs]
                lst=[]
                for ch in chains:
                    lst+=ch
                return lst
```

```
            else: # No outgoing edges, so this vertex is terminal
                return [history+[start]]
        else:
            # Locate vertices which this edge is pointing into
            fs=[i for i in range(len(self.cells)) if self.cells[i].isCoface(start,1)]
            # Is there is a vertex with remaining capcity
            if fs and not fs[0] in history:
                return self.maximalChains(fs[0],history+[start])
            else: # No such vertex
                return [history+[start]]

    def coveringSpace(self,sheets,partial=False):
        """Create a directed graph that is a covering space of this one with the specified nu
        # Construct new edge set
        edges=[c for c in self.cells if c.dimension == 1]
        newcells=edges*sheets
        edgeidx=[idx for idx in range(0,len(self.cells))
                 if self.cells[idx].dimension==1]

        # Decompactify edges on request
        if partial:
            for i in range(0,len(edges)):
                newcells[i].compactClosure=False

        # Construct new vertex set
        for c in self.cells:
            if c.dimension == 0:
                for i in range(0,sheets):
                    if i == sheets-1 and partial:
                        break # Skip last vertex copy if requested

                    # Remap cofaces
                    cofaces=[Coface((edgeidx.index(cf.index)+
                                     (i+(1-cf.orientation)/2)*len(edges))
                                    %(len(edges)*sheets),
                                    cf.orientation)
                             for cf in c.cofaces]
                    newcells.append(Cell(dimension=0,
                                         compactClosure=True,
                                         cofaces=cofaces))

        return CellComplex(newcells)

def erdosRenyiDirectedGraph(nvert,prob):
    """Create a random graph with nvert vertices and probability of an edge prob"""
    return DirectedGraph([(a,b) for a in range(nvert)+[None]
                          for b in range(nvert)+[None]
                          if random.random() < prob and (a!=None or b!=None)])
```

```
class FlowSheaf(Sheaf,DirectedGraph):
    def __init__(self,graph):
        """Create a flow sheaf from a directed graph"""

        sheafcells=[]
        for c in graph.cells:
            cofaces=[]
            j=0
            for cf in c.cofaces:
                # Compute corestrictions
                if j in range(len(c.cofaces)-1):
                    corest=np.matrix([m==j for m in range(len(c.cofaces)-1)],dtype=int)
                else:
                    corest=np.matrix([cf.orientation for cf in c.cofaces][0:-1])

                cofaces.append(SheafCoface(cf.index,cf.orientation,corest))
                j+=1

            sheafcells.append(SheafCell(dimension=c.dimension,
                                        compactClosure=c.compactClosure,
                                        cofaces=cofaces))

        Sheaf.__init__(self,sheafcells)

class TransLineSheaf(Sheaf,DirectedGraph):
    def __init__(self,graph,wavenumber):
        """Create a transmission line sheaf from a directed graph, in which edges have been g

        # Default edge lengths
        for c in graph.cells:
            if c.dimension == 1:
                if c.compactClosure==False:
                    c.length=1
                try:
                    if c.length<0:
                        c.length=1
                except:
                    c.length=1

        sheafcells=[]

        for c in graph.cells:
            cofaces=[]

            if c.dimension == 0: # Edges have interesting corestrictions
                n=len(c.cofaces)
                phaselist=[2/n for i in range(n)]
```

```
                for m in range(n):
                    if c.cofaces[m].orientation == -1:
                        corest=np.matrix([[i==m for i in range(n)],
                                          phaselist],
                                         dtype=complex)
                        corest[1,m]-=1
                        corest[1,:]*=np.exp(-1j*wavenumber*graph.cells[c.cofaces[m].index].le
                    else:
                        corest=np.matrix([phaselist,
                                          [i==m for i in range(n)]],
                                         dtype=complex)
                        corest[0,m]-=1
                        corest[0,:]*=np.exp(1j*wavenumber*graph.cells[c.cofaces[m].index].len
                    cofaces.append(SheafCoface(c.cofaces[m].index,
                                               c.cofaces[m].orientation,
                                               corest))
            else: # All other faces have trivial corestrictions
                n=2
                cofaces=[SheafCoface(cf.index,cf.orientation,[]) for cf in c.cofaces]
            sheafcells.append(SheafCell(dimension=c.dimension,
                                        compactClosure=c.compactClosure,
                                        cofaces=cofaces,
                                        stalkDim=n))

        Sheaf.__init__(self,sheafcells)


class SheafMorphismCell:
    def __init__(self,destinations=[],maps=[]):
        """Specify destinations and maps for this cell's stalk under a morphism"""
        self.destinations=destinations
        self.maps=maps

# A local section
class SectionCell:
    def __init__(self,support,value):
        """Specify support cell indices and values in each cell stalk for a local section"""
        self.support=support
        self.value=value

class Section:
    def __init__(self,sectionCells):
        self.sectionCells=sectionCells

    def support(self):
        """List the cells in the support of this section"""
        return [sc.support for sc in self.sectionCells]
```

```
    def extend(self,sheaf,cell,value=None,tol=1e-5):
        """Extend the section to another cell; returns True if successful"""
        # If the desired cell is already in the support, do nothing
        if cell in self.support():
            return True

        # Is the desired cell a coface of a cell in the support?
        for s in self.sectionCells:
            for cf in sheaf.cells[s.support].cofaces:
                if cf.index == cell:
                    # If so, extend via corestriction
                    val=np.dot(cf.corestriction,s.value)

                    # Check for consistency
                    if value != None and np.any(np.abs(val - value)>tol):
                        return False
                    value = val

