abelian Documentation

Release 1.0.1

Tommy Odland

Nov 15, 2017

Contents

CHAPTER 1

Project overview

1.1 Short description

Welcome to the documentation of `abelian`, a Python library which facilitates computations on elementary locally compact abelian groups (LCAs). The LCAs are the groups isomorphic to $\mathbb{R}$, $T = \mathbb{R}/\mathbb{Z}$, $\mathbb{Z}$, $\mathbb{Z}_n$ and direct sums of these. The library is structured into two packages, the `abelian` package and the `abelian.linalg` sub-package, which is built on the matrix class `MutableDenseMatrix` from the `sympy` library for symbolic mathematics.

1.1.1 Classes and methods

- The `LCA` class represents elementary LCAs.
  - **Fundamental methods**: identity LCA, direct sums, equality, isomorphic, element projection, Pontryagin dual.
- The `HomLCA` class represents homomorphisms between LCAs.
  - **Fundamental methods**: identity morphism, zero morphism, equality, composition, evaluation, stacking, element-wise operations, kernel, cokernel, image, coimage, dual (adjoint) morphism.
- The `LCAFunc` class represents functions from LCAs to complex numbers.
  - **Fundamental methods**: evaluation, composition, shift (translation), pullback, pushforward, point-wise operators (e.g. addition).

Algorithms for the Smith normal form and Hermite normal form are also implemented in *`smith_normal_form()`* and *`hermite_normal_form()`* respectively.

1.2 Project goals

- Represent the groups $\mathbb{R}$, T, $\mathbb{Z}$ and $\mathbb{Z}_n$ and facilitate computations on these.

- Handle the relationship between discrete and continuous groups in a natural way using group homomorphisms.
- DFT computations on discrete, finite groups and their products using the FFT.
- The software should build on the mathematical theory.

1.3 Installation

1. Download the latest version of Python, e.g. the Anaconda distribution.
2. Depending on your operating system, do one of the following:
   (a) If on **Windows**, open the Anaconda prompt and run `pip install abelian` to install `abelian` from PyPI.
   (b) If on **Linux** or **Mac**, open the terminal and run `pip install abelian` to install `abelian` from PyPI.
3. Open a Python editor (such as Spyder, which comes with Anaconda) and type `from abelian import *` to import all classes and functions from the library. You're all set, go try some examples from the *tutorials*.

CHAPTER 2

Contents

2.1 Software specification

Below is an automatically generated software specification.

2.1.1 Public classes

HomLCA(A[, target, source])	A homomorphism between elementary LCAs.
LCA(orders[, discrete])	An elementary locally compact abelian group (LCA).
LCAFunc(representation, domain)	A function from an LCA to a complex number.

2.1.2 Public functions

hermite_normal_form(A)	Compute U and H such that A*U = H.
smith_normal_form(A[, compute_unimod])	Compute U,S,V such that U*A*V = S.
solve(A, b[, p])	Solve eqn Ax = b mod p over Z.
voronoi(epimorphism[, norm_p])	Return the Voronoi transversal function.

2.1.3 Public classes (detailed)

LCAFunc

(inherits from: `Callable`)

LCAFunc(representation, domain)	A function from an LCA to a complex number.
`__call__`(list_arg, *args, **kwargs)	Override function calls, see *evaluate()*.
__init__(representation, domain)	Initialize a function G -> C.

Continued on next page

Table 2.3 – continued from previous page

__repr__()	Override the repr() function.
copy()	Return a copy of the instance.
dft([func_type])	If the domain allows it, compute DFT.
evaluate(list_arg, *args, **kwargs)	Evaluate function on a group element.
idft([func_type])	If the domain allows it, compute inv DFT.
pointwise(other, operator)	Apply pointwise binary operator.
pullback(morphism)	Return the pullback along *morphism*.
pushforward(morphism[, terms_in_sum])	Return the pushforward along *morphism*.
sample(list_of_elements, *args, **kwargs)	Sample on a list of group elements.
shift(list_shift)	Shift the function.
to_latex()	Return as a LaTeX string.
to_table(*args, **kwargs)	Return a n-dimensional table.
transversal(epimorphism[, transversal_rule, ...])	Pushforward using transversal rule.

LCA

(inherits from: Sequence, Callable)

LCA(orders[, discrete])	An elementary locally compact abelian group (LCA).
__add__(other)	Override the addition (+) operator, see sum().
__call__(element)	Override function calls, see project_element().
__contains__(other)	Override the 'in' operator, see contained_in().
__eq__(other)	Override the equality (==) operator, see equal().
__getitem__(key)	Override the slice operator, see slice().
__init__(orders[, discrete])	Initialize a new LCA.
__iter__()	Override the iteration protocol, see iterate().
__len__()	Override the len() function, see length().
__pow__(power[, modulo])	Override the pow (**) operator, see compose_self().
__repr__()	Override the repr() function.
canonical()	Return the LCA in canonical form using SNF.
compose_self(power)	Repeated direct summation.
contained_in(other)	Whether the LCA is contained in *other*.
copy()	Return a copy of the LCA.
dual()	Return the Pontryagin dual of the LCA.
elements_by_maxnorm([norm_values])	Yield elements corresponding to max norm value.
equal(other)	Whether or not two LCAs are equal.
getitem(key)	Return a slice of the LCA.
is_FGA()	Whether or not the LCA is a FGA.
isomorphic(other)	Whether or not two LCAs are isomorphic.
iterate()	Yields the groups in the direct sum one by one.
length()	The number of groups in the direct sum.
project_element(element)	Project an element onto the group.
rank()	Return the rank of the LCA.

Continued on next page

Table 2.4 – continued from previous page

remove_indices(indices)	Return a LCA with some groups removed.
remove_trivial()	Remove trivial groups from the object.
sum(other)	Return the direct sum of two LCAs.
to_latex()	Return the LCA as a LaTeX string.
trivial()	Return a trivial LCA.

HomLCA

(inherits from: Callable)

HomLCA(A[, target, source])	A homomorphism between elementary LCAs.
__add__(other)	Override the addition (+) operator, see *add()*.
__call__(source_element)	Override function calls, see *evaluate()*.
__eq__(other)	Override the equality (==) operator, see *equal()*.
__getitem__(args)	Override the slice operator, see *getitem()*.
__init__(A[, target, source])	Initialize a homomorphism.
__mul__(other)	Override the * operator, see *compose()*.
__pow__(power[, modulo])	Override the pow (**) operator, see *compose_self()*.
__radd__(other)	Override the addition (+) operator, see *add()*.
__repr__()	Override the repr() function.
__rmul__(other)	Override the * operator, see *compose()*.
add(other)	Elementwise addition.
annihilator()	Compute the annihilator monomorphism.
coimage()	Compute the coimage epimorphism.
cokernel()	Compute the cokernel epimorphism.
compose(other)	Compose two homomorphisms.
compose_self(power)	Repeated composition of an endomorphism.
copy()	Return a copy of the homomorphism.
det()	Determinant of the matrix representing the HomLCA.
dual()	Compute the dual homomorphism.
equal(other)	Whether or not two homomorphisms are equal.
evaluate(source_element)	Apply the homomorphism to an element.
getitem(args)	Return a slice of the homomorphism.
identity(group)	Return the identity morphism.
image()	Compute the image monomorphism.
kernel()	Compute the kernel monomorphism.
project_to_source()	Project columns to source group (orders).
project_to_target()	Project columns to target group.
remove_trivial_groups()	Remove trivial groups.
stack_diag(other)	Stack diagonally.
stack_horiz(other)	Stack horizontally (column wise).
stack_vert(other)	Stack vertically (row wise).
to_latex()	Return the homomorphism as a LaTeX string.
update([new_A, new_target, new_source])	Return a new homomorphism with updated properties.

Continued on next page

Table 2.5 – continued from previous page

zero(target, source)	Initialize the zero morphism.

2.2 Tutorials

Here you will find tutorials convering all the main aspects of the `abelian` library.

2.2.1 Tutorial: LCAs

This is an interactive tutorial written with real code. We start by importing the `LCA` class and setting up LaTeX printing.

```
In [1]: from abelian import LCA
        from IPython.display import display, Math

        def show(arg):
            """This function lets us show LaTeX output."""
            return display(Math(arg.to_latex()))
```

Initializing a LCA

Initializing a locally compact abelian group (LCA) is simple. Every LCA can be written as a direct sum of groups isomorphic to one of: $\mathbb{Z}_n$, $\mathbb{Z}$, $T = \mathbb{R}/\mathbb{Z}$ or $\mathbb{R}$. Specifying these groups, we can initialize LCAs. Groups are specified by:

- Order, where 0 is taken to mean infinite order.
- Whether or not they are discrete (if not, they are continuous).

```
In [2]: # Create the group Z_1 + R + Z_3
        G = LCA(orders = [1, 0, 3],
                discrete = [True, False, True])

        print(G) # Standard printing
        show(G) # LaTeX output
```

```
[Z_1, R, Z_3]
```

$$\mathbb{Z}_1 \oplus \mathbb{R} \oplus \mathbb{Z}_3$$

If no `discrete` parameter is passed, `True` is assumed and the LCA initialized will be a finitely generated abelian group (FGA).

```
In [3]: # No 'discrete' argument passed,
        # so the initializer assumes a discrete group
        G = LCA(orders = [5, 11])
        show(G)

        G.is_FGA() # Check if this group is an FGA
```

$$\mathbb{Z}_5 \oplus \mathbb{Z}_{11}$$

```
Out[3]: True
```

Manipulating LCAs

One way to create LCAs is using the direct sum, which "glues" LCAs together.

```
In [4]: # Create two groups
        # Notice how the argument names can be omitted
        G = LCA([5, 11])
        H = LCA([7, 0], [True, True])

        # Take the direct sum of G and H
        # Two ways: explicitly and using the + operator
        direct_sum = G.sum(H)
        direct_sum = G + H

        show(G)
        show(H)
        show(direct_sum)
```

$$\mathbb{Z}_5 \oplus \mathbb{Z}_{11}$$

$$\mathbb{Z}_7 \oplus \mathbb{Z}$$

$$\mathbb{Z}_5 \oplus \mathbb{Z}_{11} \oplus \mathbb{Z}_7 \oplus \mathbb{Z}$$

Python comes with a powerful slice syntax. This can be used to "split up" LCAs. LCAs of lower length can be created by slicing, using the built-in slice notation in Python.

```
In [5]: # Return groups 0 to 3 (inclusive, exclusive)
        sliced = direct_sum[0:3]
        show(sliced)

        # Return the last two groups in the LCA
        sliced = direct_sum[-2:]
        show(sliced)
```

$$\mathbb{Z}_5 \oplus \mathbb{Z}_{11} \oplus \mathbb{Z}_7$$

$$\mathbb{Z}_7 \oplus \mathbb{Z}$$

Trivial groups can be removed automatically using `remove_trivial`. Recall that the trivial group is $\mathbb{Z}_1$.

```
In [6]: # Create a group with several trivial groups
        G = LCA([1, 1, 0, 5, 1, 7])
        show(G)

        # Remove trivial groups
        G_no_trivial = G.remove_trivial()
        show(G_no_trivial)
```

$$\mathbb{Z}_1 \oplus \mathbb{Z}_1 \oplus \mathbb{Z} \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_1 \oplus \mathbb{Z}_7$$

$$\mathbb{Z} \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_7$$

Checking if an LCA is a FGA

Recall that a group G is an FGA if all the groups in the direct sum are discrete.

```
In [7]: G = LCA([1, 5], discrete = [False, True])
        G.is_FGA()
```

```
Out[7]: False
```

If G is an FGA, elements can be generated by max-norm by an efficient algorithm. The algorithm is able to generate approximately 200000 elements per second, but scales exponentially with the free rank of the group.

```
In [8]: Z = LCA([0])
        for element in (Z**2).elements_by_maxnorm([0, 1]):
            print(element)
```

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

```
In [9]: Z_5 = LCA([5])
        for element in (Z_5**2).elements_by_maxnorm([0, 1]):
            print(element)
```

```
[0, 0]
[1, 4]
[4, 4]
[1, 0]
[4, 0]
[1, 1]
[4, 1]
[0, 1]
[0, 4]
```

Dual groups

The `dual()` method returns a group isomorphic to the Pontryagin dual.

```
In [10]: show(G)
         show(G.dual())
```

$$T \oplus \mathbb{Z}_5$$

$$\mathbb{Z} \oplus \mathbb{Z}_5$$

Iteration, containment and lengths

LCAs implement the Python iteration protocol, and they subclass the abstract base class (ABC) `Sequence`. A `Sequence` is a subclass of `Reversible` and `Collection` ABCs. These ABCs force the subclasses that inherit from them to implement certain behaviors, namely:

- Iteration over the object: this yields the LCAs in the direct sum one-by-one.
- The `G in H` statement: this checks whether G is a contained in H.
- The `len(G)` built-in, this check the length of the group.

We now show this behavior with examples.

```
In [11]: G = LCA([10, 1, 0, 0], [True, False, True, False])
```

```
        # Iterate over all subgroups in G
        for subgroup in G:
            dual = subgroup.dual()
            print('The dual of', subgroup, 'is', dual)

            # Print if the group is self dual
            if dual == subgroup:
                print('   ->', subgroup, 'is self dual')
```

```
The dual of [Z_10] is [Z_10]
   -> [Z_10] is self dual
The dual of [T] is [Z]
The dual of [Z] is [T]
The dual of [R] is [R]
   -> [R] is self dual
```

Containment

A LCA G is contained in H iff there exists an injection $\phi : G \to H$ such that every source/target of the mapping are isomorphic groups.

```
In [12]: # Create two groups
         G = LCA([1, 3, 5])
         H = LCA([3, 5, 1, 8])

         # Two ways, explicitly or using the `in` keyword
         print(G.contained_in(H))
         print(G in H)
```

```
True
True
```

The length can be computed using the `length()` method, or the built-in method `len`. In contrast with `rank()`, this does not remove trivial groups.

```
In [13]: # The length is available with the len built-in function
         # Notice that the length is not the same as the rank,
         # since the rank will remove trivial subgroups first
         G = LCA([1, 3, 5])
         show(G)

         print(G.length()) # Explicit
         print(len(G)) # Using the built-in len function
         print(G.rank())
```

$$\mathbb{Z}_1 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_5$$

```
3
3
2
```

Ranks and lengths of groups

The rank can be computed by the `rank()` method.

- The `rank()` method removes trivial subgroups.

