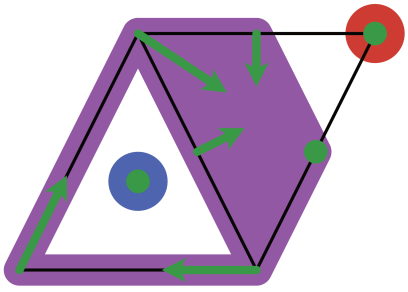
ConleyDynamics.jl

Version 0.4.4

Built by Julia 1.12.5

Thomas Wanner



February 28, 2026

Contents

Part I

Overview

Chapter 1

ConleyDynamics.jl

Conley index and multivector fields for Julia.

1.1 Introduction

ConleyDynamics.jl is a Julia package for studying combinatorial multivector fields using Conley theory. The multivector fields can be studied on arbitrary Lefschetz complexes, which include both simplicial and cubical complexes as important special cases. The concept of combinatorial multivector field generalizes Forman vector fields, which were originally introduced to study Morse theory in a discrete combinatorial setting.

Note

This documentation is also available in PDF format: ConleyDynamics.pdf.

1.2 Features

- Data structures for Lefschetz complexes, in particular simplicial and cubical complexes.
- Classical Forman combinatorial vector fields and multivector fields are supported.
- Computation of Conley indices, connection matrices, and Conley-Morse graphs.
- Basic homology algorithms over finite fields and the rationals, including persistent homology and relative homology.
- Algorithms rely on a built-in sparse matrix implementation which is geared towards computations over finite fields and the rationals.

1.3 Installation

To use ConleyDynamics.jl please install Julia 1.10 or higher. See https://julialang.org/downloads/ for instructions on how to obtain Julia for your system.

At the Julia prompt simply type

```
julia> using Pkg; Pkg.add("ConleyDynamics")
```

After Julia has finished downloading and precompiling the package and all of its dependencies, you can start using it by typing

```
julia> using ConleyDynamics
```

1.4 Manual Outline

The Tutorial briefly explains how to get started with ConleyDynamics.jl. More details, including on the underlying mathematics, are provided in the following three sections, which cover Lefschetz complexes, homology, and Conley theory including connection matrices. After a discussion of all included examples in the Examples section, the manual concludes with a description of the sparse matrix format underlying the package.

1.5 Citation

If you use ConleyDynamics.jl, please cite it. There is a JOSS paper published at https://doi.org/10.21105/joss.08085, see [Wan25]. The BibTeX entry for this paper is:

```
@article{wanner:25a,
   author = {Thomas Wanner},
   title = {Conley{D}ynamics.jl: {A} {J}ulia package for multivector
            dynamics on {L}efschetz complexes},
   journal = {Journal of Open Source Software},
   doi = {10.21105/joss.08085},
   volume = {10},
   number = {111},
   pages = {8085},
   url = {https://joss.theoj.org/papers/10.21105/joss.08085},
   year = {2025}
   }
```

1.6 License

ConleyDynamics.jl is provided under an MIT license.

Part II

Manual

Chapter 2

Tutorial

This tutorial explains the basic usage of the main components of ConleyDynamics.jl. It is not meant to be exhaustive, and more details will be provided in the more individualized sections. Also, precise mathematical definitions will be delayed until then. The presented examples are taken from the paper [BKMW20] and the book [MW25], with minor modifications. See also [Wan25].

2.1 Creating Simplicial Complexes

The fundamental mathematical object for ConleyDynamics.jl is a Lefschetz complex [Lef42]. For now we note that both simplicial complexes and cubical complexes are special cases, and ConleyDynamics.jl provides convenient interfaces for generating them.

For the sake of simplicity, this tutorial only considers the case of a simplicial complex. Recall that an abstract simplicial complex K is just a collection of finite sets, called simplices, which is closed under taking subsets. In other words, every subset of a simplex is again a simplex. Each simplex σ has an associated dimension $\dim \sigma$, which is one less than the number of its elements. One usually calls simplices of dimension 0 vertices, edges have dimension 1, and simplices of dimension 2 are triangles. It follows easily from these definitions that every simplex is the union of its vertices. The following notions associated with simplicial complexes are important for this introduction:

- A face of a simplex is any of its subsets. Notice that every simplex is a face of itself, and it is the only face that has the same dimension as the simplex. Faces whose dimension is strictly smaller are referred to as proper faces.
- The boundary of a simplex σ is the collection of all proper faces of σ. For a triangle, this amounts to all three edges and all three vertices which are part of it.
- A facet of a simplex σ is any face τ with dimension $\dim \tau = \dim \sigma - 1$. Notice that the facets of a simplex are the faces in its boundary of maximal dimension.
- The closure of a subset K_0 of a simplicial complex K consists of the collection of all faces of simplices in K_0, and we denote the closure by $\operatorname{cl} K_0$.
- A subset K_0 of a simplicial complex K is called closed, if it equals its closure. In other words, K_0 is closed if and only if for every simplex σ in K_0 all of its boundary simplices are part of K_0 as well. Thus, a closed subset of a simplicial complex is a simplicial complex in its own right.

In ConleyDynamics.jl it is easy to generate a simplicial complex. This requires two objects:

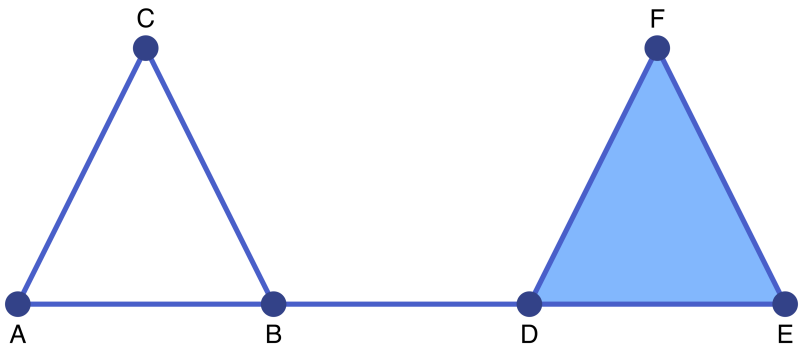


Figure 2.1: A first simplicial complex

- The vertices are described by a vector `labels` of string labels for the vertices of the simplicial complex. Thus, the length of the vector equals the number of vertices, and the k-th entry is the label for the k-th vertex.
- In addition, a second vector `simplices` has to describe enough simplices so that the simplicial complex is determined. This object is a vector of vectors, and the vector `simplices[k]` describes the index values of all the vertices in the k-th simplex. These indices are precisely the corresponding locations of the vertices in `labels`.

Simplices via labels

It is also possible to specify the list of simplices using a `Vector{Vector{String}}`, i.e., as a vector of string vectors. In this case, the entry `simplices[k]` is a list of the labels of the vertices.

Watch the label length

It is expected that the labels in `labels` all have the same number of characters. This is due to the fact that when creating the simplicial complex, ConleyDynamics.jl automatically creates labels for each of the simplices in K, by concatenating the vertex labels. Not using a fixed label size could lead to ambiguities, and will therefore raise an error message.

The following first example creates a simple simplicial complex. The complex is shown in the above figure, and it has six vertices which we label by the first six letters.

```
labels = ["A","B","C","D","E","F"]
simplices = [["A","B"],["A","C"],["B","C"],["B","D"],["D","E","F"]]
sc = create_simplicial_complex(labels,simplices)
fieldnames(typeof(sc))
```

```
(:labels, :dimensions, :boundary, :ncells, :dim, :indices)
```

Based on the simplex specifications, the generated simplicial complex K consists of three edges connecting each of the vertices A, B, and C, a two-dimensional triangle DEF, as well as the edge BD which connects the

triangle boundary and the filled triangle. The created struct `sc` is of type `LefschetzComplex`, with fieldnames as indicated in the above output. The number of cells in the complex can be seen as follows:

```
println(sc.ncells)
```

```
14
```

Note that the final simplicial complex has a total of seven edges, since also the edges of DEF are part of the simplicial complex. They are automatically generated by `create_simplicial_complex`. The dimension of K is the largest simplex dimensions, and can be recalled via

```
println(sc.dim)
```

```
2
```

The `sc` struct contains a vector of labels, which in this case takes the form

```
println(sc.labels)
```

```
["A", "B", "C", "D", "E", "F", "AB", "AC", "BC", "BD", "DE", "DF", "EF", "DEF"]
```

Finally, the Lefschetz complex data structure for our simplicial complex K includes the dimensions for the corresponding cells in the integer vector `sc.dimensions`, a dictionary `sc.indices` which associates each simplex label with its integer index, and the boundary map `sc.boundary` which will be described in more detail in Lefschetz Complexes. The latter map is internally stored as a sparse matrix over either a finite field or over the rationals. See also the discussion of Sparse Matrices.

2.2 Computing Homology and Persistence

Any simplicial complex, and in fact any Lefschetz complex, has an associated homology. Informally, homology describes the connectivity structure of the simplicial complex. More precisely, the homology consists of a sequence of integers, called the Betti numbers, which are indexed by dimension. There are Betti numbers $\beta_k(K)$ for every $k = 0, \ldots, \dim K$. The zero-dimensional Betti number $\beta_0(K)$ gives the number of connected components of K, while $\beta_1(K)$ counts the number of independent loops that can be found in K. Finally, $\beta_2(K)$ equals the number of cavities. In our case, we have

```
homology(sc)
```

```
3-element Vector{Int64}:
 1
 1
 0
```

This means that the simplicial complex K has one component, as well as one loop, and no cavities. The function `homology` returns a vector of integers, whose k-th entry is $\beta_{k-1}(K)$. We would like to point out that in ConleyDynamics.jl all homology computations are performed over fields, and therefore homology is completely described by the Betti numbers. Two types of fields are supported, and they are selected by the characteristic `p` in the sparse boundary matrix:

- If `p=0`, then the homology computation uses the field of rational numbers.
- For any prime number `p`, homology is determined over the finite field $GF(p)$ with `p` elements.

ConleyDynamics.jl also allows for the computation of relative homology. In the case of relative homology, together with the simplicial complex K one has to specify a closed subcomplex K_0. Intuitively, the relative homology $H_*(K, K_0)$ is the homology of a new space, which is obtained from K by identifying K_0 to a single point, and then decreasing the zero-dimensional Betti number by 1. Consider for example the following command:

```
relative_homology(sc, [1,6])
```

```
3-element Vector{Int64}:
 0
 2
 0
```

In this case, the subcomplex K_0 consists of the two vertices `A` and `F`, which are therefore glued together. This leads to zero Betti numbers in dimension 0 and 2 (remember that the zero-dimensional Betti number is decreased by 1!), and a one-dimensional Betti number of 2. The latter is increased by one since we obtain a second loop by moving from `A` to `F = A` along the edges `AB`, `BD`, and `DF`. Another example is the following:

```
relative_homology(sc, ["DE","DF","EF"])
```

```
3-element Vector{Int64}:
 0
 1
 1
```

Now the subcomplex K_0 consists of the edges `DE`, `DF`, and `EF` – together with the three vertices `D`, `E`, and `F` which are automatically added by `relative_homology`. Identifying them all to one point creates a hollow two-dimensional sphere, and the relative Betti numbers reflect that fact.

As the above two examples demonstrate, the subcomplex can be specified either as a list of simplex indices, or through the simplex labels. Moreover, the specfied subspace simplex list is automatically extended by `relative_homology` to include all simplex faces, i.e., it computes the simplicial closure to arrive at a closed subcomplex. Finally, note that the subcomplex can be empty:

```
relative_homology(sc, [])
```

```
3-element Vector{Int64}:
 1
 1
 0
```

As expected, in this case one obtains the standard homology of `sc`.

In addition to regular and relative homology, ConleyDynamics.jl can also compute persistent homology. For this, one has to specify a filtration of closed Lefschetz complexes

$$K_1 \subset K_2 \subset \ldots \subset K_m.$$

Persistent homology tracks the appearance and disappearance (also often called the birth and death) of topological features as one moves through the complexes in the filtration. In ConleyDynamics.jl, one can specify a Lefschetz complex filtration by assigning the integer k to each simplex that first appears in K_k. Moreover, it is expected that $K_m = K$. Then the persistent homology is computed via the following command:

```
filtration = [1,1,1,2,2,2,1,1,1,3,2,2,2,4]
phsingles, phpairs = persistent_homology(sc, filtration)
```

```
([[1], [1], Int64[]], [[(2, 3)], [(2, 4)], Tuple{Int64, Int64}[]])
```

The function returns the persistence intervals, which give the birth and death indices of each topological feature in each dimension. There are two types of intervals:

- Intervals of the form $[a, \infty)$ correspond to topological features that first appear in K_a and are still present in the final complex. The starting indices of such features in dimension `k` are contained in the list `phsingles[k+1]`.
- Intervals of the form $[a, b)$ correspond to topological features that first appear in K_a and first disappear in K_b. The corresponding pairs `(a,b)` in dimension `k` are contained in the list `phpairs[k+1]`.

In our above example, one observes intervals $[1, \infty)$ in dimensions zero and one – and these correspond to a connected component and the loop generated by the edges `AB`, `AC`, and `BC`. These appear first in K_1 and are still present in K_4. The interval $[2, 3)$ in dimension zero represents the new component created by K_2, and it disappears through merging with the older component from K_1 when the edge `BD` is introduced with K_3. Similarly, the interval $[2, 4)$ in dimension one is the loop created by the triangle `DE`, `DF`, and `EF` in K_2, which disappears with the introduction of the triangle `DEF` in K_4. Note that the interval death times respect the elder rule: When for example a component disappears through merging, the younger interval gets killed, and the older one continues to live. Similarly in higher dimensions.

2.3 Forman Vector Fields

The main focus of ConleyDynamics.jl is on the study of combinatorial topological dynamics on Lefschetz complexes. While the phase space as Lefschetz complex has been discussed above, albeit only for the special

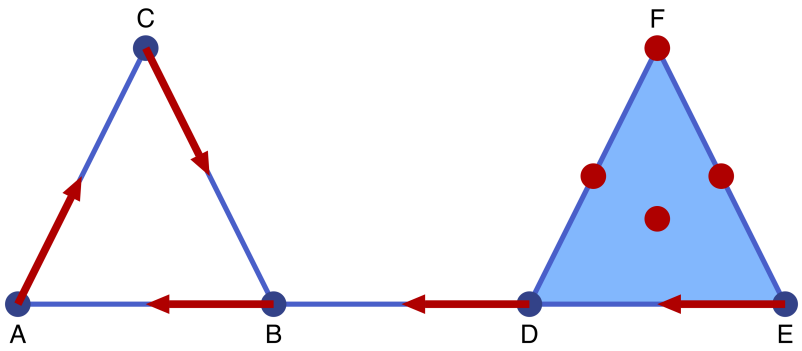


Figure 2.2: A first Forman vector field

case of a simplicial complex, the dynamics part can be given in the simplest form by a combinatorial vector field, also called a Forman vector field [For98a, For98b]. We will soon see that such vector fields are a more restrictive version of multivector fields, but they are easier to start with. The following command defines a simple Forman vector field on our sample simplicial complex K from above:

```
formanvf = [["A","AC"],["B","AB"],["C","BC"],["D","BD"],["E","DE"]]
```

```
5-element Vector{Vector{String}}:
 ["A", "AC"]
 ["B", "AB"]
 ["C", "BC"]
 ["D", "BD"]
 ["E", "DE"]
```

The Forman vector field `formanvf` is viualized in the accompanying figure.

According to the figure, a Forman vector field is comprised of arrows, as well as critical cells which are indicated by red dots. Every simplex of the underlying simplicial complex is either critical, or it is contained in a unique arrow. In other words, the collection of critical cells and arrows forms a partition of the simplicial complex K. Arrows always have to consist of precisely two simplices: The source of the arrow is a simplex σ^-, while its target is a second simplex σ^+. These two simplices have to be related in the sense that σ^- is a facet of σ^+.

As the above Julia code shows, a forman vector field is described by a vector of string vectors, where each of the latter contains the labels of the two simplices making up an arrow. Note that the critical cells are not explicitly listed, as any simplex of K that is not part of a vector is automatically assumed to be critical. Alternatively, one could define the Forman vector field as a `Vector{Vector{Int}}`, if the labels are replaced by the corresponding indices in `sc.indices`.

Intuitively, the visualization of our sample Forman vector field `formanvf` induces the following dynamical behavior on the simplicial complex `sc`:

- **Critical cells** can be though of as equilibrium states for the dynamics, i.e., they contain a stationary solution. However, depending on their dimension they can also exhibit nonconstant dynamics – which in backward time converges to the equilibrium, and in forward time flows towards the boundary of the simplex.

- **Arrow sources** always lead to flow into the interior of their target simplex σ^+.
- **Arrow targets** create flow towards the boundary of σ^+, except towards the source facet σ^-.

In the above figure, for example, the simplex EF is a critical cell, so it contains an equilibrium. At the same time, it also allows for flow towards the boundary, which consists of the vertices E and F. A solution flowing to the former then has to enter DE, flow through D to BD, before entering the periodic orbit given by

$$B \to AB \to A \to AC \to C \to BC \to B \to AB \to \dots$$

This heuristic description can be made precise. It was shown in [MW21] that for every Forman vector field on a simplicial complex there exists a classical dynamical system which exhibits dynamics consistent with the above interpretation.

2.4 Isolated Invariant Sets

The global dynamical behavior of a Forman vector field on a simplicial complex can be described by first decomposing it into smaller building blocks. An invariant set is a subset $S \subset K$ of the simplicial complex such that for every simplex $\sigma \in S$ there exists a solution through σ which is contained in S and which exists for all forward and backward time. In our example the following are sample invariant sets:

- Every critical cell σ by itself is an invariant set, since we can choose the constant solution σ in the above definition. Thus, also every union of critical cells is invariant.
- The periodic orbit $S_P = \{A, B, C, AB, AC, BC\}$ is an invariant set, since the periodic orbit mentioned earlier exists for all forward and backward time in S_P and passes through every simplex of the orbit.

While it is tempting to try to decompose the dynamics into invariant sets and "everything else", Conley realized that a better theory can be built around invariant sets which are isolated [Con78]. In our combinatorial setting, an isolated invariant set is an invariant set $S \subset K$ with the following two additional properties:

- The set S is locally closed, i.e., the associated set $\operatorname{mo} S = \operatorname{cl} S \setminus S$ is closed in the simplicial complex. Recall that the closure $\operatorname{cl} A$ of a set $A \subset K$ consists of all simplices which are subsets of simplices in A, and a set is closed if it equals its closure. The set $\operatorname{mo} S$ is called the mouth of S.
- The set S is compatible with the Forman vector field, i.e., the set is the union of critical cells and arrows. In other words, if one of the arrow ends is contained in S, then so is the other.

One can easily see that the periodic orbit S_P is an isolated invariant set, since it is compatible and closed – and therefore $\operatorname{mo} S_P = \emptyset$ is closed. Similarly, the single critical simplex $S_1 = \{DEF\}$ is an isolated invariant set, since in this case the set $\operatorname{mo} S_1 = \{D, E, F, DE, DF, EF\}$ is closed, and S_1 is compatible. On the other hand, the invariant set $S_2 = \{DEF, F\}$ is not an isolated invariant set, since the mouth $\operatorname{mo} S_2 = \{D, E, DE, DF, EF\}$ is not closed – despite the fact that S_2 is compatible. For an example of an invariant set which has a closed mouth but is not compatible, see [KMW16, Figure 5].

It follows from the definition of isolation that for every isolated invariant set $S \subset K$ the two sets $\operatorname{cl} S$ and $\operatorname{mo} S$ are closed, and that the latter is a (possibly empty) subset of the former. Thus, the relative homology of this pair is defined and we let

$$CH_*(S) = H_*(\operatorname{cl} S, \operatorname{mo} S)$$

denote the Conley index of the isolated invariant set. The Conley index can be computed using the command `conley_index`. For the three critical cells `F`, `DF`, and `DEF` one obtains the following Conley indices:

```
println(conley_index(sc, ["F"]))
println(conley_index(sc, ["DF"]))
println(conley_index(sc, ["DEF"]))
```

```
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
```

In other words, the Conley index of a critical cell of dimension k has Betti number $\beta_k = 1$, while the remaining Betti numbers vanish. This is precisely the relative homology of a k-dimensional sphere with respect to a point on the sphere. On the other hand, for the Conley index of the periodic orbit S_P one obtains:

```
conley_index(sc, ["AB", "AC", "BC", "A", "B", "C"])
```

```
3-element Vector{Int64}:
 1
 1
 0
```

This Conley index is nontrivial in dimensions 0 and 1. This is exactly the Conley index of an attracting periodic orbit in classical dynamics.

2.5 Connection Matrices

One of the main features of ConleyDynamics.jl is its capability to take a given combinatorial vector or multivector field on an arbitrary Lefschetz complex and determine its global dynamical behavior. This is done by computing the connection matrix, which in our setting is discussed in detail in [MW25]. For the sample simplicial complex `sc` and the Forman vector field `formanvf` the connection matrix information can be determined as follows:

```
cm = connection_matrix(sc, formanvf)
fieldnames(typeof(cm))
```

```
(:matrix, :columns, :poset, :labels, :morse, :conley, :complex)
```

This command calculates the connection matrix over the finite field $GF(2) = \mathbb{Z}_2$. The base field for this computation is determined by the data type of the boundary matrix in the underlying simplicial complex `sc`.

By default, if one uses the function `create_simplicial_complex` without specifying the field characteristic p, the simplicial complex is created over the finite field $\mathbb{Z}_2$, i.e., with p=2.

The `connection_matrix` function returns a struct which contains the following information regarding the global dynamics of the combinatorial dynamical system:

- The field `cm.morse` contains the Morse decomposition of the Forman vector field. This is a collection of isolated invariant sets which capture all recurrent behavior. Outside of these sets, the dynamics is gradient-like, i.e., it moves from one Morse set to another.
- Since each of the Morse sets is an isolated invariant set, they all have an associated Conley index. These are contained in the field `cm.conley`.
- In addition, the struct `cm` contains information on the actual connection matrix in the field `cm.matrix`. While the field contains the matrix, the rows and columns of the connection matrix correspond to the simplices in the underlying simplicial complex `sc` listed in `cm.labels`. These simplices represent the basis for the homology groups of all the Morse sets. Moreover, a nonzero entry in the connection matrix indicates that there has to be a connecting orbit between the Morse set containing the column label and the Morse set containing the row label.

The remaining field names of the struct `cm` are described in the section on Conley Theory.

For our example system, the Morse sets are given by

```
cm.morse
```

```
5-element Vector{Vector{String}}:
 ["A", "B", "C", "AB", "AC", "BC"]
 ["F"]
 ["DF"]
 ["EF"]
 ["DEF"]
```

There are five of them: The stable periodic orbit S_P mentioned earlier, the stable critical state F, the unstable equilibria DF and EF, as well as the two-dimensional unstable critical cell DEF. The associated Conley indices are

```
cm.conley
```

```
5-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [1, 0, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 0, 1]
```

Clearly these indices are exactly as described in the homology section, since the underlying field is still $\mathbb{Z}_2$, as determined by `sc`. For an example which involves computations over different fields, which also lead to different Conley indices, we refer to the function `example_moebius`.

Finally, the connection matrix itself is contained in `cm.matrix`. Since internally the connection matrix is stored in a sparse format, we display it after conversion to a full matrix:

```
full_from_sparse(cm.matrix)
```

```
6×6 Matrix{Int64}:
 0  0  0  1  1  0
 0  0  0  0  0  0
 0  0  0  1  1  0
 0  0  0  0  0  1
 0  0  0  0  0  1
 0  0  0  0  0  0
```

In order to see which simplices represent the columns of the matrix, we use the command

```
println(cm.labels)
```

```
["A", "AC", "F", "DF", "EF", "DEF"]
```

The right-most column contains two nonzero entries, and they imply that there are connecting orbits between the critical cell DEF and the two critical cells DF and EF, respectively. The second-to-last column establishes connecting orbits originating from EF. One of these ends at the critical vertex F, while the other one leads to A. Notice, however, that since A is part of the Morse set S_P, i.e., the periodic orbit, this second nonzero entry in the column implies the existence of a heteroclinic orbit between the equilibrium and the complete periodic solution. Similarly, there are connections between DF and both F and the periodic orbit, in view of the fourth column of the connection matrix.

A description of the remaining fields of `cm` can also be found in the API entry for `connection_matrix`. We would like to emphasize again that internally, all computations necessary for finding the connection matrix are performed automatically over the rationals or over the finite field $GF(p)$. The choice depends on the data type of the boundary matrix for the underlying Lefschetz complex, in this case the simplicial complex `sc`.

2.6 Multivector Fields

As second example of this tutorial we turn our attention to the logo of ConleyDynamics.jl. It shows a simple multivector field on a simplicial complex, and both the simplicial complex `sclogo` and the multivector field `mvflogo` can be defined using the commands

```
labels = ["A","B","C","D"]
simplices = [["A","B","C"],["B","C","D"]]
sclogo = create_simplicial_complex(labels,simplices)
mvflogo = [["A","AB"],["C","AC"],["B","BC","BD","BCD"]]
```

```
3-element Vector{Vector{String}}:
 ["A", "AB"]
 ["C", "AC"]
 ["B", "BC", "BD", "BCD"]
```

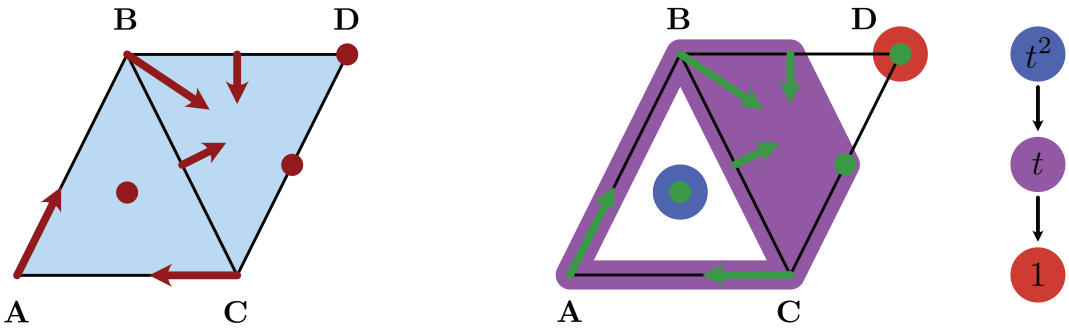


Figure 2.3: The logo multivector field

This example is taken from [MW25, Figure 2.1], and is visualized in the accompanying figure.

The multivector field `mvflogo` clearly has a different structure from the earlier Forman vector field. While the latter consists exclusively of arrows and critical cells, the former is made up of multivectors. In this context a multivector is a collection of simplices which form a locally closed set, as defined earlier in the tutorial. One can show that in the case of a simplicial complex, this is equivalent to requiring that if $\sigma_1 \subset \sigma_2$ are two simplices in the multivector, then so are all simplices τ with $\sigma_1 \subset \tau \subset \sigma_2$. In other words, multivectors are convex with respect to simplex inclusion, i.e., with respect to the face relation. A multivector field is then a partition of the simplicial complex into multivectors. See [LKMW23] for more details.

It is not difficult to see that every Forman vector field is a multivector field. Every critical cell consists of just one simplex, so it trivially satisfies the above convexity condition. In addition, the two simplices contained in an arrow do not allow for any simplex $\sigma^- \subset \tau \subset \sigma^+$ apart from $\tau = \sigma^\pm$. As in the case of Forman vector fields, multivector fields in ConleyDynamics.jl only need to list multivectors containing at least two simplices. Any simplex not contained on the list automatically gives rise to a one-element multivector.

One important difference between Forman vector fields and multivector fields is the definition of criticality. In the multivector field case, the types of multivectors are distinguished as follows:

- A multivector V is called critical, if the relative homology $H_*(\operatorname{cl} V, \operatorname{mo} V)$ is not trivial, i.e., at least one Betti number is nonzero.
- A multivector V is called regular, if the relative homology $H_*(\operatorname{cl} V, \operatorname{mo} V)$ is trivial, i.e., it vanishes in all dimensions.

One can show that in the case of a Forman vector field, critical cells are always critical in the above sense, while arrows are always regular. In our above example `mvflogo`, all three multivectors which are not singletons are regular. For example, the following computation shows that the cell ABC is a critical cell:

```
cl1, mo1 = lefschetz_clomo_pair(sclogo, ["ABC"])
relative_homology(sclogo, cl1, mo1)
```

```
3-element Vector{Int64}:
 0
 0
 1
```

The first command creates the closure-mouth pair associated with the cell ABC, i.e., the variable `cl1` is the closed triangle, while `mo1` is the closed boundary of the triangle. The next command determines the relative homology. Notice that this employs another method under the name `relative_homology`, in contrast to the one used earlier in this tutorial. For more details, see Homology Functions.

Alternatively, since every multivector is locally closed, one can also use the function `conley_index` for the same computation:

```
conley_index(sclogo, ["ABC"])
```

```
3-element Vector{Int64}:
 0
 0
 1
```

Similarly, the next sequence of commands verifies that the third nontrivial multivector `mvflogo[3]` is indeed a regular multivector:

```
cl2, mo2 = lefschetz_clomo_pair(sclogo, mvflogo[3])
relative_homology(sclogo, cl2, mo2)
```

```
3-element Vector{Int64}:
 0
 0
 0
```

The global dynamics can again be determined using the function `connection_matrix`:

```
cmlogo = connection_matrix(sclogo, mvflogo)
cmlogo.morse
```

```
3-element Vector{Vector{String}}:
 ["D"]
 ["A", "B", "C", "AB", "AC", "BC", "BD", "CD", "BCD"]
 ["ABC"]
```

As it turns out, our logo gives rise to three Morse sets, which in fact partition the simplicial complex. Their Conley indices are given by

```
cmlogo.conley
```

```
3-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 1, 0]
 [0, 0, 1]
```

Finally, the connection matrix has the form

```
full_from_sparse(cmlogo.matrix)
```

```
3×3 Matrix{Int64}:
 0  0  0
 0  0  1
 0  0  0
```

Notice that in this example, only the connection between the Morse set ABC and the large index 1 Morse set comprising almost all of the simplicial complex can be detected algebraically. In fact, there are two connections between the large Morse set and the stable equilibrium D, and they cancel algebraically.

2.7 Analyzing Planar Vector Fields

Our third and last example of the tutorial briefly indicates how ConleyDynamics.jl can be used to analyze the global dynamics of certain planar ordinary differential equations. For this, consider the planar system given by

$$\begin{aligned} \dot{x}_1 &= x_1\left(1 - x_1^2 - 3x_2^2\right) \\ \dot{x}_2 &= x_2\left(1 - 3x_1^2 - x_2^2\right) \end{aligned}$$

The right-hand side of this vector field can be implemented using the Julia function

```
function planarvf(x::Vector{Float64})
    #
    # Sample planar vector field with nontrivial Morse decomposition
    #
    x1, x2 = x
    y1 = x1 * (1.0 - x1*x1 - 3.0*x2*x2)
    y2 = x2 * (1.0 - 3.0*x1*x1 - x2*x2)
    return [y1, y2]
end
```

```
planarvf (generic function with 1 method)
```

To analyze the global dynamics of this vector field, we first create a Delaunay triangulation of the square $[-3/2, 3/2]^2$ using the commands

```
lc, coords = create_simplicial_delaunay(300, 300, 10, 30);
coordsN = convert_planar_coordinates(coords,[-1.5,-1.5], [1.5,1.5]);
cx = [c[1] for c in coordsN];
(minimum(cx), maximum(cx))
```

```
(-1.5, 1.5)
```

The first command generates the triangulation in a square box with side length 300, while trying to keep a minimum distance of about 10 between vertices. Once this has been accomplished, the second command transforms the coordinates to the desired square domain. As the last two commands show, the resulting x-coordinates do indeed lie between -3/2 and 3/2.

Next we can create a multivector field which describes the flow behavior through the edges of the triangulation. Basically, for each edge which is traversed in only one direction, the corresponding multivector respects this unidirectionality, while non-transverse edges lead to multivectors which allow for flow in both directions between the adjacent triangles. This is achieved with the commands

```
mvf = create_planar_mvf(lc, coordsN, planarvf);
mvf[1:3]
```

```
3-element Vector{Vector{Int64}}:
 [1, 610, 612, 2387]
 [2, 616, 617, 2392]
 [3, 622, 624, 2397]
```

The first command generates the multivector field, while the second one merely displays the first three resulting multivectors. Note that if the discretization is too coarse, this might lead to large multivectors that cannot resolve the underlying dynamics. In our case, we can analyze the global dynamics of the created multivector field using the commands

```
cm = connection_matrix(lc, mvf);
cm.conley
```

```
9-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [1, 0, 0]
 [0, 1, 0]
 [1, 0, 0]
 [0, 1, 0]
 [1, 0, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 0, 1]
```

As the output shows, this planar system has nine isolated invariant sets:

- One unstable equilibrium of index 2,
- four unstable equilibria of index 1,
- and four stable equilibria.

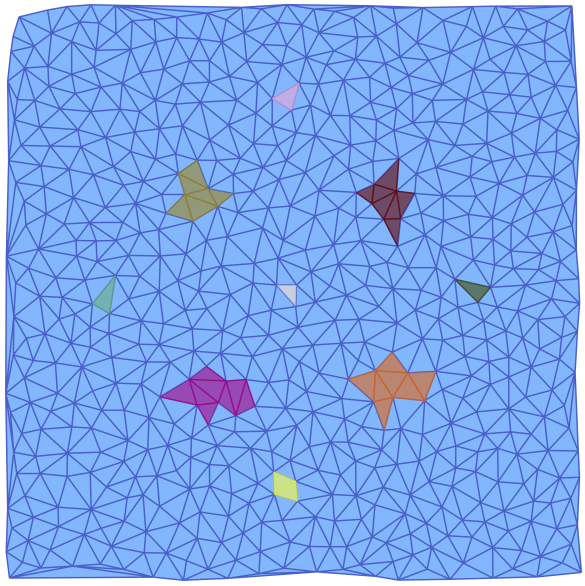


Figure 2.4: Morse sets of a planar vector field

More precisely, this computation does not in fact establish the existence of these equilibria, but of corresponding isolated invariant sets which have the respective Conley indices. The connection matrix is given by

```
full_from_sparse(cm.matrix)
```

```
9×9 Matrix{Int64}:
 0  0  1  0  0  0  1  0  0
 0  0  1  0  1  0  0  0  0
 0  0  0  0  0  0  0  0  1
 0  0  0  0  1  0  0  1  0
 0  0  0  0  0  0  0  0  1
 0  0  0  0  0  0  1  1  0
 0  0  0  0  0  0  0  0  1
 0  0  0  0  0  0  0  0  1
 0  0  0  0  0  0  0  0  0
```

It shows that there are twelve connecting orbits that are forced by the algebraic topology. Finally, we can visualize the Morse sets using the command

```
fname = "tutorialplanar.pdf"
plot_planar_simplicial_morse(lc, coordsN, fname, cm.morse, pv=true)
```

2.8 References

See the full bibliography for a complete list of references cited throughout this documentation. This section cites the following references:

[BKMW20] B. Batko, T. Kaczynski, M. Mrozek and T. Wanner. Linking combinatorial and classical dynamics: Conley index and Morse decompositions. Foundations of Computational Mathematics **20**, 967–1012 (2020).

[Con78] C. Conley. Isolated Invariant Sets and the Morse Index (American Mathematical Society, Providence, R.I., 1978).

[For98a] R. Forman. Combinatorial vector fields and dynamical systems. Mathematische Zeitschrift **228**, 629–681 (1998).

[For98b] R. Forman. Morse theory for cell complexes. Advances in Mathematics **134**, 90–145 (1998).

[KMW16] T. Kaczynski, M. Mrozek and T. Wanner. Towards a formal tie between combinatorial and classical vector field dynamics. Journal of Computational Dynamics **3**, 17–50 (2016).

[Lef42] S. Lefschetz. Algebraic Topology. Vol. 27 of American Mathematical Society Colloquium Publications (American Mathematical Society, New York, 1942).

[LKMW23] M. Lipinski, J. Kubica, M. Mrozek and T. Wanner. Conley-Morse-Forman theory for generalized combinatorial multivector fields on finite topological spaces. Journal of Applied and Computational Topology **7**, 139–184 (2023).

[MW21] M. Mrozek and T. Wanner. Creating semiflows on simplicial complexes from combinatorial vector fields. Journal of Differential Equations **304**, 375–434 (2021).

[MW25] M. Mrozek and T. Wanner. Connection Matrices in Combinatorial Topological Dynamics. SpringerBriefs in Mathematics (Springer-Verlag, Cham, 2025).

[Wan25] T. Wanner. ConleyDynamics.jl: A Julia package for multivector dynamics on Lefschetz complexes. Journal of Open Source Software **10**, 8085 (2025).

Chapter 3

Lefschetz Complexes

The fundamental structure underlying the functionality of ConleyDynamics.jl is a Lefschetz complex. It provides us with the basic model of phase space for combinatorial topological dynamics. In view of the combinatorial, and therefore discrete, character of the dynamical behavior, a Lefschetz complex is not a typical phase space in the sense of classical dynamics. While the latter one is usually a Euclidean space, a Lefschetz complex is basically a combinatorial model of it. In the following, we provide its precise mathematical definition, and explain how it can be created and modified within the package. We also discuss two important special cases, namely simplicial complexes and cubical complexes.

3.1 Basic Lefschetz Terminology

The original definition of a Lefschetz complex can be found in [Lef42], where it was simply referred to as a complex.

Definition: Lefschetz complex

Let F denote an arbitrary field. Then a pair (X, κ) is called a Lefschetz complex over F if $X = (X_k)_{k\in\mathbb{N}_0}$ is a finite set with $\mathbb{N}_0$-gradation, and $\kappa : X \times X \to F$ is a mapping such that

$$\kappa(x, y) \neq 0 \quad \text{implies} \quad x \in X_k \quad \text{and} \quad y \in X_{k-1},$$

and such that for any $x, z \in X$ one has

$$\sum_{y\in X} \kappa(x, y)\kappa(y, z) = 0 \, .$$

The elements of X are referred to as cells, the value $\kappa(x, y) \in F$ is called the incidence coefficient of the cells x and y, and the map κ is the incidence coefficient map. In addition, one defines the dimension of a cell $x \in X_k$ as the integer k, and denotes it by $k = \dim x$. Whenever the incidence coefficient map is clear from context, we often just refer to X as the Lefschetz complex.

At first glance the above definition can seem daunting. However, it is based on a straightforward geometric idea. A Lefschetz complex is a structure that is built from elementary building blocks called cells. Each cell has a dimension associated with it, and it is topologically an open ball of this dimension. Thus, cells of dimension

zero are points, also called vertices. Cells of dimension one are open curve segments, which we call edges, and two-dimensional cells are called faces and take the form of open two-dimensional membranes.

The incidence coefficient map encodes how these cells are glued together to form the Lefschetz complex X. In order to shed more light on this, consider the boundary map ∂ which is defined on cells via

$$\partial x = \sum_{y \in X} \kappa(x, y) y \ .$$

This map sends a cell x of dimension k to a specific linear combination of cells of dimension $k-1$, called the boundary of x. By using ideas from linear algebra, the boundary map can be extended to map a general linear combination of k-dimensional cells to the corresponding linear combination of the separate boundaries. For example, if one chooses the field $F = \mathbb{Q}$ of rationals, one has $\partial(x_1 - 2x_2) = \partial x_1 - 2\partial x_2$. Notice that using this extended definition of the boundary map, one can rewrite the summation condition in the definition of a Lefschetz complex in the equivalent form

$$\partial(\partial x) = 0 \quad \text{for all cells} \quad x \in X \ .$$

In other words, the boundary of any cell is itself boundaryless.

With the help of the boundary map, one can often infer the overall geometric structure of a Lefschetz complex X. For this, think of a Lefschetz complex as being build from the ground up in the following way. First, start by putting down all vertices of X at different locations in some ambient space. Since the boundary of each one-dimensional cell is made up of a linear combination of vertices, one can then add a curve segment for each one-dimensional cell, which connects the vertices in its boundary. Note that in the general version of a Lefschetz complex it is possible that an edge has only one vertex in its boundary, or maybe even none, and in these cases the edge is either only connected to the one boundary vertex, or it is an open curve segment connected to no vertex at all, respectively. Continue in this fashion to add two-dimensional faces to fill in the space between the edges in its boundary, and so on for higher dimensions. Needless to say, in the case of a general complicated Lefschetz complex this procedure is of limited use, since the boundary of a cell can be an arbitrary linear combination of cells, with coefficients that can be any nonzero numbers in the field F. Yet, in many simple cases the above intuition is sufficient.

In addition to the Lefschetz complex definition, there are a handful of other concepts which will be important for our discussion of Lefschetz complexes. Specifically, the following notions are important:

- A facet of a cell $x \in X$ is any cell y which satisfies $\kappa(x, y) \neq 0$.
- One can define a partial order on the cells of X by letting $x \leq y$ if and only if for some integer $n \in \mathbb{N}$ there exist cells $x = x_1, \ldots, x_n = y$ such that x_k is a facet of x_{k+1} for all $k = 1, \ldots, n-1$. It is not difficult to show that this defines a partial order on X, i.e., this relation is reflexive, antisymmetric, and transitive. We call this partial order the face relation. Moreover, if $x \leq y$ then x is called a face of y.
- A subset $C \subset X$ of a Lefschetz complex is called closed, if for every $x \in C$ all the faces of the cell x are also contained in the subset C.
- The closure of a subset $C \subset X$ is the collection of all faces of all cells in C, and it is denoted by $\operatorname{cl} C$. Thus, a subset of a Lefschetz complex is closed if and only if it equals its closure.
- A subset $S \subset X$ is called locally closed, if its mouth $\operatorname{mo} S = \operatorname{cl} S \setminus S$ is closed. Note that every closed set is automatically locally closed, but the reverse implication is usually false.

While the first two points merely introduce notation for describing the combinatorial boundary of cells, the remaining three points establish important topological concepts. In fact, the above definition of closedness defines a topology on the Lefschetz complex X, which is the so-called Alexandrov topology from [Ale37]. As usual in the field of topology, a subset of a Lefschetz complex will be called open, if and only if its complement is closed.

We would like to point out that while the concept of local closedness is rarely considered in standard topology courses, it is of utmost important for the study of combinatorial topological dynamics. For the moment, we just mention the following result:

Theorem: Lefschetz subcomplexes

Let X be a Lefschetz complex over a field F, and let $\kappa : X \times X \to F$ denote its incidence coefficient map. Then a subset $S \subset X$ is again a Lefschetz complex, with respect to the restriction of κ to $S \times S$, if the subset S is locally closed.

This result goes back to [MB09, Theorem 3.1]. In other words, in the category of Lefschetz complexes local closedness arises naturally. Due to its importance, we also mention the following two equivalent formulations:

- A subset $S \subset X$ is locally closed, if and only if it is the difference of two closed subsets of X.
- A subset $S \subset X$ is locally closed, if and only if it is an interval with respect to the face relation on X, i.e., whenever we have three cells with $S \ni x \leq y \leq z \in S$, then one has to have $y \in S$ as well.

The proof of these characterizations can be found in [MW25, Proposition 3.2.1] and [LKMW23, Proposition 3.10], respectively.

Lefschetz complexes are a very general mathematical concept, and they can be rather confusing at first sight. Nevertheless, they do encompass other complex types, which are more geometric in nature. As we already saw in the tutorial, every simplicial complex is automatically a Lefschetz complex, and we will further elaborate on this connection below. In addition, we will also demonstrate that cubical complexes are Lefschetz complexes. More general, any regular CW complex is a Lefschetz complex as well. For more details on this, we refer to the definition in [Mas91] and the discussion in [DKMW11].

3.2 Lefschetz Complex Data Structure

For the efficient and easy manipulation of Lefschetz complexes in ConleyDynamics.jl we make use of a specific composite data type:

`ConleyDynamics.LefschetzComplex` – Type.

```
LefschetzComplex
```

Collect the Lefschetz complex information in a struct.

The struct is created via the following fields:

- `labels::Vector{String}`: Vector of labels associated with cell indices
- `dimensions::Vector{Int}`: Vector cell dimensions
- `boundary::SparseMatrix`: Boundary matrix, columns give the cell boundaries

It is expected that the dimensions are given in increasing order, and that the square of the boundary matrix is zero. Otherwise, exceptions are raised. In addition, the following fields are created during initialization:

- `ncells::Int`: Number of cells
- `dim::Int`: Dimension of the complex
- `indices::Dict{String,Int}`: Dictionary for finding cell index from label

The coefficient field is specified by the boundary matrix.

Warning

Note that the constructor does not check whether the boundary matrix squares to zero. It is the responsibility of the user to ensure that!

source

The fields of this struct relate to the mathematical definition of a Lefschetz complex X in the following way:

- Internally, every cell of the Lefschetz complex is represented by an integer between 1 and the total number of cells. However, in order to make it easier to interpret the results of computations, each cell in a Lefschetz complex has to also be given a label. These labels are contained in the field `labels::Vector{String}`, where `labels[k]` gives the label of cell k.
- The vector `dimensions` is a `Vector{Int}` and collects the dimensions of the cells. In other words, the cell which is indexed by the integer k has dimension `dimensions[k]`. It is expected that the dimension vector is increasing, and the constructor method will verify this. Otherwise, an error is triggered.
- The incidence coefficient map κ is encoded in the sparse matrix `boundary`. This matrix is a square matrix with `ncells` rows and columns. The k-th column contains the incidence coefficients $\kappa(k,\cdot)$ in the sense that the entry in row m and column k equals the value $\kappa(k,m)$. Since for most Lefschetz complexes the majority of the incidence coefficients is zero, the matrix is represented using the sparse format `SparseMatrix`, which is described in more detail in Sparse Matrices. An exception is raised if the square of the boundary matrix is not zero.

When creating a Lefschetz complex, only the above three items have to be specified, as they define a unique Lefschetz complex X. In other words, a Lefschetz complex is generally created via the command

```
lc = LefschetzComplex(labels, dimensions, boundary)
```

During the construction of the Julia object, additional fields are initialized which simplify working with a Lefschetz complex:

- The integer `ncells` gives the total number of cells in X. Internally, these cells are numbered by integers ranging from 1 to `ncells`.
- The integer `dim` describes the overall dimension of the Lefschetz complex, which is the largest dimension of a cell.
- In order to easily determine the integer index for a cell with a specific label, the field `indices` contains a dictionary of type `Dict{String,Int}` which maps labels to indices. For example, if a cell has the label `"124.010"`, then the associated integer index is given by `indices["124.010"]`.

