Fourier Analysis on abelian groups; theory and applications

by
Tommy Odland

Master of Science Thesis in
Applied and Computational Mathematics

Department of Mathematics
University of Bergen

November 2017

Abstract

Fourier analysis expresses a function as a weighted sum of complex exponentials. The Fourier machinery can be applied when a function is defined on a locally compact abelian group (LCA). The groups $\mathbb{R}$, $T = \mathbb{R}/\mathbb{Z}$, $\mathbb{Z}$ and $\mathbb{Z}_n$ are all LCAs of great interest, but numerical computations are almost always done on the finite group $\mathbb{Z}_n$ using the Fast Fourier Transform.

To reduce a general problem to a numerical computation, sampling and periodization is necessary. In this thesis we present a new software library which facilitates Fourier analysis on elementary LCAs. The software allows the user to work directly with abstract mathematical objects, perform numerical computations and handle the relationship between discrete and continuous domains in a natural way.

The specific combination of mathematical objects and operations available in the software developed is to our knowledge not found elsewhere. Efforts have been made to efficiently open-source, document and distribute the software library, which is now available to every user of the Python programming language.

Acknowledgements

First and foremost I would like to thank my advisor, professor Hans Z. Munthe-Kaas, for suggesting this interesting project and for all of his help following it through. The versatility of this project has made it delightful to work with: there are undoubtedly abstract components to Fourier analysis and group theory, but at the same time it's an applicable and concrete field of mathematics. Building a relatively large software library has been an exciting learning experience.

I would also like to thank Amir M. Hashemi, Erlend R. Vågset, Gunvor Lemvik and Simen Midtbø for reading through a draft of the thesis and providing comments.

Contents

Notation

- **Groups**

  $\mathbb{Z}$: Additive integers

  $\mathbb{Z}_n$: Additive integers mod n

  $\mathbb{R}$: Additive reals

  $T = \mathbb{R}/\mathbb{Z}$: Additive reals mod 1

  $\mathbb{T}$: Complex numbers z with $|z| = 1$

  $GL(n, \mathbb{Z})$: Invertible matrices over $\mathbb{Z}$

- **Categories**

  Set : Category of sets

  VectR : Vector spaces over $\mathbb{R}$

  ModZ : Modules over $\mathbb{Z}$

  FinAb : FGAs

  Ab : Abelian groups

- **Objects**

  G, H : Abelian groups

  U, V : Unimodular matrices

  I_n : Identity matrix of size n

- **Binary operators**

  $(\cdot, \cdot)$: Dual pairing

  $*$: Convolution

  $\oplus$: Direct sum

- **Relations**

  $\cong$: Isomorphic

- **Arrows**

  $H \xrightarrow{\phi} \!\!\!\!\rightarrow G$: epimorphism

  $H \overset{\phi}{\hookrightarrow} G$: monomorphism

- **Other**

  $\mathcal{O}$: Big O notation

  $\phi^{\perp}$: Annihilator of ϕ

  $\widehat{G}$: Dual group of G

  $\mathcal{F}(f), \widehat{f}$: Fourier transform of f

  $\hom(\cdot, \cdot)$: Set of homomorphisms

  B^A : Functions $f : A \to B$

Abbreviations

DFT - Discrete Fourier Transform

FFT - Fast Fourier Transform

FGA - Finitely Generated Abelian group

LCA - Locally Compact Abelian group

HNF - Hermite Normal Form

SNF - Smith Normal Form

Chapter 1

Introduction

1.1 Introduction

Fourier analysis expresses a function as a weighted sum of trigonometric functions, or equivalently complex exponentials. Fourier synthesis recovers the original function from the frequency representation. These ideas are remarkably powerful, and are widely used in applied and theoretical science.

The general setting for Fourier analysis are the locally compact abelian groups (LCAs). The four common Fourier transforms are defined for functions on $\mathbb{R}$, $T = \mathbb{R}/\mathbb{Z}$, $\mathbb{Z}$ and $\mathbb{Z}_n$, and these groups are called elementary LCAs. The goal of this master project is to create software which allows for general computations and Fourier analysis on the group G, which is an elementary LCA. To do this, the software must handle periodization, discretization and interpretation of functions $f : G \to \mathbb{C}$. Ideas for such a software package were sketched in [Munthe-Kaas, 2016].

The majority of the thesis is devoted to the theory required to understand the general framework we will work in. We will make use of group theory, linear algebra, category theory and abstract Fourier analysis. To unify the reading experience, the thesis includes theoretical preliminaries and some historical remarks. After introducing the mathematics needed to understand the objects, operations and algorithms in the software, the `abelian` library is introduced in Chapter 7. A general introduction is given along with example code, and further details about the software is found in the appendices.

Some of the underlying algorithms used in the software are known, see for instance [Charles C. Sims, 1994] for algorithms which compute the Hermite and Smith normal forms. Several algorithms are the result of my own work: Algorithm 3 in Section 5.4 for the solution of a particular equation is my own, and so is Algorithm 5 in Section 6.5 used to generate group elements ordered

by norm. The theorems presented in Section 5.5 for the computation of the kernel, cokernel, image and coimage were developed in discussions with my advisor Hans Z. Munthe-Kaas.

The `abelian` software library is the main contribution of this project. It's open sourced, well-documented, has a modern test-suite and is distributed on the official Python package index. As far as I am aware, no similar software package exists. The combination of objects and operations is unique, and implementations of the underlying algorithms are relatively rare. For instance, MATLAB currently implements a Smith normal form algorithm only for square matrices, while a general implementation is found in `abelian`. Among the popular scientific Python libraries, no implementations were found. Now, Python users will have access to this algorithm and many others, as well as objects and methods for Fourier analysis and general computations on elementary LCAs.

1.1.1 Software used in this project

This document was typeset using LaTeX 2_ε. The plots were produced using the `matplotlib` library for Python, and the remaining figures were created using the extensible drawing editor `ipe`.

The `abelian` library was written in Python 3.6. The following web services were used for software distribution.

- Source code: `github.com/tommyod/abelian/`
- Documentation: `abelian.readthedocs.io/`
- Python package index: `pypi.org/project/abelian/`

1.2 Chapter overview

Chapter 1 – Introduction The introduction provides an overview of the thesis. It gives a brief introduction to Fourier analysis, states the goal of the project and details what is new. A brief comparison is made with existing software. This chapter overview is a quick account of the content and purpose of each chapter.

Chapter 2 – Preview To entice the reader, the preview chapter shows some example usage of the `abelian` software library. Three examples are presented in increasing order of complexity. Real Python code snippets are included, along with figures explaining the problems.

Chapter 3 – Preliminaries The purpose of this chapter is to serve as a reference for the later chapters. It has been divided into four parts: integers and set functions, group theory, category theory and Fourier analysis. Each section briefly summarizes some material which the reader ideally has some existing knowledge of, while also establishing the notation. Depending on the background of the reader, this chapter may be skimmed or perhaps skipped altogether.

Chapter 4 – Integer linear algebra To study finitely generated abelian groups (FGAs), we will require knowledge of the the Hermite normal form (HNF) and the Smith normal form (SNF), which are introduced as factorizations of a linear map $A : \mathbb{Z}^n \to \mathbb{Z}^m$. Unimodular matrices are defined, and algorithms for computing the HNF and SNF are presented. While these algorithms are discussed in the literature, implementations are rare.

Chapter 5 – Computing factorizations in FinAb In an abelian category, a morphism has a kernel, cokernel, image and coimage. The goal of this chapter is to develop algorithms for computing these four important morphisms in FinAb, the category of FGAs. Working our way up to this goal, we consider how to compute the morphisms in several categories. We start with the relatively well known case of the category of vector spaces over $\mathbb{R}$, denoted VectR, and work towards the goal of developing algorithms for the category FinAb.

Chapter 6 – Fourier analysis on locally compact abelian groups We examine Fourier analysis from the perspective of LCAs. From this view Fourier series, the Fourier transform and the discrete Fourier transform are the same thing. Dual groups, the dual pairing and dual homomorphisms are introduced. Pullbacks and pushforwards are discussed both in theory and practical computations. An algorithm for practical computations of pushforwards is developed. Finally we discuss computational Fourier analysis on T^d as well as $\mathbb{R}^d$, and study Fourier analysis on a hexagonal lattice in detail. We briefly compare our approach to methods used in several recent research papers on Fourier analysis on lattices.

Chapter. 7 – The `abelian` software library This chapter introduces `abelian`, a software library for computations on elementary LCAs. We start by briefly discussing the state of open-source scientific software and the Python programming language. We mention some good practices for developing a modern software library, and finally introduce `abelian`. The `abelian` software library is then used to compute several concrete examples.

A selection of the most important algorithms from the source code is found in Appendix A. An excerpt of the full software documentation is found in Appendix B.

Chapter 8 – Conclusions and further work The work is concluded and some suggestions for further work are presented.

Appendices There are two appendices: Appendix A contains some of the source code for `abelian`. Only the most important algorithms are included. Appendix B contains parts of the full documentation. The general introduction and tutorials are included, while detailed documentation of the methods with examples are omitted due to space considerations.

Chapter 2

Preview

In this chapter we present concrete examples of the type of problems that the `abelian` software library is able to handle. Real, working Python code is included in each section. The purpose of this chapter is twofold, the goals are to: (1) show what we will be working toward by way of example, and (2) informally introduce the reader to some notation and terminology which we will use. Concepts are loosely defined here, and they will be introduced more rigorously in the following chapters.

2.1 Sampling on a lattice

Sampling is of great importance in signal processing, a field which makes considerable use of Fourier analysis. While equidistant points are almost always used in one dimension, higher dimensions open up to more choices. Orthogonal, equidistant sampling is the prevalent choice in two dimensions (e.g. a digital image), but it is not the only option.

We will now demonstrate how sampling may be done with `abelian`. This example will make Figure 2.1 more concrete. In the software, an `LCA` object is a locally compact abelian group (such as $\mathbb{R}$ or $\mathbb{Z}$), a `HomLCA` object is a homomorphism between two LCAs and an `LCAFunc` is a function from an LCA to $\mathbb{C}$. In the code below we start by defining $\mathbb{R}$, which is a non-discrete LCA of infinite order. Next we initialize a Gaussian function $f : \mathbb{R}^2 \to \mathbb{C}$, which we define by the expression $x \mapsto \exp\left(-x_1^2 - x_2^2\right)$ and the domain $\mathbb{R}^2$.

```
1 from abelian import HomLCA, LCA, LCAFunc
2 R = LCA(orders = [0], discrete = [False])
3 func_expr = lambda x: exp(-sum(x_j**2 for x_j in x))
4 func = LCAFunc(func_expr, domain = R**2)
```

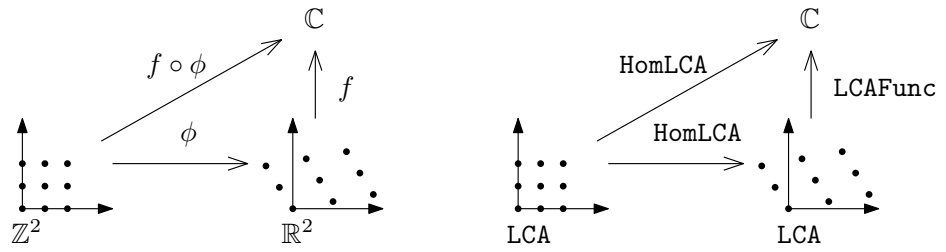


Figure 2.1: To the left: Sampling $f : \mathbb{R}^2 \to \mathbb{C}$ using a homomorphism $\phi : \mathbb{Z}^2 \to \mathbb{R}^2$. The homomorphism ϕ defines a lattice on $\mathbb{R}^2$, and the composition $f \circ \phi : \mathbb{Z}^2 \to \mathbb{C}$ defines a function on the discrete group $\mathbb{Z}^2$. To the right: the same diagram, with the objects defined in `abelian` representing the mathematical objects.

To sample f, we define a group homomorphism $\phi : \mathbb{Z}^2 \to \mathbb{R}^2$ using a sampling matrix $A = \left(\begin{smallmatrix} 1 & 2 & 3 \\ 4 & 3 & 7 \end{smallmatrix}\right)$. To demonstrate a computation of the image homomorphism, we have created a ϕ which is not injective on purpose.[1] To make ϕ injective, we compute the image and remove trivial subgroups. The image is injective by definition. To sample f, we form the composition $f \circ \phi : \mathbb{Z}^2 \to \mathbb{C}$. This composition will be defined as the pullback of f along ϕ in Section 6.5.

```
5  sample_matrix = [[1, 2, 3], [4, 3, 7]]
6  phi_sample = HomLCA(sample_matrix, target = R**2)
7  phi_sample = phi_sample.image().remove_trivial_groups()
8  sampled_func = func * phi_sample
```

The sampled function can be evaluated at a point such as $(1, 3)$. A more dynamic approach is to generate the group elements $(x_1, x_2) \in \mathbb{Z}^2$ by increasing max-norm, and then evaluate the sampled function on a stream of group elements. A never-ending stream of group elements are yielded, so the user of the software will have to write a criterion to terminate the infinite loop.

```
9  value = sampled_func([1, 3])
10 elements = phi_sample.source.elements_by_maxnorm()
11 for element in elements:
12     print(sampled_func(element))
```

With a few changes, the code above could be used to sample a function in $\mathbb{R}^d$, for instance to approximate a multi-dimensional integral.

[1] The homomorphism ϕ is not injective since columns 1 and 2 add to column 3.

2.2 Fourier series approximation

A Fourier series is a representation of a periodic function as a weighted sum of trigonometric functions. Analytical expressions for the weights can be obtained by evaluating an integral. Alternatively, a numerical approximation can be obtained by sampling, computing the discrete Fourier transform (DFT), and interpreting the results. Starting at the bottom right of Figure 2.2, the analytical solution is obtained by moving up. The approximation is obtained by going left, up, and then right.

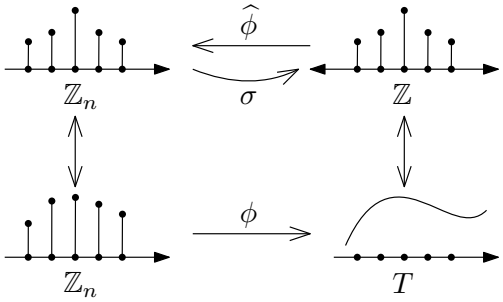


Figure 2.2: Groups, functions and homomorphisms associated with Fourier series approximation. The two-headed arrows denote dual pairs of groups.

We will now demonstrate approximation of Fourier series coefficients using the `abelian` software. We start by creating a periodic, continuous domain T and initialize $f(x) = x$ as a function on this domain.

```
1  from abelian import HomLCA, LCA, LCAFunc
2  T = LCA(orders = [1], discrete = [False])
3  func_expr = lambda x: sum(x)
4  func = LCAFunc(func_expr, domain = T)
```

To sample f, we define a homomorphism $\phi : \mathbb{Z}_n \to T$ by the rule $j \mapsto j/n$. We choose $n = 10$ sample points.

```
5  from sympy import Rational
6  n = 10
7  Z_n = LCA(orders = [n], discrete = [True])
8  phi = HomLCA([Rational(1, n)], target = T, source = Z_n)
```

The function f is moved to the domain $\mathbb{Z}_n$ using the pullback operation, which corresponds to the composition $f \circ \phi : \mathbb{Z}_n \to \mathbb{C}$. Since the domain is discrete and finite, the DFT is available to us. Using the DFT, we move the function to the dual space, which corresponds to the top row in Figure 2.2.

```
9  func_sampled = func.pullback(phi)
10 func_sample_dual = func_sampled.dft()
```

The approximated Fourier series weights (or coefficients) are now defined on $\mathbb{Z}_n$. To interpret the coefficients on $\mathbb{Z}$, we employ the dual homomorphism $\widehat{\phi}$ and a user-specified quotient transversal σ, defined in Chapter 6. For now it suffices to say that σ maps the Fourier coefficients to $\mathbb{Z}$ in such a way that the Fourier series representation consists of low-frequency trigonometric functions. In the language of signal processing, σ performs de-aliasing.

```
11 def sigma(x):
12     if x[0] < n//2:
13         return x
14     return [x[0] - n]
15
16 func_dual = func_sample_dual.transversal(phi.dual(), sigma)
17 points = [[i] for i in range(-n, n+1)]
18 fourier_coeffs = func_dual.sample(points)
```

Sampling the approximation is done in the last two lines in the code snippet above, and the result is shown in Figure 2.3. In Example 3.38 on page 24 we will solve this problem analytically and obtain exact coefficients. The analytical coefficients decay proportionally to $1/|\xi|$ as $|\xi| \to \infty$, and this behavior is clearly visible in Figure 2.3.

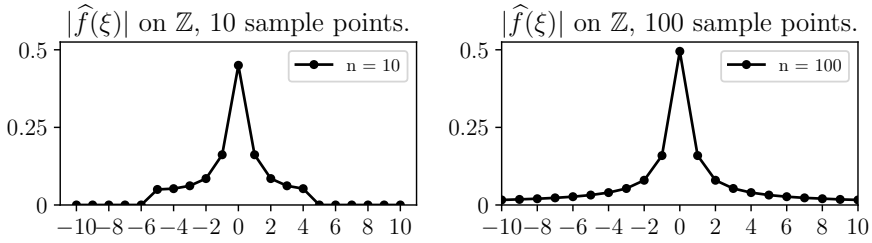


Figure 2.3: The plots show the absolute value of the approximated Fourier series coefficients for $f(x) = x$ defined on T. The number of sample points used was $n = 10$ and $n = 100$ in the leftmost and rightmost plot, respectively.

2.3 Hexagonal sampling and periodization

Hexagonal sampling has several advantages over equidistant, orthogonal sampling: equal distance to all neighboring points, higher degree of symmetry, and fewer sample points are required to reconstruct band-limited functions with an isotropic (rotation-invariant) Fourier transform. We will discuss Fourier analysis on a hexagonal lattice in detail in Section 6.9. In this section we will sample and periodize on a hexagonal lattice using `abelian`, which prepares data for the DFT to operate on.

To perform computational Fourier analysis on hexagonally sampled data, we will first move a function from $\mathbb{R}^2$ to $\mathbb{Z}^2$ by sampling, and then from $\mathbb{Z}^2$ to $\mathbb{Z}_m \oplus \mathbb{Z}_n$ by periodization. These groups, along with homomorphisms for sampling and periodization, are shown in the following diagram.

$$\begin{array}{ccccccc} & & \mathbb{Z}^2 & \overset{SA}{\curvearrowright} & & & \\ & & \downarrow{\scriptstyle A} & & & & \\ \mathbb{Z}_m \oplus \mathbb{Z}_n & \xleftarrow{\text{coker}(A)} & \mathbb{Z}^2 & \xrightarrow{S} & \mathbb{R}^2 & \xrightarrow{\text{coker}(S)} & T^2 \end{array} \tag{2.1}$$

In this example we consider the Gaussian function $f(x) = \exp(-k(x_1^2 + x_2^2))$ on $\mathbb{R}^2$, where $k > 0$ is a constant. In Diagram (2.1), the homomorphism given by the matrix $S = \left(\begin{smallmatrix} 1 & 1/2 \\ 0 & \sqrt{3}/2 \end{smallmatrix}\right)$ samples $f(x)$ defined on $\mathbb{R}^2$. The homomorphism represented by $A = \left(\begin{smallmatrix} m & 0 \\ 0 & n \end{smallmatrix}\right)$ periodizes a function on $\mathbb{Z}^2$ by defining a quotient homomorphism called the cokernel, which we will define and study in Chapter 5. See Figure 2.4 for a plot of the Gaussian and it's sampled values on a hexagonal lattice.

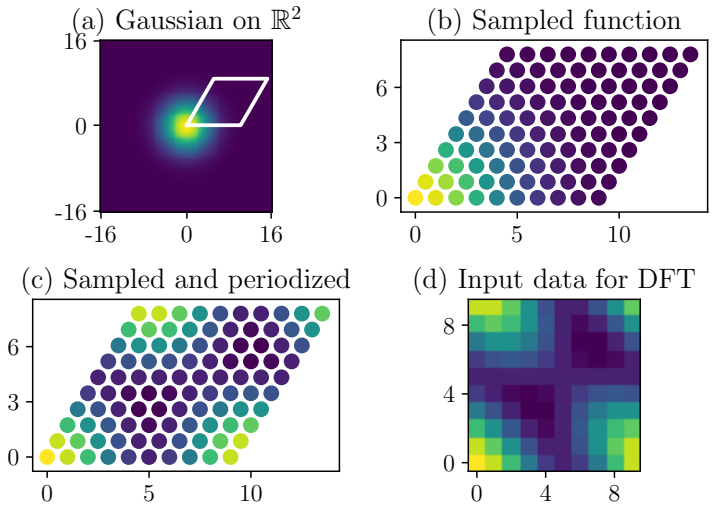


Figure 2.4: Several subplots related to hexagonal sampling and periodization. (a) The Gaussian defined on $\mathbb{R}^2$, the unit cell of the periodization homomorphism is shown in white. (b) Function values hexagonally sampled in the unit cell. (c) Sampled and periodized function. (d) The input for matrix for the DFT, i.e. a function on $\mathbb{Z}_{10} \oplus \mathbb{Z}_{10}$.

We will now demonstrate how to create the homomorphisms in Diagram (2.1) using `abelian`. The code below defines a Gaussian function on $\mathbb{R}^2$.

```
1 from abelian import HomLCA, LCA, LCAFunc
```

```
R = LCA(orders = [0], discrete = [False])
k = 0.05 # Decay of Gaussian function
func_expr = lambda x: exp(-k*sum(x_j**2 for x_j in x))
func = LCAFunc(func_expr, domain = R**2)
```

Next we create $S : \mathbb{Z}^2 \to \mathbb{R}^2$, defined by $\left(\begin{smallmatrix} 1 & 1/2 \\ 0 & \sqrt{3}/2 \end{smallmatrix}\right)$, a matrix with columns generating a hexagonal lattice. Notice that when no homomorphism source is explicitly given, `abelian` implicitly assumes $\mathbb{Z}^2$.

```
hexagonal_generators = [[1, 0.5], [0, sqrt(3)/2]]
S = HomLCA(hexagonal_generators, target = R**2)
```

Choosing $m = n = 10$ sample points, we create a periodization homomorphism by defining $A = \left(\begin{smallmatrix} 10 & 0 \\ 0 & 10 \end{smallmatrix}\right)$ and computing it's cokernel.

```
n = 10
A = HomLCA([[n, 0], [0, n]])
coker_A = A.cokernel()
```

We move the function from $\mathbb{R}^2$ to $\mathbb{Z}^2$ using the pullback (sampling), and then move it from $\mathbb{Z}^2$ to $\mathbb{Z}_{10} \oplus \mathbb{Z}_{10}$ using the pushforward (periodization). Both the pushforward and pullback are defined in Section 6.5.

```
func_sampled = func.pullback(S)
func_periodized = func_sampled.pushforward(coker_A)
```

Using the three mathematical objects `LCA`, `HomLCA` and `LCAFunc` and their associated mathematical operations, we have sampled and periodized $f(x)$ in only 12 lines of code. Figure 2.4 depicts graphically the sampling and periodization of $f(x)$.

Though many mathematical and computational details were omitted, the reader will hopefully have an idea of what the thesis is about and what the software is capable of. More code is found in Chapter 7, as well as in Appendix B. It should be said that `abelian` implements many more mathematical operations than those shown in the preceding examples.

Chapter 3

Preliminaries

This chapter on preliminaries has two purposes: (1) to remind the reader of important definitions and theorems, and (2) to establish notation and terminology for the remainder of the thesis. The reader who is comfortable with basic group theory, category theory and Fourier analysis may skim this chapter. If the reader is somewhat familiar with the concepts, this chapter will hopefully serve as a quick refresher of knowledge. Literature references are given in the text.

3.1 Properties of integers and set functions

We start with some elementary integer algorithms, and some definitions related to set functions. The integer algorithms will be used in Chapter 4 on integer linear algebra, and the definitions related to set functions will be generalized by category theory and used extensively in Chapter 5.

3.1.1 The division algorithm and Euclid's algorithm

We recall the existence of two important algorithms which will be used later in the thesis: the division algorithm and the extended Euclidean algorithm.

Definition 3.1 (Division algorithm). *Given two integers a and $b \neq 0$, the division algorithm finds a quotient q and a remainder r such that*

$$a = qb + r, \quad 0 \leq r < |b| .$$

⌟

Definition 3.2 (Extended Euclidean algorithm). *Given two non-zero integers a and b, the extended Euclidean algorithm finds the greatest common*

divisor $\gcd(a, b) = c$ *along with Bézout coefficients* r *and* s *such that*

$$ar + bs = c = \gcd(r, s).$$

⌟

Euclid's algorithm efficiently computes the greatest common divisor of two integers, while the extended algorithm also returns the Bézout coefficients.

Example 3.3 (Extended Euclidean algorithm). If $a = 14$ and $b = 8$, then extended Euclidean algorithm finds $14(-1) + 8(2) = 2 = \gcd(14, 8)$. ⌟

Quotients and remainders as defined in the division algorithm are typically calculated using integer division and the modulus operation in programming languages. The extended Euclidean algorithm is commonly implemented in higher level languages, and it's implementation is relatively simple. Details related to both of these algorithms can be found in the introductory chapter of [David Steven. Dummit, 2004].

3.1.2 Set functions

A function $f : A \to B$ is a mapping from a set A to a set B. The set A is called the domain (or source) and the set B is called the codomain (or target).

Definition 3.4 (Injective function). *A set function* $f : A \to B$ *is injective if and only if* $f(a_1) = f(a_2) \Rightarrow a_1 = a_2$*, where* $a_1, a_2 \in A$*.* ⌟

Given an injective function $f : A \to B$, we can find a right-inverse $f^{-1} : B \to A$ such that $f^{-1}(f(a)) = a$ for every $a \in A$. If $f(x)$ is injective, then the equation $f(x) = b$ may or may not have a solution, but if a solution exists it is unique.

Definition 3.5 (Surjective function). *A set function* $f : A \to B$ *is surjective if and only if for every* $b \in B$ *there exists an* $a \in A$ *such that* $f(a) = b$*.* ⌟

Given a surjective function $f : A \to B$, we can find a left-inverse $f^{-1} : B \to A$ such that $f\left(f^{-1}(b)\right) = b$ for every $b \in B$. If $f(x)$ is surjective, then the equation $f(x) = b$ has a solution, but the solution need not be unique.

Definition 3.6 (Bijective function). *A set function* $f : A \to B$ *is bijective if it is injective and surjective.* ⌟

Bijective functions have right and left inverses. Given an equation $f(x) = b$ there exists a unique solution. A bijective function between A and B may be thought of as a relabeling of the elements, since every $a \in A$ can be associated with a unique $b \in B$ and vice versa.

3.2 Group theory

Group theory is the study of an abstract mathematical structure known as a group. The group concept was formalized in the 1800s, and is still an active research area to this day. Some applications of group theory are codes and cryptography, harmonic analysis and combinatorics.

The following section summarizes some properties of abelian groups: what they are, homomorphisms between them, how to create new groups from a group, and what a finitely generated abelian group (FGA) is. Three sources have been used for this section: [Ledermann, 1996] is an introductory text about group theory, [David Steven. Dummit, 2004] presents group theory and other parts of abstract algebra, while [Aluffi, 2009] introduces group theory alongside category theory.

3.2.1 Basic group theory

Definition 3.7 (Abelian group). *A group G is a set along with a binary operation $+$ such that the following properties are satisfied.*

1. *Closure: $g_1 + g_2 \in G$ for every $g_1, g_2 \in G$.*
2. *Associativity: $(g_1 + g_2) + g_3 = g_1 + (g_2 + g_3)$ for every $g_1, g_2, g_3 \in G$.*
3. *Identity: there exists an identity element, denoted 0, such that $g + 0 = 0 + g = g$ for every $g \in G$.*
4. *Inverse: for every $g \in G$ there exists an inverse element $-g \in G$, such that $g + (-g) = (-g) + g = 0$.*

A group G is said to be abelian if, in addition to the above, the binary operation is commutative so that $g_1 + g_2 = g_2 + g_1$ for every $g_1, g_2 \in G$. ⌟

The binary operation is in principle arbitrary, but will in our case often be addition or multiplication. In the literature authors typically employ multiplicative or additive notation, where the latter is more common for abelian groups. In this thesis all groups will be abelian, and we will use the additive notation. Thus by $g_1 n$ we mean the sum $g_1 + g_1 + \cdots + g_1$ with n terms, and we define $g_1 - g_2 := g_1 + (-g_2)$ for every $g_1, g_2 \in G$.