- The `length()` method does not remove trivial subgroups.

```
In [14]: G = LCA([1, 5, 7, 0])
         show(G)
         G.rank()
```

$$\mathbb{Z}_1 \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_7 \oplus \mathbb{Z}$$

```
Out[14]: 3
```

Canonical forms and isomorphic groups

FGAs can be put into a canonical form using the Smith normal form (SNF). Two FGAs are isomorphic iff their canonical form is equal.

```
In [15]: G = LCA([1, 3, 3, 5, 8])
         show(G)
         show(G.canonical())
```

$$\mathbb{Z}_1 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_8$$

$$\mathbb{Z}_3 \oplus \mathbb{Z}_{120}$$

The groups $G = \mathbb{Z}_3 \oplus \mathbb{Z}_4$ and $H = \mathbb{Z}_{12}$ are isomorphic because they can be put into the same canonical form using the SNF.

```
In [16]: G = LCA([3, 4, 0])
         H = LCA([12, 0])
         G.isomorphic(H)
```

```
Out[16]: True
```

General LCAs are isomorphic if the FGAs are isomorphic and the remaining groups such as $\mathbb{R}$ and T can be obtained with a permutation. We show this by example.

```
In [17]: G = LCA([12, 13, 0], [True, True, False])
         H = LCA([12 * 13, 0], [True, False])
         show(G)
         show(H)
         G.isomorphic(H)
```

$$\mathbb{Z}_{12} \oplus \mathbb{Z}_{13} \oplus \mathbb{R}$$

$$\mathbb{Z}_{156} \oplus \mathbb{R}$$

```
Out[17]: True
```

Projecting elements to groups

It is possible to project elements onto groups.

```
In [18]: element = [8, 17, 7]
         G = LCA([10, 15, 20])
         G(element)
```

```
Out[18]: [8, 2, 7]
```

2.2.2 Tutorial: Homomorphisms

This is an interactive tutorial written with real code. We start by setting up LaTeX printing.

```
In [1]: from IPython.display import display, Math

        def show(arg):
            return display(Math(arg.to_latex()))
```

Initializing a homomorphism

Homomorphisms between general LCAs are represented by the `HomLCA` class. To define a homomorphism, a matrix representation is needed. In addition to the matrix, the user can also define a `target` and `source` explicitly.

Some verification of the inputs is performed by the initializer, for instance a matrix $A \in \mathbb{Z}^{2\times 2}$ cannot represent $\phi : \mathbb{Z}^m \to \mathbb{Z}^n$ unless both m and n are 2. If no `target`/`source` is given, the initializer will assume a free, discrete group, i.e. $\mathbb{Z}^m$.

```
In [2]: from abelian import LCA, HomLCA

        # Initialize the target group for the homomorphism
        target = LCA([0, 5], discrete = [False, True])

        # Initialize a homomorphism between LCAs
        phi = HomLCA([[1, 2], [3, 4]], target = target)
        show(phi)

        # Initialize a homomorphism with no source/target.
        # Source and targets are assumed to be
        # of infinite order and discrete (free-to-free)
        phi = HomLCA([[1, 2], [3, 4]])
        show(phi)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{R} \oplus \mathbb{Z}_5$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

Homomorphisms between finitely generated abelian groups (FGAs) are also represented by the `HomLCA` class.

```
In [3]: from abelian import HomLCA
        phi = HomLCA([[4, 5], [9, -3]])
        show(phi)
```

$$\begin{pmatrix} 4 & 5 \\ 9 & -3 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

Roughly speaking, for a `HomLCA` instance to represent a homomorphism between FGAs, it must have:

- FGAs as source and target.
- The matrix must contain only integer entries.

Compositions

A fundamental way to combine two functions is to compose them. We create two homomorphisms and compose them: first ψ, then ϕ. The result is the function $\phi \circ \psi$.

```
In [4]: # Create two HomLCAs
        phi = HomLCA([[4, 5], [9, -3]])
        psi = HomLCA([[1, 0, 1], [0, 1, 1]])

        # The composition of phi, then psi
        show(phi * psi)
```

$$\begin{pmatrix} 4 & 5 & 9 \\ 9 & -3 & 6 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

If the homomorphism is an endomorphism (same source and target), repeated composition can be done using exponents.

$\phi^n = \phi \circ \phi \circ \cdots \circ \phi, \quad n \geq 1$

```
In [5]: show(phi**3)
```

$$\begin{pmatrix} 289 & 290 \\ 522 & -117 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

Numbers and homomorphisms can be added to homomorphisms, in the same way that numbers and matrices are added to matrices in other software packages.

```
In [6]: show(psi)

        # Each element in the matrix is multiplied by 2
        show(psi + psi)

        # Element-wise addition
        show(psi + 10)
```

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

$$\begin{pmatrix} 2 & 0 & 2 \\ 0 & 2 & 2 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

$$\begin{pmatrix} 11 & 10 & 11 \\ 10 & 11 & 11 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

Slice notation

Slice notation is available. The first slice works on rows (target group) and the second slice works on columns (source group). Notice that in Python, indices start with 0.

```
In [7]: A = [[10, 10], [10, 15]]
        # Notice how the HomLCA converts a list
        # into an LCA, this makes it easier to create HomLCAs
        phi = HomLCA(A, target = [20, 20])
        phi = phi.project_to_source()

        # Slice in different ways
        show(phi)
        show(phi[0, :]) # First row, all columns
        show(phi[:, 0]) # All rows, first column
        show(phi[1, 1]) # Second row, second column
```

$$\begin{pmatrix} 10 & 10 \\ 10 & 15 \end{pmatrix} : \mathbb{Z}_2 \oplus \mathbb{Z}_4 \to \mathbb{Z}_{20} \oplus \mathbb{Z}_{20}$$

$$\begin{pmatrix} 10 & 10 \end{pmatrix} : \mathbb{Z}_2 \oplus \mathbb{Z}_4 \to \mathbb{Z}_{20}$$

$$\begin{pmatrix} 10 \\ 10 \end{pmatrix} : \mathbb{Z}_2 \to \mathbb{Z}_{20} \oplus \mathbb{Z}_{20}$$

$$\begin{pmatrix} 15 \end{pmatrix} : \mathbb{Z}_4 \to \mathbb{Z}_{20}$$

Stacking homomorphisms

There are three ways to stack morphisms:

- Diagonal stacking
- Horizontal stacking
- Vertical stacking

They are all shown below.

Diagonal stacking

```
In [8]: # Create two homomorphisms
        phi = HomLCA([2], target = LCA([0], [False]))
        psi = HomLCA([2])

        # Stack diagonally
        show(phi.stack_diag(psi))
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{R} \oplus \mathbb{Z}$$

Horizontal stacking

```
In [9]: # Create two homomorphisms with the same target
        target = LCA([0], [False])
        phi = HomLCA([[1, 3]], target = target)
        source = LCA([0], [False])
        psi = HomLCA([7], target=target, source=source)

        # Stack horizontally
        show(phi.stack_horiz(psi))
```

$$\begin{pmatrix} 1 & 3 & 7 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{R} \to \mathbb{R}$$

Vertical stacking

```
In [10]: # Create two homomorphisms, they have the same source
         phi = HomLCA([[1, 2]])
         psi = HomLCA([[3, 4]])

         # Stack vertically
         show(phi.stack_vert(psi))
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z} \oplus \mathbb{Z}$$

Calling homomorphisms

In Python, a `callable` is an object which implements a method for function calls. A homomorphism is a callable object, so we can use `phi(x)` to evaluate `x`, i.e. send `x` from the source to the target.

We create a homomorphism.

```
In [11]: # Create a homomorphism, specify the target
         phi = HomLCA([[2, 0], [0, 4]], [10, 12])
         # Find the source group (orders)
         phi = phi.project_to_source()
         show(phi)
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 4 \end{pmatrix} : \mathbb{Z}_5 \oplus \mathbb{Z}_3 \to \mathbb{Z}_{10} \oplus \mathbb{Z}_{12}$$

We can now call it. The argument must be in the source group.

```
In [12]: # An element in the source, represented as a list
         group_element = [1, 1]

         # Calling the homomorphism
         print(phi(group_element))

         # Since [6, 4] = [1, 1] mod [5, 3] (source group)
         # the following is equal
         print(phi([6, 4]) == phi([1, 1]))
```

```
[2, 4]
True
```

Calling and composing

We finish this tutorial by showing two ways to calculate the same thing:

- $y = (\phi \circ \psi)(x)$
- $y = \phi(\psi(x))$

```
In [13]: # Create two HomLCAs
         phi = HomLCA([[4, 5], [9, -3]])
         psi = HomLCA([[1, 0, 1], [0, 1, 1]])

         x = [1, 1, 1]
         # Compose, then call
         answer1 = (phi * psi)(x)

         # Call, then call again
         answer2 = phi(psi(x))

         # The result is the same
         print(answer1 == answer2)
```

```
True
```

2.2.3 Tutorial: Factoring homomorphisms

This is an interactive tutorial written with real code. We start by importing the `LCA` class, the `HomLCA` class and setting up LaTeX printing.

```
In [1]: from abelian import LCA, HomLCA
        from IPython.display import display, Math

        def show(arg):
            return display(Math(arg.to_latex()))
```

Initialization and source/target projections

We create a `HomLCA` instance, which may represent a homomorphism between FGAs. In this tutorial we will only consider homomorphisms between FGAs.

```
In [2]: phi = HomLCA([[5, 10, 15],
                      [10, 20, 30],
                      [10, 5, 30]],
                     target = [50, 20, 30])
        show(phi)
```

$$\begin{pmatrix} 5 & 10 & 15 \\ 10 & 20 & 30 \\ 10 & 5 & 30 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

Projecting to source

The source (or domain) is assumed to be free (infinite order). Calculating the orders is done with the `project_to_source` method, after which the orders of the columns are shown in the source group.

```
In [3]: # Project to source, i.e. orders of generator columns
        phi = phi.project_to_source()
        show(phi)
```

$$\begin{pmatrix} 5 & 10 & 15 \\ 10 & 20 & 30 \\ 10 & 5 & 30 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

Projecting to target

Projecting the columns onto the target group will make the morphism more readable. The `project_to_target()` method will project every column to the target group.

```
In [4]: # Project the generator columns to the target group
        phi = phi.project_to_target()
        show(phi)
```

$$\begin{pmatrix} 5 & 10 & 15 \\ 10 & 0 & 10 \\ 10 & 5 & 0 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

The kernel monomorphism

The kernel morphism is a monomorphism such that $\phi \circ \ker(\phi) = 0$. The kernel of ϕ is:

```
In [5]: # Calculate the kernel
        show(phi.kernel())
```

$$\begin{pmatrix} 25 & 29 & 28 \\ 10 & 2 & 28 \\ 5 & 9 & 2 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10}$$

The kernel monomorphism is not projected to source by default, but doing so is simple.

```
In [6]: show(phi.kernel().project_to_source())
```

$$\begin{pmatrix} 25 & 29 & 28 \\ 10 & 2 & 28 \\ 5 & 9 & 2 \end{pmatrix} : \mathbb{Z}_6 \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{15} \to \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10}$$

Verify that $\phi \circ \ker(\phi) = 0$.

```
In [7]: show(phi * phi.kernel())
```

$$\begin{pmatrix} 300 & 300 & 450 \\ 300 & 380 & 300 \\ 300 & 300 & 420 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

To clearly see that this is the zero morphism, use the `project_to_target()` method as such.

```
In [8]: zero = phi * phi.kernel()
        zero = zero.project_to_target()
        show(zero)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} : \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

The cokernel epimorphism

The kernel morphism is an epimorphism such that $\mathrm{coker}(\phi) \circ \phi = 0$. The cokernel of ϕ is:

```
In [9]: show(phi.cokernel())
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 4 \\ 18 & 17 & 4 \end{pmatrix} : \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30} \to \mathbb{Z}_5 \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_{20}$$

We verify the factorization.

```
In [10]: show((phi.cokernel() * phi))
```

$$\begin{pmatrix} 5 & 10 & 15 \\ 50 & 20 & 10 \\ 300 & 200 & 440 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_5 \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_{20}$$

Again it is not immediately clear that this is the zero morphism. To verify this, we again use the `project_to_target()` method as such.

```
In [11]: zero = phi.cokernel() * phi
         zero = zero.project_to_target()

         show(zero)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_5 \oplus \mathbb{Z}_5 \oplus \mathbb{Z}_{20}$$

The image/coimage factorization

The image/coimage factorization is $\phi = \mathrm{im}(\phi) \circ \mathrm{coim}(\phi)$, where the image is a monomorphism and the coimage is an epimorphism.

The image monomorphism

Finding the image is easy, just call the `image()` method.

```
In [12]: im = phi.image()
         show(im)
```

$$\begin{pmatrix} 0 & 25 & 40 \\ 0 & 10 & 0 \\ 0 & 0 & 25 \end{pmatrix} : \mathbb{Z}_1 \oplus \mathbb{Z}_2 \oplus \mathbb{Z}_{30} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

A trivial group $\mathbb{Z}_1$ is in the source. It can be removed using `remove_trivial_subgroups()`.

```
In [13]: im = im.remove_trivial_groups()
         show(im)
```

$$\begin{pmatrix} 25 & 40 \\ 10 & 0 \\ 0 & 25 \end{pmatrix} : \mathbb{Z}_2 \oplus \mathbb{Z}_{30} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

The coimage epimorphism

Finding the coimage is done by calling the `coimage()` method.

```
In [14]: coim = phi.coimage().remove_trivial_groups()
         show(coim)
```

$$\begin{pmatrix} 1 & 0 & 1 \\ 22 & 29 & 6 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_2 \oplus \mathbb{Z}_{30}$$

Verify the image/coimage factorization

We now verify that $\phi = \mathrm{im}(\phi) \circ \mathrm{coim}(\phi)$.

```
In [15]: show(phi)
         show((im * coim).project_to_target())
```

$$\begin{pmatrix} 5 & 10 & 15 \\ 10 & 0 & 10 \\ 10 & 5 & 0 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

$$\begin{pmatrix} 5 & 10 & 15 \\ 10 & 0 & 10 \\ 10 & 5 & 0 \end{pmatrix} : \mathbb{Z}_{30} \oplus \mathbb{Z}_{30} \oplus \mathbb{Z}_{10} \to \mathbb{Z}_{50} \oplus \mathbb{Z}_{20} \oplus \mathbb{Z}_{30}$$

```
In [16]: (im * coim).project_to_target() == phi
```

```
Out[16]: True
```

2.2.4 Tutorial: Functions on LCAs

This is an interactive tutorial written with real code. We start by setting up LaTeX printing, and importing the classes `LCA`, `HomLCA` and `LCAFunc`.

```
In [1]: # Imports from abelian
        from abelian import LCA, HomLCA, LCAFunc

        # Other imports
        import math
        import matplotlib.pyplot as plt
        from IPython.display import display, Math

        def show(arg):
            return display(Math(arg.to_latex()))
```

Initializing a new function

There are two ways to create a function $f : G \to \mathbb{C}$:

- On general LCAs G, the function is represented by an **analytical expression**.
- If $G = \mathbb{Z}_{\mathbf{p}}$ with $p_i \geq 1$ for every i (G is a direct sum of discrete groups with finite period), a **table of values** (multidimensional array) can also be used.

With an analytical representation

If the representation of the function is given by an analytical expression, initialization is simple.

Below we define a Gaussian function on $\mathbb{Z}$, and one on T.