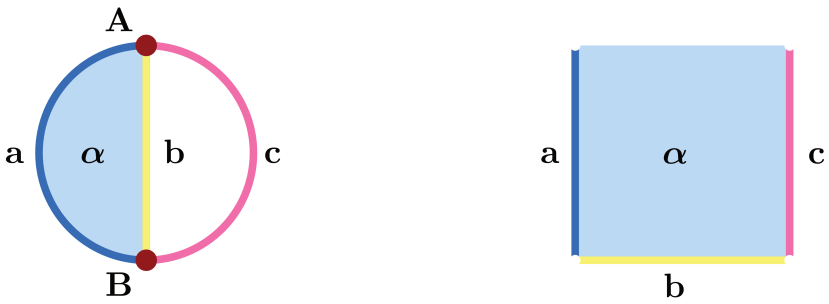


Figure 3.1: Two sample Lefschetz complexes

As mentioned above, note however that an object of type `LefschetzComplex` is created by passing only the first three the field items in the order given in `LefschetzComplex`. Consider for example the Lefschetz complex from Figure 2.4 in [MW25], see also the left complex in the next image. This complex consists of six cells with labels `A`, `B`, `a`, `b`, `c`, and `alpha`, and we initialize the vector of labels, the cell index dictionary, and the cell dimensions via the commands

```
ncL = 6
labelsL = Vector{String}(["A","B","a","b","c","alpha"])
cdimsL  = [0, 0, 1, 1, 1, 2]
```

The boundary matrix can then be defined using

```
bndmatrixL = zeros(Int, ncL, ncL)
bndmatrixL[[1,2],3] = [1; 1]     # a
bndmatrixL[[1,2],4] = [1; 1]     # b
bndmatrixL[[1,2],5] = [1; 1]     # c
bndmatrixL[[3,4],6] = [1; 1]     # alpha
bndsparseL = sparse_from_full(bndmatrixL, p=2)
```

Notice that we first create the matrix as a regular integer matrix, and then use the function `sparse_from_full` to turn it into sparse format over the field $GF(2)$ with characteristic `p = 2`. This is the most convenient method for small boundary matrices, yet for larger ones it is better to use the function `sparse_from_lists`. Finally, the Lefschetz complex is created using

```
lcL = LefschetzComplex(labelsL, cdimsL, bndsparseL)
```

Lefschetz complexes do not always have to contain cells of all dimensions. For example, the Lefschetz complex shown on the right side of the figure has no vertices, and it can be created using the commands

```
ncR = 4
labelsR  = Vector{String}(["a","b","c","alpha"])
cdimsR   = [1, 1, 1, 2]
bndmatrixR = zeros(Int, ncR, ncR)
bndmatrixR[[1,2,3],4] = [1; 1; 1]     # alpha
bndsparseR = sparse_from_full(bndmatrixR, p=2)
lcR = LefschetzComplex(labelsR, cdimsR, bndsparseR)
```

While Lefschetz complexes can always be created in ConleyDynamics.jl in this direct way, it is often more convenient to make use of special types, such as simplicial and cubical complexes, and then restrict the complex to a locally closed set using the function `lefschetz_subcomplex`. As an alternative, if one is interested in a fairly small Lefschetz complex over the field $GF(2)$, then the following special function can be used:

- `create_lefschetz_gf2` creates a Lefschetz complex over the two-element field $GF(2)$ by specifying its essential cells and boundaries. The input argument `defcellbnd` of the function has to be a vector of vectors. Each entry `defcellbnd[k]` then has to be of one of the following two forms:
  - `[String, Int, String, String, ...]`: The first `String` contains the label for the cell k, followed by its dimension in the second entry. The remaining entries are for the labels of the cells which make up the boundary.
  - `[String, Int]`: This shorther form is for cells with empty boundary. The first entry denotes the cell label, and the second its dimension.

  The cells of the resulting Lefschetz complex correspond to the union of all occurring labels. Cell labels that only occur in the boundary specification are assumed to have empty boundary, and they do not have to be specified separately in the second form above. However, if their boundary is not empty, they have to be listed via the above first form as well.

Using this function, our earlier Lefschetz complex `lcL` can be created using the commands

```
defcellbndL = [["a",1,"A","B"], ["b",1,"A","B"], ["c",1,"A","B"], ["alpha",2,"a","b"]]
lcL = create_lefschetz_gf2(defcellbndL)
```

while the Lefschetz complex `lcR` is defined via

```
defcellbndR = [["alpha",2,"a","b","c"], ["a",1], ["b",1], ["c",1]]
lcR = create_lefschetz_gf2(defcellbndR)
```

3.3 Simplicial Complexes

One of the earliest types of complexes that have been studied in topology are simplicial complexes. As already mentioned in the tutorial, an abstract simplicial complex X is a finite collection of finite sets, called simplices, which is closed under taking subsets. Each simplex σ has a dimension $\dim \sigma$, which is one less than the number of its elements.

In order to see why every simplicial complex is automatically a Lefschetz complex, we need to be able to define the incidence coefficient map κ. For this, we make use of some notions from [Mun84]. Let X_0 denote the collection of all vertices of the simplicial complex X. Then we use the notation

$$\sigma = [v_0, v_1, \ldots, v_d] \quad \text{with} \quad v_k \in X_0$$

to describe a d-dimensional simplex. Note that even though every simplex in X is just the set of its vertices, in the above representation we pick an order of the vertices, called an orientation of the simplex. This orientation can be chosen arbitrarily, and there are two equivalence classes of orientations. To get from one orientation to the other, one just has to exchange two vertices, and we write

$$[\ldots, v_i, \ldots, v_j, \ldots] \;=\; -[\ldots, v_j, \ldots, v_i, \ldots] \;.$$

For more complicated reorderings, one has to represent the corresponding vertex permutation as a sequence of such exchanges. Using these oriented simplices we can define the boundary operator

$$\partial\sigma \;=\; \partial\,[v_0, \ldots, v_d] \;=\; \sum_{i=0}^{d} (-1)^i\,[v_0, \ldots, \hat{v}_i, \ldots, v_d] \;,$$

where the notation $\hat{v}_i$ means that in the simplex behind the summation sign on the right-hand side the vertex v_i is omitted. For example, for a two-dimensional simplex one obtains

$$\partial\,[v_0, v_1, v_2] \;=\; [v_1, v_2] - [v_0, v_2] + [v_0, v_1] \;.$$

Thus, if one chooses a total order of all the vertices in the simplicial complex, and orients the individual simplices in such a way that their vertices are arranged using this overall order, then the incidence coefficient map is given by

$$\kappa\left([v_0, \ldots, v_i, \ldots, v_d]\,,\; [v_0, \ldots, \hat{v}_i, \ldots, v_d]\right) \;=\; (-1)^i \;.$$

If some or all of the simplices are represented by different orientations, one simply has to multiply the value $(-1)^i$ by the sign of a suitable vertex permutation. In either case, one can show that the so-defined map κ does indeed satisfy the definition of a Lefschetz complex. For more details, see [Mun84, Lemma 5.3].

In ConleyDynamics.jl there are three basic commands for defining a simplicial complex:

- `create_simplicial_complex` is the most general method, and it expects two input arguments. The first is usually called `labels`, and it has to have the data type `Vector{String}`. This vector lists the labels for each vertex. It is important that all of these labels have exactly the same number of characters. The second argument is usually called `simplices`, and it lists as many simplices as necessary for defining the underlying simplicial complex. This means that in practice one only needs to include the simplices which are not faces of higher-dimensional ones, see also the example below. The variable `simplices` can either be of type `Vector{Vector{String}}` or `Vector{Vector{Int}}`, depending on whether the vertices are identified via their labels or integer indices, respectively. Finally, the optional parameter `p` can be used to specify the underlying field for the boundary matrix. If `p` is a prime, then $F = GF(p)$, while for `p = 0` the function uses $F = \mathbb{Q}$. If the argument `p` is ommitted, the function defaults to `p = 2`.

- `create_simplicial_rectangle` expects two integer arguments `nx` and `ny`, and then creates a triangulation of the square $[0, nx] \times [0, ny]$ by subdividing every unit square into four triangles which meet at the center of the square. As before, the optional parameter `p` specifies the underlying field.

- `create_simplicial_delaunay` creates a planar Delaunay triangulation inside a planar rectangle. The function selects a random sample of points inside the box, while either trying to maintain a minimum distance between the points, or just using a prespecified number of points. More details on these two options can be found in the documentation for the function.

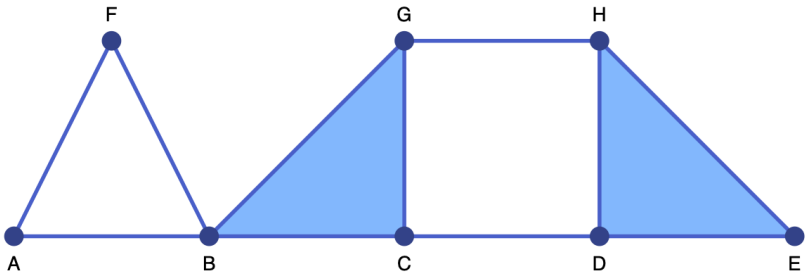


Figure 3.2: First sample simplicial complex

To illustrate the first of these functions, consider the commands

```
labels = ["A","B","C","D","E","F","G","H"]
simplices = [["A","B"],["A","F"],["B","F"],["B","C","G"],["D","E","H"],["C","D"],["G","H"]]
sc = create_simplicial_complex(labels,simplices)
```

These create the simplicial complex `sc`, in the form of a Lefschetz complex. Note that the above commands only specify the labels for the vertices. The labels for simplices of dimension at least one are automatically generated by concatenating the labels for their vertices, sorted in lexicographic order. This can be seen in the following Julia output:

```
julia> sc.labels[end-4:end]
5-element Vector{String}:
 "DH"
 "EH"
 "GH"
 "BCG"
 "DEH"
```

The simplicial complex `sc` can be visualized using the commands

```
coords = [[0,0],[2,0],[4,0],[6,0],[8,0],[1,2],[4,2],[6,2]]
ldir   = [3,3,3,3,3,1,1,1]
fname  = "lefschetzex2.pdf"
plot_planar_simplicial(sc,coords,fname,labeldir=ldir,labeldis=10,hfac=2,vfac=1.5,sfac=50)
```

Similarly, the commands

```
sc2, coords2 = create_simplicial_rectangle(5,2)
fname2 = "lefschetzex3.pdf"
plot_planar_simplicial(sc2,coords2,fname2,hfac=2.0,vfac=1.2,sfac=75)
```

define and illustrate a second simplicial complex, which triangularizes a rectangle in the plane.

For a demonstration of the Delaunay triangulation approach, please see Analyzing Planar Vector Fields.

In addition to the above methods, ConleyDynamics.jl also provides a few special simplicial complexes for illustration purposes:

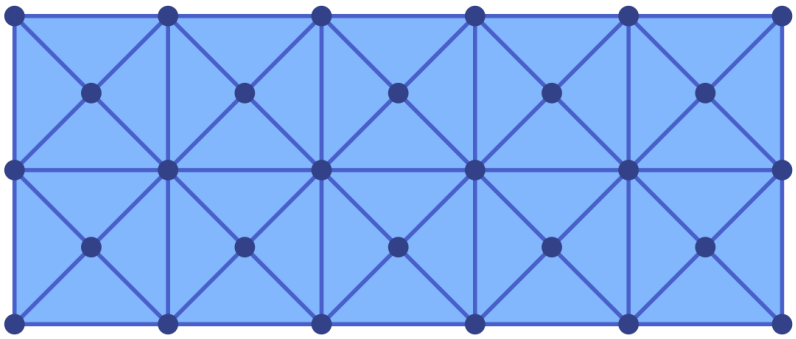


Figure 3.3: Second sample simplicial complex

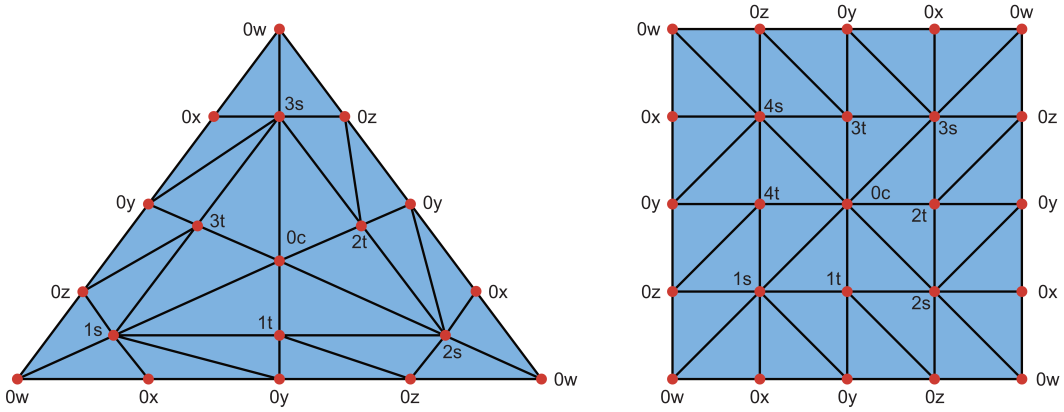


Figure 3.4: Simplicial torsion space

- `simplicial_torus` constructs a triangulation of the two-dimensional torus.
- `simplicial_klein_bottle` returns a triangulation of the two-dimensional Klein bottle.
- `simplicial_projective_plane` constructs a triangulation of the projective plane.
- `simplicial_torsion_space` returns a simplicial complex K with the integer homology groups $H_0(K) \cong \mathbb{Z}$, as well as $H_1(K) \cong \mathbb{Z}_n$ and $H_2(K) \cong 0$. In other words, this simplicial complex has nontrivial torsion in dimension 1. For $n = 3$ and $n = 4$ these triangulations are shown in the following figure. Notice that points with the same label have to be identified. This vertex naming scheme is the same as is used in the function.

These examples can be used to illustrate homology over different fields.

3.4 Cubical Complexes

The second important special case of a Lefschetz complex is called cubical complex, and it has been discussed in detail in [KMM04]. In the following, we only present the definitions that are essential for our purposes.

Loosely speaking, a cubical complex is a collection of cubes of varying dimensions in some Euclidean space $\mathbb{R}^d$. More precisely, we say that an interval $I \subset \mathbb{R}$ is an elementary interval if it is of the form

$$I = [\ell, \ell+1] \quad \text{or} \quad I = [\ell, \ell] \quad \text{for some integer} \quad \ell \in \mathbb{Z}\,.$$

If the elementary interval I consists of only one point, then it is called degenerate, and it is nondegenerate if it is of length one. Elementary intervals are the building blocks for the cubes in a cubical complex. For a complex in $\mathbb{R}^d$, an elementary cube Q is of the form

$$Q \;=\; I_1 \times I_2 \times \ldots \times I_d \;\subset\; \mathbb{R}^d\,,$$

where $I_1, \ldots, I_d$ are elementary intervals. The dimension $\dim Q$ of an elementary cube is given by the number of nondegenerate intervals in its representation. For example, the cube $Q = [0,0]\times[1,1]$ is a zero-dimensional elementary cube in $\mathbb{R}^2$ which contains only the point $(0,1)$, while the elementary cube $R = [2,2] \times [3,4]$ is one-dimensional, and consists of the closed vertical line segment between the points $(2,3)$ and $(2,4)$.

After these preparations, the definition of a cubical complex is now straightforward. A cubical complex X in $\mathbb{R}^d$ is a finite collection of elementary cubes in $\mathbb{R}^d$ which is closed under the inclusion of elementary subcubes. More precisely, if $Q \in X$ is an elementary cube in the cubical complex, and if $R \subset Q$ is any elementary cube contained in Q, then one also has $R \in X$.

The definition of a cubical complex is reminiscent of that of a simplicial complex. It is therefore not surprising that also in the cubical case one can describe the incidence coefficient map κ explicitly, and thus recognize a cubical complex as a Lefschetz complex. For this, we need more notation.

Let $Q = I_1 \times I_2 \times \ldots \times I_d$ denote an elementary cube, and let the nondegenerate elementary intervals in this decomposition be given by $I_{i_1}, \ldots, I_{i_n}$, where $I_{i_j} = [k_j, k_j+1]$ and $j = 1, \ldots, n = \dim Q$. For every index j, we further define the two $(n-1)$-dimensional elementary cubes

$$\begin{aligned} Q_j^- &= I_1 \times \ldots \times I_{i_j-1} \times [k_j, k_j] \times I_{i_j+1} \times \ldots \times I_d\,, \\ Q_j^+ &= I_1 \times \ldots \times I_{i_j-1} \times [k_j+1, k_j+1] \times I_{i_j+1} \times \ldots \times I_d\,. \end{aligned}$$

Geometrically, the two elementary cubes Q_j^- and Q_j^+ are directly opposite sides of the elementary cube Q. Using them, one can define the algebraic boundary of the cube as

$$\partial Q \;=\; \sum_{j=1}^{n} (-1)^{j-1} \left(Q_j^+ - Q_j^-\right).$$

This formula is the cubical analogue of the boundary operator in a simplicial complex, and it allows us to define the incidence coefficient map via

$$\kappa\left(Q,\, Q_j^+\right) \;=\; (-1)^{j-1} \quad \text{and} \quad \kappa\left(Q,\, Q_j^-\right) \;=\; (-1)^{j}\,, \quad \text{for all} \quad j = 1, \ldots, n\,.$$

For all remaining pairs of elementary cubes in X we let $\kappa = 0$. Then it was shown in [KMM04, Proposition 2.37] that the so-defined incidence coefficient map satisfies the summation condition in the definition of a Lefschetz complex, i.e., we have $\partial(\partial Q) = 0$ for every $Q \in X$. This in turn implies that every cubical complex is indeed a Lefschetz complex.

Cubical complexes in ConleyDynamics.jl are a little more restricted. Since a cubical complex in the above sense is always finite, one can assume without loss of generality that the left endpoints of all involved elementary intervals are nonnegative. In other words, we always assume that the cubical complex only contains elementary cubes from the set $(\mathbb{R}_0^+)^d$. This allows for a simple encoding of elementary cubes via labels of a fixed length, and without having to worry about the sign of an integer.

To describe this, fix a dimension d of the ambient space. Then every elementary cube in $(\mathbb{R}_0^+)^d$ has the following label, which depends on a coordinate width L:

- The first $d \cdot L$ characters of the label encode the starting points of the elementary intervals $I_1, \ldots, I_d$ in the standard representation of the elementary cube. For this, the starting points, which are nonnegative integers, are concatenated without spaces, but with leading zeros. For example, with $L = 2$ the string `"010203"` would correspond to the starting points 1, 2, and 3. Note that for given coordinate width L, one can only encode starting points between 0 and $10^L - 1$.
- The next entry in the label string is a period `.`.
- The remaining d characters of the string are integers 0 or 1, which give the interval lengths of $I_1, \ldots, I_d$.

For example, for $L = 2$ the string `"030600.000"` corresponds to the point $(3, 6, 0)$ in three dimensions. Similarly, the label `"030600.101"` represents the two-dimensional elementary cube $[3, 4] \times [6, 6] \times [0, 1] \subset \mathbb{R}^3$. Note, however, that the label representation is not unique, since it depends on the coordinate width L. Thus, with $L = 1$ the latter cube could also be written as `"360.101"`, or with $L = 3$ as `"003006000.101"`. As we will see in a moment, though, **within a given cubical complex all labels have to use the same coordinate width** L! This implies in particular that for a given coordinate width L one can only represent bounded cubical complexes which are contained in the d-dimensional box $[0, 10^L - 1]^d$.

The following three helper functions simplify the work with these types of cube labels:

- `cube_field_size` determines the field sizes of a given cube label. The first return value gives the dimension d of the ambient space, while the second value returns the coordinate width L.
- `cube_information` returns all information encoded in the cube label. The function returns an integer vector of length $2d + 1$, where d is the dimension of the ambient space. The first d entries give the vector of elementary interval starting points, while the next d values yield the corresponding interval lengths. The last entry specifies the dimension of the cube.
- `cube_label` creates a label from a cube's coordinate information. As function parameters, one has to specify d and L, and then pass an integer vector of length six which specifies the coordinates of the starting points and the interval lengths as in the previous item.

In ConleyDynamics.jl there are four basic commands for defining a cubical complex and working with it:

- `create_cubical_complex` creates a cubical complex in the Lefschetz complex data format. The complex is specified via a list of all the highest-dimensional cubes which are necessary to define the cubical complex. For this, every cube has to be given using the above-described special label format, with the same coordinate width L. In other words, all label strings have to be of the same length! If the optional parameter p is specified, the complex will be defined over a field with characteristic p, analogous to the case of a simplicial complex. If the characteristic is not specified, then the function defaults to the field $GF(2)$.
- `get_cubical_coords` determines the coordinates of all vertices of a given cubical complex from the cube labels. This vector can then be used for plotting purposes, see below.

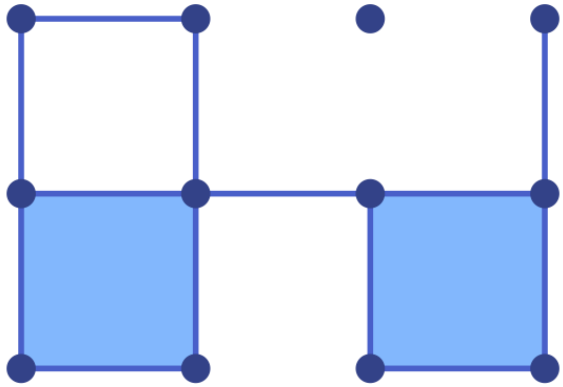


Figure 3.5: First sample cubical complex

- `create_cubical_rectangle` creates a cubical complex covering a rectangle in the plane. The rectangle is given by the subset $[0, nx] \times [0, ny]$ of the plane, where the nonnegative integers nx and ny have to be passed as arguments to the function. The function returns the cubical complex, and a vector of coordinates for the vertices. The latter can also be randomly perturbed as described in more detail in the function documentation.
- `create_cubical_box` creates a cubical complex covering a box in three-dimensional Euclidean space. The box is given by the subset $[0, nx] \times [0, ny] \times [0, nz]$ of space, where the nonnegative integers nx, ny, and nz have to be passed as arguments to the function. The optional parameters are the same as in the planar version.

To illustrate the first of these functions, consider the commands

```
cubes = ["00.11", "01.01", "02.10", "11.10", "11.01", "22.00", "20.11", "31.01"]
cc = create_cubical_complex(cubes)
```

These create the cubical complex cc, in the form of a Lefschetz complex. It can be visualized using the commands

```
coords = get_cubical_coords(cc)
fname  = "lefschetzex4.pdf"
plot_planar_cubical(cc,coords,fname,hfac=2.2,vfac=1.1,cubefac=60)
```

Similarly, the commands

```
cc2, coords2 = create_cubical_rectangle(5,2)
fname2 = "lefschetzex5.pdf"
plot_planar_cubical(cc2,coords2,fname2,hfac=1.7,vfac=1.2,cubefac=75)
```

define and illustrate a second cubical complex.

Finally, it is also possible to perturb the vertices in a cubical rectangle to obtain a Lefschetz complex consisting of quadrilaterals in the plane. This can be accomplied as follows:

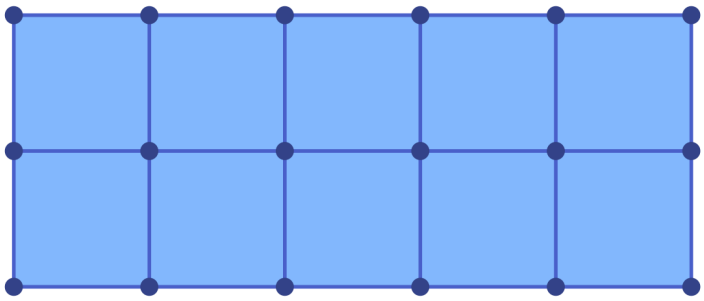


Figure 3.6: Second sample cubical complex

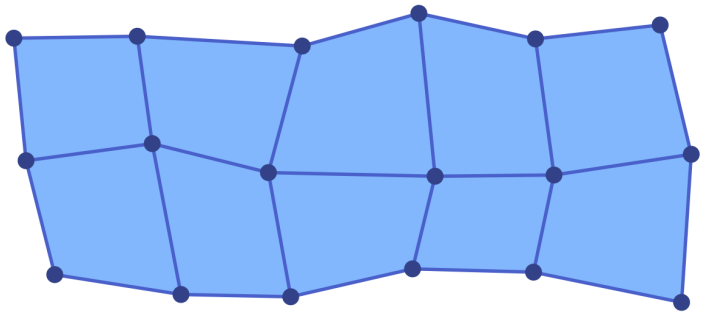


Figure 3.7: A randomly perturbed cubical complex

```
cc3, coords3 = create_cubical_rectangle(5,2,randomize=0.2)
fname3 = "lefschetzex6.pdf"
plot_planar_cubical(cc3,coords3,fname3,hfac=1.7,vfac=1.2,cubefac=75)
```

The resulting Lefschetz complex is visualized in the last figure of this section.

3.5 Lefschetz Complex Operations

Once a Lefschetz complex has been created, there are a number of manipulations and queries that one has to be able to perform on the complex. At the moment, ConleyDynamics.jl supplies a number of functions for this. The following four functions provide basic information:

- `lefschetz_information` provides basic information about the Lefschetz complex, including its Euler characteristic and homology, as well as cell counts by dimension and sparsity percentage of the boundary matrix.
- `lefschetz_cell_count` returns the number of cells in each dimension in a vector of length `lc.dim + 1`. If the optional parameter `bounds=true` is passed, then the function also returns two integer vectors `lo` and `hi`. These contain the beginning and end indices of the cells in each dimension.
- `lefschetz_field` returns the field F over which the Lefschetz complex is defined as a `String`.
- `lefschetz_is_closed` determines whether a given Lefschetz complex cell subset is closed or not.

- `lefschetz_is_locally_closed` checks whether a given Lefschetz complex cell subset is locally closed or not.

The next set of functions can be used to extract certain topological features from a Lefschetz complex:

- `lefschetz_boundary` computes the support of the boundary $\partial\sigma$ of a Lefschetz complex cell σ. In other words, it returns the vector of all facets of σ. The cell can either be specified via its index or its label, and the return format corresponds to the input format.
- `lefschetz_coboundary` returns all cells which lie in the coboundary of the specified cell σ, i.e., it returns all cells which have σ as a facet.
- `lefschetz_closure` determines the closure of a given cell subset, i.e., the union of all faces of cells in the cell subset.
- `lefschetz_interior` determines the interior of a given cell subset. For this, the Lefschetz complex is interpreted as a finite topological space, where a set A is open if and only if for every cell $\sigma \in A$ all of its cofaces are also contained in the set A.
- `lefschetz_openhull` computes the open hull of a cell subset, i.e., the smallest open set which contains the given cell subset. For this, we again interpret a Lefschetz complex as a finite topological space as above.
- `lefschetz_topboundary` computes the topological boundary of a cell subset, i.e., the set difference of the closure and the interior of the set. Note that this usually differs from the (algebraic) boundary mentioned above.
- `lefschetz_lchull` finds the locally closed hull of a Lefschetz complex subset. This is the smallest locally closed set which contains the given cell subset. One can show that it is the intersection of the closure and the open hull of the cell subset.
- `lefschetz_clomo_pair` determines the closure-mouth-pair associated with a Lefschetz complex subset.
- `lefschetz_skeleton` computes the k-dimensional skeleton of a Lefschetz complex or of a given Lefschetz complex subset. While in the first case the k-skeleton of the full Lefschetz complex is returned, in the second case it returns the k-skeleton of the closure of the given subset.
- `manifold_boundary` returns a list of cells which form the "manifold boundary" of the given Lefschetz complex. More precisely, if the complex has dimension d, then it determines all cells of dimension $d-1$ which have at most one cell in their coboundary, as well as all cells of dimensions less than $d-1$ which have no cell in their coboundary, and finally returns the closure of the resulting cell subset.

The following functions create Lefschetz subcomplexes from a Lefschetz complex:

- `lefschetz_subcomplex` determines a Lefschetz subcomplex from a given Lefschetz complex. The subcomplex has to be locally closed, and it is given by a collection of cells.
- `lefschetz_closed_subcomplex` extracts a closed Lefschetz subcomplex from the given Lefschetz complex. The subcomplex is the closure of the specified collection of cells.

In addition, it is possible to create a new Lefschetz complex from an existing one using the following functions:

- `permute_lefschetz_complex` determines a new Lefschetz complex which is obtained from the original one by a permutation of the cells. Note that the permutation has to respect the ordering of the cells by dimension, otherwise an error is raised. In other words, the permutation has to decompose into permutations within each dimension. This is automatically done if no permutation is explicitly specified and the function creates a random one.

- `lefschetz_reduction` uses elementary reductions to transform a Lefschetz complex into a smaller one while preserving homology. In fact, the new complex is chain homotopic to the original one. For this, one has to specify a sequence of reduction pairs, which are disjoint pairs of cells whose dimensions differ by one, and such that one cell is a face of the other, once the earlier reductions have been performed. This function is based on [KMS98]. It returns a new Lefschetz complex which no longer contains the cells contained in the reduction pairs.

- `lefschetz_reduction_maps` is an extension of the previous function `lefschetz_reduction`. While it performs the same basic Lefschetz complex reduction, it does compute useful additional information. Besides the reduced Lefschetz complex, it also returns the chain equivalences between the original and the reduced complex, as well as the chain homotopy relating one of their compositions to the identity. The return values of this function are as follows:

  - `lcred`: The first return variable contains the reduced Lefschetz complex, just as in the previous function.
  - `pp`: This is a sparse matrix representation of the chain equivalence between the original complex and the reduced one.
  - `jj`: This sparse matrix gives the chain equivalence between the reduced complex and the original one.
  - `hh`: The third sparse matrix represents the chain homotopy which shows that the composition `jj * pp` is chain homotopic to the identity.

  Note that these maps are constructed in such a way that the product `pp * jj` always is the identity.

- `lefschetz_newbasis` creates a new Lefschetz complex `lcnew` from the given Lefschetz complex `lc` via a change of basis. The new basis has to be specified in the sparse matrix `basis`, whose columns represent the new basis in terms of the existing one. The basis matrix has to respect the grading by dimension, i.e., the cells which are used to form a new basis chain have to have the same dimensions as the cell which is being replaced.

- `lefschetz_newbasis_maps` extends the previous function in that besides the new Lefschetz complex `lcnew` it also returns the chain maps `pp` and `jj` which are the respective isomorphisms from the original complex `lc` to `lcnew`, and vice versa, as well as the zero chain homotopy `hh`.

- `compose_reductions` can be used to find the composed chain maps and chain homotopy for a pair of subsequent reductions. These can be produced either via the function `lefschetz_reduction_maps` or the function `lefschetz_newbasis_maps`.

It was shown in [EM23] that suitable reduction pairs for `lefschetz_reduction` can be found easily via the concept of filters. A filter on a Lefschetz complex is a function $\varphi : X \to \mathbb{R}$ which has the property that $\varphi(x) \leq \varphi(y)$ if x is a face of y. Every such filter induces shallow pairs, which in the case of an injective filter generate a gradient Forman vector field on the Lefschetz complex. It turns out that these shallow pairs can be used to reduce the complex. In ConleyDynamics.jl there are three functions for working with filters:

- `create_random_filter` creates a random injective filter on a Lefschetz complex. The filter is created by assigning integers to cell groups, increasing with dimension. Within each dimension the assignment is

random, but all filter values of cells of dimension k are less than all filter values of cells with dimension $k+1$. The function returns the filter as `Vector{Int}`, with indices corresponding to the cell indices in the Lefschetz complex.

- `filter_shallow_pairs` finds all shallow pairs for the filter φ. These are face-coface pairs (x, y) whose dimensions differ by one, and such that y has the smallest filter value on the coboundary of x, and x has the largest filter value on the boundary of y. For injective filters, these pairs give rise to a Forman vector field on the underlying Lefschetz complex. For noninjective filters this is not true in general.
- `filter_induced_mvf` returns the smallest multivector field which has the property that every shallow pair is contained in a multivector. For injective filters this is a Forman vector field, but in the noninjective case it can be a general multivector field.

Notice that a special class of filters, called filtrations, are used in the context of persistent homology. There are two functions devoted to dealing with them:

- `lefschetz_filtration` computes a filtration on a Lefschetz subset. Based on integer filtration values assigned to some cells of the given Lefschetz complex, it determines the smallest closed subcomplex `lcsub` which contains all cells with nonzero filtration values, as well as filtration values `fvalsub` on this subcomplex, which give rise to a filtration of closed subcomplexes, and which can be used to compute persistent homology.
- `lefschetz_filtration_mvf` determines the multivector field associated with a Lefschetz complex filtration created by the previous function. For the Lefschetz complex `lc`, and the filtration `fvalues` given on the Lefschetz complex, this multivector field is generated by the filtration in the following way. For every filtration value k, the cells which are assigned this value form a locally closed set, which can then be decomposed into connected components that are all locally closed as well. The union of all these connected components, as k ranges from 1 to N, forms the multivector field `mvf` that is returned by the function.

There is also one helper function which can sometimes be useful:

- `lefschetz_gfp_conversion` changes the base field of the given Lefschetz complex from the rationals $\mathbb{Q}$ to a finite field $GF(p)$. Note that it is not possible to perform the reverse conversion.

In addition, ConleyDynamics.jl provides the following helper functions for the fundamental objects of cells and cell subsets, which can be represented either by integer cell indices or by cell labels:

- `convert_cells` converts a vector of cells from integer to label format, or vice versa.
- `convert_cellsubsets` converts a vector of cell subsets from integer to label format, and vice versa.
- `cellsubset_bounding_box` computes the bounding box for a cell subset, provided coordinates for the vertices of the Lefschetz complex are provided. The bounding box is purely based on the location of the vertices.
- `cellsubset_distance` computes the distance of a cell subset from a point. More precisely, this function computes the minimal distance of a vertex in the closure of the cell subset to the given point. While this function does not provide the Hausdorff distance, it serves as a simple measure for how far the cell subset is from a give point.

- `cellsubset_planar_area` computes the area of a planar cell subset. This function assumes that the complex is two-dimensional and that the maximal 2-cells in the cell subset are all polygonal with straight boundary edges. If these conditions are not met an error is raised.
- `locate_planar_cellsubsets` expects a list of cell subsets, for example the collection of Morse sets, or a multivector field, and it extracts the subsets whose closure lies in the interior of a geometric object G, and also the ones whose closure intersects the boundary of G. At the moment, G can be either a planar rectangle with sides parallel to the coordinate axes, or a circle in the plane. The same command is used for both, the version of G is selected through the function arguments via multiple dispatch.
- `cellsubset_location_rectangle` determines the location of the closure of a given cell subset relative to a rectangle in the plane. It distinguishes between being in the interior, intersecting the boundary, or lying in the exterior. The rectangle has to be parallel to the coordinate axes.
- `cellsubset_location_circle` determines the location of the closure of a given cell subset relative to a circle in the plane. It distinguishes between being in the interior, intersecting the boundary, or lying in the exterior.

Finally, there are a couple of geometry helper functions which allow for the transformation of vertex coordinates in a Lefschetz complex:

- `convert_planar_coordinates` transforms a given collection of planar coordinates in such a way that the extreme coordinates fit precisely in a given rectangle in the plane.
- `convert_spatial_coordinates` transforms a given collection of spatial coordinates in such a way that the extreme coordinates fit precisely in a given rectangular box in space.
- `signed_distance_rectangle` determines the signed distance from a given point to the boundary of a rectangle in the plane. Negative values correspond to the interior, positive values to the exterior.
- `signed_distance_circle` determines the signed distance from a given point to the boundary of a circle in the plane. Negative values correspond to the interior, positive values to the exterior.
- `segment_intersects_rectangle` determines whether a line segment intersects the boundary of an axes-parallel rectangle in the plane.
- `segment_intersects_circle` determines whether a line segment intersects a circle in the plane.

For more details on the usage of any of these functions, please see their documentation in the API section of the manual.

3.6 References

See the full bibliography for a complete list of references cited throughout this documentation. This section cites the following references:

[Ale37] P. Alexandrov. Diskrete Räume. Mathematiceskii Sbornik (N.S.) **2**, 501–518 (1937).

[DKMW11] P. Dłotko, T. Kaczynski, M. Mrozek and T. Wanner. Coreduction homology algorithm for regular CW-complexes. Discrete & Computational Geometry **46**, 361–388 (2011).

[EM23] H. Edelsbrunner and M. Mrozek. The depth poset of a filtered Lefschetz complex (2023), arXiv:2311.14364v2 [math.AT].

[KMM04] T. Kaczynski, K. Mischaikow and M. Mrozek. Computational Homology. Vol. 157 of Applied Mathematical Sciences (Springer-Verlag, New York, 2004).

[KMS98] T. Kaczynski, M. Mrozek and M. Slusarek. Homology computation by reduction of chain complexes. Computers & Mathematics with Applications **35**, 59–70 (1998).

[Lef42] S. Lefschetz. Algebraic Topology. Vol. 27 of American Mathematical Society Colloquium Publications (American Mathematical Society, New York, 1942).

[LKMW23] M. Lipinski, J. Kubica, M. Mrozek and T. Wanner. Conley-Morse-Forman theory for generalized combinatorial multivector fields on finite topological spaces. Journal of Applied and Computational Topology **7**, 139–184 (2023).

[Mas91] W. S. Massey. A Basic Course in Algebraic Topology. Vol. 127 of Graduate Texts in Mathematics (Springer-Verlag, New York, 1991).

[MB09] M. Mrozek and B. Batko. Coreduction homology algorithm. Discrete & Computational Geometry **41**, 96–118 (2009).

[MW25] M. Mrozek and T. Wanner. Connection Matrices in Combinatorial Topological Dynamics. SpringerBriefs in Mathematics (Springer-Verlag, Cham, 2025).

[Mun84] J. R. Munkres. Elements of Algebraic Topology. SpringerBriefs in Mathematics (Addison-Wesley, Menlo Park, 1984).

Chapter 4

Homology

Conley's theory for the qualitative study of dynamical systems is based on fundamental concepts from algebraic topology. One of these is homology, which studies the topological properties of spaces using algebraic means. As part of ConleyDynamics.jl a number of homology methods are included. It should be noted that these algorithms are not meant for truly large-scale problems, but mostly for illustrative purposes. They are based on the persistence algorithm described in [EH10], and have been extended to work for arbitrary Lefschetz complexes over either the rationals or a finite field of prime order. For more serious applications one could use professional implementations such as Gudhi, see [GUD24].

4.1 Lefschetz Complex Homology

The most important notion of homology used in ConleyDynamics.jl is Lefschetz homology. It generalizes both simplicial homology as described in [Mun84], and cubical homology in the sense of [KMM04]. In order to fix our notation, we provide a brief introduction in the following. For more details, see [Lef42].

As we saw earlier, a Lefschetz complex X is a collection of cells with associated nonnegative dimensions, together with a boundary map ∂ which is induced by the incidence coefficient map κ. The fundamental idea behind homology is to turn this underlying information into an algebraic form in such a way that the boundary map becomes a linear map. For this, define the k-th chain group as

$$C_k(X) = \left\{ \sum_{i=1}^{m} \alpha_i \sigma_i \ : \ \alpha_1, \dots, \alpha_m \in F \quad \text{and} \quad \sigma_1, \dots, \sigma_m \in X_k \right\}.$$

Since X_k denotes the collection of all cells of dimension k, this definition can be rephrased by saying that $C_k(X)$ consists of all formal linear combinations of k-dimensional cells with coefficients in the underlying field F. It is not difficult to see that $C_k(X)$ is in fact a vector space over F. Moreover, its dimension is equal to the number of k-dimensional cells in X. The collection of all chain groups is $C(X) = (C_k(X))_{k \in \mathbb{Z}}$, where we let $C_k(X) = \{0\}$ for all $k < 0$ and $k > \dim X$.

We now turn our attention to the boundary map. It was already explained how the incidence coefficient map κ can be used to define the boundary $\partial\sigma \in C_{k-1}(X)$ for every k-dimensional cell $\sigma \in X_k$. If one further defines

$$\partial \left(\sum_{i=1}^{m} \alpha_i \sigma_i \right) = \sum_{i=1}^{m} \alpha_i \partial\sigma_i \in C_{k-1}(X) \quad \text{for} \quad \sum_{i=1}^{m} \alpha_i \sigma_i \in C_k(X) \ ,$$

then one obtains a map $\partial : C_k(X) \to C_{k-1}(X)$. It is not difficult to verify that this map is both well-defined and linear. Sometimes, we write ∂_k instead of ∂ to emphasize that we consider the boundary map defined on the k-th chain group $C_k(X)$.

Altogether, the above definitions have equipped us with a sequence of vector spaces and maps between them in the form

$$\ldots \xrightarrow{\partial_{k+2}} C_{k+1}(X) \xrightarrow{\partial_{k+1}} C_k(X) \xrightarrow{\partial_k} C_{k-1}(X) \xrightarrow{\partial_{k-1}} \ldots \xrightarrow{\partial_1} C_0(X) \xrightarrow{\partial_0} \{0\} \xrightarrow{\partial_{-1}} \ldots,$$

and the properties of a Lefschetz complex further imply that

$$\partial_k \circ \partial_{k+1} = 0 \quad \text{for all} \quad k \in \mathbb{Z}.$$

In other words, the pair $(C(X), \partial)$ is a chain complex, which consists of a sequence of vector spaces over F and linear maps between them. Recall from linear algebra that any linear map induces two important subspaces, which in the context of algebraic topology are given special names as follows:

- The elements of the subspace $Z_k(X) = \ker \partial_k$ are called the k-cycles of X.
- The elements of the subspace $B_k(X) = \text{im } \partial_{k+1}$ are called the k-boundaries of X.

Both of these vector spaces are subspaces of the k-th chain group $C_k(X)$. Furthermore, in view of the above identity $\partial_k \circ \partial_{k+1} = 0$, one immediately obtains the subspace inclusion $B_k(X) \subset Z_k(X)$. We can therefore define the quotient space

$$H_k(X) \;=\; Z_k(X)/B_k(X) \;=\; \ker \partial_k / \text{im } \partial_{k+1}.$$

This vector space is called the k-th homology group of the Lefschetz complex X. It is again a vector space over F, and therefore its dimension provides important information. In view of this, the dimension of the k-th homology group $H_k(X)$ is called the k-th Betti number of X, and abbreviated as $\beta_k(X) = \dim H_k(X)$.

In order to shed some light on the actual meaning of homology, and in particular the Betti numbers, we turn to an example. Consider the simplicial complex `sc` that was already introduced in the Tutorial, and which can be created using the commands

```
labels = ["A","B","C","D","E","F"]
simplices = [["A","B"],["A","C"],["B","C"],["B","D"],["D","E","F"]]
sc = create_simplicial_complex(labels,simplices)
```

This two-dimensional simplicial complex is shown in the figure.

For simplicity, we consider the associated Lefschetz complex X over the field $F = GF(2)$. Then chains in a chain group are just a sum of individual cells of the same dimension.

In this simple example, one can determine the cycles and boundaries for $k = 1$ directly. The vector space $Z_1(X)$ of 1-cycles contains the two nonzero chains $c_1 = \text{AB} + \text{BC} + \text{AC}$ and $c_2 = \text{DE} + \text{EF} + \text{DF}$, since one can verify that $\partial c_1 = \partial c_2 = 0$. These are, however, not all nontrivial 1-cycles, as their sum $c_1 + c_2$ is another

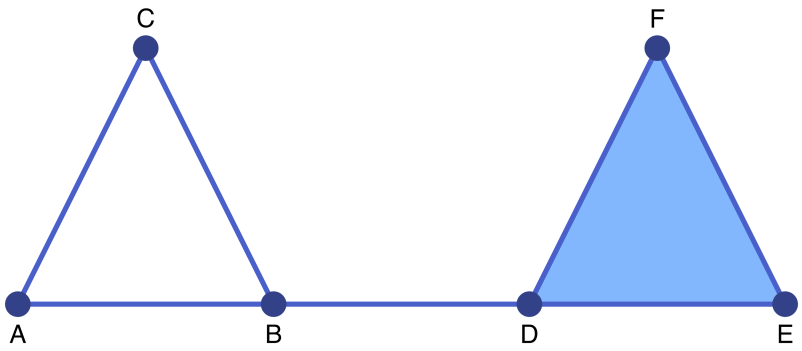


Figure 4.1: The simplicial complex from the tutorial

one. Thus, the first cycle group is given by $Z_1(X) = \{0, c_1, c_2, c_1 + c_2\}$. It is a vector space over $F = GF(2)$ of dimension two, and any two nonzero elements of $Z_1(X)$ form a basis. What about the 1-boundaries? The simplicial complex X contains only one 2-cell, namely DEF, and its boundary is given by the chain c_2. Thus, the first boundary group is given by $B_1(X) = \{0, c_2\}$, which is a one-dimensional vector space over F.

Combined, one can show that the first homology group $H_1(X)$ consists of the two equivalence classes

$$H_1(X) = \{B_1(X),\ c_1 + B_1(X)\}\ ,$$

where the class $B_1(X)$ is the zero element in $H_1(X)$. This implies that the first homology group is one-dimensional, and we have $\beta_1(X) = 1$. In some sense, the basis element of $H_1(X)$, which is the unique nonzero equivalence class given by $c_1 + B_1(X)$, is represented by the cycle c_1.

The above mathematically precise description can be summarized as follows. All three nontrivial cycles in $Z_1(X)$ have the potential to enclose two-dimensional holes in the simplicial complex X, since they are chains without boundary. However, some of these potential holes have been filled in by two-dimensional cells. Thus, while c_1 does indeed represent a hole, the chain c_2 does not, since its interior is filled in by DEF. Note that the cycle $c_1 + c_2$ does not create a second hole, since we have $(c_1 + c_2) - c_1 = c_2 \in B_1(X)$. In other words, the first Betti number counts the number of independent holes in the complex X.

One can extend this discussion also to other dimensions and to general Lefschetz complexes X. In this way, one obtains the following informal interpretations of the Betti numbers:

- $\beta_0(X)$ counts the number of connected components of X,
- $\beta_1(X)$ counts the number of independent holes in X,
- $\beta_2(X)$ counts the number of independent cavities in X.

In general, one can show that $\beta_k(X)$ represents the number of independent k-dimensional holes in the Lefschetz complex X. For more details, see [Mun84].

The package ConleyDynamics.jl provides one function to compute standard homology:

- `homology` expects one input argument, which has to be of the Lefschetz complex type `LefschetzComplex`. It returns a vector `betti` of integers, whose length is one more than the dimension of the complex. The k-th Betti number $\beta_k(X)$ is returned in `betti[k+1]`.

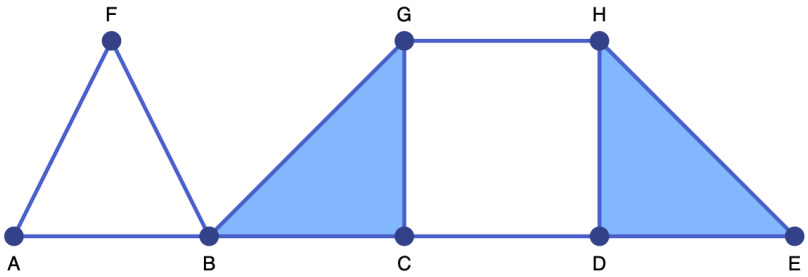


Figure 4.2: Sample simplicial complex

We would like to point out that the field F is implicit in the data structure for the Lefschetz complex X, and therefore it does not have to be specified. It can always be queried using the function `lefschetz_field`. For the above example one obtains

```
julia> homology(sc)
3-element Vector{Int64}:
 1
 1
 0
```

This clearly gives the correct Betti numbers, as we have already seen that this simplicial complex has one hole, and it is obviously connected.

The simplicial complex shown in the second figure can be created using the commands

```
labels2 = ["A","B","C","D","E","F","G","H"]
simplices2 = [["A","B"],["A","F"],["B","F"],["B","C","G"],["D","E","H"],["C","D"],["G","H"]]
sc2 = create_simplicial_complex(labels2,simplices2)
```

and its homology can then be determined as follows:

```
julia> homology(sc2)
3-element Vector{Int64}:
 1
 2
 0
```

This complex is also connected, and therefore one has $\beta_0(X) = 1$. However, this time one obtains two independent holes, which results in $\beta_1(X) = 2$.

Similarly, the cubical complex depicted in the next figure can be generated via

```
cubes = ["00.11", "01.01", "02.10", "11.10", "11.01", "22.00", "20.11", "31.01"]
cc = create_cubical_complex(cubes)
```

and its Betti numbers are given by

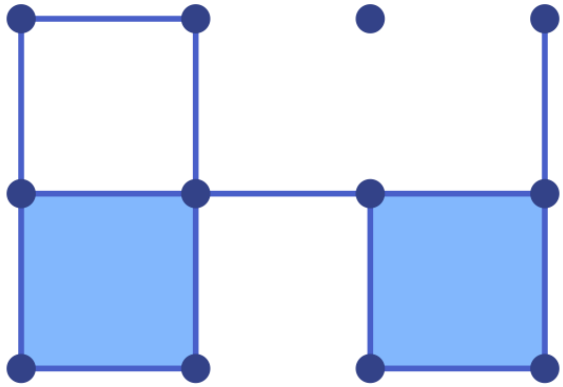


Figure 4.3: Sample cubical complex

```
julia> homology(cc)
3-element Vector{Int64}:
 2
 1
 0
```

In this case, the complex has two components and one hole. As a final example, consider a simplicial complex which consists of the manifold boundary of a single cube. Such a complex can be generated using the commands

```
cc2,~ = create_cubical_box(1, 1, 1)
mbcells = manifold_boundary(cc2)
cc2bnd = lefschetz_subcomplex(cc2, mbcells)
```

This time, the homology of the resulting cubical complex is given by the Betti numbers

```
julia> homology(cc2bnd)
3-element Vector{Int64}:
 1
 0
 1
```

This complex is connected and has no holes, but it does have one cavity. As shown, these observations translate into the Betti numbers $\beta_0(X) = 1$ and $\beta_1(X) = 0$, as well as $\beta_2(X) = 1$.