Definition 3.8 (Order of a group). *The order of a group G is the number of elements in the group, and will be denoted by $|G|$. If G does not have a finite number of elements, then the order of G is said to be infinite.* ⌟

Definition 3.9 (Order of a group element). *The order of an element $g \in G$ is the smallest positive integer m such that $gm = 0$. If no such m exists, then the order of g is said to be infinite. The order of g will be denoted by $|g|$.* ⌟

Proposition 3.10 (Computing the order of a group element in $\mathbb{Z}_n$). *The order of an element $g \in \mathbb{Z}_n$ is $n/\gcd(g,n)$ when $n > 1$, when $n = 1$ the order is 1 by definition.*

Proof. The order of g is the smallest m such that $gm = 0 \mod n$. We write $gm = nk$ for some positive k. The least common multiple of g and n is the value of gm for which this is satisfied. Therefore we have $gm = \operatorname{lcm}(g,n)$, and we solve for m using the fact that $\operatorname{lcm}(g,n) = gn/\gcd(g,n)$. □

Definition 3.11 (Generators of a group). *If every element in a group G can be expressed as a linear combination of a set of group elements $S = \{g_1, g_2, \ldots, g_n\}$, then we say that G is generated by the set S. We denote this as $G = \langle S \rangle = \langle g_1, g_2, \ldots, g_n \rangle$.* ⌟

Definition 3.12 (Cyclic group). *A cyclic group G is a group generated by a single element $g \in G$. A cyclic group can be written as $G = \langle g \rangle$.* ⌟

Example 3.13 (Order and generators of $\mathbb{Z}_4$). Consider the group $\mathbb{Z}_4 = \{0,1,2,3\}$. The order of $\mathbb{Z}_4$ is 4, while $|0| = 1$, $|1| = 4$, $|2| = 2$ and $|3| = 4$. Both 1 and 3 are generators of $\mathbb{Z}_4$, since repeated addition will produce every element in $\mathbb{Z}_4$. The group $\mathbb{Z}_4$ is a cyclic group. ⌟

Definition 3.14 (Group homomorphism). *Let G be a group with binary operation $\bullet$ and let H be a group with binary operation $\circ$. A group homomorphism is a function $\phi : H \to G$ which preserves the binary operation in the sense that*

$$\phi(h_1 \circ h_2) = \phi(g_1) \bullet \phi(g_2)$$

for every $h_1, h_2 \in H$ and every $g_1, g_2 \in G$. In other words, the following diagram commutes.

$$\begin{array}{ccc} H \times H & \xrightarrow{\phi \times \phi} & G \times G \\ \downarrow{\scriptstyle \circ} & & \downarrow{\scriptstyle \bullet} \\ H & \xrightarrow{\phi} & G \end{array}$$

⌟

Definition 3.15 (Isomorphic groups). *If there exists a bijective group homomorphism between two groups G and H, then G and H are said to be isomorphic, denoted $G \cong H$.* ⌟

Stated differently, two groups are isomorphic if they are unique up to a relabeling of the elements, where the binary operation respects the relabeling.

Definition 3.16 (Kernel of a homomorphism). *Given a homomorphism $\phi : G \to H$, the kernel of ϕ is the set of elements in G which map to the identity element in H, i.e.* $\ker(\phi) = \{g \in G \mid \phi(g) = 0\}$. ⌟

Definition 3.17 (Image of a homomorphism). *Given a homomorphism $\phi : G \to H$, the image of ϕ is the set of elements in H which is mapped to from some $g \in H$, i.e.* $\mathrm{im}(\phi) = \{h \in H \mid h = \phi(g) \text{ for some } g \in G\}$. ⌟

3.2.2 Creating smaller and larger groups

Having defined abelian groups and their basic properties, we now turn our attention to some ways of creating "smaller" and "larger" groups.

Definition 3.18 (Subgroup). *Let G be a group. A subgroup of G is a subset which is also a group under the same binary operation as G. If H is a subgroup of G, we denote it by $H \leq G$.* ⌟

Two special subgroups of a group G are $\{0\}$ (the trivial subgroup) and G itself. A subgroup which is not G itself is called a proper subgroup, analogous to the notion of a proper subset in set theory.

Example 3.19 (Subgroup defined by homomorphism). The homomorphism $\phi(g) = 2g$ with source $\mathbb{Z}$ and target $\mathbb{Z}_6$ defines a proper subgroup of $\mathbb{Z}_6$, namely $\{0, 2, 4\}$, which is isomorphic to $\mathbb{Z}_3$. ⌟

Definition 3.20 (Cosets). *Let G be an abelian group with a subgroup $H \leq G$, then $g + H = \{g + h \mid h \in H\}$ is the coset of H in G with respect to g.* ⌟

Definition 3.21 (Quotient group). *Let H be a subgroup of an abelian group G. The quotient group G/H has elements corresponding to all cosets $g+H$, and the group operation is defined as $(g_1+H)+(g_2+H) = g_1+g_2+H$.* ⌟

For a more rigorous construction of the quotient group, see Chapter 3 of

[Ledermann, 1996]. Two abelian groups which can be formed using the quotient are $\mathbb{Z}_n \cong \mathbb{Z}/(n\mathbb{Z})$ and $T = \mathbb{R}/\mathbb{Z}$, we will study both of these in the thesis. The natural way to combine two groups is using the direct sum.

Definition 3.22 (Direct sum). *Let G be an abelian group with binary operation $\bullet$ and let H be an abelian group with binary operation $\circ$. The direct sum of G and H is a group $G \oplus H$. The elements $(g, h) \in G \oplus H$ are Cartesian products and the binary operation $+$ is defined as $(g_1, h_1) + (g_2, h_2) = (g_1 \bullet g_2, h_1 \circ h_2)$ for every $h_1, h_2 \in H$ and every $g_1, g_2 \in G$.* ⌟

3.2.3 Finitely generated abelian groups

We now define FGAs and state some of their important properties. These groups, and homomorphisms between them, will be studied in Chapter 5.

Definition 3.23 (Finitely generated abelian group). *A finitely generated abelian group (FGA) is a commutative group which is generated by a finite set of generators. If G is an FGA, then G may be expressed as $\langle g_1, g_2, \ldots, g_n \rangle$ for a finite n.* ⌟

Every abelian group of finite order is finitely generated, but the converse is not true in general: an FGA is not necessarily of finite order. For instance the group $\mathbb{Z}$ is generated by $\langle 1 \rangle$, so it is an FGA, but not of finite order.

Every FGA is isomorphic to $\mathbb{Z}$, $\mathbb{Z}_n$ or a direct sum of these groups. A compact notation for an FGA is to write $\mathbb{Z}_{\mathbf{p}}$, where $p = (p_1, p_2, \ldots, p_k)$ is a vector of integers with $p_i \geq 0$ for $i = 1, 2, \ldots, k$. If $p_i = 0$, then the i'th group in the direct sum is $\mathbb{Z}$. If $p_i > 0$, then the i'th group in the direct sum is $\mathbb{Z}_{p_i}$. Notice that p is only in boldface when it's a subscript, to remind us that it's a vector (or multi-index) and not an integer.

The fundamental theorem of FGAs provides a canonical way to structure an FGA. There are two canonical decompositions of FGAs, the invariant factor decomposition and the elementary divisor decomposition. We will use the invariant factor decomposition, which we now define.

Theorem 3.24 (Fundamental theorem of FGAs). *Every FGA is isomorphic to a unique direct sum of a free abelian group and torsion group. If G is an FGA, then*

$$G \cong \mathbb{Z}^r \oplus \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2} \oplus \cdots \oplus \mathbb{Z}_{n_s},$$

where $\mathbb{Z}^r$ is the free subgroup (with $r \geq 0$) and $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2} \oplus \cdots \oplus \mathbb{Z}_{n_s}$ is the torsion subgroup. Furthermore the integers $n_1, n_2, \ldots, n_s$ satisfy the following properties:

1. $n_i \geq 2$ *for every* $i = 1, 2, \ldots, s$.
2. n_i *divides* n_{i+1} *for every* $i = 1, 2, \ldots, s-1$.

Proof. See Chapter 8 in [Ledermann, 1996] for a proof. A more general proof may be found in chapter 12 in [David Steven. Dummit, 2004]. □

The terms "free subgroup" and "torsion subgroup" are prevalent in the literature. The free subgroup is the part isomorphic to $\mathbb{Z}^r$ for some $r \geq 0$, and the torsion subgroup is a group of finite order, i.e. $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2} \oplus \cdots \oplus \mathbb{Z}_{n_s}$ where every $n_i \geq 2$ for $i = 1, 2, \ldots, s$. The trivial subgroup $\mathbb{Z}_1 \cong \mathbb{Z}^0$ is the special case when $r = 0$ and $s = 0$.

Definition 3.25 (Rank of a finitely generated abelian group). *Let* $G = \mathbb{Z}_{\mathbf{p}}$*, where* $p = (p_1, p_2, \ldots, p_k)$*. We take* $p_i = 0$ *to mean that the i'th group is* $\mathbb{Z}$*. We define the free rank of* G *as the number of zeros in* p*. We define the rank of* G *as the number of non-trivial groups in* G*, i.e. the number of* p_i*'s for which* $p_i \neq 1$*.* ⌟

The definition of the rank of an FGA varies somewhat in the literature. The definition given above has the pleasant property that for $G = \mathbb{Z}_{\mathbf{p}}$ we have

$$\operatorname{rank}(\mathbb{Z}_{\mathbf{p}}) = \bigoplus_{p_i \in \mathbf{p}} \operatorname{rank}(\mathbb{Z}_{p_i})$$

for both the free rank and the rank.

Definition 3.26 (Canonical generators for an FGA). *Given an FGA in the form* $G = \mathbb{Z}_{\mathbf{p}}$*, where* $p = (p_1, p_2, \ldots, p_k)$*, the canonical generators is the set* $\langle g_1, g_2, \ldots, g_k \rangle$ *with* $g_i = (\delta_{i1}, \delta_{i2}, \ldots, \delta_{ij}, \ldots, \delta_{ik})$ *for* $i = 1, 2, \ldots, k$*. The symbol* δ_{ij} *above denotes the Kronecker delta function, defined as* 1 *if* $i = j$ *and* 0 *if* $i \neq j$*.* ⌟

Example 3.27 (Orders, generators and rank). Consider the FGA $G = \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z}_1 \oplus \mathbb{Z}_2$. In compact notation this would be written as $G = \mathbb{Z}_{\mathbf{p}}$ with $p = (0, 0, 1, 2)$. The order of G is infinite, the canonical generators are $\langle (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1) \rangle$ and G contains one trivial group, namely $\mathbb{Z}_1$. The rank of G is 3 and the free rank of G is 2. ⌟

We will now extend Proposition 3.10, which tells us how to compute the order of an element in $\mathbb{Z}_n$, to arbitrary FGAs.

Proposition 3.28 (Order of a group element in an FGA). *Let* $G = \mathbb{Z}_{\boldsymbol{p}}$ *with* $p = (p_1, p_2, \ldots, p_k)$*, where* $p_i = 0$ *is taken to mean that the i'th group in*

the direct sum is $\mathbb{Z}$*. Let* $g = (g_1, g_2, \ldots, g_k)$ *be an element in* G*. The order of* $g \in G$ *can be computed as*

$$|g| = \operatorname*{lcm}_{i=1}^{k} \left(\frac{p_i}{\gcd(p_i, g_i)} \right),$$

where the lcm *function is taken over all* k *arguments. In the division we define* $0/0$ *as 1.*

Proof. We consider the i'th component of g. If $p_i \neq 0$, then Proposition 3.10 on page 15 established that the order of the i'th component is $p_i / \gcd(p_i, g_i)$. If $p_i = 0$ and $g_i \neq 0$, then the order is infinite, which we associate with 0, this is calculated correctly by $p_i / \gcd(p_i, g_i)$. If $p_i = 0$ and $g_i = 0$, then the order is 1, which is also calculated correctly by $p_i / \gcd(p_i, g_i)$ if we define $0/0$ as 1. Having calculated the order of the i'th component of g, the order of g is the least common multiple of each of the component-wise orders. □

Example 3.29 (A finitely generated abelian group). Consider the FGA $G = \mathbb{Z}_8 \oplus \mathbb{Z}_5$ and two group elements $g_1 = (2, 3)$ and $g_2 = (4, 7)$. One way to visualize G and the two elements g_1 and g_2 is shown in Figure 3.1.

We observe that $(4, 7) \cong (4, 2)$ in G since $7 \cong 2$ in $\mathbb{Z}_5$. Therefore we take g_2 to be represented by $(4, 2)$. The generating set $\langle g_1, g_2 \rangle$ generates a subgroup of G with order 20, i.e. $\langle g_1, g_2 \rangle < G$. The orders of the generators are

$$|g_1| = \operatorname{lcm}\left(\frac{8}{\gcd(8,2)}, \frac{5}{\gcd(5,3)} \right) = \operatorname{lcm}\left(\frac{8}{2}, \frac{5}{1} \right) = \operatorname{lcm}(4, 5) = 20,$$
$$|g_2| = \operatorname{lcm}\left(\frac{8}{\gcd(8,4)}, \frac{5}{\gcd(5,2)} \right) = \operatorname{lcm}\left(\frac{8}{4}, \frac{5}{1} \right) = \operatorname{lcm}(2, 5) = 10.$$

The group generated by g_2 is a subgroup of the one generated by g_1, so that $\langle g_2 \rangle < \langle g_1 \rangle = \langle g_1, g_2 \rangle < G$. We will return to this example again on page 51. ⌟

3.3 Category theory

Category theory is a unifying, abstract framework in which many mathematical structures can be studied. The initial formulation was made in the 1940s by Eilenberg and Mac Lane, and since then it has matured and found numerous applications in science. Although there exist plenty of exotic categories, we will limit our study to relatively simple ones. There are two reasons why we are interested in category theory: (1) it will help us express ideas using diagrams with objects and arrows and (2) it will provide clues as to how software with mathematical objects can be designed. The sources for the following is [Aluffi, 2009] and [Harold. Simmons, 2011].

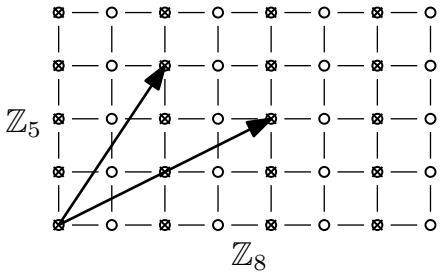


Figure 3.1: A visualization of the group $G = \mathbb{Z}_8 \oplus \mathbb{Z}_5$. The generators $g_1 = (2, 3)$ and $g_2 = (4, 2)$ are shown as arrows. The elements in G which are in $\langle g_1, g_2 \rangle$ are marked with tiny crosses.

3.3.1 Definition, morphisms and diagrams

Definition 3.30 (Category). *A category consists of the following:*

- *A collection of entities called objects: A, B, C, ...*
- *A collection of entities called arrows: f, g, h, ...*

The objects and arrows must satisfy the following properties: (1) every arrow has a source and target object, (2) every object has an identity arrow $\mathrm{Id}_A : A \to A$*, and (3) composing arrows is associative, i.e.* $(f \circ g) \circ h = f \circ (g \circ h)$.

⌟

Name	Objects	Arrows
Set	Sets	Functions
VectR	Vector spaces over $\mathbb{R}$	Linear functions
ModZ	Modules over $\mathbb{Z}$	Linear functions
FinAb	Finitely generated abelian groups	Group homomorphisms
Ab	abelian groups	Group homomorphisms

Table 3.1: The categories that will be used in this thesis.

In Table 3.1, some examples of categories are listed. All of these categories will be used in the thesis, and it is straightforward to verify the existence of an identity and the associative property for each of them.

Monomorphisms and epimorphisms

Monomorphisms and epimorphisms can be thought of as generalizations of injective and surjective functions, respectively. In the category Set the definitions coincide perfectly, but the category theoretic definitions are given

with respect to arrows and objects instead of sets and functions. Such definitions are sometimes called "arrow-theoretic."

Definition 3.31 (Monomorphism). *Consider the diagram:*

$$A \underset{g_2}{\overset{g_1}{\rightrightarrows}} B \xrightarrow{f} C$$

The arrow f is called a monomorphism if $f \circ g_1 = f \circ g_2 \Rightarrow g_1 = g_2$ for every $g_1, g_2 : A \to B$. ⌟

Definition 3.32 (Epimorphism). *Consider the diagram:*

$$A \xrightarrow{f} B \underset{g_2}{\overset{g_1}{\rightrightarrows}} C$$

The arrow f is called an epimorphism if $g_1 \circ f = g_2 \circ f \Rightarrow g_1 = g_2$ for every $g_1, g_2 : A \to B$. ⌟

In this thesis, special arrows will be used to represent monomorphisms and epimorphisms. Hooked arrows will represent monomorphisms and two headed arrows will represent epimorphisms.

Arrow	**Meaning**
$A \xhookrightarrow{f} B$	f is a monomorphism
$A \xrightarrow{g} \!\!\!\!\rightarrow B$	g is an epimorphism

Commutative diagrams

A drawing with objects and arrows is called a diagram, and a commutative diagram is one in which every directed path from an arbitrary object to another commutes.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle n} \\ C & \xrightarrow{m} & D \end{array} \tag{3.1}$$

Stating that Diagram (3.1) commutes means that $n \circ f = m \circ g$.

3.3.2 Initial objects, products and their duals

Going back to Definitions 3.31 and 3.32 of monomorphisms and epimorphisms, we note that each definition can be obtained by switching the direc-

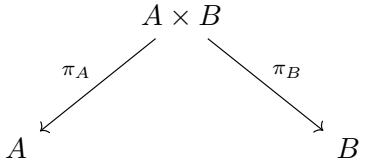
tion of the arrows in the other. In category theory, the dual of a construct is obtained by reversing every arrow, and monomorphisms and epimorphisms are therefore dual constructs. Another important dual pair of definitions are initial objects and final objects.

Definition 3.33 (Initial object). *An object I in a category C is said to be initial if for each object A in C there exists a unique arrow $I \longrightarrow A$.* ⌟

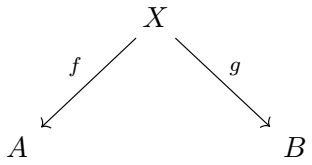
Definition 3.34 (Final object). *An object F in a category C is said to be final if for each object A in C there exists a unique arrow $A \longrightarrow F$.* ⌟

In Set, the initial object is the empty set $\{\}$ and the final object is a set containing a single element $\{a\}$. In VectR the final and initial object both correspond to $\mathbb{R}^0$, and when they coincide the object is referred to as the zero object. It can be shown that initial and final objects are unique up to isomorphism. Initial and final objects allow us to define zero morphisms, which we will define and make use of in Chapter 5.

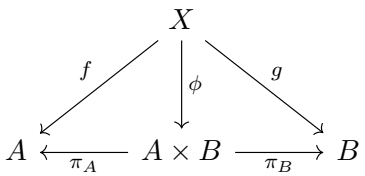
Definition 3.35 (Categorical product). *A product is an object $A \times B$ along with arrows π_A and π_B to A and B, respectively.*



The product satisfies the following property: For every



there exists a unique arrow ϕ such that the following diagram commutes.



⌟

Definition 3.36 (Categorical coproduct). *The categorical coproduct is the dual of the categorical product, so reversing every arrow in Definition 3.35 gives the definition of the coproduct.* ⌟

The coproduct is the dual of the product, and it is common to use the "co-" prefix to signify this. Notice that the definition of the product includes the object $A \times B$ along with the morphisms π_A and π_B, which are called canonical projections. In Set the product is the Cartesian product, while in Ab both the product and coproduct correspond to the direct sum.

Name	Initial object	Final object	Product
Set	$\emptyset$	$\{a\}$	Cartesian product
VectR	$\mathbb{R}^0$	$\mathbb{R}^0$	Direct sum
ModZ	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	Direct sum
FinAb	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	Direct sum
Ab	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	$\mathbb{Z}^0 = \mathbb{Z}_1 = \{0\}$	Direct sum

Table 3.2: Categorical constructs as realized by some concrete categories.

3.4 Fourier analysis

Fourier analysis is named after the French mathematician and physicist Jean-Baptiste Joseph Fourier. He introduced what is now known as Fourier series as a result of his work on heat flow in the early 1800s. The advent of the computer and the rediscovery of the fast Fourier transform (FFT) by Cooley and Tukey in 1965 made Fourier analysis computationally tractable for large amounts of data. Today Fourier analysis enjoys a wide range of applications in both theoretical and applied science.

The purpose of this section is to reintroduce some of the main formulas and results from the study of Fourier series, the discrete Fourier transform and the Fourier transform. A more thorough, generalized exposition of Fourier analysis from the group-theoretic perspective will be given in Chapter 6. For an introduction to Fourier analysis with applications, the reader is referred to [Albert. Boggess, 2009]. A very detailed account of the material is also given in [C. Gasquet, 1999].

3.4.1 Fourier series

Fourier series expresses a periodic function as a sum of trigonometric functions, or equivalently as complex exponentials. The relationship between the trigonometric functions and the complex exponential is given by Euler's formula, which states that $e^{ix} = \cos(x) + i\sin(x)$.

Definition 3.37 (Fourier series). *A piecewise-continuous function $f(x) : \mathbb{R} \to \mathbb{C}$ with a period on a bounded interval $[a, b)$ has the following Fourier*

series representation, where $d = b - a$.

$$f(x) = \sum_{\xi=-\infty}^{\infty} \widehat{f}(\xi) \exp\left(2\pi i \frac{x\xi}{d}\right), \quad \widehat{f}(\xi) = \frac{1}{d}\int_a^b f(x) \exp\left(-2\pi i \frac{x\xi}{d}\right) dx$$

The expression $f(x)$ *above is the Fourier series representation, and the function* $\widehat{f}(\xi) : \mathbb{Z} \to \mathbb{C}$ *is called the Fourier series coefficients.* ⌟

A piecewise continuous function $f(x)$ on a bounded interval $[a, b)$ is said to be of bounded variation, and it can be shown that functions of bounded variation have Fourier series representations which converge pointwise. If there are jump discontinuities, then the Gibbs phenomenon will occur, seen as an overshoot/undershoot near the discontinuities. The Gibbs phenomenon is shown in Figure 3.2.

If $f(x)$ is continuous and differentiable (except possibly at a finite number of points) on $[a, b)$ and $f'(x)$ is piecewise continuous, then the Fourier series representation converges uniformly. The smoother the function is the more rapidly the Fourier coefficients $\widehat{f}(\xi)$ will decay as $|\xi| \to \infty$. For a detailed account about convergence, see for instance Lesson 5 in [C. Gasquet, 1999].

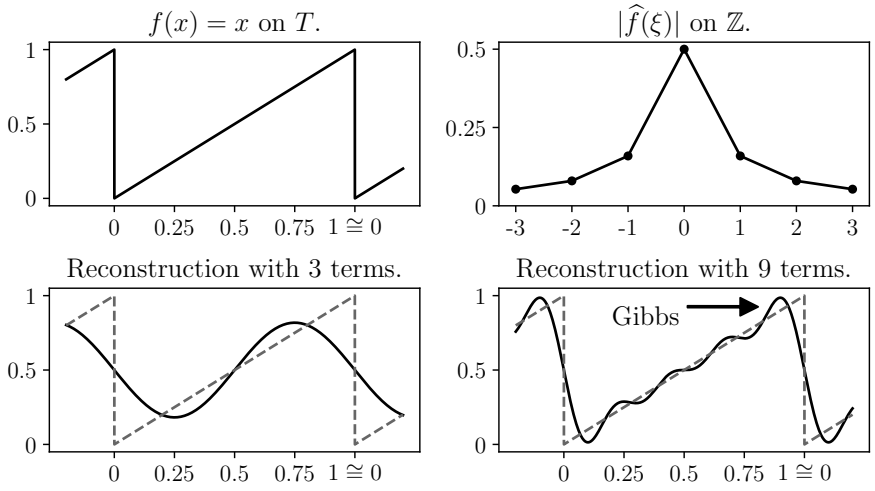


Figure 3.2: The top left plot shows $f(x) = x$ on the periodic domain $T = \mathbb{R}/\mathbb{Z}$. The top right plot shows the Fourier coefficients of $f(x)$, plotted in absolute value since $\widehat{f}(\xi)$ is in general a complex number. The bottom left plot shows the reconstruction of $f(x)$ using 3 terms in the sum in Definition 3.37 of the Fourier series. The bottom right plot shows the reconstruction using 9 terms in the sum, where the Gibbs phenomenon is starting to show.

Example 3.38 (Fourier series). Consider $f(x) = x$ defined on $T = [0, 1)$.

We calculate $\widehat{f}(\xi) = \int_0^1 x \exp(-2\pi i x\xi)\, dx$. When $\xi = 0$ the integral is $1/2$, when $\xi \neq 0$ we use partial integration to obtain $-1/(2\pi i\xi)$. In summary, the Fourier series coefficients are given by

$$\widehat{f}(\xi) = \begin{cases} 1/2 & \text{if } \xi = 0 \\ -1/(2\pi i\xi) & \text{if } \xi \in \mathbb{Z}\setminus\{0\}. \end{cases}$$

The function $f(x)$ and it's Fourier series coefficients are plotted in Figure 3.2, along with approximations obtained by truncating the sum. ⌟

3.4.2 The Fourier transform

The Fourier transform may be thought of as the limit of Fourier series, as the time period goes to infinity. While the Fourier series representation sums over integer frequencies, the Fourier transform integrates over the real line to reproduce a function.

Definition 3.39 (Fourier transform). *A sufficiently nice function $f(x)$ can be reconstructed using the Fourier transform of $f(x)$, which is denoted as $\widehat{f}(\xi)$ or $\mathcal{F}(f(x))$.*

$$f(x) = \int_{-\infty}^{\infty} \widehat{f}(\xi) \exp(2\pi i x\xi)\, d\xi \qquad \widehat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \exp(-2\pi i x\xi)\, dx$$

⌟

If a function $f(x)$ and it's Fourier transform $\mathcal{F}(f) = \widehat{f}(\xi)$ are both absolutely integrable (i.e. $\int_{\mathbb{R}} |f(x)|\, dx < \infty$), then $f(x)$ can be recovered from its Fourier transform almost everywhere. A discussion of the many properties of the Fourier transform is outside the scope of this thesis, but we mention a particularly important fact: the Fourier transform diagonalizes convolutions.

Definition 3.40 (Convolution of functions on $\mathbb{R}$). *Let $f : \mathbb{R} \to \mathbb{C}$ and $g : \mathbb{R} \to \mathbb{C}$ be functions, and denote the space of such functions as $\mathbb{C}^{\mathbb{R}}$. The convolution of f and g, denoted $f * g : \mathbb{C}^{\mathbb{R}} \times \mathbb{C}^{\mathbb{R}} \to \mathbb{C}^{\mathbb{R}}$, is defined as*

$$(f * g)(\xi) = \int_{-\infty}^{\infty} f(x) g(\xi - x)\, dx.$$

The convolution is a new function, and analogous definitions exists for domains other than $\mathbb{R}$. ⌟

The Fourier transform converts convolution to point-wise multiplication, so that $\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$. In other words, the following diagram commutes.

$$\begin{array}{ccc} \mathbb{C}^{\mathbb{R}} \times \mathbb{C}^{\mathbb{R}} & \xrightarrow{\mathcal{F}(\cdot)\times\mathcal{F}(\cdot)} & \mathbb{C}^{\mathbb{R}} \\ \Big\downarrow \text{convolution} & & \Big\downarrow \text{multiplication} \\ \mathbb{C}^{\mathbb{R}} \times \mathbb{C}^{\mathbb{R}} & \xrightarrow[\mathcal{F}(\cdot)]{} & \mathbb{C}^{\mathbb{R}} \end{array} \tag{3.2}$$

Example 3.41 (Fourier transform of a Gaussian). We calculate the Fourier transform of the Gaussian $f(x) = \exp(-kx^2)$ defined on $\mathbb{R}$. The Fourier transform is $\widehat{f}(\xi) = \int_{-\infty}^{\infty} \exp\left(-kx^2 - 2\pi i x\xi\right)\, dx$, and after completing the square in the exponent and simplifying we obtain

$$\widehat{f}(\xi) = \exp\left(-\pi^2\xi^2/k\right) \int_{-\infty}^{\infty} \exp\left(-k\left(x + \pi i\xi/k\right)^2\right)\, dx.$$

A change of variable to $\zeta = x + \pi i\xi/k$ and using the well-known fact that $\int_{-\infty}^{\infty} \exp(-k\zeta^2)\, d\zeta = \sqrt{\pi/k}$ yields the answer, which is

$$\mathcal{F}\left(\exp(-kx^2)\right) = \sqrt{\pi/k} \exp\left(\frac{-\pi^2\xi^2}{k}\right).$$

In other words, the Fourier transform of a Gaussian function is a Gaussian. When $k = \pi$ then $\mathcal{F}(f) = f$. The multidimensional extension is

$$\mathcal{F}\left(\exp(-kx^T x)\right) = (\pi/k)^{d/2} \exp\left(-\pi^2\xi^T\xi/k\right),$$

where $x, \xi \in \mathbb{R}^d$ and $x^T x = \sum_{j=1}^{d} x_j^2$. ⌟

3.4.3 The discrete Fourier transform

The discrete Fourier transform performs Fourier analysis on an array of numbers, making it important in numerical computations. It can be derived as an approximation to Fourier coefficients, as is done in [C. Gasquet, 1999]. With the discrete Fourier transform, there are no issues with convergence as long as $|f(x)| < \infty$ for every $x \in \mathbb{Z}_n$.