```
In [2]: def gaussian(vector_arg, k = 0.1):
            return math.exp(-sum(i**2 for i in vector_arg)*k)

        # Gaussian function on Z
        Z = LCA([0])
        gauss_on_Z = LCAFunc(gaussian, domain = Z)
        print(gauss_on_Z) # Printing
        show(gauss_on_Z) # LaTeX output

        # Gaussian function on T
        T = LCA([1], [False])
        gauss_on_T = LCAFunc(gaussian, domain = T)
        show(gauss_on_T) # LaTeX output
```

```
LCAFunc on domain [Z]
```

$$\text{function} \in \mathbb{C}^G,\ G = \mathbb{Z}$$

$$\text{function} \in \mathbb{C}^G,\ G = T$$

Notice how the `print` built-in and the `to_latex()` method will show human-readable output.

With a table of values

Functions on $\mathbb{Z}_{\mathbf{p}}$ can be defined using a table of values, if $p_i \geq 1$ for every $p_i \in \mathbf{p}$.

```
In [3]: # Create a table of values
        table_data = [[1,2,3,4,5],
                      [2,3,4,5,6],
                      [3,4,5,6,7]]

        # Create a domain matching the table
        domain = LCA([3, 5])

        table_func = LCAFunc(table_data, domain)
        show(table_func)
        print(table_func([1, 1])) # [1, 1] maps to 3
```

$$\text{function} \in \mathbb{C}^G,\ G = \mathbb{Z}_3 \oplus \mathbb{Z}_5$$

```
3
```

Function evaluation

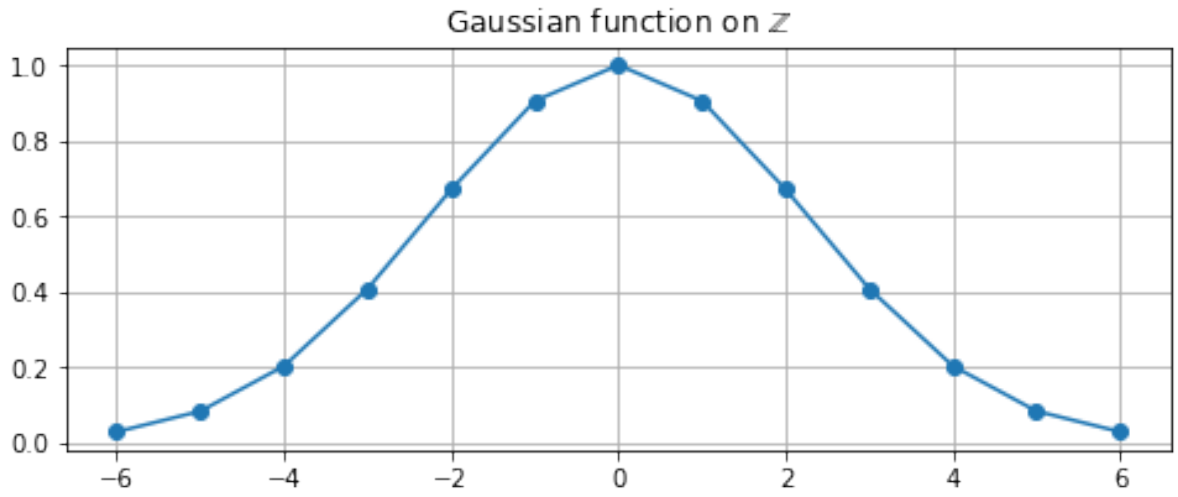
A function $f \in \mathbb{C}^G$ is callable. To call (i.e. evaluate) a function, pass a group element.

```
In [4]: # An element in Z
        element = [0]

        # Evaluate the function
        gauss_on_Z(element)
```

```
Out[4]: 1.0
```

The `sample()` method can be used to sample a function on a list of group elements in the domain.

```
In [5]: # Create a list of sample points [-6, ..., 6]
        sample_points = [[i] for i in range(-6, 7)]

        # Sample the function, returns a list of values
        sampled_func = gauss_on_Z.sample(sample_points)

        # Plot the result of sampling the function
        plt.figure(figsize = (8, 3))
        plt.title('Gaussian function on $\mathbb{Z}$')
        plt.plot(sample_points, sampled_func, '-o')
        plt.grid(True)
        plt.show()
```



Shifts

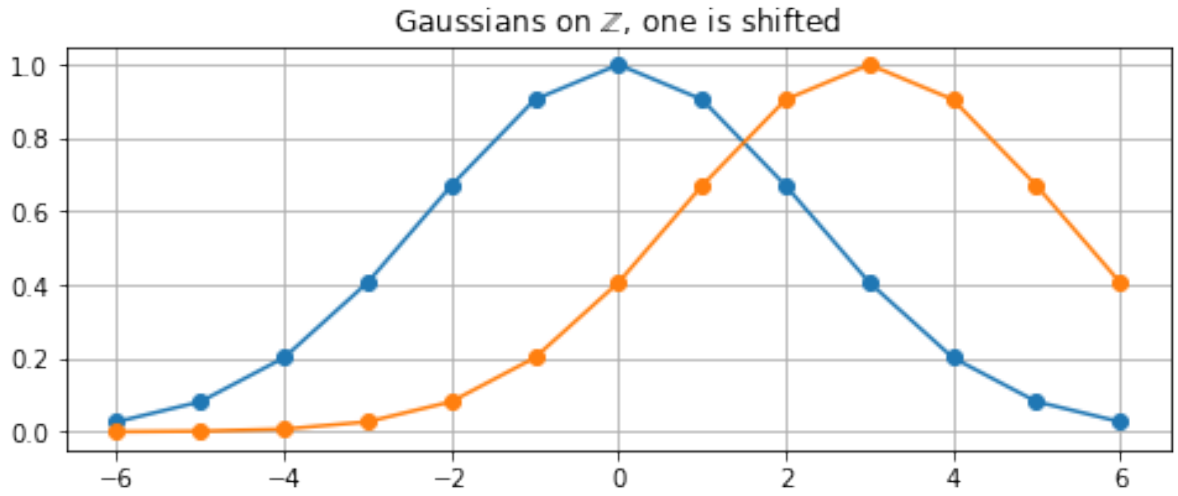
Let $f : G \to \mathbb{C}$ be a function. The shift operator (or translation operator) S_h is defined as

$$S_h[f(g)] = f(g - h).$$

The shift operator shifts $f(g)$ by h, where $h, g \in G$.

The shift operator is implemented as a method called `shift`.

```
In [6]: # The group element to shift by
        shift_by = [3]

        # Shift the function
        shifted_gauss = gauss_on_Z.shift(shift_by)

        # Create sample poits and sample
        sample_points = [[i] for i in range(-6, 7)]
        sampled1 = gauss_on_Z.sample(sample_points)
        sampled2 = shifted_gauss.sample(sample_points)

        # Create a plot
        plt.figure(figsize = (8, 3))
        ttl = 'Gaussians on $\mathbb{Z}$, one is shifted'
        plt.title(ttl)
        plt.plot(sample_points, sampled1, '-o')
        plt.plot(sample_points, sampled2, '-o')
        plt.grid(True)
        plt.show()
```



Pullbacks

Let $\phi : G \to H$ be a homomorphism and let $f : H \to \mathbb{C}$ be a function. The pullback of f along ϕ, denoted $\phi^*(f)$, is defined as

$$\phi^*(f) := f \circ \phi.$$

The pullback "moves" the domain of the function f to G, i.e. $\phi^*(f) : G \to \mathbb{C}$. The pullback is of `f` is calculated using the `pullback` method, as shown below.

```
In [7]: def linear(arg):
            return sum(arg)

        # The original function
        f = LCAFunc(linear, LCA([10]))
        show(f)

        # A homomorphism phi
        phi = HomLCA([2], target = [10])
        show(phi)

        # The pullback of f along phi
        g = f.pullback(phi)
        show(g)
```

$$\text{function} \in \mathbb{C}^G,\ G = \mathbb{Z}_{10}$$

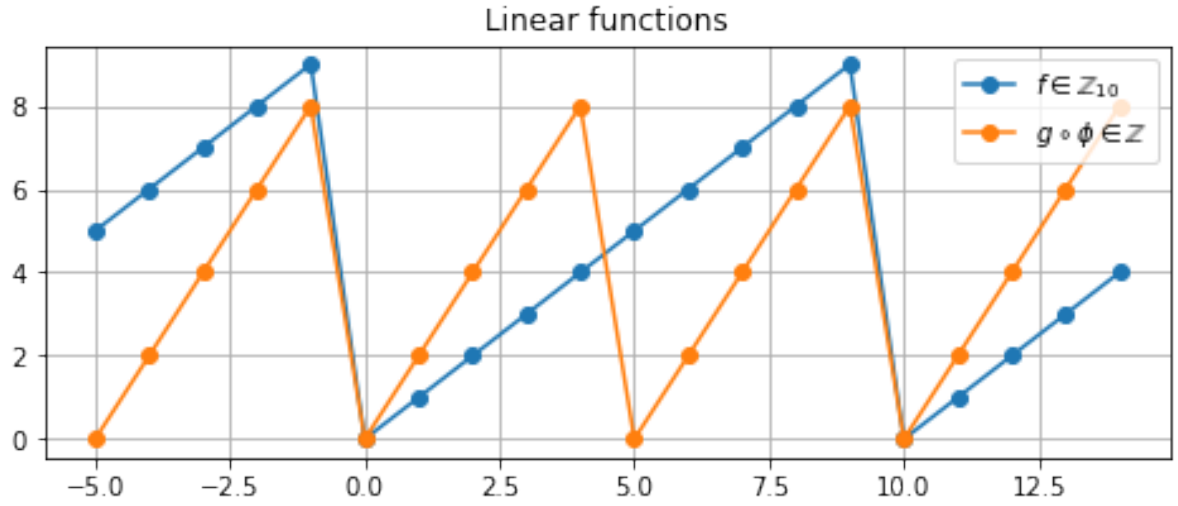
$$(2) : \mathbb{Z} \to \mathbb{Z}_{10}$$

$$\text{function} \in \mathbb{C}^G,\ G = \mathbb{Z}$$

We now sample the functions and plot them.

```
In [8]: # Sample the functions and plot them
        sample_points = [[i] for i in range(-5, 15)]
        f_sampled = f.sample(sample_points)
        g_sampled = g.sample(sample_points)

        # Plot the original function and the pullback
        plt.figure(figsize = (8, 3))
        plt.title('Linear functions')
```

```
label = '$f \in \mathbb{Z}_{10}$'
plt.plot(sample_points, f_sampled, '-o', label = label)
label = '$g \circ \phi \in \mathbb{Z}$'
plt.plot(sample_points, g_sampled, '-o', label = label)
plt.grid(True)
plt.legend(loc = 'best')
plt.show()
```



Pushforwards

Let $\phi : G \to H$ be a epimorphism and let $f : G \to \mathbb{C}$ be a function. The pushforward of f along ϕ, denoted $\phi_*(f)$, is defined as

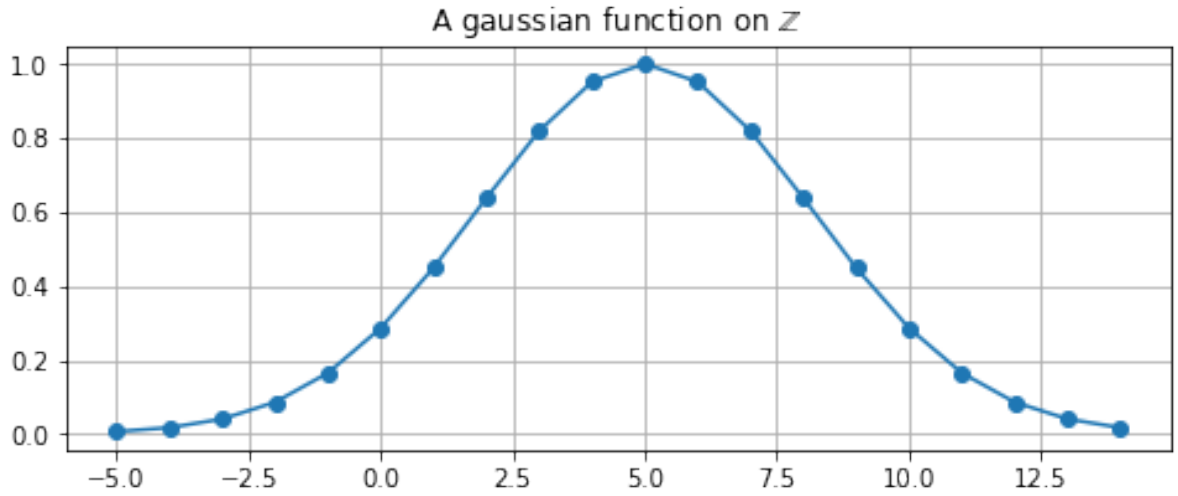
$$(\phi_*(f))(g) := \sum_{k \in \ker \phi} f(k + h), \quad \phi(g) = h$$

The pullback "moves" the domain of the function f to H, i.e. $\phi_*(f) : H \to \mathbb{C}$. First a solution is obtained, then we sum over the kernel. Since such a sum may contain an infinite number of terms, we bound it using a norm. Below is an example where we:

- Define a Gaussian $f(x) = \exp(-kx^2)$ on $\mathbb{Z}$
- Use pushforward to "move" it with $\phi(g) = g \in \mathrm{Hom}(\mathbb{Z}, \mathbb{Z}_{10})$

```
In [9]: # We create a function on Z and plot it
        def gaussian(arg, k = 0.05):
            """
            A gaussian function.
            """
            return math.exp(-sum(i**2 for i in arg)*k)

        # Create gaussian on Z, shift it by 5
        gauss_on_Z = LCAFunc(gaussian, LCA([0]))
        gauss_on_Z = gauss_on_Z.shift([5])

        # Sample points and sampled function
        s_points = [[i] for i in range(-5, 15)]
        f_sampled = gauss_on_Z.sample(s_points)

        # Plot it
```

```
plt.figure(figsize = (8, 3))
plt.title('A gaussian function on $\mathbb{Z}$')
plt.plot(s_points, f_sampled, '-o')
plt.grid(True)
plt.show()
```

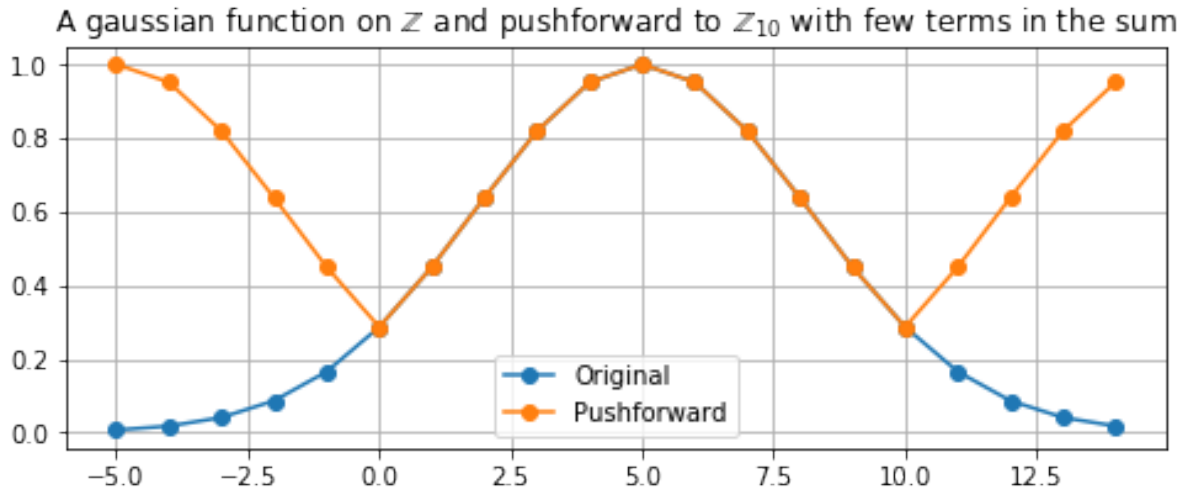