Beyond these simple illustrative examples, homology can be a useful tool in a variety of applied settings. For example, it can be used to quantify the evolution of material microstructures during phase separation processes, see for example [GMW05].

4.2 Relative Homology

For the definition of the Conley index of an isolated invariant set another notion of homology is essential, namely relative homology. For this, we assume that X is a Lefschetz complex, and $Y \subset X$ is a closed subset. In other words, for every cell in Y, all of its faces are contained in Y as well. Then relative homology defines

a sequence of groups $H_k(X, Y)$ for $k \in \mathbb{Z}$ which basically measures the topological properties of X if the subset Y is contracted to a point and then forgotten.

This admittedly very vague definition can be made precise in a number of ways. Two of these can easily be described:

- We have already seen that any locally closed subset of a Lefschetz complex is again a Lefschetz complex. Since the subset $Y \subset X$ is closed, its complement $X \setminus Y$ is open, and hence locally closed as well. Thus, the complement $X \setminus Y$ is again a Lefschetz complex. It has been shown in [MB09, Theorem 3.5] that then

$$H_k(X, Y) \cong H_k(X \setminus Y) \quad \text{for all} \quad k \in \mathbb{Z}.$$

  In other words, the relative homology of the pair (X, Y) is just the regular homology of the Lefschetz complex given by the set $X \setminus Y$, with the incidence coefficient map inherited from X.

- On a more topological level, one can also think of the relative homology of (X, Y) in the following way. In the complex X, identify all cells in Y to a single point, in the sense of the quotient space X/Y defined in a standard topology course. Then one can show that

$$H_k(X, Y) \cong \tilde{H}_k(X/Y) \quad \text{for all} \quad k \in \mathbb{Z},$$

  where $\tilde{H}_k(Z)$ denotes the reduced homology of a space Z. While the details of this latter notion of homology can be found in [Mun84, Section 7], for our purposes it suffices to note that the Betti numbers in reduced homology can be obtained from the ones in regular homology by decreasing the 0-th Betti number by 1, and keeping all other Betti numbers unchanged.

The precise mathematical definition of relative homology can be found in [Mun84, Section 9], and it is briefly introduced in the following. Since the k-th chain group of a Lefschetz complex consists of all formal linear combinations of k-dimensional cells, one can consider the vector space $C_k(Y)$ as a subspace of $C_k(X)$. Thus, it makes sense to form the quotient groups

$$C_k(X, Y) = C_k(X)/C_k(Y)$$

as in linear algebra. Moreover, if one considers a class $[x] \in C_k(X, Y)$ represented by some $x \in C_k(X)$, then the definition

$$\partial[x] = [\partial x] \in C_{k-1}(X, Y) \quad \text{for} \quad [x] \in C_k(X, Y)$$

gives a well-defined linear map $\partial : C_k(X, Y) \to C_{k-1}(X, Y)$ which satisfies $\partial \circ \partial = 0$. In other words, the collection $(C_k(X, Y))_{k \in \mathbb{Z}}$ equipped with this boundary operator ∂ is a chain complex, and its associated homology groups $H_k(X, Y)$ are called the relative homology groups of the pair (X, Y). Notice that by forming the quotient spaces $C_k(X)/C_k(Y)$, the chains in the subspace are all identified and set to zero, as mentioned earlier.

In ConleyDynamics.jl, relative homology can be computed using `relative_homology`. There are two possible ways to invoke this function:

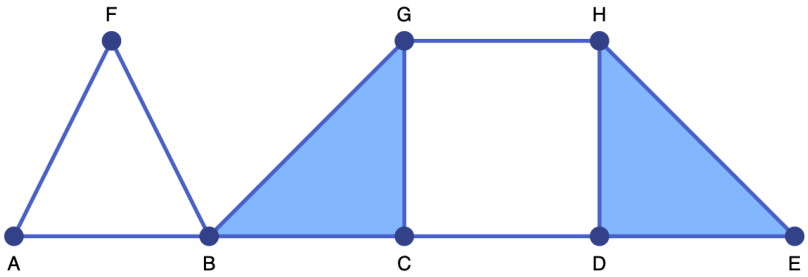


Figure 4.4: Sample simplicial complex

- The method `relative_homology(lc::LefschetzComplex, subc::Cells)` expects a Lefschetz complex `lc` which represents X, together with a list of cells `subc`. The closure of this cell list determines the closed subcomplex Y.
- The method `relative_homology(lc::LefschetzComplex, subc::Cells, subc0::Cells)` expects an ambient Lefschetz complex specified by the argument `lc`. The Lefschetz complex X is then the closure of the cell list `subc`, while the subcomplex Y is given by the closure of the cell list `subc0`. These closures are automatically computed by the function.

Both versions of `relative_homology` return the relative homology as a vector `betti` of Betti numbers, where `betti[k]` is the Betti number in dimension `k-1`. Notice also that the necessary cell list arguments have to be variables of the type `Cells = Union{Vector{Int},Vector{String}}`, i.e., they can be given in either label or index form.

In order to briefly illustrate the different usages of the command `relative_homology`, we consider again the simplicial complex shown in the figure, which can be generated using the commands

```
labels2 = ["A","B","C","D","E","F","G","H"]
simplices2 = [["A","B"],["A","F"],["B","F"],["B","C","G"],["D","E","H"],["C","D"],["G","H"]]
sc2 = create_simplicial_complex(labels2,simplices2)
```

If we identify the vertices A and E, then an additional loop is created along the bottom of the original simplicial complex. This leads to the following relative homology:

```
julia> relative_homology(sc2, ["A","E"])
3-element Vector{Int64}:
 0
 3
 0
```

Note that the 0-th Betti number becomes zero, since these identified vertices are considered as zero in the chain group $C_0(X,Y)$. On the other hand, if we consider the boundary of the triangle DEH as the subcomplex Y, then one obtains:

```
julia> relative_homology(sc2, ["DE","DH","EH"])
3-element Vector{Int64}:
 0
 2
 1
```

Again, the 0-th Betti number is reduced by one. But this time, the first Betti number does not change, as no new holes are created. Nevertheless, collapsing the boundary of the triangle to a point does create a cavity, and therefore the 2-nd Betti number is now one. One can also just consider the closure of the triangle BCG as a Lefschetz complex X, and use its boundary as subcomplex Y. In this case we get:

```
julia> relative_homology(sc2, ["BCG"], ["BC","BG","CG"])
3-element Vector{Int64}:
 0
 0
 1
```

This is the reduced homology of a two-dimensional sphere, which is the topological space obtained from the quotient space X/Y. As our final example, consider the closed edge AB as Lefschetz complex X, and the vertex B as subcomplex Y, then the relative homology of the pair (X, Y) is given by

```
julia> relative_homology(sc2, ["AB"], ["B"])
3-element Vector{Int64}:
 0
 0
 0
```

In this case, all Betti numbers are zero. This can also be seen by recalling that this relative homology is isomorphic to the relative homology of the two-element Lefschetz complex which consists only of the edge AB and the vertex A:

```
julia> homology(lefschetz_subcomplex(sc2, ["A","AB"]))
2-element Vector{Int64}:
 0
 0
```

Note that one obtains a Betti number vector of length two, since this subcomplex has dimension one.

4.3 Persistent Homology

Even though the notion of persistence is not strictly necessary for the study of combinatorial topological dynamics, the package ConleyDynamics.jl provides rudimentary support for the computation of persistence intervals for filtrations of Lefschetz complexes. A detailed introduction to persistence can be found in the book [EH10], and we briefly provide an intuitive definition and some examples below.

Persistent homology is concerned with the creation and destruction of topological features in a sequence of nested Lefschetz complexes. More precisely, consider the sequence of Lefschetz complexes

$$X^{(1)} \subset X^{(2)} \subset \ldots \subset X^{(n)},$$

where we assume that $X^{(k)}$ is closed in $X^{(n)}$ for all $1 \leq k < n$. This is called a filtration of Lefschetz complexes. As we have seen above, every one of the complexes $X^{(k)}$ has associated homology groups, and the notion of persistence is concerned with how these groups change as k is increased from 1 to n. More precisely, this is based on the following intuition:

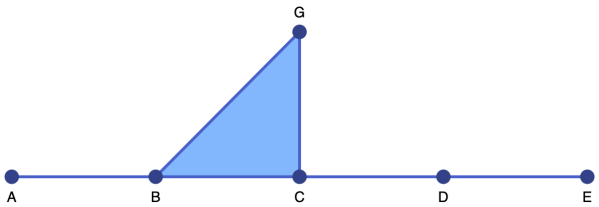


Figure 4.5: The 1-st complex in the filtration

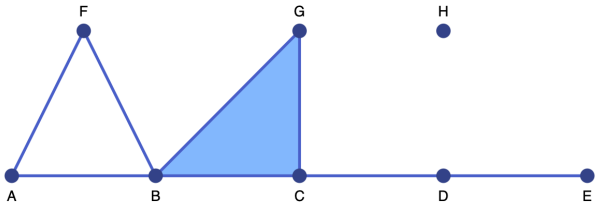


Figure 4.6: The 2-nd complex in the filtration

- For $k = 1$, the Betti numbers of the Lefschetz complex $X^{(1)}$ describe how many nontrivial holes the complex has in each dimension. Each of these holes is represented by a cycle which generates the associated homology class. One then says that each of these homology classes is born at $k = 1$.
- As one passes from the complex $X^{(k)}$ to the complex $X^{(k+1)}$, for $k = 1, \ldots, n-1$, these Betti numbers can change in a number of ways:
  - A new homology class is created in $X^{(k+1)}$, which leads to an increase in the corresponding Betti number. As before, this means that a new homology class is born at level $k + 1$.
  - A homology class that was present in $X^{(k)}$ is no longer present in $X^{(k+1)}$. On the one hand, this could be the result of the merging of two separate topological features, such as for example two separate connected components of the complex $X^{(k)}$ which become one connected component in $X^{(k+1)}$. On the other hand, the corresponding hole in the complex could have been filled in through the introduction of cells in the set difference $X^{(k+1)} \setminus X^{(k)}$. In either case, we say that the homology class died at level $k + 1$.
- The persistent homology associated with this filtration then consists of a collection of persistence intervals for each dimension. All of these intervals are of the form $[b, d)$, where b denotes the birth time and d the death time of a topological feature. Note that some homology classes might still be present in the homology of the final Lefschetz complex $X^{(n)}$, and in this case one obtains an interval of the form $[b, \infty)$, i.e., the feature never dies.

With the above intuitive description one usually can work out the collection of persistence intervals for small and simple examples. There is, however, one final ambiguity that has to be resolved. Suppose that two topological features are born at times b_1 and b_2, and they merge to a single feature at time d. Which of these survives into the next complex, and which dies? In this situation, the elder rule applies, which says that the older feature persists. Thus, if $b_1 \geq b_2$, then one obtains the persistence interval $[b_1, d)$, while the death time e in the interval $[b_2, e)$ will be determined by a later level, i.e., we have $e > d$.

In order to illustrate this informal definition, we consider the filtration given by the four simplicial complexes shown in the figures. All of these are subcomplexes of, and the last one is equal to, one of our earlier examples.

In this simple example, the persistence intervals in each dimension can be determined easily:

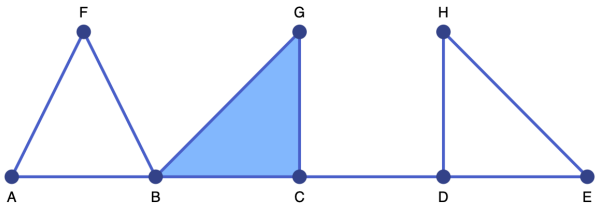


Figure 4.7: The 3-rd complex in the filtration

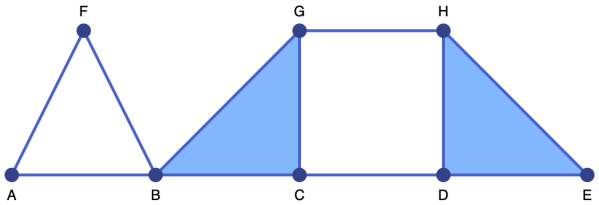


Figure 4.8: The 4-th complex in the filtration

- **Dimension 0**: The first complex has one connected component. A second component, namely the vertex H, is added in $X^{(2)}$. Both of these components merge in $X^{(3)}$, and no additional components are created. In view of the elder rule, this gives the two persistence intervals $[1, \infty)$ and $[2, 3)$.
- **Dimension 1**: The first hole is created in $X^{(2)}$, given by the cycle $\mathrm{AB} + \mathrm{AF} + \mathrm{BF}$. Moreover, in $X^{(3)}$ the hole determined by the chain $\mathrm{DE} + \mathrm{DH} + \mathrm{EH}$ is added to the mix. Finally, in $X^{(4)}$ the latter hole is removed through filling in the triangle DEH, while the hole bounded by the cycle $\mathrm{CD} + \mathrm{DH} + \mathrm{CG} + \mathrm{GH}$ is created. This gives the unbounded persistence intervals $[2, \infty)$ and $[4, \infty)$, as well as the bounded one $[3, 4)$.
- **Dimension 2**: None of the complexes form any cavities, and therefore there are no persistence intervals in dimension two.

In ConleyDynamics.jl, there are two functions that provide basic persistence functionality:

- The function `persistent_homology` computes the persistence intervals, and it is usually invoked using the command `pinf, ppairs = persistent_homology(lc, filtration)`. It expects two arguments: The first is an underlying Lefschetz complex `lc` of type `LefschetzComplex`, which has to be the complex $X^{(n)}$ in the above notation. The second argument `filtration` is of type `Vector{Int}` and has to have length `lc.ncells`. For each cell index, it contains the integer level k of the the first complex $X^{(k)}$ in which the cell appears. The function returns two vectors, `pinf` and `ppairs`, each of which have length `1 + lc.dim`, and which contain the following information:
  - `pinf[k]` contains a vector of birth times for all unbounded persistence intervals in dimension $k-1$. It is an empty vector if no such intervals exist.
  - `ppairs[k]` contains a vector of pairs `(b,d)` for each of the bounded persistence interval $[b, d)$ in dimension $k - 1$. Again, this vector is empty if no such intervals exist.

  Note that the integer vector `filtration` has to contain every integer between 1 and n at least once, and only these integers. An error is raised if this is not the case, or if the resulting subcomplexes $X^{(k)}$ are not closed.

- The function `lefschetz_filtration` is meant to simplify the construction of the argument `filtration`, especially in the situation that $X^{(n)}$ is a proper subcomplex of some ambient Lefschetz complex X. The function is invoked using the form `lcsub, filtration = lefschetz_filtration(lc, partialfil)`. The argument `lc` contains the Lefschetz complex X. The argument `partialfil` has the type `Vector{Int}` and is of length `lc.ncells`. For each cell index `j` it contains an integer `partialfil[j]` between 0 and n. If the cell appears first in complex $X^{(k)}$, then `partialfil[j] = k`. This time, however, not all cells have to be specified, since the function automatically computes the complex closure at every level. Clearly, this means that the final complex is the closure of all cells with positive `partialfil`-values, and this can be a proper subcomplex of `lc`. The function therefore returns this subcomplex `lcsub`, together with a filtration `filtration` which satisfies the requirements of the function `persistent_homology`.

We close this section with two examples illustrating these functions. As first example, we consider the filtration given above, which consists of four simplicial complexes. In this case the persistence can be computed using the commands:

```
labels     = ["A","B","C","D","E","F","G","H"]
simplices  = [["A","B"],["A","F"],["B","F"],["B","C","G"],["D","E","H"],["C","D"],["G","H"]]
sc         = create_simplicial_complex(labels,simplices)
filtration = [1,1,1,1,1,2,1,2,
              1,2,1,2,1,1,1,1,3,3,4,
              1,4]
pinf, ppairs = persistent_homology(sc, filtration)
```

The first three lines establish the simplicial complex $X^{(4)}$, while the next command defines the filtration. For easier reading, we used different lines for the cells of each dimension. Finally, the last command computes the persistence intervals. The unbounded ones have the birth times

```
julia> pinf
3-element Vector{Vector{Int64}}:
 [1]
 [2, 4]
 []
```

while the bounded ones are given by

```
julia> ppairs
3-element Vector{Vector{Tuple{Int64, Int64}}}:
 [(2, 3)]
 [(3, 4)]
 []
```

This is in accordance with our earlier observations. Notice also that the Betti numbers of the final complex $X^{(4)}$ in the filtration can easily be determined via

```
julia> length.(pinf)
3-element Vector{Int64}:
 1
 2
 0
```

With the second example we illustrate the use of the function `lefschetz_filtration`. For this, suppose that the ambient Lefschetz complex X is the final simplicial complex in the filtration of the previous example. Within this complex, we consider the following new filtration:

- The complex $X^{(1)}$ is the closure of $\{\mathrm{CD}, \mathrm{GH}\}$.
- The complex $X^{(2)}$ is the closure of $X^{(1)} \cup \{\mathrm{BC}, \mathrm{BG}, \mathrm{DEH}\}$.
- The complex $X^{(3)}$ is the closure of $X^{(2)} \cup \{\mathrm{BCG}\}$.

The persistent intervals of this filtration can be determined using the following commands:

```
labels    = ["A","B","C","D","E","F","G","H"]
simplices = [["A","B"],["A","F"],["B","F"],["B","C","G"],["D","E","H"],["C","D"],["G","H"]]
sc        = create_simplicial_complex(labels,simplices)
tmpfil    = fill(Int(0),sc.ncells)
tmpfil[sc.indices["CD"]]  = 1
tmpfil[sc.indices["GH"]]  = 1
tmpfil[sc.indices["BC"]]  = 2
tmpfil[sc.indices["BG"]]  = 2
tmpfil[sc.indices["DEH"]] = 2
tmpfil[sc.indices["BCG"]] = 3
scsub, filtration = lefschetz_filtration(sc, tmpfil)
psinf, pspairs = persistent_homology(scsub, filtration)
```

The unbounded persistence intervals have birth times

```
julia> psinf
3-element Vector{Vector{Int64}}:
 [1]
 [2]
 []
```

while the bounded persistence intervals are

```
julia> pspairs
3-element Vector{Vector{Tuple{Int64, Int64}}}:
 [(1, 2)]
 []
 []
```

Their correctness can immediately be established.

As we mentioned earlier, more information on persistence can be found in [EH10], which also contains a detailed discussion of the implemented persistence algorithm in the context of simplicial complexes. Further examples of persistence computations for general Lefschetz complexes are given in [DW18].

4.4 References

See the full bibliography for a complete list of references cited throughout this documentation. This section cites the following references:

[DW18] P. Dłotko and T. Wanner. Rigorous cubical approximation and persistent homology of continuous functions. Computers & Mathematics with Applications **75**, 1648–1666 (2018).

[EH10] H. Edelsbrunner and J. L. Harer. Computational Topology (American Mathematical Society, Providence, 2010).

[GMW05] M. Gameiro, K. Mischaikow and T. Wanner. Evolution of pattern complexity in the Cahn-Hilliard theory of phase separation. Acta Materialia **53**, 693–704 (2005).

[KMM04] T. Kaczynski, K. Mischaikow and M. Mrozek. Computational Homology. Vol. 157 of Applied Mathematical Sciences (Springer-Verlag, New York, 2004).

[Lef42] S. Lefschetz. Algebraic Topology. Vol. 27 of American Mathematical Society Colloquium Publications (American Mathematical Society, New York, 1942).

[MB09] M. Mrozek and B. Batko. Coreduction homology algorithm. Discrete & Computational Geometry **41**, 96–118 (2009).

[Mun84] J. R. Munkres. Elements of Algebraic Topology. SpringerBriefs in Mathematics (Addison-Wesley, Menlo Park, 1984).

[GUD24] GUDHI Project. GUDHI User and Reference Manual. 3.10.1 Edition (GUDHI Editorial Board, 2024).

Chapter 5

Conley Theory

The main motivation for ConleyDynamics.jl is the development of an accessible tool for studying the global dynamics of multivector fields on Lefschetz complexes. Having already discussed the latter, we now turn our attention to multivector fields and their global dynamics. This involves a detailed discussion of multivector fields, isolated invariant sets, their Conley index, as well as Morse decompositions and connection matrices. We also describe how a variety of isolated invariant sets can be constructed using Morse decomposition intervals, and apply these tools to the analysis of simple planar and three-dimensional ordinary differential equations.

5.1 Multivector Fields

Suppose that X is a Lefschetz complex as described in Lefschetz Complexes, see in particular the definition in Basic Lefschetz Terminology. Assume further that the Lefschetz complex is defined over a field F, which is either the rational numbers $\mathbb{Q}$ or a finite field of prime order. Then a multivector field on X is defined as follows.

Definition: Multivector field

A multivector field $\mathcal{V}$ on a Lefschetz complex X is a partition of X into locally closed sets.

Recall from our detailed discussion in Basic Lefschetz Terminology that a set $V \subset X$ is called locally closed if its mouth $\operatorname{mo} V = \operatorname{cl} V \setminus V$ is closed, where closedness in turn is defined via the face relation in a Lefschetz complex. This implies that for each multivector $V \in \mathcal{V}$ the relative homology $H_*(\operatorname{cl} V, \operatorname{mo} V)$ is well-defined, and it allows for the following classification of multivectors:

- A critical multivector is a multivector for which $H_*(\operatorname{cl} V, \operatorname{mo} V) \neq 0$.
- A regular multivector is a multivector for which $H_*(\operatorname{cl} V, \operatorname{mo} V) = 0$.

Since a multivector is locally closed, it is a Lefschetz subcomplex of X as well, and we have already seen that its Lefschetz homology satisfies $H_*(V) \cong H_*(\operatorname{cl} V, \operatorname{mo} V)$. For more details, see Relative Homology.

The above classification of multivectors is motivated by the case of classical Forman vector fields. These are a special case of multivector fields, in that they also form a partition of the underlying Lefschetz complex. This time, however, there are only two types of multivectors:

- A critical cell is a multivector consisting of exactly one cell of the Lefschetz complex. One can easily see that in this case the k-th homology group is isomorphic to F, as long as the cell has dimension k. All other homology groups vanish. Thus, every critical cell is a critical multivector.

- A Forman arrow is a multivector consisting of two cells σ^- and σ^+, where σ^- is a facet of σ^+. In other words, one has to have $\kappa(\sigma^+, \sigma^-) \neq 0$, which also implies that $1 + \dim \sigma^- = \dim \sigma^+$. One can show that all homology groups of a Forman arrow are zero, and therefore it is a regular multivector.

In ConleyDynamics.jl, multivector fields can be created in two different ways. The direct method is to specify all multivectors of length larger than one in an array of type `Vector{Vector{Int}}` or `Vector{Vector{String}}`, depending on whether the involved cells are referenced via their indices or labels. Recall that it is easy to convert between these two forms using the command `convert_cellsubsets`. The subsets specified by the vector entries have to be disjoint. They do not, however, have to exhaust the underlying Lefschetz complex X. Any cells that are not part of a specified multivector will be considered as one-element critical cells. This reduces the size of the representation in many situations.

For large Lefschetz complexes, the above method becomes quickly impractical. In such a case it is easier to determine a multivector field indirectly, through a mechanism involving dynamical transitions. This is based on the following result.

Theorem: Multivector fields via dynamical transitions

Let X be a Lefschetz complex and let $\mathcal{D}$ denote an arbitrary collection of subsets of X. Then there exists a uniquely determined minimal multivector field $\mathcal{V}$ which satisfies the following:

- For every $D \in \mathcal{D}$ there exists a $V \in \mathcal{V}$ such that $D \subset V$.

Note that the sets in $\mathcal{D}$ do not have to be disjoint, and their union does not have to exhaust X. One can think of the sets in $\mathcal{D}$ as all allowable dynamical transitions.

The above result shows that as long as one has an idea about the transitions that a system has to be allowed to do, one can always find a smallest multivector field which realizes them. Needless to say, if too many transitions are specified, then it is possible that the result leads to the trivial multivector field $\mathcal{V} = \{X\}$. In most cases, however, the resulting multivector field is more useful. See also the examples later in this section of the manual.

The package ConleyDynamics.jl provides a number of functions for creating and manipulating multivector fields on Lefschetz complexes:

- The function `create_mvf_hull` implements the above theorem on dynamical transitions. It expects two input arguments: A Lefschetz complex `lc`, as well as a vector `mvfbase` that defines the dynamical transitions in $\mathcal{D}$. The latter has to have type `Vector{Vector{Int}}` or `Vector{Vector{String}}`.
- The function `mvf_information` displays basic information about a given multivector field. It expects both a Lefschetz complex and a multivector field as arguments, and returns a `Dict{String,Any}` with the information. The keys of this dictionary are as follows:
  - `"N mv"`: Number of multivectors
  - `"N critical"`: Number of critcal multivectors
  - `"N regular"`: Number of regular multivectors
  - `"Lengths critical"`: Length distribution of critical multivectors
  - `"Lengths regular"`: Length distribution of regular multivectors

  In the last two cases, the dictionary entries are vectors of pairs `(length,frequency)`, where each pair indicates that there are `frequency` multivectors of length `length`.

- The function `extract_multivectors` expects as input arguments a Lefschetz complex and a multivector field, as well as a list of cells specified as a `Vector{Int}` or a `Vector{String}`. It returns a list of all multivectors that contain the specified cells.
- The function `create_planar_mvf` creates a multivector field which approximates the dynamics of a given planar vector field. It expects as arguments a two-dimensional Lefschetz complex, a vector of planar coordinates for the vertices of the complex, as well as a function which implements the vector field. It returns a multivector field based on the dynamical transitions induced by the vector field directions on the vertices and edges of the Lefschetz complex. While the complex does not have to be a triangulation, it is expected that the one-dimensional cells are straight line segments between the two boundary vertices.
- The utility function `planar_nontransverse_edges` expects the same arguments as the previous one, and returns a list of nontransverse edges as `Vector{Int}`, which contains the corresponding edge indices. The optional parameter `npts` determines how many points along an edge are evaluated for the transversality check.
- The function `create_spatial_mvf` creates a multivector field which approximates the dynamics of a given spatial vector field. While it expects the same arguments as its planar counterpart, the Lefschetz complex has to be of one of the following two types:
  - The Lefschetz complex is a tetrahedral mesh of a region in three dimensions, i.e., it is a simplicial complex.
  - The Lefschetz complex is a three-dimensional cubical complex, i.e., it is the closure of a collection of three-dimensional cubes in space.

  In the second case, the vertex coordinates can be slightly perturbed from the original position in the cubical lattice, as long as the overall structure of the complex stays intact. In that case, the faces are interpreted as Bezier surfaces with straight edges.

All of these functions will be illustrated in more detail in the examples which are presented later in this section. See also the Tutorial for another planar vector field analysis.

5.2 Invariance and Conley Index

A multivector field induces dynamics on the underlying Lefschetz complex through the iteration of a multivalued map. This flow map is given by

$$\Pi_{\mathcal{V}}(x) = \operatorname{cl} x \cup [x]_{\mathcal{V}} \qquad \text{for all} \qquad x \in X$$

where $[x]_{\mathcal{V}}$ denotes the unique multivector in $\mathcal{V}$ which contains x. The definition of the flow map shows that the induced dynamics combines two types of behavior:

- From a cell x, it is always possible to flow towards the boundary of the cell, i.e., to any one of its faces.
- In addition, it is always possible to move freely within a multivector.

The multivalued map $\Pi_{\mathcal{V}} : X \multimap X$ naturally leads to a solution concept for multivector fields. A path is a sequence $x_0, x_1, \ldots, x_n \in X$ such that $x_k \in \Pi_{\mathcal{V}}(x_{k-1})$ for all indices $k = 1, \ldots, n$. Paths of bi-infinite length are called solutions. More precisely, a solution of the combinatorial dynamical system induced by the

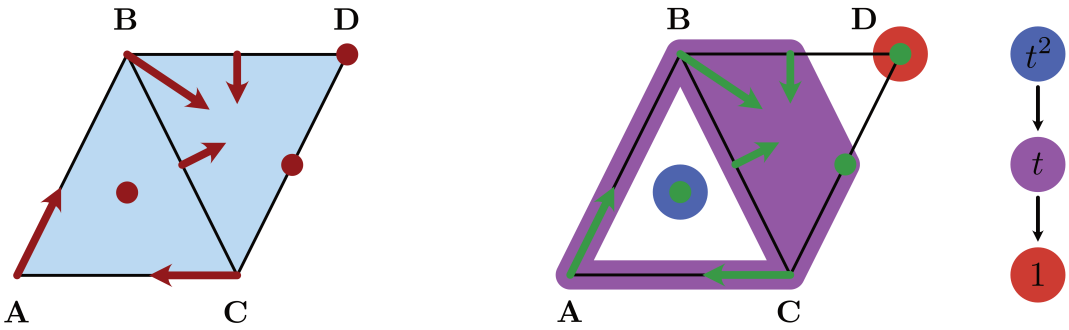


Figure 5.1: The logo multivector field

multivector field is then a map $\rho : \mathbb{Z} \to X$ which satisfies $\rho(k+1) \in \Pi_{\mathcal{V}}(\rho(k))$ for all $k \in \mathbb{Z}$. We say that this solution passes through the cell $x \in X$ if in addition one has $\rho(0) = x$. It is clear from the definition of the flow map that every constant map is a solution, since we have the inclusion $x \in \Pi_{\mathcal{V}}(x)$. Thus, rather than considering solutions in the above (classical) sense, we focus on a more restrictive notion.

Definition: Essential solution

Let $\rho : \mathbb{Z} \to X$ be a solution for the multivector field $\mathcal{V}$. Then ρ is an essential solution, if the following holds:

- If for $k \in \mathbb{Z}$ the cell $\rho(k)$ lies in a regular multivector $V \in \mathcal{V}$, then there exist integers $\ell_1 < k < \ell_2$ for which we have $\rho(\ell_i) \notin V$ for $i = 1, 2$.

In other words, an essential solution has to leave a regular multivector both in forward and in backward time. It can, however, stay in a critical multivector for as long as it wants.

The notion of essential solution has its origin in the distinction between critical and regular multivectors. In Forman's theory, which is based on classical Morse theory, critical cells correspond to stationary solutions or equilibria of the underlying flow. Thus, it has to be possible to stay in a critical multivector for all times, whether in forward or backward time, or even for all times. On the other hand, a Forman arrow indicates prescribed non-negotiable motion, and therefore a regular multivector corresponds to motion which goes from the multivector to its mouth.

The multivector field from the package logo, which is shown in the accompanying image, consists of three critical cells, two Forman arrows, as well as one multivector which consists of four cells. Beyond the constant essenetial solutions in each of the three critical cells, another essential solution is the periodic orbit

$$\rho_P \text{ given by } \ldots \to \mathbf{A} \to \mathbf{AB} \to \mathbf{B} \to \mathbf{BCD} \to \mathbf{C} \to \mathbf{AC} \to \mathbf{A} \to \ldots$$

Notice that this is just one of many realizations of this particular periodic motion, since an essential solution can take many different paths through a multivector.

Using the concept of essential solutions we can now introduce the notion of invariance. Informally, we say that a subset of a Lefschetz complex is invariant if through every cell in the set there exists an essential solution which stays in the set. In other words, we have the choice of staying in the set, even though there might be other solutions that do leave. More generally, for every subset $A \subset X$ one can ask whether there are elements

$x \in A$ for which there exists an essential solution which passes through x and stays in A for all times. This leads to the definition of the invariant part of A as

$$\mathrm{Inv}_{\mathcal{V}}(A) = \{x \in A \,:\, \text{there exists an essential solution } \rho : \mathbb{Z} \to A \text{ through } x\}$$

It is certainly possible that the invariant part of a set is empty. If, however, the invariant part of A is all of A, i.e., if we have $\mathrm{Inv}_{\mathcal{V}}(A) = A$, then the set A is called invariant. In the context of our above logo example, the image of the essential solution ρ_P is clearly an invariant set.

Invariant sets are the fundamental building blocks for the global dynamics of a dynamical system. Yet, in general they are difficult to study. Conley realized in [Con78] that if one restricts the attention to a more specialized notion of invariance, then topological methods can be used to formulate a coherent general theory. For this, we need to introduce the notion of isolated invariant set:

Definition: Isolated invariant set

A closed set $N \subset X$ isolates an invariant set $S \subset N$, if the following two conditions are satisfied:

- Every path in N with endpoints in S is a path in S.
- We have $\Pi_{\mathcal{V}}(S) \subset N$.

An invariant set S is an isolated invariant set, if there exists a closed set N which isolates S.

It is clear that the whole Lefschetz complex X isolates its invariant part. Therefore, the set $\mathrm{Inv}_{\mathcal{V}}(X)$ is an isolated invariant set. Moreover, one can readily show that if N is an isolating set for an isolated invariant set S, then any closed set $S \subset M \subset N$ also isolates S. Thus, the closure $\mathrm{cl}\, S$ is the smallest isolating set for S. With these observations in mind, one obtains the following result from [LKMW23]:

Theorem: Characterization of isolated invariant sets

An invariant set $S \subset X$ is an isolated invariant set, if and only if the following two conditions hold:

- S is $\mathcal{V}$-compatible, i.e., it is the union of multivectors.
- S is locally closed.

In this case, the isolated invariant set S is isolated by its closure $\mathrm{cl}\, S$.

Returning to our earlier logo example, notice that the cells visited by the periodic essential solution ρ_P do not form an isolated invariant set, but rather just an invariant set. However, if we consider the larger set S_P which consists of all cells except for the cells $\mathbf{ABC}$ and $\mathbf{D}$, then we do obtain an isolated invariant set which contains the periodic orbit ρ_P.

With this characterization at hand, identifying isolated invariant sets becomes straightforward. In addition, since isolated invariant sets are locally closed, we can now also define their Conley index:

Definition: Conley index

Let $S \subset X$ be an isolated invariant set the multivalued flow map $\Pi_{\mathcal{V}}$. Then the Conley index of S is the relative (or Lefschetz) homology

$$CH_*(S) = H_*(\operatorname{cl} S, \operatorname{mo} S) \cong H_*(S)$$

In addition, the Poincare polynomial of S is defined as

$$p_S(t) = \sum_{k=0}^{\infty} \beta_k(S) t^k \,, \quad \text{where} \quad \beta_k(S) = \dim CH_k(S) \,.$$

The Poincare polynomial is a concise way to encode the homology information.

Since the Conley index is nothing more than the relative homology of the closure-mouth-pair associated with a locally closed set, one could easily use the homology functions described in Homology for its computation. However, we have included a wrapper function to keep the notation uniform. In addition, ConleyDynamics.jl contains a function which provides basic information about an isolated invariant set. These two functions can be described as follows:

- The function `conley_index` determines the Conley index of an isolated invariant set. It expects a Lefschetz complex as its first argument, while the second one has to be a list of cells which specifies the isolated invariant set, and which is either of type `Vector{Vector{Int}}` or `Vector{Vector{String}}`. An error is raised if the second argument does not specify a locally closed set.
- The function `isoinvset_information` expects a Lefschetz complex `lc::LefschetzComplex`, a multivector field `mvf::CellSubsets`, as well as an isolated invariant set `iis::Cells` as its three arguments. It returns a `Dict{String,Any}` with the information. The keys of this dictionary are as follows:
  - `"Conley index"` contains the Conley index of the isolated invariant set.
  - `"N multivectors"` contains the number of multivectors in the isolated invariant set.

5.3 Morse Decompositions

We now turn our attention to the global dynamics of a combinatorial dynamical system. This is accomplished through the notion of Morse decomposition, and it requires some auxilliary definitions:

- Suppose we are given a solution $\varphi : \mathbb{Z} \to X$ for the multivector field $\mathcal{V}$. Then the long-term limiting behavior of φ can be described using the ultimate backward and forward images

  $$\operatorname{uim}^- \varphi = \bigcap_{t \in \mathbb{Z}^-} \varphi\left((-\infty, t]\right) \qquad \text{and} \qquad \operatorname{uim}^+ \varphi = \bigcap_{t \in \mathbb{Z}^+} \varphi\left([t, +\infty)\right).$$

  Notice that since X is finite, there has to exist a $k \in \mathbb{N}$ such that

  $$\operatorname{uim}^- \varphi = \varphi\left((-\infty, -k]\right) \neq \emptyset \qquad \text{and} \qquad \operatorname{uim}^+ \varphi = \varphi\left([k, +\infty)\right) \neq \emptyset.$$

- The $\mathcal{V}$-hull of a set $A \subset X$ is the intersection of all $\mathcal{V}$-compatible and locally closed sets containing A. It is denoted by $\langle A \rangle_{\mathcal{V}}$, and is the smallest candidate for an isolated invariant set which contains A.
- The α- and ω-limit sets of φ are then defined as

$$\alpha(\varphi) = \left\langle \mathrm{uim}^{-}\varphi \right\rangle_{\mathcal{V}} \qquad \text{and} \qquad \omega(\varphi) = \left\langle \mathrm{uim}^{+}\varphi \right\rangle_{\mathcal{V}}.$$

While in general the $\mathcal{V}$-hull of a set does not have to be invariant, the following result shows that for every essential solution both of its limit sets are in fact isolated invariant sets.

Theorem: Limit sets are nontrivial

Let φ be an essential solution in X. Then both limit sets $\alpha(\varphi)$ and $\omega(\varphi)$ are nonempty isolated invariant sets.

We briefly pause to illustrate these concepts in the context of the above logo example. For the periodic essential solution ρ_P, both its ultimate backward and forward images are precisely the cells visited by the solution. The $\mathcal{V}$-hull of $\mathrm{im}\,\rho_P$ is the set S_P which consists of all cells except the index 0 and 2 critical cells. It was already mentioned earlier that this indeed defines an isolated invariant set.

The above notions allow us to decompose the global dynamics of a multivector field. Loosely speaking, this is accomplished by separating the dynamics into a recurrent part given by an indexed collection of isolated invariant sets, and the gradient dynamics between them. This can be abstracted through the concept of a Morse decomposition.

Definition: Morse decomposition

Assume that X is an invariant set for the multivector field $\mathcal{V}$ and that $(\mathbb{P}, \leq)$ is a finite poset. Then an indexed collection $\mathcal{M} = \{M_p : p \in \mathbb{P}\}$ is called a Morse decomposition of X if the following conditions are satisfied:

- The indexed family $\mathcal{M}$ is a family of mutually disjoint, isolated invariant subsets of X.
- For every essential solution φ in X either one has $\mathrm{im}\,\varphi \subset M_r$ for an $r \in \mathbb{P}$ or there exist two poset elements $p, q \in \mathbb{P}$ such that $q > p$ and

$$\alpha(\varphi) \subset M_q \qquad \text{and} \qquad \omega(\varphi) \subset M_p.$$

The elements of $\mathcal{M}$ are called Morse sets. We would like to point out that some of the Morse sets could be empty.

Given a combinatorial multivector field $\mathcal{V}$ on an arbitrary Lefschetz complex X, there always exists a finest Morse decomposition $\mathcal{M}$. It can be found by determining those strongly connected components of the digraph associated with the multivalued flow map $\Pi_{\mathcal{V}} : X \multimap X$ which contain essential solutions. The associated Conley-Morse graph is the partial order induced on $\mathcal{M}$ by the existence of connections, and represented as a directed graph labelled with the Conley indices of the isolated invariant sets in $\mathcal{M}$ in terms of their Poincare polynomials.

In order to capture the dynamics between two subsets $A, B \subset X$ one can define the connection set from A to B as the cell collection

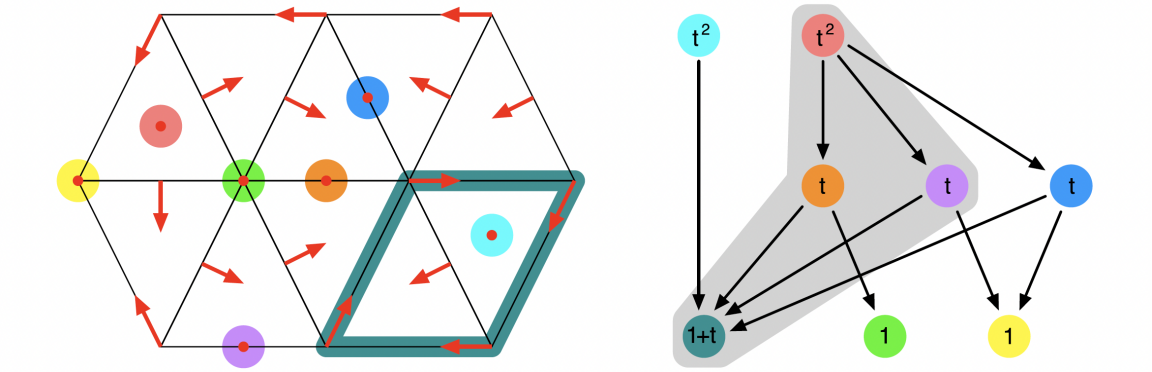


Figure 5.2: Morse decomposition of the planar flow

$$\mathcal{C}(A,B) = \{x \in X \,:\, \exists \text{ essential solution } \varphi \text{ through } x \text{ with } \alpha(\varphi) \subset A \text{ and } \omega(\varphi) \subset B\}\,.$$

Then $\mathcal{C}(A,B)$ is an isolated invariant set. We would like to point out, however, that the connection set can be, and in fact will be, empty in many cases.

While the Morse sets of a Morse decomposition are the fundamental building blocks for the global dynamics, there usually are many additional isolated invariant sets for the multivector field $\mathcal{V}$. Of particular interest are Morse intervals. To define them, let $I \subset \mathbb{P}$ denote an interval in the index poset. Then

$$M_I \;=\; \bigcup_{p\in I} M_p \;\cup\; \bigcup_{p,q\in I} \mathcal{C}(M_q, M_p)$$

is always an isolated invariant set. Nevertheless, not every isolated invariant set is of this form. For example, the figure contains the multivector field which was discussed in [BKMW20, Figure 3]. While the underlying simplicial complex and the Forman vector field are depicted in the left panel, the associated Conley-Morse graph is shown on the right. For this combinatorial dynamical system, there exists an isolated invariant set which contains only the four Morse sets within the gray region under the graph. More details can be found in A Planar Forman Vector Field.

Morse decompositions and intervals can be easily computed and manipulated in ConleyDynamics.jl using the following commands:

- The function `morse_sets` expects a Lefschetz complex and a multivector field as arguments, and returns the Morse sets of the finest Morse decomposition as a `Vector{Vector{Int}}` or `Vector{Vector{String}}`, matching the format used for the multivector field. If the optional argument `poset=true` is added, then the function also returns a matrix which encodes the Hasse diagram of the poset $\mathbb{P}$. Note that this is the transitive reduction of the full poset, i.e., it only contains necessary relations.
- The function `morse_interval` computes the isolated invariant set for a Morse set interval. The three input arguments are the underlying Lefschetz complex, a multivector field, and a collection of Morse sets. The latter should be determined using the function `morse_sets`. The function returns the smallest isolated invariant set which contains the Morse sets and their connections as a `Vector{Int}`. The result can be converted to label form using `convert_cells`.

- The function `restrict_dynamics` restricts a multivector field to a Lefschetz subcomplex. The function expects three arguments: A Lefschetz complex `lc`, a multivector field `mvf`, and a subcomplex of the Lefschetz complex which is given by the locally closed set represented by `lcsub`. It returns the associated Lefschetz subcomplex `lcreduced` and the induced multivector field `mvfreduced` on the subcomplex. The multivectors of the new multivector field are the intersections of the original multivectors and the subcomplex.

- Finally, the function `remove_exit_set` removes the exit set for a multivector field on a Lefschetz subcomplex. It is assumed that the Lefschetz complex `lc` is a topological manifold and that `mvf` contains a multivector field that is created via either `create_planar_mvf` or `create_spatial_mvf`. The function identifies cells on the boundary at which the flows exits the region covered by the Lefschetz complex. If this exit set is closed, one has found an isolated invariant set and the function returns a Lefschetz complex `lcr` restricted to it, as well as the restricted multivector field `mvfr`. If the exit set is not closed, a warning is displayed and the function returns the restricted Lefschetz complex and multivector field obtained by removing the closure of the exit set. In the latter case, unexpected results might be obtained.

The first two of these functions rely heavily on the Julia package Graphs.jl.

5.4 Connection Matrices

While a Morse decomposition represents the basic structure of the global dynamics of a combinatorial dynamical system, it does not directly provide more detailed information about the dynamics between them – except for the poset order on the Morse sets. But which of the associated connecting sets actually have to be nonempty? The algebra behind this question is captured by the connection matrix. The precise notion of connection matrix was introduced in [Fra89], see also [HMS21], as well as the book [MW25] which treats connection matrices specifically in the setting of multivector fields and provides a precise definition of connection matrix equivalence, even across varying posets.

Since the precise definition of a connection matrix is beyond the scope of this manual, we only state what it is as an object, what its main properties are, and how it can be computed in ConleyDynamics.jl. Assume therefore that we are given a Morse decomposition $\mathcal{M}$ of an isolated invariant set S. Then the connection matrix is a linear map

$$\Delta \ : \ \bigoplus_{q\in\mathbb{P}} CH_*(M_q) \to \bigoplus_{p\in\mathbb{P}} CH_*(M_p),$$

i.e., it is a linear map which is defined on the direct sum of all Conley indices of the Morse sets in the Morse decomposition. One usually writes the connection matrix Δ as a matrix in the form $\Delta = (\Delta(p,q))_{p,q\in\mathbb{P}}$, which is indexed by the poset $\mathbb{P}$, and where the entries $\Delta(p,q) : CH_*(M_q) \to CH_*(M_p)$ are linear maps between homological Conley indices. If I denotes an interval in the poset $\mathbb{P}$, then one further defines the restricted connection matrix

$$\Delta(I) \ = \ (\Delta(p,q))_{p,q\in I} \ : \ \bigoplus_{p\in I} CH_*(M_p) \to \bigoplus_{p\in I} CH_*(M_p).$$

Any connection matrix Δ has the following fundamental properties:

- The matrix Δ is strictly upper triangular, i.e., if $\Delta(p,q) \neq 0$ then $p < q$.

- The matrix Δ is a boundary operator, i.e., we have $\Delta \circ \Delta = 0$, and Δ maps k-th level homology to $(k-1)$-st level homology for all $k \in \mathbb{Z}$.
- For every interval I in $\mathbb{P}$ we have

$$H_*\Delta(I) \;=\; \ker \Delta(I)/\mathrm{im}\,\Delta(I) \;\cong\; CH_*(M_I).$$

  In other words, the Conley index of a Morse interval can be determined via the homology of the associated connection matrix minor $\Delta(I)$.
- If $\{p, q\}$ is an interval in $\mathbb{P}$ and $\Delta(p, q) \neq 0$, then the connection set $\mathcal{C}(M_q, M_p)$ is not empty.

We would like to point out that these properties do not characterize connection matrices. In practice, a given multivector field can have several different connection matrices. These in some sense encode different types of dynamical behavior that can occur in the system. Nonuniqueness, however, cannot be observed if the underlying system is a gradient combinatorial Forman vector field on a Lefschetz complex. These are multivector fields in which every multivector is either a singleton, and therefore a critical cell, or a two-element Forman arrow. In addition, a gradient combinatorial Forman vector field cannot have any nontrivial periodic solutions, i.e., periodic solutions which are not constant and therefore critical cells. For such combinatorial vector fields, the following result was shown in [MW25].

Theorem: Uniqueness of connection matrices

If $\mathcal{V}$ is a gradient combinatorial Forman vector field and $\mathcal{M}$ its finest Morse decomposition, then the connection matrix is uniquely determined.

In ConleyDynamics.jl connection matrices can be computed over arbitrary finite fields or the rationals, using the persistence-like algorithm introduced in [DLMS24]:

- The function `connection_matrix` computes a connection matrix for the multivector field `mvf` on the Lefschetz complex `lc` over the field associated with the Lefschetz complex boundary matrix. The function returns an object of type `ConleyMorseCM`, which is further described below. If the optional argument `returnbasis=true` is given, then the function also returns a dictionary which gives the basis for the connection matrix columns in terms of the original cell labels.

The connection matrix is returned in an object with the composite data type `ConleyMorseCM`. Its docstring is as follows:

`ConleyDynamics.ConleyMorseCM` – Type.

```
ConleyMorseCM{T}
```

Collect the connection matrix information in a struct.