Definition 3.42 (Discrete Fourier transform). *The discrete Fourier transform of $f(x)$ on $\mathbb{Z}_n$ is denoted as $\widehat{f}(\xi)$, and defined below. The equation to the left reconstructs $f(x)$ from $\widehat{f}(\xi)$.*

$$f(x) = \sum_{\xi=0}^{n-1} \widehat{f}(\xi) \exp\left(2\pi i \frac{x\xi}{n}\right) \qquad \widehat{f}(\xi) = \frac{1}{n} \sum_{x=0}^{n-1} f(x) \exp\left(-2\pi i \frac{x\xi}{n}\right)$$

⌟

The transform of a vector $f(x_j) = f_j$ of length n can be thought of as matrix multiplication $\widehat{f}_k = n^{-1} \sum_{j=1}^{n} F_{kj} f_j$, where

$$F_{kj} = \exp\left(-2\pi i \frac{(k-1)(j-1)}{n}\right)$$

for rows $k = 1, 2, \ldots, n$ and columns $j = 1, 2, \ldots, n$. Chapter 4 in [Strang, 1986] introduces the FFT as an efficient algorithm for this matrix-vector product, bringing the cost down from the ordinary $\mathcal{O}(n^2)$ complexity of matrix-vector multiplication to $\mathcal{O}(n \log n)$. A very enjoyable and detailed account is found in [Dasgupta, 2008], where the FFT is put into algorithmic context—it's a divide and conquer algorithm.

Chapter 4

Integer linear algebra

This chapter is about linear functions $A : \mathbb{Z}^n \to \mathbb{Z}^m$, which are represented by matrices with integer entries. In the abstract setting, such maps are called homomorphisms between $\mathbb{Z}$-modules. Our approach will be concrete, but a more abstract introduction to module theory is found for instance in Chapter 17 of [Nicholas. Loehr, 2014].

We will examine unimodular matrices, which are automorphisms on $\mathbb{Z}^n$, and define some special unimodular matrices which will be used to factor A. The factorizations we will introduce are the Hermite normal form (HNF) and the Smith normal form (SNF). The definitions of these factorizations, along with algorithms for computing them, will be presented. This chapter is mainly concerned with "what" and "how"—the "why" is explained in detail in Chapter 5, where the factorizations will be used to solve more general problems.

4.1 Unimodular matrices

Definition 4.1 (Unimodular matrix). *A matrix $U \in \mathbb{Z}^{n\times n}$ is unimodular if* $\det(U) = \pm 1$. *The set of all unimodular matrices of size $n \times n$ will be denoted by* $\mathrm{GL}(n, \mathbb{Z})$, *the general linear group of size n over $\mathbb{Z}$.* ⌟

The unimodular matrices are the integer matrices $U \in \mathbb{Z}^{n\times n}$ whose inverses U^{-1} are also in $\mathbb{Z}^{n\times n}$. Starting with any matrix $U \in \mathrm{GL}(n, \mathbb{Z})$, three elementary unimodular operations may be applied to U without spoiling the unimodularity.

Definition 4.2 (Elementary unimodular operations). *There are three elementary unimodular operations: (1) multiplying a column by -1, (2)*

swapping (permuting) two columns and (3) adding an integer multiple of a column to another column. These three operations can also be applied to rows. ⌟

The elementary column and row operations may be expressed as matrix multiplications acting on the right and left, respectively. The matrices corresponding to the three elementary operations listed in Definition 4.2 above have determinants -1, -1 and 1, respectively. Every unimodular matrix $U \in \mathrm{GL}(n, \mathbb{Z})$ can be generated by a sequence of elementary unimodular operations applied to the identity matrix.

4.1.1 Non-trivial unimodular transformations

We now introduce some unimodular transformations, represented by matrices, which will be used in algorithms for computing the HNF and the SNF. They are based on Definitions 3.1 and 3.2 of the division algorithm and the extended Euclidean algorithm, which were presented on page 12.

Definition 4.3 (Division algorithm transform). *Consider a row vector* $\begin{pmatrix} r & s \end{pmatrix}$ *with* $r, s \in \mathbb{Z}$ *and* $s > 0$*. The row vector can be transformed so that the leftmost entry becomes non-negative and smaller than* s *by multiplication with a unimodular matrix. Using the division algorithm we can write* $r = sa + b$ *where* a *is the integer quotient of the division* r/s *and* $b \geq 0$ *is the remainder. Using the quotient* a *from the division algorithm we construct the unimodular transformation matrix as given below.*

$$\begin{pmatrix} r & s \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -a & 1 \end{pmatrix} = \begin{pmatrix} b & s \end{pmatrix}$$

⌟

Definition 4.4 (Elementary Hermite transform). *Consider a row vector* $\begin{pmatrix} r & s \end{pmatrix}$ *with* $r, s \in \mathbb{Z}$*. In order to compute the HNF, we need to transform* $\begin{pmatrix} r & s \end{pmatrix}$ *to* $\begin{pmatrix} c & 0 \end{pmatrix}$ *with* $c \in \mathbb{Z}$ *by right-multiplication with a unimodular matrix. This is achieved by using the extended Euclidean algorithm to compute integers* $a, b, c \in \mathbb{Z}$ *such that* $ar + bs = c = \gcd(r, s)$*. The desired unimodular matrix is given below.*

$$\begin{pmatrix} r & s \end{pmatrix} \begin{pmatrix} a & -\frac{s}{c} \\ b & \frac{r}{c} \end{pmatrix} = \begin{pmatrix} c & 0 \end{pmatrix} = \begin{pmatrix} \gcd(r, s) & 0 \end{pmatrix}$$

⌟

Definition 4.5 (Divisibility transform). *When computing the SNF, we will need to transform a diagonal matrix* $\mathrm{diag}(r, s)$ *with* $r, s \in \mathbb{Z}$ *to a diagonal*

matrix $\operatorname{diag}(a_1, a_2) \in \mathbb{Z}$ *where* a_1 *divides* a_2*. To achieve this, unimodular transformations are used. We use the extended Euclidean algorithm to compute integers* $a, b, c \in \mathbb{Z}$ *such that* $ar + bs = c = \gcd(r, s)$*. The required transformation is*

$$\begin{pmatrix} a & b \\ -\frac{s}{c} & \frac{r}{c} \end{pmatrix} \begin{pmatrix} r & 0 \\ 0 & s \end{pmatrix} \begin{pmatrix} 1 & -b\frac{s}{c} \\ 1 & a\frac{r}{c} \end{pmatrix} = \begin{pmatrix} c & 0 \\ 0 & \frac{rs}{c} \end{pmatrix}, \tag{4.1}$$

where $c = \gcd(r, s)$ *and* $rs/c = \operatorname{lcm}(r, s)$*. It is easily verified that both of the unimodular matrices above have determinants equal to* 1*.* ⌟

The source of the divisibility transform is Chapter 8 in [Charles C. Sims, 1994].

4.2 The Hermite normal form

Recall from linear algebra over $\mathbb{R}$ that the reduced echelon form of a matrix $A \in \mathbb{R}^{m \times n}$ is the result of repeatedly applying elementary operations to A, see for instance Chapter 4 in [Nicholas. Loehr, 2014] for a description of the algorithm. Analogously, a matrix $A \in \mathbb{Z}^{m \times n}$ can be put in HNF by repeatedly performing elementary unimodular operations on A. We define the HNF, assert that every integer matrix can be put into HNF, and present an algorithm for doing so. More information about the HNF is found in Chapter 8 in [Charles C. Sims, 1994], as well as in [Munthe-Kaas, 2016].

Definition 4.6 (Hermite normal form). *A matrix* $H \in \mathbb{Z}^{m \times n}$ *is said to be in column Hermite normal form if the following conditions hold.*

i) There exists an $0 \leq r \leq n$ *such that the first* r *columns of* H *contain one or several non-zero entries and the remaining* $n - r$ *columns are identically zero.*

ii) For each column $1 \leq j \leq r$*, the first non-zero entry from the top is called the pivot element. The pivot elements* $\mathbf{h}_{i(j),j}$ *are positive and* $i(j+1) > i(j)$ *for all* $1 \leq j \leq r$*.*

iii) For every pivot element $\mathbf{h}_{i(j),j}$ *it is true that* $0 \leq h_{i,s} < \mathbf{h}_{i(j),j}$ *for all* $1 \leq s < j$*.*

⌟

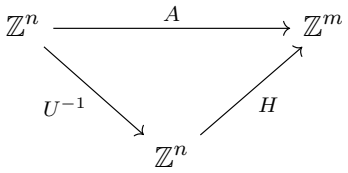
Example 4.7 (Structure of a matrix in HNF). Below we observe the structure of a somewhat typical matrix in HNF. The $\mathbf{h}_{i(j),j}$'s are pivots, and every entry to the left of a pivot must be smaller than the pivot. No

requirement is made of h_{31} and h_{32}, since row 3 has no pivots.

$$\begin{pmatrix} \mathbf{h}_{i(1),1} & & & & \\ h_{21} & \mathbf{h}_{i(2),2} & & & \mathbf{0} \\ h_{31} & h_{32} & & & \\ h_{41} & h_{42} & \mathbf{h}_{i(3),3} & & \\ h_{51} & h_{52} & h_{53} & \mathbf{h}_{i(4),4} & \end{pmatrix}$$

⌟

Theorem 4.8 (Hermite normal form). *Every matrix $A \in \mathbb{Z}^{m\times n}$ can be put in (column) Hermite normal form by some $U \in \mathrm{GL}(n,\mathbb{Z})$. We obtain $AU = H$, where H is in Hermite normal form, and the following diagram commutes.*



Proof. Algorithm 1 on page 34 provides the constructive proof. □

Example 4.9 (Matrix in HNF). Below is an example of $AU = H$. It is straightforward to check that $U \in \mathrm{GL}(3,\mathbb{Z})$ by verifying that $\det(U) = 1$. It is also straightforward to verify that H is in HNF. The pivots of H are 1, 1 and 8.

$$\begin{pmatrix} 0 & 1 & 6 \\ 0 & 1 & 7 \\ 8 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 7 & -6 & 0 \\ -1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 6 & 2 & 8 \end{pmatrix}$$

⌟

4.3 The Smith normal form

Just as the reduced echelon form and the Hermite normal form are analogous, the Smith normal form (SNF) of $A \in \mathbb{Z}^{m\times n}$ is analogous to the singular value decomposition of $A \in \mathbb{R}^{m\times n}$. While the singular value decomposition uses orthogonal matrices to diagonalize $A \in \mathbb{R}^{m\times n}$, the SNF uses elementary unimodular operations to diagonalize $A \in \mathbb{Z}^{m\times n}$. We define the SNF, assert that every integer matrix can be put into SNF, and present an algorithm for doing so.

Definition 4.10 (Smith normal form). *An integer matrix $\Sigma \in \mathbb{Z}^{m\times n}$ is said to be in Smith normal form if the following conditions hold.*

i) Σ is diagonal, i.e. $\Sigma_{ij} = 0$ when $i \neq j$.

ii) The first r diagonals $\sigma_i = \Sigma_{ii}$ are positive, and σ_i divides σ_{i+1} for all $1 \leq i < r-1$, where r denotes the number of positive diagonal entries. The remaining $\min(m,n) - r$ diagonal entries are zero.

⌟

Theorem 4.11 (Smith normal form). *Every integer matrix $A \in \mathbb{Z}^{m\times n}$ can be put into SNF by matrices U and V such that $UAV = \Sigma$, where $U \in \mathrm{GL}(m,\mathbb{Z})$, $V \in \mathrm{GL}(n,\mathbb{Z})$ and $\Sigma \in \mathbb{Z}^{m\times n}$ is in SNF. In other words, $A : \mathbb{Z}^n \to \mathbb{Z}^m$ can be factored such that the following diagram commutes.*

$$\begin{array}{ccc} \mathbb{Z}^n & \xrightarrow{\ A\ } & \mathbb{Z}^m \\ {\scriptstyle V^{-1}}\downarrow & & \uparrow{\scriptstyle U^{-1}} \\ \mathbb{Z}^n & \xrightarrow[\ \Sigma\ ]{} & \mathbb{Z}^m \end{array}$$

Proof. Algorithm 2 on page 36 provides a constructive proof. □

Example 4.12 (Matrix in Smith normal form). Here is an example of $UAV = \Sigma$. The matrices U and V both have determinant 1 and are unimodular. The division criterion is clearly satisfied for 1, 1 and 8.

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 6 \\ 0 & 1 & 7 \\ 8 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 1 & -6 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 8 \end{pmatrix}$$

⌟

4.4 Algorithms and computational issues

We now present algorithms for putting $A \in \mathbb{Z}^{m\times n}$ in HNF and SNF. The algorithms provide constructive proofs of the factorizations. The primary source for the algorithms is Chapter 9 in [Derek F. Holt, 2005]. An attempt has been made to make the algorithms more readable by introducing unimodular matrices ρ_1, ρ_2 and ρ_3, as is done in [Jäger and Wagner, 2009].

Although various algorithms exist in the literature, general implementations are somewhat hard to find: MATLAB only has an implementation of the SNF for square A, and none of the popular scientific libraries in Python implement these algorithms.

The algorithms were implemented in Python as part of this thesis, and the implementations are found in Appendix A. As we will see in Chapter 5, these algorithms are crucial for computations with FGAs. In the actual code, several optimizations were made. The most significant optimization

is that sparse matrices are never explicitly formed and multiplied—instead their actions are implemented. This optimization brings the cost of some multiplications down from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$. Additional savings can be realized when taking linear combinations of rows and columns with a significant number of zeros, these zeros are ignored in the optimized code. The book [Lloyd N. Trefethen, 1997] is a good reference for how to efficiently implement linear algebra algorithms.

4.4.1 Unimodular matrices for the normal forms

We now extend the division algorithm transform, elementary Hermite transform and divisibility transform presented on page 29 to $n \times n$ matrices. We also define a permutation matrix. These matrices will be used notationally in Algorithm 1 and Algorithm 2 on pages 34 and 36. In the actual implementation, none of these matrices should be explicitly formed.

- The matrix $P(i, j, n) \in \mathrm{GL}(n, \mathbb{Z})$ is the permutation matrix, which permutes columns/rows i and j.
- Based on Definition 4.3 we define $\rho_1(r, s, i, j, n) \in \mathrm{GL}(n, \mathbb{Z})$ as I_n with entry $I_{j,i} := -a$, where a is the integer quotient in the division r/s. The effect of right-multiplication by ρ_1 is "subtract a times column j from column i", and the effect of left-multiplication by ρ_1 is "subtract a times column i from column j."
- Based on Definition 4.4 we define $\rho_2(r, s, i, j, n) \in \mathrm{GL}(n, \mathbb{Z})$ as I_n with the following entries from the extended Euclidean algorithm $ar + bs = c = \gcd(r, s)$.

  - $I_{i,i} := a$
  - $I_{j,i} := b$
  - $I_{i,j} := -s/c$
  - $I_{j,j} := r/c$

  The effect of right-multiplication by ρ_2 is "multiply column i by a and add b times column j, and multiply column j by r/c and subtract s/c times column i."
- Based on Definition 4.5 we define $\rho_3(r, s, i, j, n) \in \mathrm{GL}(n, \mathbb{Z})$ as I_n with the following entries from the extended Euclidean algorithm $ar + bs = c = \gcd(r, s)$.

  - $I_{i,i} := 1$
  - $I_{j,i} := 1$
  - $I_{i,j} := -b\frac{s}{c}$
  - $I_{j,j} := a\frac{r}{c}$

  This matrix is an $n \times n$ extension of the rightmost factor in Equation (4.1). The effect of right-multiplication by ρ_3 is "add column j to

column i, and multiply column j by $a\frac{r}{c}$ and subtract $b\frac{s}{c}$ times column i."

4.4.2 Algorithm for the Hermite normal form

Algorithm 1: Algorithm for the Hermite normal form.

Input : Matrix $A \in \mathbb{Z}^{m\times n}$
Output : Matrices $H \in \mathbb{Z}^{m\times n}$ and $U \in \mathrm{GL}(n,\mathbb{Z})$ such that $H = AU$

```
1  H := copy(A); U := I_n                          // Initialize H and U.
2  j, i := 1, 1                                    // Initialize counters.
3  while i ≤ m do
4  |  if H[i, j :] = 0 then
5  |  |  i := i + 1                  // Skip row if it's identically zero.
6  |  |  continue
7  |  end
8  |  for k in [j + 1, ..., n] do
9  |  |  H := H · ρ2(H[i, j], H[i, k], j, k, n)      // Create zeros to the
10 |  |  U := U · ρ2(U[i, j], U[i, k], j, k, n)      // right of the pivot.
11 |  end
12 |  for k in [0, ..., j − 1] do
13 |  |  H := H · ρ1(H[i, k], H[i, j], k, j, n)     // Reduce elements left
14 |  |  U := U · ρ1(H[i, k], H[i, j], k, j, n)     // of pivot h to [0, h).
15 |  end
16 |  i, j := i + 1, j + 1                  // Increment the counters by 1.
17 |  if j > n then
18 |  |  break
19 |  end
20 end
```

Explanation of Algorithm 1. The algorithm initializes two variables i and j, which are incremented as the algorithm progresses. The counter i is the current row, and the counter j is the current column.

If the if-block on line 4 is triggered, the entries to the right of, and including, the current position (i, j) are identically zero. If this is the case, then row i has no pivot element, and the counter i is incremented and the while-loop starts over in the next row.

If we are on line 8, then some entry to the right of, or including, the current position (i, j) is non-zero. When applying ρ_2 to column j and $k \in [j +$

$1, \ldots, n]$ we assure that (i, j) ends up with a positive pivot, and that every entry to the right of this pivot becomes zero. We require that every element to the right of (i, j) be zero, since if $(i, j + k)$ were not zero for some k, then the second criterion in Definition 4.6 would be violated—there is only one pivot per column.

What remains to do is ensure that every element to the left of the pivot at (i, j) is smaller than the pivot, as stated in Definition 4.6. This is exactly what happens on line 12. Applying ρ_1 to columns j and $k \in [j + 1, \ldots, n]$ ensures that all entries to the left of entry (i, j) satisfy the criterion.

The final lines increment the counters i and j. If $j > n$, then the algorithm is finished. When the algorithm terminates, every row i has been iterated over and the matrix satisfies the definition of the HNF. □

4.4.3 Algorithm for the Smith normal form

Explanation of Algorithm 2. In stage 1, the algorithm will successively create zeros to the right of and below diagonal elements starting in the top-left corner.

The while loop in line 3 will run until all entries below and to the right of (f, f) are zero. Assume that not every such entry is zero. The algorithm will move the minimal entry in the sub-matrix $\Sigma[f :, f :]$[1] to the (f, f) position. Using left and right multiplication by ρ_1, every element to the right of and below (f, f) will be made non-negative and smaller than f. Since the positive entries to the right of and below (f, f) will decrease in every iteration, the while-loop in line 3 will eventually terminate. When it terminates, the row and column corresponding to position (f, f) have been diagonalized, and when stage 1 ends the entire matrix Σ is diagonal.

When stage 2 starts, the matrix is in diagonal form, but the diagonal elements do not necessarily obey the divisibility criterion given in Definition 4.10. The while-loop in line 3 will run on every diagonal entry $\Sigma[f, f]$ for $f \in [1, \ldots, \min(m, n)]$. The nested for-loops in line 21 successively use Equation (4.1) to ensure divisibility.

Let σ_f be the value at entry (f, f). When the inner loop in stage 2 finishes, entry (f, f) will be set to $\gcd(\sigma_f, \gcd(\sigma_{f+1}, \ldots))$. This is equal to $\gcd(\sigma_f, \sigma_{f+1}, \ldots)$, meaning that $\sigma_f = \gcd(\sigma_f, \sigma_{f+1}, \ldots)$ and therefore it divides all other diagonals. Once the outer loop finishes, the divisibility criterion is fulfilled, which can be shown by induction. □

[1] By $\Sigma[f :, f :]$ we mean the lower right submatrix which includes the position (f, f).

Algorithm 2: Algorithm for the Smith normal form.

Input : Matrix $A \in \mathbb{Z}^{m \times n}$
Output : Matrices $\Sigma \in \mathbb{Z}^{m \times n}$, $U \in \mathrm{GL}(m, \mathbb{Z})$ and $V \in \mathrm{GL}(n, \mathbb{Z})$ such that $UAV = \Sigma$

$U := I_m$; $V := I_n$; $\Sigma := \mathrm{copy}(A)$ // Initialize matrices.

// STAGE 1: Create diagonal matrix with positive entries
for f **in** $[1, \ldots, \min(m, n)]$ **do**
  // While the current row/column is not in diagonal form.
  while not $\{\Sigma[:, f]\} \cup \{\Sigma[f, :]\} \setminus \{\Sigma[f, f]\} = \{0\}$ **do**
    /* Find the minimal element (in absolute value) of bottom-right sub-matrix and permute it to the pivot position. */
    find $(s, t) = \mathrm{argmin}_{(i,j) \geq f} |\Sigma[i, j]|$ // Find minimal element.
    $\Sigma := P(s, f)\Sigma$; $U := P(s, f)U$ // Permute rows and columns
    $\Sigma := \Sigma P(t, f)$; $V := VP(t, f)$ // to switch $\Sigma[f, f]$ and $\Sigma[s, t]$.
    if $\Sigma[f, f] < 0$ **then**
      $R := I_n$; $I_n(f, f) := -1$ // If the pivot is negative,
      $\Sigma := \Sigma R$; $V := VR$ // make it positive.
    end
    /* Reduce row and column entries using ρ_1 so that every element in the row/col is smaller than the pivot. */
    for k **in** $[f + 1, \ldots, n]$ **do**
      $R := \rho_1(\Sigma[f, k], \Sigma[f, f], k, f, n)$ // Reduce entries in row
      $\Sigma := \Sigma R$, $V := VR$ // using ρ_1, apply to V too.
    end
    for k **in** $[f + 1, \ldots, m]$ **do**
      $L := \rho_1(\Sigma[k, f], \Sigma[f, f], f, k, m)$ // Reduce entries in col
      $\Sigma := L\Sigma$; $U := LU$ // using ρ_1, apply to U too.
    end
  end
end

// STAGE 2: Enforce divisibility of diagonal entries.
for f **in** $[1, \ldots, \min(m, n)]$ **do**
  for k **in** $[f + 1, \ldots, \min(m, n)]$ **do**
    if $\Sigma[k, k] \% \Sigma[f, f] = 0$ **then**
      continue // Skip if divisible, or both entries zero.
    end
    $r := \Sigma[f, f]$; $s := \Sigma[k, k]$ // Enforce divisibility criterion
    $L := \rho_2(r, s, k, f, m)$; $R := \rho_3(r, s, f, k, n)$ // by applying
    $\Sigma := L\Sigma R$; $U := LU$; $V := VR$ // divisibility transform.
  end
end

Chapter 5

Computing factorizations in FinAb

The primary goal of this chapter is to understand and be able to compute certain factorizations of homomorphisms between FGAs, i.e. morphisms in FinAb. The categories VectR, ModZ and FinAb are abelian categories, in which every morphism has a kernel, cokernel, image and coimage morphism. These special morphisms, which allow us to factor an arbitrary morphism ϕ, will provide us with understanding about ϕ.

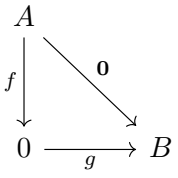
We work our way towards something which, to the extent of our knowledge, is not found in the literature: explicit algorithms for the kernel, cokernel, image and coimage morphisms in FinAb. As an intermediate step, a new algorithm for solving a particular type of equation in FinAb is also presented in Section 5.4. We start by giving general definitions, examine VectR, then ModZ, and finally give algorithms for FinAb.

5.1 Factorizations in abelian categories

Informally, an abelian category is an abstract setting in which the kernel, cokernel and image/coimage factorizations exist and exhibit nice properties. Furthermore, morphisms can be added and subtracted in abelian categories. We will now give definitions of the kernel, cokernel, image and coimage (which need not exist in a general category), and then define an abelian category more precisely. For a more thorough discussion, see Chapter 9 in [Aluffi, 2009] or Chapter 8 in [Mac Lane, 1998].

Definition 5.1 (Zero morphism). *Consider a category in which initial and final objects coincide, i.e. a category where there exists an object* 0

which is both initial and final. The zero morphism is the unique morphism $\mathbf{0} : A \to B$ *making the following diagram commute.*

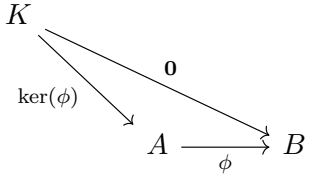


⌟

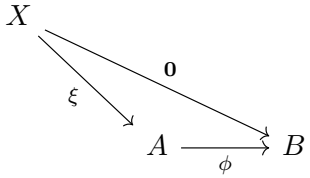
5.1.1 The kernel and cokernel

Definition 5.2 (Kernel morphism). *Consider a morphism* $\phi : A \to B$*. The kernel morphism, denoted* $\ker(\phi)$*, is a morphism such that:*

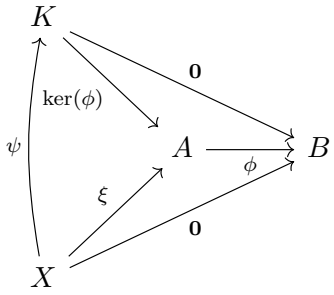
1. *The following diagram commutes.*



2. *For all*



there exists a unique ψ *such that the following diagram commutes.*



⌟

Note that the kernel morphism is not strictly the same as the kernel of a group homomorphism as per Definition 3.16. The kernel morphism is a morphism, while the kernel of a group homomorphism is a subgroup.

Definition 5.3 (Cokernel morphism). *The cokernel is the dual of the kernel. We switch the direction of every arrow in Definition 5.2 of the kernel morphism to obtain the definition of the cokernel.* ⌟

Definition 5.4 (Abelian category). *A category is abelian if (1) it has a zero object, (2) it has finite products and finite coproducts, (3) it has kernels and cokernels, (4) every kernel is a monomorphism and every monomorphism is the kernel of some morphism and (5) every cokernel is an epimorphism and every epimorphism is the cokernel of some morphism.* ⌟

Some examples of abelian categories are VectR, ModZ, FinAb and Ab.

5.1.2 The image and coimage

Definition 5.5 (Image and coimage morphism). *The image of a morphism $\phi : A \to B$ in an abelian category is defined as*

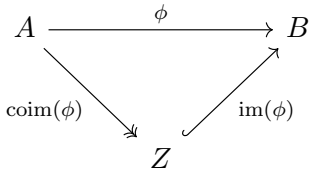
$$\mathrm{im}(\phi) = \ker\left(\mathrm{coker}(\phi)\right).$$

Dually, the coimage of a morphism $\phi : A \to B$ in an abelian category is defined as

$$\mathrm{coim}(\phi) = \mathrm{coker}\left(\ker(\phi)\right).$$

⌟

Proposition 5.6 (Factoring with image/coimage). *In an abelian category, every morphism $\phi : A \to B$ may be factored as* $\mathrm{im}(\phi) \circ \mathrm{coim}(\phi)$.



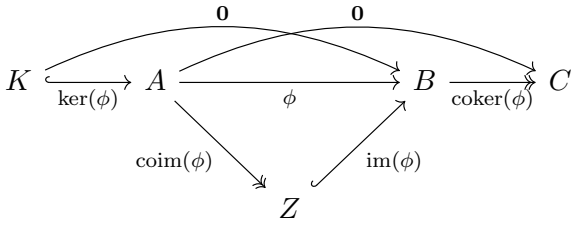
The object Z in $A \xrightarrow{\mathrm{coim}(\phi)} Z \xrightarrow{\mathrm{im}(\phi)} B$ is unique up to isomorphism.

Proof. A detailed proof is given in Chapter 9 in [Aluffi, 2009]. □

5.1.3 A summary of factorizations

We summarize the factorizations available in an abelian category, they are: $\phi \circ \ker(\phi) = \mathbf{0}$, $\mathrm{coker}(\phi) \circ \phi = \mathbf{0}$ and $\phi = \mathrm{im}(\phi) \circ \mathrm{coim}(\phi)$. The diagram below

shows a morphism ϕ along with the factorizations. In the next section, we will see how these factorizations reveal the structure of the morphism ϕ.