```
In [10]: # Use a pushforward to periodize the function
         phi = HomLCA([1], target = [10])
         show(phi)
```

$$(1) : \mathbb{Z} \to \mathbb{Z}_{10}$$

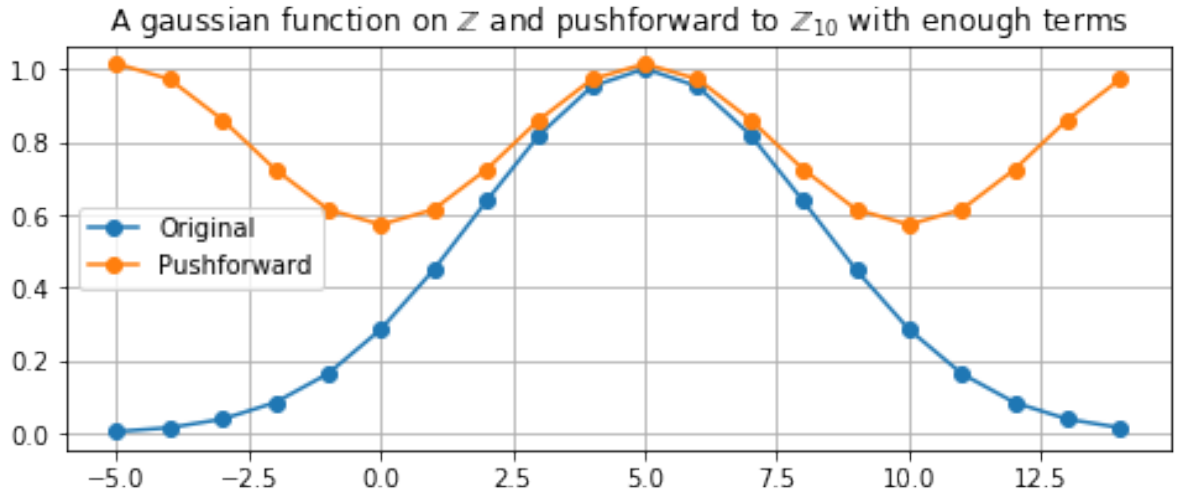
First we do a pushforward with only one term. **Not enough terms are present** in the sum to capture what the pushforward would look like if the sum went to infinity.

```
In [11]: terms = 1

         # Pushforward of the function along phi
         gauss_on_Z_10 = gauss_on_Z.pushforward(phi, terms)

         # Sample the functions and plot them
         pushforward_sampled = gauss_on_Z_10.sample(sample_points)

         plt.figure(figsize = (8, 3))
         label = 'A gaussian function on $\mathbb{Z}$ and \
         pushforward to $\mathbb{Z}_{10}$ with few terms in the sum'
         plt.title(label)
         plt.plot(s_points, f_sampled, '-o', label ='Original')
         plt.plot(s_points, pushforward_sampled, '-o', label ='Pushforward')
         plt.legend(loc = 'best')
         plt.grid(True)
         plt.show()
```



Next we do a pushforward with more terms in the sum, this captures what the pushforward would look like if the sum went to infinity.

```
In [12]: terms = 9

         gauss_on_Z_10 = gauss_on_Z.pushforward(phi, terms)

         # Sample the functions and plot them
         pushforward_sampled = gauss_on_Z_10.sample(sample_points)

         plt.figure(figsize = (8, 3))
         plt.title('A gaussian function on $\mathbb{Z}$ and \
         pushforward to $\mathbb{Z}_{10}$ with enough terms')
         plt.plot(s_points, f_sampled, '-o', label ='Original')
         plt.plot(s_points, pushforward_sampled, '-o', label ='Pushforward')
         plt.legend(loc = 'best')
         plt.grid(True)
         plt.show()
```



2.2.5 Tutorial: Fourier series

This is an interactive tutorial written with real code. We start by setting up LaTeX printing and importing some classes.

```
In [1]: # Imports related to plotting and LaTeX
        import matplotlib.pyplot as plt
        %matplotlib inline
        from IPython.display import display, Math
        from IPython.display import set_matplotlib_formats
        set_matplotlib_formats('pdf', 'png')
        def show(arg):
            return display(Math(arg.to_latex()))
```

```
In [2]: # Imports related to mathematics
        import numpy as np
        from abelian import LCA, HomLCA, LCAFunc
        from sympy import Rational, pi
```

Overview: $f(x) = x$ defined on $T = \mathbb{R}/\mathbb{Z}$

In this example we compute the Fourier series coefficients for $f(x) = x$ with domain $T = \mathbb{R}/\mathbb{Z}$.

We will proceed as follows:

1. Define a function $f(x) = x$ on T.
2. Sample using pullback along $\phi_{\text{sample}} : \mathbb{Z}_n \to T$. Specifically, we will use $\phi(n) = 1/n$ to sample uniformly.
3. Compute the DFT of the sampled function using the `dft` method.
4. Use a transversal rule to move the DFT from $\mathbb{Z}_n$ to $\widehat{T} = \mathbb{Z}$.
5. Plot the result and compare with the analytical solution, which can be obtained by computing the complex Fourier coefficients of the Fourier integral by hand.

We start by defining the function on the domain.

Defining the function

```
In [3]: def identity(arg_list):
            return sum(arg_list)

        # Create the domain T and a function on it
        T = LCA(orders = [1], discrete = [False])
        function = LCAFunc(identity, T)
        show(function)
```

$$\text{function} \in \mathbb{C}^G,\ G = T$$

We now create a monomorphism ϕ_{sample} to sample the function, where we make use of the `Rational` class to avoid numerical errors.

Sampling using pullback

```
In [4]: # Set up the number of sample points
        n = 8

        # Create the source of the monomorphism
        Z_n = LCA([n])
        phi_sample = HomLCA([Rational(1, n)],T, Z_n)
        show(phi_sample)
```

$$\left(\tfrac{1}{8}\right) : \mathbb{Z}_8 \to T$$

We sample the function using the pullback.

```
In [5]: # Pullback along phi_sample
        function_sampled = function.pullback(phi_sample)
```

Then we compute the DFT (discrete Fourier transform). The DFT is available on functions defined on $\mathbb{Z}_{\mathbf{p}}$ with $p_i \geq 1$, i.e. on FGAs with finite orders.

The DFT

```
In [6]: # Take the DFT (a multidimensional FFT is used)
        function_sampled_dual = function_sampled.dft()
```

Transversal

We use a transversal rule, along with $\widehat{\phi}_{\text{sample}}$, to push the function to $\widehat{T} = \mathbb{Z}$.

```
In [7]: # Set up a transversal rule
        def transversal_rule(arg_list):
            x = arg_list[0] # First element of vector/list
            if x < n/2:
                return [x]
            else:
                return [x - n]

        # Calculate the Fourier series coefficients
        phi_d = phi_sample.dual()
        rule = transversal_rule
        coeffs = function_sampled_dual.transversal(phi_d, rule)
        show(coeffs)
```

$$\text{function} \in \mathbb{C}^G,\ G = \mathbb{Z}$$

Comparing with analytical solution

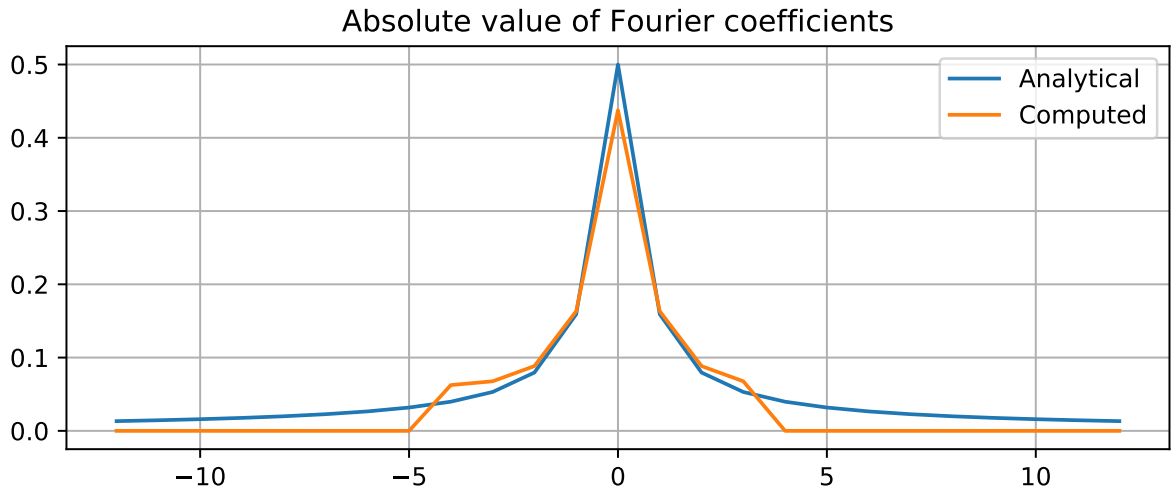
Let us compare this result with the analytical solution, which is

$$c_k = \begin{cases} 1/2 & \text{if } k = 0 \\ -1/2\pi i k & \text{else.} \end{cases}$$

```
In [8]: # Set up a function for the analytical solution
        def analytical(k):
            if k == 0:
                return 1/2
            return complex(0, 1)/(2*pi*k)

        # Sample the analytical and computed functions
        sample_values = list(range(-int(1.5*n), int(1.5*n)+1))
        analytical_sampled = list(map(analytical, sample_values))
        computed_sampled = coeffs.sample(sample_values)

        # Because the forward DFT does not scale, we scale manually
        computed_sampled = [k/n for k in computed_sampled]
```

Finally, we create the plot comparing the computed coefficients with the ones obtained analytically. Notice how the computed values drop to zero outside of the transversal region.

```
In [9]: # Since we are working with complex numbers
        # and we wish to plot them, we convert
        # to absolute values first
        length = lambda x: float(abs(x))
        analytical_abs = list(map(length, analytical_sampled))
        computed_abs = list(map(length, computed_sampled))

        # Plot it
        plt.figure(figsize = (8,3))
        plt.title('Absolute value of Fourier coefficients')
        plt.plot(sample_values, analytical_abs, label = 'Analytical')
        plt.plot(sample_values, computed_abs, label = 'Computed')
        plt.grid(True)
        plt.legend(loc = 'best')
        plt.show()
```



2.3 API

2.3.1 Library structure

The `abelian` library consists of two packages, `abelian` and the `abelian.linalg` sub-package.

- `abelian` - Provides access to high-level mathematical objects: LCAs, homomorphisms between LCAs and functions from an LCA to the complex numbers.
  - `abelian.linalg` - Lower-level linear algebra routines. Most notably the Hermite normal form, the Smith normal form, an equation solver for the equation $Ax = b \ mod \ p$ over the integers, as well as functions for generating elements of a finitely generated abelian group (FGA) ordered by maximum-norm.

2.3.2 Full API

abelian

abelian package

Subpackages

abelian.linalg package

Submodules

abelian.linalg.factorizations module

This module contains factorization algorithms for matrices over the integers. All the inputs and outputs are of type `MutableDenseMatrix`.

hermite_normal_form(*A*)

Compute U and H such that A*U = H.

This algorithm computes the column version of the Hermite normal form, and returns a tuple of matrices (U, H) such that A*U = H. The matrix U is an unimodular transformation matrix and H is the result of the transformation, i.e. H is in Hermite normal form.

Parameters **A** (`MutableDenseMatrix`) – The matrix to decompose.

Returns

- **U** (`MutableDenseMatrix`) – An unimodular matrix, i.e. integer matrix with determinant +/- 1.
- **H** (`MutableDenseMatrix`) – A matrix in Hermite normal form.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],
...             [3, 4]])
>>> U, H = hermite_normal_form(A)
>>> # Verify that U is unimodular (determinant +/- 1)
>>> U.det() in [1, -1]
True
>>> # Verify the decomposition
>>> A*U == H
True
```

smith_normal_form(*A*, *compute_unimod=True*)

Compute U,S,V such that U*A*V = S.

This algorithm computes the Smith normal form of an integer matrix. If *compute_unimod* is True, it returns matrices (U, S, V) such that U*A*V = S, where U and V are unimodular and S is in Smith normal form. If *compute_unimod* is False, it returns S and does not compute U and V.

Parameters

- **A** (MutableDenseMatrix) – The matrix to factor.
- **compute_unimod** (*bool*) – Whether or not to compute and return unimodular matrices U and V.

Returns

- **U** (MutableDenseMatrix) – An unimodular matrix, i.e. integer matrix with determinant +/- 1.
- **S** (MutableDenseMatrix) – A matrix in Smith normal form.
- **V** (MutableDenseMatrix) – An unimodular matrix, i.e. integer matrix with determinant +/- 1.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],
...             [3, 4]])
>>> U, S, V = smith_normal_form(A)
>>> # Verify that U and V are both unimodular
>>> U.det() in [1, -1] and V.det() in [1, -1]
True
>>> # Verify the factorization
>>> U * A * V == S
True
>>> # Compute without U and V, verify that the result is the same
>>> K = smith_normal_form(A, compute_unimod=False)
>>> K == S
True
```

abelian.linalg.factorizations_reals module

This module contains functions which calculate mapping properties of homomorphisms between R^n and R^m using the singular value decomposition (SVD). All the inputs and outputs are of type MutableDenseMatrix.

numerical_SVD(*A*)

Compute U,S,V such that U*S*V = A.

The input is converted to numerical data, the SVD is computed using the np.linalg.svd routine, which wraps the LAPACK routine _gesdd. The data is then converted to a sympy matrix and returned.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns

- **U** (MutableDenseMatrix) – A (close to) orthogonal sympy matrix.
- **S** (MutableDenseMatrix) – A diagonal sympy matrix matrix.
- **V** (MutableDenseMatrix) – A (close to) orthogonal sympy matrix.

Examples

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> U, S, V = numerical_SVD(A)
>>> # U is orthogonal (up to machine precision or so)
>>> abs(abs(U.det()) - 1) < 10e-10
True
>>> # Verify that the decomposition is close to the original
>>> sum(abs(k) for k in (U*S*V - A))  < 10e-10
True
```

numerical_rank(*A*)

Convert to numerical matrix and compute rank.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **r** – The rank of A.