The struct has the following fields:

- `matrix::SparseMatrix{T}`: Connection matrix
- `columns::Vector{Int}`: Corresponding columns in the boundary matrix
- `poset::Vector{Int}`: Poset indices for the connection matrix columns

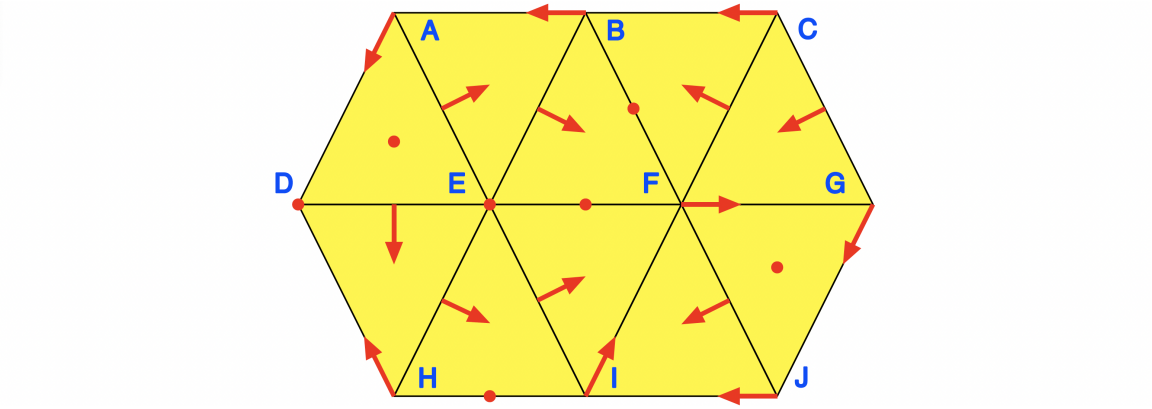


Figure 5.3: A planar simplicial complex flow

- `labels::Vector{String}`: Labels for the connection matrix columns
- `morse::Vector{Vector{String}}`: Vector of Morse sets in original complex
- `conley::Vector{Vector{Int}}`: Vector of Conley indices for the Morse sets
- `complex::LefschetzComplex`: The Conley complex as a Lefschetz complex

source

To illustrate these fields further, we briefly illustrate them for the example associated with the last figure, see again A Planar Forman Vector Field. For reference, the underlying simplicial complex and Forman vector field are shown in the next figure.

The underlying Lefschetz complex, multivector field, and connection matrix can be computed over the field $GF(2)$ as follows:

```
lc, mvf, coords = example_forman2d()
cm = connection_matrix(lc, mvf)
sparse_show(cm.matrix)
```

```
0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

The field `cm.poset` indicates which Morse set each column belongs to, while the field `cm.labels` shows which cell label the column corresponds to. For the example one obtains:

```
print(cm.poset)
```

```
[1, 2, 3, 3, 4, 5, 6, 7, 8]
```

```
print(cm.labels)
```

```
["D", "E", "F", "GJ", "BF", "EF", "HI", "ADE", "FGJ"]
```

Note that except for the third and fourth column, all columns belong to unique Morse sets whose Conley index is a one-dimensional vector space. The third and fourth column correspond to the periodic orbit, whose Conley index is a two-dimensional vector space. The Conley indices for all eight Morse sets can be seen in the field `cm.conley`:

```
cm.conley
```

```
8-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [1, 0, 0]
 [1, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 0, 1]
 [0, 0, 1]
```

The full associated Morse sets are list in `cm.morse`:

```
cm.morse
```

```
8-element Vector{Vector{String}}:
 ["D"]
 ["E"]
 ["F", "G", "I", "J", "FG", "FI", "GJ", "IJ"]
 ["BF"]
 ["EF"]
 ["HI"]
 ["ADE"]
 ["FGJ"]
```

As the final struct field, the entry `cm.complex` returns the connection matrix as a Lefschetz complex in its own right. This is useful for determining the Conley indices of Morse intervals. In our example, the cells of the new Lefschetz complex are given by

```
cm.complex.labels
```

```
9-element Vector{String}:
 "D"
 "E"
 "F"
 "GJ"
 "BF"
 "EF"
 "HI"
 "ADE"
 "FGJ"
```

The Morse interval consisting of the two index 2 critical cells $\mathbf{ADE}$ and $\mathbf{FGJ}$ should have as Conley index the sum of the two individual indices, and the following computation demonstrates this:

```
conley_index(cm.complex, ["ADE", "FGJ"])
```

```
3-element Vector{Int64}:
 0
 0
 2
```

In contrast, since there is exactly one connecting orbit between $\mathbf{ADE}$ and $\mathbf{BF}$, the Conley index of this interval should be trivial:

```
conley_index(cm.complex, ["ADE", "BF"])
```

```
3-element Vector{Int64}:
 0
 0
 0
```

Finally, there are exactly two connecting orbits between the Morse sets $\mathbf{ADE}$ and $\mathbf{EF}$, and therefore the Conley index of this last interval is again the sum of the separate indices:

```
conley_index(cm.complex, ["ADE", "EF"])
```

```
3-element Vector{Int64}:
 0
 1
 1
```

In order to simplify the inspection of connection matrices, the function `sparse_show` has a special method for an argument of type `ConleyMorseCM`. In our example, it produces the following output:

```
sparse_show(cm)
```

```
    ┆   D   E   F  GJ  BF  EF  HI ADE FGJ
────┆──────────────────────────────────────
  D ┆   0   0   0   0   1   0   1   0   0
  E ┆   0   0   0   0   0   1   0   0   0
  F ┆   0   0   0   0   1   1   1   0   0
 GJ ┆   0   0   0   0   0   0   0   0   1
 BF ┆   0   0   0   0   0   0   0   1   0
 EF ┆   0   0   0   0   0   0   0   0   0
 HI ┆   0   0   0   0   0   0   0   1   0
ADE ┆   0   0   0   0   0   0   0   0   0
FGJ ┆   0   0   0   0   0   0   0   0   0
```

In this way, one can easily see which Morse sets correspond to the columns and rows of the connection matrix.

5.5 Extracting Subsystems

We briefly return to one of the examples in the tutorial. More precisely, we consider the planar ordinary differential equation given by

$$\begin{aligned} \dot{x}_1 &= x_1\left(1 - x_1^2 - 3x_2^2\right) \\ \dot{x}_2 &= x_2\left(1 - 3x_1^2 - x_2^2\right) \end{aligned}$$

The dynamics of this system is characterized by the existence of a global attractor in the shape of a closed disk. Inside the attractor, there are nine different Morse sets:

- The origin is an equilibrium of index 2, i.e., it is an unstable stationary state with a two-dimensional unstable manifold.
- The four points $(\pm 1/2, \pm 1/2)$ are unstable equilibria of index 1, i.e., with a one-dimensional unstable manifold.
- Finally, the four points $(\pm 1, 0)$ and $(0, \pm 1)$ are asymptotically stable stationary states.

We saw in the tutorial that the Morse decomposition of this system can easily be found using ConleyDynamics.jl, as well as the associated connection matrix. Yet, in certain situations one might only be interested in part of the dynamics on the attractor. Moreover, while the Morse sets describe the recurrent part of the dynamics, they do not provide information on the geometry of the connecting sets between the Morse sets. In the following, we illustrate how this can be analyzed further.

The right-hand side of the above vector field can be implemented using the Julia function

```
function planarvf(x::Vector{Float64})
    #
    # Sample planar vector field with nontrivial Morse decomposition
    #
    x1, x2 = x
    y1 = x1 * (1.0 - x1*x1 - 3.0*x2*x2)
    y2 = x2 * (1.0 - 3.0*x1*x1 - x2*x2)
    return [y1, y2]
end
```

```
planarvf (generic function with 1 method)
```

To analyze the resulting global dynamical behavior, we first create a simplicial mesh covering the square $[-6/5, 6/5]^2$ using the commands

```
lc, coords = create_simplicial_delaunay(300, 300, 5, 50);
coordsN = convert_planar_coordinates(coords,[-1.2,-1.2], [1.2,1.2]);
lc.ncells
```

```
14395
```

The integer in the output gives the number of cells in the created Lefschetz complex X. Note that we are using a Delaunay triangulation over an initial box of size 300×300, where the target triangle size is about 5. This box is then rescaled to cover the above square. We can then create a multivector field on the simplicial complex `lc` and find its Morse decomposition using the commands

```
mvf = create_planar_mvf(lc, coordsN, planarvf);
morsedecomp = morse_sets(lc, mvf);
length(morsedecomp)
```

```
9
```

As expected, ConleyDynamics.jl finds exactly nine Morse sets. Their Conley indices can be computed and stored in a `Vector{Vector{Int}}` using the command

```
conleyindices = [conley_index(lc, mset) for mset in morsedecomp]
```

```
9-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [1, 0, 0]
 [0, 1, 0]
 [1, 0, 0]
```

```
[0, 1, 0]
[1, 0, 0]
[0, 1, 0]
[0, 1, 0]
[0, 0, 1]
```

These Conley indices correspond to the dynamical behavior near the equilibrium solutions described above.

Suppose now that rather than finding the connection matrix for the complete Morse decomposition, we would only like to consider a part of it. This can be done as long as we restrict our attention to an interval in the Morse decomposition. Such an interval $\mathcal{I}$ can be created from a selection $\mathcal{S}$ of the Morse sets in the following way:

- In addition to the Morse sets in $\mathcal{S}$, the interval $\mathcal{I}$ contains all Morse sets that lie between two Morse sets in $\mathcal{S}$ with respect to the poset order underlying the Morse decomposition. Recall that this poset order can be computed via `morse_sets` by activating the extra return object `hasse`, which describes the Hasse diagram of the poset.

With every interval $\mathcal{I}$ of the Morse decomposition one can assign a smallest isolated invariant set $X_{\mathcal{I}} \subset X$ which describes the complete dynamics within and between the Morse sets in $\mathcal{I}$. In fact, in can be characterized as follows:

- The set $X_{\mathcal{I}}$ consists of all cells in the underlying Lefschetz complex X through which one can find a solution which originates in one Morse set of $\mathcal{I}$ and ends in another Morse set of $\mathcal{I}$, where the two involved Morse sets can be the same. In other words, one needs to combine the interval Morse sets with all connecting orbits between them.

The two above steps can be performed in ConleyDynamics.jl using the function `morse_interval`.

In our example, we consider two intervals. The first interval consists of the five Morse sets corresponding to all unstable equilibrium solutions, while the second one considers the four index 1 and the four stable stationary states. The associated isolated invariant sets for these two intervals can be computed as follows:

```
subset1 = findall(x -> x[2]+x[3]>0, conleyindices);
subset2 = findall(x -> x[1]+x[2]>0, conleyindices);
lcsub1 = morse_interval(lc, mvf, morsedecomp[subset1]);
lcsub2 = morse_interval(lc, mvf, morsedecomp[subset2]);
[length(subset1), length(subset2), length(lcsub1), length(lcsub2)]
```

```
4-element Vector{Int64}:
    5
    8
 1201
 2256
```

The output shows that we have in fact extracted five and eight Morse sets, respectively. It also shows that the Lefschetz complexes corresponding to these two isolated invariant sets are much smaller than X.

So far, we have just determined the collections of cells that correspond to the two isolated invariant sets for these intervals. We can now restrict the combinatorial dynamics to these subsets. Note that since they are both isolated invariant sets, they are locally closed in X, and therefore the restrictions provide us with two new Lefschetz complexes `lcr1` and `lcr2`, along with induced multivector fields `mvfr1` and `mvfr2`, respectively. In ConleyDynamics.jl, this is achieved using the commands

```
lcr1, mvfr1 = restrict_dynamics(lc, mvf, lcsub1);
lcr2, mvfr2 = restrict_dynamics(lc, mvf, lcsub2);
[lcr1.ncells, lcr2.ncells]
```

```
2-element Vector{Int64}:
 1201
 2256
```

It is now easy to find the connection matrices for these two intervals. The first connection matrix is given by

```
cmr1 = connection_matrix(lcr1, mvfr1);
cmr1.conley
```

```
5-element Vector{Vector{Int64}}:
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 0, 1]
```

```
full_from_sparse(cmr1.matrix)
```

```
5×5 Matrix{Int64}:
 0  0  0  0  1
 0  0  0  0  1
 0  0  0  0  1
 0  0  0  0  1
 0  0  0  0  0
```

It clearly shows that the unstable index 2 Morse set has connecting orbits to every one of the four index 1 equilibria. Similarly, the second connection matrix can be determined as

```
cmr2 = connection_matrix(lcr2, mvfr2);
cmr2.conley
```

```
8-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [1, 0, 0]
 [1, 0, 0]
 [1, 0, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
```

```
full_from_sparse(cmr2.matrix)
```

```
8×8 Matrix{Int64}:
 0  0  0  0  0  1  1  0
 0  0  0  0  0  0  1  1
 0  0  0  0  1  1  0  0
 0  0  0  0  1  0  0  1
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
```

In this case, every index 1 equilibrium is connected two its two neighboring stable stationary states via heteroclinics that are detected by the connection matrix.

The Lefschetz complexes associated with the two Morse decomposition intervals can also be visualized in ConleyDynamics.jl. For this, recall that the function `plot_planar_simplicial_morse` can plot an underlying simplicial complex together with any collection of cell subsets. For our purposes, we use the following commands:

```
show1 = [[lcr1.labels]; cmr1.morse];
show2 = [[lcr2.labels]; cmr2.morse];
fname1 = "/Users/wanner/Desktop/invariantinterval2d1.png"
fname2 = "/Users/wanner/Desktop/invariantinterval2d2.png"
plot_planar_simplicial_morse(lc, coordsN, fname1, show1, vfac=1.1, hfac=2.0)
plot_planar_simplicial_morse(lc, coordsN, fname2, show2, vfac=1.1, hfac=2.0)
```

The variable `show1` collects not only the Morse sets that are part of the first connection matrix `cmr1`, but also the support of the Lefschetz complex `lcr1`. This support is accessed via `[lcr1.labels]`, and we add it as a first vector of cells in `show1`. Similarly, we determine the support of the second isolated invariant set, together with the Morse sets of `cmr2`. The remaining four commands create two images.

The first image shows the five Morse sets surrounding the stationary states at the origin and at $(\pm 1/2, \pm 1/2)$. In addition, it highlights the support of the isolated invariant set associated with this Morse decomposition interval. One can clearly see rough outer approximations for the four heteroclinics which start at the origin and end at the index 1 equilibria. These approximations are necessarily coarse, since we are not working with a very fine triangulation.

Finally, the second image depicts the eight Morse sets enclosing the index 1 and the stable stationary states. It also shows the support of the Lefschetz complex `lcr2` which is associated with this Morse decomposition

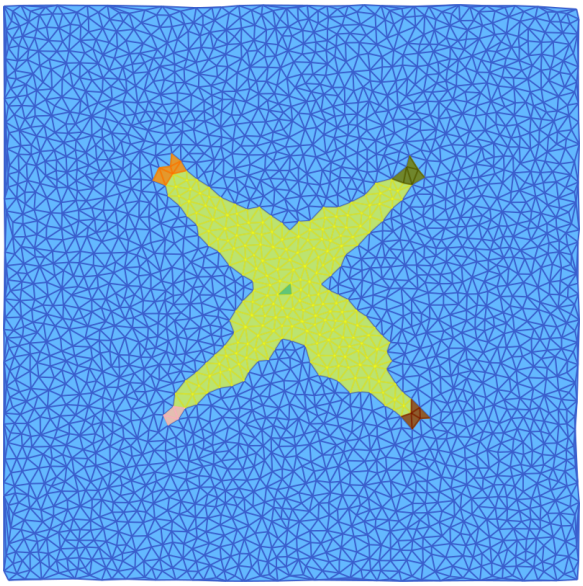


Figure 5.4: Interval support for the first interval

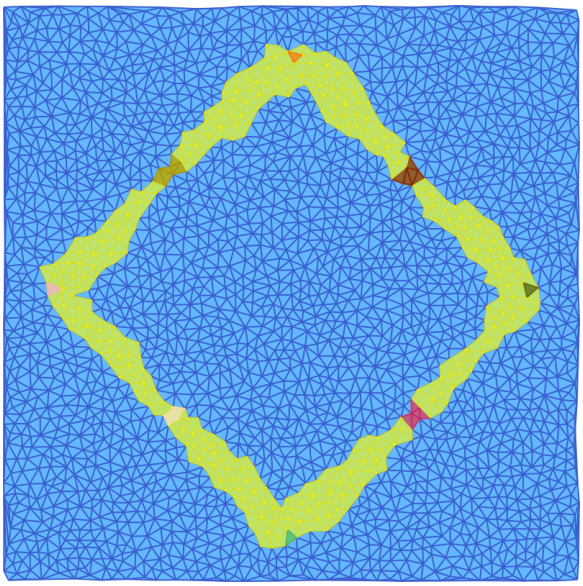


Figure 5.5: Interval support for the second interval

interval. In this case, it covers eight different heteroclinic orbits, which are in fact better approximated than the four in the previous image.

5.6 Analysis of a Planar System

Our next example illustrates how ConleyDynamics.jl can be used to analyze the global dynamics of a planar ordinary differential equations. For this, consider the planar system

$$\begin{aligned} \dot{x}_1 &= x_2 - x_1\left(x_1^2 + x_2^2 - 4\right)\left(x_1^2 + x_2^2 - 1\right) \\ \dot{x}_2 &= -x_1 - x_2\left(x_1^2 + x_2^2 - 4\right)\left(x_1^2 + x_2^2 - 1\right) \end{aligned}$$

This system has already been considered in [MSTW22]. The right-hand side of this vector field can be implemented using the Julia function

```
function circlevf(x::Vector{Float64})
    #
    # Sample vector field with nontrivial Morse decomposition
    #
    x1, x2 = x
    c0 = x1*x1 + x2*x2
    c1 = (c0 - 4.0) * (c0 - 1.0)
    y1 =  x2 - x1 * c1
    y2 = -x1 - x2 * c1
    return [y1, y2]
end
```

```
circlevf (generic function with 1 method)
```

To analyze the global dynamics of this vector field, we first create a cubical complex covering the square $[-3,3]^2$ using the commands

```
n = 51
lc, coords = create_cubical_rectangle(n,n,p=2);
coordsN = convert_planar_coordinates(coords,[-3.0,-3.0],[3.0,3.0]);
lc.ncells
```

```
10609
```

As the last result shows, this gives a Lefschetz complex with 10609 cells. The multivector field can be generated using

```
mvf = create_planar_mvf(lc, coordsN, circlevf);
length(mvf)
```

```
2449
```

This multivector field consists of 2437 multivectors. Finally, the connection matrix can be determined using the command

```
cm = connection_matrix(lc, mvf);
cm.conley
```

```
3-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [1, 0, 0]
 [0, 1, 1]
```

Therefore, the above planar system has three isolated invariant sets. One has the Conley index of a stable equilibrium, while the other two have that of a stable and an unstable periodic orbit. The columns of the connection matrix correspond to these invariant sets as follows

```
cm.poset
```

```
5-element Vector{Int64}:
 1
 1
 2
 3
 3
```

The connection matrix itself is given by

```
sparse_show(cm)
```

```
       ┆ 0820.00 3735.10 2525.00 2020.01 3430.11
-------┆-----------------------------------------
0820.00┆       0       0       0       1       0
3735.10┆       0       0       0       0       1
2525.00┆       0       0       0       1       0
2020.01┆       0       0       0       0       0
3430.11┆       0       0       0       0       0
```

This implies that there are connecting orbits from the unstable periodic orbit to both the stable equilibrium, and the stable periodic orbit. To visualize these Morse sets, we employ the commands

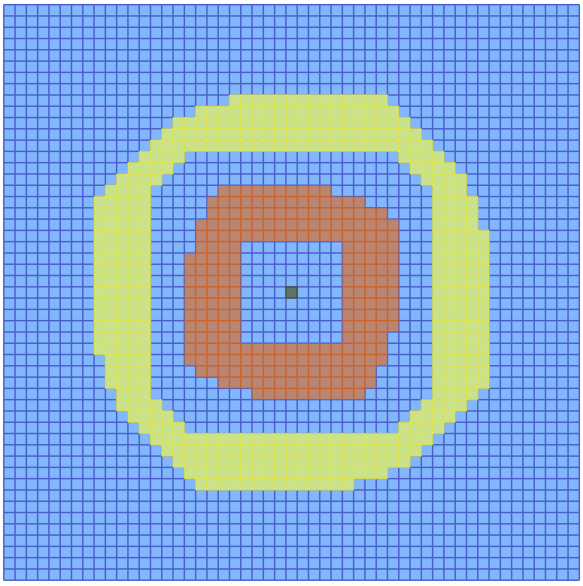


Figure 5.6: Morse sets of the planar circles vector field

```
fname = "cubicalcircles.pdf"
plot_planar_cubical_morse(lc, fname, cm.morse, pv=true)
```

In the above example we used the original fixed cubical grid, which is just a scaled version of the grid on the integer lattice. It is also possible to work with a randomized grid, in which the coordinates of the vertices are randomly perturbed. This can be achieved with the following commands:

```
nR = 75
lcR, coordsR = create_cubical_rectangle(nR,nR,p=2,randomize=0.33);
coordsRN = convert_planar_coordinates(coordsR,[-3.0,-3.0],[3.0,3.0]);
mvfR = create_planar_mvf(lcR, coordsRN, circlevf);
cmR = connection_matrix(lcR, mvfR);
fnameR = "cubicalcirclesR.pdf"
plot_planar_cubical_morse(lcR, coordsRN, fnameR, cmR.morse, pv=true, vfac=1.1, hfac=2.0)
```

To constrast the above example with the use of a Delaunay triangulation, we reanalyze the vector field in the following way:

```
lc2, coords2 = create_simplicial_delaunay(400, 400, 10, 30, p=2)
coords2N = convert_planar_coordinates(coords2,[-3.0,-3.0], [3.0,3.0])
mvf2 = create_planar_mvf(lc2, coords2N, circlevf)
cm2 = connection_matrix(lc2, mvf2)

fname2 = "cubicalcircles2.pdf"
plot_planar_simplicial_morse(lc2, coords2N, fname2, cm2.morse, pv=true)
```

In this case, the Morse sets can be visualized as in the figure.

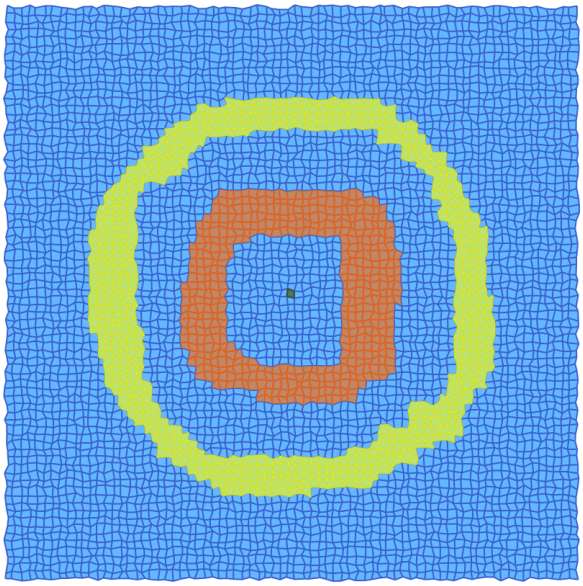


Figure 5.7: Morse sets of the planar circles vector field via randomized cubes

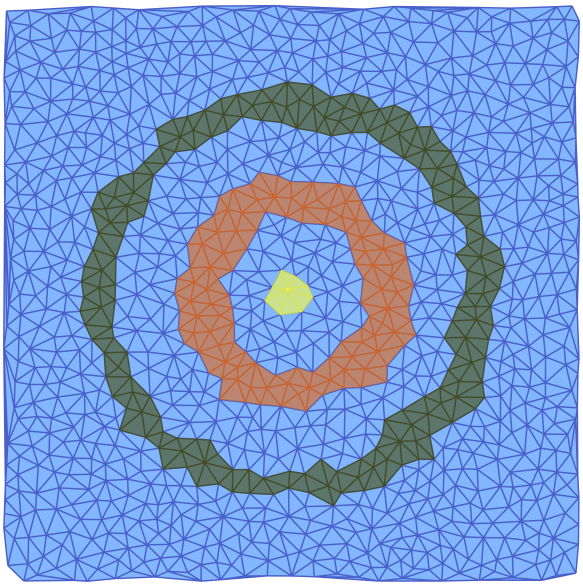


Figure 5.8: Morse sets of the planar circles vector field via Delaunay

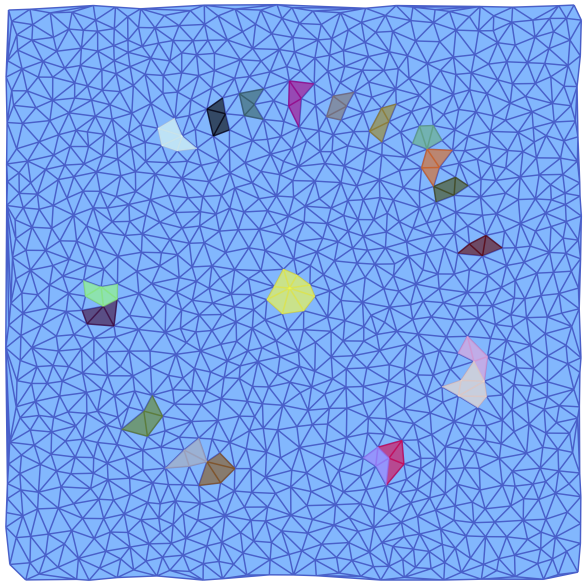


Figure 5.9: Large multivectors in the Delaunay multivector field

Notice that we can also show the individual multivectors in more detail. For the above example, we can plot all multivectors of the multivector field `mvf2` which consist of at least 10 cells using the commands

```
mv_indices = findall(x -> (length(x)>9), mvf2)
large_mv = mvf2[mv_indices]

fname3 = "cubicalcircles3.pdf"
plot_planar_simplicial_morse(lc2, coords2N, fname3, large_mv, pv=true)
```

Note that in this example, there are only 20 large multivectors.

5.7 Analysis of a Spatial System

It is also possible to analyze simple three-dimensional ordinary differential equations in ConleyDynamics.jl. To provide one such example, consider the system

$$\begin{aligned}
\dot{x}_1 &= (\lambda-1)x_1 - \tfrac{3\lambda}{2\pi}\left((x_1^3 - x_1^2x_3 + x_2^2x_3 + 2x_1\left(x_2^2+x_3^2\right)\right)\\
\dot{x}_2 &= (\lambda-4)x_2 - \tfrac{3\lambda}{2\pi}\,x_2\left(2x_1^2 + x_2^2 + 2x_1x_3 + 2x_3^2\right)\\
\dot{x}_3 &= (\lambda-9)x_3 + \tfrac{\lambda}{2\pi}\left(x_1\left(x_1^2-3x_2^2\right) - 3x_3\left(2x_1^2+2x_2^2+x_3^2\right)\right)
\end{aligned}$$

This system arises in the study of the so-called Allen-Cahn equation, which is the parabolic partial differential equation given by

$$u_t = \Delta u + \lambda\left(u-u^3\right) \quad \text{in} \quad \Omega=(0,\pi) \quad \text{with} \quad u=0 \quad \text{on} \quad \partial\Omega\,.$$

This partial differential equation can be interpreted as an infinite-dimensional system of ordinary differential equations, see for example [SW24, Section 6.1]. For this, one has to expand the unknown function $u(t,\cdot)$ as a generalized Fourier series with respect to the basis functions

$$\varphi_k(x) = \sqrt{\frac{2}{\pi}}\,\sin(k\pi x) \quad \text{for} \quad k \in \mathbb{N}\,.$$

If one truncates this series representation after three terms, and projects the right-hand side of the partial differential equation onto the linear space spanned by the first three basis functions, then the three coefficients of the approximating sum satisfy the above three-dimensional ordinary differential equation. Thus, this system provides a model for the dynamics of the partial differential equation, at least for sufficiently small values of the parameter λ. It can be implemented in Julia using the following commands:

```
function allencahn3d(x::Vector{Float64})
    #
    # Allen-Cahn projection
    #
    lambda = 3.0 * pi
    c      = lambda / pi
    x1, x2, x3 = x
    y1 = (lambda-1)*x1 - 1.5*c * (x1*x1*x1-x1*x1*x3+x2*x2*x3+2*x1*(x2*x2+x3*x3))
    y2 = (lambda-4)*x2 - 1.5*c * x2 * (2*x1*x1+x2*x2+2*x1*x3+2*x3*x3)
    y3 = (lambda-9)*x3 + 0.5*c * (x1*(x1*x1-3*x2*x2)-3*x3*(2*x1*x1+2*x2*x2+x3*x3))
    return [y1, y2, y3]
end
```

Notice that for our example we use the parameter value $\lambda = 3\pi$. In this particular case, one can show numerically that the system has seven equilibrium solutions. These are approximately given as follows:

- Two equilibria $\pm(1.45165,\ 0,\ 0.24396)$ of index 0.
- Two equilibria $\pm(0,\ 1.09796,\ 0)$ of index 1.
- Two equilibria $\pm(0,\ 0,\ 0.307238)$ of index 2.
- One equilibrium $(0,\ 0,\ 0)$ of index 3.

In order to find the associated Morse decomposition, one can use the commands

```
N = 25
bmax = [1.8, 1.5, 1.0]
lc, coordsI = create_cubical_box(N,N,N);
coordsN = convert_spatial_coordinates(coordsI, -bmax, bmax);
mvf = create_spatial_mvf(lc, coordsN, allencahn3d);
```

These commands create a cubical box of size $25 \times 25 \times 25$ which covers the region $[-1.8, 1.8] \times [-1.5, 1.5] \times [-1.0, 1.0]$. In addition, we construct a multivector field `mvf` which encapsulates the possible dynamics of the system. After these preparations, the Morse decomposition can be computed via

```
morsedecomp = morse_sets(lc, mvf);
morseinterval = morse_interval(lc, mvf, morsedecomp);
lci, mvfi = restrict_dynamics(lc, mvf, morseinterval);
cmi = connection_matrix(lci, mvfi);
```

While the first command finds the actual Morse decomposition, the second one restricts the Lefschetz complex and the multivector field to the smallest isolated invariant set which contains all Morse sets and connecting orbits between them. The last command finds the connection matrix.

To see whether the above commands did indeed find the correct dynamical behavior, we first inspect the computed Conley indices of the Morse sets:

```
julia> cmi.conley
7-element Vector{Vector{Int64}}:
 [1, 0, 0, 0]
 [1, 0, 0, 0]
 [0, 1, 0, 0]
 [0, 1, 0, 0]
 [0, 0, 1, 0]
 [0, 0, 1, 0]
 [0, 0, 0, 1]
```

Clearly, these are the correct indices based on our numerical information concerning the stationary states of the system. The connection matrix is given by:

```
julia> full_from_sparse(cmi.matrix)
7×7 Matrix{Int64}:
 0  0  1  1  0  0  0
 0  0  1  1  0  0  0
 0  0  0  0  1  1  0
 0  0  0  0  1  1  0
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1
 0  0  0  0  0  0  0
```

Thus, there are a total of ten connecting orbits that are induced through algebraic topology. The index 3 equilibrium at the origin has connections to each of the index 2 solutions, which lie above and below the origin in the direction of the x_3-axis. Each of the latter two stationary states has connections to both index 1 equilibria. Finally, each of these is connected to both stable states.

The location of the computed Morse sets is illustrated in the accompanying figure, which uses x, y, and z instead of the variable names x_1, x_2, and x_3, respectively. Notice that while the stationary states of index 0, 2, and 3 are all well-localized, this cannot be said about the two equilibria of index 1. The computed enclosures for the latter two are elongated cubical sets which are shown along the upper left and lower right of the figure. This overestimation is a result of the use of a strict cubical grid, combined with the small discretization size `N = 25`. Nevertheless, the above simple code does reproduce the overall global dynamical behavior of the ordinary differential equation correctly.

One can also compute over Morse intervals, rather than the complete Morse decomposition. The final two images show two views of the Morse interval which corresponds to one of the index 1 equilibria, and the two stable stationary states. These computations were performed with the finer resolution `N = 51`.

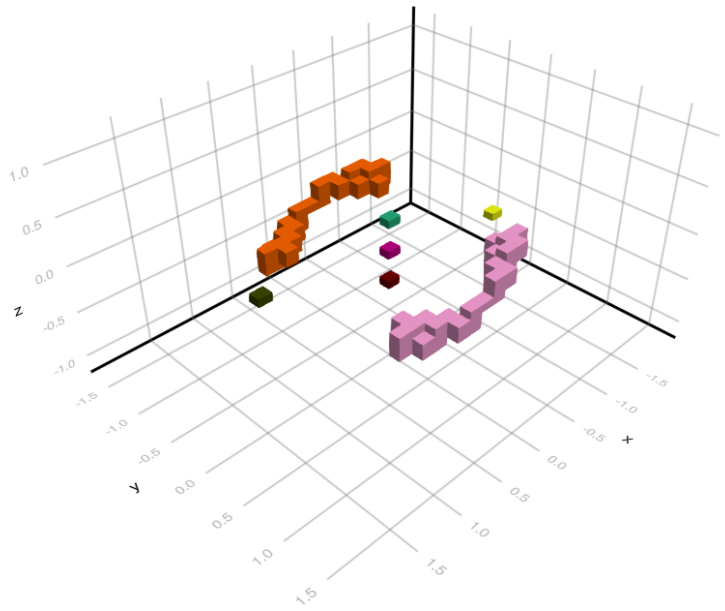


Figure 5.10: The dynamics of an Allen-Cahn model

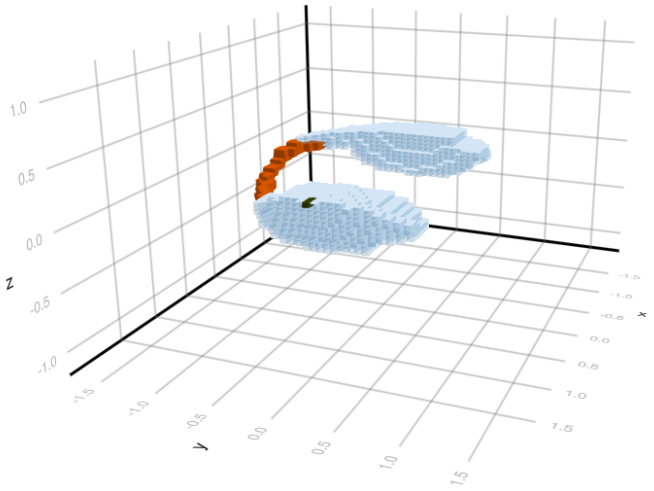


Figure 5.11: Allen-Cahn Morse interval, View 1

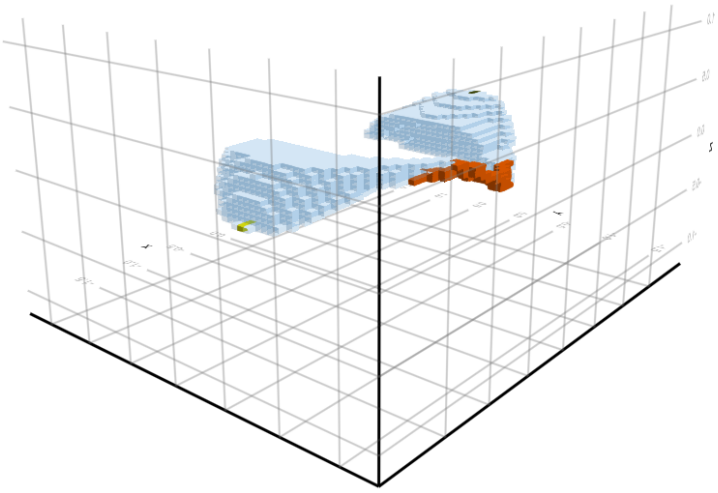


Figure 5.12: Allen-Cahn Morse interval, View 2

We would like to emphasize that there are many techniques in the literature that can be used to identify isolated invariant sets and their Conley indices. Rather than giving a detailed list, we refer to [SW14b] and the references therein. For example, in [SW14b] ideas from computational topology were used to rigorously establish candidate sets in three dimensions as an isolating block. The associated Matlab code can be found at [SW14a].

5.8 Forman's Morse Complex

The package ConleyDynamics.jl also provides some support for studying Forman gradient vector fields directly, using the notions introduced in [For98b]. In this paper, Forman used the concept of a combinatorial flow Φ to study the forward orbits generated by the Forman vector field. The map Φ is a chain map which is chain homotopic to the identity, and it therefore mimics the concept of a time-1-map associated with a dynamical system. Forman shows that upon iteration this map stabilizes as a map Φ^∞, which encodes the connecting orbits in the associated combinatorial dynamical system. This stabilized combinatorial flow can then be used to find the connection matrix in this case. For more details, we refer to [MW25, Chapter 8]. The combinatorial flow and its stabilized version can be computed using the following two functions.

- `forman_comb_flow` expects a Lefschetz complex and a Forman vector field as input arguments. It returns matrix representations of Forman's combinatorial flow Φ, as well as of the associated chain homotopy Γ between the flow chain map and the identity.
- `forman_stab_flow` also requires both a Lefschetz complex and a Forman vector field as arguments. It then returns matrix representations of Forman's stabilized combinatorial flow Φ^∞, and of the associated chain homotopy Γ^∞. This time, there is a third return argument `stabilized`. This boolean flag indicates whether or not the combinatorial flow stabilized. If it did not, then the returned chain map is the last computed iterate, together with the corresponding chain homotopy. There are two possible reasons for failing stabilization: Either the underlying Forman vector field is not gradient (note that this is not checked!), or the maximal number of iterations has been reached. In the latter case, one just has to pass the optional paramter `maxit` with a larger number of allowed iterations.
- `forman_gpaths` can be used to find gradient paths in a Forman gradient vector field. There are two methods for this function, which are accessible via multiple dispatch as follows.

  - The call `forman_gpaths(lc, fvf, x)` determines all maximal gradient paths of the Forman gradient field `fvf` on the Lefschetz complex `lc` starting at `x`. These are solution paths which consist exclusively of Forman vectors, i.e., they contain an even number of cells whose dimensions alternate between $\dim x$ and $1 + \dim x$. Every cell of dimension $\dim x$ is an arrow source which is followed by its arrow target, while every cell of dimension $1 + \dim x$ is an arrow target which is succeeded by a cell in its boundary, and which in turn is the source of a different arrow, as long as such a cell exists.
  - The call `forman_gpaths(lc, fvf, x, y)` computes all Forman gradient paths between the cells `x` and `y`. Nontrivial paths are only returned if $|\dim x - \dim y| \leq 1$. More precisely, the following paths are returned:
    * If $\dim x = \dim y - 1$, the function returns all solution paths between `x` and `y` which consist entirely of Forman arrows. All the sources have the dimension of `x`, and the targets the dimension of `y`.
    * For $\dim x = \dim y$ the function finds all solution paths `p` between `x` and `y` for which `p[1:end-1]` is a Forman gradient path in the above sense, and `p[end]` lies in the boundary of `p[end-1]` and is different from the cell `p[end-2]`. For this, the length of the path has to be at least 3. If on the other hand one has $x = y$, then only a 1-element path is returned.
    * Finally, if $\dim x = \dim y + 1$, then the function returns all solution paths `p` between `x` and `y` for which `p[2:end]` is a Forman gradient path in the sense of the second item, and `p[2]` lies in the boundary of `p[1]`.

    In all other cases an empty collection is returned.

- `forman_path_weight` expects the arguments `lc:LefschetzComplex` and `path::Cell` and computes the weight of the Forman gradient path `path` in the Lefschetz complex `lc`. It is expected that the dimensions of the first and the last cell in the path differ by at most 1. In case they have equal dimension, and the path has length larger than 1, the first cell has to be an arrow source. This function also has a second method associated with its name, in which the second argument is of type `CellSubsets`, i.e., it contains a collection of Forman gradient paths. In the case, the function returns the sum of all weights of these paths.

For analyzing or applying Forman's combinatorial flow, one needs to work with sparse vector representations of chains. These are elements of the chain groups of the underlying Lefschetz complex, which are then represented as sparse matrices consisting of exactly one column. In this form, they can be multiplied by the matrix of the combinatorial flow, which in turn determines the image of the chain under the flow. In general, these vectors are extremely sparse, and only contain a handfull of nonzero entries. ConleyDynamics.jl therefore provides the following two functions for the creation and analysis of sparse chain vectors:

- `chain_vector` is a function that simplifies the creation of a sparse vector representation of a chain. The chain can be specified by simply listing the cells in the support of the chain. If no coefficients are specified, then the chain is just the sum of these cells, each with coefficient 1. The function has several associated methods: The coefficients of the chain can be omitted or specified, and the cells can be listed in index or label form. More details can be found in the description of each method.

- `chain_support` extracts the support of a chain given as a sparse vector. In its simplest form, the function returns a `Vector{String}` which contains the labels of all the cells which have nonzero coefficients in the chain. If one passes the optional parameter `coeff=true`, then the function returns two arguments: In addition to the vector of labels as above, it also returns the vector of associated coefficients.

5.9 References

See the full bibliography for a complete list of references cited throughout this documentation. This section cites the following references:

[BKMW20] B. Batko, T. Kaczynski, M. Mrozek and T. Wanner. Linking combinatorial and classical dynamics: Conley index and Morse decompositions. Foundations of Computational Mathematics **20**, 967–1012 (2020).

[Con78] C. Conley. Isolated Invariant Sets and the Morse Index (American Mathematical Society, Providence, R.I., 1978).

[DLMS24] T. K. Dey, M. Lipiński, M. Mrozek and R. Slechta. Computing connection matrices via persistence-like reductions. SIAM Journal on Applied Dynamical Systems **23**, 81–97 (2024).

[For98b] R. Forman. Morse theory for cell complexes. Advances in Mathematics **134**, 90–145 (1998).

[Fra89] R. Franzosa. The connection matrix theory for Morse decompositions. Transactions of the American Mathematical Society **311**, 561–592 (1989).

[HMS21] S. Harker, K. Mischaikow and K. Spendlove. A computational framework for connection matrix theory. Journal of Applied and Computational Topology **5**, 459–529 (2021).

[LKMW23] M. Lipinski, J. Kubica, M. Mrozek and T. Wanner. Conley-Morse-Forman theory for generalized combinatorial multivector fields on finite topological spaces. Journal of Applied and Computational Topology **7**, 139–184 (2023).

[MSTW22] M. Mrozek, R. Srzednicki, J. Thorpe and T. Wanner. Combinatorial vs. classical dynamics: Recurrence. Communications in Nonlinear Science and Numerical Simulation **108**, Paper No. 106226, 30 pages (2022).

[MW25] M. Mrozek and T. Wanner. Connection Matrices in Combinatorial Topological Dynamics. SpringerBriefs in Mathematics (Springer-Verlag, Cham, 2025).

[SW24] E. Sander and T. Wanner. Theory and Numerics of Partial Differential Equations (SIAM, Philadelphia, 2024). In preparation, 1007 pages.

[SW14a] T. Stephens and T. Wanner. Isolating block validation in Matlab, https://github.com/almost6heads/isoblockval (2014).

[SW14b] T. Stephens and T. Wanner. Rigorous validation of isolating blocks for flows and their Conley indices. SIAM Journal on Applied Dynamical Systems **13**, 1847–1878 (2014).

Chapter 6

Examples

In order to illustrate the basic functionality of ConleyDynamics.jl, this section collects a number of examples. Many of these are taken from the paper [BKMW20] and the book [MW25], and they consider both Forman vector fields and general multivector fields on a variety of Lefschetz complexes. Each example has its own associated function, so that users can quickly create examples on their own by taking the respective source files as templates.

6.1 A One-Dimensional Forman Field

Our first example is taken from [BKMW20, Figure 1], and it is a Forman vector field on a one-dimensional simplicial complex as shown in the figure.

The simplicial complex and Forman vector field can be created using the function `example_forman1d`:

`ConleyDynamics.example_forman1d` – Method.

```
lc, mvf, coords = example_forman1d()
```

Create the simplicial complex and multivector field for the example from Figure 1 in the FoCM 2020 paper by Batko, Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices. The Lefschetz complex is defined over the finite field GF(2).

Examples

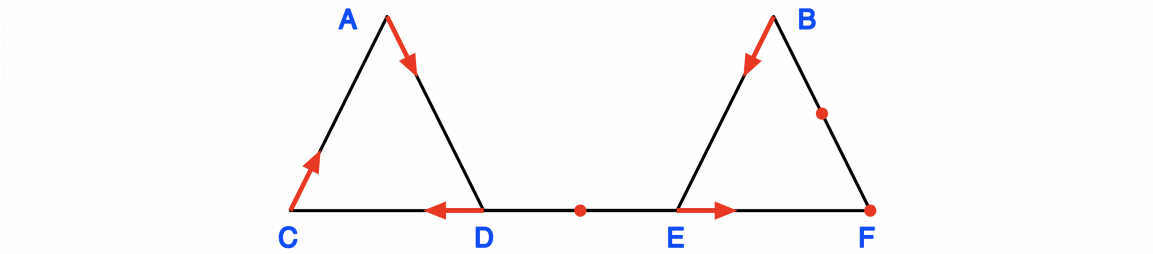


Figure 6.1: A one-dimensional simplicial complex flow

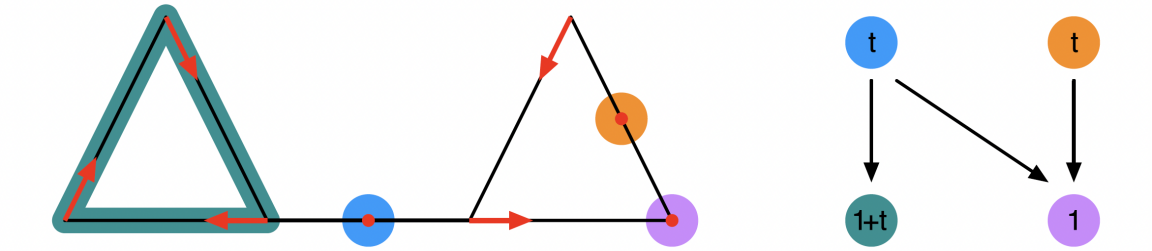


Figure 6.2: Morse decomposition of the one-dimensional example

```
julia> lc, mvf = example_forman1d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm)
   ┆  A AD  F BF DE
---┆--------------
 A ┆  0  0  0  0  1
AD ┆  0  0  0  0  0
 F ┆  0  0  0  0  1
BF ┆  0  0  0  0  0
DE ┆  0  0  0  0  0

julia> sparse_show(cm.matrix)
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 0

julia> println(cm.labels)
["A", "AD", "F", "BF", "DE"]
```

source

The commands from the docstring show that the connection matrix has five rows and columns. The last three of these correspond to the critical cells F, BF, and DE, while the first two correspond to the two generators of the homological Conley index of the periodic orbit, given by A and AD.

The full Morse decomposition of this combinatorial dynamical system is depicted in the second figure, and all four Morse set are indicated in the simplicial complex by different colors. They are also listed, together with their Conley indices, in the following Julia output:

```
julia> cm.morse
4-element Vector{Vector{String}}:
 ["A", "C", "D", "AC", "AD", "CD"]
 ["F"]
 ["BF"]
 ["DE"]

julia> cm.conley
4-element Vector{Vector{Int64}}:
```

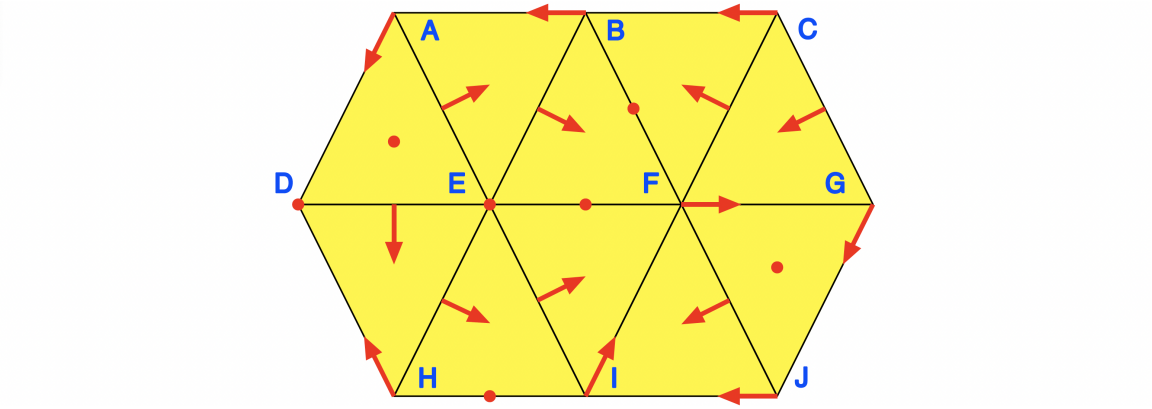


Figure 6.3: A planar simplicial complex flow

```
[1, 1]
[1, 0]
[0, 1]
[0, 1]
```

Notice that only two heteroclinic orbits are reflected in the connection matrix. These are the connections between the unstable cell DE and both the equilibrium F and the periodic orbit. In contrast, the two heteroclinics between the index one cell BF and F cancel algebraically.

6.2 A Planar Forman Vector Field

Our second example was originally discussed in the context of [BKMW20, Figure 3], and it consists of a Forman vector field on a topological disk, as shown in the associated figure.

The disk is represented as a simplicial complex with 10 vertices, 19 edges, and 10 triangles. The Forman vector field has 7 critical cells, and 16 arrows. Both the simplicial complex and the Forman vector field can be defined using the function `example_forman2d`:

`ConleyDynamics.example_forman2d` – Method.