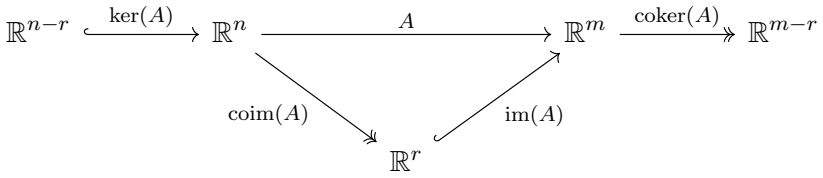


(5.1)

5.2 Factorizations in VectR

The four fundamental subspaces of a matrix $A \in \mathbb{R}^{m \times n}$ are introduced in Chapter 2 in [Strang, 1976] as the nullspace of A (kernel), the nullspace of A^T (cokernel), the column space (image) and the row space (coimage). In [Strang, 1993] it is explained how the singular value decomposition yields orthogonal bases for these subspaces, i.e. how the morphisms are found computationally.

In VectR, Diagram (5.1) looks like the following (zero morphisms omitted).



(5.2)

5.2.1 Concrete interpretations

We now give some intuition as to what these morphisms tell us about ϕ. These interpretations are for VectR, but the same underlying principles apply in other abelian categories.

- **The kernel monomorphism**. The kernel monomorphism is a basis for the nullspace, i.e. a basis for the space of solutions to $Ax = 0$. The dimensionality of the source of the kernel, $\mathbb{R}^{n-r}$, measures how far A is from being a monomorphism—if the kernel source has dimensionality $\mathbb{R}^0$, then $n = r$ and A is a monomorphism.
- **The cokernel epimorphism**. Dual to the kernel, the target of the cokernel epimorphism measures how far A is from being an epimorphism—

if the cokernel target has dimensionality $\mathbb{R}^0$, then $m = r$ and A is an epimorphism.

- **The image monomorphism**. The morphism $\mathrm{im}(A)$ is the injective part of A. The image monomorphism gives a basis for the solution space (or column space, or range) of A. It is in a sense the "most economical" expression of A—the column space is unchanged, but all "redundant" information is removed.
- **The coimage epimorphism**. Dually to the image morphism, the coimage is the surjective part of A. The coimage projects onto the source of the image such that $\mathrm{coim}(A) \circ \mathrm{im}(A) = A$.

5.2.2 Computing the factorizations

We now mention two methods for computing the the kernel, cokernel, image and coimage. Issues such as numerical stability and computational complexity are not considered. For a discussion on elementary operations and the reduced echelon form of real matrices, see Chapter 4 in [Nicholas. Loehr, 2014]. For information about the singular value decomposition and how to efficiently compute it, see Chapter 8 in [Golub and Van Loan, 2012].

Using the reduced column echelon form

Imitating Algorithm 1 for the HNF, but allowing elementary operations for real matrices instead of the unimodular ones, we can write $AE = R$, where $A \in \mathbb{R}^{m\times n}$, $E \in \mathbb{R}^{n\times n}$ and $R \in \mathbb{R}^{m\times n}$. The invertible matrix E stores elementary column operations. The matrix R is in reduced column echelon form. Splitting the decomposition into blocks we obtain

$$\underset{m\times n}{A}\begin{pmatrix} \underset{n\times r}{E_1} & \underset{n\times(n-r)}{E_2}\end{pmatrix} = \begin{pmatrix} \underset{m\times r}{R_1} & \underset{m\times(n-r)}{0}\end{pmatrix} \quad \Leftrightarrow$$

$$\underset{m\times n}{A} = \begin{pmatrix} \underset{m\times r}{R_1} & \underset{m\times(n-r)}{0}\end{pmatrix} \begin{pmatrix} \underset{r\times n}{E_1^{-1}} \\ \underset{(n-r)\times n}{E_2^{-1}}\end{pmatrix} = \underset{m\times n}{R_1 E_1^{-1}} + \underset{m\times n}{0E_2^{-1}}.$$

From these factorizations we obtain $\mathrm{im}(A) = R_1 \in \mathbb{R}^{m\times r}$, $\mathrm{coim}(A) = E_1^{-1} \in \mathbb{R}^{r\times n}$, $\ker(A) = E_2 \in \mathbb{R}^{n\times(n-r)}$. The cokernel can be obtained as $\mathrm{coker}(A) = \ker(A^T)^T$, by virtue of the fundamental theorem of linear algebra, see [Strang, 1993]. Using this procedure, the reduced column echelon form is computed twice, once for A and once for A^T. The resulting morphisms are in general not orthogonal.

Using the singular value decomposition

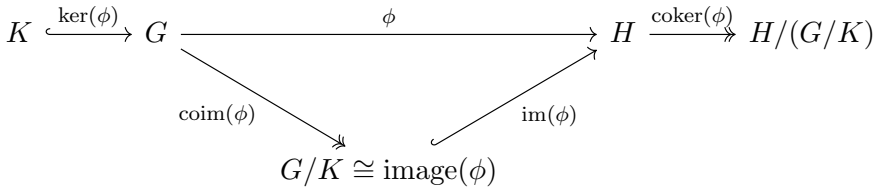
In [Golub and Van Loan, 2012], the authors state that "nothing takes apart a matrix as conclusively as the SVD." Indeed, the singular value decomposition relates the kernel, cokernel, image and coimage to one decomposition of $A \in \mathbb{R}^{m \times n}$, and the column spaces of the resulting morphisms are orthogonal. This is done explicitly in [Munthe-Kaas, 2016], and we do not repeat it here.

5.3 Factoring free-to-free morphisms in FinAb

A free-to-free morphism between FGAs is a morphism in which both the source and target are free, i.e. a morphism of the form $A : \mathbb{Z}^n \to \mathbb{Z}^m$ where A is an integer matrix. This was discussed in the previous chapter, and the situation coincides with the category ModZ of modules over $\mathbb{Z}$, where a module over a ring (such as $\mathbb{Z}$) is similar to a vector space over a field (such as $\mathbb{R}$). For an abstract introduction to modules and vector spaces, see for instance chapters 10 and 11 in [David Steven. Dummit, 2004].

5.3.1 Structure of the groups

In FinAb, the cokernel morphism is a projection onto a quotient. For $\phi : G \to H$, the coimage morphism projects onto G/K, where K is the kernel of the homomorphism, i.e. a subgroup of G. Furthermore, the cokernel projects onto $H/\operatorname{image}(\phi)$, where $\operatorname{image}(\phi)$ is the source of the image morphism.



The statement that $G/K \cong \operatorname{image}(\phi)$ for $\phi : G \to H$ is called the first isomorphism theorem. More information about it may be found in Chapter 3 of [Ledermann, 1996]. See Chapter 3 of [Mac Lane, 1998] for the statement about the target of the cokernel morphism.

5.3.2 The Hermite normal form

We demonstrate how to find the kernel, coimage and image using the HNF. First we employ Theorem 4.8 to write $AU = H$, where $U \in \mathrm{GL}(n, \mathbb{Z})$ and

$H \in \mathbb{Z}^{m,n}$ is in HNF. Mimicking the reduced column echelon form on page 41, we write

$$\underset{m\times n}{A}\begin{pmatrix} U_1 & U_2 \\ {\scriptstyle n\times r} & {\scriptstyle n\times(n-r)}\end{pmatrix} = \begin{pmatrix} H_1 & 0 \\ {\scriptstyle m\times r} & {\scriptstyle m\times(n-r)}\end{pmatrix} \quad \Leftrightarrow$$

$$\underset{m\times n}{A} = \begin{pmatrix} H_1 & 0 \\ {\scriptstyle m\times r} & {\scriptstyle m\times(n-r)}\end{pmatrix}\begin{pmatrix} U_1^{-1} \\ {\scriptstyle r\times n} \\ U_2^{-1} \\ {\scriptstyle (n-r)\times n}\end{pmatrix} = \underset{m\times n}{H_1 U_1^{-1}} + \underset{m\times n}{0U_2^{-1}}.$$

We obtain $\mathrm{im}(A) = H_1 \in \hom(\mathbb{Z}^r, \mathbb{Z}^m)$, $\mathrm{coim}(A) = U_1^{-1} \in \hom(\mathbb{Z}^n, \mathbb{Z}^r)$ and $\ker(A) = U_2 \in \hom(\mathbb{Z}^{n-r}, \mathbb{Z}^n)$, as is done in [Munthe-Kaas, 2016].

5.3.3 The Smith normal form

Using the SNF, we can compute every morphism and factorization. We use Theorem 4.11 to write $UAV = \Sigma$, where $U \in \mathrm{GL}(m, \mathbb{Z})$, $V \in \mathrm{GL}(n, \mathbb{Z})$ and $\Sigma \in \mathbb{Z}^{m\times n}$ is diagonal. We block up the matrices as

$$\begin{pmatrix} U_1 \\ {\scriptstyle r\times m} \\ U_2 \\ {\scriptstyle (m-r)\times m}\end{pmatrix}\underset{m\times n}{A}\begin{pmatrix} V_1 & V_2 \\ {\scriptstyle n\times r} & {\scriptstyle n\times(n-r)}\end{pmatrix} = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ {\scriptstyle r\times r} & {\scriptstyle r\times(n-r)} \\ \Sigma_{21} & \Sigma_{22} \\ {\scriptstyle (m-r)\times r} & {\scriptstyle (m-r)\times(n-r)}\end{pmatrix},$$

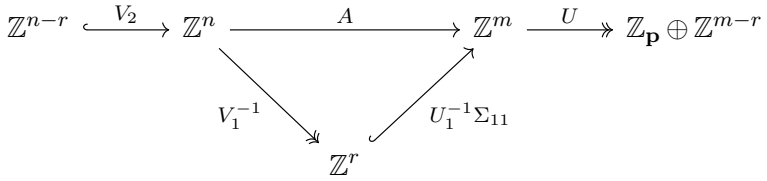
or alternatively

$$\underset{m\times n}{A} = \begin{pmatrix} U_1^{-1} & U_2^{-1} \\ {\scriptstyle m\times r} & {\scriptstyle m\times(m-r)}\end{pmatrix}\begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ {\scriptstyle r\times r} & {\scriptstyle r\times(n-r)} \\ \Sigma_{21} & \Sigma_{22} \\ {\scriptstyle (m-r)\times r} & {\scriptstyle (m-r)\times(n-r)}\end{pmatrix}\begin{pmatrix} V_1^{-1} \\ {\scriptstyle r\times n} \\ V_2^{-1} \\ {\scriptstyle (n-r)\times n}\end{pmatrix},$$

where r is the number of non-zero entries in Σ_{11}, and Σ_{12}, Σ_{21} and Σ_{22} are all identically zero. The morphisms are then:

- $\ker(A) = V_2 \in \hom(\mathbb{Z}^{n-r}, \mathbb{Z}^n)$
- $\mathrm{coker}(A) = U \in \hom(\mathbb{Z}^m, \mathbb{Z}_{\mathbf{p}} \oplus \mathbb{Z}^{m-r})$, where $p = \mathrm{diag}(\Sigma_{11})$
- $\mathrm{im}(A) = U_1^{-1}\Sigma_{11} \in \hom(\mathbb{Z}^r, \mathbb{Z}^m)$
- $\mathrm{coim}(A) = V_1^{-1} \in \hom(\mathbb{Z}^n, \mathbb{Z}^r)$

In summary, Diagram (5.1) looks like the following in ModZ.



(5.3)

For more information, see [Munthe-Kaas, 2016].

5.4 Solving equations in FinAb

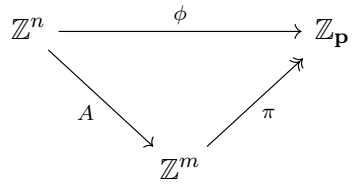
In Section 5.5 we will need the solution to a particular equation. We solve this problem here so as not to interrupt the flow of ideas later. The algorithm presented in this section is my own.

Definition 5.7 (Left-free morphism). *A morphism ϕ in the category* FinAb *is left free if it can be written as $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$, where $p = (p_1, p_2, \ldots, p_m)$ is a vector (or multi-index). If $p_i = 0$, then the i'th term in the direct sum is taken to be $\mathbb{Z}$. If $p_i \geq 1$, then the i'th term is $\mathbb{Z}_{p_i}$.* ⌟

We remind the reader that we will only typeset the p in $\mathbb{Z}_{\mathbf{p}}$ using boldface when it is a subscript, to remind ourselves that p is not an integer.

Consider the problem of solving $\phi(x) = g$ for x when $\phi : \mathbb{Z}^m \to \mathbb{Z}_{\mathbf{p}}$. The homomorphism consists of matrix multiplication with A, followed by a canonical projection π onto $\mathbb{Z}_{\mathbf{p}}$, which mods each component in the group element with the corresponding group order in the direct sum $\mathbb{Z}_{\mathbf{p}}$.

Problem 5.8. *Given a left free morphism $\phi : \mathbb{Z}^m \to \mathbb{Z}_{\mathbf{p}}$, where $\phi(x) = \pi(A(x))$ and $\pi(x) = x \mod p$, solve $\phi(x) = g$ for x if possible.*



(5.4)

This problem is a generalization of the integer equation $ax = g \mod p$, and can can have zero, one, or an infinite number of integer solutions. A single solution is possible if p is zero, in which case the mod operation is the identity function.

Proposition 5.9 (Solving $\phi(x) = g$ when ϕ is left-free). *Algorithm 3 solves Problem 5.8. If no solution exists, the algorithms reveals it. If several solutions exists, one of them is returned.*

Explanation of Algorithm 3. The explanation is split into three parts. First we will find a left-inverse of π. Then we form a block-matrix $(A \mid \ker(\pi) \mid g)$ and realize that the solution must be in its kernel. Finally we iteratively

Algorithm 3: Solving $\phi(x) = g$ when ϕ is left-free.

Input : Matrix $A \in \mathbb{Z}^{m \times n}$, group element $g \in \mathbb{Z}_{\mathbf{p}}$ and vector $p \in \mathbb{Z}^m$.
Output : Group element $x \in \mathbb{Z}^n$ such that $\phi(x) = g$.

```
1  ker π := remove_zero_cols(diag(p))
2  K := free_kernel(A | ker π | g)
3  Remove columns from K where the last column entry is zero.
4  while K has more than 1 column do
5  |  r := K[−1, 1]      // Get the last column entry in the first column.
6  |  s := K[−1, 2]      // Get the last column entry in the second column.
7  |  Compute a, b such that ar + bs = gcd(r, s)
8  |  new_col := aK[:, 1] + bK[:, 2]  // Linear combination of cols 1 and 2.
9  |  Replace columns K[:, 1] and K[:, 2] with new_col in the matrix K.
10 end
11 x := −K[: n, 1]      // Get the first n entries of K, multiply by −1.
```

employ the extended Euclidean algorithm to find a solution.

$$\begin{array}{ccccc} \mathbb{Z}^n & & \xrightarrow{\quad\phi\quad} & & \mathbb{Z}_{\mathbf{p}} \\ & \searrow^{A} & & \nearrow_{\pi} & \\ \mathbb{Z}^k & \xrightarrow[\ker(\pi)]{} & \mathbb{Z}^m & \xleftarrow{\quad\pi^{-1}\quad} & \end{array} \tag{5.5}$$

We find a left-inverse of the canonical projection $\pi : \mathbb{Z}^m \to \mathbb{Z}_{\mathbf{p}}$. Let k be the number of non-zero entries in p. The kernel of the canonical projection π is a diagonal matrix $\mathrm{diag}(p_1, p_2, \ldots, p_m)$ with a modification; zero columns are removed to assure that $\ker(\pi) : \mathbb{Z}^k \to \mathbb{Z}^m$ is a monomorphism. Clearly $\pi(x + \ker(\pi)\xi) = \pi(x)$ for every $\xi \in \mathbb{Z}^k$, and thus $\pi^{-1}(x) = x + \ker(\pi)\xi$ is a left-inverse of π such that $\pi(\pi^{-1}(g)) = g$ for every $g \in \mathbb{Z}_{\mathbf{p}}$.

We want to solve $\pi(A(x)) = g$. If a solution exists, then $\pi(g + \ker(\pi)\xi) = g$ for some $\xi \in \mathbb{Z}^k$. Comparing these two equation we observe that $Ax = g + \ker(\pi)\xi$, which we write as $Ax - \ker(\pi)\xi - g = 0$, where every term is in $\mathbb{Z}^m$. We factor this as

$$(A \mid \ker(\pi) \mid g)(x; -\xi; -1) = 0, \tag{5.6}$$

where the semicolons denote row separation in a column vector. This is a mapping from $\mathbb{Z}^n$ and $\mathbb{Z}^k$ to $\mathbb{Z}^m$, where the result is added/subtracted. All these groups are free, so we can compute the kernel using the SNF as previously explained using Diagram (5.3). Writing the kernel factorization of $(A \mid \ker(\pi) \mid g)$ as

$$(A \mid \ker(\pi) \mid g)\ker(A \mid \ker(\pi) \mid g)\mu = 0$$

and comparing with Equation (5.6) we realize that

$$\ker\left(A \mid \ker(\pi) \mid g\right)\mu = \begin{pmatrix} x \\ -\xi \\ -1 \end{pmatrix}. \tag{5.7}$$

To find a solution x, we search for an integer combination of the columns of $K := \ker(A \mid \ker(\pi) \mid g)$ such that the resulting bottom column entry is -1.

To find such a combination, we multiply both sides of Equation (5.7) by -1 and instead look for a combination which gives 1 in the bottom column entry. It is only possible to write 1 as a linear combination of two integers if their greatest common divisor is 1. This suggests an approach for finding a linear combination of the columns of $K := \ker(A \mid \ker(\pi) \mid g)$ using the extended Euclidean algorithm.

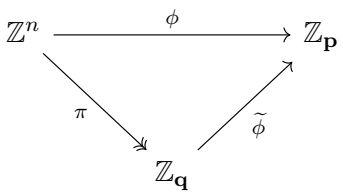
We use the extended Euclidean algorithm repeatedly as shown in line 4 in Algorithm 3 to take linear combinations of the columns until only one column remains. If we are able to find such a column, we multiply by -1 in line 11 and return the first n entries corresponding to x in the column vector in Equation (5.7). An x which solves the equation is returned if it exists. □

5.5 Factoring left-free morphisms in FinAb

We have seen how to compute factorizations of morphisms in VectR and free-to-free morphisms in FinAb. The time has come to consider left-free morphisms in FinAb, i.e. morphisms of the form $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$.

Such a homomorphism is represented by a matrix $A \in \mathbb{Z}^{m\times n}$, where we think of each column as a generator. Calculating $\phi(x) = Ax \mod p$ takes linear combinations of the generator columns in A, and projects onto $\mathbb{Z}_{\mathbf{p}}$. There are two projections that can be performed on ϕ without spoiling the homomorphism property $\phi(x+y) = \phi(x)+\phi(y)$, namely: (1) projecting the source group to the orders of the generators and (2) projecting the generators to the target group.

Definition 5.10 (Canonical projection to source). *Given $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ represented by a matrix $A \in \mathbb{Z}^{m\times n}$, there exists a projection $\pi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{q}}$ such that $\phi = \widetilde{\phi} \circ \pi$.*



The group $\mathbb{Z}_{\mathbf{q}}$ contains the orders of the columns of A, i.e. the i'th component of q in $\mathbb{Z}_{\mathbf{q}}$ is the order of the i'th column of A. ⌟

We can compute $\mathbb{Z}_{\mathbf{q}}$ efficiently using Proposition 3.28, which explains how to compute the order of a group element in an FGA. The proposition is used on every column to compute the structure of $\mathbb{Z}_{\mathbf{q}}$.

Definition 5.11 (Canonical projection to target). *Consider $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ represented by a matrix $A = (a_1, a_2, \ldots, a_m) \in \mathbb{Z}^{m \times n}$ where a_i are columns for $i = 1, 2, \ldots, m$. The canonical projection to the target is to set $a_i := a_i \mod p$ for $i = 1, 2, \ldots, m$.* ⌟

The target projection projects each generator to the target group.

Example 5.12 (Projections to source and target). We demonstrate projections to source and target as given by Definitions 5.10 and 5.11. Consider the left-free morphism

$$\begin{pmatrix} 2 & 6 \\ 3 & 1 \end{pmatrix} \in \hom(\mathbb{Z}^2, \mathbb{Z}_4 \oplus \mathbb{Z}_3).$$

Projecting to the target $\mathbb{Z}_4 \oplus \mathbb{Z}_3$ yields the morphism

$$\begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix} \in \hom(\mathbb{Z}^2, \mathbb{Z}_4 \oplus \mathbb{Z}_3),$$

where each column is projected onto the target group. Projecting this morphism to the source reveals the orders of the columns, we obtain

$$\begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix} \in \hom(\mathbb{Z}_2 \oplus \mathbb{Z}_6, \mathbb{Z}_4 \oplus \mathbb{Z}_3).$$

In general such a projection to source does not find the image morphism, as the projection is not guaranteed to yield a monomorphism. ⌟

5.5.1 The kernel

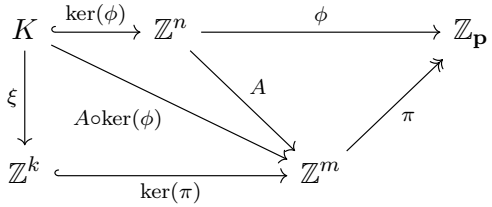
The following theorem explicitly states how to compute the kernel morphism.

Theorem 5.13 (Computing the kernel in FinAb). *The kernel of a left-free morphism $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ is the first n rows of*

$$\begin{pmatrix} \ker(\phi) \\ \xi \end{pmatrix} = \ker\left(A \mid \ker(\pi)\right),$$

where $\ker(\pi)$ *is computed as* remove_zero_cols(diag(π)) *and the outer kernel is computed as a free-to-free kernel.*

Proof.



(5.8)

We want to find $\ker(\phi)$ such that $\phi \circ \ker(\phi) = \pi \circ A \circ \ker(\phi) = \mathbf{0}$, where $\mathbf{0}$ is the zero morphism. Since $\pi \circ \ker(\pi) = \mathbf{0}$, the morphism $A \circ \ker(\phi)$ factors through $\ker(\pi)$ such that $A \circ \ker(\phi) = \ker(\pi) \circ \xi$ for some ξ. This is due to the universal property of kernels, as per Definition 5.2. The situation is depicted in Diagram (5.8), which commutes.

We factor $A \circ \ker(\phi) = \ker(\pi) \circ \xi$ as

$$(A \mid \ker(\pi)) \begin{pmatrix} \ker(\phi) \\ \xi \end{pmatrix} = \mathbf{0},$$

which we compare with the kernel

$$(A \mid \ker(\pi)) \ker (A \mid \ker(\pi)) = \mathbf{0}.$$

Again we appeal to the universal property of kernels to write

$$(\ker(\phi); \xi) = \ker (A \mid \ker(\pi))\, \psi.$$

We take ψ to be the identity morphism, since it must be a bijection: if ψ is not injective, $\ker(\phi)$ would not be injective. If ψ is not surjective, then it would remove elements in the kernel of $(A \mid \ker(\pi))$ from the kernel of ϕ. We want every element, since there is no element in the kernel of $(A \mid \ker(\pi))$ which is not in the kernel of ϕ. The morphism ψ must therefore be bijective, and we take it to be the identity. The kernel of ϕ is then given by $(\ker(\phi); \xi) = \ker (A \mid \ker(\pi))$, which is what we wanted to prove. □

5.5.2 The cokernel

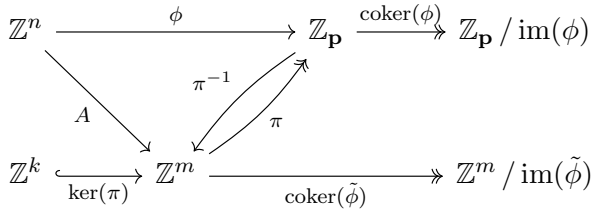
We now demonstrate how to compute the cokernel morphism.

Theorem 5.14 (Computing the cokernel in FinAb). *The cokernel epimorphism of a left-free morphism $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ may be computed as*

$$\operatorname{coker}(\phi) = \operatorname{coker} (A \mid \ker(\pi)),$$

where $\ker(\pi)$ *is computed as* remove_zero_cols(diag(π)) *and the outer cokernel is computed as a free-to-free cokernel. The target group of the cokernel epimorphism can be computed using the SNF of* $(A \mid \ker(\pi))$.

Proof.



(5.9)

This proof is in two parts. We will first show that the quotient group $\mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$, i.e. the target of $\operatorname{coker}(\phi)$, has the same structure as the target of the cokernel of a map $\tilde{\phi}$ "lifted" to $\mathbb{Z}^m$. Then we show that the mappings $\operatorname{coker}(\phi)$ and $\operatorname{coker}(\tilde{\phi})$ are also the same. Diagram (5.9) gives an overview.

First notice that π is an epimorphism, and has a left-inverse π^{-1} such that $\pi\left(\pi^{-1}(x)\right) = x$ for every $x \in \mathbb{Z}_{\mathbf{p}}$. The left-inverse is $\pi^{-1}(x) = x + \ker(\pi)\xi$, with $\xi \in \mathbb{Z}^k$. We "lift" the mapping from $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ to $\tilde{\phi} : \mathbb{Z}^n \to \mathbb{Z}^m$, by introducing $\tilde{\phi} = \pi^{-1} \circ \phi = Ax + \ker(\pi)\xi$ as shown in Figure 5.1. The structures of the quotient groups $\mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$ and $\mathbb{Z}^m / \operatorname{im}(\tilde{\phi})$ are identical. Therefore we may use $\tilde{\phi}$, which is free-to-free, to compute the structure of $\mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$. To do this, we write $\tilde{\phi}$ as $(A \mid \ker(\pi))$ and compute the target of the cokernel using the SNF as explained in Section 5.3.3.

Thus we can compute $\operatorname{coker}(\tilde{\phi})$, and this morphism is in fact identical to $\operatorname{coker}(\phi)$. To see why, recall that $\operatorname{coker}(\phi) = \operatorname{coker}(\tilde{\phi}) \circ \pi^{-1}$, and that

$$\operatorname{coker}(\tilde{\phi}) \circ \pi^{-1}(x) = \operatorname{coker}(\tilde{\phi})(x + \ker(\pi)\xi).$$

Therefore $\operatorname{coker}(\phi)$ equals $\operatorname{coker}(\tilde{\phi})$ if ξ can be taken as zero. No finite group in $\mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$ is of larger order than any finite group in $\mathbb{Z}_{\mathbf{p}}$, so changing ξ in $\operatorname{coker}(\tilde{\phi})(x + \ker(\pi)\xi)$ does not change the mapping $\operatorname{coker}(\tilde{\phi}) \circ \pi^{-1}(x)$—the result is projected to $\mathbb{Z}^m / \operatorname{im}(\tilde{\phi}) \cong \mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$ anyway, rendering the choice of ξ irrelevant.

Both the structure of $\mathbb{Z}_{\mathbf{p}} / \operatorname{im}(\phi)$ and the morphism $\operatorname{coker}(\phi)$ can be computed by using the free-to-free cokernel of $(A \mid \ker(\pi))$. In other words, $\operatorname{coker}(\phi) = \operatorname{coker}(A \mid \ker(\pi))$ as claimed. □

5.5.3 The coimage

Theorem 5.15 (Computing the coimage in FinAb). *The coimage epimorphism of a left-free morphism $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$ may be computed as*

$$\operatorname{coim}(\phi) = \operatorname{coker}(\ker(\phi)),$$

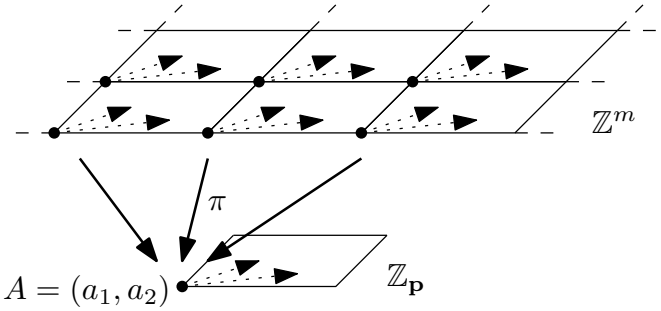


Figure 5.1: The columns of A generate a subgroup of $\mathbb{Z}_{\mathbf{p}}$. The mapping $\phi = \pi \circ A$ can be lifted to $\mathbb{Z}^m$, where the lifted map is $\tilde{\phi} = Ax + \ker(\pi)\xi$. Intuitively, this corresponds to "ignoring" the canonical projection π.

where the inner kernel computation is left-free and the outer cokernel computation is free-to-free.