Return type int

Examples

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> numerical_rank(A)
2
>>> A = Matrix([[0, 0], [0, 10e-10]])
>>> numerical_rank(A)
1
```

real_coimage(*A*)

Find the coimage of A, when the entries are real.

Converts the matrix to a numerical input, computes the SVD, finds the coimage epimorphism (row space of A), converts back to a sympy-matrix and returns.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **K** – The coimage of A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0, 0],
...             [0, 1, 0]])
>>> im = real_image(A)
>>> coim = real_coimage(A)
>>> # Verify the decomposition
>>> sum(abs(k) for k in (A - im * coim)) < 10e-15
True
```

real_cokernel(*A*)

Find the cokernel of A, when the entries are real.

Converts the matrix to a numerical input, computes the SVD, finds the cokernel epimorphism (null space of A^T), converts back to a sympy-matrix and returns.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **K** – The cokernel of A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0],
...             [0, 1],
...             [2, 2]])
>>> coker = real_cokernel(A)
>>> # Verify the decomposition
>>> sum(abs(k) for k in (coker * A)) < 10e-15
True
```

real_image(*A*)

Find the image of A, when the entries are real.

Converts the matrix to a numerical input, computes the SVD, finds the image monomorphism (column space), converts back to a sympy-matrix and returns.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **K** – The image of A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0],
...             [0, 1],
...             [1, 1]])
>>> im = real_image(A)
>>> coim = real_coimage(A)
>>> # Verify the decomposition
>>> sum(abs(k) for k in (A - im * coim)) < 10e-15
True
```

real_kernel(*A*)

Find the kernel of A, when the entries are real.

Converts the matrix to a numerical input, computes the SVD, finds the kernel monomorphism (null space of A), converts back to a sympy-matrix and returns.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **K** – The kernel of A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0, 1],
...             [0, 1, 1],
...             [2, 2, 4]])
>>> ker = real_kernel(A)
>>> # Verify the decomposition
>>> sum(abs(k) for k in (A * ker)) < 10e-15
True
```

abelian.linalg.free_to_free module

This module contains functions which calculate mapping properties of free-to-free homomorphisms. All the inputs and outputs are of type `MutableDenseMatrix`.

elements_increasing_norm(*free_rank*, *end_value=None*)

Continually yield every element in Z^r of increasing max-norm.

Parameters **free_rank** (*int*) – The free rank (like dimension) of Z^r, i.e. free_rank = r.

Yields *tuple* – Elements in Z^r with increasing maxnorm.

Examples

```
>>> free_rank = 2 # Like dimension
>>> for count, element in enumerate(elements_increasing_norm(free_
↪rank)):
...     if count >= 9:
...         break
...     print(count, element, max(abs(k) for k in element))
0 (0, 0) 0
1 (1, -1) 1
2 (-1, -1) 1
3 (1, 0) 1
4 (-1, 0) 1
5 (1, 1) 1
6 (-1, 1) 1
7 (0, 1) 1
8 (0, -1) 1
```

elements_of_maxnorm(*free_rank*, *maxnorm_value*)

Yield every element of Z^r such that max_norm(element) = maxnorm_value.

Parameters

- **free_rank** (*int*) – The free rank (like dimension) of Z^r, i.e. free_rank = r.
- **maxnorm_value** (*int*) – The value of the maximum norm of the elements generated.

Yields *tuple* – Elements in Z^r that satisfy the norm criterion.

Examples

```
>>> free_rank = 3 # Like dimension
>>> maxnorm_value = 4
>>> elements = list(elements_of_maxnorm(free_rank, maxnorm_value))
>>> # Verify that the max norm is the correct value
>>> all(max(abs(k) for k in e) for e in elements)
True
>>> # Verify the number of elements
>>> n = maxnorm_value
>>> len(elements) == ((2*n + 1)**free_rank - (2*n - 1)**free_rank)
True
```

elements_of_maxnorm_FGA(*orders*, *maxnorm_value*)

Yield every element of Z_'orders' such that max_norm(element) = maxnorm_value.

Parameters

- **orders** (*list*) – Orders in Z_orders, where 0 means infinite order, i.e. [2, 0] is Z_2 + Z.
- **maxnorm_value** (*int*) – The value of the maximum norm of the elements generated.

Yields *tuple* – Elements in Z_orders that satisfy the norm criterion.

Examples

```
>>> orders = [0, 0]
>>> norm_value = 1
>>> elements = list(elements_of_maxnorm_FGA(orders, norm_value))
>>> len(elements)
8
>>> orders = [0, 3]
>>> norm_value = 2
>>> for element in elements_of_maxnorm_FGA(orders, norm_value):
...     print(element)
(2, 2)
(-2, 2)
(2, 0)
(-2, 0)
(2, 1)
(-2, 1)
```

free_coimage(*A*)

Computes the free-to-free coimage epimorphism of A.

Let $A : \mathbb{Z}^n -> \mathbb{Z}^m$ be a homomorphism from a free (infinite order) finitely generated Abelian group (FGA) to another free FGA. Associated with this homomorphism is the coimage epimorphism. The coimage epimorphism has the property that A factors through the composition of the coimage and image morphisms, i.e. $\mathrm{im}(A) \circ \mathrm{coim}(A) = A$.

Parameters **A** (MutableDenseMatrix) – A sympy integer matrix.

Returns **coim_A** – The coimage epimorphism associated with A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0],
...             [0, 1],
...             [1, 1]])
>>> # Clearly the image is A itself, so coim(A) must be I
>>> free_coimage(A) == Matrix.eye(2)
True
>>> # Verify the image(A) * coimage(A) = A factorization
>>> free_image(A) * free_coimage(A) == A
True
```

free_cokernel(*A*)

Computes the free-to-free cokernel epimorphism of A.

Let $A : \mathbb{Z}^n -> \mathbb{Z}^m$ be a homomorphism from a free (infinite order) finitely generated Abelian group (FGA) to another free FGA. Associated with this homomorphism is the cokernel epimorphism. The cokernel epimorphism has the property that $\mathrm{coker}(A) \circ A = \mathbf{0}$, where $\mathbf{0}$ denotes the zero morphism.

Parameters **A** (MutableDenseMatrix) – A sympy integer matrix.

Returns **coker_A** – The cokernel epimorphism associated with *A*.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> from abelian.linalg.utils import matrix_mod_vector
>>> A = Matrix([[1, 0],
...             [0, 1],
...             [1, 1]])
>>> coker_A = free_cokernel(A)
>>> quotient = free_quotient(A)
>>> # Compute coker(A) * A and verify that it's 0 in the
>>> # target group of coker(A).
>>> product = matrix_mod_vector(coker_A * A, quotient)
>>> product == 0 * product
True
```

free_image(*A*)

Computes the free-to-free image monomorphism of A.

Let $A : \mathbb{Z}^n -> \mathbb{Z}^m$ be a homomorphism from a free (infinite order) finitely generated Abelian group (FGA) to another free FGA. Associated with this homomorphism is the image monomorphism. The image monomorphism has the property that A factors through the composition of the coimage and image morphisms, i.e. $\mathrm{im}(A) \circ \mathrm{coim}(A) = A$.

Parameters **A** (MutableDenseMatrix) – A sympy integer matrix.

Returns **im_A** – The image monomorphism associated with *A*.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0, 1],
...             [0, 1, 1]])
>>> # Clearly the image is the identity matrix
>>> free_image(A) == Matrix.eye(2)
True
>>> # Verify the image(A) * coimage(A) = A factorization
>>> free_image(A) * free_coimage(A) == A
True
```

free_kernel(*A*)

Computes the free-to-free kernel monomorphism of A.

Let $A : \mathbb{Z}^n -> \mathbb{Z}^m$ be a homomorphism from a free (infinite order) finitely generated Abelian group (FGA) to another free FGA. Associated with this homomorphism is the kernel monomorphism. The kernel monomorphism has the property that $A \circ \ker(A) = \mathbf{0}$, where $\mathbf{0}$ denotes the zero morphism.

Parameters **A** (MutableDenseMatrix) – A sympy integer matrix.

Returns **ker_A** – The kernel monomorphism associated with A.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 0, 1],
...             [0, 1, 1]])
>>> ker_A = free_kernel(A)
>>> # Verify the factorization
>>> A * ker_A == Matrix([0, 0])
True
```

free_quotient(*A*)

Compute the quotient group Z^m / im(A).

Let $A : \mathbb{Z}^n -> \mathbb{Z}^m$ be a homomorphism from a free (infinite order) finitely generated Abelian group (FGA) to another free FGA. Associated with this homomorphism is the cokernel epimorphism, which maps from $A : \mathbb{Z}^n$ to $A : \mathbb{Z}^m / \operatorname{im}(A)$.

Parameters **A** (MutableDenseMatrix) – A sympy integer matrix.

Returns **quotient** – The structure of the quotient group target(A)/im(A).

Return type MutableDenseMatrix

Examples

```
>>> from sympy import diag
>>> A = diag(1, 2, 3)
>>> free_quotient(A) == Matrix([1, 1, 6])
True
```

mod(*a*, *b*)

Mod for integers, tuples and lists.

Parameters

- **a** (*int, tuple or list*) – The argument.
- **b** (*int, tuple or list*) – The order.

Returns A mod b.

Return type int, tuple or list

Examples

```
>>> mod(7, 5) # Integer data
2
>>> mod((5, 8), (4, 4)) # Tuple data
(1, 0)
>>> mod([5, 8], [4, 4]) # List data
[1, 0]
```

abelian.linalg.solvers module

This module contains equation solvers. All the inputs and outputs are of type MutableDenseMatrix.

solve(*A*, *b*, *p=None*)

Solve eqn Ax = b mod p over Z.

The data (A, b, p) must be integer. The equation Ax = b mod p is solved, if a solution exists. If A is an epimorphism but not a monomorphism (i.e. overdetermined), one of the possible solutions is returned. If A is a monomorphism but not an epimorphism (i.e. underdetermined), a solution will be returned if one exists. If there is no solution, None is returned.

Parameters

- **A** (MutableDenseMatrix) – A sympy integer matrix of size m x n.
- **b** (MutableDenseMatrix) – A sympy column matrix of size m x 1.
- **p** (MutableDenseMatrix) – A sympy column matrix of size m x 1. This column matrix represents the orders of the target group of A. If None, p will be set to the zero vector, i.e. infinite order in all components.

Returns **x** – A solution to A*x = b mod p, where x is of size n x 1. If no solution is found, None is returned.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> from abelian.linalg.utils import vector_mod_vector
>>> A = Matrix([[5, 0, 3],
...             [0, 3, 4]])
>>> x = Matrix([2, -1, 2])
>>> p = Matrix([9, 9])
>>> b = vector_mod_vector(A*x, p)
>>> x_sol = solve(A, b, p)
>>> vector_mod_vector(A*x_sol, p) == b
True
```

solve_epi(*A*, *B*, *p=None*)

Solve the equation X * mod p * A = B, where A is an epimorphism.

The algorithm will produce a solution if (mod p * A) has a one sided inverse such that A_inv * A = I, i.e. A is an epimorphism.

Parameters

- **A** (MutableDenseMatrix) – A sympy integer matrix of size m x n.
- **B** (MutableDenseMatrix) – A sympy column matrix of size k x n.
- **p** (MutableDenseMatrix) – A sympy column matrix of size m x 1. This column matrix represents the orders of the target group of A. If None, p will be set to the zero vector, i.e. infinite order.

Returns **x** – A solution to X * mod p * A = B.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> from abelian.linalg.utils import vector_mod_vector
>>> A = Matrix([[5, 0, 3],
...             [0, 3, 4]])
>>> X = Matrix([[1, 1],
...             [0, 1]])
>>> B = X * A
>>> X_sol = solve_epi(A, B)
>>> X_sol * A == B
True
```

abelian.linalg.utils module

This module contains a set of utility functions which are used by the other modules in the *linalg* package. The functions defined herein operate on matrices, or are at the very least related to linear algebra computations.

columns_as_list(*A*)

Returns the columns of A as a list of lists.

Parameters **A** (MutableDenseMatrix) – A sympy matrix.

Returns **list_of_cols** – A list of lists, where each sub_list is a column, e.g. structure [[col1], [col2], ...].

Return type list

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],
...             [3, 4]])
>>> list_of_cols = columns_as_list(A)
>>> list_of_cols
[[1, 3], [2, 4]]
```

diag_times_mat(*diagonal*, *A*)

Multiply a diagonal and a dense matrix.

Multiplies a column vector *diagonal* with *A*, in that order. This algorithm exploids the diagonal structure to reduce the number of computations.

Parameters

- **diag** (MutableDenseMatrix) – The diagonal of a matrix, represented as a sympy column vector.
- **A** (MutableDenseMatrix) – A dense matrix.

Returns **product** – The product *diag* times *A*.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix, diag
>>> A = Matrix([[1, 2],
...             [3, 4]])
>>> diagonal = Matrix([2, 3])
>>> diag_times_mat(diagonal, A) == diag(2, 3) * A
True
```

diagonal_rank(*S*)

Count the number of non-zero diagonals in S, where S is in Smith normal form.

Parameters **S** (MutableDenseMatrix) – A sympy matrix in Smith normal form.

Returns **num_nonzeros** – The number of non-zeros on the diagonal.

Return type int

Examples

```
>>> from sympy import diag
>>> diagonal_rank(diag(1,2,0,0,0,0))
2
>>> diagonal_rank(diag(1,2,4,8,0,0))
4
```

difference(*iterable1*, *iterable2*, *p=None*)

Compute the difference with a p-norm.

Parameters

- **iterable1** (MutableDenseMatrix or list) – The iterable to compute the norm over.
- **iterable2** (MutableDenseMatrix or list) – The iterable to compute the norm over.
- **p** (*float*) – The p-value in the p-norm. Should be between 1 and infinity (None).

Returns **norm** – The computed norm of the difference.

Return type float

Examples

```
>>> 2 + 2
4
```

mat_times_diag(*A*, *diagonal*)

Multiply a dense matrix and a diagonal.

Multiplies *A* with a column vector *diagonal*, which is interpreted as the diagonal of a matrix. This algorithm exploids the diagonal structure to reduce the number of computations.

Parameters

- **A** (MutableDenseMatrix) – A dense matrix.
- **diag** (MutableDenseMatrix) – The diagonal of a matrix, represented as a sympy column vector.

Returns **product** – The product *A* times *diag*.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix, diag
>>> A = Matrix([[1, 2],
...             [3, 4]])
>>> diagonal = Matrix([2, 3])
>>> mat_times_diag(A, diagonal) == A * diag(2, 3)
True
```

matrix_mod_vector(*A*, *mod_col*)

Returns a copy of *A* with every column modded by *mod_col*.

Parameters

- **vector** (MutableDenseMatrix) – A sympy matrix of size m x n.
- **mod_col** (MutableDenseMatrix) – A sympy column vector, i.e. a sympy matrix of dimension m x 1.