```
lc, mvf, coords = example_forman2d()
```

Create the simplicial complex and multivector field for the example from Figure 3 in the FoCM 2020 paper by Batko, Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` over the finite field GF(2) and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices.

Examples

```
julia> lc, mvf = example_forman2d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm)
```

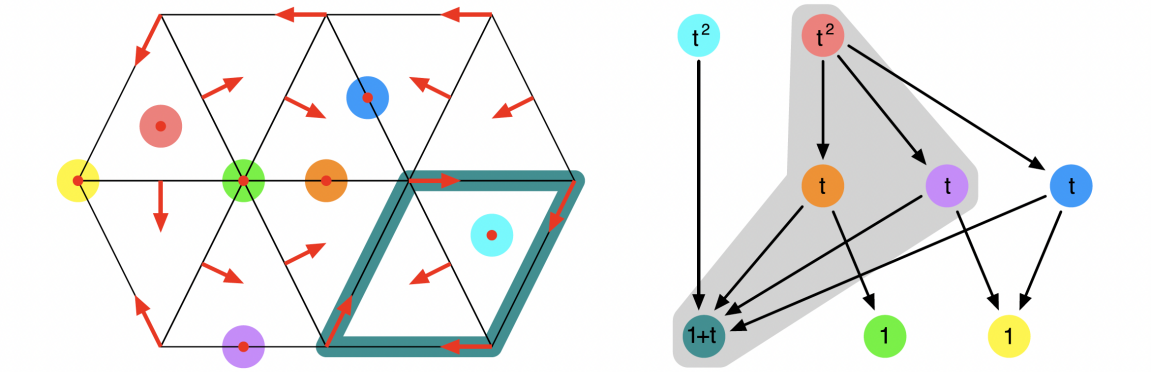


Figure 6.4: Morse decomposition of the planar flow

```
   ┆   D   E   F  GJ  BF  EF  HI ADE FGJ
───┆──────────────────────────────────
  D┆   0   0   0   0   1   0   1   0   0
  E┆   0   0   0   0   0   1   0   0   0
  F┆   0   0   0   0   1   1   1   0   0
 GJ┆   0   0   0   0   0   0   0   0   1
 BF┆   0   0   0   0   0   0   0   1   0
 EF┆   0   0   0   0   0   0   0   0   0
 HI┆   0   0   0   0   0   0   0   1   0
ADE┆   0   0   0   0   0   0   0   0   0
FGJ┆   0   0   0   0   0   0   0   0   0

julia> sparse_show(cm.matrix)
 0 0 0 0 1 0 1 0 0
 0 0 0 0 0 1 0 0 0
 0 0 0 0 1 1 1 0 0
 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm.labels)
["D", "E", "F", "GJ", "BF", "EF", "HI", "ADE", "FGJ"]
```

source

The Morse decomposition of this example is shown in the second figure. Its eight Morse sets and associated Conley indices are given by

```
julia> cm.morse
8-element Vector{Vector{String}}:
 ["D"]
 ["E"]
 ["F", "G", "I", "J", "FG", "FI", "GJ", "IJ"]
 ["BF"]
 ["EF"]
```

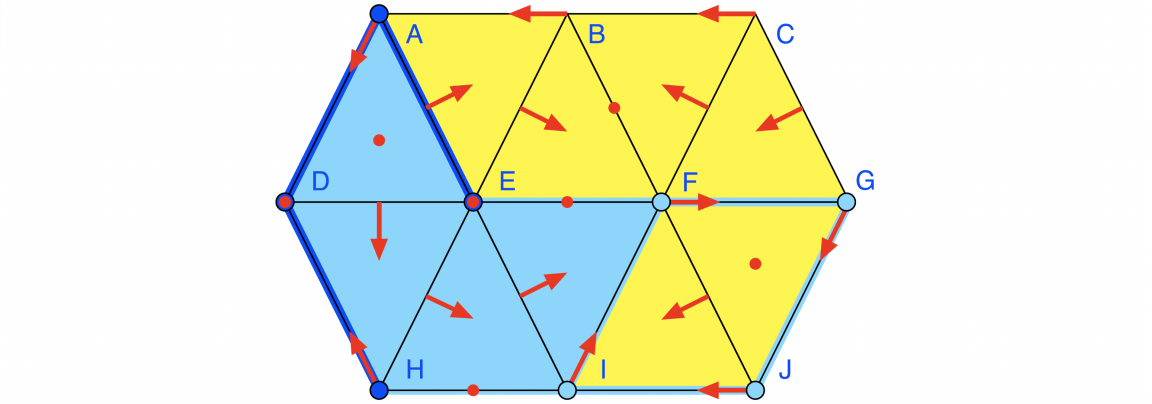


Figure 6.5: A nontrivial isolated invariant set

```
 ["HI"]
 ["ADE"]
 ["FGJ"]

julia> cm.conley
8-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [1, 0, 0]
 [1, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 0, 1]
 [0, 0, 1]
```

We would like to point out that the collection of Morse sets does not in general encompass all possible isolated invariant sets for the combinatorial dynamical system. Consider for example the set S shown in light blue in the next figure.

Its mouth is depicted in dark blue, and it is clearly closed. In addition, the set S decomposes into arrows and critical cells, and one can show that it is invariant. Thus, it is in fact an isolated invariant set for this Forman vector field. Note also that this cell does not correspond to an interval in the Conley-Morse graph either, and this is indicated via gray shading in the above image of the graph. The set S, together with its closure and its mouth, can be generated using the following commands:

```
S = ["ADE","DEH","EFI","EHI",
     "DE","EF","EH","EI","FG","FI","GJ","HI","IJ",
     "F","G","I","J"]
clS, moS = lefschetz_clomo_pair(lc,S)
```

Then the Conley index of S is given by

```
julia> conley_index(lc,S)
3-element Vector{Int64}:
 0
```

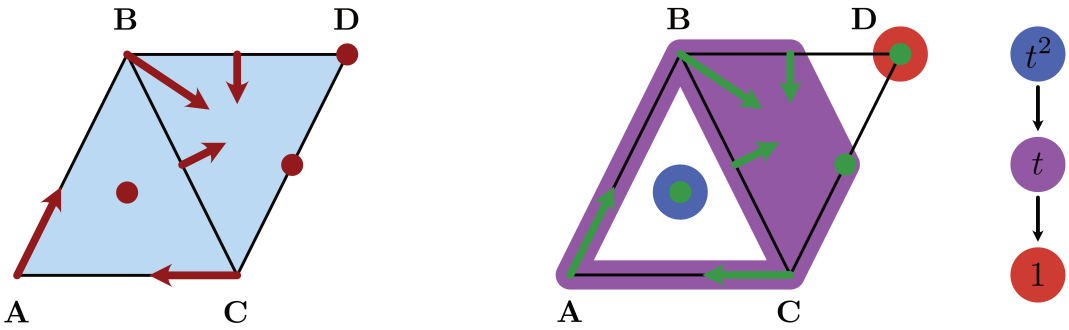


Figure 6.6: The logo multivector field

```
 1
 0

julia> relative_homology(lc,clS,moS)
3-element Vector{Int64}:
 0
 1
 0
```

Notice that this is the same as the relative homology of the pair $(\mathrm{cl}\, S, \mathrm{mo}\, S)$, as expected.

6.3 The Multivector Field from the Logo

This example is taken from [MW25, Figure 2.1], and it is visualized in the accompanying figure.

Clearly, this is the multivector field from the ConleyDynamics.jl logo. Since it was already discussed in detail in the Tutorial, we only show how the underlying simplicial complex and the multivector field can be created quickly using the function `example_julia_logo`:

`ConleyDynamics.example_julia_logo` – Method.

```
lc, mvf = example_julia_logo()
```

Create the simplicial complex and multivector field for the example from Figure 1 in the connection matrix paper by Mrozek & Wanner.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_julia_logo();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0
 0 0 1
 0 0 0
```

```
julia> print(cm.labels)
["D", "AC", "ABC"]
```

source

The Morse sets and associated Conley indices can be accessed using the commands:

```
julia> cm.morse
3-element Vector{Vector{String}}:
 ["D"]
 ["A", "B", "C", "AB", "AC", "BC", "BD", "CD", "BCD"]
 ["ABC"]

julia> cm.conley
3-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 1, 0]
 [0, 0, 1]
```

Notice that in this example, every simplex of the underlying simplicial complex is contained in one of the Morse sets.

6.4 Critical Flow on a Simplex

The next example considers the arguably simplest situation of a Forman vector field on a simplicial complex. The simplicial complex X is given by a single simplex of dimension n, together with all its faces, while the Forman vector field on X contains only singletons. In other words, every simplex in the complex is a critical cell. Thus, this combinatorial dynamical system has one equilibrium of index n, and $n+1$ stable equilibria. In addition, there are $2^{n+1}-n-3$ additional stationary states whose indices lie strictly between 0 and n, as well as a wealth of algebraically induced heteroclinic orbits. All of these can be found by using the connection matrix for the problem, as outlined in the following description for the function `example_critical_simplex`.

`ConleyDynamics.example_critical_simplex` – Method.

```
lc, mvf = example_critical_simplex(dim)
```

Create a simplicial complex of dimension `dim` as well as a multivector field on it in which every cell is critical.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_critical_simplex(2);

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0 1 1 0 0
 0 0 0 1 0 1 0
```

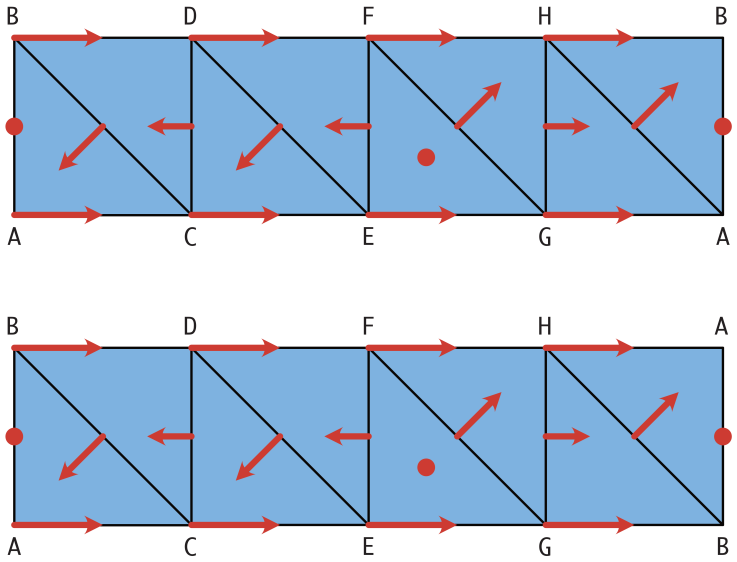


Figure 6.7: Combinatorial flow on cylinder and Moebius strip

```
0 0 0 0 1 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 0 0 0 0 0 0

julia> print(cm.labels)
["A", "B", "C", "AB", "AC", "BC", "ABC"]
```

source

6.5 Flow on a Cylinder and a Moebius Strip

The next example considers again Forman vector fields, but this time on a cylinder and on a Moebius strip. The underlying simplicial complexes are given by a horizontal strip of eight triangles, whose left and right vertical edges are identified. For the first complex `lc1` these edges are identified without twist, while for the complex `lc2` they are twisted. See also the labels in the figure.

Both complexes consist of eight vertices, sixteen edges, and eight triangles. The two complexes and Forman vector fields can be generated using the function `example_critical_simplex`, whose usage can be described as follows.

`ConleyDynamics.example_moebius` – Function.

```
lc1, mvf1, lc2, mvf2 = example_moebius(p)
```

Create two simplicial complexes for a cylinder and Moebius strip, respectively, together with associated multivector fields on them.

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector fields `mvf1` and `mvf2`. Both complexes are over a field with characteristic p. Positive prime characteristic uses the finite field GF(p), while zero characteristic gives the rationals.

The multivector field is the same, and it has one critical cell each in dimension 1 and 2 in the interior of the strip. The boundary consists of two periodic orbits for `lc1` and `mvf1`, and of one periodic orbit in the Moebius case `lc2` and `mvf2`. The latter case leads to different connection matrices for the fields GF(2) and GF(7), for example.

Examples

```
julia> lc1, mvf1, lc2, mvf2 = example_moebius(0);

julia> lc2p2 = lefschetz_gfp_conversion(lc2,2);

julia> lc2p7 = lefschetz_gfp_conversion(lc2,7);

julia> cmp2 = connection_matrix(lc2p2, mvf2);

julia> cmp7 = connection_matrix(lc2p7, mvf2);

julia> sparse_show(cmp2.matrix)
 0 0 0 0
 0 0 0 1
 0 0 0 0
 0 0 0 0

julia> sparse_show(cmp7.matrix)
 0 0 0 0
 0 0 0 1
 0 0 0 2
 0 0 0 0
```

source

Note that for the combinatorial flow on the Moebius strip `lc2` the choice of field characteristic p leads to potentially different connection matrices. While for characteristic $p = 2$ the connection matrix has only one nontrivial entry, it has two for $p = 7$.

We only briefly include some sample computations for the latter case. One can create the complexes, Forman vector fields, and associated connection matrices for $p = 7$ using the following commands:

```
lc1, mvf1, lc2, mvf2 = example_moebius(7)
cm1 = connection_matrix(lc1,mvf1)
cm2 = connection_matrix(lc2,mvf2)
```

For the first example, the combinatorial flow on the cylinder has four Morse sets. Two critical equilibria of indices 1 and 2, as well as two periodic orbits. This can be shown as follows:

```
julia> cm1.morse
4-element Vector{Vector{String}}:
 ["A", "C", "E", "G", "AC", "AG", "CE", "EG"]
 ["B", "D", "F", "H", "BD", "BH", "DF", "FH"]
 ["AB"]
```

```
 ["EFG"]

julia> cm1.conley
4-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [1, 1, 0]
 [0, 1, 0]
 [0, 0, 1]

julia> sparse_show(cm1.matrix)
[0   0   0   0   6   0]
[0   0   0   0   0   1]
[0   0   0   0   1   0]
[0   0   0   0   0   6]
[0   0   0   0   0   0]
[0   0   0   0   0   0]

julia> print(cm1.labels)
["A", "AG", "B", "BH", "AB", "EFG"]
```

In fact, the connection matrix implies the existence of connecting orbits from both the index 2 and the index 1 equilibrium to the two periodic orbits. The connections between the stationary states cannot be detected algebraically.

For the second example, the combinatorial flow on the Moebius strip, one only obtains three Morse sets. This time, there is only one periodic orbit which loops around both the top and bottom edges in the figure. This is confirmed by the commands

```
julia> cm2.morse
3-element Vector{Vector{String}}:
 ["A", "B", "C", "D", "E", "F", "G", "H", "AC", "AH", "BD", "BG", "CE", "DF", "EG", "FH"]
 ["AB"]
 ["EFG"]

julia> cm2.conley
3-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [0, 1, 0]
 [0, 0, 1]

julia> sparse_show(cm2.matrix)
[0   0   0   0]
[0   0   0   1]
[0   0   0   2]
[0   0   0   0]

julia> print(cm2.labels)
["A", "BG", "AB", "EFG"]
```

In this case, the connection matrix is able to identify the connecting orbits between the index 2 stationary state and both the periodic orbit and the index 1 equilibrium. The latter one is not recognized over the field $GF(2)$.

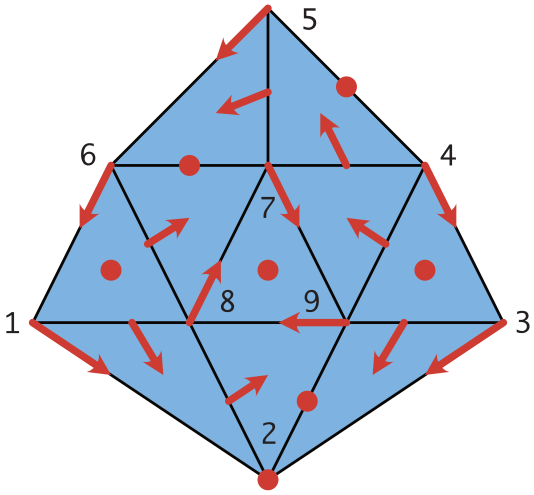


Figure 6.8: An example with nonunique connection matrices

6.6 Nonunique Connection Matrices

Our next example is concerned with another Forman vector field, but this time on a larger simplicial complex, as shown in the figure.

The simplicial complex is topologically a disk, and it consists of 9 vertices, 18 edges, and 10 triangles. The Forman vector field has 1 critical vertex, 3 critical edges, and 3 critical triangles, as well as 15 Forman arrows. The following example shows that for this combinatorial dynamical system, there are two fundamentally different connection matrices.

`ConleyDynamics.example_nonunique` – Method.

```
lc1, lc2, mvf, coords1, coords2 = example_nonunique()
```

Create two representations of a simplicial complex and one multivector field which illustrates nonunique connection matrices.

The two complexes `lc1` and `lc2` represent the same simplicial complex over GF(2), but differ in the ordering of the labels.

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector field `mvf`. If desired for plotting, the fourth and fifth return values `coords1` and `coords2` give vectors of coordinates for the vertices of the two complexes.

Examples

```
julia> lc1, lc2, mvf = example_nonunique();

julia> cm1 = connection_matrix(lc1, mvf);

julia> cm2 = connection_matrix(lc2, mvf);

julia> sparse_show(cm1.matrix)
 0 0 0 1 0 1 0 0 0
```

```
 0 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 0 1 1
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm1.labels)
["2", "7", "79", "29", "45", "67", "168", "349", "789"]

julia> sparse_show(cm2.matrix)
 0 0 0 1 0 1 0 0 0
 0 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 1 0 1
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm2.labels)
["2", "8", "78", "29", "45", "67", "168", "349", "789"]
```

source

As mentioned in the docstring for the function `example_nonunique`, the two Lefschetz complexes `lc1` and `lc2` both represent the above simplicial complex. However, they differ in the ordering of the vertex labels. This can be seen from the commands

```
julia> print(lc1.labels[1:9])
["1", "2", "3", "4", "5", "6", "7", "8", "9"]
julia> print(lc2.labels[1:9])
["1", "2", "3", "4", "5", "6", "8", "9", "7"]
```

In other words, `lc1` and `lc2` are different representations of the same complex. Nevertheless, computing the connection matrices as in the example gives two distinct connection matrices. This is purely a consequence of the different ordering of the rows and columns in the boundary matrix.

To shed further light on this issue, notice that the triangle at the center of the complex forms an attracting periodic orbit, whose Conley index has Betti numbers 1 in dimensions 0 and 1. One can break this periodic orbit by removing one of its three arrows, and replacing it with two critical cells of dimensions 0 and 1. The next image shows two different ways of doing this.

In the image on the left, the vector `["7", "79"]` is removed, while the one on the right breaks up `["8", "78"]`. The corresponding modified Forman vector fields, and their connection matrices, can be created as follows:

```
mvf1 = deepcopy(mvf);
mvf2 = deepcopy(mvf);
deleteat!(mvf1,6);
deleteat!(mvf2,8);
```

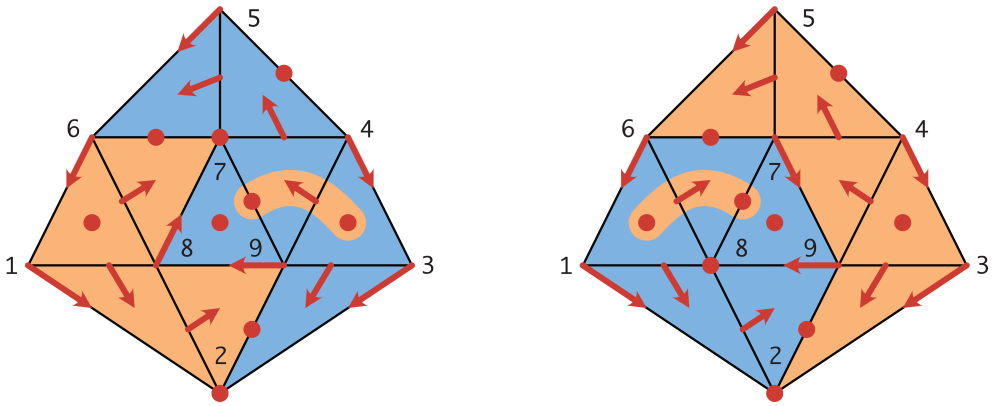


Figure 6.9: Forcing different connection matrices

```
cm1mod = connection_matrix(lc1, mvf1);
cm2mod = connection_matrix(lc2, mvf2);
```

Both of the new Forman vector fields are gradient vector fields, and in view of a result in [MW25], their connection matrices are therefore uniquely determined. The connection matrix for the vector field `mvf1` is of the form

```
julia> sparse_show(cm1mod.matrix)
[0   0   1   0   1   0   0   0   0]
[0   0   1   0   1   0   0   0   0]
[0   0   0   0   0   0   1   1   0]
[0   0   0   0   0   0   0   1   0]
[0   0   0   0   0   0   1   1   0]
[0   0   0   0   0   0   0   1   1]
[0   0   0   0   0   0   0   0   0]
[0   0   0   0   0   0   0   0   0]
[0   0   0   0   0   0   0   0   0]

julia> print(cm1mod.labels)
["2", "7", "29", "45", "67", "79", "168", "349", "789"]
```

Notice that this matrix shows that there is a connection from the triangle 349 to the edge 79, but there are no connections from the triangle 168 to the critical edge on the center triangle. In fact, up to reordering the columns and rows, this connection matrix is the same as `cm1` in the example.

Similarly, the connection matrix for the second modified Forman vector field `mvf2` is uniquely determined, and it is given by

```
julia> sparse_show(cm2mod.matrix)
[0   0   1   0   1   0   0   0   0]
[0   0   1   0   1   0   0   0   0]
[0   0   0   0   0   0   1   1   0]
[0   0   0   0   0   0   0   1   0]
[0   0   0   0   0   0   1   1   0]
[0   0   0   0   0   0   1   0   1]
```

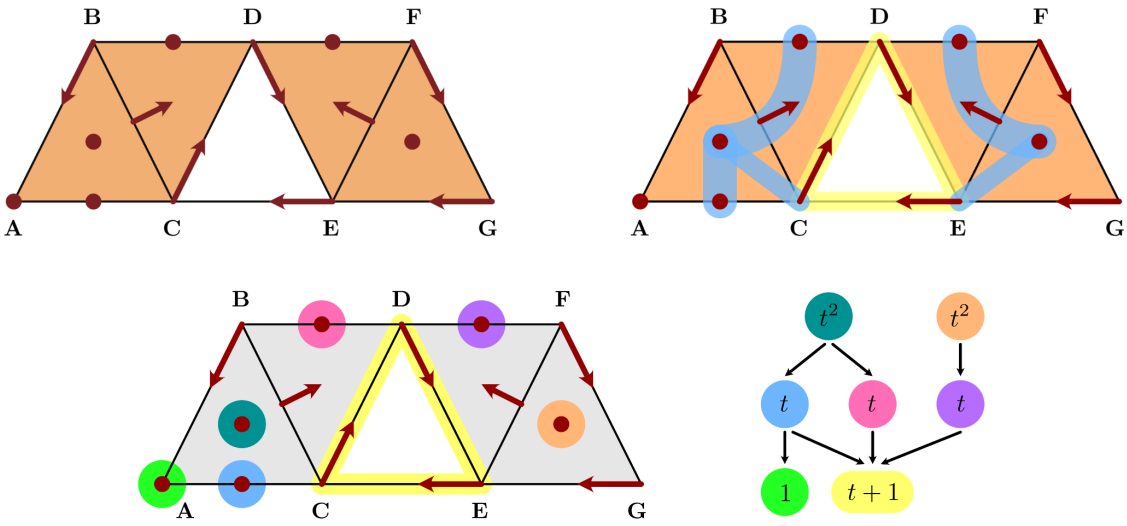


Figure 6.10: An example with three connection matrices

```
[0   0   0   0   0   0   0   0   0]
[0   0   0   0   0   0   0   0   0]
[0   0   0   0   0   0   0   0   0]

julia> print(cm2mod.labels)
["2", "8", "29", "45", "67", "78", "168", "349", "789"]
```

Now there is a connection from the triangle 168 to the edge 78, but there are no connections from the triangle 349 to the critical edge on the center triangle. This time, up to a permutation of the columns and the rows, this connection matrix is the same as `cm2` in the example.

6.7 Forcing Three Connection Matrices

The next example is taken from [MW25, Figure 2.2], and it revolves around the combinatorial vector field on a simplicial complex shown in the top left part of the figure.

This combinatorial vector field is again of Forman type. It has a periodic orbit, which is shown in yellow in the top right part of the figure. In addition to three index 1 equilibria, there are two of index 2. The top right part shows that from these two index 2 cells there are a combined total of five connecting orbits to the index 1 cells and to the periodic orbit. Its Morse decomposition is shown in the lower part of the figure. While the Morse sets are indicated by different colors, the Conley-Morse graph is shown on the lower right.

As the following docstring for `example_three_cm` demonstrates, the connection matrix, which this time is computed over the rationals $\mathbb{Q}$, only identifies three of the five connecting orbits between index 2 invariant sets and index 1 sets.

`ConleyDynamics.example_three_cm` – Method.

```
lc, mvf, coords = example_three_cm(mvftype)
```

Create the simplicial complex and multivector field for the example from Figure 2 in the connection matrix paper by Mrozek & Wanner.

Depending on the value of `mvftype`, return the periodic orbit (0=default) or one of the three gradient (1,2,3) examples.

The function returns the Lefschetz complex `lc` over the rational field and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices.

Examples

```
julia> lc, mvf = example_three_cm(0);

julia> cm = connection_matrix(lc, mvf);

julia> print(cm.labels)
["A", "C", "CE", "AC", "BD", "DF", "ABC", "EFG"]

julia> full_from_sparse(cm.matrix)
8×8 Matrix{Rational{Int64}}:
 0  0  0  -1  -1  0   0  0
 0  0  0   1   1  0   0  0
 0  0  0   0   0  0   0  0
 0  0  0   0   0  0  -1  0
 0  0  0   0   0  0   1  0
 0  0  0   0   0  0   0  1
 0  0  0   0   0  0   0  0
 0  0  0   0   0  0   0  0
```

source

It turns out that this combinatorial dynamical system has multiple possible connection matrices as well. In fact, it has three. In order to find them we use the same approach as in the last example, and break the periodic orbit by turning one of its arrows into two critical cells. Since there are three arrows in the periodic orbit, this can be accomplished in three different ways. They are indicated in the next figure.

The resulting Forman vector fields are all of gradient type, and therefore have a unique connection matrix. These three vector fields can be obtained via the function `example_three_cm` by specifying the integer argument as 1, 2, or 3. For the first vector field one obtains the following connection matrix:

```
julia> lc1, mvf1 = example_three_cm(1);

julia> cm1 = connection_matrix(lc1, mvf1);

julia> print(cm1.labels)
["A", "C", "AC", "BD", "CD", "DF", "ABC", "EFG"]

julia> full_from_sparse(cm1.matrix)
8×8 Matrix{Rational{Int64}}:
 0  0  -1  -1  0  0   0  0
 0  0   1   1  0  0   0  0
 0  0   0   0  0  0  -1  0
 0  0   0   0  0  0   1  0
 0  0   0   0  0  0  -1  0
 0  0   0   0  0  0   0  1
```

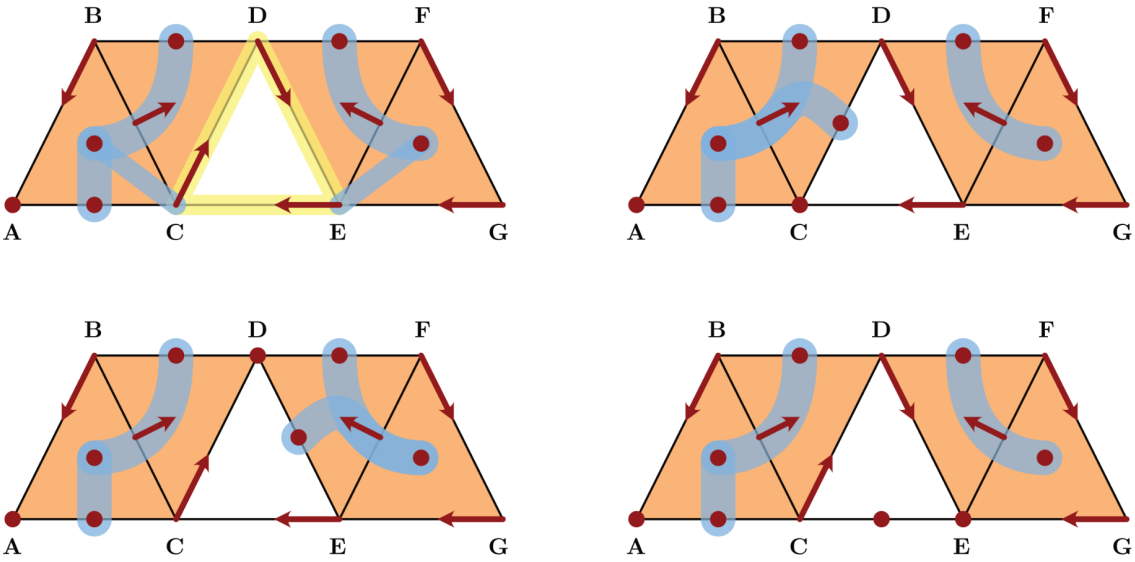


Figure 6.11: Three different ways to break up the periodic orbit

```
 0  0   0   0  0  0   0  0
 0  0   0   0  0  0   0  0
```

In contrast, the second vector field leads to:

```
julia> lc2, mvf2 = example_three_cm(2);

julia> cm2 = connection_matrix(lc2, mvf2);

julia> print(cm2.labels)
["A", "D", "AC", "BD", "DE", "DF", "ABC", "EFG"]

julia> full_from_sparse(cm2.matrix)
8×8 Matrix{Rational{Int64}}:
 0  0  -1  -1  0  0   0   0
 0  0   1   1  0  0   0   0
 0  0   0   0  0  0  -1   0
 0  0   0   0  0  0   1   0
 0  0   0   0  0  0   0  -1
 0  0   0   0  0  0   0   1
 0  0   0   0  0  0   0   0
 0  0   0   0  0  0   0   0
```

Finally, the third gradient vector field gives:

```
julia> lc3, mvf3 = example_three_cm(3);

julia> cm3 = connection_matrix(lc3, mvf3);
```

```
julia> print(cm3.labels)
["A", "E", "AC", "BD", "CE", "DF", "ABC", "EFG"]

julia> full_from_sparse(cm3.matrix)
8×8 Matrix{Rational{Int64}}:
 0  0  -1  -1  0  0   0  0
 0  0   1   1  0  0   0  0
 0  0   0   0  0  0  -1  0
 0  0   0   0  0  0   1  0
 0  0   0   0  0  0   0  0
 0  0   0   0  0  0   0  1
 0  0   0   0  0  0   0  0
 0  0   0   0  0  0   0  0
```

This is finally the connection matrix that was originally returned for the Forman vector field with periodic orbit. One could have obtained the remaining two also through cell permutations.

Notice that these three matrices combined do identify all of the above connections. It was shown in [MW25] that these matrices are different connection matrices for the original Forman vector field with periodic orbit, as long as the newly introduced index 1 and 0 equilibria are identified with the Conley index of the periodic solution. For the sake of completeness, the next figure shows the Morse decompositions for all three combinatorial gradient flows. In the Conley-Morse graphs, blue arrows correspond to the heteroclinic orbits that are identified by the associated connection matrix.

In the Conley-Morse graphs, we used the same color yellow for the two Morse sets that are generated by breaking the periodic orbit through the introduction of two critical cells. It can be seen from the images that while the actual Morse set structure stays fixed, the poset order in the Conley-Morse graphs changes from case to case.

6.8 A Lefschetz Multiflow Example

The next example is taken from [MW25, Figure 2.3], and it is a combinatorial multivector field on a true Lefschetz complex, as shown in the left panel of the associated figure.

The example is a combinatorial representation of the multiflow shown on the right, which features nonunique forward dynamics at the point labeled 5. Note that the underlying Lefschetz complex is defined as the subset of the depicted simplicial complex, where the vertices A, B, D, E, and F have been removed. This Lefschetz complex `lc` and the depicted multivector field `mvf` can be created using the function `example_multiflow`:

`ConleyDynamics.example_multiflow` – Method.

```
lc, mvf = example_multiflow()
```

Create the Lefschetz complex and multivector field for the example from Figure 3 in the connection matrix paper by Mrozek & Wanner.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_multiflow();

julia> cm = connection_matrix(lc, mvf);
```

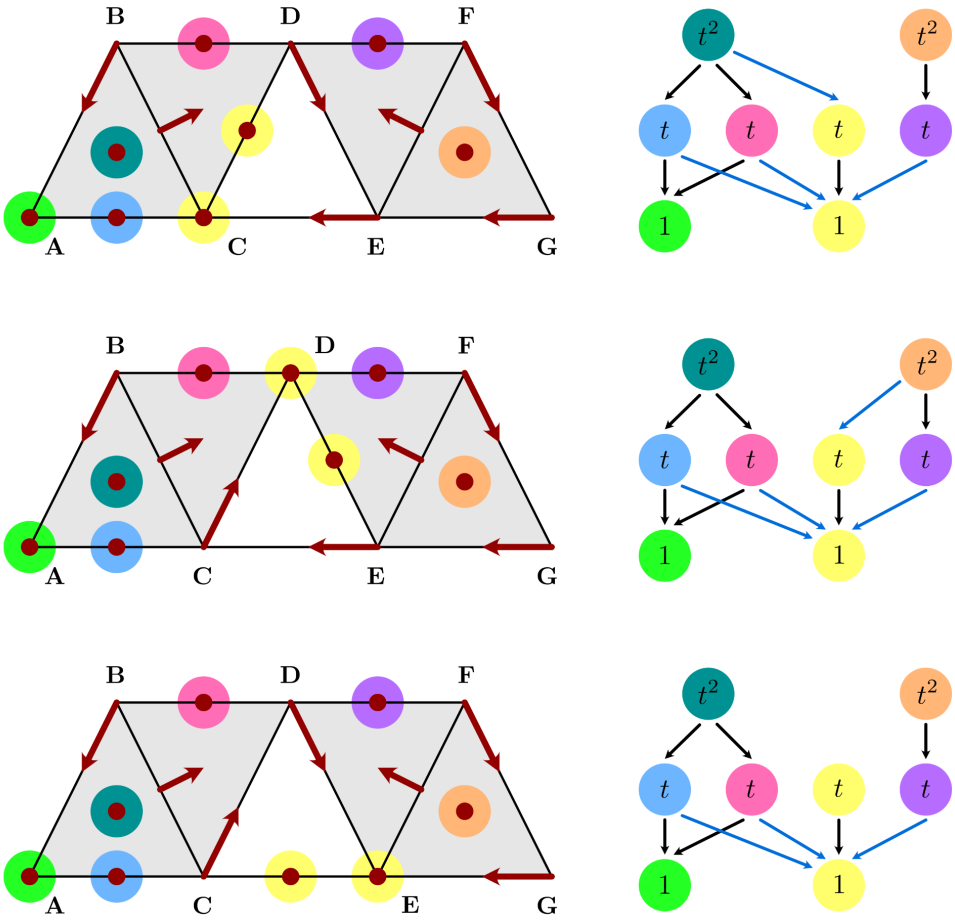


Figure 6.12: Morse decompositions for the three gradient vector fields

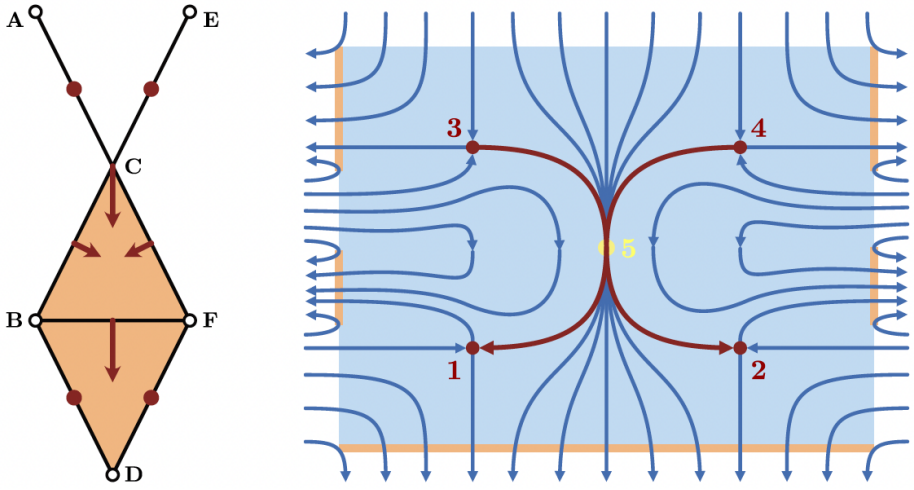


Figure 6.13: A multiflow example with trivial connection matrix

```
julia> sparse_show(cm.matrix)
 0 0 0 0
 0 0 0 0
 0 0 0 0
 0 0 0 0

julia> print(cm.labels)
["BD", "DF", "AC", "CE"]
```

source

As the docstring shows, this example has a trivial connection matrix. In other words, there are no connecting orbits in this combinatorial dynamical system that are forced by topology. In fact, one can easily see that due to the multivalued nature of the dynamical system, one cannot expect any particular heteroclinic to be present.

The Morse decomposition of the system, and the associated Conley indices encompass precisely the four critical edges:

```
julia> cm.morse
4-element Vector{Vector{String}}:
 ["BD"]
 ["DF"]
 ["AC"]
 ["CE"]

julia> cm.conley
4-element Vector{Vector{Int64}}:
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
 [0, 1, 0]
```

Combined with the fact that the connection matrix is trivial, this means that the homology of the underlying Lefschetz complex `lc` is the sum of the Conley indices of the Morse sets. This can be confirmed using the function `homology`:

```
julia> homology(lc)
3-element Vector{Int64}:
 0
 4
 0
```

As we mentioned earlier, this is the same as the relative homology of the full simplicial complex with respect to the union of the five removed vertices.

6.9 Small Complex with Periodicity

In [MW25, Figure 2.4] we introduced a small Lefschetz complex with periodic orbit and nonunique connection matrices. This complex consists of one 2-cell, three 1-cells, and two 0-cells, and it is shown in the leftmost panel of the figure.

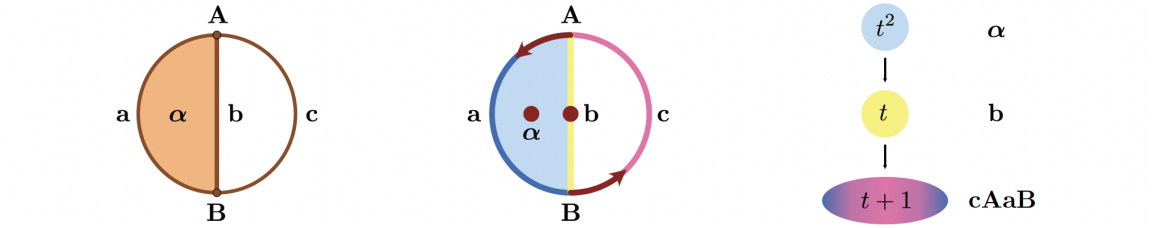


Figure 6.14: A small Lefschetz complex with periodic orbit

On the complex, consider the multivector field depicted in the middle of the figure, which consists of the critical cells α and b, as well as the two regular multivectors $\{A, a\}$ and $\{B, c\}$. For this small example, one can easily determine the Morse decomposition, and it is shown in the rightmost panel. The example can be generated in ConleyDynamics.jl using the function `example_small_periodicity`:

`ConleyDynamics.example_small_periodicity` – Method.

```
lc1, lc2, mvf = example_small_periodicity()
```

Create two representations of the Lefschetz complex and the multivector field for the example from Figure 4 in the connection matrix paper by Mrozek & Wanner.

The complexes `lc1` and `lc2` are just two representations of the same complex, but they lead to different connection matrices. Both Lefschetz complexes are defined over the finite field GF(2).

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector field `mvf`.

Examples

```
julia> lc1, lc2, mvf = example_small_periodicity();

julia> cm1 = connection_matrix(lc1, mvf);

julia> cm2 = connection_matrix(lc2, mvf);

julia> full_from_sparse(cm1.matrix)
4×4 Matrix{Int64}:
 0  0  0  0
 0  0  0  1
 0  0  0  1
 0  0  0  0

julia> print(cm1.labels)
["A", "a", "b", "alpha"]

julia> full_from_sparse(cm2.matrix)
4×4 Matrix{Int64}:
 0  0  0  0
 0  0  0  0
 0  0  0  1
 0  0  0  0

julia> print(cm2.labels)
["A", "c", "b", "alpha"]
```

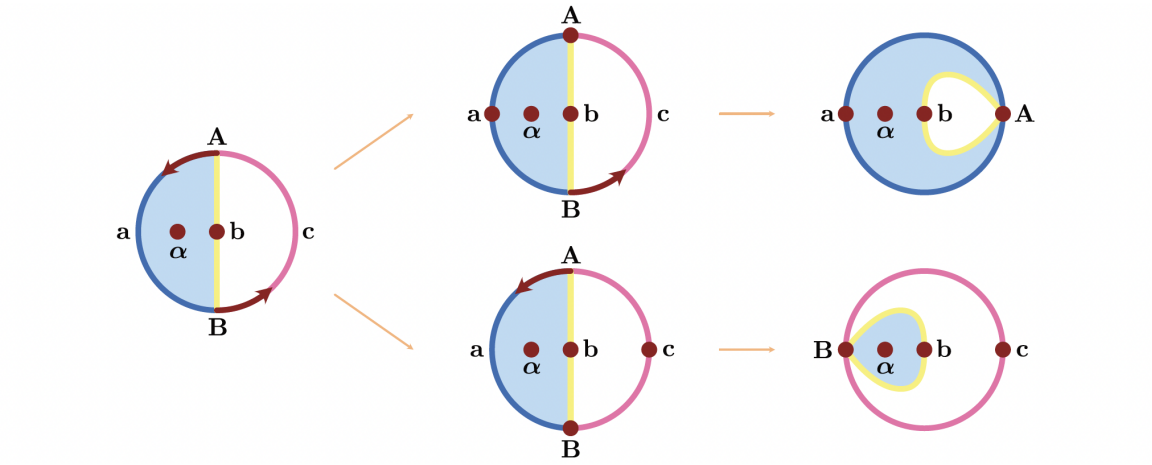


Figure 6.15: Direct derivation of connection matrices

source

The function provides two different representations of the same Lefschetz complex, which only differ in the ordering of the 1-cells. This can be seen from the commands:

```
julia> print(lc1.labels)
["A", "B", "a", "b", "c", "alpha"]

julia> print(lc2.labels)
["A", "B", "c", "a", "b", "alpha"]
```

As the above docstring shows, these different versions lead to two different connection matrices `cm1` and `cm2`.

In this small example, one can easily determine the connection matrices directly, as illustrated in the second figure. While the detailed explanation can be found in [MW25], this is basically accomplished by contracting one of the two regular multivectors in a process called elementary reduction. While the details of this approach are described in [KMS98], it relies on identifying a reduction pair, which contains a cell and one of its faces of one dimension less. These two cells are then removed from the Lefschetz complex and the boundary is modified in such a way that the new smaller complex still has the same homology as the previous one. If in our above example one uses the reduction pair $\{B, c\}$, then the boundary matrix of the reduced Lefschetz complex is the connection matrix `cm1`, while `cm2` is the result of using the reduction pair $\{A, a\}$. These manipulations can be done in ConleyDynamics.jl using the function `lefschetz_reduction`:

```
julia> rc1 = lefschetz_reduction(lc1, "B", "c");

julia> rc2 = lefschetz_reduction(lc1, "A", "a");
```

These two commands compute the reduced Lefschetz complexes `rc1` and `rc2` for the respective elementary reduction pairs $\{B, c\}$ and $\{A, a\}$. As the following commands show, the first one leads to our earlier connection matrix `cm1`:

```
julia> full_from_sparse(rc1.boundary)
4×4 Matrix{Int64}:
```

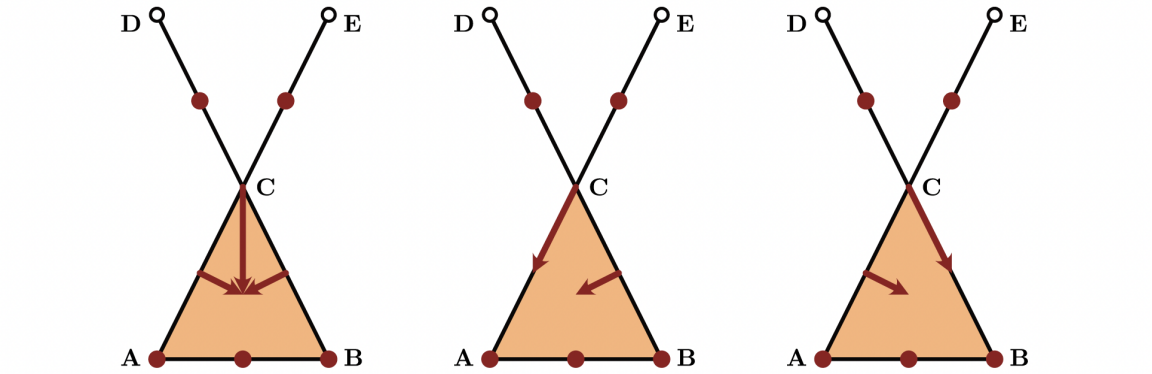


Figure 6.16: Subdividing a multivector

```
 0  0  0  0
 0  0  0  1
 0  0  0  1
 0  0  0  0

julia> println(rc1.labels)
["A", "a", "b", "alpha"]
```

Similarly, the connection matrix cm2 is the result of the second reduction:

```
julia> full_from_sparse(rc2.boundary)
4×4 Matrix{Int64}:
 0  0  0  0
 0  0  0  1
 0  0  0  0
 0  0  0  0

julia> println(rc2.labels)
["B", "b", "c", "alpha"]
```

6.10 Subdividing a Multivector

Our next example taken from [MW25] is concerned with turning a given multivector field into a Forman vector field by further subdividing the multivectors. For this, consider the three complexes and combinatorial vector fields shown in the associated figure, see also [MW25, Figure 7.1].

The depicted combinatorial dynamical systems all use the same Lefschetz complex, which is obtained from a simplicial complex by removing two vertices. In addition to the multivector field shown in the leftmost panel, we also consider two Forman vector fields. Notice that the multivector field has one multivector of size four, which is given by $\{\mathrm{C}, \mathrm{AC}, \mathrm{BC}, \mathrm{ABC}\}$. This vector can be split into two Forman arrows, and in view of the required local closedness this can be achieved in precisely two ways. The splitting into the arrows $\{\mathrm{C}, \mathrm{AC}\}$ and $\{\mathrm{BC}, \mathrm{ABC}\}$ gives the Forman vector field shown in the middle panel, while the one depicted in the rightmost panel uses the arrows $\{\mathrm{C}, \mathrm{BC}\}$ and $\{\mathrm{AC}, \mathrm{ABC}\}$. These fields can be created using the function `example_subdivision`:

`ConleyDynamics.example_subdivision` – Method.

```
lc, mvf = example_subdivision(mvftype)
```

Create the Lefschetz complex and multivector field for the example from Figure 11 in the connection matrix paper by Mrozek & Wanner.