Proof.

$$\begin{array}{ccccc} \mathbb{Z}^k & \xhookrightarrow{\ker(\phi)} & \mathbb{Z}^n & \xrightarrow{\quad\phi\quad} & \mathbb{Z}_{\mathbf{p}} \\ & & \searrow^{\operatorname{coim}(\phi)} & & \nearrow_{\operatorname{im}(\phi)} \\ & & & \mathbb{Z}^n \,/\, \mathbb{Z}^k & \end{array} \tag{5.10}$$

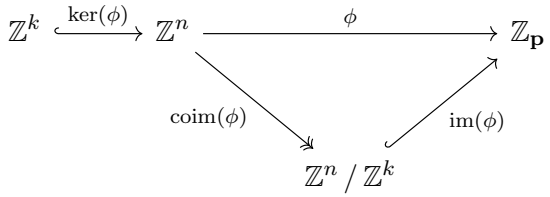
We use Definition 5.5 of the coimage epimorphism as it was stated. The argument of the inner kernel is a left-free morphism, which we already know how to compute. The monomorphism $\ker(\phi)$ is free-to-free, and we know how to compute the cokernel of free-to-free morphisms from Diagram (5.3). □

5.5.4 The image

Computing the image morphism requires more cleverness than the coimage computation. The definition is $\operatorname{im}(\phi) = \ker(\operatorname{coker}(\phi))$, but we cannot use this directly since $\operatorname{coker}(\phi)$ is not free-to-free in general. The uniqueness of the coimage/image factorization of ϕ suggests an alternative route.

Theorem 5.16 (Computing the image in FinAb). *Consider the diagram below, and assume that we know the structure of every group and every*

morphism except the image morphism.



(5.11)

The image can be computed by going through each canonical generator of $\mathbb{Z}^n \,/\, \mathbb{Z}^k$ *and computing* $\phi \circ \operatorname{coim}(\phi)^{-1}$.

Proof. Let $\operatorname{im}(\phi)$ be an unknown morphism and let $\operatorname{im}(\phi) \circ \operatorname{coim}(\phi) = \phi$. We solve this equation for $\operatorname{im}(\phi)$. Algorithm 4 iterates through every canonical generator $e_1, e_2, e_3, \dots$ in the quotient space $\mathbb{Z}^n \,/\, \mathbb{Z}^k$, solves $\operatorname{coim}(\phi)^{-1}(e_i)$ using Algorithm 3 and applies ϕ to the result. Since the factorization always exists and since Algorithm 3 solves the equation, the algorithm will return the image morphism. □

Algorithm 4: Solving for $\operatorname{im}(\phi)$

for *every* $e_i \in \text{canonical\_generators}\left(\mathbb{Z}^n \,/\, \mathbb{Z}^k\right)$ **do**
  $\operatorname{im}(\phi)_i = \phi\left(\operatorname{coim}(\phi)^{-1}(e_i)\right)$ `// Solve equation, then apply` ϕ.
end
$\operatorname{im}(\phi) := (\operatorname{im}(\phi)_1, \operatorname{im}(\phi)_2, \dots)$ `// Concatenate the individual results.`
return $\operatorname{im}(\phi)$

We summarize this section with a concrete example.

Example 5.17 (Factoring a left-free morphism). Consider $\phi : \mathbb{Z}^2 \to \mathbb{Z}_8 \oplus \mathbb{Z}_5$ given by the matrix $A = \left(\begin{smallmatrix} 4 & 2 \\ 7 & 3 \end{smallmatrix}\right)$. This is a continuation of Example 3.29 on page 19, where we looked at the same set of generators. Figure 3.1 on page 20 provides a visualization of this group. The projection to source gives $\widetilde{\phi} \in \hom\left(\mathbb{Z}_{10} \oplus \mathbb{Z}_{20}, \mathbb{Z}_8 \oplus \mathbb{Z}_5\right)$, revealing the orders of the columns to be 10 and 20.

We form the matrix $(A|\ker(\pi)) = \left(\begin{smallmatrix} 4 & 2 & 8 & 0 \\ 7 & 3 & 0 & 5 \end{smallmatrix}\right)$, which has a SNF

$$\underbrace{\begin{pmatrix} -1 & 1 \\ 3 & -2 \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} 4 & 2 & 8 & 0 \\ 7 & 3 & 0 & 5 \end{pmatrix}}_{A|\ker(\pi)} \underbrace{\begin{pmatrix} 0 & -1 & 12 & -5 \\ 1 & 3 & -28 & 10 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{V} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix}}_{\Sigma}, \tag{5.12}$$

and the inverse matrices are given by

$$U^{-1} = \begin{pmatrix} 2 & 1 \\ 3 & 1 \end{pmatrix} \quad \text{and} \quad V^{-1} = \begin{pmatrix} 3 & 1 & -8 & 5 \\ -1 & 0 & 12 & -5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

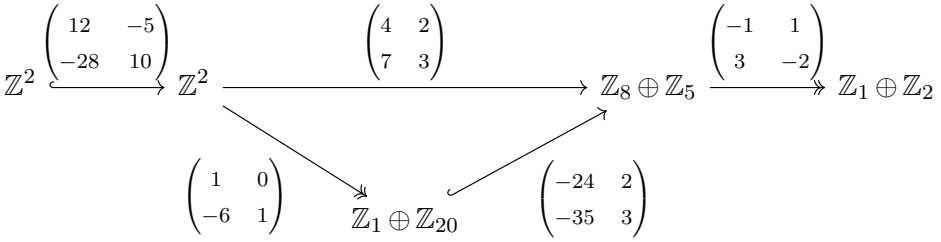
The kernel of ϕ consists of the first $m = 2$ rows of the kernel of $(A|\ker(\pi))$, i.e. V_2 in Diagram (5.3). The kernel of $(A|\ker(\pi))$ consists of the last $n - r = 4 - 2$ columns of V. So $\ker(\phi)$ consist of the first 2 rows in the last 2 columns of V, such that $\ker(\phi) = \left(\begin{smallmatrix} 12 & -5 \\ -28 & 10 \end{smallmatrix}\right) \in \hom\left(\mathbb{Z}^2, \mathbb{Z}^2\right)$.

The cokernel is $\operatorname{coker}(A|\ker(\pi))$, which we know from the SNF to be U in Equation (5.12). We have $\operatorname{coker}(\phi) = \left(\begin{smallmatrix} -1 & 1 \\ 3 & -2 \end{smallmatrix}\right) \in \hom\left(\mathbb{Z}_8 \oplus \mathbb{Z}_5, \mathbb{Z}_1 \oplus \mathbb{Z}_2\right)$.

The coimage is the cokernel of the kernel. Since $\ker(\phi) = \left(\begin{smallmatrix} 12 & -5 \\ -28 & 10 \end{smallmatrix}\right)$, we use the SNF factorization of this matrix to obtain $\left(\begin{smallmatrix} 1 & 0 \\ -6 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} 12 & -5 \\ -28 & 10 \end{smallmatrix}\right)\left(\begin{smallmatrix} -2 & -5 \\ -5 & -12 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 20 \end{smallmatrix}\right)$. The cokernel is U in the SNF, thus the coimage, or cokernel of the kernel, is $U = \left(\begin{smallmatrix} 1 & 0 \\ -6 & 1 \end{smallmatrix}\right) \in \hom\left(\mathbb{Z}^2, \mathbb{Z}_1 \oplus \mathbb{Z}_{20}\right)$.

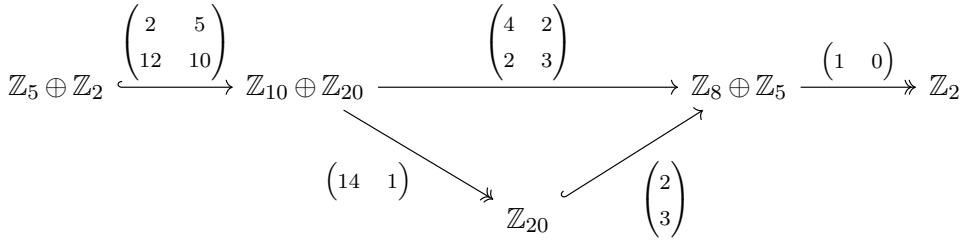
The image is solved for using Algorithm 4. Solving $\left(\begin{smallmatrix} 1 & 0 \\ -6 & 1 \end{smallmatrix}\right)x = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$ yields $x = \left(\begin{smallmatrix} 1 \\ -14 \end{smallmatrix}\right)$. We compute $Ax = \left(\begin{smallmatrix} 4 & 2 \\ 7 & 3 \end{smallmatrix}\right)\left(\begin{smallmatrix} 1 \\ -14 \end{smallmatrix}\right) = \left(\begin{smallmatrix} -24 \\ -35 \end{smallmatrix}\right)$. This is the first column generator in the image. We now solve $\left(\begin{smallmatrix} 1 & 0 \\ -6 & 1 \end{smallmatrix}\right)x = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$ for x, which yields $x = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$. Then $Ax = \left(\begin{smallmatrix} 4 & 2 \\ 7 & 3 \end{smallmatrix}\right)\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 2 \\ 3 \end{smallmatrix}\right)$, and we obtain $\operatorname{im}(\phi) = \left(\begin{smallmatrix} -24 & 2 \\ -35 & 3 \end{smallmatrix}\right) \in \hom\left(\mathbb{Z}_1 \oplus \mathbb{Z}_{20}, \mathbb{Z}_8 \oplus \mathbb{Z}_5\right)$.

Based on the computations above, we obtain the following diagram.



The diagram is cluttered with trivial groups, which provide little information. To "clean" it, we remove trivial groups and the corresponding rows and columns of the target and source, respectively. After removing trivial groups, we project every morphism to source and target using Definitions 5.10 and

5.11. The result is the following diagram, which contains no trivial groups.



(5.13)

In Section 7.4 we will re-do this example using the software developed as part of this thesis. The software solution uses a mere 9 lines of code to perform all of the preceding computations. ⌟

5.6 Morphisms in Ab

In the previous section we investigated homomorphisms between FGAs in FinAb. We now turn our attention to homomorphisms between elementary LCAs ($\mathbb{R}$, T, $\mathbb{Z}$, $\mathbb{Z}_n$ and direct sums) in Ab, the category of abelian groups.

5.6.1 Homomorphisms represented by a number

Consider $\phi : H \to G$ and let $\phi(h) = \alpha h$ for some number α. We fix H and G and ask investigate which restrictions must be placed on α for ϕ to be a homomorphism. For instance, α must be an integer when G is an FGA. If it were not, then $\phi(1)$ would not map to an element in the target group G. In addition, ϕ must be a homomorphism so that $\phi(h_1 + h_2) = \phi(h_1) + \phi(h_2)$ for every $h_1, h_2 \in H$. In Table 5.1, we show what requirements α must fulfill for various sources H and targets G if $\phi(h) = \alpha h$ is to be a homomorphism.

		Source (H)			
		$\mathbb{R}$	T	$\mathbb{Z}$	$\mathbb{Z}_n$
Target (G)	$\mathbb{R}$	$\alpha \in \mathbb{R}$	–	$\alpha \in \mathbb{R}$	–
	T	$\alpha \in \mathbb{R}$	$\alpha \in \mathbb{Z}$	$\alpha \in \mathbb{R}$	$\alpha = k/n$, $k \in \mathbb{Z}$
	$\mathbb{Z}$	–	–	$\alpha \in \mathbb{Z}$	–
	$\mathbb{Z}_m$	–	–	$\alpha \in \mathbb{Z}$	$\alpha = mk/\gcd(m,n)$, $k \in \mathbb{Z}$

Table 5.1: Restrictions on α such that the function $\phi(h) = \alpha h$ is in $\hom(H, G)$, i.e. $\phi : H \to G$ is a homomorphism. A dash denotes that no such α exists.

There are two requirements: (1) every element which the canonical projection π maps to 0 in the source must map to 0 is the target, and (2) the elements $\phi(h) = \alpha h$ must be group elements in G for every $h \in H$.

Example 5.18. Consider $\phi : \mathbb{Z}_n \to T$. We must have $\phi(jn) = \alpha jn \in \mathbb{Z}$ for every $j \in \mathbb{Z}_n$ if $\phi(0) = 0$. If $\alpha = 1/n$ then $\phi(jn) = j$, which is always in $\mathbb{Z}$. Any integer multiple also works, so $\alpha = k/n$ with $k \in \mathbb{Z}$ satisfies the criteria in general. ⌟

5.6.2 Homomorphisms represented by a matrix

Consider the case when $\phi : H \to G$ is represented by $\phi(h) = Ah$, where A is an $m \times n$ matrix. To verify or disprove that A represents a valid homomorphism in accordance with Table 5.1, it is necessary to check every $A_{ij} : G_j \to H_i$, since $H_i = \sum_j A_{ij} G_j$ by matrix multiplication. If every $A_{ij} : G_j \to H_i$ represents a homomorphism for every $G_j \in G$ and $H_i \in H$, then $A : G \to H$ represents a homomorphism as a whole.

Full factorizations in terms of the kernel, cokernel, image and coimage is possible and well-understood in the following cases.

- $\phi : \mathbb{R}^n \to \mathbb{R}^m$. This corresponds to the familiar VectR case, in which we can may the SVD as briefly stated in Section 5.2.2.
- $\phi : \mathbb{Z}^n \to \mathbb{Z}^m$. This is the case when we have a free-to-free morphism in FinAb, and the SNF may be used as explained in Section 5.3.3.
- $\phi : \mathbb{Z}_{\mathbf{q}} \to \mathbb{Z}_{\mathbf{p}}$. This case is computed by first allowing the source to be free, then using the algorithms given in Section 5.5 on $\phi : \mathbb{Z}^n \to \mathbb{Z}_{\mathbf{p}}$, then projecting to source as in Definition 5.10 to compute the orders.

Chapter 6

Fourier analysis on locally compact abelian groups

In the 1930s, the Soviet mathematician Lev Pontryagin proved a duality theorem for locally compact abelian groups (LCAs). In the decades that followed, the ideas of Fourier analysis were generalized further by mathematicians such as Laurent Schwartz. He showed that the Fourier transform can operate on distributions, which are a generalization of the classical functions.

This chapter introduces Fourier analysis on the elementary LCAs $\mathbb{R}$, T, $\mathbb{Z}$ and $\mathbb{Z}_n$. We survey Fourier analysis from the group perspective, which is a relatively abstract approach. Books such as [Ramakrishnan, 1999], [Reiter, 1968], [R. E. Edwards, 1979] and [Rudin, 1967] provide this perspective. Our goal is computer representation and computation, so we will not dwell on topological aspects and questions of convergence. Much of the content of this chapter is explained in [Munthe-Kaas, 2016] in a very lucid manner.

6.1 Locally compact abelian groups

The most general setting for Fourier analysis is an LCA. We state the definition as given in Chapter 1 in [Ramakrishnan, 1999].

Definition 6.1 (Locally compact group). *A topological group which is locally compact and Hausdorff is called a locally compact group.* ⌟

A topological group is a group with a continuous binary operation and a continuous inverse. Intuitively, a topological space is locally compact if

every element has a closed compact neighborhood. A space is Hausdorff if for any two points x and y, we can find neighborhoods around x and y such that the two neighborhoods are disjoint. Every group encountered in this thesis is locally compact. We immediately refer the interested reader to [Hewitt, 1963] for more on topology, as we will not discuss it further.

In this thesis we are interested in the groups $\mathbb{R}$, T, $\mathbb{Z}$ and $\mathbb{Z}_n$. These groups and direct sums of them are called elementary LCAs.

Definition 6.2 (Elementary locally compact abelian group). *An elementary locally compact abelian group (LCA) is a group isomorphic to* $G = \mathbb{R}^a \oplus T^b \oplus \mathbb{Z}^c \oplus \mathbb{Z}_{\mathbf{p}}$, *where* $a, b, c \geq 0$ *and* $p = (p_1, p_2, \ldots, p_k)$ *with* $p_i \geq 1$ *for* $i = 1, 2, \ldots, k$. ⌟

Every LCA encountered in this thesis is elementary, but there exist LCAs which not elementary.

6.2 Characters and the dual group

In general, the Fourier representation of $f : G \to \mathbb{C}$ expresses f as a weighted linear combination of the characters of G.

Definition 6.3 (Group character). *A character of a group* G *is a continuous homomorphism* $\chi : G \to \mathbb{T}$, *where* $\mathbb{T}$ *is the group of complex numbers with absolute value* 1, *i.e.* $\mathbb{T} = \{z \in \mathbb{C} \mid |z| = 1\}$. ⌟

We now introduce the dual group of a LCA. In linear algebra, the dual space of a vector space V is the set of all linear functionals $\tau : V \to \mathbb{R}$, i.e. $V^* = \{\tau : V \to \mathbb{R} \mid \tau \text{ is linear}\}$. The situation for LCAs is analogous: the dual group of an LCA G is the set of all group homomorphisms $\chi : G \to \mathbb{T}$, i.e. $\widehat{G} = \{\chi : G \to \mathbb{T} \mid \chi \text{ is a homomorphism}\}$.

Definition 6.4 (Dual group). *The dual group of* G, *denoted* $\widehat{G}$, *is the set of all characters on* G. *These characters themselves form a group.* ⌟

The fact that the set of characters of G themselves constitute a group also has an analogue in linear algebra, where the set of linear functionals V^* is a vector space, see for instance [Roman, 2005]. The dual of an elementary LCA G is isomorphic to an elementary LCA. The dual of a discrete group is continuous and the dual of continuous group is discrete. The word compact is a generalization of the concept of finite group to topological groups. The groups T and $\mathbb{Z}_n$ are compact, and dual of a compact group

is non-compact, and vice versa. Specifically $\widehat{\mathbb{R}} \cong \mathbb{R}$, $\widehat{T} \cong \mathbb{Z}$, $\widehat{\mathbb{Z}} \cong T$ and $\widehat{\mathbb{Z}_n} \cong \mathbb{Z}_n$. Furthermore, duality "distributes" over direct sums in the sense that $\widehat{G \oplus H} = \widehat{G} \oplus \widehat{H}$. These and other important properties of duality and LCAs are reported in [Reiter, 1968].

Definition 6.5 (Dual pairing). *The dual pairing $(\cdot,\cdot)$ of G and the dual $\widehat{G}$ is a function $(\cdot,\cdot) : \widehat{G} \times G \to \mathbb{T}$. Fixing $\xi \in \widehat{G}$, the functions $(\xi,\cdot) : G \to \mathbb{T}$ are isomorphic to $\widehat{G}$, and vice versa.* ⌟

Consider $\xi \in \widehat{G}$ and $x \in G$, then the dual pairing is $(\xi, x) = \exp(2\pi i \langle \xi, x \rangle)$. The bracket $\langle \xi, x \rangle$ is defined as $\sum_i \xi_i x_i / C_i$, similar to a dot product of vectors apart from the constants C_i. If G_i is compact, then $C_i = \int_{G_i} 1 \, dx$. If G_i is non-compact, then $C_i = 1$. The constant C_i may be thought of as the volume of G_i.

Example 6.6 (Dual of $\mathbb{Z}_4$). Consider the group $\mathbb{Z}_4 = \{0, 1, 2, 3\}$. Clearly $x \to \exp(2\pi i x)$ is a character for every $x \in \mathbb{Z}_4$. This maps every element of $\mathbb{Z}_4$ to the identity 1, and is called the principal character. Every $\chi_\xi(x) = \exp(2\pi i \xi x/4)$ for $\xi \in \mathbb{Z}_4$ is a character, and under multiplication this is a group isomorphic to $\mathbb{Z}_4$. In other words, the dual group of $\mathbb{Z}_4$ is $\mathbb{Z}_4$, and the dual pairing is $(\xi, x) = \exp(2\pi i \xi x/4)$. This example generalizes to $\mathbb{Z}_n$. ⌟

The Pontryagin duality theorem states that the dual of $\widehat{G}$ is isomorphic to G, i.e. $\widehat{\widehat{G}} \cong G$ for every LCA G. As stated in [Rudin, 1967]: "Every LCA is the dual group of its dual group." Since $\widehat{G}$ is an LCA, the operation of taking the dual can be iterated and it can be shown that taking the dual twice is an isomorphism of topological groups. For proofs, see the introductory chapter of [Rudin, 1967] or Chapter 3 in [Ramakrishnan, 1999].

6.3 The invariant integral

Definition 6.7 (Translation operator). *The translation operator T_a translates a function $f : G \to \mathbb{C}$, such that $T_a(f(x)) = f(x - a)$. The a is an element in G, and $T_a : G \times \mathbb{C}^G \to \mathbb{C}^G$ where $\mathbb{C}^G$ is the set of all functions $f : G \to \mathbb{C}$.* ⌟

The translation operator is also referred to as the shift operator or delay operator. There exists an integral which is invariant under translation.

Definition 6.8 (Invariant integral). *On every LCA G there exists a unique (up to a constant) invariant integral $I : \mathbb{C}^{\mathbb{R}} \to \mathbb{C}$ such that $I(f) \geq 0$ for non-negative functions f and $I(T_a(f)) = I(f)$ for every $a \in G$.* ⌟

The invariant integrals are the usual integral and sum, shown in Table 6.1. For a readable, but more theoretical introduction to measurable functions, Lebesgue integration and other related matters, see [C. Gasquet, 1999].

Group	Compact	Discrete	Dual Group	Invariant integral	$\langle \cdot, \cdot \rangle$
$\mathbb{R}$	no	no	$\mathbb{R}$	$\int_{\mathbb{R}} dx$	ξx
T	yes	no	$\mathbb{Z}$	$\int_T dx$	ξx
$\mathbb{Z}$	no	yes	T	$\sum_{x\in\mathbb{Z}}$	ξx
$\mathbb{Z}_n$	yes	yes	$\mathbb{Z}_n$	$\sum_{x\in\mathbb{Z}_n}$	$\xi x/n$

Table 6.1: The elementary LCAs, their properties, duals, invariant integrals and brackets $\langle \cdot, \cdot \rangle$ in the dual pairing $(\cdot, \cdot) = \exp(2\pi i \langle \cdot, \cdot \rangle)$.

6.4 The Fourier transform

We now generalize the Fourier transform to an LCA G.

Definition 6.9 (Fourier transform on an LCA). *Let $f : G \to \mathbb{C}$ be a sufficiently nice function on an LCA. The Fourier transform of f, denoted $\widehat{f}$ or $\mathcal{F}(f)$, is given by*

$$\widehat{f}(\xi) = \mathcal{F}(f) = \int_G f(x)\overline{(\xi, x)}\, dx, \tag{6.1}$$

where $\int_G \cdot\, dx$ is the invariant integral and $(\xi, x) = \exp(2\pi i \langle \xi, x \rangle)$. The bar denotes complex conjugation. Reconstruction of f is given by

$$f(x) = \mathcal{F}^{-1}(\widehat{f}) = \frac{1}{C} \int_{\widehat{G}} \widehat{f}(\xi)(\xi, x)\, d\xi, \tag{6.2}$$

where C is the product of $\int_{G_i} 1\, dx_i$ for the compact groups in $G = G_1 \oplus G_2 \oplus \cdots \oplus G_n$, interpreted as the total volume of the compact groups. ⌟

If $G = G_1 \oplus G_2 \oplus \cdots \oplus G_n$, then the integrals in Equations (6.1) and (6.2) are interpreted as $\int_{G_1} \int_{G_2} \cdots \int_{G_n}$. For more on multidimensional extensions of Fourier transforms, see Chapter 9 in [Vretblad, 2003]. Table 6.2 is adapted from [Munthe-Kaas, 2016], and gives explicit formulas for the special cases when G is $\mathbb{R}^n$, T^n, $\mathbb{Z}^n$ and $\mathbb{Z}_{\mathbf{p}}$.

There are connections between the characters, translations, convolutions and the Fourier transform. The convolution $(w * g)(x) = \int_G w(a) g(x-a)\, da$ is a weighted sum of translations, since $\int_G w(a) g(x-a)\, da = \int_G w(a) T_a(g(x))\, da$. The characters $\chi_\xi(x) = \exp(2\pi i \xi x)$ are eigenfunctions of the translation operator. We have $T_a(\chi_\xi(x)) = \exp(2\pi i \xi a)\chi_\xi(x)$, so $\chi_\xi(x)$ is an eigenfunction and $\exp(2\pi i \xi a)$ is an eigenvalue.

G	$\widehat{G}$	$\langle \cdot, \cdot \rangle$	$\widehat{f}(\cdot)$	$f(\cdot)$
$\mathbb{R}^n$	$\mathbb{R}^n$	$\sum_{k=1}^n x_k \xi_k$	$\int_{\mathbb{R}^n} f(x)\overline{(\xi, x)}\, dx$	$\int_{\mathbb{R}^n} f(\xi)(\xi, x)\, d\xi$
T^n	$\mathbb{Z}^n$	$\sum_{k=1}^n x_k \xi_k$	$\int_{T^n} f(x)\overline{(\xi, x)}\, dx$	$\sum_{\xi \in \mathbb{Z}^n} f(\xi)(\xi, x)$
$\mathbb{Z}^n$	T^n	$\sum_{k=1}^n x_k \xi_k$	$\sum_{x \in \mathbb{Z}^n} f(x)\overline{(\xi, x)}$	$\int_{T^n} f(\xi)(\xi, x)\, d\xi$
$\mathbb{Z}_{\mathbf{p}}$	$\mathbb{Z}_{\mathbf{p}}$	$\sum_{k=1}^n x_k \xi_k / p_k$	$\sum_{x \in \mathbb{Z}^n} f(x)\overline{(\xi, x)}$	$\lvert\mathbb{Z}_{\mathbf{p}}\rvert^{-1} \sum_{\xi \in \mathbb{Z}^n} f(\xi)(\xi, x)$

Table 6.2: Multidimensional transforms. The Fourier transform, Fourier series, discrete-time Fourier transform and discrete Fourier transform. Here every group in $\mathbb{Z}_{\mathbf{p}} = \mathbb{Z}_{p_1} \oplus \mathbb{Z}_{p_2} \oplus \cdots \oplus \mathbb{Z}_{p_n}$ is of finite order, and $(\xi, x) = \exp(2\pi i \langle \xi, x \rangle)$.

The first order derivative can be expressed as $[f(x) - T_h(f(x))]/h$ as $h \to 0$. The convolution theorem states that the Fourier transform diagonalizes the convolution $(w * g)(x) = \int_G w(a) g(x - a)\, da$, such that $\mathcal{F}(w * g) = \mathcal{F}(w) \cdot \mathcal{F}(g)$. Thinking of the convolution as a weighted sum, the Fourier transform takes the weight function $w(x)$ to a single point—a notion formalized by the Dirac delta distribution δ, intuitively defined as a "spike" such that $\int \delta f(x)\, dx = f(0)$.

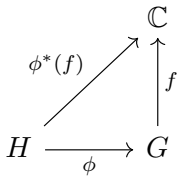
Fourier transforms are not only defined for classical functions, but also for the more general distributions. A distribution is defined by it's action on a set of suitable test functions. The most common example is δ, which acts on a test function to "evaluate at zero." Distributions will not be important in this thesis, but the curious reader is referred to [C. Gasquet, 1999] for a complete introduction to distributions, their derivative and their Fourier transform.

6.5 Pullbacks and pushforwards on groups

We now define the pullback and the pushforward. Pullbacks and pushforward are also categorical constructions, but here we will employ more concrete definitions. The pullback lets us move a function from a group to another.

Definition 6.10 (Pullback). *Let $f : G \to \mathbb{C}$ be a function and let $\phi : H \to G$ be a group homomorphism. The pullback of f along ϕ is defined as*

$\phi^*(f) = f \circ \phi$. *In other words, the following diagram commutes.*



⌟

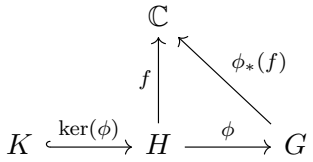
The pullback moves f from the target of ϕ to its source. The pushforward accomplishes the opposite, it moves f from the source of ϕ to its target.

Definition 6.11 (Pushforward). *Let $f : H \to \mathbb{C}$ be a function and let $\phi : H \to G$ be a group homomorphism. The pushforward of f along ϕ is defined as*

$$\phi_*(f)(x) = \sum_{h \in S} f(h), \text{ where } S = \{h \in H \mid \phi(h) = x\}.$$

If $S = \emptyset$ for some $x \in G$, then we take $\phi_*(f)(x)$ to be zero. ⌟

The diagram below shows the pushforward of f along ϕ. In words, we sum the function values over every $h \in H$ such that $\phi(h) = x$, and assign this value to $\phi_*(f)(x)$. We assume that the sum in Definition 6.11 is absolutely convergent, i.e. $\sum_{h \in S} |f(h)| \leq \infty$, so that the sum converges and the order of the terms is insignificant.