Returns **A** – A copy of the input with each column modded.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[5, 6],
...             [8, 5],
...             [3, 5]])
>>> mod_col = Matrix([4, 6, 3])
>>> A_modded = matrix_mod_vector(A, mod_col)
>>> A_modded == Matrix([[1, 2],
...                     [2, 5], [0, 2]])
True
```

nonzero_columns(*H*)

Counts the number of columns in H not identically zero.

Parameters **H** (MutableDenseMatrix) – A sympy matrix.

Returns **nonzero_cols** – The number of columns of A not indentically zero.

Return type int

Examples

```
>>> from sympy import Matrix, diag
>>> A = Matrix([[0, 2],
...             [0, 4]])
>>> nonzero_columns(A)
1
>>> nonzero_columns(Matrix.eye(5))
5
>>> nonzero_columns(diag(0,1,0,3,5,0))
3
```

nonzero_diag_as_list(*S*)

Return a list of the non-zero diagonals entries of S.

Parameters **S** (MutableDenseMatrix) – A sympy matrix, typically in Smith normal form.

Returns **nonzero_diags** – A list of the non-zero diagonal entries of S.

Return type list

Examples

```
>>> from sympy import diag
>>> nonzero_diag_as_list(diag(1,2,0,0,0,0))
[1, 2]
>>> nonzero_diag_as_list(diag(1,2,4,8,0,0))
[1, 2, 4, 8]
```

norm(*vector*, *p=2*)

The p-norm of an iterable.

Parameters

- **vector** (MutableDenseMatrix or list) – The iterable to compute the norm over.
- **p** (*float*) – The p-value in the p-norm. Should be between 1 and infinity (None).

Returns **norm** – The computed norm.

Return type float

Examples

```
>>> vector = [1, 2, 3]
>>> norm(vector, 1)
6.0
>>> norm(tuple(vector), None)
3.0
>>> norm(iter(vector), None)
3.0
>>> norm(vector, None)
3.0
>>> norm(vector, 2)
3.7416573867739413
>>> from sympy import Matrix
>>> vector = Matrix(vector)
>>> norm(vector, 1)
6.0
>>> norm(vector, None)
3.0
>>> norm(vector, 2)
3.7416573867739413
```

order_of_vector(*v*, *mod_vector*)

Returns the order of the element *v* in a FGA like *mod_vector*.

Parameters

- **v** (MutableDenseMatrix or a list) – An iterable object with integers. This is the group element.
- **mod_vector** (MutableDenseMatrix or a list) – An iterable object with integers. This is the orders of the group.

Returns **order** – The order of *v* in *mod_vector*.

Return type int

Examples

```
>>> from sympy import Matrix
>>> order_of_vector([1,2,3], [2,4,6]) # Order of 2
2
>>> order_of_vector([1,2,3], [0, 0, 0]) # Order of 0 (infinite order)
0
>>> order_of_vector([1,2,3], [7, 5, 2]) # lcm(7, 10, 2) is 70
70
>>> order_of_vector([0,0,0], [0,0,0]) # Identity element
1
>>> order_of_vector([0,2, 3], [0,0,0]) # Non-trivial element
0
>>> order_of_vector([1, 0, 1], [5, 0, 0])
0
```

reciprocal_entrywise(*A*)

Returns the entrywise reciprocal of a matrix or vector.

Will skip zero entries.

Parameters **A** (MutableDenseMatrix) – A sympy matrix, or vector (m x 1 matrix).

Returns **reciprocal** – The entrywise reciprocal of *A*.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix, diag
>>> D = diag(1, 2, 3)
>>> D_inv = reciprocal_entrywise(D)
>>> D * D_inv == Matrix.eye(3)
True
>>> A = Matrix([[1, 5], [4, 1]])
>>> A_recip = reciprocal_entrywise(A)
>>> A_recip == Matrix([[1, 1/5], [1/4, 1]])
True
```

remove_cols(*A*, *cols_to_remove*)

Return a copy of A where the columns with indices in *cols_to_remove* are removed.

Parameters

- **A** (MutableDenseMatrix) – A sympy matrix.
- **cols_to_remove** (*list*) – A list of column indices to remove from *A*.

Returns **A** – A copy of the input matrix with removed columns.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[5, 6, 7, 8]])
>>> B = remove_cols(A, [0, 2])
>>> B == Matrix([[6, 8]])
True
```

remove_rows(*A*, *rows_to_remove*)

Return a copy of A where the rows with indices in *rows_to_remove* are removed.

Parameters

- **A** (MutableDenseMatrix) – A sympy matrix.
- **rows_to_remove** (*list*) – A list of row indices to remove from *A*.

Returns **A** – A copy of the input matrix with removed rows.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[5, 6, 7, 8]]).T
>>> B = remove_rows(A, [0, 2])
>>> B == Matrix([[6, 8]]).T
True
```

remove_zero_columns(*M*)

Return a copy of M where the columns that are identically zero are deleted.

Parameters **M** (MutableDenseMatrix) – A sympy matrix with zero or more columns which are identically zero.

Returns **M** – A copy of the input matrix with all zero columns removed.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix, diag
>>> A = Matrix([[0, 1],
...             [0, 2]])
>>> remove_zero_columns(A) == Matrix([1, 2])
True
>>> A = diag(0,1,2)
>>> A_del = Matrix([[0, 0],
...                 [1, 0],
...                 [0, 2]])
>>> remove_zero_columns(A) == A_del
True
```

vector_mod_vector(*vector*, *mod_vector*)

Return *vector % mod_vector*, a vectorized mod operation.

Parameters

- **vector** (MutableDenseMatrix) – A sympy column vector, i.e. a sympy matrix of dimension m x 1.
- **mod_vector** (MutableDenseMatrix) – A sympy column vector, i.e. a sympy matrix of dimension m x 1.

Returns **modded_vector** – The result of the mod operation on every entry.

Return type MutableDenseMatrix

Examples

```
>>> from sympy import Matrix
>>> element = Matrix([5, 7, 9])
>>> mod_vect = Matrix([3, 3, 5])
>>> modded = vector_mod_vector(element, mod_vect)
>>> modded == Matrix([2, 1, 4])
True
```

Module contents

Submodules

abelian.functions module

This module consists of a class for functions on LCAs, called LCAFunc. Such a function represents a function from a LCA G to the complex numbers C.

class LCAFunc(*representation*, *domain*)

Bases: collections.abc.Callable

A function from an LCA to a complex number.

__init__(*representation*, *domain*)

Initialize a function G -> C.

Parameters

- **representation** (*function or n-dimensional list of domain allows it*) – A function which takes in a list as a first argument, representing the group element. Alternatively a list of lists if the domain is discrete and of finite order.
- **domain** (LCA) – An elementary locally compact abelian group, which is the domain of the function.

Examples

If a function representation is used, functions on domains are relatively straightforward.

```
>>> def power(list_arg, exponent = 2):
...     return sum(x**exponent for x in list_arg)
>>> from abelian import LCAFunc, LCA
>>> # A function on R/Z = T
>>> f = LCAFunc(power, LCA([1], [False]))
>>> f([0.5])
0.25
>>> f([1.5], exponent = 3) == 0.5**3
True
>>> # A function on Z_p
>>> f = LCAFunc(power, LCA([5, 10]))
>>> f([1,1]) == f([6, 11])
True
>>> f([2, 2], exponent = 1)
4
```

If a table representation is used, the function can be defined on direct sums of Z_n.

```
>>> # Define a table: a list of lists
>>> table = [[1, 2],
...          [3, 4],
...          [5, 6]]
>>> f = LCAFunc(table, LCA([3, 2]))
>>> f([1, 1])
4
>>> f([3, 1])
2
>>> import numpy as np
>>> f = LCAFunc(np.array(table), LCA([3, 2]))
>>> f([1, 1])
4
```

copy()

Return a copy of the instance.

Returns **function** – A copy of *self*.

Return type *LCAFunc*

Examples

```
>>> from abelian import LCA, LCAFunc
>>> f = LCAFunc(lambda x:sum(x), LCA([0]))
>>> g = f.copy()
>>> f([1]) == g([1])
True
```

dft(*func_type=None*)

If the domain allows it, compute DFT.

This method uses the n-dimensional Fast Fourier Transform (FFT) to compute the n-dimensional Discrete Fourier Transform. The data is converted to a `ndarray` object for efficient numerical computation, then the `fftn()` function is used to compute the fast fourier transform.

This implementation is different from the implementation in `fftn()` by a factor. While the `fftn()` function divides by m*n on the inverse transform, this implementation does it on the forward transform, and vice verca.

Parameters **func_type** (*str*) – If None, compute the function values using pure python. If 'ogrid', use a numpy.ogrid (open mesh-grid) to compute the functino values. If 'mgrid', use a numpy.mgrid (dense mesh-grid) to compute the function values.

Returns **function** – The discrete Fourier transformation of the original function.

Return type *LCAFunc*

Examples

```
>>> from abelian import LCA, LCAFunc
>>> # Create a simple linear function on Z_5 + Z_4 + Z_3
>>> domain = LCA([5, 4, 3])
>>> def linear(list_arg):
...     return sum(list_arg)
>>> func = LCAFunc(linear, domain)
>>> func([1, 2, 1])
4
>>> # Take the discrete fourier transform and evaluate
>>> func_dft = func.dft()
>>> func_dft([0, 0, 0]) / (5*4*3) # Divide by number of points
(4.5+0j)
>>> # Take the inverse discrete fourier transform
>>> func_dft_idft = func_dft.idft()
>>> # Numerics might not make this equal, but mathematically it is
>>> abs(func_dft_idft([1, 2, 1]) - func([1, 2, 1])) < 10e-10
True
```

evaluate(*list_arg*, **args*, ***kwargs*)

Evaluate function on a group element.

Parameters

- **list_arg** (*list*) – The first argument, which must be a list (interpreted as vector).
- ***args** (*tuple*) – An unpacked tuple of arguments.
- ****kwargs** (*dict*) – An unpacked dictionary of arguments.

Returns **value** – A complex number (could be real or integer).

Return type complex, float or int

Examples

```
>>> from abelian import LCA, LCAFunc
>>> R = LCA([0], [False])
>>> function = LCAFunc(lambda x: 1, domain = R**2)
>>> function([1, 2])
1
```

Some subtle concepts are shown below.

```
>>> function(1)
Traceback (most recent call last):
...
ValueError: Argument to function must be list.
>>> function([1])
Traceback (most recent call last):
...
ValueError: LCAFunc argument does not match domain length.
>>> type(function([1, 1])) in (int, float, complex)
True
```

idft (*func_type=None*)

If the domain allows it, compute inv DFT.

This is a wrapper around np.fft.ifftn.

Parameters **func_type** (*str*) – If None, compute the function values using pure python. If 'ogrid', use a numpy.ogrid (open mesh-grid) to compute the functino values. If 'mgrid', use a numpy.mgrid (dense mesh-grid) to compute the function values.

Returns **function** – The inverse discrete Fourier transformation of the original function.

Return type *LCAFunc*

Examples

```
>>> from abelian import LCA, LCAFunc
>>> # Create a simple linear function on Z_5 + Z_4 + Z_3
>>> domain = LCA([5, 4, 3])
>>> def linear(list_arg):
...     x, y, z = list_arg
...     return complex(x + y, z - x)
>>> func = LCAFunc(linear, domain)
>>> func([1, 2, 1])
(3+0j)
>>> func_idft = func.idft()
>>> func_idft([0, 0, 0]) * (5*4*3)
(210-60j)
```

pointwise (*other*, *operator*)

Apply pointwise binary operator.

Parameters

- **other** (LCAFunc) – Another Functin on the same domain.
- **operator** (*function*) – A binary operator.

Returns **function** – The resulting function, new = operator(self, other).

Return type *LCAFunc*

Examples

```
>>> from abelian import LCA
>>> domain = LCA([5])
>>> function1 = LCAFunc(lambda arg: sum(arg), domain)
>>> function2 = LCAFunc(lambda arg: sum(arg)*2, domain)
>>> from operator import add
>>> pointwise_add = function1.pointwise(function2, add)
>>> function1([2]) + function2([2]) == pointwise_add([2])
True
>>> from operator import mul
>>> sample_points = [0, 1, 2, 3, 4]
>>> pointwise_mul = function1.pointwise(function2, mul)
>>> pointwise_mul.sample(sample_points) # i * 2*i = 2*i*i
[0, 2, 8, 18, 32]
```

pullback (*morphism*)

Return the pullback along *morphism*.

The pullback is the composition *morphism*, then *self*. The domain of *self* must match the target of the morphism.

Parameters **morphism** (HomLCA) – A homomorphism between LCAs

Returns **pullback** – The pullback of *self* along *morphism*.

Return type *LCAFunc*

Examples

Using a simple function and homomorphism.

```
>>> from abelian import HomLCA, LCA
>>> # Create a function on Z
>>> f = LCAFunc(lambda list_arg:list_arg[0]**2, LCA([0]))
>>> # Create a homomorphism from Z to Z
>>> phi = HomLCA([2])
>>> # Pull f back along phi
>>> f_pullback = f.pullback(phi)
>>> f_pullback([4]) == 64 # (2*4)**2 == 64
True
```

Using a simple function and homomorphism represented as matrix.

```
>>> from abelian import HomLCA, LCA
>>> def func(list_arg):
...     x, y = tuple(list_arg)
...     return x ** 2 + y ** 2
>>> domain = LCA([5, 3])
>>> f = LCAFunc(func, domain)
>>> phi = HomLCA([1, 1], target=domain)
>>> f_pullback = f.pullback(phi)
>>>
>>> f_pullback([8]) == 13
True
```

pushforward(*morphism*, *terms_in_sum=50*)

Return the pushforward along *morphism*.

The pushforward is computed by solving an equation, finding the kernel, and iterating through the kernel. The pushfoward approximates a possibly infinite sum by *terms_in_sum* terms.

Parameters

- **morphism** (HomLCA) – A homomorphism between LCAs.
- **terms_in_sum** (*int*) – The number of terms in the sum to use, i.e. the number of solutions to the equation to iterate over.
- **norm_condition** (*function*) – If not None, a function can be used to terminate the sum. The norm_condition must be a function of a group element, and when the function is false for every v in the kernel such that maxnorm(v) = C for a given C, then the sum terminates.

Returns **pushforward** – The pushforward of *self* along *morphism*.

Return type *LCAFunc*

Examples

The first example is a homomorphism R -> T.

```
>>> from abelian import LCA, LCAFunc, HomLCA
>>> R = LCA([0], [False])
>>> T = LCA([1], [False])
>>> epimorphism = HomLCA([1], source = R, target = T)
>>> func_expr = lambda x: 2**-sum(x_j**2 for x_j in x)
>>> func = LCAFunc(func_expr, domain = R)
>>> func.pushforward(epimorphism, 1)([0]) # 1 term in the sum
1.0
>>> func.pushforward(epimorphism, 3)([0]) # 1 + 0.5*2
2.0
>>> func.pushforward(epimorphism, 5)([0]) # 1 + 0.5*2 + 0.0625*2
2.125
```

The first example is a homomorphism Z -> Z_2.