        # Are there are any cofaces for the desired cell in the support?
        if value == None: # Attempt to assign a new value...
            # Stack the corestrictions and values associated to existing support
            lst=[(cf.corestriction,s.value)
                 for cf in sheaf.cells[cell].cofaces
                 for s in self.sectionCells
                 if cf.index == s.support]
            if lst:
                crs=np.vstack([e[0] for e in lst])
                vals=np.vstack([e[1] for e in lst])

                # If the problem of solving for the value at this cell is
                # underdetermined, refrain from solving it
                if matrixrank(crs,tol) < crs.shape[1]:
                    return True

                # Attempt to solve for the value at desired cell
                val,res,j2,j3=np.linalg.lstsq(crs,vals)
                if np.any(np.abs(res)>tol):
                    return False
                value = val
        else: # ...or check consistency with an old one
            for cf in sheaf.cells[cell].cofaces:
                for s in self.sectionCells:
                    if s.support == cf.index:
                        if np.any(np.abs(np.dot(cf.corestriction,value)-s.value)>tol):
                            return False

        # A value was successfully assigned (if no value was assigned,
        # do nothing, but it's still possible to extend)
```

```
        if value != None:
            self.sectionCells.append(SectionCell(cell,value))

        return True

class PersistenceSheaf(Sheaf):
    # Compute k-th persistence sheaf
    # Input: list of sheaves
    #        list of triples: source sheaf index, destination sheaf index, sheaf morphism dat
    def __init__(self,sheaves,morphisms,k):
        """Compute the k-th degree persistence sheaf over a graph"""

        persheaf=[]

        # Loop over sheaves
        for i in range(len(sheaves)):
            # Loop over morphisms initiated from this sheaf
            cofaces=[]
            for (s,d,mor) in morphisms:
                if s==i:
                    cofaces.append(SheafCoface(d,
                                               1,
                                               inducedMap(sheaves[i],sheaves[d],mor,k)))
            if cofaces:
                persheaf.append(SheafCell(dimension=0,
                                          compactClosure=True,
                                          cofaces=cofaces))
            else: # If cell does not have cofaces, compute stalk from Betti number
                persheaf.append(SheafCell(dimension=1,
                                          compactClosure=len([d for (s,d,mor) in morphisms
                                                              if d==i])>1,
                                          stalkDim=sheaves[i].betti(k)))
        # Initialize the sheaf
        Sheaf.__init__(self,persheaf)

## Functions

def cumulative_sum(values, start=0):
    yield start
    for v in values:
        start += v
        yield start

def matrixrank(A,tol=1e-5):
    u, s, vh = np.linalg.svd(A)
    return sum(s > tol)

def kernel(A, tol=1e-5):
```

```
    u, s, vh = np.linalg.svd(A)
    sing=np.zeros(vh.shape[0],dtype=np.complex)
    sing[:s.size]=s
    null_mask = (sing <= tol)
    null_space = np.compress(null_mask, vh, axis=0)
    return null_space.conj().T

def cokernel(A, tol=1e-5):
    u, s, vh = np.linalg.svd(A)
    sing=np.zeros(u.shape[1],dtype=np.complex)
    sing[:s.size]=s
    null_mask = (sing <= tol)
    return np.compress(null_mask, u, axis=1)

def isSubchain(bigger,smaller):
    """Determine if smaller is a subchain of bigger"""
    try:
        idx=bigger.index(smaller[0])
    except:
        return False
    for s in smaller:
        try:
            if bigger[idx] != s:
                return False
            else:
                idx += 1
        except:
            return False
    return True

def subchainMatrix(chainsCols,chainsRows):
    """Construct a binary matrix specifying which chains in the second set are subchains of t
    mat=np.zeros((len(chainsRows),len(chainsCols)))
    for i in range(len(chainsCols)):
        for j in range(len(chainsRows)):
            if isSubchain(chainsCols[i],chainsRows[j]) or isSubchain(chainsRows[j],chainsCols
                mat[j,i]=1
    return mat

# Input: domain sheaf
#        range sheaf
#        list of sheaf morphism data, one for each cell
#        k
# Output: matrix
def inducedMap(sheaf1,sheaf2,morphism,k,compactSupport=False,tol=1e-5):
    """Compute k-th induced map on cohomology for a sheaf morphism"""

    # Compute cohomology basis for each sheaf
```

```
    Hk_1=sheaf1.cohomology(k,compactSupport)
    Hk_2=sheaf2.cohomology(k,compactSupport)

    if (not Hk_1.size) or (not Hk_2.size):
        return []

    # Extract the k-skeleta of each sheaf
    k_1,ksizes_1,kidx_1=sheaf1.kcells(k,compactSupport)
    k_2,ksizes_2,kidx_2=sheaf2.kcells(k,compactSupport)

    # Construct chain map
    rows=sum(ksizes_2)
    cols=sum(ksizes_1)
    m=np.zeros((rows,cols),dtype=np.complex)

    for i in range(len(k_1)):
        for j,map in zip(morphism[k_1[i]].destinations,morphism[k_1[i]].maps):
            if sheaf2.cells[j].dimension==k:
                ridx=[q for q in range(len(k_2)) if k_2[q]==j]
                if ridx:
                    ridx=ridx[0]
                    m[kidx_2[ridx]:kidx_2[ridx+1],kidx_1[i]:kidx_1[i+1]]+=map

    # Map basis for domain sheaf's cohomology through the chain map
    im=np.dot(m,Hk_1)

    # Expand output in new basis
    map,j1,j2,j3 = np.linalg.lstsq(im,Hk_2)

    return map.conj().T
```

Department of Mathematics and Statistics, American University, 4400 Massachusetts Ave NW, Washington, DC 20016
E-mail address: michaelr@american.edu