Depending on the value of `mvftype`, return the multivector (0=default) or one of the two combinatorial vector field (1,2) examples.

The function returns the Lefschetz complex `lc` over the rationals and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_subdivision(1);

julia> cm = connection_matrix(lc, mvf);

julia> full_from_sparse(cm.matrix)
5×5 Matrix{Rational{Int64}}:
 0  0  -1  -1  -1
 0  0   1   0   0
 0  0   0   0   0
 0  0   0   0   0
 0  0   0   0   0
```

source

The different combinatorial vector fields can be selected via the function argument, which is an integer between `0` and 2, from left to right in the figure. Thus, all fields and connection matrices can be computed using the commands

```
lc0, mvf0 = example_subdivision(0)
lc1, mvf1 = example_subdivision(1)
lc2, mvf2 = example_subdivision(2)
cm0 = connection_matrix(lc0,mvf0)
cm1 = connection_matrix(lc1,mvf1)
cm2 = connection_matrix(lc2,mvf2)
```

All three vector fields give rise to the same Morse decomposition, since in each case the vectors of length at least two are regular. This can be seen for the multivector field below, and is analogous for the two Forman vector fields.

```
julia> cm0.morse
5-element Vector{Vector{String}}:
 ["A"]
 ["B"]
 ["AB"]
 ["CD"]
 ["CE"]

julia> cm0.conley
5-element Vector{Vector{Int64}}:
 [1, 0, 0]
```

```
[1, 0, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
```

The three connection matrices are as follows:

```
julia> full_from_sparse(cm0.matrix)
5×5 Matrix{Rational{Int64}}:
 0  0  -1   0   0
 0  0   1  -1  -1
 0  0   0   0   0
 0  0   0   0   0
 0  0   0   0   0

julia> full_from_sparse(cm1.matrix)
5×5 Matrix{Rational{Int64}}:
 0  0  -1  -1  -1
 0  0   1   0   0
 0  0   0   0   0
 0  0   0   0   0
 0  0   0   0   0

julia> full_from_sparse(cm2.matrix)
5×5 Matrix{Rational{Int64}}:
 0  0  -1   0   0
 0  0   1  -1  -1
 0  0   0   0   0
 0  0   0   0   0
 0  0   0   0   0
```

Notice that the connection matrices for the two Forman vector fields are uniquely determined, since both are gradient vector fields. The matrices are, however, different. At first glance, the connection matrix for `mvf0` is equal to the one for `mvf2`. Yet, this is another example of nonuniqueness, and one could produce also the connection matrix for `mvf1` through a reordering of the cells in the Lefschetz complex.

6.11 A Combinatorial Lorenz System

Our next example is taken from [KMW16, Figure 3]. It is a Forman vector field on a two-dimensional simplicial complex, as shown in the accompanying figure.

Notice that the underlying simplicial complex does not represent a manifold in this case, but rather a branched manifold. As indicated in the figure, the two triangles $\mathbf{hin}$ and $\mathbf{hio}$ only intersect in the edge $\mathbf{hi}$, and this edge is also contained in the third triangle $\mathbf{chi}$. This system is a combinatorial version of the famous Lorenz butterfly, which is well-known for its chaotic behavior. The simplicial complex and the Forman vector field can be created using the function `example_clorenz`:

`ConleyDynamics.example_clorenz` – Method.

```
lc, mvf = example_clorenz()
```

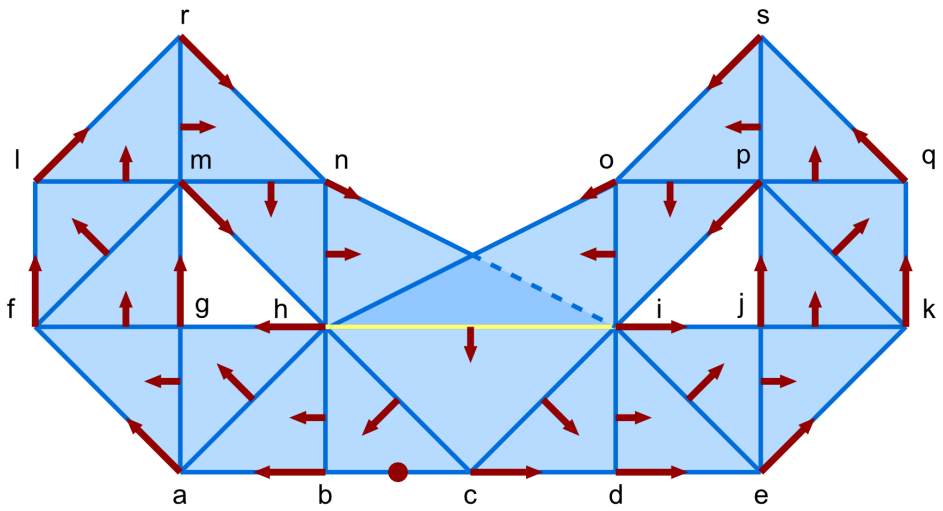


Figure 6.17: A combinatorial Lorenz system

Create the simplicial complex and multivector field for the example from Figure 3 in the JCD 2016 paper by Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` over the finite field GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_clorenz();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 0

julia> print(cm.labels)
["i", "ip", "g", "gm", "bc"]

julia> ms, ps = morse_sets(lc, mvf, poset=true);

julia> [conley_index(lc, mset) for mset in ms]
4-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [1, 1, 0]
 [0, 1, 0]
 [0, 0, 0]

julia> ps
4×4 Matrix{Bool}:
 0  0  1  0
 0  0  1  0
 0  0  0  1
```

```
 0  0  0  0
```

source

The first of the above commands creates the simplicial complex `lc` and the Forman vector field `mvf`. These are then analyzed using the following commands:

- Using `cm = connection_matrix(lc, mvf)` one can compute the connection matrix of the example. This connection matrix has two nonzero entries, which indicate connecting orbits from the index 1 critical cell $\mathbf{bc}$ to each of the stable periodic orbits spanned by the vertices $\mathbf{h}$, $\mathbf{g}$, $\mathbf{m}$ and $\mathbf{i}$, $\mathbf{j}$, $\mathbf{p}$, respectively.
- The command `ms, ps = morse_sets(lc, mvf, poset=true)` shows, however, that there is more to this example. While the above connection matrix indicates only three isolated invariant sets, there is actually a fourth one. In view of the Conley index computation for these sets, the additional Morse set has trivial index, and therefore does not show up in the connection matrix. Notice that the partial order given by the flow, which is indicated by the matrix `ps`, implies that the Morse set with trivial index has a connection to the index 1 critical cell.

Upon closer inspection one can see that the last Morse set is comprised of all triangles, as well as all edges which are contained in at least two triangles. This set contains infinitely many periodic orbits. For any bi-infinite sequence of symbols L and R there is a periodic orbit which loops around the left hole for L and around the right hole for R, as one traverses the symbol sequence from left to right. Such behavior is one potential indicator for chaos. In fact, it was shown in [MW21] that it is possible to define a classical semiflow on the branched manifold defined by `lc` which mimics the behavior of the Forman vector field. And according to [MSTW22], any such admissible semiflow does indeed have infinitely many periodic orbits in the set determined by the triangles.

6.12 Chaos in the Dunce Hat

In the above example we observed chaotic behavior in a branched manifold. It is also possible to introduce similar behavior in the Dunce hat.

The required simplicial complex and Forman vector field can be created using the function `example_dunce_chaos`:

`ConleyDynamics.example_dunce_chaos` – Function.

```
sc, vfG, vfC = example_dunce_chaos()
```

Create a minimal simplicial complex representation of the Dunce hat, as well as two Forman vector fields.

The function returns the simplicial representation of the Dunce hat in `sc` over the finite field `GF(2)`. The Forman vector field `vfG` is a gradient vector field with unique connection matrix. The field `vfC` is a modification of this field which merges the critical cells of dimensions 1 and 2 into a Forman arrow. The resulting Forman vector field is no longer gradient, and in fact exhibits Lorez-like chaos.

Examples

```
julia> sc, vfG, vfC = example_dunce_chaos();

julia> homology(sc)
3-element Vector{Int64}:
```

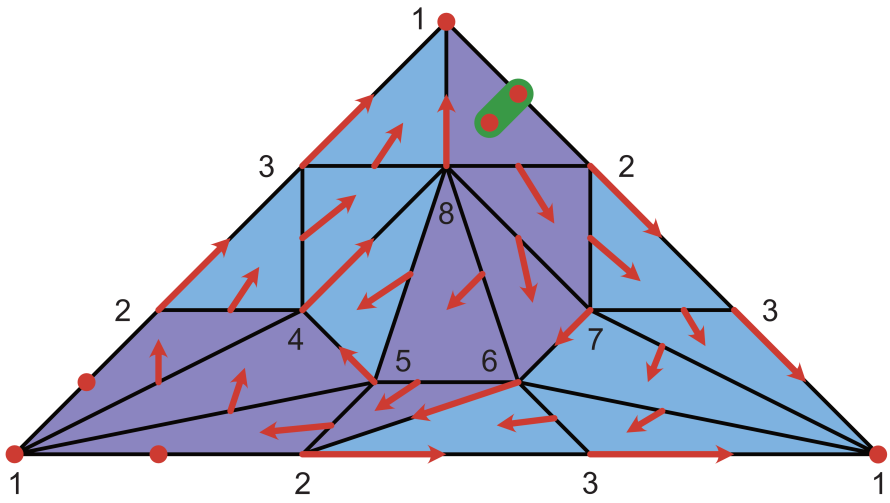


Figure 6.18: Forman chaos in the Dunce hat

```
 1
 0
 0

julia> cmG = connection_matrix(sc, vfG);

julia> sparse_show(cmG.matrix)
 0 0 0
 0 0 1
 0 0 0

julia> print(cmG.labels)
["1", "12", "128"]

julia> cmC = connection_matrix(sc, vfC);

julia> sparse_show(cmC.matrix)
 0

julia> print(cmC.labels)
["1"]

julia> msC, psC = morse_sets(sc, vfC, poset=true);

julia> [conley_index(sc, mset) for mset in msC]
2-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 0, 0]

julia> psC
2×2 Matrix{Bool}:
 0  1
 0  0
```

```
julia> msC
2-element Vector{Vector{String}}:
 ["1"]
 ["12", "14", "15", "25", "28", "56", "68", "78", "124", "125", "128", "145", "256", "278",
↪  "568", "678"]
```

source

The first of the above commands creates the simplicial complex `sc` and two Forman vector fields. The simplicial complex contains a minimal triangulation of the Dunce hat, which has 8 vertices, 24 edges, and 17 triangles. The variable `vfG` returns a gradient vector field, as shown in the associated figure. This vector field has the three critical simplices 1, 12, and 128. Finally, the chaotic Forman vector field is returned in `vfC`. Is it obtained from the gradient field by combining the two critical cells 12 and 128 into a Forman arrow, which is indicated in green in the figure. This reverses the flow between these two simplices. In more detail, the above example provides the following analysis:

- The command `homology(sc)` shows that the Dunce hat has the homology of a point, i.e., it is contractible.
- The next command `cmG = connection_matrix(sc, vfG)` determines the connection matrix for the gradient system. As the entry `cmG.labels` shows, the Morse sets are the three simplices 1, 12, and 128, and the nontrivial entry in the connection matrix indicates at least one connection between the index 2 critical cell and the index 1 critical cell. In fact, there are three such connections.
- The command `cmC = connection_matrix(sc, vfC)` computes the connection matrix. This time, the matrix is the zero matrix with one row and column, which correspond to the stable critical cell 1.
- But there is nontrivial dynamics, as the command `msC, psC = morse_sets(sc, vfC, poset=true)` demonstrates. It produces two Morse sets. In addition to the stable critical simplex, one also obtains an isolated invariant set with trivial index. This set exhibits chaotic behavior, and is shown in purple in the accompanying figure.
- The return variable `psC` gives the flow-induced order between the Morse sets, which shows that there are heteroclinic orbits between the chaotic Morse set and the stable equilibrium.
- The variable `msC` contains the two Morse sets. While the first is the stable equilibrium, the second one consists of 8 edges and 8 triangles.

One can see from the figure that the chaotic isolated invariant set for the Forman vector field `vfC` has two periodic orbits. Starting with the edge 12 in the upper right, they both traverse the edges 28, 78, 68, 56, and 25. But while the first periodic orbit then returns directly to 12 along the bottom edge, the second one moves first through 15 and 14, before returning to 12 along the left edge.

6.13 Chaos in a Space with Torsion

Our next example describes gradient Forman vector fields on simplicial complexes with torsion whose connection matrices can have large entries, as long as the underlying field has a large enough characteristic. In addition, by merging two critical cells in the Forman gradient vector field into an arrow, once obtains a Forman vector field with chaotic behavior. The simplicial complexes are based on the function `simplicial_torsion_space`, and the Forman vector fields can be seen in the associated figure for the cases $n = 3$ and $n = 4$.

After specifying the torsion coefficient n and the field characteristic p, the underlying simplicial complex and the two Forman vector fields can be created using the function `example_torsion_chaos`:

`ConleyDynamics.example_torsion_chaos` – Function.

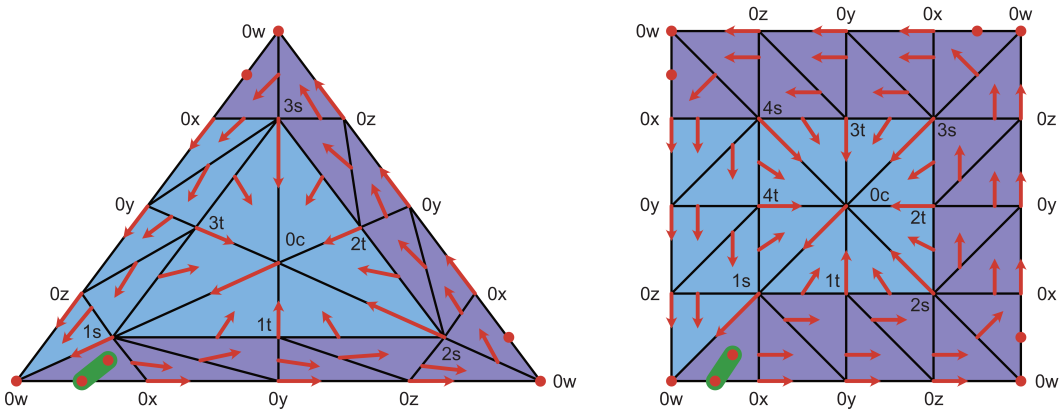


Figure 6.19: Forman chaos in a space with torsion

```
sc, vfG, vfC = example_torsion_chaos(n::Int, p::Int)
```

Create a triangulation of a space with 1-dimensional torsion, as well as two Forman vectr fields on this complex.

The function returns a simplicial complex `sc` which has the following integer homology groups:

- In dimension 0 it is the group of integers.
- In dimension 1 it is the integers modulo n.
- In dimension 2 it is the trivial group.

In other words, the simplicial complex has nontrivial torsion in dimension 1. It is a triangulation of an n-gon, in which all boundary edges are oriented counterclockwise, and all of these edges are identified. The parameter p specifies the characteristic of the underlying field.

In addition, two Forman vector fields `vfG` and `vfC` are returned. The first one is a gradient vector field whose connection matrix has a large connection matrix entry. In fact, if p is any prime larger than n then there will be an entry n in the matrix. The scond Forman vector field contains a chaotic Morse set. This Morse set will have trivial Morse index for most p. On the other hand, for prime `p = n` the set has the Morse index of an unstable periodic orbit.

Examples

```
julia> sc, vfG, vfC = example_torsion_chaos(3,7);

julia> homology(sc)
3-element Vector{Int64}:
 1
 0
 0

julia> cmG = connection_matrix(sc, vfG);

julia> sparse_show(cmG.matrix)
 0 0 0
```

```
 0 0 3
 0 0 0

julia> print(cmG.labels)
["0w", "0w0x", "0w0x1s"]

julia> cmC = connection_matrix(sc, vfC);

julia> sparse_show(cmC.matrix)
 0

julia> print(cmC.labels)
["0w"]

julia> msC, psC = morse_sets(sc, vfC, poset=true);

julia> [conley_index(sc, mset) for mset in msC]
2-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 0, 0]

julia> psC
2×2 Matrix{Bool}:
 0  1
 0  0

julia> length.(msC)
2-element Vector{Int64}:
  1
 26
```

source

The first of the above commands creates the simplicial complex `sc` and the two Forman vector fields. The gradient vector field is returned in `vfG`, and it corresponds to the Forman vector fields shown in red in the accompanying figure. The return variable `vfC` gives the chaotic Forman vector field. Is it obtained from the gradient field by combining the two critical cells $0\mathrm{w}0\mathrm{x}$ and $0\mathrm{w}0\mathrm{x}1\mathrm{s}$ into a Forman arrow, which is indicated in green in the figure. Note that this reverses the flow between these two simplices. In the above example, we use $n = 3$ and $p = 7$.

The gradient Forman vector field `vfG` is then analyzed using the following commands:

- The command `homology(sc)` shows that in the field $GF(7)$, the space has the homology of a point.
- The next command `cmG = connection_matrix(sc, vfG)` determines the connection matrix for the gradient system. As the entry `cmG.labels` shows, the Morse sets are the three simplices $0\mathrm{w}$, $0\mathrm{w}0\mathrm{x}$, and $0\mathrm{w}0\mathrm{x}1\mathrm{s}$, and the nontrivial entry in the connection matrix indicates three connections between the index 2 critical cell and the index 1 critical cell.

The next commands analyze the chaotic Forman vector field `vfC`:

- As before, the command `cmC = connection_matrix(sc, vfC)` computes the connection matrix. This time, the matrix is the zero matrix with one row and column, which correspond to the stable critical cell $0\mathrm{w}$.

- But there is nontrivial dynamics, as the command `msC, psC = morse_sets(sc, vfC, poset=true)` demonstrates. It produces two Morse sets. In addition to the stable critical cell, one also obtains an isolated invariant set with trivial index. This set exhibits chaotic behavior, and is shown in purple in the accompanying figure. In the case $n = 3$, it consists of 26 simplices. In the general case, it will have $12n - 10$ simplices.

- The return variable `psC` gives the flow-induced order between the Morse sets, which shows that there are heteroclinic orbits between the chaotic Morse set and the stable equilibrium.

6.14 References

See the full bibliography for a complete list of references cited throughout this documentation. This section cites the following references:

[BKMW20] B. Batko, T. Kaczynski, M. Mrozek and T. Wanner. Linking combinatorial and classical dynamics: Conley index and Morse decompositions. Foundations of Computational Mathematics **20**, 967–1012 (2020).

[KMS98] T. Kaczynski, M. Mrozek and M. Slusarek. Homology computation by reduction of chain complexes. Computers & Mathematics with Applications **35**, 59–70 (1998).

[KMW16] T. Kaczynski, M. Mrozek and T. Wanner. Towards a formal tie between combinatorial and classical vector field dynamics. Journal of Computational Dynamics **3**, 17–50 (2016).

[MSTW22] M. Mrozek, R. Srzednicki, J. Thorpe and T. Wanner. Combinatorial vs. classical dynamics: Recurrence. Communications in Nonlinear Science and Numerical Simulation **108**, Paper No. 106226, 30 pages (2022).

[MW21] M. Mrozek and T. Wanner. Creating semiflows on simplicial complexes from combinatorial vector fields. Journal of Differential Equations **304**, 375–434 (2021).

[MW25] M. Mrozek and T. Wanner. Connection Matrices in Combinatorial Topological Dynamics. SpringerBriefs in Mathematics (Springer-Verlag, Cham, 2025).

Chapter 7

Sparse Matrices

While Julia provides a data structure for sparse matrix computations, the employed design decisions make it difficult to use this implementation for computations over finite fields. This is mainly due to the fact that in the Julia implementation, it is assumed that one can determine the zero and one elements from the data type alone. However, a finite field data type generally also depends on additional parameters, such as the characteristic of the field.

Since the algorithms underlying ConleyDynamics.jl only require basic row and column operations, a specialized sparse matrix implementation is provided in the package. It is briefly described in the following.

7.1 Sparse Matrix Format

Sparse matrices in this package have to be of the composite data type `SparseMatrix`, which is structured as follows:

`ConleyDynamics.SparseMatrix` – Type.

```
SparseMatrix{T}
```

Composite data type for a sparse matrix with entries of type T.

The struct has the following fields:

- `const nrow::Int`: Number of rows
- `const ncol::Int`: Number of columns
- `const char::Int`: Characteristic of type T
- `const zero::T`: Number 0 of type T
- `const one::T`: Number 1 of type T
- `entries::Vector{Vector{T}}`: Matrix entries corresponding to `columns`
- `columns::Vector{Vector{Int}}`: `columns[k]` points to nonzero entries in column k
- `rows::Vector{Vector{Int}}`: `rows[k]` points to nonzero entries in the k-th row

source

In this struct, the type T has to be either `Int` or `Rational{Int}`, depending on whether the sparse matrix is interpreted as a matrix with entries in the finite field $GF(p)$ for some prime p, or over the field of rationals, respectively. The data type has the following fields:

- `nrow::Int` designates the number of rows.
- `ncol::Int` gives the number of columns.
- `char::Int` specifies the characteristic of the underlying field F. If `char=0`, then the field is the rationals $\mathbb{Q}$, and one has to have `T = Rational{Int}`. If, on the other hand, the finite field $F = GF(p)$ is used, then `char=p` has to be a prime number. In this case, the data type of the matrix entries has to be `T = Int`.
- `zero::T` provides 0 in the data type T.
- `one::T` provides 1 in the data type T.
- `columns::Vector{Vector{Int}}` is a vector of integer vectors, which contains the row indices of nonzero matrix entries in each column. More precisely, `columns[k]` contains an increasing list of row indices, which give the locations of all nonzero entries in column k. Note that the list for each colum has to be strictly increasing.
- `rows::Vector{Vector{Int}}` is a vector of integer vectors, which contains the column indices of nonzero matrix entries in each row. It is the precise dual to the previous field. This time, `rows[k]` contains an increasing list of column indices, which correspond to the nonzero entries of the matrix in the k-th row.
- `entries::Vector{Vector{T}}` is a vector of vectors which contains the actual matrix entries. It is organized in exactly the same way as the field `columns`. In other words, for every `k = 1,..,ncol` the matrix entry in column k and row `columns[k][j]` is given by `entries[k][j]`, where `j` indexes the nonzero column entries from top to bottom.

This data structure is clearly redundant, in the sense that the field `rows` is not needed to uniquely determine the matrix. However, the type `SparseMatrix` is fundamental for almost every aspect of ConleyDynamics.jl, as it is used to encode the incidence coefficient map κ, and therefore also the matrix representation of the boundary operator ∂. And for many operations on or queries of Lefschetz complexes, one needs fast access to both the cells in the boundary and the coboundary of a given cells. While the boundary can easily be accessed via the field `columns`, the fast coboundary access is aided by the field `rows`.

We would like to point out that in view of the different underlying fields, sparse matrices should only be manipulated using the specific commands provided by the package. These are described in detail below. If there is a need for additional functionality beyond these first methods, it can be added at a later point in time.

7.2 Creating Sparse Matrices

The package provides a number of methods for creating sparse matrices with the data type `SparseMatrix`. These are geared towards their usage within ConleyDynamics.jl and are therefore by no means exhaustive:

- `sparse_from_full` is usually invoked in the form `A = sparse_from_full(AF, p=PP)`. The first input argument `AF` has to be a regular Julia integer matrix. This matrix is then converted to sparse format and returned as `A`. If the optional parameter `p` is omitted, the resulting sparse matrix is over the rational numbers $\mathbb{Q}$, otherwise it is over the finite field with characteristic `PP`.
- `full_from_sparse` converts a given sparse matrix into a standard full matrix in Julia. The data type of the entries is either `Rational{Int}` or `Int`, depending on whether the sparse input matrix is considered over the rationals $\mathbb{Q}$ or over a finite field, respectively. When invoking this command, be mindful of the size of the sparse matrix!
- `sparse_from_lists` creates a sparse matrix solely based on its nonzero entries and their locations. It expects the following required input arguments, in the order they are listed:

  - `nr::Int`: Number of rows
  - `nc::Int`: Number of columns
  - `tchar`: Field characteristic, which has to be 0 if $F = \mathbb{Q}$ and a positive prime otherwise
  - `tzero::T`: Number 0 of type T
  - `tone::T`: Number 1 of type T
  - `r::Vector{Int}`: Vector of row indices
  - `c::Vector{Int}`: Vector of column indices
  - `v::Vector{T}`: Vector of matrix entries

  The function assumes that the vectors `r`, `c`, and `v` have the same length and that the matrix has entry `v[k]` at the location `(r[k],c[k])`. Zero entries will be ignored, and multiple entries for the same matrix position raise an error. Furthermore, if `tchar>0`, then the entries in `v` are all replaced by their values modulo `tchar`. As mentioned before, if `tchar=0` then the entry type has to be `T = Rational{Int}`, otherwise we have `T = Int`.

- `lists_from_sparse` takes a sparse matrix and disassembles it into the separate ingredients specified in the discussion of the previous function. In this sense, it is precisely the inverse method of `sparse_from_lists`.

- `sparse_identity` creates a sparse identity matrix. It is invoked as `A = sparse_identity(n, p=PP)`, and returns a sparse identity matrix `A` with `n` rows and `n` columns. If the optional characteristic parameter specified and positive, then the matrix is considered over the finite field with characteristic `PP`, otherwise it is over the rationals $\mathbb{Q}$.

- `sparse_zero` creates a sparse zero matrix. It is invoked as `A = sparse_zero(nr, nc, p=PP)`, and returns a sparse zero matrix `A` with `nr` rows and `nc` columns. If the optional characteristic parameter specified and positive, then the matrix is considered over the finite field with characteristic `PP`, otherwise it is over the rationals $\mathbb{Q}$.

Of these methods, the function `sparse_from_lists` provides the easiest and quickest way to create a sparse matrix.

7.3 Sparse Matrix Access

Access to the entries of sparse matrices is provided via the following commands:

- `sparse_get_entry` extracts the matrix entry `val` of the matrix `A` located in row `ri` and column `ci`, if it is invoked using the command `val = sparse_get_entry(A,ri,ci)`.

- `sparse_set_entry!` sets the matrix entry of the matrix `A` located in row `ri` and column `ci` to the value 'val', if it is invoked using the command `sparse_set_entry!(A,ri,ci,val)`. Internally, this commands makes sure that the above-defined format of the fields of a sparse matrix is preserved. Note that the data type of `val` has to match the type of `A.zero`. If the matrix is a sparse matrix over the integers, it is considered as a sparse matrix over a finite field. Thus, the value `val` is automatically reduced modulo `A.char` before being assigned to the entry.

- `sparse_get_column` is invoked as `Acol = sparse_get_column(A,ci)`, and it returns the full `ci`-th column of the matrix `A` as a `Vector{T}` of length `A.nrow`.

- `sparse_get_nz_column` returns the row indices for the nonzero entries in the `ci`-th column of the sparse matrix `A`, if invoked as `rivec = sparse_get_nz_column(A,ci)`.

- sparse_get_nz_row returns the column indices for the nonzero entries in the ri-th row of the sparse matrix A, if invoked as civec = sparse_get_nz_row(A,ri).
- sparse_minor creates a minor from a given sparse matrix A. For this, one needs to specify the row and column indices of the minor in the integer vectors rvec and cvec, respectively, and then invoke the function using the command AM = sparse_minor(A,rvec,cvec). Note that the entries in rvec and cvec do not have to be in increasing order, but they are not allowed to contain repeated indices.

One can also read and set sparse matrix values using the overloaded methods y = A[i,j] and A[i,j] = val. In the latter case, if the matrix is defined as SparseMatrix{Int}, then it is interpreted as being over a finite field. In this case, the value val is automatically reduced via modular arithmetic modulo A.char before the assignment.

7.4 Elementary Matrix Operations

The following commands perform the basic sparse matrix operations that are needed for the functionality of the package:

- sparse_add_column! is invoked using the form sparse_add_column!(A,ci1,ci2,cn,cd), and it replaces the ci1-th column column[ci1] of A by column[ci1] + (cn/cd) * column[ci2]. This operation automatically performs the computations over the field F underlying the sparse matrix A. In other words, if this field is finite, then it determines the inverse of the argument cd as part of the computation.
- sparse_add_row! is invoked using the form sparse_add_row!(A,ri1,ri2,cn,cd), and it replaces the ri1-th row row[ri1] of A by row[ri1] + (cn/cd) * row[ri2]. As before, this operation automatically performs the computations over the field F underlying the sparse matrix A.
- sparse_permute creates a new sparse matrix by permuting the row and column indices. It is invoked using the command AP = sparse_permute(A,pr,pc), and the integer vectors pr and pc have to describe the row and column permutations, respectively.
- sparse_remove! is invoked as sparse_remove!(A,ri,ci) and removes the sparse matrix entry in the ri-th row and ci-th colum, i.e., it effectively sets the entry equal to zero.
- sparse_add computes the matrix sum of two sparse matrices. Exceptions are raised if the matrix sum is not defined, or if the involved sparse matrices are defined over different fields. One can also use the operator form A+B to compute the sum of sparse matrices.
- sparse_subtract computes the matrix difference of two sparse matrices. Exceptions are raised if the matrix difference is not defined, or if the involved sparse matrices are defined over different fields. One can also use the operator form A-B to compute the sum of sparse matrices.
- sparse_multiply computes the matrix product of two sparse matrices. Exceptions are raised if the matrix product is not defined, or if the involved sparse matrices are defined over different fields. One can also use the operator form A*B to compute the product of sparse matrices.
- sparse_scale computes the scalar product of a number and a sparse matrix. An exception is raised if the scalar and the matrix entries are not of the same type. One can also use the operator form sfac*A to compute the scalar product.
- sparse_inverse computes the inverse matrix of a sparse matrix. Exceptions are raised if the matrix is not square or not invertible.

There are also the useful helper functions `scalar_inverse`, `scalar_multiply`, and `scalar_add` which compute the inverse of a scalar, the product of two scalars, or the sum of two scalars, respectively. Depending on the input type, these computations are performed either over the rationals, or in modular arithmetic. See the function documentations for more details.

As mentioned earlier, additional operations can easily be implemented if they become necessary.

7.5 Sparse Matrix Information

Finally, ConleyDynamics.jl provides the following functions for quickly extracting certain information from sparse matrices:

- `sparse_size` is invoked as `size = sparse_size(A,dim)`, and it returns the number of rows if `dim=1`, or the number of columns for `dim=2`.
- `sparse_low` returns the largest row index `ri` of a nonzero entry in the `ci`-th column of the matrix `A`, if used in the form `ri = sparse_low(A,ci)`. In other words, it returns the row index of the lowest nonzero matrix entry in the column.
- `sparse_is_zero` checks whether a sparse matrix is the zero matrix.
- `sparse_is_identity` checks whether a sparse matrix is the identity matrix.
- `sparse_is_equal` checks whether two sparse matrices are the same. For this, they not only have to have the same size and the same entries, they also need to be defined over the same field. This function can also be invoked using `A == B`.
- `sparse_is_sut` checks whether a given sparse matrix is strictly upper triangular, and returns either `true` or `false`.
- `sparse_fullness` returns the fullness of a sparse matrix as a floating point number. Here fullness refers to the ratio of the number of nonzero matrix elements and the total number of matrix entries.
- `sparse_sparsity` computes the sparseness of a sparse matrix, which is defined as 1 minus its fullness, i.e., it is the ratio of the number of zero matrix elements and the total number of matrix entries.
- `sparse_nz_count` determines the number of nonzero matrix entries.
- `sparse_show` can be used to display a sparse matrix in traditional matrix form at the Julia REPL prompt. This function has additional methods which allow the user to display row and column labels, which have to be specified as second and third arguments. In addition, if `cm` is a connection matrix, then the command `sparse_show(cm)` shows the connection matrix including the Conley index labels.

Chapter 8

References

[Ale37] P. Alexandrov. Diskrete Räume. Mathematiceskii Sbornik (N.S.) **2**, 501–518 (1937).

[BKMW20] B. Batko, T. Kaczynski, M. Mrozek and T. Wanner. Linking combinatorial and classical dynamics: Conley index and Morse decompositions. Foundations of Computational Mathematics **20**, 967–1012 (2020).

[Con78] C. Conley. Isolated Invariant Sets and the Morse Index (American Mathematical Society, Providence, R.I., 1978).

[DKMW11] P. Dłotko, T. Kaczynski, M. Mrozek and T. Wanner. Coreduction homology algorithm for regular CW-complexes. Discrete & Computational Geometry **46**, 361–388 (2011).

[DLMS24] T. K. Dey, M. Lipiński, M. Mrozek and R. Slechta. Computing connection matrices via persistence-like reductions. SIAM Journal on Applied Dynamical Systems **23**, 81–97 (2024).

[DW18] P. Dłotko and T. Wanner. Rigorous cubical approximation and persistent homology of continuous functions. Computers & Mathematics with Applications **75**, 1648–1666 (2018).

[EH10] H. Edelsbrunner and J. L. Harer. Computational Topology (American Mathematical Society, Providence, 2010).

[EM23] H. Edelsbrunner and M. Mrozek. The depth poset of a filtered Lefschetz complex (2023), arXiv:2311.14364v2 [math.AT].

[For98a] R. Forman. Combinatorial vector fields and dynamical systems. Mathematische Zeitschrift **228**, 629–681 (1998).

[For98b] R. Forman. Morse theory for cell complexes. Advances in Mathematics **134**, 90–145 (1998).

[Fra89] R. Franzosa. The connection matrix theory for Morse decompositions. Transactions of the American Mathematical Society **311**, 561–592 (1989).

[GMW05] M. Gameiro, K. Mischaikow and T. Wanner. Evolution of pattern complexity in the Cahn-Hilliard theory of phase separation. Acta Materialia **53**, 693–704 (2005).

[HMS21] S. Harker, K. Mischaikow and K. Spendlove. A computational framework for connection matrix theory. Journal of Applied and Computational Topology **5**, 459–529 (2021).

[KMM04] T. Kaczynski, K. Mischaikow and M. Mrozek. Computational Homology. Vol. 157 of Applied Mathematical Sciences (Springer-Verlag, New York, 2004).

[KMS98] T. Kaczynski, M. Mrozek and M. Slusarek. Homology computation by reduction of chain complexes. Computers & Mathematics with Applications **35**, 59–70 (1998).

[KMW16] T. Kaczynski, M. Mrozek and T. Wanner. Towards a formal tie between combinatorial and classical vector field dynamics. Journal of Computational Dynamics **3**, 17–50 (2016).

[Lef42] S. Lefschetz. Algebraic Topology. Vol. 27 of American Mathematical Society Colloquium Publications (American Mathematical Society, New York, 1942).

[LKMW23] M. Lipinski, J. Kubica, M. Mrozek and T. Wanner. Conley-Morse-Forman theory for generalized combinatorial multivector fields on finite topological spaces. Journal of Applied and Computational Topology **7**, 139–184 (2023).

[Mas91] W. S. Massey. A Basic Course in Algebraic Topology. Vol. 127 of Graduate Texts in Mathematics (Springer-Verlag, New York, 1991).

[MB09] M. Mrozek and B. Batko. Coreduction homology algorithm. Discrete & Computational Geometry **41**, 96–118 (2009).

[MSTW22] M. Mrozek, R. Srzednicki, J. Thorpe and T. Wanner. Combinatorial vs. classical dynamics: Recurrence. Communications in Nonlinear Science and Numerical Simulation **108**, Paper No. 106226, 30 pages (2022).

[MW21] M. Mrozek and T. Wanner. Creating semiflows on simplicial complexes from combinatorial vector fields. Journal of Differential Equations **304**, 375–434 (2021).

[MW25] M. Mrozek and T. Wanner. Connection Matrices in Combinatorial Topological Dynamics. SpringerBriefs in Mathematics (Springer-Verlag, Cham, 2025).

[Mun84] J. R. Munkres. Elements of Algebraic Topology. SpringerBriefs in Mathematics (Addison-Wesley, Menlo Park, 1984).

[SW24] E. Sander and T. Wanner. Theory and Numerics of Partial Differential Equations (SIAM, Philadelphia, 2024). In preparation, 1007 pages.

[SW14a] T. Stephens and T. Wanner. Isolating block validation in Matlab, https://github.com/almost6heads/isoblockval (2014).

[SW14b] T. Stephens and T. Wanner. Rigorous validation of isolating blocks for flows and their Conley indices. SIAM Journal on Applied Dynamical Systems **13**, 1847–1878 (2014).

[Wan25] T. Wanner. ConleyDynamics.jl: A Julia package for multivector dynamics on Lefschetz complexes. Journal of Open Source Software **10**, 8085 (2025).

[GUD24] GUDHI Project. GUDHI User and Reference Manual. 3.10.1 Edition (GUDHI Editorial Board, 2024).

Part III

Core API

Chapter 9

Composite Data Structures

The package relies on a number of basic composite data structures that encompass more complicated objects. For the internal representation of sparse matrices we refer to Internal Sparse Matrix Representation.

`ConleyDynamics` – Module.

```
module ConleyDynamics
```

Collection of tools for computational Conley theory.

source

9.1 Lefschetz Complex Type

`ConleyDynamics.LefschetzComplex` – Type.

```
LefschetzComplex
```

Collect the Lefschetz complex information in a struct.

The struct is created via the following fields:

- `labels::Vector{String}`: Vector of labels associated with cell indices
- `dimensions::Vector{Int}`: Vector cell dimensions
- `boundary::SparseMatrix`: Boundary matrix, columns give the cell boundaries

It is expected that the dimensions are given in increasing order, and that the square of the boundary matrix is zero. Otherwise, exceptions are raised. In addition, the following fields are created during initialization:

- `ncells::Int`: Number of cells
- `dim::Int`: Dimension of the complex
- `indices::Dict{String,Int}`: Dictionary for finding cell index from label

The coefficient field is specified by the boundary matrix.

Warning

Note that the constructor does not check whether the boundary matrix squares to zero. It is the responsibility of the user to ensure that!

source

`Base.show` – Method.

```
Base.show(io::IO, ::MIME"text/plain", lc::LefschetzComplex)
```

Display Lefschetz complex information when hitting return in REPL.

source

9.2 Cell Subset Types

`ConleyDynamics.Cell` – Type.

```
Cell = Union{Int,String}
```

A cell of a Lefschetz complex.

This data type is used to represent a cell of a Lefschetz complex. The cell can be specified either via its index, or its label.

source

`ConleyDynamics.Cells` – Type.

```
Cells = Union{Vector{Int},Vector{String}}
```

A list of cells of a Lefschetz complex.

This data type is used to represent subsets of a Lefschetz complex. It is used for individual isolated invariant sets, locally closed subsets, and multivectors.

source

`ConleyDynamics.CellSubsets` – Type.

```
CellSubsets = Union{Vector{Vector{Int}},Vector{Vector{String}}}
```

A collection of cell lists.

This data type is used to represent a collection of subsets of a Lefschetz complex. It is used for Morse decompositions and for multivector fields.

source

9.3 Conley-Morse Graph Type

`ConleyDynamics.ConleyMorseCM` – Type.

```
ConleyMorseCM{T}
```

Collect the connection matrix information in a struct.

The struct has the following fields:

- `matrix::SparseMatrix{T}`: Connection matrix
- `columns::Vector{Int}`: Corresponding columns in the boundary matrix
- `poset::Vector{Int}`: Poset indices for the connection matrix columns
- `labels::Vector{String}`: Labels for the connection matrix columns
- `morse::Vector{Vector{String}}`: Vector of Morse sets in original complex
- `conley::Vector{Vector{Int}}`: Vector of Conley indices for the Morse sets
- `complex::LefschetzComplex`: The Conley complex as a Lefschetz complex

source

`Base.show` – Method.

```
Base.show(io::IO, ::MIME"text/plain", cm::ConleyMorseCM)
```

Display connection matrix information when hitting return in REPL.

source

Chapter 10

Lefschetz Complex Functions

10.1 Simplicial Complexes

`ConleyDynamics.create_simplicial_complex` – Function.

```
create_simplicial_complex(labels::Vector{String},
                          simplices::Vector{Vector{Int}};
                          p::Int=2)
```

Initialize a Lefschetz complex from a simplicial complex. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The vector `labels` contains a label for every vertex, while `simplices` contains all the highest-dimensional simplices necessary to define the simplicial complex. Every simplex is represented as a vector of `Int`, with entries corresponding to the vertex indices.

Warning

Note that the labels all have to have the same character length!

source

```
create_simplicial_complex(labels::Vector{String},
                          simplices::Vector{Vector{String}};
                          p::Int=2)
```

Initialize a Lefschetz complex from a simplicial complex. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The vector `labels` contains a label for every vertex, while `simplices` contains all the highest-dimensional simplices necessary to define the simplicial complex.

source

`ConleyDynamics.create_simplicial_rectangle` – Function.

```
create_simplicial_rectangle(nx::Int, ny::Int; p::Int=2)
```

Create a simplicial complex covering a rectangle in the plane. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The rectangle is given by the subset `[0,nx] x [0,ny]` of the plane. Each unit square is represented by four triangles, which meet in the center point of the square. Labels have the following meaning:

- The label XXXYYYb corresponds to the point `(XXX, YYY)`.
- The label XXXYYYc corresponds to `(XXX + 1/2, YYY + 1/2)`.

The number of characters in XXX and YYY matches the number of digits of the larger number of `nx` and `ny`. The function returns the following objects:

- A simplicial complex `sc::LefschetzComplex`.
- A vector `coords::Vector{Vector{Float64}}` of vertex coordinates.

source

`ConleyDynamics.create_simplicial_delaunay` – Function.

```
create_simplicial_delaunay(boxw::Real, boxh::Real, pdist::Real, attmpt::Int;
                           p::Int=2)
```

Create a planar Delaunay triangulation inside a box. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The function selects a random sample of points inside the rectangular box `[0,boxw] x [0,boxh]`, while trying to maintain a minimum distance of `pdist` between the points. The argument `attmpt` specifies the number of attempts when trying to add points. A standard value is 20, and larger values tend to fill holes better, but at the expense of runtime. From the random sample, the function then creates a Delaunay triangulation, and returns the following objects:

- A simplicial complex `sc::LefschetzComplex`.
- A vector `coords::Vector{Vector{Float64}}` of vertex coordinates.

Note that the function does not provide a full triangulation of the given rectangle. Close to the boundary there will be gaps.

source

```
create_simplicial_delaunay(boxw::Real, boxh::Real, npoints::Int;
                           p::Int=2)
```

Create a planar Delaunay triangulation inside a box. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The function selects a random sample of `npoints` points inside the rectangular box `[0,boxw] x [0,boxh]`. From the random sample, the function then creates a Delaunay triangulation, and returns the following objects:

- A simplicial complex `sc::LefschetzComplex`.
- A vector `coords::Vector{Vector{Float64}}` of vertex coordinates.

Note that the function does not provide a full triangulation of the given rectangle. Close to the boundary there will be gaps.

source

`ConleyDynamics.simplicial_torus` – Function.

```
sc = simplicial_torus(p::Int)
```

Create a triangulation of the two-dimensional torus.

The function returns a simplicial complex which represents a two-dimensional torus. The argument `p` specifies the characteristic of the underlying field. This triangulation is taken from Figure 6.4 in Munkres' book on Algebraic Topology. The boundary vertices are labeled as letters as in the book, the five center vertices are labeled by 1 through 5.

Examples

```
julia> println(homology(simplicial_torus(0)))
[1, 2, 1]

julia> println(homology(simplicial_torus(2)))
[1, 2, 1]

julia> println(homology(simplicial_torus(3)))
[1, 2, 1]
```

source

`ConleyDynamics.simplicial_klein_bottle` – Function.

```
sc = simplicial_klein_bottle(p::Int)
```

Create a triangulation of the two-dimensional Klein bottle.

The function returns a simplicial complex which represents the two-dimensional Klein bottle. The argument `p` specifies the characteristic of the underlying field. This triangulation is taken from Figure 6.6 in Munkres' book on Algebraic Topology. The boundary vertices are labeled as letters as in the book, the five center vertices are labeled by 1 through 5.

Examples

```
julia> println(homology(simplicial_klein_bottle(0)))
[1, 1, 0]

julia> println(homology(simplicial_klein_bottle(2)))
[1, 2, 1]

julia> println(homology(simplicial_klein_bottle(3)))
[1, 1, 0]
```

source

ConleyDynamics.simplicial_projective_plane – Function.

```
sc = simplicial_projective_plane(p::Int)
```

Create a triangulation of the projective plane.

The function returns a simplicial complex which represents the projective plane. The argument p specifies the characteristic of the underlying field. This triangulation is taken from Figure 6.6 in Munkres' book on Algebraic Topology. The boundary vertices are labeled as letters as in the book, the five center vertices are labeled by 1 through 5.

Examples

```
julia> println(homology(simplicial_projective_plane(0)))
[1, 0, 0]

julia> println(homology(simplicial_projective_plane(2)))
[1, 1, 1]

julia> println(homology(simplicial_projective_plane(3)))
[1, 0, 0]
```

source

ConleyDynamics.simplicial_torsion_space – Function.

```
sc = simplicial_torsion_space(n::Int, p::Int)
```

Create a triangulation of a space with 1-dimensional torsion.

The function returns a simplicial complex which has the following integer homology groups:

- In dimension 0 it is the group of integers.
- In dimension 1 it is the integers modulo n.
- In dimension 2 it is the trivial group.

In other words, the simplicial complex has nontrivial torsion in dimension 1. It is a triangulation of an n-gon, in which all boundary edges are oriented counterclockwise, and all of these edges are identified. The parameter p specifies the characteristic of the underlying field.

Examples

```
julia> println(homology(simplicial_torsion_space(6,2)))
[1, 1, 1]

julia> println(homology(simplicial_torsion_space(6,3)))
[1, 1, 1]

julia> println(homology(simplicial_torsion_space(6,5)))
[1, 0, 0]
```

source

10.2 Cubical Complexes

`ConleyDynamics.create_cubical_complex` – Function.

```
create_cubical_complex(cubes::Vector{String}; p::Int=2)
```

Initialize a Lefschetz complex from a cubical complex. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The vector cubes contains a list of all the highest-dimensional cubes necessary to define the cubical complex. Every cube is represented as a string as follows:

- d integers, which correspond to the coordinates of a point in d-dimensional Euclidean space
- a point `.`
- d integers 0 or 1, which give the interval length in the respective dimension

The first d integers all have to occupy the same number of characters. In addition, if the occupied space is L characters for each coordinate, the coordinates only can take values from 0 to `10^L - 2`. This is due to the fact that the boundary operator will add one to certain coordinates, and they still need to be representable withing the same L digits.

For example, the string `030600.101` corresponds to the point `(3,6,0)` in three dimensions. The dimensions are 1, 0, and 1, and therefore this string corresponds to the cube `[3,4] x [6] x [0,1]`. The same cube could have also been represented by `360.101` or by `003006000.101`.

Warning

Note that the labels all have to have the same format!

Example

```
julia> cubes = ["00.11", "01.01", "02.10", "11.10", "11.01", "22.00"];

julia> lc = create_cubical_complex(cubes);

julia> lc.ncells
17

julia> homology(lc)
3-element Vector{Int64}:
 2
 1
 0
```

source

`ConleyDynamics.create_cubical_rectangle` – Function.

```
create_cubical_rectangle(nx::Int, ny::Int;
                         p::Int=2, randomize::Real=0.0)
```

Create a cubical complex covering a rectangle in the plane. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The rectangle is given by the subset `[0,nx] x [0,ny]` of the plane, and each unit square gives a two-dimensional cube in the resulting cubical complex. The function returns the following objects:

- A cubical complex `cc::LefschetzComplex`
- A vector `coords::Vector{Vector{Float64}}` of vertex coordinates

If the optional parameter `randomize` is assigned a positive real fraction `r` less that 0.5, then the actual coordinates will be randomized. They are chosen uniformly from discs of radius `r` centered at each vertex.

source

`ConleyDynamics.create_cubical_box` – Function.

```
create_cubical_box(nx::Int, ny::Int, nz::Int;
                   p::Int=2, randomize::Real=0.0)
```

Create a cubical complex covering a box in space. The complex is over the rationals if `p=0`, and over `GF(p)` if `p>0`.