```
>>> from abelian import LCA, LCAFunc, HomLCA
>>> Z = LCA([0], [True])
>>> Z_2 = LCA([2], [True])
>>> epimorphism = HomLCA([1], source = Z, target = Z_2)
>>> func_expr = lambda x: 2**-sum(x_j**2 for x_j in x)
>>> func = LCAFunc(func_expr, domain = Z)
>>> func.pushforward(epimorphism, 1)([0]) # 1 term in the sum
1.0
>>> func.pushforward(epimorphism, 3)([0]) # 1 + 0.5*2 + 0.0625*2
1.125
```

The third example is a homomorphism R -> R.

```
>>> from abelian import LCA, LCAFunc, HomLCA
>>> R = LCA([0], [False])
```

```
>>> epimorphism = HomLCA([1], source = R, target = R)
>>> func_expr = lambda x: 2**-sum(x_j**2 for x_j in x)
>>> func = LCAFunc(func_expr, domain = R)
>>> func.pushforward(epimorphism, 3)([0]) # 1 term in the sum
1.0
```

sample(*list_of_elements*, **args*, ***kwargs*)

Sample on a list of group elements.

Parameters **list_of_elements** (*list*) – A list of groups elements, where each element is also a list.

Returns **sampled_vals** – A list of sampled values at the elements.

Return type list

Examples

```
>>> from abelian import LCAFunc, LCA
>>> func = LCAFunc(lambda x : sum(x), LCA([0, 0]))
>>> sample_points = [[0, 0], [1, 2], [2, 1], [3, 3]]
>>> func.sample(sample_points)
[0, 3, 3, 6]
```

shift(*list_shift*)

Shift the function.

Parameters **list_shift** (*list*) – A list of shifts.

Returns **function** – A new function which is shifted.

Return type *LCAFunc*

Examples

```
>>> from abelian import LCAFunc, LCA
>>> func = LCAFunc(lambda x: sum(x), LCA([0]))
>>> func.sample([0, 1, 2, 3])
[0, 1, 2, 3]
>>> func.shift([2]).sample([0, 1, 2, 3])
[-2, -1, 0, 1]
```

to_latex()

Return as a LaTeX string.

Returns **latex_str** – The object as a latex string.

Return type str

to_table(**args*, ***kwargs*)

Return a n-dimensional table.

Returns **table** – The table representation.

Return type n-dimensional list

Examples

```
>>> from abelian import LCA, LCAFunc
>>> domain = LCA([5, 5])
>>> f = LCAFunc(lambda x: sum(x), domain)
>>> table = f.to_table()
>>> table[1][1]
(2+0j)
```

Using a table from the start.

```
>>> from abelian import LCA, LCAFunc
>>> import numpy as np
>>> domain = LCA([5, 5])
>>> f = LCAFunc(np.eye(5), domain)
>>> table = f.to_table()
>>> table[1][1]
1.0
>>> type(table)
<class 'numpy.ndarray'>
>>> f = LCAFunc([[1, 2], [2, 4]], LCA([2, 2]))
>>> f.to_table()
[[1, 2], [2, 4]]
```

transversal(*epimorphism*, *transversal_rule=None*, *default_value=0*)

Pushforward using transversal rule.

If (transversal * epimorphism)(x) = x, then x is pushed forward using the transversal rule. If not, then the default_value value is returned.

Parameters

- **epimorphism** (HomLCA) – An epimorphism.
- **transversal_rule** (*function*) – A function with signature *func(list_arg, *args, **kwargs)*.

Returns **function** – The pushforward of *self* along the transversal of the epimorphism.

Return type *LCAFunc*

Examples

```
>>> from abelian import LCA, LCAFunc, HomLCA
>>> n = 5 # Sice of the domain, Z_n
>>> f_on_Zn = LCAFunc(lambda x: sum(x)**2, LCA([n]))
>>> # To move this function to Z, create an epimorphism and a
>>> # transversal rule
>>> epimorphism = HomLCA([1], source = [0], target = [n])
>>> def transversal_rule(x):
...     if sum(x) < n/2:
...         return [sum(x)]
...     elif sum(x) >= n/2:
...         return [sum(x) - n]
...     else:
```

```
...         return None
>>> # Do the pushforward with the transversal rule
>>> f_on_Z = f_on_Zn.transversal(epimorphism, transversal_rule)
>>> f_on_Z.sample(list(range(-n, n+1)))
[0, 0, 0, 9.0, 16.0, 0.0, 1.0, 4.0, 0, 0, 0]
```

voronoi(*epimorphism*, *norm_p=2*)

Return the Voronoi transversal function.

This higher-order function returns a quotient transversal which maps x to the y which is cloest to the low-frequency fourier mode.

Parameters

- **epimorphism_kernel** (HomLCA) – The kernel of the epimorphism that we want to find a section for.
- **norm** (*function or None*) – A norm function, if None, the max-norm is used.

Returns **sigma** – A function x -> y.

Return type function

Examples

```
>>> # An orthogonal example
>>> from abelian import LCAFunc, HomLCA, LCA
>>> Z_10 = LCA([10])
>>> epimorphism = HomLCA([1], target = Z_10)
```

abelian.groups module

This module consists of a class for elementary locally compact abelian groups, the LCA class.

class LCA(*orders*, *discrete=None*)

Bases: collections.abc.Sequence, collections.abc.Callable

An elementary locally compact abelian group (LCA).

__init__(*orders*, *discrete=None*)

Initialize a new LCA.

This class represents locally compact abelian groups, defined by their orders and whether or not they are discrete. An order of 0 means infinite order. The possible groups are:

- $\mathbb{Z}_n$: order = *n*, discrete = *True*
- $\mathbb{Z}$: order = *0*, discrete = *True*
- T : order = *1*, discrete = *False*
- $\mathbb{R}$: order = *0*, discrete = *False*

Every locally compact abelian group is isomorphic to a direct sum or one or several of the groups above.

Parameters

- **orders** (*list*) – A list of orders, e.g. [6, 8, 11].
- **discrete** (*list*) – A list of booleans such as [True, False, ...] or alternatively a list of letters such as ['d', 'c', ...], where 'd' stands for discrete and 'c' stands for continuous. If None, it defaults to discrete.

Examples

```
>>> # Create G = Z_5 + Z_6 + Z_7 in three ways
>>> G1 = LCA([5, 6, 7])
>>> G2 = LCA([5, 6, 7], [True, True, True])
>>> G3 = LCA([5, 6, 7], ['d']*3)
>>> (G1 == G2 == G3)
True
```

```
>>> # Create G = R + Z
>>> G = LCA(orders = [0, 0], discrete = [False, True])
```

```
>>> G = LCA([], [])
>>> G
[]
```

canonical()

Return the LCA in canonical form using SNF.

The canonical form decomposition will:

1. Put the torsion (discrete with order >= 1) subgroup in a canonical form using invariant factor decomposition from the Smith Normal Form decomposition.
2. Sort the non-torsion subgroup.

Returns **group** – The LCA in canonical form.

Return type *LCA*

Examples

```
>>> G = LCA([4, 3])
>>> G.canonical() == LCA([12])
True
>>> G = LCA([1, 1, 8, 2, 4], ['c', 'd', 'd', 'd', 'd'])
>>> G.canonical() == LCA([1], ['c']) + LCA([2, 4, 8])
True
```

compose_self(*power*)

Repeated direct summation.

Returns **group** – A new group.

Return type *LCA*

Examples

```
>>> R = LCA([0], [False])
>>> (R + R) == R**2
True
>>> R**0 == LCA.trivial()
True
>>> Z = LCA([0])
>>> (Z + R)**2 == Z + R + Z + R
True
```

contained_in(*other*)

Whether the LCA is contained in *other*.

A LCA G is contained in another LCA H iff there exists an injection from the elements of G to H such that every source/target of the mapping is isomorphic. In other words, every group in G must be found in H, and no two groups in G can be identified with the same isomorphic group is H.

Parameters **other** (LCA) – A locally compact abelian group.

Returns **is_subgroup** – Whether or not *self* is contained in other.

Return type bool

Examples

```
>>> # Simple example
>>> G = LCA([2, 2, 3])
>>> H = LCA([2, 2, 3, 3])
>>> G.contained_in(H)
True
>>> # Order does not matter
>>> G = LCA([2, 3, 2])
>>> H = LCA([2, 2, 3, 3])
>>> G.contained_in(H)
True
>>> # Trivial groups are not removed
>>> G = LCA([2, 3, 2, 1])
>>> H = LCA([2, 2, 3, 3])
>>> G in H
False
```

copy()

Return a copy of the LCA.

Returns **group** – A copy of the LCA.

Return type *LCA*

Examples

```
>>> G = LCA([1, 5, 7], [False, True, True])
>>> H = G.copy()
```

```
>>> G == H
True
```

dual()

Return the Pontryagin dual of the LCA.

Returns a group isomorphic to the Pontryagin dual.

Returns **group** – The Pontryagin dual of the LCA.

Return type *LCA*

Examples

```
>>> G = LCA([5, 1], [True, False])
>>> H = LCA([5, 0], [True, True])
>>> G.dual() == H
True
>>> G.dual().dual() == G
True
>>> self_dual = LCA([5])
>>> self_dual.dual() == self_dual
True
```

elements_by_maxnorm(*norm_values=None*)

Yield elements corresponding to max norm value.

If the group is discrete, elements can be generated by maxnorm.

Parameters **norm_values** (*iterable*) – An iterable containing integer norm values.

Yields **group_element** (*list*) – Group elements with max norm specified by the input iterable.

Examples

```
>>> G = LCA([0, 0])
>>> for element in G.elements_by_maxnorm([0, 1]):
...     print(element)
[0, 0]
[1, -1]
[-1, -1]
[1, 0]
[-1, 0]
[1, 1]
[-1, 1]
[0, 1]
[0, -1]
>>> G = LCA([5, 8])
>>> for element in G.elements_by_maxnorm([4, 5]):
...     print(element)
[3, 4]
[4, 4]
```

```
[0, 4]
[1, 4]
[2, 4]
```

equal(*other*)

Whether or not two LCAs are equal.

Two LCAs are equal iff the list of *orders* and the list of *discrete* are both equal.

Parameters **other** (LCA) – The LCA to compare equality with.

Returns **equal** – Whether or not the LCAs are equal.

Return type bool

Examples

```
>>> G = LCA([1, 5, 7], [False, True, True])
>>> H = G.copy()
>>> G == H  # The `==` operator is overloaded
True
>>> G.equal(H)  # Equality using the method
True
```

getitem(*key*)

Return a slice of the LCA.

Parameters **key** (slice) – A slice object, or an integer.

Returns **group** – A slice of the FGA as specified by the slice object.

Return type *LCA*

Examples

```
>>> G = LCA([5, 6, 1])
>>> G[0:2] == LCA([5, 6])
True
>>> G[0] == LCA([5])
True
```

is_FGA()

Whether or not the LCA is a FGA.

A locally compact abelian group (LCA) is a finitely generated abelian group (FGA) iff all the groups in the direct sum are discrete.

Returns **is_FGA** – True if the object is an FGA, False if not.

Return type bool

Examples

```
>>> G = LCA([5, 1], [True, False])
>>> G.is_FGA()
False
>>> G = LCA([1, 7], [False, True])
>>> G.dual().is_FGA()
True
```

isomorphic(*other*)

Whether or not two LCAs are isomorphic.

Two LCAs are isomorphic iff they can be put into the same canonical form.

Parameters **other** (LCA) – The LCA to compare with.

Returns **isomorphic** – Whether or not the LCAs are isomorphic.

Return type bool

Examples

```
>>> G = LCA([3, 4])
>>> H = LCA([12])
>>> G.isomorphic(H)
True
>>> G.equal(H)
False
```

```
>>> LCA([2, 6]).isomorphic(LCA([12]))
False
```

```
>>> G = LCA([0, 0, 1, 3, 4], [False, True, True, True, True])
>>> H = LCA([0, 0, 3, 4], [True, False, True, True])
>>> G.isomorphic(H)
True
```

```
>>> LCA([]).isomorphic(LCA.trivial())
True
```

iterate()

Yields the groups in the direct sum one by one.

Iterate through the groups and yield the individual groups in the direct sum, one by one.

Yields **group** (*LCA*) – A single group in the direct sum.

Examples

```
>>> G = LCA([5, 1], [True, False])
>>> groups = [LCA([5], [True]), LCA([1], [False])]
>>> for i, group in enumerate(G):
...     group == groups[i]
```

```
True
True
```

length()

The number of groups in the direct sum.

Returns **length** – The number of groups in the direct sum.

Return type int

Examples

```
>>> G = LCA([])
>>> G.length()
0
```

```
>>> G = LCA([0, 1, 1, 5])
>>> G.length()
4
```

project_element(*element*)

Project an element onto the group.

Parameters **element** (MutableDenseMatrix or list) – The group element to project to the LCA.

Returns **element** – The group element projected to the LCA.

Return type MutableDenseMatrix or list

Examples

```
>>> from sympy import Matrix
>>> G = LCA([5, 9])
>>> g = [6, 22]
>>> G.project_element(g)
[1, 4]
>>> g = Matrix([13, 13])
>>> G.project_element(g) == Matrix([3, 4])
True
```

rank()

Return the rank of the LCA.

Returns **rank** – An integer greater than or equal to 0.

Return type int

Examples

```
>>> G = LCA([5, 6, 1])
>>> G.rank()
2
```

```
>>> LCA([1]).rank()
0
```

```
>>> G = LCA([5, 6, 1])
>>> H = LCA([1])
>>> G.rank() + H.rank() == (G + H).rank()
True
```

remove_indices(*indices*)

Return a LCA with some groups removed.

Parameters **indices** (*list*) – A list of indices corresponding to LCAs to remove.

Returns **group** – The LCA with some groups removed.

Return type *LCA*

Examples

```
>>> G = LCA([5, 8, 9])
>>> G.remove_indices([0, 2]) == LCA([8])
True
```

remove_trivial()

Remove trivial groups from the object.

Returns **group** – The group with trivial groups removed.

Return type *LCA*

Examples

```
>>> G = LCA([5, 1, 1])
>>> G.remove_trivial() == LCA([5])
True
```

sum(*other*)

Return the direct sum of two LCAs.

Parameters **other** (LCA) – The LCA to take direct sum with.

Returns **group** – The direct sum of self and other.

Return type *LCA*

Examples

```
>>> G = LCA([5])
>>> H = LCA([7])
>>> G + H == LCA([5, 7])  # The `+` operator is overloaded
```

```
True
>>> G + H == G.sum(H)  # Directs sums two ways
True
```

to_latex()
Return the LCA as a LaTeX string.

Returns **latex_str** – A string with LaTeX code for the object.

Return type str

Examples

```
>>> G = LCA([5, 0], [True, False])
>>> G.to_latex()
'\\mathbb{Z}_{5} \\oplus \\mathbb{R}'
```

classmethod trivial()
Return a trivial LCA.

Returns **group** – A trivial LCA.

Return type *LCA*

Examples

```
>>> trivial = LCA.trivial()
>>> Z = LCA([0])
>>> (Z + trivial).isomorphic(Z)
True
```

abelian.morphisms module

This module consists of classes representing homomorphisms between elementary LCAs, the HomLCA class.

class HomLCA(*A*, *target=None*, *source=None*)
Bases: collections.abc.Callable

A homomorphism between elementary LCAs.

__init__(*A*, *target=None*, *source=None*)
Initialize a homomorphism.

Parameters

- **A** (MutableDenseMatrix or list) – A sympy matrix representing the homomorphism. The user may also use a list of lists in the form [row1, row2, ...] as input.
- **target** (*LCA or list*) – The target of the homomorphism. If None, a discrete target of infinite order is used as the default.

- **source** (*LCA or list*) – The source of the homomorphism. If None, a discrete source of infinite order is used as the default.

Examples

```
>>> # If no source/target is given, a free discrete group is␣
↪assumed
>>> phi = HomLCA([[1,2],
...               [3,4]])
>>> phi.source.is_FGA() and phi.target.is_FGA()
True
```

```
>>> # If no source is given, a free discrete group is assumed
>>> phi = HomLCA([[1,2],
...               [3,4]], target = [5, 5])
>>> phi.source.is_FGA()
True
```

```
>>> # The homomorphism must be valid
>>> from abelian import LCA, HomLCA
>>> T = LCA(orders = [1], discrete = [False])
>>> R = LCA(orders = [0], discrete = [False])
>>> phi = HomLCA([1], target = R, source = T)
Traceback (most recent call last):
    ...
ValueError: 1: [T] -> [R] is not homomorphism
```

add(*other*)

Elementwise addition.

Elementwise addition of the underlying matrix.

Parameters **other** (*HomLCA or numeric*) – A homomorphism to add to the current one, or a number.

Returns **homomorphism** – A new homomorphism with the argument added.

Return type *HomLCA*

annihilator()

Compute the annihilator monomorphism.

coimage()

Compute the coimage epimorphism.

Returns **homomorphism** – The coimage homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([[4, 4],
...               [2, 8]], target = [16, 16])
>>> im = phi.image().remove_trivial_groups()
>>> coim = phi.coimage().remove_trivial_groups()
```

```
>>> phi == (im * coim).project_to_target()
True
```

cokernel()
Compute the cokernel epimorphism.