Computationally, we first solve $\phi(h) = x$ for any $h \in H$ that solves the equation. This equation can have zero, one, many or an infinite number of solutions, and is solved using Algorithm 3 from Section 5.4 when the groups are FGAs. Next we compute $\ker(\phi)$ and iterate through the elements $v \in K$ to generate solutions to the equation. If a full iteration over K computationally infeasible, we assume that $f(h)$ decays as $\|h\| \to \infty$, and consequently restrict the sum using a norm. We approximate a sum with infinite terms by summing over a subset $\{v \in K \mid \|v\| \leq C\}$, where $\|\cdot\|$ is a norm and C is a constant. In summary, if $\phi(h) = x$ we may approximate the pushforward $\phi_*(f)(x)$ as

$$\phi_*(f)(x) \approx \sum_{\|v\| \leq C} f\left(h + \ker(\phi)(v)\right). \tag{6.3}$$

6.6 Computing pushforwards

If we are able to generate elements $v \in K$ by increasing norm, we can approximate the sum in Definition 6.11 of the pushforward by the truncated sum given in Equation (6.3). We could for instance decide on a fixed number of terms, or adaptively terminate the sum when $|f| < \epsilon$ for some small ϵ. We now present an algorithm for generating elements by the infinity norm, defined as $\|v\|_\infty = \max_i |v_i|$.

Problem 6.12 (Generate elements in $\mathbb{Z}^r$ by infinity norm). *Consider $\mathbb{Z}^r$, where r is the free rank. We wish to generate every element $v_1, v_2, v_3, \cdots \in \mathbb{Z}^r$ in ordered fashion such that $\|v_i\|_\infty \leq \|v_{i+1}\|_\infty$ for every $i = 1, 2, 3, \ldots$.*

The problem is solved by utilizing Algorithm 5. Given a fixed $\mathbb{Z}^r$, we apply the algorithm with $C = 1, 2, 3, \ldots$ to generate every element in $\mathbb{Z}^r$ sorted by the infinity norm. To motivate the algorithm, first observe that $2r$ Cartesian products (or equivalently, lower rank hypercubes) with $(2C-1)^{r-1}$ group elements cover the elements $\{v \in \mathbb{Z}^r \mid \|v\|_\infty = C\}$, which can be thought of as an r-dimensional cube. Figure 6.1 shows the situation when $r = 2$.

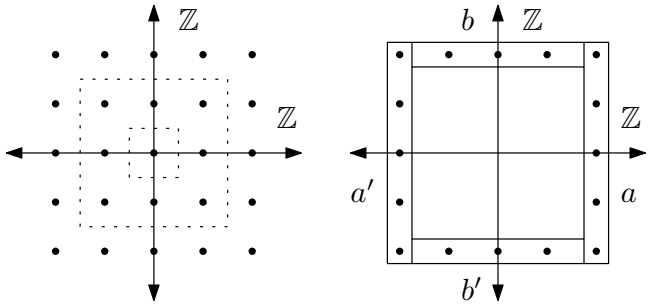


Figure 6.1: The left part of the figure shows the elements $v \in \mathbb{Z}^2$ with $\|v\|_\infty$ equal to 0, 1 and 2, separated by dotted lines. The right part of the figure shows how Algorithm 5 generates elements $v \in \mathbb{Z}^2$ with $\|v\|_\infty = 2$. The elements are contained in 4 1-dimensional hypercubes a, a', b and b'. The algorithm iterates over the hypercubes a and b, yielding elements (and reflected elements from a' and b'). Care must be taken so as not to yield corner elements more than once.

The algorithm generates every element in these hypercubes sequentially, keeping track of boundary elements so as not to yield the same elements more than once. A hypercube with side length $(2C+1)$ has $(2C+1)^r - (2C-1)^r$ boundary elements. Since $(2C+1)^r - (2C-1)^r \approx 2r(2C)^{r-1}$ by the binomial theorem, the running time of the algorithm is $\mathcal{O}\left(r(2C)^{r-1}\right)$.

Algorithm 5: Efficiently generating every $v \in \mathbb{Z}^r$ with $\|v\|_\infty = C$.

Input : Free rank r, norm value C.
Output : Every $v \in \mathbb{Z}^r$ such that $\|v\|_\infty = C$.

for p **in** $[1, 2, \ldots, r]$ **do**
  `// Iterate over the` r $(r-1)$`-dimensional hypercubes enclosing the` r`-dimensional hypercube.`
  $B = [0] \times (p-1) + [1] \times (r-p)$ `// Boundary conditions for cube` p`.`
  for v **in** $\bigtimes_{i=1}^{r-1} [-C + B[i], C - B[i]]$ **do**
    `// Loop over elements` v `on the` $r-1$ `dimensional`
    `// cube, yield the element and it's reflection.`
    yield v with C inserted into p'th position
    yield v with $-C$ inserted into p'th position `// Reflection.`
  end
end

To solve the problem for a more general $\mathbb{Z}_{\mathbf{p}}$, the algorithm can be extended further. A naive approach would be to use Algorithm 5, project the elements, and make sure not to yield duplicates by using a lookup table. Such lookups could be costly in terms of computational time. Consider for instsance that while there are 6 elements $v \in \mathbb{Z}_3 \oplus \mathbb{Z}$ such that $\|v\| = 5$, there are 40 elements with $\|v\| = 5$ in $\mathbb{Z}^2$. Significant reductions in computational time can be made when the orders of the groups in $\mathbb{Z}_{\mathbf{p}}$ are small compared to C.

Three modifications were applied to Algorithm 5 to construct an efficient algorithm for $\mathbb{Z}_{\mathbf{p}}$: (1) hypercubes "outside" of $\mathbb{Z}_{\mathbf{p}}$ are immediately discarded, (2) if the size of a hypercube may be reduced by the structure of $\mathbb{Z}_{\mathbf{p}}$, this is done before iteration of elements starts (the inner for-loop), and (3) if a reflection "wraps around" in $\mathbb{Z}_{\mathbf{p}}$ it is not yielded twice. This more general algorithm is implemented in the software, see Appendix A.

6.7 Dual homomorphisms

We now introduce the dual homomorphism, which is analogous to the adjoint in linear algebra.

Definition 6.13 (Dual homomorphism). *Given $\phi : H \to G$, the dual homomorphism $\widehat{\phi} : \widehat{G} \to \widehat{H}$ is the adjoint with respect to the dual pairing.*

$$(\widehat{g}, \phi(h))_G = (\widehat{\phi}(\widehat{g}), h)_H$$

$$\begin{array}{ccc} \widehat{h} \in \widehat{H} & \xleftarrow{\widehat{\phi}} & \widehat{G} \ni \widehat{g} \\ | & & | \\ h \in H & \xrightarrow{\phi} & G \ni g \end{array}$$

In the diagram above, the lines denote dual pairs of groups. ⌟

In linear algebra over $\mathbb{R}$, the adjoint is the transpose, since $\langle x, Ay\rangle = \langle A^T x, y\rangle$. For details about adjoints in linear algebra, see [Roman, 2005].

Proposition 6.14 (Dual of a homomorphism between FGAs). *Consider a homomorphism between FGAs $\phi : \mathbb{Z}_{\mathbf{q}} \to \mathbb{Z}_{\mathbf{p}}$ given by a matrix A. We index the elements as shown in the diagram below.*

$$\begin{array}{ccc} \widehat{h} \in \widehat{\mathbb{Z}}_{\mathbf{q}} & \xleftarrow{\widehat{\phi}} & \widehat{\mathbb{Z}}_{\mathbf{p}} \ni \widehat{g} \\ | & & | \\ h \in \mathbb{Z}_{\mathbf{q}} & \xrightarrow{\phi} & \mathbb{Z}_{\mathbf{p}} \ni g \end{array}$$

If $\phi(h) = Ah$, then $\widehat{\phi}(\widehat{g}) = \operatorname{diag}(q) A^T \operatorname{diag}(p) \widehat{g}$.

Proof. Recall that the pairing $(\widehat{g}, g)_{\mathbb{Z}_{\mathbf{p}}}$ is given by $\exp(2\pi i \langle \widehat{g}, g\rangle)$, or more explicitly as $\exp\left(2\pi i \sum_j \widehat{g}_j g_j / p_j\right)$. We write this in matrix notation as $\exp\left(2\pi i \widehat{g}^T \operatorname{diag}(1/p) g\right)$. If the j'th group in $\mathbb{Z}_{\mathbf{p}}$ is $\mathbb{Z}$, then we take the j'th diagonal entry to be 1. If the j'th group is $\mathbb{Z}_{p_j}$, then we take j'th diagonal entry to be p_j. We assume that $\widehat{\phi}(\widehat{g}) = B\widehat{g}$ for some matrix B. Comparing

$$\begin{aligned} (\widehat{g}, \phi(h))_{\mathbb{Z}_{\mathbf{p}}} &= \exp\left(2\pi i \widehat{g}^T \operatorname{diag}(1/p) A h\right) \quad \text{with} \\ (\widehat{\phi}(\widehat{g}), h)_{\mathbb{Z}_{\mathbf{q}}} &= \exp\left(2\pi i (B\widehat{g})^T \operatorname{diag}(1/q) h\right) \end{aligned}$$

we observe that

$$\widehat{g}^T \operatorname{diag}(1/p) A h = \widehat{g}^T B^T \operatorname{diag}(1/q) h$$

This implies that $\operatorname{diag}(1/p) A = B^T \operatorname{diag}(1/q)$, and solving for B yields $\operatorname{diag}(q) A^T \operatorname{diag}(p)$. □

We will now investigate how the Fourier transform behaves under a change of variables. The following theorem is found in Chapter 4 of [Madisetti, 2009]. Though the theorem concerns morphisms $\phi : \mathbb{R}^n \to \mathbb{R}^n$, factors proportional to determinants also appear when sampling with $\phi : \mathbb{Z}^n \to \mathbb{R}^n$. In general the factors also depend on how dual pairings and Fourier transforms are defined.

Theorem 6.15 (Change of variables in the Fourier transform). *Let $f(g)$ be a function on $G = \mathbb{R}^n$ with Fourier transform $\widehat{f}(\widehat{g})$ on $\widehat{G} = \mathbb{R}^n$. Suppose that $\phi : H \to G$ is a homomorphism, where $H = \mathbb{R}^n$. Assume that ϕ is represented by a matrix A with $\det(A) \neq 0$, i.e. ϕ is a bijection. If we change variables $f(g) \mapsto f(h)$, then the Fourier transform scales according to $\widehat{f}(\widehat{g}) \mapsto |\det(A)|\, \widehat{f}(\widehat{h})$.*

Proof. By the definition of the Fourier transform we know that $\widehat{f}(\widehat{g}) = \mathcal{F}(f(g)) = \int_G f(g)(\widehat{g}, g)\, dg$. Changing variables according to $g \mapsto h$ gives

$$\int_G f(h)(\widehat{g}, g)\, dg = \int_G f(\phi^{-1}(g))(\widehat{g}, g)\, dg = \int_G f(\phi^{-1}(g))(\widehat{g}, \phi(h))\, dg =$$
$$\int_G f(h)(\widehat{\phi}(\widehat{g}), h)\, dg = \int_H f(h)(\widehat{h}, h)\, |\det(A)|\, dh = |\det(A)|\, \widehat{f}(\widehat{h}),$$

where we used the defining property of the dual homomorphism, and the Jacobian of ϕ to change variables in the multidimensional integral. □

Theorem 6.16 (Fundamental duality theorem of LCAs). *The following diagram commutes. If ϕ is an epimorphism, then its dual is a monomorphism, and vice versa.*

$$\begin{array}{ccccc}
\widehat{H} & \xleftarrow{\ \widehat{\phi}\ } & \widehat{G} & \xleftarrow{\ \widehat{\psi}\ } & \widehat{K} \\
\downarrow & & \downarrow & & \downarrow \\
H & \xrightarrow{\ \phi\ } & G & \xrightarrow{\ \psi\ } & K
\end{array}$$

The theorem above is found in Section 3 in [Munthe-Kaas, 2016], where related concepts such as chain complexes and short exact sequences are also discussed. Another source for duality of subgroups and quotients is Chapter 4 in [Reiter, 1968].

Definition 6.17 (Annihilator homomorphism). *The annihilator homomorphism of $\phi : H \to G$ is the kernel of the dual of ϕ, as depicted in the diagram below.*

$$\begin{array}{ccccc}
\widehat{H} & \xleftarrow{\ \widehat{\phi}\ } & \widehat{G} & \xleftarrow{\ \phi^{\perp}\ } & \widehat{K} \\
\downarrow & & \downarrow & & \downarrow \\
H & \xrightarrow{\ \phi\ } & G & \xrightarrow{\ \mathrm{coker}(\phi)\ } & K
\end{array}$$

The annihilator of ϕ is denoted $\phi^{\perp}$. ⌟

The annihilator of ϕ "annihilates" the dual pairing in the sense that

$$(\phi^\perp(\hat{k}), \phi(h))_G = 1$$

for every $\hat{k} \in \widehat{K}$ and $h \in H$. To see this, we use the defining property of the dual homomorphism to write $(\phi^\perp(\hat{k}), \phi(h))_G$ as $(\widehat{\phi}(\phi^\perp(\hat{k})), h)_H$. Notice that the composition $\widehat{\phi} \circ \phi^\perp$ maps to $0 \in H$ by the definition of the kernel morphism, and thus $(0, h)_H = 1$ for every $h \in H$ and $\hat{k} \in \widehat{K}$ as claimed. In the following section we will define lattices, and observe that when ϕ generates a lattice, $\phi^\perp$ generates a dual (or reciprocal) lattice.

6.8 Sampling and periodization

The FFT algorithm for the DFT is used in practical numerical computations. To make use of its $\mathcal{O}(n \log(n))$ runtime, the problem must be moved to a discrete, compact domain, i.e. $\mathbb{Z}_{\mathbf{p}}$ with $p_i \geq 1$. As done in [Munthe-Kaas, 2016], we will employ pullbacks and pushforwards of functions to "move" $f(x)$ to $\mathbb{Z}_{\mathbf{p}}$. Pullbacks along monomorphisms will sample f, while pushforwards along epimorphisms will periodize $f(x)$. We first consider Fourier analysis on T^d, then on $\mathbb{R}^d$.

6.8.1 Fourier analysis on T^d

In order to approximate the Fourier series coefficients of a function $f(x)$ defined on T^d, sampling is necessary to bring the function to $\mathbb{Z}_{\mathbf{p}}$ where DFT can be used. Sampling is done using a monomorphism ϕ, which defines a subgroup of T^d. The subgroup defined by ϕ is a lattice, which we define as follows.

Definition 6.18 (Lattice). *A lattice is a discrete and closed subgroup $H < G$ such that G/H is compact.* ⌟

In the language of morphisms, when $\phi : H \to G$ is a lattice, then $\operatorname{coker}(\phi) : G \to H/\operatorname{im}(\phi)$ maps to a compact group. Two examples are given by

$$\mathbb{Z}^d \xhookrightarrow{\phi} \mathbb{R}^d \xrightarrow{\operatorname{coker}(\phi)} T^d \qquad \text{and} \qquad \mathbb{Z}^d \xhookrightarrow{\phi} \mathbb{Z}^d \xrightarrow{\operatorname{coker}(\phi)} \mathbb{Z}_{\mathbf{q}},$$

where T^d and $\mathbb{Z}_{\mathbf{q}}$ are compact. Lattices are used in areas such as crystallography and multiple integration. Using a lattice to approximate a multidimensional integral corresponds to the evaluation of $\widehat{f}(0)$, i.e. evaluating the Fourier transform in the origin. Lattice integration makes use of Fourier analysis as well as group theory, see for instance [Sloan and Joe, 1994].

To approximate Fourier coefficients, we follow the arrows in Diagram (6.4): (1) first we sample to $\mathbb{Z}_{\mathbf{p}}$ using a pullback of $f(x)$ along ϕ, (2) then we compute the Fourier transform using the DFT, and (3) finally we move the coefficients from $\mathbb{Z}_{\mathbf{p}}$ to $\mathbb{Z}^d$.

$$\begin{array}{ccc} \mathbb{Z}_{\mathbf{p}} & \overset{\widehat{\phi}}{\twoheadleftarrow} & \mathbb{Z}^d \\ \| & \overset{\sigma}{\curvearrowright} & \| \\ \mathbb{Z}_{\mathbf{p}} & \overset{\phi}{\hookrightarrow} & T^d \end{array} \qquad \begin{array}{ccc} \mathbb{Z}_{\mathbf{p}} & \overset{\text{periodize}}{\twoheadleftarrow} & \mathbb{Z}^d \\ \text{FFT}\downarrow\| & \overset{\text{transversal}}{\curvearrowright} & \| \\ \mathbb{Z}_{\mathbf{p}} & \overset{\text{sample}}{\hookrightarrow} & T^d \end{array} \tag{6.4}$$

To interpret the Fourier coefficients on $\mathbb{Z}^d$, we employ a transversal morphism σ, which we now define.

Definition 6.19 (Transversal of an epimorphism). *A transversal of an epimorphism $\phi : H \to G$ is a map σ such that $(\phi \circ \sigma)(g) = g$ for every $g \in G$. In other words, σ is a left-inverse of ϕ.* ⌟

In category theory, a morphism with this property is called a section. A transversal is in general not a homomorphism. We used a transversal in the code example in Section 2.2, where we investigated $f(x) = x$ on T. At the time, we did not discuss the choice of the transversal σ, which is what we will examine now.

6.8.2 The Voronoi transversal

Consider again an epimorphism $\phi : H \to G$ and a transversal $\sigma : G \to H$. In the two-dimensional, orthogonal case elements are typically mapped as shown in Figure 6.2. This function is called `fftshift` is MATLAB and in Python[1], and it is fast to compute. The elements of $G = \mathbb{Z}_n \oplus \mathbb{Z}_n$ are mapped to $H = \mathbb{Z} \oplus \mathbb{Z}$ so that they are close to the origin. In the language of group theory, the transversal picks a coset representative. In signal processing, this is called de-alisaing.

In general, the Voronoi transversal is a natural choice. Intuitively, the Voronoi transversal maps the coefficients such that the sampled points are interpolated using low-frequency complex exponentials (as opposed to high-frequency ones) in the Fourier reconstruction. Geometrically, this amounts to mapping elements in such a way that they end up close to the origin, which is made concrete in the following definition.

[1] More specifically, the Numpy library for numerical computing in Python.

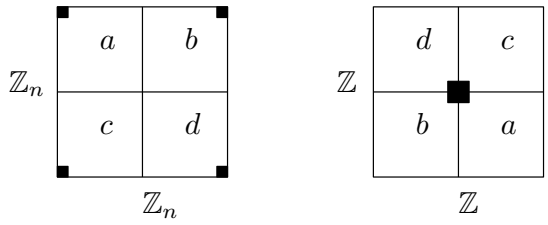


Figure 6.2: When interpreting coefficients sampled orthogonally, a function on $\mathbb{Z}_n \oplus \mathbb{Z}_n$ is typically moved to $\mathbb{Z}^2$ by shifting the areas a, b, c and d as shown in the figure. The black dots indicate corners close to the origin.

Definition 6.20 (Voronoi transversal). *Consider the following diagram, where* $\ker(\phi)$ *defines a lattice in* H*.*

$$G \underset{\sigma}{\overset{\phi}{\twoheadleftarrow}} H \xhookleftarrow{\ker(\phi)} K$$

The Voronoi transversal maps $g \in G$ *to a* $h \in H$ *such that*

$$\|\sigma(g)\| \leq \|\sigma(g) - \ker(\phi)(k)\|$$

for every $k \in K$*. The image of* σ *is a polyhedron around the origin in* H*.* ⌟

In crystallography, the image of the Voronoi transversal is a Wigner–Seitz primitive cell in the lattice generated by $\ker(\phi)$. The Wigner–Seitz primitive cell contains exactly one lattice point, the origin, such that every point in the cell is closer to the origin than neighboring lattice points. For more about lattices, reciprocal lattices and cells in the context of crystallography, see Chapter 6 in [Christopher. Hammond, 2009]. We will say more about reciprocal lattices, which are generated by the annihilator morphism, in the next section.

Algorithm 6 gives a high-level account of how to compute a general Voronoi transversal as given in Defintion 6.20. In line 1 we solve $\phi(y') = x$. If $\phi : \mathbb{Z}^d \to \mathbb{Z}_{\mathbf{p}}$, then ϕ is left-free and Algorithm 3 on page 45 can be used. If $\phi : \mathbb{R}^d \to T^d$, then a linear algebra solver can be used. The algorithm uses $\ker(\phi)$ to generate neighboring solutions and shifts the solution to minimize a norm $\|\cdot\| : H \to \mathbb{R}_{\geq 0}$.

6.8.3 Fourier analysis on $\mathbb{R}^d$

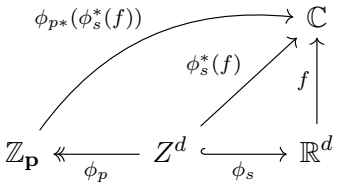
To interpret a function on $\mathbb{R}^d$ as a function on $\mathbb{Z}_{\mathbf{p}}$ we must sample and periodize. This is done with a pullback and a pushforward, as shown in the

Algorithm 6: Computing the Voronoi transversal.

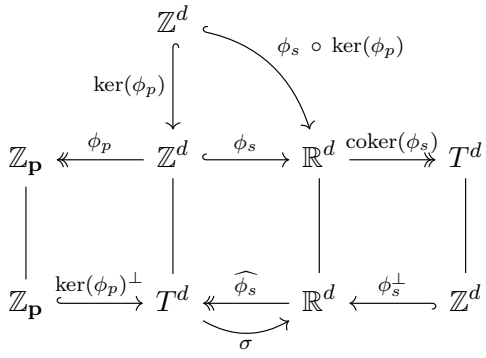
Input : Epimorphism $\phi : H \to G$, element $x \in G$.
Output : Element $y \in H$ which minimizes norm and solves $\phi(y) = x$.

```
// Solve using linear algebra routine or equation solver for FinAb.
```
1 Solve $\phi(y') = x$ for some solution $y' \in H$.
2 Compute the kernel of ϕ.
```
// Find group elements in the source of the kernel to generate
   alternative, nearby solutions with respect to y'.
```
3 $K := \{k \in \text{source}(\ker(\phi)) \mid \|k\|_\infty \in \{0, 1\}\}$
```
// Find the k ∈ K which minimizes a norm.
```
4 $k^* := \text{argmin}_{k \in K} \|y' - \ker(\phi)(k)\|$
```
// Shift to the minimizing solution and return.
```
5 **return** $y' - \ker(\phi)(k^*)$

diagram below. Periodizing first and then sampling would yield the same result.



We now examine Diagram (6.5), which describes sampling, periodization, dualizing and interpretation of the result. The homomorphism ϕ_s defines a sampling lattice in $\mathbb{R}^d$. The homomorphism $\ker(\phi_p)$ defines a periodization lattice in $\mathbb{Z}^d$ with cokernel ϕ_p, and pushforward along ϕ_p periodizes the (now sampled) function. The composition $\phi_s \circ \ker(\phi_p)$ defines a sub-lattice of ϕ_s in $\mathbb{R}^d$, which is sometimes referred to as the periodization lattice.



(6.5)

We turn our attention to the bottom row, representing the dual space. The homomorphism $\ker(\phi_p)^\perp$ defines the dual sampling lattice, which is the annihilator of the primal periodization lattice. Dually, the homomorphism $\phi_s^\perp$

defines the dual periodization lattice, which is the annihilator of the primal sampling lattice. In Chapter 4 of [Madisetti, 2009], the author uses sampling and periodization lattices along with so-called lattice combs to prove this result. The dual lattice is also referred to as the reciprocal lattice in literature.

To interpret $\widehat{f}(\xi)$ on as a function on $\mathbb{R}^d$ we use the composition $\sigma \circ \ker(\phi_p)^\perp : \mathbb{Z}_{\mathbf{p}} \to \mathbb{R}^d$. The scaling must be accounted for by the factors $|\det(\phi_s)|$ and $|\det(\ker(\phi_p))|$, and may also vary slightly depending on which definition of the DFT that is used. Diagram (6.5) is inspired by a similar diagram in Section 3.8 on lattice rules in [Munthe-Kaas, 2016], and in the following section we will examine the diagram in a more concrete setting.

6.9 Hexagonal Fourier analysis in $\mathbb{R}^2$

In this section Fourier analysis on $\mathbb{R}^d$ will be made concrete by considering the hexagonal lattice on $\mathbb{R}^2$. We introduce the hexagonal lattice, show how to sample and periodize a function on such a lattice, review some recent research in this area and briefly examine lattices in $\mathbb{R}^3$.

6.9.1 The hexagonal lattice

Hexagonal sampling is optimal in $\mathbb{R}^2$ in the following sense: there is no sampling pattern that requires fewer sample points to reconstruct a band-limited[2] function with an isotropic spectrum. Intuitively, this is because a circle inscribed in a hexagon almost fills the hexagon, and the hexagons tile the plane, as seen in Figure 6.3. The efficiency of the hexagonal lattice is 90.8%, while a square orthogonal lattice only achieves 78.5%. Other advantages include greater angular resolution, higher symmetry and equal distance to all neighboring points. For an overview of the advantages of hexagonal lattices, the reader is referred to [Xiangjian He and Wenjing Jia, 2005].

While hardware supporting hexagonal image processing and display is scarce, extensive research has been conducted on hexagonal sampling. Detailed information about sampling, convolution, and computing the FFT on a hexagonal lattice is found in [Mersereau, 1979]. The author recognizes that the hexagonal pattern is a special case of a skewed sampling raster, as seen in rightmost part of Figure 6.3. He also states that "any sample which is a member of one fundamental period can be exchanged for the corresponding point in another fundamental period," and recognizes that a fundamental

[2] A function is band-limited if it's Fourier transform is zero outside of a region of finite support.

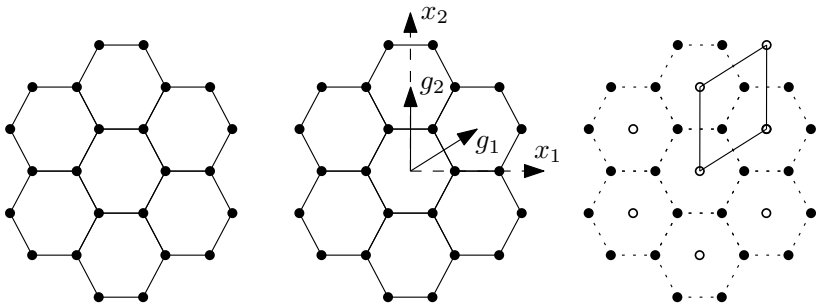


Figure 6.3: The figure on the left shows part of a plane tiled with hexagons. The middle figure shows the generators g_1 and g_2 and the plane axes x_1 and x_2. The figure to the right shows the parallelogram spanned by g_1 and g_2, which is a fundamental domain, or fundamental period.

period can be a parallelogram or a hexagon, as shown in Figure 6.4. Informally, a fundamental period may be thought of as a geometric shape which constitutes a periodic tiling of the plane, or a general d-dimensional space.

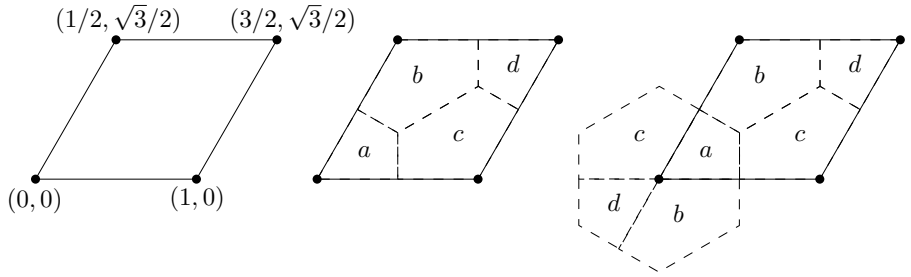
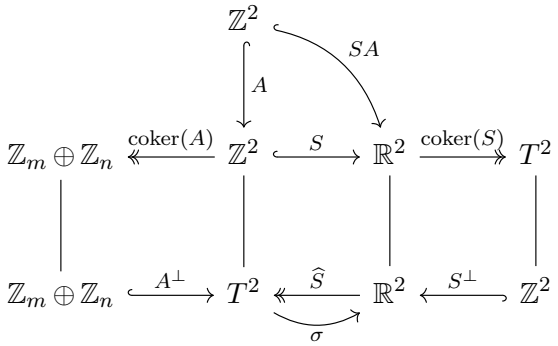


Figure 6.4: The figure on the left shows generators for a hexagonal lattice. The middle and right figure show that the fundamental period may be taken to be a hexagon or a parallelogram, since they both tile $\mathbb{R}^2$.

We will now periodize and sample a function on $\mathbb{R}^2$ using a hexagonal lattice. Diagram (6.6) below shows the setup for Fourier analysis with hexagonal sampling in a parallelogram. We wish to be concrete, so we use matrices A and S notationally and explicitly give entries for the matrices. In Section 7.6 we will use software created as part of this thesis to compute every morphism

in the diagram when A and S are known.