The box is given by the subset `[0,nx] x [0,ny] x [0,nz]` of space, and each unit cube gives a three-dimensional cube in the resulting cubical complex. The function returns the following objects:

- A cubical complex `cc::LefschetzComplex`
- A vector `coords::Vector{Vector{Float64}}` of vertex coordinates

If the optional parameter `randomize` is assigned a positive real fraction `r` less that 0.5, then the actual coordinates will be randomized. They are chosen uniformly from balls of radius `r` centered at each vertex.

source

`ConleyDynamics.cube_field_size` – Function.

```
cube_field_size(cube::String)
```

Determine the field sizes of a given cube label.

The function returns the dimension of the ambient space in the first output parameter `pointdim`, and the length of the individual coordinate fields in the second return variable `pointlen`.

Example

```
julia> cube_field_size("011654003020.0110")
(4, 3)
```

source

`ConleyDynamics.cube_information` – Function.

```
cube_information(cube::String)
```

Compute a cube's coordinate information.

The function returns an integer vector with the cubes coordinate information. The return vector `intinfo` contains in its components the following data:

- `1:pointdim`: Coordinates of the anchor point
- `1+pointdim:2*pointdim`: Interval length in each dimension
- `1+2*pointdim`: Dimension of the cube

Note that `pointdim` equals the dimension of the points specifying the cube.

Example

```
julia> cube_information("011654003.011")
7-element Vector{Int64}:
  11
 654
   3
   0
   1
   1
   2
```

source

`ConleyDynamics.cube_label` – Function.

```
cube_label(pointdim::Int, pointlen::Int, pointinfo::Vector{Int})
```

Create a label from a cube's coordinate information.

The dimension of the ambient Eucliden space is `pointdim`, while the field length for each coordinate is `pointlen`. The vector `pointinfo` has to be of length at least two times `pointdim`. The first `pointdim` entries contain the coordinates of the anchor point, while the next `pointdim` entries are either 0 or 1 depending on the size of the interval. For example, if `poindim = 3` and `pointinfo = [1,2,3,1,0,1]`, then we represent the cube in three-dimensional space given by `[1,2] x [2] x [3 4]`.

Example

```
julia> cube_label(3,2,[10,23,5,1,1,0])
"102305.110"
```

source

`ConleyDynamics.get_cubical_coords` – Function.

```
get_cubical_coords(cc::LefschetzComplex)
```

Compute the vertex coordinates for a cubical complex.

The variable cc has to contain a cubical complex, and the function returns a vector of coordinates for the vertices of the complex, that can then be used for plotting. “‘

source

10.3 Lefschetz Complex Creation

ConleyDynamics.create_lefschetz_gf2 – Function.

```
create_lefschetz_gf2(defcellbnd)
```

Create a Lefschetz complex over GF(2) by specifying its essential cells and boundaries.

The input argument defcellbnd has to be a vector of vectors. Each entry defcellbnd[k] has to be of one of the following two forms:

- [String, Int, String, String, ...]: The first String contains the label for the cell k, followed by its dimension in the second entry. The remaining entries are for the labels of the cells which make up the boundary.
- [String, Int]: This shorther form is for cells with empty boundary. The first entry denotes the cell label, and the second its dimension.

The cells of the resulting Lefschetz complex correspond to the union of all occurring labels. Cell labels that only occur in the boundary specification are assumed to have empty boundary, and they do not have to be specified separately in the second form above. However, if their boundary is not empty, they have to be listed via the above first form as well.

Examples

```
julia> defcellbnd = [["A",0], ["a",1,"B","C"], ["b",1,"B","C"]];

julia> push!(defcellbnd, ["c",1,"B","C"]);

julia> push!(defcellbnd, ["alpha",2,"b","c"]);

julia> lc = create_lefschetz_gf2(defcellbnd);

julia> lc.labels
7-element Vector{String}:
 "A"
 "B"
 "C"
 "a"
 "b"
 "c"
 "alpha"
```

```
julia> homology(lc)
3-element Vector{Int64}:
 2
 1
 0
```

source

ConleyDynamics.lefschetz_subcomplex – Function.

```
lefschetz_subcomplex(lc::LefschetzComplex, subcomp::Vector{Int})
```

Extract a subcomplex from a Lefschetz complex. The subcomplex has to be locally closed, and is given by the collection of cells in subcomp.

source

```
lefschetz_subcomplex(lc::LefschetzComplex, subcomp::Vector{String})
```

Extract a subcomplex from a Lefschetz complex. The subcomplex has to be locally closed, and is given by the collection of cells in subcomp.

source

ConleyDynamics.lefschetz_closed_subcomplex – Function.

```
lefschetz_closed_subcomplex(lc::LefschetzComplex, subcomp::Vector{Int})
```

Extract a closed subcomplex from a Lefschetz complex. The subcomplex is the closure of the collection of cells given in subcomp.

source

```
lefschetz_closed_subcomplex(lc::LefschetzComplex, subcomp::Vector{String})
```

Extract a closed subcomplex from a Lefschetz complex. The subcomplex is the closure of the collection of cells given in subcomp.

source

ConleyDynamics.lefschetz_reduction – Function.

```
lefschetz_reduction(lc::LefschetzComplex, redpairs::Vector{Vector{Int}})
```

Apply a sequence of elementary reductions to a Lefschetz complex.

The reduction pairs have to be specified in the argument redpairs. Each entry has to be a vector of length two which contains an elementary reduction pair in index form. In particular, the dimensions of the two

cells in the pair have to differ by one, and once the pair is reached in the reduction sequence, one cell has to be a face of the other. The function returns a new Lefschetz complex, where all cells in `redpairs` have been removed.

source

```
lefschetz_reduction(lc::LefschetzComplex, redpairs::Vector{Vector{String}})
```

Apply a sequence of elementary reductions to a Lefschetz complex.

The reduction pairs have to be specified in the argument `redpairs`. Each entry has to be a vector of length two which contains an elementary reduction pair in label form. In particular, the dimensions of the two cells in the pair have to differ by one, and once the pair is reached in the reduction sequence, one cell has to be a face of the other. The function returns a new Lefschetz complex, where all cells in `redpairs` have been removed.

source

```
lefschetz_reduction(lc::LefschetzComplex, r1::Int, r2::Int)
```

Apply a single elementary reduction to a Lefschetz complex.

This method expects that the two cells `r1` and `r2` which form the reduction pair are given in index form. The function returns the reduced Lefschetz complex.

source

```
lefschetz_reduction(lc::LefschetzComplex, r1::String, r2::String)
```

Apply a single elementary reduction to a Lefschetz complex.

This method expects that the two cells `r1` and `r2` which form the reduction pair are given in label form. The function returns the reduced Lefschetz complex.

source

`ConleyDynamics.lefschetz_reduction_maps` – Function.

```
lefschetz_reduction_maps(lc::LefschetzComplex, redpairs::Vector{Vector{Int}})
```

Apply a sequence of elementary reductions to a Lefschetz complex and return the associated chain maps.

The reduction pairs have to be specified in the argument `redpairs`. Each entry has to be a vector of length two which contains an elementary reduction pair in index form. In particular, the dimensions of the two cells in the pair have to differ by one, and once the pair is reached in the reduction sequence, one cell has to be a face of the other. The function returns a new Lefschetz complex, where all cells in `redpairs` have been removed, as well as the associated chain maps.

The return values are as follows:

- `lcred`: The first variable contains the reduced Lefschetz complex.
- `pp`: This is a sparse matrix representation of the chain equivalence between the original complex and the reduced one.

- `jj`: This is a sparse matrix representation of the chain equivalence between the reduced complex and the original one.
- hh: This is a sparse matrix representation of the chain homotopy which shows that the composition `jj * pp` is chain homotopic to the identity.

Examples

```
julia> labels = ["a","b","c","d"];

julia> simplices = [["a","b"], ["b","c"], ["c","d"]];

julia> sc = create_simplicial_complex(labels, simplices, p=0);

julia> redpairs = [["b", "bc"], ["d", "cd"]];

julia> scr, pp, jj, hh = lefschetz_reduction_maps(sc,redpairs);

julia> bnd = deepcopy(sc.boundary);

julia> bndr = deepcopy(scr.boundary);

julia> ii = sparse_identity(sc.ncells, p=0);

julia> sparse_nz_count(pp*bnd - bndr*pp)
0

julia> sparse_nz_count(jj*bndr - bnd*jj)
0

julia> full_from_sparse(pp*jj)
3×3 Matrix{Rational{Int64}}:
 1  0  0
 0  1  0
 0  0  1

julia> full_from_sparse(jj*pp)
7×7 Matrix{Rational{Int64}}:
 1  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  1  1  1  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  1  0  0
 0  0  0  0  1  0  0
 0  0  0  0  0  0  0

julia> sparse_nz_count(ii + bnd*hh + hh*bnd - jj*pp)
0
```

source

```
lefschetz_reduction_maps(lc::LefschetzComplex, redpairs::Vector{Vector{String}})
```

Apply a sequence of elementary reductions to a Lefschetz complex and return the chain maps.

The reduction pairs have to be specified in the argument `redpairs`. Each entry has to be a vector of length two which contains an elementary reduction pair in label form. In particular, the dimensions of the two cells in the pair have to differ by one, and once the pair is reached in the reduction sequence, one cell has to be a face of the other. The function returns a new Lefschetz complex, where all cells in `redpairs` have been removed, as well as the involved chain maps.

source

```
lefschetz_reduction_maps(lc::LefschetzComplex, r1::Int, r2::Int)
```

Apply a single elementary reduction to a Lefschetz complex and return the chain maps.

This method expects that the two cells `r1` and `r2` which form the reduction pair are given in index form. The function returns the reduced Lefschetz complex, as well as the involved chain maps.

source

```
lefschetz_reduction_maps(lc::LefschetzComplex, r1::String, r2::String)
```

Apply a single elementary reduction to a Lefschetz complex and return the chain maps.

This method expects that the two cells `r1` and `r2` which form the reduction pair are given in label form. The function returns the reduced Lefschetz complex, as well as the involved chain maps.

source

`ConleyDynamics.lefschetz_newbasis` – Function.

```
lefschetz_newbasis(lc::LefschetzComplex, basis::SparseMatrix; maps::Bool=false)
```

Create a new Lefschetz complex via change of basis.

The new basis has to be specified in the sparse matrix `basis`, whose columns represent the new basis in terms of the existing one. This matrix has to respect the grading by dimension, i.e., the cells which are used to form a new basis chain have to have the same dimensions as the cell which is being replaced. The function returns the new Lefschetz complex `lcnew`. If the optional parameter `maps = true` is passed, the the function also returns the chain maps `pp` and `jj` which are the isomorphisms from `lc` to `lcnew`, and vice versa, as well as the zero chain homotopy `hh`.

source

`ConleyDynamics.lefschetz_newbasis_maps` – Function.

```
lefschetz_newbasis_maps(lc::LefschetzComplex, basis::SparseMatrix)
```

Create a new Lefschetz complex via change of basis and return the associated chain maps.

The new basis has to be specified in the sparse matrix `basis`, whose columns represent the new basis in terms of the existing one. This matrix has to respect the grading by dimension, i.e., the cells which are used to form a new basis chain have to have the same dimensions as the cell which is being replaced. The function returns the new Lefschetz complex `lcnew`, as well as the chain maps `pp` and `jj` which are the isomorphisms from `lc` to `lcnew`, and vice versa.

source

`ConleyDynamics.compose_reductions` – Function.

```
compose_reductions(pp1::SparseMatrix, jj1::SparseMatrix, hh1::SparseMatrix,
                   pp2::SparseMatrix, jj2::SparseMatrix, hh2::SparseMatrix)
```

Combine the chain maps and chain homotopies of two reductions.

The function expects the chain maps and chain homotopies of two Lefschetz complex reductions, and computes the chain maps and chain homotopy for the composition of both. The maps `ppX`, `jjX`, and `hhX` can be obtained via either `lefschetz_reduction_maps` or `lefschetz_newbasis_maps`.

source

`ConleyDynamics.permute_lefschetz_complex` – Function.

```
permute_lefschetz_complex(lc::LefschetzComplex,
                          permutation::Vector{Int})
```

Permute the indices of a Lefschetz complex.

The vector `permutation` contains a permutation of the indices for the given Lefschetz complex `lc`. If no permutation is specified, or if the length of the vector is not correct, then a randomly generated one will be used. Note that the permutation has to respect the ordering of the cells by dimension, otherwise an error is raised. In other words, the permutation has to decompose into permutations within each dimension. This is automatically done if no permutation is explicitly specified.

source

10.4 Lefschetz Complex Queries

`ConleyDynamics.lefschetz_information` – Function.

```
lefschetz_information(lc::LefschetzComplex)
```

Extract basic information about a Lefschetz complex.

The input argument `lc` contains the Lefschetz complex. The function returns the information in the form of a `Dict{String,Any}`. You can use the command `keys` to see the keyset of the return dictionary:

- `"Dimension"`: Dimension of the Lefschetz complex
- `"Coefficient field"`: Underlying coefficient field
- `"Euler characteristic"`: Euler characteristic of the complex
- `"Homology"`: Betti numbers of the Lefschetz complex
- `"Boundary sparsity"`: Sparsity percentage of the boundary matrix
- `"Number of cells"`: Total number of cells in the complex
- `"Cell counts by dim"`: Cell counts by dimension

In the last case, the dictionary entry is a vector of pairs `(dimension, cell count)`.

source

`ConleyDynamics.lefschetz_cell_count` – Function.

```
lefschetz_cell_count(lc::LefschetzComplex; bounds::Bool=false)
```

Returns the number of cells in each dimension.

The function returns the number of cells in each dimension. The return variable is of type `Vector{Int}`, has length `lc.dim + 1`, and its k-th entry contains the number of cells in dimension k-1. If the optional parameter `bounds=true` is passed, then the function also returns two integer vectors `lo` and `hi`. These contain the beginning and end indices of the cells in each dimension.

source

`ConleyDynamics.lefschetz_field` – Function.

```
fieldstr = lefschetz_field(lc::LefschetzComplex)
```

Returns the Lefschetz complex coefficient field.

source

`ConleyDynamics.lefschetz_is_closed` – Function.

```
lefschetz_is_closed(lc::LefschetzComplex, subcomp::Vector{Int})
```

Determine whether a Lefschetz complex subset is closed.

source

```
lefschetz_is_closed(lc::LefschetzComplex, subcomp::Vector{String})
```

Determine whether a Lefschetz complex subset is closed.

source

`ConleyDynamics.lefschetz_is_locally_closed` – Function.

```
lefschetz_is_locally_closed(lc::LefschetzComplex, subcomp::Vector{Int})
```

Determine whether a Lefschetz complex subset is locally closed.

source

```
lefschetz_is_locally_closed(lc::LefschetzComplex, subcomp::Vector{String})
```

Determine whether a Lefschetz complex subset is locally closed.

source

10.5 Topological Features

`ConleyDynamics.lefschetz_boundary` – Function.

```
lefschetz_boundary(lc::LefschetzComplex, cellI::Int)
```

Compute the support of the boundary of a Lefschetz complex cell.

This method returns the boundary support as a `Vector{Int}`.

source

```
lefschetz_boundary(lc::LefschetzComplex, cellS::String)
```

Compute the support of the boundary of a Lefschetz complex cell.

This method returns the boundary support as a `Vector{String}`.

source

`ConleyDynamics.lefschetz_coboundary` – Function.

```
lefschetz_coboundary(lc::LefschetzComplex, cellI::Int)
```

Compute the support of the coboundary of a Lefschetz complex cell.

This method returns the boundary support as a `Vector{Int}`.

source

```
lefschetz_coboundary(lc::LefschetzComplex, cellS::String)
```

Compute the support of the coboundary of a Lefschetz complex cell.

This method returns the boundary support as a `Vector{String}`.

source

`ConleyDynamics.lefschetz_closure` – Function.

```
lefschetz_closure(lc::LefschetzComplex, subcomp::Vector{Int})
```

Compute the closure of a Lefschetz complex subset.

source

```
lefschetz_closure(lc::LefschetzComplex, subcomp::Vector{String})
```

Compute the closure of a Lefschetz complex subset.

source

ConleyDynamics.lefschetz_interior – Function.

```
lefschetz_interior(lc::LefschetzComplex, subcomp::Vector{Int})
```

Compute the interior of a Lefschetz complex subset.

source

```
lefschetz_interior(lc::LefschetzComplex, subcomp::Vector{String})
```

Compute the interior of a Lefschetz complex subset.

source

ConleyDynamics.lefschetz_topboundary – Function.

```
lefschetz_topboundary(lc::LefschetzComplex, subcomp::Vector{Int})
```

Compute the topological boundary of a Lefschetz complex subset.

In contrast to the algebraic boundary defined via the boundary operator, this function computes the boundary of the Lefschetz complex subset specified in `subcomp` if the Lefschetz complex is interpreted as a finite topological space. In other words, the topological boundary is the set difference of the closure and the interior of the subset.

source

```
lefschetz_topboundary(lc::LefschetzComplex, subcomp::Vector{String})
```

Compute the topological boundary of a Lefschetz complex subset.

In contrast to the algebraic boundary defined via the boundary operator, this function computes the boundary of the Lefschetz complex subset specified in `subcomp` if the Lefschetz complex is interpreted as a finite topological space. In other words, the topological boundary is the set difference of the closure and the interior of the subset.

source

ConleyDynamics.lefschetz_openhull – Function.

```
lefschetz_openhull(lc::LefschetzComplex, subcomp::Vector{Int})
```

Compute the open hull of a Lefschetz complex subset.

source

```
lefschetz_openhull(lc::LefschetzComplex, subcomp::Vector{String})
```

Compute the open hull of a Lefschetz complex subset.

source

`ConleyDynamics.lefschetz_lchull` – Function.

```
lefschetz_lchull(lc::LefschetzComplex, subcomp::Vector{Int})
```

Compute the locally closed hull of a Lefschetz complex subset.

The locally closed hull is the smallest locally closed set which contains the given cells. It is the intersection of the closure and the open hull.

source

```
lefschetz_lchull(lc::LefschetzComplex, subcomp::Vector{String})
```

Compute the locally closed hull of a Lefschetz complex subset.

The locally closed hull is the smallest locally closed set which contains the given cells. It is the intersection of the closure and the open hull.

source

`ConleyDynamics.lefschetz_clomo_pair` – Function.

```
lefschetz_clomopair(lc::LefschetzComplex, subcomp::Vector{Int})
```

Determine the closure-mouth-pair associated with a Lefschetz complex subset.

The function returns the pair `(closure,mouth)`.

source

```
lefschetz_clomopair(lc::LefschetzComplex, subcomp::Vector{String})
```

Determine the closure-mouth-pair associated with a Lefschetz complex subset.

The function returns the pair `(closure,mouth)`.

source

`ConleyDynamics.lefschetz_skeleton` – Function.

```
lefschetz_skeleton(lc::LefschetzComplex, subcomp::Vector{Int}, skdim::Int)
```

Compute the `skdim`-dimensional skeleton of a Lefschetz complex subset.

The computed skeleton is for the closure of the subcomplex given by `subcomp`.

source

```
lefschetz_skeleton(lc::LefschetzComplex, subcomp::Vector{String}, skdim::Int)
```

Compute the `skdim`-dimensional skeleton of a Lefschetz complex subset.

The computed skeleton is for the closure of the subcomplex given by `subcomp`.

source

```
lefschetz_skeleton(lc::LefschetzComplex, skdim::Int)
```

Compute the `skdim`-dimensional skeleton of a Lefschetz complex.

The computed skeleton is for the full Lefschetz complex.

source

`ConleyDynamics.manifold_boundary` – Function.

```
manifold_boundary(lc::LefschetzComplex)
```

Extract the manifold boundary from a Lefschetz complex.

The function expects a Lefschetz complex which represents a compact d-dimensional manifold with boundary. It returns a list of all cells which lie on the topological boundary of the manifold, in the form of a `Vector{Int}`.

source

10.6 Filters on Lefschetz Complexes

`ConleyDynamics.create_random_filter` – Function.

```
create_random_filter(lc::LefschetzComplex)
```

Create a random injective filter on a Lefschetz complex.

The function creates a random injective filter on a Lefschetz complex. The filter is created by assigning integers to cell groups, increasing with dimension. Within each dimension the assignment is random, but all filter values of cells of dimension k are less than all filter values of cells with dimension k+1. The function returns the filter as `Vector{Int}`, with indices corresponding to the cell indices in the Lefschetz complex.

source

`ConleyDynamics.filter_shallow_pairs` – Function.

```
filter_shallow_pairs(lc::LefschetzComplex, phi)
```

Find all shallow pairs for a filter.

This function finds all shallow pairs for the filter `phi`. These are face-coface pairs `(x,y)` whose dimensions differ by one, and such that `y` has the smallest filter value on the coboundary of `x`, and `x` has the largest filter value on the boundary of `y`.

If the filter is injective, these pairs give rise to a Forman vector field on the underlying Lefschetz complex. For noninjective filters this is not true in general.

source

`ConleyDynamics.filter_induced_mvf` – Function.

```
filter_induced_mvf(lc::LefschetzComplex, phi)
```

Compute the multivector field induced by a filter.

This function returns the smallest multivector field which has the property that every shallow pair is contained in a multivector. For injective filters this is a Forman vector field, but in the noninjective case it can be a general multivector field.

source

`ConleyDynamics.lefschetz_filtration` – Function.

```
lefschetz_filtration(lc::LefschetzComplex, fvalues::Vector{Int})
```

Compute a filtration on a Lefschetz subset.

The considered Lefschetz complex is given in `lc`. The vector `fvalues` assigns an integer between 0 and N to every cell in `lc`. For every k the complex L_k is given by the closure of all cells with values between 1 and k. The function returns the following variables:

- `lcsub`: The subcomplex L_N
- `fvalsub`: The filtration on the subcomplex with values 1,...,N

Example

```
julia> labels = ["A","B","C","D","E","F","G"];

julia> simplices = [["A","B","D"],["B","D","E"],["B","C","E"],["C","E","F"],["F","G"]];

julia> sc = create_simplicial_complex(labels,simplices);

julia> filtration = [0,0,0,0,0,0,0,1,1,0,1,2,0,4,2,4,0,5,3,7,6];

julia> lcsub, fvalsub = lefschetz_filtration(sc,filtration);

julia> phinf, phint = persistent_homology(lcsub,fvalsub);

julia> phinf
3-element Vector{Vector{Int64}}:
 [1]
 []
 []

julia> phint
3-element Vector{Vector{Tuple{Int64, Int64}}}:
 []
 [(1, 5), (2, 7), (4, 6)]
 []
```

source

```
lefschetz_filtration(lc::LefschetzComplex, strfilt::Vector{Vector{String}})
```

Compute a filtration on a Lefschetz subset.

The considered Lefschetz complex is given in `lc`. The vector of string vectors `strfilt` contains the necessary simplices to build the filtration. The list `strfilt[k]` contains the simplices that are added at the k-th step, together with their closures. Thus, for every k the complex L_k is given by the closure of all cells listed in `strfilt[i]` for `i` between 1 and k. The function returns the following variables:

- `lcsub`: The subcomplex L_N, where `N = length(strfilt)`
- `fvalsub`: The filtration on the subcomplex with values 1,...,N

Example

```
julia> labels = ["A","B","C","D","E","F","G"];

julia> simplices = [["A","B","D"],["B","D","E"],["B","C","E"],["C","E","F"],["F","G"]];

julia> sc = create_simplicial_complex(labels,simplices);

julia> strfiltration =
↪  [["AB","AD","BD"],["BE","DE"],["BCE"],["CF","EF"],["ABD"],["CEF"],["BDE"]];

julia> lcsub, fvalsub = lefschetz_filtration(sc, strfiltration);

julia> phinf, phint = persistent_homology(lcsub,fvalsub);

julia> phinf
3-element Vector{Vector{Int64}}:
 [1]
 []
 []

julia> phint
3-element Vector{Vector{Tuple{Int64, Int64}}}:
 []
 [(1, 5), (2, 7), (4, 6)]
 []
```

source

`ConleyDynamics.lefschetz_filtration_mvf` – Function.

```
lefschetz_filtration_mvf(lc::LefschetzComplex, fvalues::Vector{Int})
```

Compute the multivector field associated with a Lefschetz complex filtration.

For the Lefschetz complex `lc`, and the filtration `fvalues` given on the Lefschetz complex, this function determines the multivector field generated by the filtration in the following way. For every filtration value k, the cells which are assigned this value form a locally closed set, which can be decomposed into connected components that are all locally closed. The union of all these connected components, as k ranges from 1

to N, forms the multivector field mvf that is returned by the function. The filtration should be generated by the function lefschetz_filtration.

source

10.7 Lefschetz Helper Functions

ConleyDynamics.lefschetz_gfp_conversion – Function.

```
lcgfp = lefschetz_gfp_conversion(lc::LefschetzComplex, p::Int)
```

Convert a Lefschetz complex to the same complex over a finite field.

It is expected that the boundary matrix of the given Lefschetz complex lc is defined over the rationals, and that the target characteristic p is a prime.

source

10.8 Cell Subset Helper Functions

ConleyDynamics.convert_cells – Function.

```
convert_cells(lc::LefschetzComplex, cl::Vector{Int})
```

Convert cell list cl in the Lefschetz complex lc from index form to label form.

source

```
convert_cells(lc::LefschetzComplex, cl::Vector{String})
```

Convert cell list cl in the Lefschetz complex lc from label form to index form.

source

ConleyDynamics.convert_cellsubsets – Function.

```
convert_cellsubsets(lc::LefschetzComplex, clsub::Vector{Vector{Int}})
```

Convert CellSubsets clsub in the Lefschetz complex lc from index form to label form.

source

```
convert_cellsubsets(lc::LefschetzComplex, clsub::Vector{Vector{String}})
```

Convert CellSubsets clsub in the Lefschetz complex lc from label form to index form.

source

ConleyDynamics.cellsubset_bounding_box – Function.

```
cellsubset_bounding_box(lc, coords, csubset)
```

Compute the bounding box for a cell subset.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
this function computes the smallest enclosing box for the closure of the cell subset given in `csubset::Cells`.
The function returns the coordinates `bmin` and `bmax` of the minimal and maximal corners of the box, respectively.

source

`ConleyDynamics.cellsubset_distance` – Function.

```
cellsubset_distance(lc, coords, csubset, dpoint)
```

Compute the distance of a cell subset from a point.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
this function computes the smallest distance of the vertices in the closure of the cell subset given in `csubset::Cells` from the point given in `dpoint::Vector{<:Real}`.

source

`ConleyDynamics.cellsubset_planar_area` – Function.

```
cellsubset_planar_area(lc, coords, csubset)
```

Compute the area of a planar cell subset.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
this function computes the area of the cell subset given in `csubset::Cells`. The function assumes that the complex is two-dimensional and that the maximal 2-cells in the cell subset are all polygonal with straight boundary edges. If these conditions are not met an error is raised.

source

`ConleyDynamics.locate_planar_cellsubsets` – Function.

```
locate_planar_cellsubsets(lc, coords, csubsets, rmin, rmax)
```

Locate cell subsets relative to a planar rectangle.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
and the cell subsets in `csubsets::CellSubsets` this function extracts the indices of all cell subset closures which lie in the interior, or intersect the boundary of the rectangle specified by the minimal and maximal corners `rmin::Vector{<:Real}` and `rmax::Vector{<:Real}`, respectively. The function returns

- `indexI::Vector{Int}`: indices of cell subset closures inside the rectangle,
- `indexB::Vector{Int}`: indices of cell subset closures which intersect the rectangle boundary.

source

```
locate_planar_cellsubsets(lc, coords, csubsets, c, r)
```

Locate cell subsets relative to a planar circle.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
and the cell subsets in `csubsets::CellSubsets` this function extracts the indices of all cell subset closures
which lie in the interior, or intersect the boundary of the circle specified by the center point `c::Vector{<:Real}`
and radius `r::Real`. The function returns

- `indexI::Vector{Int}`: indices of cell subset closures inside the circle,
- `indexB::Vector{Int}`: indices of cell subset closures which intersect the circle.

source

`ConleyDynamics.cellsubset_location_rectangle` – Function.

```
cellsubset_location_rectangle(lc, coords, csubset, rmin, rmax)
```

Determine the location of a cell subset relative to a rectangle.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
this function determines the location of the closure of the cellsubset given in `csubset::Cells` relative to the
rectangle specified by the minimal and maximal corners `rmin::Vector{<:Real}` and `rmax::Vector{<:Real}`,
respectively. The function returns

- 1 if the set lies in the interior of the rectangle,
- 2 of the set intersects the rectangle boundary, and
- 3 if the set lies in the exterior of the rectangle.

source

`ConleyDynamics.cellsubset_location_circle` – Function.

```
cellsubset_location_circle(lc, coords, csubset, c, r)
```

Determine the location of a cell subset relative to a circle.

For the Lefschetz complex `lc::LefschetzComplex`, whose vertices have the coordinates given in `coords::Vector{<:Vector{`
this function determines the location of the closure of the cellsubset given in `csubset::Cells` relative to
the circle specified by the center point `c::Vector{<:Real}` and the radius `r::Real`. The function returns

- 1 if the set lies in the interior of the circle,
- 2 of the set intersects the circle, and
- 3 if the set lies in the exterior of the circle.

source

10.9 Geometry Helper Functions

`ConleyDynamics.convert_planar_coordinates` – Function.

```
convert_planar_coordinates(coords::Vector{<:Vector{<:Real}},
                           p0::Vector{<:Real},
                           p1::Vector{<:Real})
```

Convert a given collection of planar coordinates.

The vector `coords` contains pairs of coordinates, which are then transformed to fit into the box with vertices `p0 = (p0x,p0y)` and `p1 = (p1x,p1y)`. It is assumed that `p0` denotes the lower left box corner, while `p1` is the upper right corner. The function shifts and scales the coordinates in such a way that every side of the box contains at least one point. Upon completion, it returns a new coordinate vector `coordsNew`.

More precisely, if the x-coordinates are spanning the interval `[xmin,xmax]` and the y-coordinates span `[ymin,ymax]`, then the point `(x,y)` is transformed to `(xn,yn)` with:

- `xn = p0x + (p1x-p0x) * (x-cxmin) / (cxmax-cxmin)`
- `yn = p0y + (p1y-p0y) * (y-cymin) / (cymax-cymin)`

source

`ConleyDynamics.convert_spatial_coordinates` – Function.

```
convert_spatial_coordinates(coords::Vector{<:Vector{<:Real}},
                            p0::Vector{<:Real},
                            p1::Vector{<:Real})
```

Convert a given collection of spatial coordinates.

The vector `coords` contains triples of coordinates, which are then transformed to fit into the box with vertices `p0 = (p0x,p0y,p0z)` and `p1 = (p1x,p1y,p1z)`. It is assumed that each coordinate of `p0` is strictly smaller than the corresponding coordinate of `p1`. The function shifts and scales the coordinates in such a way that every side of the box contains at least one point. Upon completion, it returns a new coordinate vector `coordsNew`.

More precisely, if the x-coordinates are spanning the interval `[xmin,xmax]`, the y-coordinates span `[ymin,ymax]`, and the z-coordinates span `[zmin,zmax]`, then the point `(x,y,z)` is transformed to `(xn,yn,zn)` with:

- `xn = p0x + (p1x-p0x) * (x-cxmin) / (cxmax-cxmin)`
- `yn = p0y + (p1y-p0y) * (y-cymin) / (cymax-cymin)`
- `zn = p0z + (p1z-p0z) * (z-czmin) / (czmax-czmin)`

source

`ConleyDynamics.signed_distance_rectangle` – Function.

```
signed_distance_rectangle(p, rmin, rmax)
```

Compute the signed distance from a point to a rectangle.

The point in question is given as `p::Vector{<:Real}`. The rectangle has to be parallel to the coordinate axes and is given via its lower left corner `rmin::Vector{<:Real}` and its upper right corner `rmax::Vector{<:Real}`. The function returns the signed distance, which is negative inside the rectangle, and positive outside.

source

`ConleyDynamics.signed_distance_circle` – Function.

```
signed_distance_circle(p, c, r)
```

Compute the signed distance from a point to a circle.

The point is specified as `p::Vector{<:Real}`, while the circle is given by its center point `c::Vector{<:Real}` and radius `r::Real`. The function returns the signed distance, which is negative inside the circle, and positive outside.

source

`ConleyDynamics.segment_intersects_rectangle` – Function.

```
segment_intersects_rectangle(p1, p2, rmin, rmax)
```

Determine whether a line segment intersects a rectangle boundary.

The endpoints of the line segment are specified in the form `p1,p2::Vector{<:Real}`. The rectangle is parallel to the coordinate axes and is given via its lower left corner `rmin::Vector{<:Real}` and its upper right corner `rmax::Vector{<:Real}`. The function returns `true` if the line segment intersects the rectangle boundary, otherwise `false`.

source

`ConleyDynamics.segment_intersects_circle` – Function.

```
segment_intersects_circle(p1, p2, c, r)
```

Determine whether a line segment intersects a circle.

The endpoints of the line segment are specified in the form `p1,p2::Vector{<:Real}`, while the circle is given by its center point `c::Vector{<:Real}` and radius `r::Real`. The function returns `true` if the line segment intersects the circle, otherwise it returns `false`.

source

Chapter 11

Homology Functions

11.1 Regular Homology

ConleyDynamics.homology – Function.

```
homology(lc::LefschetzComplex)
```

Compute the homology of a Lefschetz complex.

The homology is returned as a vector `betti` of Betti numbers, where `betti[k]` is the Betti number in dimension `k-1`. The computations are performed over the field associated with the Lefschetz complex boundary matrix.

source

ConleyDynamics.relative_homology – Function.

```
relative_homology(lc::LefschetzComplex,subc::Cells)
```

Compute the relative homology of a Lefschetz complex with respect to a subcomplex. The computation is performed over the field associated with the Lefschetz boundary matrix.

The subcomplex is the closure of the cells in `subc`, which can be given either as indices or labels. The homology is returned as a vector `betti` of Betti numbers, where `betti[k]` is the Betti number in dimension `k-1`.

source

```
relative_homology(lc::LefschetzComplex,subc::Cells,subc0::Cells)
```

Compute the relative homology of a Lefschetz complex with respect to a subcomplex. The computation is performed over the field associated with the Lefschetz boundary matrix.

In this implementation, relative homology of the pair `cl(subc), cl(subc0))` is computed. An error is raised if `cl(subc0)` is not a subset of `cl(subc)`. The homology is returned as a vector `betti` of Betti numbers, where `betti[k]` is the Betti number in dimension `k-1`.

source

11.2 Persistent Homology

`ConleyDynamics.persistent_homology` – Function.

```
persistent_homology(lc::LefschetzComplex, filtration::Vector{Int})
```

Complete the persistent homology of a Lefschetz complex filtration over the field associated with the Lefschetz complex boundary matrix.

The function returns the two values

- `phsingles::Vector{Vector{Int}}`
- `phpairs::Vector{Vector{Tuple{Int,Int}}}`

It assumes that the order given by the filtration values is admissible, i.e., the permuted boundary matrix is strictly upper triangular. The function returns the starting filtration values for infinite length persistence intervals in `phsingles`, and the birth- and death-filtration values for finite length persistence intervals in `phpairs`.

source

11.3 Reduction Algorithm

`ConleyDynamics.ph_reduce!` – Function.

```
ph_reduce!(matrix::SparseMatrix; [returnbasis=true])
```

Apply the persistence reduction algorithm to the matrix.

The function returns the values

- `phsingles::Vector{Vector{Int}}`
- `phpairs::Vector{Vector{Tuple{Int,Int}}}`
- `basis::SparseMatrix` (if `returnbasis=true`)

It assumes that `matrix` is strictly upper triangular. The function returns the starting columns for infinite length persistence intervals in `phsingles`, and the birth- and death-columns for finite length persistence intervals in `phpairs`. If the optional argument `returnbasis=true` is given, then the function also returns the computed basis matrix B with `reduced = matrix * B`.

source

Chapter 12

Conley Theory Functions

12.1 Multivector Fields

`ConleyDynamics.mvf_information` – Function.

```
mvf_information(lc::LefschetzComplex, mvf::CellSubsets)
```

Extract basic information about a multivector field.

The input argument `lc` contains the Lefschetz complex, and `mvf` describes the multivector field. The function returns the information in the form of a `Dict{String,Any}`. You can use the command keys to see the keyset of the return dictionary:

- `"N mv"`: Number of multivectors
- `"N critical"`: Number of critcal multivectors
- `"N regular"`: Number of regular multivectors
- `"Lengths critical"`: Length distribution of critical multivectors
- `"Lengths regular"`: Length distribution of regular multivectors

In the last two cases, the dictionary entries are vectors of pairs `(length,frequency)`, where each pair indicates that there are `frequency` multivectors of length `length`.

source

`ConleyDynamics.create_mvf_hull` – Function.

```
create_mvf_hull(lc::LefschetzComplex, mvfbase::Vector{Vector{Int}})
```

Create the smallest multivector field containing the given sets.

The resulting multivector field has the property that every set of the form `mvfbase[k]` is contained in a minimal multivector. Notice that these sets do not have to be disjoint, and that not even their locally closed hulls have to be disjoint. In the latter case, this leads to two such sets having to be contained in the same multivector. If the sets in `mvfbase` are poorly chosen, one might end up with extremely large multivectors due to the above potential merging of locally closed hulls.

source

```
create_mvf_hull(lc::LefschetzComplex, mvfbase::Vector{Vector{String}})
```

Create the smallest multivector field containing the given sets.

The resulting multivector field has the property that every set of the form `mvfbase[k]` is contained in a minimal multivector. Notice that these sets do not have to be disjoint, and that not even their locally closed hulls have to be disjoint. In the latter case, this leads to two such sets having to be contained in the same multivector. If the sets in `mvfbase` are poorly chosen, one might end up with extremely large multivectors due to the above potential merging of locally closed hulls.

source

`ConleyDynamics.create_planar_mvf` – Function.

```
create_planar_mvf(lc::LefschetzComplex, coords::Vector{<:Vector{<:Real}}, vf)
```

Create a planar multivector field from a regular vector field.

The function expects a planar Lefschetz complex `lc` and a coordinate vector `coords` of coordinates for all the 0-dimensional cells in the complex. Moreover, the underlying vector field is specified by the function `vf(z::Vector{Float64})::Vector{Float64}`, where both the input and output vectors have length two. The function `create_planar_mvf` returns a multivector field `mvf` on `lc`, which can then be further analyzed using for example the function `connection_matrix`.

The input data `lc` and `coords` can be generated using one of the following methods:

- `create_cubical_rectangle`
- `create_simplicial_rectangle`
- `create_simplicial_delaunay`

In each case, the provided coordinate vector can be transformed to the correct bounding box values using the conversion function `convert_planar_coordinates`.

Example 1

Suppose we define a sample vector field using the commands

```
function samplevf(x::Vector{Float64})
    #
    # Sample vector field with nontrivial Morse decomposition
    #
    x1, x2 = x
    y1 = x1 * (1.0 - x1*x1 - 3.0*x2*x2)
    y2 = x2 * (1.0 - 3.0*x1*x1 - x2*x2)
    return [y1, y2]
end
```

One first creates a triangulation of the enclosing box, which in this case is given by `[-2,2] x [-2,2]` using the commands

```
n = 21
lc, coords = create_simplicial_rectangle(n,n);
coordsN = convert_planar_coordinates(coords,[-2.0,-2.0],[2.0,2.0]);
```

The multivector field is then generated using

```
mvf = create_planar_mvf(lc,coordsN,samplevf);
```

and the commands

```
cm = connection_matrix(lc, mvf);
cm.conley
full_from_sparse(cm.matrix)
```

finally show that this vector field gives rise to a Morse decomposition with nine Morse sets, and twelve connecting orbits. Using the commands

```
fname = "morse_test.pdf"
plot_planar_simplicial_morse(lc, coordsN, fname, cm.morse, pv=true)
```

these Morse sets can be visualized. The image will be saved in `fname`.

Example 2

An example with periodic orbits can be generated using the vector field

```
function samplevf2(x::Vector{Float64})
    #
    # Sample vector field with nontrivial Morse decomposition
    #
    x1, x2 = x
    c0 = x1*x1 + x2*x2
    c1 = (c0 - 4.0) * (c0 - 1.0)
    y1 = -x2 + x1 * c1
    y2 =  x1 + x2 * c1
    return [-y1, -y2]
end
```

The Morse decomposition can now be computed via

```
n2 = 51
lc2, coords2 = create_cubical_rectangle(n2,n2);
coords2N = convert_planar_coordinates(coords2,[-4.0,-4.0],[4.0,4.0]);
mvf2 = create_planar_mvf(lc2,coords2N,samplevf2);
cm2 = connection_matrix(lc2, mvf2);
cm2.conley
cm2.poset
full_from_sparse(cm2.matrix)

fname2 = "morse_test2.pdf"
plot_planar_cubical_morse(lc2, fname2, cm2.morse, pv=true)
```

In this case, one obtains three Morse sets: One is a stable equilibrium, one is an unstable periodic orbit, and the last is a stable periodic orbit.

source

ConleyDynamics.create_spatial_mvf – Function.

```
create_spatial_mvf(lc::LefschetzComplex, coords::Vector{<:Vector{<:Real}}, vf)
```

Create a spatial multivector field from a regular vector field.

The function expects a three-dimensional Lefschetz complex `lc` and a coordinate vector `coords` of coordinates for all the 0-dimensional cells in the complex. Moreover, the underlying vector field is specified by the function `vf(z::Vector{Float64})::Vector{Float64}`, where both the input and output vectors have length three. The function `create_spatial_mvf` returns a multivector field `mvf` on `lc`, which can then be further analyzed using for example the function `connection_matrix`.

The input data `lc` and `coords` has to be of one of the following two types:

- `lc` is a tetrahedral mesh of a region in three dimensions. In other words, the underlying Lefschetz complex is in fact a simplicial complex, and the vector `coords` contains the vertex coordinates.
- `lc` is a three-dimensional cubical complex, i.e., it is the closure of a collection of three-dimensional cubes in space. The vertex coordinates can be slightly perturbed from the original position in the cubical lattice, as long as the overall structure of the complex stays intact. In that case, the faces are interpreted as Bezier surfaces with straight edges.

Example 1

Suppose we define a sample vector field using the commands

```
function samplevf(x::Vector{Float64})
    #
    # Sample vector field with nontrivial Morse decomposition
    #
    x1, x2, x3 = x
    y1 = x1 * (1.0 - x1*x1)
    y2 = -x2
    y3 = -x3
    return [y1, y2, y3]
end
```

One first creates a cubical complex covering the interesting dynamics, say the trapping region `[-1.5,1.5] x [-1,1] x [-1,1]`, using the commands

```
lc, coords = create_cubical_box(3,3,3);
coordsN = convert_spatial_coordinates(coords,[-1.5,-1.0,-1.0],[1.5,1.0,1.0]);
```

The multivector field is then generated using

```
mvf = create_spatial_mvf(lc,coordsN,samplevf);
```

and the commands

```
cm = connection_matrix(lc, mvf);
cm.conley
full_from_sparse(cm.matrix)
```

finally show that this vector field gives rise to a Morse decomposition with three Morse sets, and two connecting orbits.

source

ConleyDynamics.extract_multivectors – Function.

```
extract_multivectors(lc::LefschetzComplex, mvf::Vector{Vector{Int}},
                     scells::Vector{Int})
```

Extract all multivectors containing a provided selection of cells.

The function returns all multivectors which contain at least one of the cells in the input vector `scells`. The return argument has type `Vector{Vector{Int}}`.

source

```
extract_multivectors(lc::LefschetzComplex, mvf::Vector{Vector{String}},
                     scells::Vector{String})
```

Extract all multivectors containing a provided selection of cells.

The function returns all multivectors which contain at least one of the cells in the input vector `scells`. The return argument has type `Vector{Vector{String}}`.

source

ConleyDynamics.planar_nontransverse_edges – Function.

```
planar_nontransverse_edges(lc::LefschetzComplex, coords::Vector{<:Vector{<:Real}}, vf;
                           npts::Int=100)
```

Find all edges of a planar Lefschetz complex which are not flow transverse.

The Lefschetz complex is given in `lc`, the coordinates of all vertices of the complex in `coords`, and the vector field is specified in `vf`. The optional parameter `npts` determines how many points along an edge are evaluated for the transversality check. The function returns a list of nontransverse edges as `Vector{Int}`, which contains the edge indices.

source

12.2 Conley Index Computations

ConleyDynamics.isoinvset_information – Function.

```
isoinvset_information(lc::LefschetzComplex, mvf::CellSubsets, iis::Cells)
```

Compute basic information about an isolated invariant set.

The input argument `lc` contains the Lefschetz complex, and `mvf` describes the multivector field. The isolated invariant set is specified in the argument `iis`. The function returns the information in the form of a `Dict{String,Any}`. The command `keys` can be used to see the keyset of the return dictionary. These describe the following information:

- `"Conley index"` contains the Conley index of the isolated invariant set.
- `"N multivectors"` contains the number of multivectors in the isolated invariant set.

source

`ConleyDynamics.conley_index` – Function.

```
conley_index(lc::LefschetzComplex, subcomp::Vector{String})
```

Determine the Conley index of a Lefschetz complex subset.

The function raises an error if the subset `subcomp` is not locally closed. The computations are performed over the field associated with the Lefschetz complex boundary matrix.

source

```
conley_index(lc::LefschetzComplex, subcomp::Vector{Int})
```

Determine the Conley index of a Lefschetz complex subset.

The function raises an error if the subset `subcomp` is not locally closed. The computations are performed over the field associated with the Lefschetz complex boundary matrix.

source

`ConleyDynamics.morse_sets` – Function.

```
morse_sets(lc::LefschetzComplex, mvf::CellSubsets; poset::Bool=false)
```

Find the nontrivial Morse sets of a multivector field on a Lefschetz complex.

The input argument `lc` contains the Lefschetz complex, and `mvf` describes the multivector field. The function returns the nontrivial Morse sets as a `Vector{Vector{Int}}`. If the optional argument `poset=true` is added, then the function returns both the Morse sets and the adjacency matrix of the Hasse diagram of the underlying poset.

source

`ConleyDynamics.morse_interval` – Function.

```
morse_interval(lc::LefschetzComplex, mvf::CellSubsets,
               ms::CellSubsets)
```

Find the isolated invariant set for a Morse set interval.

The input argument `lc` contains the Lefschetz complex, and `mvf` describes the multivector field. The collection of Morse sets are contained in`ms`. All of these sets should be Morse sets in the sense of being strongly connected components of the flow graph. (Nevertheless, this will be enforced in the function!) In other words, the sets in `ms` should be determined using the function `morse_sets`!

The function returns the smallest isolated invariant set which contains the Morse sets and their connections as a `Vector{Int}`.

source

`ConleyDynamics.restrict_dynamics` – Function.

```
restrict_dynamics(lc::LefschetzComplex, mvf::CellSubsets, lcsub::Cells)
```

Restrict a multivector field to a Lefschetz subcomplex.

For a given multivector field `mvf` on a Lefschetz complex `lc`, and a subcomplex which is given by the locally closed set represented by `lcsub`, create the associated Lefschetz subcomplex `lcreduced` and the induced multivector field `mvfreduced` on the subcomplex. The multivectors of the new multivector field are the intersections of the original multivectors and the subcomplex.

source

`ConleyDynamics.remove_exit_set` – Function.

```
remove_exit_set(lc::LefschetzComplex, mvf::CellSubsets)
```

Exit set removal for a multivector field on a Lefschetz subcomplex.

It is assumed that the Lefschetz complex `lc` is a topological manifold and that `mvf` contains a multivector field that is created via either `create_planar_mvf` or `create_spatial_mvf`. The function identifies cells on the boundary at which the flows exits the region covered by the Lefschetz complex. If this exit set is closed, we have found an isolated invariant set and the function returns a Lefschetz complex `lcr` restricted to it, as well as the restricted multivector field `mvfr`. If the exit set is not closed, a warning is displayed and the function returns the restricted Lefschetz complex and multivector field obtained by removing the closure of the exit set. In the latter case, unexpected results might be obtained.

source

12.3 Connection Matrix Computation

`ConleyDynamics.connection_matrix` – Function.

```
connection_matrix(lc::LefschetzComplex, mvf::CellSubsets;
                  [returnbasis::Bool])
```

Compute a connection matrix for the multivector field `mvf` on the Lefschetz complex `lc` over the field associated with the Lefschetz complex boundary matrix.