Returns **homomorphism** – The cokernel homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([[1, 0], [0, 1], [1, 1]])
>>> coker = phi.cokernel()
>>> coker.target.isomorphic(LCA([1, 1, 0]))
True
```

compose(*other*)
Compose two homomorphisms.

The composition of *self* and *other* is first other, then self.

Parameters **other** (HomLCA) – The homomorphism to compose with.

Returns **homomorphism** – The composition of *self* and *other*, i.e. *self* (*other* (x)).

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([[1, 0, 1],
...               [0, 1, 1]])
>>> ker_phi = HomLCA([1, 1, -1])
>>> (phi * ker_phi) == HomLCA([0, 0])
True
>>> phi.compose(ker_phi) == HomLCA([0, 0])
True
```

compose_self(*power*)
Repeated composition of an endomorphism.

Parameters **power** (*int*) – The number of times to compose with self.

Returns **homomorphism** – The endomorphism composed with itself *power* times.

Return type *HomLCA*

Examples

```
>>> from sympy import diag
>>> phi = HomLCA(diag(2, 3))
>>> phi**3 == HomLCA(diag(2**3, 3**3))
True
```

copy()

Return a copy of the homomorphism.

Returns **homomorphism** – A copy of the homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([1, 2, 3])
>>> phi.copy() == phi
True
```

det()

Determinant of the matrix representing the HomLCA.

Returns **determinant** – Determinant of the matrix, if possible.

Return type float

Examples

```
>>> from abelian import LCA, HomLCA
>>> phi = HomLCA([[2, 0], [0, 3]])
>>> phi.det()
6
>>> R = LCA([0], [False])
>>> phi = HomLCA([[2.5, 0], [0, 2.5]], R**2, R**2)
>>> phi.det() == 6.25
True
>>> HomLCA([[3, 1]]).det()
Traceback (most recent call last):
...
sympy.matrices.common.NonSquareMatrixError
```

dual()

Compute the dual homomorphism.

TODO: Write detailed description.

Returns **dual** – The dual homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([2])
>>> phi_dual = phi.dual()
>>> phi_dual.source == phi_dual.target
True
```

Computing duals by first calculating orders

```
>>> # Project, then find dual
>>> phi = HomLCA([2], target = [10])
>>> phi_proj = phi.project_to_source()
>>> phi_project_dual = phi_proj.dual()
>>> phi_project_dual == HomLCA([1], [5], [10])
True
>>> # Do not project
>>> phi_dual = phi.dual()
>>> phi_dual == HomLCA([1/5], LCA([1], [False]), [10])
True
```

equal (*other*)

Whether or not two homomorphisms are equal.

Two HomLCAs are equal iff (1) the sources are equal, (2) the targets are equal and (3) the matrices representing the homomorphisms are equal.

Parameters **other** (HomLCA) – A HomLCA to compare equality with.

Returns **equal** – Whether or not the HomLCAs are equal.

Return type bool

Examples

```
>>> phi = HomLCA([1], target=[0], source = [0]) # Explicit
>>> psi = HomLCA([1])    # Shorter, defaults to the above
>>> phi == psi
True
```

evaluate (*source_element*)

Apply the homomorphism to an element.

Parameters **source_element** –

Examples

```
>>> from sympy import diag
>>> phi = HomLCA(diag(3, 4), target = [5, 6])
>>> phi.evaluate([2, 3])
[1, 0]
>>> phi.evaluate(Matrix([2, 3]))
Matrix([
[1],
[0]])
```

getitem(*args*)

Return a slice of the homomorphism.

Slices the object with the common matrix slice notation, e.g. A[rows, cols], where the *rows* and *cols* objects can be either integers or slice objects. If the homomorphism is represented by a column or row matrix, then the notation A[key] will also work. The underlying matrix and the source and target LCAs are all sliced.

Parameters **args** (*slice*) – A slice or a tuple with (slice_row, slice_col).

Returns **homomorphism** – A sliced homomorphism.

Return type *HomLCA*

Examples

The homomorphism is sliced using two input arguments.

```
>>> from sympy import diag
>>> phi = HomLCA(diag(4,5,6))
>>> phi[0,:] == HomLCA([[4, 0, 0]])
True
>>> phi[:,1] == HomLCA([0, 5, 0])
True
```

If the homomorphism is represented by a row or column, one arg will do.

```
>>> phi = HomLCA([1,2,3])
>>> phi[0:2] == HomLCA([1,2])
True
```

classmethod identity(*group*)

Return the identity morphism.

Examples

```
>>> from abelian import LCA, HomLCA
>>> H = LCA([5, 6, 7])
>>> G = LCA([0, 0])
>>> phi = HomLCA([[1,2], [3,4], [5,6]], source = G, target = H)
>>> Id_H = HomLCA.identity(H)
>>> Id_G = HomLCA.identity(G)
>>> # Verify the properties of the identity morphism
>>> Id_H * phi == phi
True
>>> phi * Id_G == phi
True
```

image()

Compute the image monomorphism.

Returns **homomorphism** – The image homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([[4, 4],
...               [2, 8]], target = [64, 32])
>>> im = phi.image().remove_trivial_groups()
>>> coim = phi.coimage().remove_trivial_groups()
>>> phi == (im * coim).project_to_target()
True
```

```
>>> # Image computations are also allowed when target is R
>>> R = LCA(orders = [0], discrete = [False])
>>> sample_matrix = [[1, 2, 3], [2, 3, 5]]
>>> phi_sample = HomLCA(sample_matrix, target = R + R)
>>> phi_sample_im = phi_sample.image().remove_trivial_groups()
>>> phi_sample_im == phi_sample[:, 1:]
True
```

kernel()

Compute the kernel monomorphism.

Returns **homomorphism** – The kernel homomorphism.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([[1, 0, 1], [0, 1, 1]])
>>> phi.kernel() == HomLCA([-1, -1, 1])
True
```

project_to_source()

Project columns to source group (orders).

Returns **homomorphism** – A homomorphism with orders in the source FGA.

Return type *HomLCA*

Examples

```
>>> target = [3, 6]
>>> phi = HomLCA([[1, 0],
...               [3, 3]], target = target)
>>> phi = phi.project_to_source()
>>> phi.source.orders == [6, 2]
True
```

project_to_target()

Project columns to target group.

Returns **homomorphism** – A homomorphism with columns projected to the target FGA.

Return type *HomLCA*

Examples

```
>>> target = [7, 12]
>>> phi = HomLCA([[15, 12],
...               [9,  17]], target = target)
>>> phi_proj = HomLCA([[1, 5],
...                    [9, 5]], target = target)
>>> phi.project_to_target() == phi_proj
True
```

remove_trivial_groups()

Remove trivial groups.

A group is trivial if it is discrete with order 1, i.e. Z_1. Removing trivial groups from the target group means removing the Z_1 groups from the target, along with the corresponding rows of the matrix representing the homomorphism. Removing trivial groups from the source group means removing the groups Z_1 from the source, i.e. removing every column (generator) with order 1.

Returns **homomorphism** – A homomorphism where the trivial groups have been removed from the source and the target. The corresponding rows and columns of the matrix representing the homomorphism are also removed.

Return type *HomLCA*

Examples

```
>>> target = [1, 7]
>>> phi = HomLCA([[2, 1], [7, 2]], target=target)
>>> projected = HomLCA([[2]], target=[7], source = [7])
>>> phi.project_to_source().remove_trivial_groups() == projected
True
```

shape

The shape (*rows*, *cols*).

Returns **shape** – A tuple with the shape of the underlying matrix A, i.e. (*rows*, *cols*).

Return type tuple

stack_diag(*other*)

Stack diagonally.

Parameters **other** (HomLCA) – A homomorphism to stack with the current one.

Returns **stacked_vert** – The result of stacking the homomorphisms on diagonally.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([1])
>>> psi = HomLCA([2])
```

```
>>> phi.stack_diag(psi) == HomLCA([[1, 0], [0, 2]])
True
```

stack_horiz(*other*)

Stack horizontally (column wise).

The targets must be the same, the sources will be concatenated. The stacking is done to create a matrix with structure [self, other], i.e. "Putting *self* to the left of *other*."

Parameters **other** (HomLCA) – A homomorphism to stack with the current one.

Returns **stacked_vert** – The result of stacking the homomorphisms side by side.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([1])
>>> psi = HomLCA([2])
>>> phi.stack_horiz(psi) == HomLCA([[1, 2]])
True
```

stack_vert(*other*)

Stack vertically (row wise).

The sources must be the same, the targets will be concatenated. The stacking is done to create a matrix with structure [[self], [other]], i.e. "Putting *self* on top of of *other*."

Parameters **other** (HomLCA) – A homomorphism to stack with the current one.

Returns **stacked_vert** – The result of stacking the homomorphisms on top of each other.

Return type *HomLCA*

Examples

```
>>> phi = HomLCA([1])
>>> psi = HomLCA([2])
>>> phi.stack_vert(psi) == HomLCA([1, 2])
True
```

to_latex()

Return the homomorphism as a LaTeX string.

Returns **latex** – The HomLCA formatted as a LaTeX string.

Return type str

Examples

```
>>> phi = HomLCA([1])
>>> phi.to_latex()
'\\begin{pmatrix}1\\end{pmatrix}:\\mathbb{Z} \\to \\mathbb{Z}'
```

update (*new_A=None*, *new_target=None*, *new_source=None*)
Return a new homomorphism with updated properties.

classmethod zero (*target*, *source*)
Initialize the zero morphism.

Parameters

- **target** (*LCA or list*) – The target of the homomorphism. If None, a discrete target of infinite order is used as the default.
- **source** (*LCA or list*) – The source of the homomorphism. If None, a discrete source of infinite order is used as the default.

Examples

```
>>> zero = HomLCA.zero([0]*3, [0]*3)
>>> zero([1, 5, 7]) == [0, 0, 0]
True
```

abelian.utils module

arg (*min_or_max*, *iterable*, *function_of_element*)
Call a nested list like a function.

Parameters

- **list_of_lists** (*list*) – A nested list of lists.
- **arg** (*list*) – The argument [dim1, dim2, . . .].

Returns **value** – The object in the list of lists.

Return type object

Examples

```
>>> iterable = [-8, -4, -2, 3, 5]
>>> arg(min, iterable, abs)
-2
>>> iterable = range(-10, 10)
>>> arg(max, iterable, lambda x: -(x - 3)**2)
3
```

call_nested_list (*list_of_lists*, *arg*)
Call a nested list like a function.

Parameters

- **list_of_lists** (*list*) – A nested list of lists.
- **arg** (*list*) – The argument [dim1, dim2, . . .].

Returns **value** – The object in the list of lists.

Return type object

Examples

```
>>> table = [1 ,2 ,3]
>>> call_nested_list(table, [0])
1
```

```
>>> table = [[1, 2], [1, 2], [1, 2]]
>>> call_nested_list(table, [0, 0])
1
```

```
>>> table = [[[1, 2, 3, 4], [1, 2, 3, 4]],
...          [[1, 2, 3, 4], [1, 2, 3, 4]],
...          [[1, 2, 3, 4], [1, 2, 3, 4]]]
>>> call_nested_list(table, [0, 0])
[1, 2, 3, 4]
>>> call_nested_list(table, [0, 0, 0])
1
```

copy_func(*f*)

Based on http://stackoverflow.com/a/6528148/190597 (Glenn Maynard)

function_to_table(*function*, *dims*, **args*, ***kwargs*)

Parameters

- **function** (*function*) – A function with signature *(list_arg, *args, **kwargs)*.
- **dims** (*list*) – A list of dimensions such as [8, 6, 3].

mod(*a*, *b*)

Returns a % b, with a % 0 = a.

Parameters

- **a** (*int*) – The first argument in a % b.
- **b** (*int*) – The second argument in a % b.

Returns *a* modulus *b*.

Return type int

Examples

```
>>> mod(5, 2)
1
>>> mod(5, 0)
5
```

verify_dims_list(*list_of_lists*, *dims*)

Verify the dimensions of a list of lists.

Parameters

- **list_of_lists** (*list*) – A nested list of lists.
- **dims** (*list*) – A list of dimensions.

Returns **verified** – Whether or not the dimensions match the *dims* parameter.

Return type bool

Examples

```
>>> table = [1 ,2 ,3]
>>> dims = [3]
>>> verify_dims_list(table, dims)
True
```

```
>>> table = [[1, 2], [1, 2], [1, 2]]
>>> dims = [3, 2]
>>> verify_dims_list(table, dims)
True
```

```
>>> table = [[[1, 2, 3, 4], [1, 2, 3, 4]],
...          [[1, 2, 3, 4], [1, 2, 4]],
...          [[1, 2, 3, 4], [1, 2, 3, 4]]]
>>> dims = [3, 2, 4] # Not correct, notice the missing value above
>>> verify_dims_list(table, dims)
False
```

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

D

E

F

G

V

Z