(6.6)

The matrices in Diagram (6.6) are given by the following equations. In these cases, the dual is the transpose and the annihilator is the transpose of the inverse. This is not the case for homomorphisms between general FGAs, as we saw in Proposition 6.14.

$$S = \begin{pmatrix} 1 & 1/2 \\ 0 & \sqrt{3}/2 \end{pmatrix} \qquad A = \begin{pmatrix} m & 0 \\ 0 & n \end{pmatrix} \qquad A^\perp = \begin{pmatrix} 1/m & 0 \\ 0 & 1/n \end{pmatrix}$$

$$\operatorname{coker}(A) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \widehat{S} = \begin{pmatrix} 1 & 0 \\ 1/2 & \sqrt{3}/2 \end{pmatrix} \qquad S^\perp = \begin{pmatrix} 1 & 0 \\ -\frac{\sqrt{3}}{3} & \frac{2\sqrt{3}}{3} \end{pmatrix}$$

As explained in the previous section, we sample and periodize $f(x)$ on $\mathbb{R}^2$ by a pullback along S followed by a pushforward along $\operatorname{coker}(A)$. On $\mathbb{Z}_m \oplus \mathbb{Z}_n$ we use a standard multidimensional FFT implementation of the DFT to move to the bottom row of Diagram (6.6). To interpret the results, we map the coordinates on $\mathbb{Z}_m \oplus \mathbb{Z}_n$ using $\sigma \circ A^\perp$, where σ is the Voronoi transversal. Scaling proportional to $|\det(S)|$ must be applied to the approximated Fourier coefficients.

6.9.2 Research on hexagonal lattices

In [Ehrhardt, 1993], the author describes a hexagonal FFT with rectangular output. He samples with $S = \left(\begin{smallmatrix} 2d/\sqrt{3} & -d/\sqrt{3} \\ 0 & d \end{smallmatrix}\right)$ and periodizes with $A = \left(\begin{smallmatrix} N_1 & N_2/2 \\ 0 & N_2 \end{smallmatrix}\right)$. The output is rectangular since the product SA is a diagonal matrix. To see this, notice that matrix given by $S^\perp$ is a transversal of $\widehat{S}$, since $S^\perp$ is a left inverse such that $S^\perp \circ \widehat{S} = \mathrm{Id}_{T^2}$. Interpretation of the coefficients can be done with $\sigma \circ A^\perp = S^\perp \circ A^\perp = S^{-T} \circ A^{-T} = (SA)^{-T}$, which is diagonal when SA is diagonal; and therefore the output is rectangular.

In [Vince and Zheng, 2007], the authors show how to reduce DFT computations on an arbitrary lattice in $\mathbb{R}^d$ to the standard multidimensional FFT. They introduce lattices generated by matrices L and L_0, and state that if

the quotient L/L_0 has divisors $N_1, N_2, \ldots, N_d$, then the DFT can be computed using the FFT. This amounts to finding a diagonal integer matrix $D = \operatorname{diag}(N_1, N_2, \ldots, N_d)$ in the left part of Digram (6.7) below, which is done using the SNF in the paper. Comparing their approach to our own, their D is our A, their sampling matrix L is our S, and so forth. This is shown in Diagram (6.7) below. While the authors search for a diagonal D, we chose our A to be diagonal.

$$\begin{array}{ccc} \mathbb{Z}^2 & & \\ {\scriptstyle D}\downarrow & \searrow^{L_0} & \\ \mathbb{Z}^2 & \xhookrightarrow{L} & \mathbb{R}^2 \end{array} \qquad \begin{array}{ccc} \mathbb{Z}^2 & & \\ {\scriptstyle A}\downarrow & \searrow^{SA} & \\ \mathbb{Z}^2 & \xhookrightarrow{S} & \mathbb{R}^2 \end{array} \tag{6.7}$$

In [Birdsong and Rummelt, 2016] the authors consider a hexagonal FFT using an array set addressing (ASA) coordinate system, which describes a hexagonal grid as two interleaved rectangular arrays, see the left part of Figure 6.5. The authors express the FFT in the ASA coordinate system, and show that it can be computed using standard FFT routines.

An approach to the problem in line with the theory presented in this thesis would be different. The data is already sampled and periodized, and can interpreted as shown on the right in Figure 6.5, since both shapes are fundamental periods. The parallelogram data can be mapped to $\mathbb{Z}_n \oplus \mathbb{Z}_n$, where the FFT is available. Interpretation may then be done as in the bottom row of Diagram (6.6)—by using the annihilator of the parallelogram periodicity matrix and a transversal of the the dual sampling homomorphism.

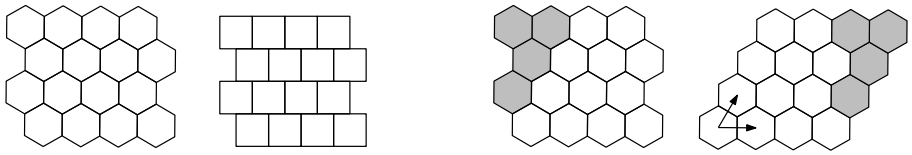


Figure 6.5: The figure on the left demonstrates how a hexagonally sampled rectangular area may be thought of as two interleaved rectangular arrays. The figure on the right shows how the shape can be interpreted as a parallelogram by changing the fundamental domain

6.9.3 Lattices in $\mathbb{R}^3$ and beyond

Four years after publishing a thorough paper on hexagonal sampling in $\mathbb{R}^2$, Mersereau published a paper on Fourier analysis on periodically sampled multidimensional signals, see [Mersereau and Speake, 1983]. He summarizes by stating that "almost anything that can be done in the one-dimensional

case or the rectangular multidimensional case can be generalized." His conclusion seems correct. In more recent papers, authors typically consider special cases, derive FFTs in novel ways, or attempt to speed up algorithms.

Just as in $\mathbb{R}^2$, orthogonal sampling is sub-optimal for isotropically band-limited functions. The optimal lattice is the body-centered cubic (BCC), which is 29.3% more efficient than naive, orthogonal sampling. The face-centered cubic (FCC) is also a viable contestant, but it is not as efficient as the BCC lattice. The lattices are generated by the matrices

$$S_{\text{FCC}} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad S_{\text{BCC}} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix},$$

which are given in [Zheng and Gu, 2014]. These two lattices constitute a reciprocal pair: when a FCC lattice is used for sampling, the annihilator is a BCC lattice, and vice versa.

As the number of dimensions increase, the Voronoi cells become increasingly complex. The Voronoi cell for the BCC lattice is a truncated octahedron, and the Voronoi cell for the FCC lattice is a rhombic dodecahedron, see Figure 6.6. In higher dimensions, general algorithms for the Voronoi transversal (such as the one presented on page 68) suffer from the curse of dimensionality and are increasingly slow as a result. The orthogonal case is computationally fast due to algorithms such as `fftshift`. A middle ground may be achieved if one computes transversals "by hand" for specific, non-orthogonal cases, as is done in [Zheng and Gu, 2014].

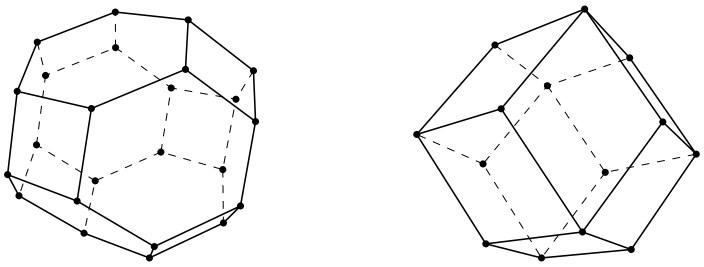


Figure 6.6: Left: truncated octahedron. Right: rhombic dodecahedron.

To summarize, the well-known ideas, algorithms and interpretations of the one-dimensional case are available in higher dimensions. Orthogonal sampling is not optimal, but it is still the popular choice. The popularity is likely due to a combination of existing hardware, ease of interpretation and simple, efficient Voronoi transversal via routines such as `fftshift`.

Chapter 7

The abelian software library

The primary goal of this project is to create software for computations on elementary LCAs. The result of this endeavor is `abelian`, an open-source Python library. We spend the first section of this chapter briefly discussing Python, it's popularity and some principles of software development.

We give a detailed introduction of `abelian`: the purpose and philosophy, an overview of features and objects, and several computational examples with code. Most mathematical objects and definitions presented in this thesis are implemented in the library using three classes: `LCA`, `HomLCA` and `LCAFunc`.

7.1 Scientific programming and Python

The Python programming language was developed in the early 1990s by Guido van Rossum. It's a high-level, object-oriented language which is perhaps best known for it's readability and portability. Scientists are increasingly adopting Python. An example of it's popularity comes from the programming website Stack Overflow, where the Python question-tag has overtaken every other major programming language in high-income countries, see [Robinson, 2017].

In a talk at the PyCon 2017 conference, Jake Vanderplas attributes the popularity to (1) a high degree of interoperability with other programming languages, (2) a "batteries included" philosophy and great number of high quality third party libraries, (3) the simplicity and dynamic nature of Python and (4) it's open ethos, making it well fit to science. See [VanderPlas, 2017] for more on the popularity of Python.

We now mention two Python software projects which `abelian` depends on. Numpy is a library which implements homogeneous n-dimensional arrays fa-

cilitating efficient numerical computations, see [Walt et al., 2011] for details. Sympy is a library for symbolic computations with mathematical objects such as variables, functions, matrices, and so forth, see [Meurer et al., 2017]. Both of these are examples of the many popular scientific Python libraries.

`abelian` was written in Python because (1) the author had some prior knowledge of the language, (2) implementations of FFTs and a matrix-class were available through Numpy and Sympy respectively and (3) Python is free, popular and has a package management system, all of which increase the chance of others using `abelian` in the future.

7.2 Principles of software development

In [Buckheit and Donoho, 1995], it was stated that:

"An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures."

A similar philosophy to the above has been guiding the work with `abelian`. Software hastily developed and merely attached as an appendix in a thesis has little chance of ever being found, let alone used, by anyone. In an attempt to create high-quality software, the following principles have been guiding the project.

- **Object orientation** – While Python does not force object orientation, it is still an object oriented language. It seems natural to make use of object oriented features when representing mathematical objects in software, at least to the extent that it increases readability and improves structure. Object orientation also makes it easier for users to incorporate the objects in their own projects, and to further extend the code.
- **Testing** – Many modern software projects include a test suite. Test driven development is a common way to write software, in which the tests are written before the actual code. In Python there are libraries which make testing easier, such as the unittest and doctest libraries. A unit test is a test for a small portion of the code, and a doctest is a test which is part of the documentation.
- **Documentation** – Documentation is a broad term which incorporates consistent naming conventions, descriptive variables names, inline comments in the code, docstrings for functions and methods, installation instructions for the users, tutorials, and so forth. Conven-

tions exist for all of the above, and the Python community puts high value on documentation and code readability.

- **Distribution** – There exists an official Python package index, and uploading code to it is free. Once published, anyone can install, remove and upgrade the software by executing a single command in the terminal. By publishing the source code on a public repository such as Github, the project can be shared and improved efficiently.

For more information about the Python language and object orientation, the reader is referred to [Ramalho, 2015]. A Python book which includes chapters on testing, documentation and distribution is [Alchin and Browning, 2014]. For more details, see [Kristian. Rother, 2017], which describes testing and debugging in great detail.

7.3 Introducing `abelian`

The goal of this project was to write a software library for computations on elementary LCAs. The end result is the `abelian` software library, written in Python, open sourced on Github and distributed on the Python package index.

- Github – `https://github.com/tommyod/abelian/`
- Python package index – `https://pypi.org/project/abelian/`
- Documentation – `http://abelian.readthedocs.io/en/latest/`

The following sections give an overview of `abelian`. We start with the purpose and philosophy, give an overview of the most important software components, and give several example computations.

7.3.1 Purpose and philosophy

The purpose of the `abelian` library is to provide the end-user with a free, comprehensible interface to mathematical objects such that computations on elementary LCAs can be performed. More specifically, the software provides a framework for the following:

- Computations on the elementary LCAs, i.e. $\mathbb{R}$, T, $\mathbb{Z}$ and $\mathbb{Z}_n$ and direct sums of these groups.
- Sampling and periodization of continuous functions and interpretations of sampled values onto continuous groups. Numerical computations are performed on discrete groups of finite order using the FFT.
- Sampling functions on continuous groups on arbitrary lattices.

- The software should reflect the mathematical theory.

In Section 7.2 we examined principles for software development. `abelian` adheres to those principles in the following ways:

- Object orientation is used throughout the software. Class methods are categorized as fundamental, derived and miscellaneous. Fundamental methods are those which correspond to mathematical definitions, derived methods are based on the fundamental ones, and the remaining are miscellaneous methods.
- More than 150 tests were written, where approximately 50 are stochastic and the remaining 100 are deterministic. The stochastic tests guard against human errors in proofs and code. The deterministic tests allow for efficient backtesting when new features are added, making sure no functionality breaks.
- The documentation is found in the source code, and is also published on the web. Every function has a description, a list of input and output arguments and types, and one or several examples. Tutorials covering the main features have been written.
- The source code is distributed on Github, and the projected is uploaded to the Python package index, so that users may download it to their computers effortlessly.

7.3.2 Software overview

The library consists of two packages, the main package `abelian` and a subpackage named `abelian.linalg` with linear algebra functions.

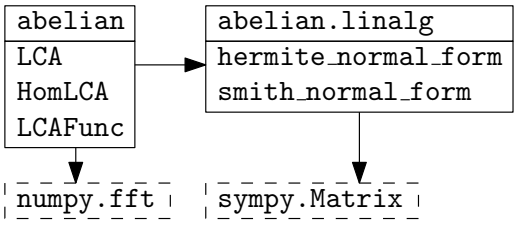
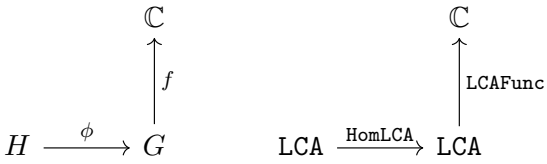


Figure 7.1: A diagram of the main components of `abelian`. Arrows are read as "imports from", class names are capitalized and external software dependencies are contained in dashed boxes.

`abelian.linalg` is a collection of linear algebra functions which are built on the `sympy.Matrix` class, which supports arbitrary precision integer matrices. Here, algorithms such as the SNF and HNF are implemented, along with many utility functions and factorization functions for free-to-free morphisms based on the SNF.

abelian imports functions from abelian.linalg. Many of the class methods in abelian call functions in abelian.linalg to perform matrix computations, this helps separate the "what" from the "how." The typical user will primarily interact with the objects defined in abelian, but working with abelian.linalg directly is also possible if a user is interested in computing matrix factorizations directly. The three main classes defined in abelian are LCA, HomLCA and LCAFunc. Each class represents a mathematical object, as depicted in Diagram (7.1) below.



(7.1)

We now present the main classes, along with a short description and a list of the fundamental methods defined in each class. More details are contained in the software documentation in Appendix B.

- LCA - **Elementary LCA.**
  - Elementary LCAs are implemented up to isomorphism. Both homomorphisms $\phi : H \to G$ and functions $f : G \to \mathbb{C}$ require an LCA instance to be defined. Some LCAs are also FGAs.
  - Fundamental methods: initialization of the identity object, direct sums, equality checking, whether two LCA instances are isomorphic, whether or not an LCA is an FGA, projection of group elements onto LCAs.
- HomLCA - **Homomorphism between two LCAs, i.e.** $\phi : H \to G$.
  - Homomorphisms are created using a matrix representation, a source LCA and a target LCA. If no source/target is given, the software will implicitly assume a free FGA as source and target.
  - Fundamental methods: initialization of the identity morphism, the zero morphism, composition, dual homomorphisms, evaluation, diagonal, horizontal and vertical stacking, element-wise addition and multiplication. Methods for computing the kernel, cokernel, image, coimage.
- LCAFunc - **A function** $f : G \to \mathbb{C}$ **from an LCA to** $\mathbb{C}$.
  - To create an LCAFunc instance, an LCA instance is needed as a domain, and a function representation is needed. A function is typically represented as an evaluation rule, but a representation based on n-dimensional table values is an option if the domain is discrete and compact.

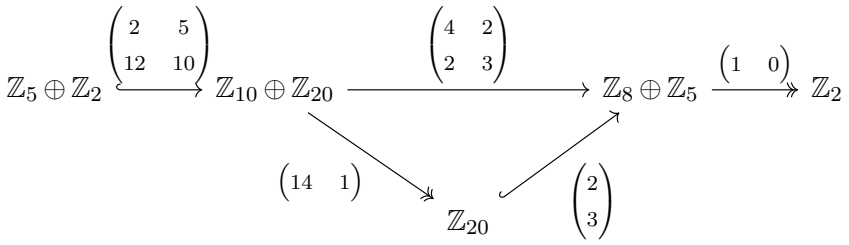
– Fundamental methods: evaluation, shifts, pullbacks, pushforwards, pointwise operators, composition.

7.4 Example 1: Factoring a homomorphism

We now re-do Example 5.17 on page 51, where we found the kernel, cokernel, image and coimage of $\phi : \mathbb{Z}^2 \to \mathbb{Z}_8 \oplus \mathbb{Z}_5$ given by the matrix $A = \left(\begin{smallmatrix} 4 & 2 \\ 7 & 3 \end{smallmatrix}\right)$. We use the `HomLCA` class and the `LCA` class.

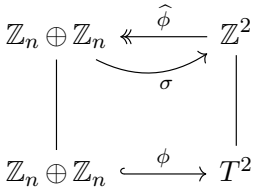
```
# Import the classes and create a homomorphism
from abelian import HomLCA, LCA
target = LCA([8, 5]) # Create Z_8 + Z_5
phi = HomLCA([[4, 2],[7, 3]], target = target)

# Compute cokernel, then remove trivial groups
cokernel = phi.cokernel().remove_trivial_groups()

# Compute image, then remove trivial groups
image = phi.image().remove_trivial_groups()

# Compute coimage, remove trivial, then project
coimage = phi.coimage().remove_trivial_groups()
coimage = coimage.project_to_source()

# Project phi, compute kernel
phi_projected = phi.project_to_source()
kernel = phi_projected.kernel().project_to_source()

print(kernel)
# source: [Z_5, Z_2]     target: [Z_10, Z_20]
# Matrix([[2, 5], [12, 10]])
```

Notice in the code above that every object is immutable—the methods return new `HomLCA` instances instead of modifying existing object instances. We have computed every morphism in the following diagram, which was first introduced on page 53.



7.5 Example 2: Fourier series approximation

In Section 2.2 we approximated the Fourier series coefficients of $f(x) = x$ defined on T. In Example 3.38 on page 24 we found the analytical solution. We will now use `abelian` to obtain approximated Fourier coefficients for $f(x) = x_1 + x_2$ defined on T^2. The example is similar, but two dimensional—notice how easily the code generalizes to several dimensions. The code will be guided by the following diagram, which was introduced as Diagram (6.4) on page 66 in a more general setting.



In the following code, we first define $f(x) = x_1 + x_2$ on T^2, then we define the sampling monomorphism ϕ by $\left(\begin{smallmatrix} 1/10 & 0 \\ 0 & 1/10 \end{smallmatrix}\right)$. We compute the pullback $\phi^*(f)$ and dualize using the DFT. Finally, we must use σ to move from $\mathbb{Z}_n \oplus \mathbb{Z}_n$ to $\mathbb{Z}^2$. A user might specify their own σ, but if no σ is explicitly given the software will use the Voronoi transversal.

```
# Import objects, create function on T^2
from abelian import HomLCA, LCA, LCAFunc
from sympy import Rational, diag
T = LCA(orders = [1], discrete = [False])
func = LCAFunc(lambda x: sum(x), domain = T**2)

# Create homomorphism to sample function
n = 10
Z_n = LCA(orders = [n], discrete = [True])
phi = HomLCA(diag(Rational(1, n), Rational(1, n)),
             target = T**2, source = Z_n**2)

# Sample, dualize
func_sampled = func.pullback(phi)
func_sample_dual = func_sampled.dft()

# Transversal - minimizes distance
func_dual = func_sample_dual.transversal(phi.dual())
```

The code above generated the data which was used to produce Figure 7.2.

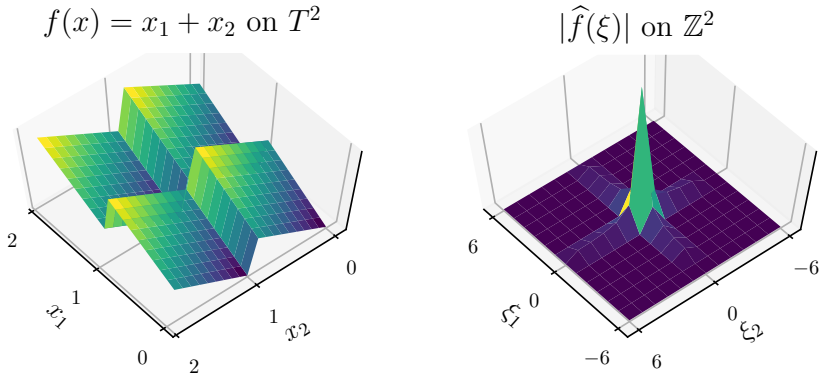
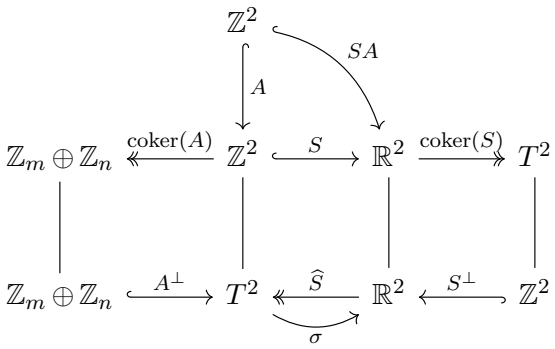


Figure 7.2: The plot on the left shows $f(x) = x_1 + x_2$ on T^2, shown on $\mathbb{R}^2$ to emphasize the periodicity. The plot on the right shows the absolute value of the approximated Fourier coefficients on $\mathbb{Z}^2$. Sampling was done with $n = 10$ orthogonal sample points in each direction.

7.6 Example 3: Hexagonal Fourier analysis

Hexagonal Fourier analysis was introduced in Section 6.9. The following diagram was first introduced on page 71 as Diagram (6.6).



We saw in Example 3.41 on page 26 that the Gaussian is invariant under Fourier transforms. We will use `abelian` to define a Gaussian on $\mathbb{R}^2$, sample, periodize and interpret the results using a Voronoi transversal as per Definition 6.20. The first part of the code is similar to the code introduced in Section 2.3 in the Preview chapter.

```
# Import objects, create function on R^n
from abelian import HomLCA, LCA, LCAFunc, voronoi
R = LCA(orders = [0], discrete = [False])
k = 0.33 # Decay of exponential
func_expr = lambda x: exp(-k*sum(x_j**2 for x_j in x))
func = LCAFunc(func_expr, domain = R**2)

```

```
# Create a homomorphism to sample
hexagonal_generators = [[1, 0.5], [0, sqrt(3)/2]]
phi_sample = HomLCA(hexagonal_generators, target = R**2)

# Create a homomorphism to periodize
n = 10
phi_periodize = HomLCA([[n, 0], [0, n]])
coker_phi_p = phi_periodize.cokernel()

# Move function from R**2 to Z**2 to Z_n**2
func_sampled = func.pullback(phi_sample)
func_periodized = func_sampled.pushforward(coker_phi_p, 25)
```

We have defined the functions and homomorphism and moved the function from $\mathbb{R}^2$ to $\mathbb{Z}_{10} \oplus \mathbb{Z}_{10}$. To interpret the results, we compute the DFT and let abelian compute a Voronoi transversal minimizing the 2-norm.

```
# Move function to dual space, then to T**2
func_dual = func_periodized.dft()
phi_periodize_ann = phi_periodize.annihilator()

# Compute a Voronoi transversal function, interpret on R**2
scaling_factor = phi_sample.det()
sigma = voronoi(phi_sample.dual(), norm_p = 2)
for element in func_dual.domain.elements_by_maxnorm():
    value = func_dual(element) * scaling_factor
    coords_on_R = sigma(phi_periodize_ann(element))
```

In the code above, we use the composition $\sigma \circ A^{\perp}$ to interpret the results. Evaluation of $\widehat{f}(\xi)$ on $\mathbb{R}^2$ directly makes little sense numerically, since $\widehat{f}(\xi)$ is zero almost everywhere on $\mathbb{R}^2$ and the results would be difficult to interpret. Figure 7.3 shows the result of the above code. In Example 3.41 on page 26 we claimed that the Fourier transform of a d-dimensional Gaussian is given by

$$\mathcal{F}\left(\exp(-kx^T x)\right) = (\pi/k)^{d/2} \exp\left(-\pi^2 \xi^T \xi / k\right),$$

and this can be used to verify that the above code is correct.

We summarize this chapter by summarizing the virtues of abelian: it allows the user to work directly with mathematical objects and definitions, sampling and periodizing analytical functions is done in a natural way using group homomorphisms, the default values are sensible, the syntax is readable and the code generalizes easily to higher dimensions.

While the three preceding examples display much of the functionality, there are many features which were omitted in the examples—particularly initialization and binary operations defined on the objects.

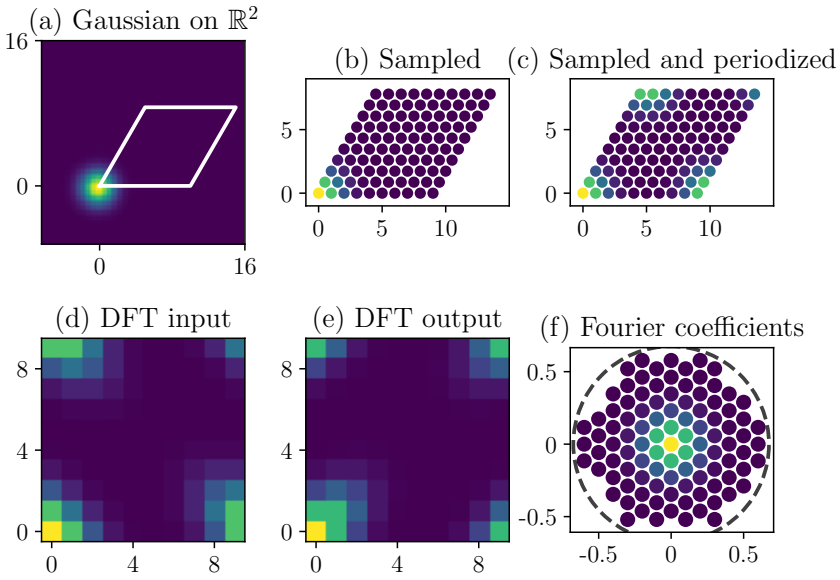


Figure 7.3: All the data for this figure was generated by the code in Section 7.6. (a) The Gaussian and the a fundamental domain for the periodization lattice. (b) Sampled function values. (c) Sampled and periodized function values. (d) Input to the DFT algorithm. (e) Output from the DFT algorithm. (f) Fourier series coefficients as mapped to $\mathbb{R}^2$ by the Voronoi transversal. Notice that the points are all inscribed in a circle.

Chapter 8

Conclusion and further work

We conclude the thesis and suggest further work. The conclusion summarizes the work done, the goals we set out to reach, and how we reached them. To avoid repetition of the Introduction and Preview chapters, the conclusion is kept rather short.

8.1 Conclusion

In this thesis we have outlined the theory of Fourier analysis on the elementary LCAs: $\mathbb{R}$, $T = \mathbb{R}/\mathbb{Z}$, $\mathbb{Z}$, $\mathbb{Z}_n$, and direct sums of these groups. We presented a new software library called `abelian`, which implements high-level mathematical objects for general computations on elementary LCAs. Chapters 3 to 6 consist of theory, and in Chapter 7 we gave a high-level description of the `abelian` software library. Computational examples were introduced in Chapter 2, as well as in Chapter 7.

One of the most important algorithms for computations with homomorphisms between FGAs is the Smith normal form, which we introduced in Section 4.3. We also presented some results not found in literature: the equation solver in Section 5.4, the theorems and algorithms in Section 5.5 for factoring left-free homomorphisms between FGAs, and the algorithm used to compute pushforwards in Sections 6.5 and 6.6.

On a more abstract level than algorithms is the choice of how to represent mathematical objects and which operations to implement. This was discussed in Chapter 7, and the `abelian` library makes use of three classes: `LCA`, `HomLCA` and `LCAFunc`. The classes represent LCAs, homomorphisms between LCAs and functions from an LCA to $\mathbb{C}$, respectively. The versatility of these three classes was demonstrated with several examples at the end of Chapter 7. The most complex example presented was Fourier analysis

on a hexagonal lattice; we discussed theory and surveyed recent research in Section 6.9, and provided code in Section 7.6.

In the end, the goals of the project were reached: `abelian` facilitates computations on elementary LCAs, arbitrary sampling and periodization using group homomorphisms, and implements the mathematical objects and operations associated with the theory. The software has tests, documentation, and is now published on several well-respected services on the web.

8.2 Further work

It is always possible to extend and modify the software. Some use-cases might warrant implementations of algorithms or operations not yet defined. Speed is always a concern when computations are large, and low-level optimizations which might increase speed could be considered. It would be interesting to examine more real-world examples of use-cases.