The function returns an object of type `ConleyMorseCM`. If the optional argument `returnbasis::Bool=true` is given, then the function also returns a dictionary which gives the basis for the connection matrix columns in terms of the original labels.

source

`ConleyDynamics.cm_reduce!` – Function.

```
cm_reduce!(matrix::SparseMatrix, psetvec::Vector{Int};
           [returnbasis::Bool],[returntm::Bool])
```

Compute the connection matrix.

Assumes that `matrix` is upper triangular and filtered according to `psetvec`. Modifies the argument `matrix`.

Return values:

- `cmatrix`: Connection matrix
- `cmatrix_cols`: Columns of the connection matrix in the boundary
- `basisvecs` (optional): If the argument `returnbasis=true` is given, this returns information about the computed basis. The k-th entry of `basisvecs` is a vector containing the columns making up the k-th basis vector, which corresponds to column `cmatrix_cols[k]`.
- `tmatrix` (optional): If the argument `returntm=true` is given in addition to `returnbasis=true`, then instead of `basisvecs` the function returns the complete transformation matrix. In this case, `basicvecs` is not returned.

source

12.4 Forman Vector Fields

`ConleyDynamics.forman_comb_flow` – Function.

```
forman_comb_flow(lc::LefschetzComplex, fvf::Vector{Vector{String}})
```

Compute Forman's combinatorial flow and the associated chain homotopy.

This function returns matrix representations of Forman's combinatorial flow `phi`, and of the associated chain homotopy `gamma`.

Example

```
julia> labels = ["a","b","c","d"];

julia> simplices = [["a","b"], ["b","c"], ["c","d"]];

julia> lc = create_simplicial_complex(labels, simplices, p=5);

julia> fvf = [["ab","b"], ["bc","c"]];
```

```
julia> phi, gamma = forman_comb_flow(lc, fvf);

julia> full_from_sparse(gamma)
7×7 Matrix{Int64}:
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  4  0  0  0  0  0
 0  0  4  0  0  0  0
 0  0  0  0  0  0  0

julia> full_from_sparse(phi)
7×7 Matrix{Int64}:
 1  1  0  0  0  0  0
 0  0  1  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  1  0  0  0
 0  0  0  0  0  1  0
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1

julia> full_from_sparse(phi*phi)
7×7 Matrix{Int64}:
 1  1  1  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  1  0  0  0
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1
```

source

```
forman_comb_flow(lc::LefschetzComplex, fvf::Vector{Vector{String}})
```

Compute Forman's combinatorial flow and the associated chain homotopy.

This function returns matrix representations of Forman's combinatorial flow `phi`, and of the associated chain homotopy `gamma`.

source

`ConleyDynamics.forman_stab_flow` – Function.

```
forman_stab_flow(lc::LefschetzComplex, fvf::CellSubsets; maxit::Int=25)
```

Compute Forman's stabilized combinatorial flow and the associated chain homotopy which relates it to the identity.

This function returns matrix representations of Forman's stabilized combinatorial flow `phiI`, and of the associated chain homotopy `gammaI`. The third return argument is a boolean flag which indicates whether

or not the combinatorial flow stabilized. If it did not, then either the underlying Forman vector field is not gradient (this is not checked!), or the maximal number of iterations has been reached. In the latter case, one has to pass the optional paramter `maxit` with a larger number of allowed iterations.

Example

```
julia> labels = ["a","b","c","d"];

julia> simplices = [["a","b"], ["b","c"], ["c","d"]];

julia> lc = create_simplicial_complex(labels, simplices, p=5);

julia> fvf = [["ab","b"], ["bc","c"]];

julia> phiI, gammaI, stabilized = forman_stab_flow(lc, fvf);

julia> stabilized
true

julia> full_from_sparse(gammaI)
7×7 Matrix{Int64}:
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  4  4  0  0  0  0
 0  0  4  0  0  0  0
 0  0  0  0  0  0  0

julia> full_from_sparse(phiI)
7×7 Matrix{Int64}:
 1  1  1  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  0  0  0  0
 0  0  0  1  0  0  0
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1
 0  0  0  0  0  0  1
```

source

`ConleyDynamics.forman_gpaths` – Function.

```
forman_gpaths(lc::LefschetzComplex, fvf::CellSubsets, x::Cell)
```

Find all Forman gradient paths starting at a source cell.

For the Forman gradient vector field `fvf` on the Lefschetz complex `lc` this function finds all maximal gradient paths starting at `x`. These are solution paths which consist exclusively of Forman vectors, i.e., they contain an even number of cells whose dimensions alternate between `dim(x)` and `dim(x) + 1`. Every cell of dimension `dim(x)` is an arrow source which is followed by its arrow target, while every cell of dimension `dim(x) + 1` is an arrow target which is succeeded by a cell in its boundary which is the source of a different arrow, as long as such a cell exists.

source

```
forman_gpaths(lc::LefschetzComplex, fvf::CellSubsets,
                  x::Cell, y::Cell)
```

Find all Forman gradient paths between two cells.

For the Forman gradient vector field `fvf` on the Lefschetz complex `lc` this function finds all gradient paths between the cells `x` and `y`. The dimensions of these cells have to satisfy one of the following three conditions:

- `dim(x) = dim(y) - 1`: In this case the function returns all solution paths between `x` and `y` which consist entirely of Forman arrows. All the sources have the dimension of `x`, and the targets the dimension of `y`.
- `dim(x) = dim(y)`: In this case the function returns all solution paths `p` between `x` and `y` for which `p[1:end-1]` is a Forman gradient path in the above sense, and `p[end]` lies in the boundary of `p[end-1]` and is different from the cell `p[end-2]`.
- `dim(x) = dim(y) + 1`: In this case the function returns all solution paths `p` between `x` and `y` for which `p[2:end-1]` is a Forman gradient path in the sense of the first item, where `p[2]` lies in the boundary of `p[1]`, and `p[end]` is contained in the boundary of `p[end-1]`.

In all other cases an empty collection is returned.

source

`ConleyDynamics.forman_path_weight` – Function.

```
forman_path_weight(lc::LefschetzComplex, path::Cells)
```

Compute the weight of a Forman gradient path.

For the Lefschetz complex `lc` this function computes the weight of the Forman gradient path given in `path`. It is expected that the dimensions of the first and the last cell in the path differ by at most 1. In case they have equal dimension, and the path has length larger than 1, the first cell has to be an arrow source.

source

```
forman_path_weight(lc::LefschetzComplex, paths::CellSubsets)
```

Accumulated weight of a collection of Forman gradient paths.

For the Lefschetz complex `lc` this function computes the sum of the weights of the collection of Forman gradient paths given in `paths`.

source

`ConleyDynamics.chain_vector` – Function.

```
chain_vector(lc::LefschetzComplex, chcells::Vector{Int})
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has a 1 in every cell location indicated by the entries in the input argument `chcells`.

source

```
chain_vector(lc::LefschetzComplex, chcells::Vector{String})
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has a 1 for every cell indicated by the labels listed in the input argument `chcells`.

source

```
chain_vector(lc::LefschetzComplex, chcell::Int)
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has a 1 at the cell index specified in the second argument.

source

```
chain_vector(lc::LefschetzComplex, chcell::String)
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has a 1 at the cell specified by the second argument.

source

```
chain_vector(lc::LefschetzComplex, chcells::Vector{Int}, chcoeff)
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has the entry `chcoeff[k]` in the `chcells[k]`-th row. In other words, it constructs the vector representation of the chain consisting of the cells specified in `chcells`, with coefficients as specified in `chcoeff`.

source

```
chain_vector(lc::LefschetzComplex, chcells::Vector{String}, chcoeff)
```

Create a sparse vector representing a chain.

This function returns a sparse matrix in the form of a column vector, which has the entry `chcoeff[k]` in the row corresponding to the cell with label `chcells[k]`. In other words, it constructs the vector representation of the chain consisting of the cells specified in `chcells`, with coefficients as specified in `chcoeff`.

source

```
chain_vector(lc::LefschetzComplex, chcell::Int, chcoeff)
```

Create a sparse vector representing a chain.

This short-cut method specifies a chain consiting of one cell and its coefficient. The cell is given as its index.

source

```
chain_vector(lc::LefschetzComplex, chcell::String, chcoeff)
```

Create a sparse vector representing a chain.

This short-cut method specifies a chain consiting of one cell and its coefficient. The cell is given via its label.

source

`ConleyDynamics.chain_support` – Function.

```
chain_support(lc::LefschetzComplex, cvec::SparseMatrix; coeff::Bool=false)
```

Determine the support of a chain given as a sparse vector.

This function returns the support of a chain, given in the form of a sparse vector. In the default usage, the function returns a `Vector{String}` which contains the labels of all the cells which have nonzero coefficients in the chain. If one passes the optional parameter `coeff=true`, then the function returns two arguments: In addition to the vector of labels as above, it also returns the vector of associated coefficients.

source

Chapter 13

Example Functions

13.1 Examples from Batko et al.

`ConleyDynamics.example_forman1d` – Function.

```
lc, mvf, coords = example_forman1d()
```

Create the simplicial complex and multivector field for the example from Figure 1 in the FoCM 2020 paper by Batko, Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices. The Lefschetz complex is defined over the finite field GF(2).

Examples

```
julia> lc, mvf = example_forman1d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm)
   ┆  A AD  F BF DE
---┆---------------
 A ┆  0  0  0  0  1
AD ┆  0  0  0  0  0
 F ┆  0  0  0  0  1
BF ┆  0  0  0  0  0
DE ┆  0  0  0  0  0

julia> sparse_show(cm.matrix)
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 0

julia> println(cm.labels)
["A", "AD", "F", "BF", "DE"]
```

source

ConleyDynamics.example_forman2d – Function.

```
lc, mvf, coords = example_forman2d()
```

Create the simplicial complex and multivector field for the example from Figure 3 in the FoCM 2020 paper by Batko, Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` over the finite field GF(2) and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices.

Examples

```
julia> lc, mvf = example_forman2d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm)
   ┆   D   E   F  GJ  BF  EF  HI ADE FGJ
───┆──────────────────────────────────
  D┆   0   0   0   0   1   0   1   0   0
  E┆   0   0   0   0   0   1   0   0   0
  F┆   0   0   0   0   1   1   1   0   0
 GJ┆   0   0   0   0   0   0   0   0   1
 BF┆   0   0   0   0   0   0   0   1   0
 EF┆   0   0   0   0   0   0   0   0   0
 HI┆   0   0   0   0   0   0   0   1   0
ADE┆   0   0   0   0   0   0   0   0   0
FGJ┆   0   0   0   0   0   0   0   0   0

julia> sparse_show(cm.matrix)
 0 0 0 0 1 0 1 0 0
 0 0 0 0 0 1 0 0 0
 0 0 0 0 1 1 1 0 0
 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm.labels)
["D", "E", "F", "GJ", "BF", "EF", "HI", "ADE", "FGJ"]
```

source

13.2 Examples from Mrozek & Wanner

ConleyDynamics.example_julia_logo – Function.

```
lc, mvf = example_julia_logo()
```

Create the simplicial complex and multivector field for the example from Figure 1 in the connection matrix paper by Mrozek & Wanner.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_julia_logo();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0
 0 0 1
 0 0 0

julia> print(cm.labels)
["D", "AC", "ABC"]
```

source

`ConleyDynamics.example_three_cm` – Function.

```
lc, mvf, coords = example_three_cm(mvftype)
```

Create the simplicial complex and multivector field for the example from Figure 2 in the connection matrix paper by Mrozek & Wanner.

Depending on the value of `mvftype`, return the periodic orbit (0=default) or one of the three gradient (1,2,3) examples.

The function returns the Lefschetz complex `lc` over the rational field and the multivector field `mvf`. If desired for plotting, the third return value `coords` gives a vector of coordinates for the vertices.

Examples

```
julia> lc, mvf = example_three_cm(0);

julia> cm = connection_matrix(lc, mvf);

julia> print(cm.labels)
["A", "C", "CE", "AC", "BD", "DF", "ABC", "EFG"]

julia> full_from_sparse(cm.matrix)
8×8 Matrix{Rational{Int64}}:
 0  0  0  -1  -1  0   0  0
 0  0  0   1   1  0   0  0
 0  0  0   0   0  0   0  0
 0  0  0   0   0  0  -1  0
 0  0  0   0   0  0   1  0
 0  0  0   0   0  0   0  1
 0  0  0   0   0  0   0  0
 0  0  0   0   0  0   0  0
```

source

`ConleyDynamics.example_multiflow` – Function.

```
lc, mvf = example_multiflow()
```

Create the Lefschetz complex and multivector field for the example from Figure 3 in the connection matrix paper by Mrozek & Wanner.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_multiflow();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0 0
 0 0 0 0
 0 0 0 0
 0 0 0 0

julia> print(cm.labels)
["BD", "DF", "AC", "CE"]
```

source

`ConleyDynamics.example_small_periodicity` – Function.

```
lc1, lc2, mvf = example_small_periodicity()
```

Create two representations of the Lefschetz complex and the multivector field for the example from Figure 4 in the connection matrix paper by Mrozek & Wanner.

The complexes `lc1` and `lc2` are just two representations of the same complex, but they lead to different connection matrices. Both Lefschetz complexes are defined over the finite field GF(2).

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector field `mvf`.

Examples

```
julia> lc1, lc2, mvf = example_small_periodicity();

julia> cm1 = connection_matrix(lc1, mvf);

julia> cm2 = connection_matrix(lc2, mvf);

julia> full_from_sparse(cm1.matrix)
4×4 Matrix{Int64}:
 0  0  0  0
 0  0  0  1
 0  0  0  1
 0  0  0  0
```

```
julia> print(cm1.labels)
["A", "a", "b", "alpha"]

julia> full_from_sparse(cm2.matrix)
4×4 Matrix{Int64}:
 0  0  0  0
 0  0  0  0
 0  0  0  1
 0  0  0  0

julia> print(cm2.labels)
["A", "c", "b", "alpha"]
```

source

ConleyDynamics.example_subdivision – Function.

```
lc, mvf = example_subdivision(mvftype)
```

Create the Lefschetz complex and multivector field for the example from Figure 11 in the connection matrix paper by Mrozek & Wanner.

Depending on the value of `mvftype`, return the multivector (0=default) or one of the two combinatorial vector field (1,2) examples.

The function returns the Lefschetz complex `lc` over the rationals and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_subdivision(1);

julia> cm = connection_matrix(lc, mvf);

julia> full_from_sparse(cm.matrix)
5×5 Matrix{Rational{Int64}}:
 0  0  -1  -1  -1
 0  0   1   0   0
 0  0   0   0   0
 0  0   0   0   0
 0  0   0   0   0
```

source

13.3 General Examples

ConleyDynamics.example_critical_simplex – Function.

```
lc, mvf = example_critical_simplex(dim)
```

Create a simplicial complex of dimension `dim` as well as a multivector field on it in which every cell is critical.

The function returns the Lefschetz complex `lc` over GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_critical_simplex(2);

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0 1 1 0 0
 0 0 0 1 0 1 0
 0 0 0 0 1 1 0
 0 0 0 0 0 0 1
 0 0 0 0 0 0 1
 0 0 0 0 0 0 1
 0 0 0 0 0 0 0

julia> print(cm.labels)
["A", "B", "C", "AB", "AC", "BC", "ABC"]
```

source

`ConleyDynamics.example_moebius` – Function.

```
lc1, mvf1, lc2, mvf2 = example_moebius(p)
```

Create two simplicial complexes for a cylinder and Moebius strip, respectively, together with associated multivector fields on them.

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector fields `mvf1` and `mvf2`. Both complexes are over a field with characteristic `p`. Positive prime characteristic uses the finite field GF(p), while zero characteristic gives the rationals.

The multivector field is the same, and it has one critical cell each in dimension 1 and 2 in the interior of the strip. The boundary consists of two periodic orbits for `lc1` and `mvf1`, and of one periodic orbit in the Moebius case `lc2` and `mvf2`. The latter case leads to different connection matrices for the fields GF(2) and GF(7), for example.

Examples

```
julia> lc1, mvf1, lc2, mvf2 = example_moebius(0);

julia> lc2p2 = lefschetz_gfp_conversion(lc2,2);

julia> lc2p7 = lefschetz_gfp_conversion(lc2,7);

julia> cmp2 = connection_matrix(lc2p2, mvf2);

julia> cmp7 = connection_matrix(lc2p7, mvf2);

julia> sparse_show(cmp2.matrix)
```

```
 0 0 0 0
 0 0 0 1
 0 0 0 0
 0 0 0 0

julia> sparse_show(cmp7.matrix)
 0 0 0 0
 0 0 0 1
 0 0 0 2
 0 0 0 0
```

source

ConleyDynamics.example_nonunique – Function.

```
lc1, lc2, mvf, coords1, coords2 = example_nonunique()
```

Create two representations of a simplicial complex and one multivector field which illustrates nonunique connection matrices.

The two complexes `lc1` and `lc2` represent the same simplicial complex over GF(2), but differ in the ordering of the labels.

The function returns the Lefschetz complexes `lc1` and `lc2`, as well as the multivector field `mvf`. If desired for plotting, the fourth and fifth return values `coords1` and `coords2` give vectors of coordinates for the vertices of the two complexes.

Examples

```
julia> lc1, lc2, mvf = example_nonunique();

julia> cm1 = connection_matrix(lc1, mvf);

julia> cm2 = connection_matrix(lc2, mvf);

julia> sparse_show(cm1.matrix)
 0 0 0 1 0 1 0 0 0
 0 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 0 1 1
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm1.labels)
["2", "7", "79", "29", "45", "67", "168", "349", "789"]

julia> sparse_show(cm2.matrix)
 0 0 0 1 0 1 0 0 0
 0 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 1 0 1
```

```
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 1 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0

julia> print(cm2.labels)
["2", "8", "78", "29", "45", "67", "168", "349", "789"]
```

source

ConleyDynamics.example_clorenz – Function.

```
lc, mvf = example_clorenz()
```

Create the simplicial complex and multivector field for the example from Figure 3 in the JCD 2016 paper by Kaczynski, Mrozek, and Wanner.

The function returns the Lefschetz complex `lc` over the finite field GF(2) and the multivector field `mvf`.

Examples

```
julia> lc, mvf = example_clorenz();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix)
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 0 0

julia> print(cm.labels)
["i", "ip", "g", "gm", "bc"]

julia> ms, ps = morse_sets(lc, mvf, poset=true);

julia> [conley_index(lc, mset) for mset in ms]
4-element Vector{Vector{Int64}}:
 [1, 1, 0]
 [1, 1, 0]
 [0, 1, 0]
 [0, 0, 0]

julia> ps
4×4 Matrix{Bool}:
 0  0  1  0
 0  0  1  0
 0  0  0  1
 0  0  0  0
```

source

ConleyDynamics.example_dunce_chaos – Function.

```
sc, vfG, vfC = example_dunce_chaos()
```

Create a minimal simplicial complex representation of the Dunce hat, as well as two Forman vector fields.

The function returns the simplicial representation of the Dunce hat in `sc` over the finite field `GF(2)`. The Forman vector field `vfG` is a gradient vector field with unique connection matrix. The field `vfC` is a modification of this field which merges the critical cells of dimensions 1 and 2 into a Forman arrow. The resulting Forman vector field is no longer gradient, and in fact exhibits Lorez-like chaos.

Examples

```
julia> sc, vfG, vfC = example_dunce_chaos();

julia> homology(sc)
3-element Vector{Int64}:
 1
 0
 0

julia> cmG = connection_matrix(sc, vfG);

julia> sparse_show(cmG.matrix)
 0 0 0
 0 0 1
 0 0 0

julia> print(cmG.labels)
["1", "12", "128"]

julia> cmC = connection_matrix(sc, vfC);

julia> sparse_show(cmC.matrix)
 0

julia> print(cmC.labels)
["1"]

julia> msC, psC = morse_sets(sc, vfC, poset=true);

julia> [conley_index(sc, mset) for mset in msC]
2-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 0, 0]

julia> psC
2×2 Matrix{Bool}:
 0  1
 0  0

julia> msC
2-element Vector{Vector{String}}:
```

```
 ["1"]
 ["12", "14", "15", "25", "28", "56", "68", "78", "124", "125", "128", "145", "256", "278",
↪  "568", "678"]
```

source

ConleyDynamics.example_torsion_chaos – Function.

```
sc, vfG, vfC = example_torsion_chaos(n::Int, p::Int)
```

Create a triangulation of a space with 1-dimensional torsion, as well as two Forman vectr fields on this complex.

The function returns a simplicial complex `sc` which has the following integer homology groups:

- In dimension 0 it is the group of integers.
- In dimension 1 it is the integers modulo n.
- In dimension 2 it is the trivial group.

In other words, the simplicial complex has nontrivial torsion in dimension 1. It is a triangulation of an n-gon, in which all boundary edges are oriented counterclockwise, and all of these edges are identified. The parameter p specifies the characteristic of the underlying field.

In addition, two Forman vector fields `vfG` and `vfC` are returned. The first one is a gradient vector field whose connection matrix has a large connection matrix entry. In fact, if p is any prime larger than n then there will be an entry n in the matrix. The scond Forman vector field contains a chaotic Morse set. This Morse set will have trivial Morse index for most p. On the other hand, for prime `p = n` the set has the Morse index of an unstable periodic orbit.

Examples

```
julia> sc, vfG, vfC = example_torsion_chaos(3,7);

julia> homology(sc)
3-element Vector{Int64}:
 1
 0
 0

julia> cmG = connection_matrix(sc, vfG);

julia> sparse_show(cmG.matrix)
 0 0 0
 0 0 3
 0 0 0

julia> print(cmG.labels)
["0w", "0w0x", "0w0x1s"]

julia> cmC = connection_matrix(sc, vfC);

julia> sparse_show(cmC.matrix)
```

```
 0

julia> print(cmC.labels)
["0w"]

julia> msC, psC = morse_sets(sc, vfC, poset=true);

julia> [conley_index(sc, mset) for mset in msC]
2-element Vector{Vector{Int64}}:
 [1, 0, 0]
 [0, 0, 0]

julia> psC
2×2 Matrix{Bool}:
 0  1
 0  0

julia> length.(msC)
2-element Vector{Int64}:
  1
 26
```

source

Chapter 14

Plotting Functions

14.1 Visualizing Simplicial Complexes

`ConleyDynamics.plot_planar_simplicial` – Function.

```
plot_planar_simplicial(sc::LefschetzComplex,
                       coords::Vector{<:Vector{<:Real}},
                       fname::String;
                       [mvf::CellSubsets=Vector{Vector{Int}}([]),]
                       [labeldir::Vector{<:Real}=Vector{Int}([]),]
                       [labeldis::Real=8,]
                       [hfac::Real=1.2,]
                       [vfac::Real=1.2,]
                       [sfac::Real=0,]
                       [pdim::Vector{Bool}=[true,true,true],]
                       [pv::Bool=false])
```

Create an image of a planar simplicial complex, and if specified, a Forman vector field on it.

The vector `coords` contains coordinates for every one of the vertices of the simplicial complex `sc`. The image will be saved in the file with name `fname`, and the ending determines the image type. Accepted are `.pdf`, `.svg`, `.png`, and `.eps`.

If the optional `mvf` is specified and is a Forman vector field, then this Forman vector field is drawn as well. The optional vector `labeldir` contains directions for the vertex labels, and `labeldis` the distance from the vertex. The directions have to be reals between 0 and 4, with 0,1,2,3 corresponding to E,N,W,S. The optional constants `hfac` and `vfac` contain the horizontal and vertical scale vectors, while `sfac` describes a uniform scale. If `sfac=0` the latter is automatically determined. The vector `pdim` specifies in which dimensions cells are drawn; the default shows vertices, edges, and triangles. Finally if one passes the argument `pv=true`, then in addition to saving the file a preview is displayed.

Examples

Suppose we have created a simplicial complex using the commands

```
sc, coords = create_simplicial_delaunay(300, 300, 30, 20)
fname = "sc_plot_test.pdf"
```

Then the following code creates an image of the simplicial complex without labels, but with a preview:

```
plot_planar_simplicial(sc, coords, fname, pv=true)
```

If we want to see the labels, we can use

```
ldir = fill(0.5, sc.ncells);
plot_planar_simplicial(sc, coords, fname, labeldir=ldir, labeldis=10, pv=true)
```

This command puts all labels in the North-East direction at a distance of 10.

source

ConleyDynamics.plot_planar_simplicial_morse – Function.

```
plot_planar_simplicial_morse(sc::LefschetzComplex,
                             coords::Vector{<:Vector{<:Real}},
                             fname::String,
                             morsesets::CellSubsets;
                             [hfac::Real=1.2,]
                             [vfac::Real=1.2,]
                             [sfac::Real=0,]
                             [pdim::Vector{Bool}=[false,true,true],]
                             [pv::Bool=false,]
                             [ci::Bool=false])
```

Create an image of a planar simplicial complex, together with Morse sets, or also selected multivectors.

The vector `coords` contains coordinates for every one of the vertices of the simplicial complex `sc`. The image will be saved in the file with name `fname`, and the ending determines the image type. Accepted are `.pdf`, `.svg`, `.png`, and `.eps`.

The vector `morsesets` contains a list of Morse sets, or more general, subsets of the simplicial complex. For every k, the set described by `morsesets[k]` will be shown in a distinct color.

The optional constants `hfac` and `vfac` contain the horizontal and vertical scale vectors for the margins, while `sfac` describes a uniform scale. If `sfac=0` the latter is automatically determined. The vector `pdim` specifies in which dimensions cells are drawn; the default only shows edges and triangles. If one passes the argument `pv=true`, then in addition to saving the file a preview is displayed. Lastly, passing the argument `ci=true` will color code the Morse sets according to their respective Conley indices.

source

14.2 Visualizing Cubical Complexes

ConleyDynamics.plot_planar_cubical – Function.

```
plot_planar_cubical(cc::LefschetzComplex,
                    coords::Vector{<:Vector{<:Real}},
                    fname::String;
                    [hfac::Real=1.2,]
                    [vfac::Real=1.2,]
                    [cubefac::Real=0,]
                    [pdim::Vector{Bool}=[true,true,true],]
                    [pv::Bool=false])
```

Create an image of a planar cubical complex.

The vector `coords` contains coordinates for every one of the vertices of the cubical complex `cc`. The image will be saved in the file with name `fname`, and the ending determines the image type. Accepted are `.pdf`, `.svg`, `.png`, and `.eps`. The optional constants `hfac` and `vfac` contain the horizontal and vertical scale vectors. The optional argument `cubefac` specifies the side length of an elementary cube for plotting, and it will be automatically determined otherwise. The vector `pdim` specifies which cell dimensions should be plotted, with `pdim[k]` representing dimension `k-1`. Finally if one passes the argument `pv=true`, then in addition to saving the file a preview is displayed.

Examples

Suppose we have created a cubical complex using the commands

```
cubes = ["00.11", "01.01", "02.10", "11.10", "11.01", "22.00"]
coords = [[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,1],[2,2]]
cc = create_cubical_complex(cubes)
fname = "cc_plot_test.pdf"
```

Then the following code creates an image of the simplicial complex without labels, but with a preview:

```
plot_planar_cubical(cc, coords, fname, pv=true)
```

If one only wants to plot the edges in the complex, but not the vertices or rectangles, then one can use:

```
plot_planar_cubical(cc, coords, fname, pv=true, pdim=[false,true,false])
```

source

```
plot_planar_cubical(cc::LefschetzComplex,
                    fname::String;
                    [hfac::Real=1.2,]
                    [vfac::Real=1.2,]
                    [cubefac::Real=0,]
                    [pdim::Vector{Bool}=[true,true,true],]
                    [pv::Bool=false])
```

Create an image of a planar cubical complex.

This is an alternative method which does not require the specification of the vertex coordinates. They will be taken from the cube vertex labels.

source

`ConleyDynamics.plot_planar_cubical_morse` – Function.

```
plot_planar_cubical_morse(cc::LefschetzComplex,
                          coords::Vector{<:Vector{<:Real}},
                          fname::String,
                          morsesets::CellSubsets;
                          [hfac::Real=1.2,]
```

```
                          [vfac::Real=1.2,]
                          [cubefac::Real=0,]
                          [pdim::Vector{Bool}=[false,true,true],]
                          [pv::Bool=false])
```

Create an image of a planar cubical complex, together with Morse sets, or also selected multivectors.

The vector `coords` contains coordinates for every one of the vertices of the cubical complex `cc`. The image will be saved in the file with name `fname`, and the ending determines the image type. Accepted are `.pdf`, `.svg`, `.png`, and `.eps`.

The vector `morsesets` contains a list of Morse sets, or more general, subsets of the cubical complex. For every k, the set described by `morsesets[k]` will be shown in a distinct color.

The optional constants `hfac` and `vfac` contain the horizontal and vertical scale vectors for the margins, while `cubefac` describes a uniform scale. If `cubefac=0` the latter is automatically determined. The vector `pdim` specifies in which dimensions cells are drawn; the default only shows edges and squares. Finally if one passes the argument `pv=true`, then in addition to saving the file a preview is displayed.

source

```
plot_planar_cubical_morse(cc::LefschetzComplex,
                          fname::String,
                          morsesets::CellSubsets;
                          [hfac::Real=1.2,]
                          [vfac::Real=1.2,]
                          [cubefac::Real=0,]
                          [pdim::Vector{Bool}=[false,true,true],]
                          [pv::Bool=false])
```

Create an image of a planar cubical complex, together with Morse sets, or also selected multivectors.

This is an alternative method which does not require the specification of the vertex coordinates. They will be taken from the cube vertex labels.

source

Chapter 15

Sparse Matrix Functions

15.1 Internal Sparse Matrix Representation

ConleyDynamics.SparseMatrix – Type.

```
SparseMatrix{T}
```

Composite data type for a sparse matrix with entries of type T.

The struct has the following fields:

- `const nrow::Int`: Number of rows
- `const ncol::Int`: Number of columns
- `const char::Int`: Characteristic of type T
- `const zero::T`: Number 0 of type T
- `const one::T`: Number 1 of type T
- `entries::Vector{Vector{T}}`: Matrix entries corresponding to `columns`
- `columns::Vector{Vector{Int}}`: `columns[k]` points to nonzero entries in column k
- `rows::Vector{Vector{Int}}`: `rows[k]` points to nonzero entries in the k-th row

source

15.2 Access Functions

ConleyDynamics.sparse_get_entry – Function.

```
sparse_get_entry(matrix::SparseMatrix, ri::Int, ci::Int)
```

Get the sparse matrix entry at location `(ri,ci)`.

source

Base.getindex – Method.

```
Base.getindex(matrix::SparseMatrix, ri::Int, ci::Int)
```

Get the sparse matrix entry at location `(ri,ci)`.

source

`ConleyDynamics.sparse_set_entry!` – Function.

```
sparse_set_entry!(matrix::SparseMatrix, ri::Int, ci::Int, val)
```

Set the sparse matrix entry at location `(ri,ci)` to `val`.

source

```
sparse_set_entry!(matrix::SparseMatrix{Int}, ri::Int, ci::Int, val)
```

Set the sparse matrix entry at location `(ri,ci)` to `val`.

This method assumes that the sparse matrix is over the integers, which means that it is interpreted over a finite field. Thus, the new entry is automatically reduced via modular arithmetic.

source

`Base.setindex!` – Method.

```
Base.setindex!(matrix::SparseMatrix, val, ri::Int, ci::Int)
```

Set the sparse matrix entry at location `(ri,ci)` to `val`.

source

`ConleyDynamics.sparse_get_column` – Function.

```
sparse_get_column(matrix::SparseMatrix, ci::Int)
```

Get the ci-th column of the sparse matrix.

source

`ConleyDynamics.sparse_get_nz_column` – Function.

```
sparse_get_nz_column(matrix::SparseMatrix, ci::Int)
```

Get the row indices for the nonzero entries in the ci-th column of the sparse matrix.

source

`ConleyDynamics.sparse_get_nz_row` – Function.

```
sparse_get_nz_row(matrix::SparseMatrix, ri::Int)
```

Get the column indices for the nonzero entries in the ri-th row of the sparse matrix.

source

ConleyDynamics.sparse_minor – Function.

```
smp = sparse_minor(sm::SparseMatrix, rvec::Vector{Int}, cvec::Vector{Int})
```

Create sparse submatrix by specifying the desired row and column indices.

source

15.3 Basic Functions

ConleyDynamics.sparse_size – Function.

```
sparse_size(matrix::SparseMatrix, dim::Int)
```

Number of rows (dim=1) or columns (dim=2) of a sparse matrix.

source

ConleyDynamics.sparse_low – Function.

```
sparse_low(matrix::SparseMatrix, col::Int)
```

Row index of the lowest nonzero matrix entry in column col.

source

ConleyDynamics.sparse_is_zero – Function.

```
sparse_is_zero(sm::SparseMatrix)
```

Test whether the sparse matrix sm is the zero matrix.

source

ConleyDynamics.sparse_is_identity – Function.

```
sparse_is_identity(A::SparseMatrix)
```

Test whether the sparse matrix A is the identity matrix.

source

`ConleyDynamics.sparse_is_equal` – Function.

```
sparse_is_equal(A::SparseMatrix, B::SparseMatrix)
```

Test whether the sparse matrices `A` and `B` are equal.

For equality, the two sparse matrices not only have to have the same size and the same entries, they also have to be of the same type and be defined over the same field.

source

`Base.:(==)` – Method.

```
Base.:(==)(A::SparseMatrix,B::SparseMatrix)
```

Test whether the sparse matrices `A` and `B` are equal.

For equality, the two sparse matrices not only have to have the same size and the same entries, they also have to be of the same type and be defined over the same field.

source

`ConleyDynamics.sparse_is_sut` – Function.

```
bool = sparse_is_sut(sm::SparseMatrix)
```

Check whether the sparse matrix is strictly upper triangular.

source

`ConleyDynamics.sparse_identity` – Function.

```
sparse_identity(n::Int; p::Int=0)
```

Create a sparse identity matrix with `n` rows and columns.

The optional argument `p` specifies the field characteristic. If `p=0` then the sparse matrix is over the rationals, while if `p>0` is a prime, then the matrix is an integer matrix whose entries are interpreted in `GF(p)`.

source

`ConleyDynamics.sparse_zero` – Function.

```
sparse_zero(nr::Int, nc::Int; p::Int=0)
```

Create a sparse zero matrix with `nr` rows and `nc` columns.

The optional argument `p` specifies the field characteristic. If `p=0` then the sparse matrix is over the rationals, while if `p>0` is a prime, then the matrix is an integer matrix whose entries are interpreted in `GF(p)`.

source

ConleyDynamics.sparse_fullness – Function.

```
sparse_fullness(sm::SparseMatrix)
```

Return the fullness of the sparse matrix `sm`, which equals the percentage of nonzero elements.

source

ConleyDynamics.sparse_sparsity – Function.

```
sparse_sparsity(sm::SparseMatrix)
```

Return the sparsity of the sparse matrix `sm`, which equals the percentage of zero entries.

source

ConleyDynamics.sparse_nz_count – Function.

```
sparse_nz_count(sm::SparseMatrix)
```

Return the number of nonzero entries of the sparse matrix `sm`.

source

ConleyDynamics.sparse_show – Function.

```
sparse_show(sm::SparseMatrix)
```

Display a sparse matrix in readable format.

This function will print the complete matrix, so be careful with sparse matrices of large size!

source

```
sparse_show(sm::SparseMatrix, rlabels::Vector{String}, clabels::Vector{String})
```

Display a sparse matrix in readable format, with labels.

The input parameter `clabels` contains the labels for the columns, while `rlabels` give the row labels.

Examples

```
julia> lc, mvf = example_forman1d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm.matrix, cm.labels, cm.labels)
   ┆  A AD  F BF DE
---┆---------------
 A ┆  0  0  0  0  1
```

```
AD┆  0  0  0  0  0
 F┆  0  0  0  0  1
BF┆  0  0  0  0  0
DE┆  0  0  0  0  0
```

source

```
sparse_show(sm::SparseMatrix, labels::Vector{String})
```

Display a sparse matrix in readable format, with labels.

This function assumes that the matrix is square, and that the labels are the same for rows and columns. They are provided in `labels`.

source

```
sparse_show(cm::ConleyMorseCM)
```

Display the connection matrix with labels.

This function displays the (sparse) connection matrix including its labels. It uses `sparse_show(cm.matrix, cm.labels, cm.labels)`.

Examples

```
julia> lc, mvf = example_forman1d();

julia> cm = connection_matrix(lc, mvf);

julia> sparse_show(cm)
  ┆  A AD  F BF DE
--┆---------------
 A┆  0  0  0  0  1
AD┆  0  0  0  0  0
 F┆  0  0  0  0  1
BF┆  0  0  0  0  0
DE┆  0  0  0  0  0
```

source

`Base.show` – Method.

```
Base.show(io::IO, ::MIME"text/plain", sm::SparseMatrix)
```

Display the sparse matrix `sm` when hitting return in REPL.

source

15.4 Elementary Matrix Operations

`ConleyDynamics.sparse_add_column!` – Function.

```
sparse_add_column!(matrix::SparseMatrix, ci1::Int, ci2::Int, cn, cd)
```

Replace `column[ci1]` by `column[ci1] + (cn/cd) * column[ci2]`.

source

```
sparse_add_column!(matrix::SparseMatrix{Int}, ci1::Int, ci2::Int,
                   cn::Int, cd::Int)
```

Replace `column[ci1]` by `column[ci1] + (cn/cd) * column[ci2]`.

The computation is performed mod p, where the characteristic is taken from `matrix.char`. An error is thrown if `matrix.char==0`.

source

`ConleyDynamics.sparse_add_row!` – Function.

```
sparse_add_row!(matrix::SparseMatrix, ri1::Int, ri2::Int, cn, cd)
```

Replace `row[ri1]` by `row[ri1] + (cn/cd) * row[ri2]`.

source

```
sparse_add_row!(matrix::SparseMatrix{Int}, ri1::Int, ri2::Int,
                cn::Int, cd::Int)
```

Replace `row[ri1]` by `row[ri1] + (cn/cd) * row[ri2]`.

The computation is performed mod p, where the characteristic is taken from `matrix.char`. An error is thrown if `matrix.char==0`.

source

`ConleyDynamics.sparse_permute` – Function.

```
sparse_permute(sm::SparseMatrix, pr::Vector{Int}, pc::Vector{Int})
```

Create sparse matrix by permuting the row and column indices.

The vector `pr` describes the row permutation, and `pc` the column permutation.

source

`ConleyDynamics.sparse_inverse` – Function.

```
sparse_inverse(matrix::SparseMatrix)
```

Compute the inverse of a sparse matrix.

The function automatically performs the computations over the underlying field of the sparse matrix.

source

`ConleyDynamics.sparse_remove!` – Function.

```
sparse_remove!(matrix::SparseMatrix, ri::Int, ci::Int)
```

Remove the sparse matrix entry at location `(ri,ci)`.

source

`ConleyDynamics.sparse_add` – Function.

```
sparse_add(A::SparseMatrix, B::SparseMatrix)
```

Add two sparse matrices.

Exceptions are raised if the matrix sum is not defined or the entry types do not match.

source

`ConleyDynamics.sparse_subtract` – Function.

```
sparse_subtract(A::SparseMatrix, B::SparseMatrix)
```

Subtract two sparse matrices.

The function returns `A - B`. Exceptions are raised if the matrix difference is not defined or the entry types do not match.

source

`ConleyDynamics.sparse_multiply` – Function.

```
sparse_multiply(A::SparseMatrix, B::SparseMatrix)
```

Multiply two sparse matrices.

Exceptions are raised if the matrix product is not defined or the entry types do not match.

source

`ConleyDynamics.sparse_scale` – Function.

```
sparse_scale(sfac, A::SparseMatrix)
```

Scale a sparse matrix by a scalar.

An exception is raised if the entry types do not match.

source

Base.:+ – Method.

```
Base.:+(A::SparseMatrix, B::SparseMatrix)
```

Add two sparse matrices.

Exceptions are raised if the matrix sum is not defined or the entry types do not match.

source

Base.:- – Method.

```
Base.:-(A::SparseMatrix, B::SparseMatrix)
```

Subtract two sparse matrices.

Exceptions are raised if the matrix difference is not defined or the entry types do not match.

source

Base.:* – Method.

```
Base.:*(A::SparseMatrix, B::SparseMatrix)
```

Multiply two sparse matrices.

Exceptions are raised if the matrix product is not defined or the entry types do not match.

source

Base.:* – Method.

```
Base.:*(sfac, A::SparseMatrix)
```

Compute the scalar product of `sfac` and the sparse matrix `A`.

An exception is raised if the scalar `sfac` does not have the same type as the matrix entries.

source

15.5 Conversion Functions

ConleyDynamics.sparse_from_full – Function.

```
sparse_from_full(matrix::Matrix{Int}; [p::Int=0])
```

Create sparse matrix from full integer matrix. If the optional argument p is specified and positive, then the returned matrix is an integer matrix which is interpreted over GF(p). On the other hand, if p is omitted or equal to zero, then the return matrix has rational type.

source

ConleyDynamics.full_from_sparse – Function.

```
full_from_sparse(sm::SparseMatrix)
```

Create full matrix from sparse matrix.

source

ConleyDynamics.sparse_from_lists – Function.

```
sparse_from_lists(nr, nc, tchar, tzero, tone, r, c, v)
```

Create sparse matrix from lists describing the entries.

The vectors r, c, and v have to have the same length and the matrix has entry v[k] at (r[k],c[k]). Zero entries will be ignored, and multiple entries for the same matrix position raise an error.

The input arguments have the following meaning:

- nr::Int: Number of rows
- nc::Int: Number of columns
- tchar: Field characteristic if T==Int
- tzero::T: Number 0 of type T
- tone::T: Number 1 of type T
- r::Vector{Int}: Vector of row indices
- c::Vector{Int}: Vector of column indices
- v::Vector{T}: Vector of matrix entries

If tchar>0, then the entries in v are all replaced by their values mod tchar.

source

ConleyDynamics.lists_from_sparse – Function.

```
nr, nc, tchar, tzero, tone, r, c, v = lists_from_sparse(sm::SparseMatrix)
```

Create list representation from sparse matrix.

The output variables are exactly what is needed to create a sparse matrix object using sparse_from_lists.

source

15.6 Sparse Helper Functions

`ConleyDynamics.scalar_inverse` – Function.

```
scalar_inverse(s, p::Int)
```

Compute the inverse of a scalar.

source

```
scalar_inverse(s::Int, p::Int)
```

Compute the inverse of a scalar.

This function computes the inverse in modular arithmetic with base `p`.

source

`ConleyDynamics.scalar_multiply` – Function.

```
scalar_multiply(s1, s2, p::Int)
```

Compute the product of two scalars.

source

```
scalar_multiply(s1::Int, s2::Int, p::Int)
```

Compute the product of two scalars.

This function computes the product in modular arithmetic with base `p`.

source

`ConleyDynamics.scalar_add` – Function.

```
scalar_add(s1, s2, p::Int)
```

Compute the sum of two scalars.

source

```
scalar_add(s1::Int, s2::Int, p::Int)
```

Compute the sum of two scalars.

This function computes the sum in modular arithmetic with base `p`.

source

Chapter 16

Complete API Index

16.1 Composite Data Structures

- ConleyDynamics
- ConleyDynamics.Cell
- ConleyDynamics.CellSubsets
- ConleyDynamics.Cells
- ConleyDynamics.ConleyMorseCM
- ConleyDynamics.LefschetzComplex
- Base.show
- Base.show

16.2 Lefschetz Complex Functions

- ConleyDynamics.cellsubset_bounding_box
- ConleyDynamics.cellsubset_distance
- ConleyDynamics.cellsubset_location_circle
- ConleyDynamics.cellsubset_location_rectangle
- ConleyDynamics.cellsubset_planar_area
- ConleyDynamics.compose_reductions
- ConleyDynamics.convert_cells
- ConleyDynamics.convert_cellsubsets
- ConleyDynamics.convert_planar_coordinates
- ConleyDynamics.convert_spatial_coordinates
- ConleyDynamics.create_cubical_box
- ConleyDynamics.create_cubical_complex

- ConleyDynamics.lefschetz_skeleton
- ConleyDynamics.lefschetz_subcomplex
- ConleyDynamics.lefschetz_topboundary
- ConleyDynamics.locate_planar_cellsubsets
- ConleyDynamics.manifold_boundary
- ConleyDynamics.permute_lefschetz_complex
- ConleyDynamics.segment_intersects_circle
- ConleyDynamics.segment_intersects_rectangle
- ConleyDynamics.signed_distance_circle
- ConleyDynamics.signed_distance_rectangle
- ConleyDynamics.simplicial_klein_bottle
- ConleyDynamics.simplicial_projective_plane
- ConleyDynamics.simplicial_torsion_space
- ConleyDynamics.simplicial_torus

16.3 Homology Functions

- ConleyDynamics.homology
- ConleyDynamics.persistent_homology
- ConleyDynamics.ph_reduce!
- ConleyDynamics.relative_homology

16.4 Conley Theory Functions

- ConleyDynamics.chain_support
- ConleyDynamics.chain_vector
- ConleyDynamics.cm_reduce!
- ConleyDynamics.conley_index
- ConleyDynamics.connection_matrix
- ConleyDynamics.create_mvf_hull
- ConleyDynamics.create_planar_mvf
- ConleyDynamics.create_spatial_mvf
- ConleyDynamics.extract_multivectors
- ConleyDynamics.forman_comb_flow

- ConleyDynamics.forman_gpaths
- ConleyDynamics.forman_path_weight
- ConleyDynamics.forman_stab_flow
- ConleyDynamics.isoinvset_information
- ConleyDynamics.morse_interval
- ConleyDynamics.morse_sets
- ConleyDynamics.mvf_information
- ConleyDynamics.planar_nontransverse_edges
- ConleyDynamics.remove_exit_set
- ConleyDynamics.restrict_dynamics

16.5 Example Functions

- ConleyDynamics.example_clorenz
- ConleyDynamics.example_critical_simplex
- ConleyDynamics.example_dunce_chaos
- ConleyDynamics.example_forman1d
- ConleyDynamics.example_forman2d
- ConleyDynamics.example_julia_logo
- ConleyDynamics.example_moebius
- ConleyDynamics.example_multiflow
- ConleyDynamics.example_nonunique
- ConleyDynamics.example_small_periodicity
- ConleyDynamics.example_subdivision
- ConleyDynamics.example_three_cm
- ConleyDynamics.example_torsion_chaos

16.6 Plotting Functions

- ConleyDynamics.plot_planar_cubical
- ConleyDynamics.plot_planar_cubical_morse
- ConleyDynamics.plot_planar_simplicial
- ConleyDynamics.plot_planar_simplicial_morse

16.7 Sparse Matrix Functions

- ConleyDynamics.SparseMatrix
- Base.:*
- Base.:*
- Base.:+
- Base.:-
- Base.:(==)
- Base.getindex
- Base.setindex!
- Base.show
- ConleyDynamics.full_from_sparse
- ConleyDynamics.lists_from_sparse
- ConleyDynamics.scalar_add
- ConleyDynamics.scalar_inverse
- ConleyDynamics.scalar_multiply
- ConleyDynamics.sparse_add
- ConleyDynamics.sparse_add_column!
- ConleyDynamics.sparse_add_row!
- ConleyDynamics.sparse_from_full
- ConleyDynamics.sparse_from_lists
- ConleyDynamics.sparse_fullness
- ConleyDynamics.sparse_get_column
- ConleyDynamics.sparse_get_entry
- ConleyDynamics.sparse_get_nz_column
- ConleyDynamics.sparse_get_nz_row
- ConleyDynamics.sparse_identity
- ConleyDynamics.sparse_inverse
- ConleyDynamics.sparse_is_equal
- ConleyDynamics.sparse_is_identity
- ConleyDynamics.sparse_is_sut
- ConleyDynamics.sparse_is_zero
- ConleyDynamics.sparse_low

- ConleyDynamics.sparse_minor
- ConleyDynamics.sparse_multiply
- ConleyDynamics.sparse_nz_count
- ConleyDynamics.sparse_permute
- ConleyDynamics.sparse_remove!
- ConleyDynamics.sparse_scale
- ConleyDynamics.sparse_set_entry!
- ConleyDynamics.sparse_show
- ConleyDynamics.sparse_size
- ConleyDynamics.sparse_sparsity
- ConleyDynamics.sparse_subtract
- ConleyDynamics.sparse_zero