It might be a good idea to merge the software developed as part of this project with a more mature library. There are at least two reasons why merging with a large, mature software library[1] might be wise: (1) it would immediately make the algorithms and objects available to a large portion of the scientific users of Python, as Sympy is included in most scientific Python distributions, and (2) it would force the software to comply with a set of high standards that are enforced by the library developers.

[1] Sympy would be an example of such a library. Recall that Sympy is a Python library for symbolic mathematics, introduced on page 75.

Bibliography

[Albert. Boggess, 2009] Albert. Boggess (2009). *A first course in wavelets with Fourier analysis*. Wiley, 2nd ed. edition.

[Alchin and Browning, 2014] Alchin, M. and Browning, J. B. (2014). *Pro Python*. Apress, New York, 2nd ed. edition edition.

[Aluffi, 2009] Aluffi, P. (2009). *Algebra: chapter 0*, volume vol. 104 of *Graduate studies in mathematics*. American Mathematical Society, Providence, R.I.

[Birdsong and Rummelt, 2016] Birdsong, J. B. and Rummelt, N. I. (2016). The hexagonal fast fourier transform. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1809–1812.

[Buckheit and Donoho, 1995] Buckheit, J. B. and Donoho, D. L. (1995). WaveLab and Reproducible Research. In *Wavelets and Statistics*, Lecture Notes in Statistics, pages 55–81. Springer, New York, NY. DOI: 10.1007/978-1-4612-2544-7_5.

[C. Gasquet, 1999] C. Gasquet (1999). *Fourier analysis and applications: filtering, numerical computation, wavelets*, volume 30 of *Texts in applied mathematics*. Springer, New York.

[Charles C. Sims, 1994] Charles C. Sims (1994). *Computation with finitely presented groups*, volume 48 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, Cambridge.

[Christopher. Hammond, 2009] Christopher. Hammond (2009). *The Basics of Crystallography and Diffraction*. International Union of Crystallography Texts on Crystallography, 12. OUP Oxford, Oxford, 3rd ed. edition.

[Dasgupta, 2008] Dasgupta, S. (2008). *Algorithms*. McGraw Hill.

[David Steven. Dummit, 2004] David Steven. Dummit (2004). *Abstract algebra*. Wiley, Hoboken, N.J, 3rd ed. edition.

[Derek F. Holt, 2005] Derek F. Holt (2005). *Handbook of computational group theory*. Discrete mathematics and its applications. Chapman & Hall/CRC, Boca Raton, Fla.

[Ehrhardt, 1993] Ehrhardt, J. C. (1993). Hexagonal fast Fourier transform with rectangular output. *Signal Processing, IEEE Transactions on*, 41(3):1469–1472.

[Golub and Van Loan, 2012] Golub, G. H. and Van Loan, C. F. (2012). *Matrix Computations*. Johns Hopkins University Press, Baltimore, fourth edition edition edition.

[Harold. Simmons, 2011] Harold. Simmons (2011). *An introduction to category theory*. Cambridge University Press, Cambridge.

[Hewitt, 1963] Hewitt, E. (1963). *Abstract harmonic analysis: Vol. 1 : Structure of topological groups, integration theory, group representations*, volume Vol. 1 of *Die Grundlehren der mathematischen Wissenschaften*. Springer, Berlin.

[Jäger and Wagner, 2009] Jäger, G. and Wagner, C. (2009). Efficient parallelizations of Hermite and Smith normal form algorithms. *Parallel Computing*, 35(6):345–357.

[Kristian. Rother, 2017] Kristian. Rother (2017). *Pro Python Best Practices: Debugging, Testing and Maintenance*. Imprint: Apress, Apress.

[Ledermann, 1996] Ledermann, W. (1996). *Introduction to group theory*. Longman mathematics series. Addison Wesley Longman, Harlow, 2nd ed. edition.

[Lloyd N. Trefethen, 1997] Lloyd N. Trefethen (1997). *Numerical linear algebra*. Society for Industrial and Applied Mathematics.

[Mac Lane, 1998] Mac Lane, S. (1998). *Categories for the working mathematician*, volume 5 of *Graduate texts in mathematics*. Springer, New York, 2nd ed. edition.

[Madisetti, 2009] Madisetti, V. (2009). *Digital Signal Processing Fundamentals*. Electrical Engineering Handbook. CRC Press. DOI: 10.1201/9781420046076.

[Mersereau and Speake, 1983] Mersereau, R. and Speake, T. (1983). The processing of periodically sampled multidimensional signals. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 31(1):188–194.

[Mersereau, 1979] Mersereau, R. M. (1979). The processing of hexagonally sampled two-dimensional signals. *Proceedings of the IEEE*, 67(6):930–949.

[Meurer et al., 2017] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103.

[Munthe-Kaas, 2016] Munthe-Kaas, H. Z. (2016). Groups and Symmetries in Numerical Linear Algebra. In *Exploiting Hidden Structure in Matrix Computations: Algorithms and Applications*, Lecture Notes in Mathematics, pages 319–406. Springer, Cham. DOI: 10.1007/978-3-319-49887-4_5.

[Nicholas. Loehr, 2014] Nicholas. Loehr (2014). *Advanced Linear Algebra*. Discrete Mathematics and Its Applications. Taylor and Francis.

[R. E. Edwards, 1979] R. E. Edwards (1979). *Fourier series: a modern introduction : 1*, volume 1 of *Graduate texts in mathematics*. Springer, New York, 2nd ed. edition.

[Ramakrishnan, 1999] Ramakrishnan, D. (1999). *Fourier analysis on number fields*, volume 186 of *Graduate texts in mathematics*. Springer, New York.

[Ramalho, 2015] Ramalho, L. (2015). *Fluent Python*. O'Reilly, 1st edition. edition.

[Reiter, 1968] Reiter, H. (1968). *Classical harmonic analysis and locally compact groups*. Oxford mathematical monographs. Clarendon Press, Oxford.

[Robinson, 2017] Robinson, D. (2017). The incredible growth of python. `https://stackoverflow.blog/2017/09/06/incredible-growth-python/`. Accessed: 2017-09-21.

[Roman, 2005] Roman, S. (2005). *Advanced linear algebra*, volume 135 of *Graduate texts in mathematics*. Springer, New York, 2nd ed. edition.

[Rudin, 1967] Rudin, W. (1967). *Fourier analysis on groups*, volume 12 of *Interscience tracts in pure and applied mathematics*. Interscience, New York.

[Sloan and Joe, 1994] Sloan, I. H. and Joe, S. (1994). *Lattice Methods for Multiple Integration*. Clarendon Press, Oxford : New York, 1 edition edition.

[Strang, 1976] Strang, G. (1976). *Linear algebra and its applications*. Academic Press, San Diego, 3nd ed. edition.

[Strang, 1986] Strang, G. (1986). *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, Mass.

[Strang, 1993] Strang, G. (1993). The Fundamental Theorem of Linear Algebra. *The American Mathematical Monthly*, 100(9):848–855.

[VanderPlas, 2017] VanderPlas, J. (2017). *The Unexpected Effectiveness of Python in Science*. PyCon 2017. https://speakerdeck.com/jakevdp/the-unexpected-effectiveness-of-python-in-science.

[Vince and Zheng, 2007] Vince, A. and Zheng, X. (2007). Computing the Discrete Fourier Transform on a Hexagonal Lattice. *Journal of Mathematical Imaging and Vision*, 28(2):125–133.

[Vretblad, 2003] Vretblad, A. (2003). *Fourier analysis and its applications*, volume 223 of *Graduate texts in mathematics*. Springer, New York.

[Walt et al., 2011] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30.

[Xiangjian He and Wenjing Jia, 2005] Xiangjian He and Wenjing Jia (2005). Hexagonal Structure for Intelligent Vision. In *Information and Communication Technologies, 2005. ICICT 2005. First International Conference on*, pages 52–64. IEEE.

[Zheng and Gu, 2014] Zheng, X. and Gu, F. (2014). Fast Fourier Transform on FCC and BCC Lattices with Outputs on FCC and BCC Lattices Respectively. *Journal of Mathematical Imaging and Vision*, 49(3):530–550.

Appendices

Appendix A

Source code

Below are some of the algorithms used in `abelian`. The code below constitutes a small part of the complete code. The total line count of `abelian` is over 5000 lines, so to keep the length of the thesis at a reasonable level, full source code is not given. For a complete code listing with comments, see github.com/tommyod/abelian.

Hermite normal form

This is an implementation of Algorithm 1 for the Hermite normal form.

```
def hermite_normal_form(A):
    """
    Compute U and H such that A*U = H.
    """

    # Get size and set up matrices U and H
    m, n = A.shape
    j = 0
    H = A.copy()
    U = Matrix.eye(n)

    # Iterate down the rows of the matrix
    for i in range(0, m):

        # If every entry to the right is a zero, no pivot will
        # be found for this row and we move on to the next one.
        if H[i, j:] == H[i, j:] * 0:
            continue

        # Create zeros to the right of the pivot H[i, j]
        for k in range(j + 1, n):

            # Skip the column if the element is zero
            # In this case the column index j in not incremented
            if H[i, k] == 0:
                continue

```

```
            # Apply the 'elementary hermite transform' to the
            # columns of H, ignoring the top i rows of H as
            # they are identically zero. The implementation
            # of the 'elementary hermite transform' does not
            # explicitly compute matrix products.
            # Equivalent to right-multiplication with
            # Matrix([[a, -s/g],
            #         [b, r/g]])

            # Extended Euclidean algorithm,
            # i.e. r*a + s*b = g = gcd(r, s)
            r, s = H[i, j], H[i, k]
            a, b, g = gcdex(r, s)

            # Apply the matrix product of H and U
            H[i:, j], H[i:, k] = (a * H[i:, j] + b * H[i:, k],
                                  -(s/g)*H[i:, j]+(r/g)*H[i:, k])
            U[:, j], U[:, k] = (a * U[:, j] + b * U[:, k],
                                -(s/g)*U[:, j]+(r/g)*U[:, k])

        # Make sure the pivot element is positive.
        # Some savings achieved by realizing that the
        # first i rows of H are identically zero --
        # thus no multiplication is needed.
        if H[i, j] < 0:
            H[i:, j] = -H[i:, j]
            U[:, j] = -U[:, j]

        # Making all elements to the left of the
        # pivot H[i, j] smaller than the pivot and
        # positive using division algorithm transform
        for k in range(0, j):
            # Compute quotient in the division algorithm,
            # subtracting the quotient times H[:, j]
            # leaves a positive remainder
            a = H[i, k] // H[i, j]
            H[:, k] = H[:, k] - a * H[:, j]
            U[:, k] = U[:, k] - a * U[:, j]

        # Increment j (the column index).
        # Break if j is out of dimension
        j += 1
        if j >= n:
            # j is out of dimension, break
            break

    return U, H
```

Smith normal form

This is an implementation of Algorithm 2 for the Smith normal form.

```
def smith_normal_form(A, compute_unimod = True):
    """
    Compute U,S,V such that U*A*V = S.
    """

    # Get size and set up the unimodular matrices U and V
    m, n = A.shape
```

```
    min_m_n = min(m, n)
    S = A.copy()
    if compute_unimod:
        U, V = Matrix.eye(m), Matrix.eye(n)

    def row_col_all_zero(matrix, f):
        """
        Are all entries to the right of and below `f` zero?
        """
        for entry in matrix[f, f + 1:]:
            if entry != 0:
                return False
        for entry in matrix[f + 1:, f]:
            if entry != 0:
                return False
        return True

    # Main loop, iterate over all sub-matrices to reduce
    f = 0
    while f < min_m_n:

        # While there are non-zero elements to reduce in
        # row/column f and the diagonal element is not positive
        while not (row_col_all_zero(S, f) and S[f, f] >= 0):

            # Find index pair of minimum non-zero entry
            # (in absolute value) in the sub-matrix S[f:, f:].
            inds = ((i, j) for j in range(f, n)
                    for i in range(f, m))

            key_val_pairs = ((index, abs(S[index])) for index
                             in inds if abs(S[index]) != 0)

            (i, j), min_val = min(key_val_pairs,
                                  key=lambda k: k[1])

            # Permute S to move the minimal
            # element to the pivot location
            S[f:, j], S[f:, f] = S[f:, f], S[f:, j]
            S[i, f:], S[f, f:] = S[f, f:], S[i, f:]
            if compute_unimod:
                V[:, j], V[:, f] = V[:, f], V[:, j]
                U[i, :], U[f, :] = U[f, :], U[i, :]

            # If the freshly permuted pivot
            # is negative, make it positive
            if S[f, f] < 0:
                S[f:, f] = -S[f:, f]
                if compute_unimod:
                    V[:, f] = -V[:, f]

            # Reduce row f so every entry
            # is smaller than pivot
            for k in range(f + 1, n):
                if S[f, k] == 0:
                    continue

                # Subtract a times column
                # f from column k
                a = S[f, k] // S[f, f]
                S[f:, k] = S[f:, k] - a * S[f:, f]
                if compute_unimod:
```

```
                    V[:, k] = V[:, k] - a * V[:, f]

            # Reduce column f so every
            # entry is smaller than pivot
            for k in range(f + 1, m):
                if S[k, f] == 0:
                    continue

                # Subtract a times row f from row k
                a = S[k, f] // S[f, f]
                S[k, f:] = S[k, f:] - a * S[f, f:]
                if compute_unimod:
                    U[k, :] = U[k, :] - a * U[f, :]

        f += 1

    # Enforce divisibility criterion using
    # the 'divisibility transformation' matrices.
    for f in range(min_m_n):
        for k in range(f + 1, min_m_n):

            # Divisibility criterion is fulfilled
            if mod(S[k, k], S[f, f]) == 0:
                continue

            # S[f, f] does not divide S[k, k]
            r, s = S[f, f], S[k, k]
            a, b, c = gcdex(r, s)
            S[f, f], S[k, k] = c, (r * s) // c

            # Modify unimodular transformation matrices,
            # but without explicitly multiplying matrices
            if compute_unimod:
                V[:, f], V[:, k] = (V[:, f] + V[:, k],
                                    -b * (s / c) * V[:,f] +\
                                    a * (r / c) * V[:, k])
                U[f, :], U[k, :] = (a * U[f, :] + b * U[k, :],
                                    -(s / c) * U[f, :] +\
                                    (r / c) * U[k, :])

    if compute_unimod:
        return U, S, V
    else:
        return S
```

Solving $\phi(x) = g$ when ϕ is left-free

This is an implementation of Algorithm 3, which solves an equation with a left-free homomorphism.

```
def solve(A, b, p = None):
    """
    Solve eqn Ax = b mod p over Z.
    """

    # If no orders are supplied by the user,
    # set the orders to zero,
    # i.e. infinite order or free-to-free.
```

```
    if p is None:
        m, n = b.shape
        p = Matrix(m, n, lambda i, j: 0)

    # Verify that the dimensions are correct
    (A_rows, A_cols) = A.shape
    (b_rows, b_cols) = b.shape
    (p_rows, p_cols) = p.shape
    if not (A_rows == b_rows == p_rows):
        raise ValueError('Dimension mismatch.')

    # Find the kernel of the
    # projection onto the space Z_`p`
    ker_pi = remove_zero_columns(diag(*p))

    # Stack A | ker(pi) | b
    joined_A_D_b = A.row_join(ker_pi).row_join(b)

    # Compute ker( A | ker(pi) | b)
    # using the free-to-free kernel
    kernel = free_kernel(joined_A_D_b)

    # The solution must be a linear combination
    # of the columns of ker( A | ker(pi) | b)
    # such that the resulting vector has a -1
    # in the bottom entry.

    # Remove all columns with zero in the bottom entry
    m, n = kernel.shape
    col_indices = [j for j in range(n)
                   if kernel[-1, j] == 0]
    kernel = remove_cols(kernel, col_indices)

    # Return None if the kernel is empty
    m, n = kernel.shape
    if n == 0:
        return None

    # Iteratively 'collapse' the columns using the
    # extended euclidean algorithm till the result is 1
    m, n = kernel.shape
    while n > 1:
        # Compute the new column
        # from the first two current ones
        f, g = kernel[-1, 0], kernel[-1, 1]
        (s, t, h) = gcdex(f, g)  # s*f + t*g = h.
        new_col = s * kernel[:, 0] + t * kernel[:, 1]

        # If there are only two columns,
        # we have found the kernel
        if n == 2:
            kernel = new_col
            break

        # Delete current columns and insert the new one
        kernel = remove_cols(kernel, [0, 1])
        kernel = new_col.row_join(kernel)

        # Calculate new n value for the while-loop
        (m, n) = kernel.shape

    # Find shape of input, since shape of output depends on it
```

```
    (m, n) = A.shape

    # Make sure that the bottom row is -1 or 1.
    # It will always be 1 if the above while loop initiated,
    # but if it never initiated then value could be -1
    if kernel[-1, 0] not in [1, -1]:
        return None

    # The solution to the problem is contained the first
    # n rows of the kernel, which is a column vector.
    # Multiply by -1 if needed to
    # make sure the bottom entry is -1
    if kernel[-1, 0] == 1:
        return -kernel[:n, 0]
    else:
        return kernel[:n, 0]
```

Elements in $\mathbb{Z}^r$ with given max-norm

This is an implementation of Algorithm 5 for generating group elements in $\mathbb{Z}^r$ with a given maximum norm.

```
def elements_of_maxnorm(free_rank, maxnorm_value):
    """
    Yield every element of Z^r such
    that max_norm(element) = maxnorm_value.
    """
    # Special case when the norm is 0,
    # yield the (0, 0, ...) element
    if maxnorm_value == 0:
        yield tuple([0] * free_rank)
        return

    # There are two 'walls' per hypercube, front and back
    for wall in range(free_rank):

        # In each hypercube, the boundaries
        # must shrink, two at a time
        border_reduced = [1] * wall + [0] * (free_rank-wall-1)

        # The arguments into the
        # cartesian product for the hypercube
        prod_arg = [range(-maxnorm_value+k, maxnorm_value+1-k)\
                    for k in border_reduced]

        # Take cartesian products along the boundaries
        # of the r-dimensional hypercube.
        # Yield from opposite sides of the hypercube
        for b_element in itertools.product(*prod_arg):
            start, end = b_element[:wall], b_element[wall:]
            yield start + (maxnorm_value,) + end
            yield start + (-maxnorm_value,) + end
```

Elements in $\mathbb{Z}_\mathbf{p}$ with given max-norm

This is an extension of Algorithm 5. The algorithm below generates group elements in $\mathbb{Z}_\mathbf{p}$ with a given max-norm.

```
def elements_of_maxnorm_FGA(orders, maxnorm_value):
    """
    Yield every element of Z_`orders` such
    that max_norm(element) = maxnorm_value.
    """

    # Special case when the norm is zero,
    # yield (0, 0, ...) and terminate
    if maxnorm_value == 0:
        yield tuple([0] * len(orders))
        return

    # Will be used in the loop,
    # so we compute it out-of-loop here
    dimension = len(orders)

    # The 'wall' is the dimension held constant
    for wallnum, dim in enumerate(orders):

        # If the wall is outside the dimension, skip it
        if (dim != 0) and maxnorm_value > (dim // 2):
            continue

        # Set up the cartesian product argument,
        # making sure to remove boundary elements
        # so they are not yielded twice
        border_reduced = [1]*wallnum + [0]*(dimension-wallnum-1)
        prod_arg = [range(-maxnorm_value+k, maxnorm_value+1-k) \
                    for k in border_reduced]

        # The dimensions that are not constant
        non_const_dims = orders[:]
        non_const_dims.pop(wallnum)

        # Go through every argument in the cartesian
        # product, and reduce the iterator if it's
        # partially outside of the order
        for i in range(len(prod_arg)):
            iterator = prod_arg[i]
            order = non_const_dims[i]

            # If the order is finite, we might be
            # able to reduce the cartesian product
            # by a significant amount
            if order != 0:
                start, stop = iterator.start, iterator.stop
                prod_arg[i] = range(max(-order // 2 + 1, start),
                                    min(order // 2 + 1, stop))

        # Go through the Cartesian product / hypercube
        for prod in itertools.product(*prod_arg):

            # The first and last part of the element
            first = mod(prod[:wallnum], non_const_dims[:wallnum])
            last = mod(prod[wallnum:], non_const_dims[wallnum:])

```

```
57                mid1 = mod(maxnorm_value, dim)
58                mid2 = mod(-maxnorm_value, dim)
59                yield first + (mid1, ) + last
60                if middle1 != middle2:
61                    yield first + (mid2, ) + last
```

Appendix B

Software documentation

Below is part of the software documentation for `abelian`. The included `.pdf` file was automatically generated using the Python documentation generator `sphinx`. A full version, which includes API reference for every function and method, is available online at abelian.readthedocs.io. Due to the size of the full documentation, which is around 80 pages, only the most essential part is included here.

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

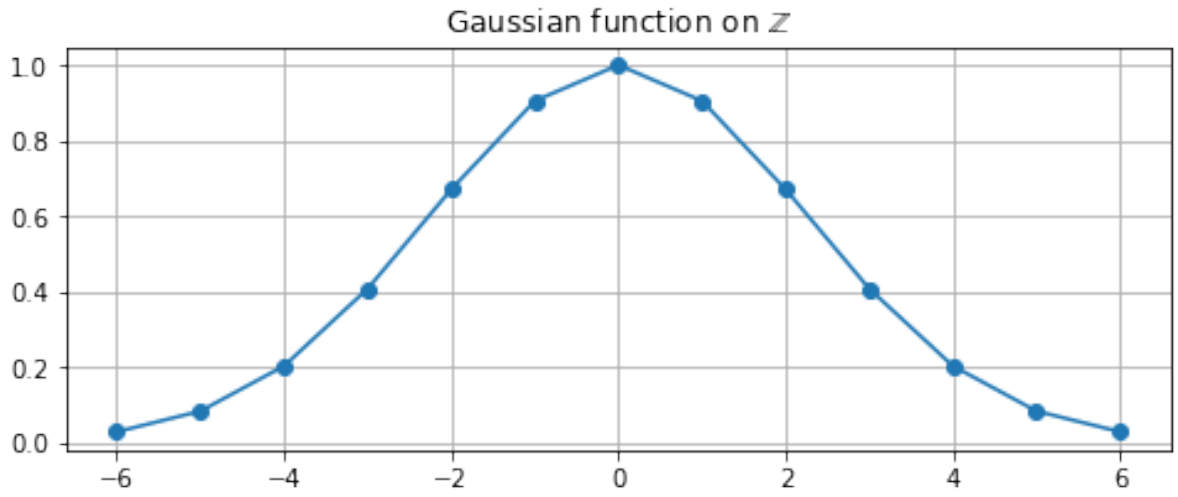
A function $f \in \mathbb{C}^G$ is callable. To call (i.e. evaluate) a function, pass a group element.

```
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

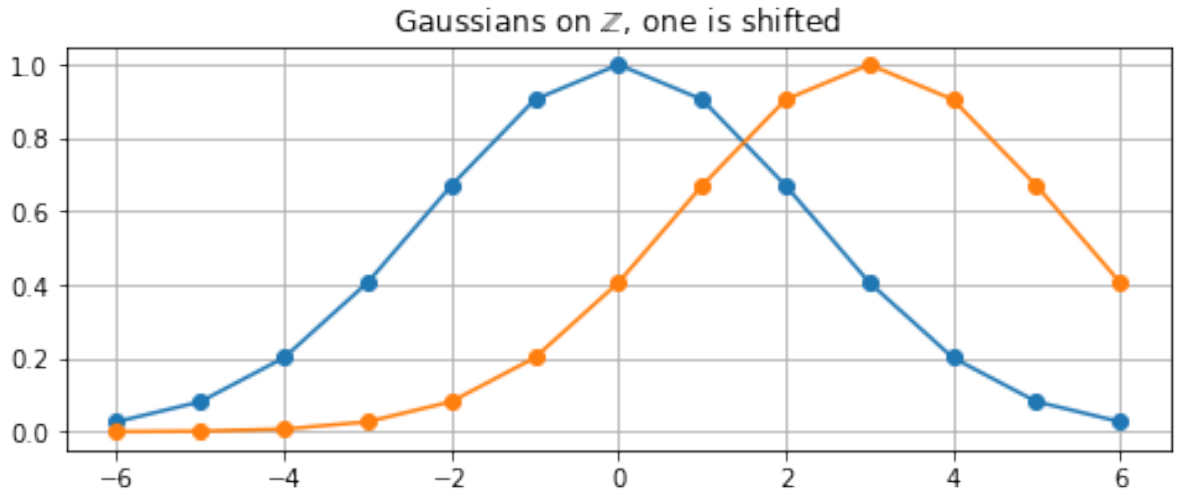
Let $f : G \to \mathbb{C}$ be a function. The shift operator (or translation operator) S_h is defined as

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

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)
```

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

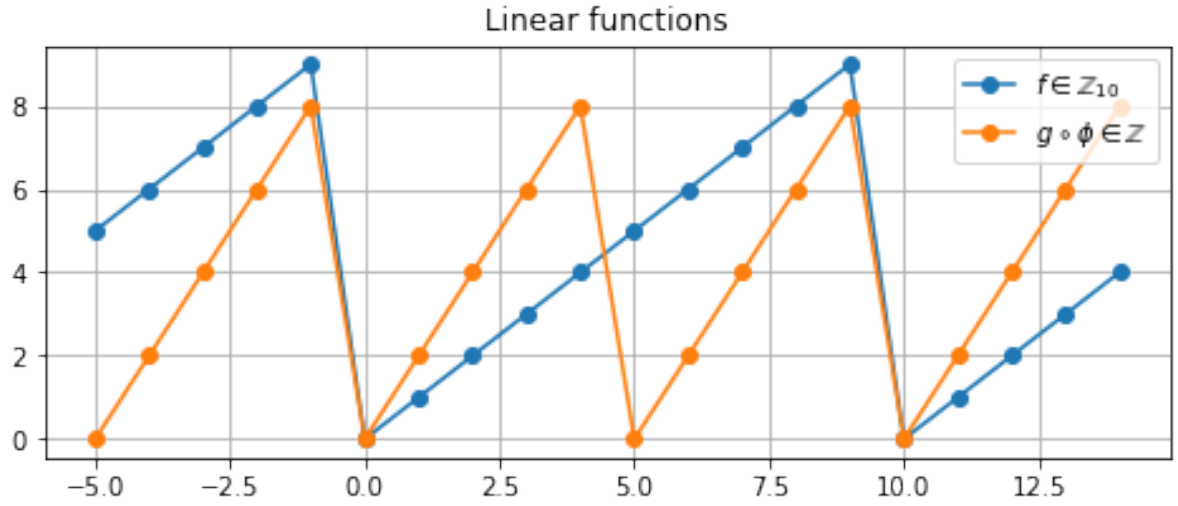
$$(2) : \mathbb{Z} \to \mathbb{Z}_{10}$$

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

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

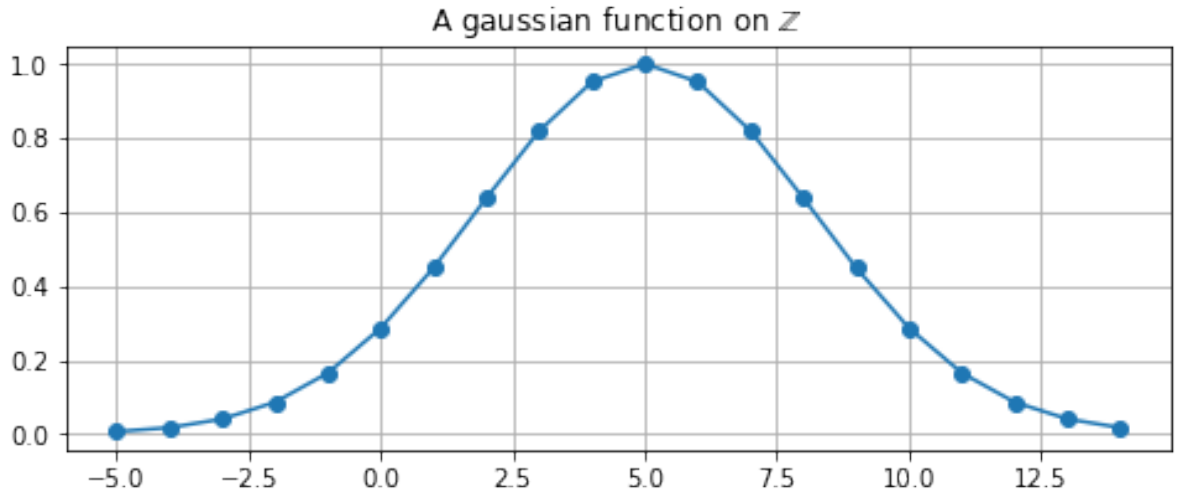
$$(\phi_*(f))(g) := \sum_{k \in \ker \phi} f(k + h), \quad \phi(g) = h$$

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()
```

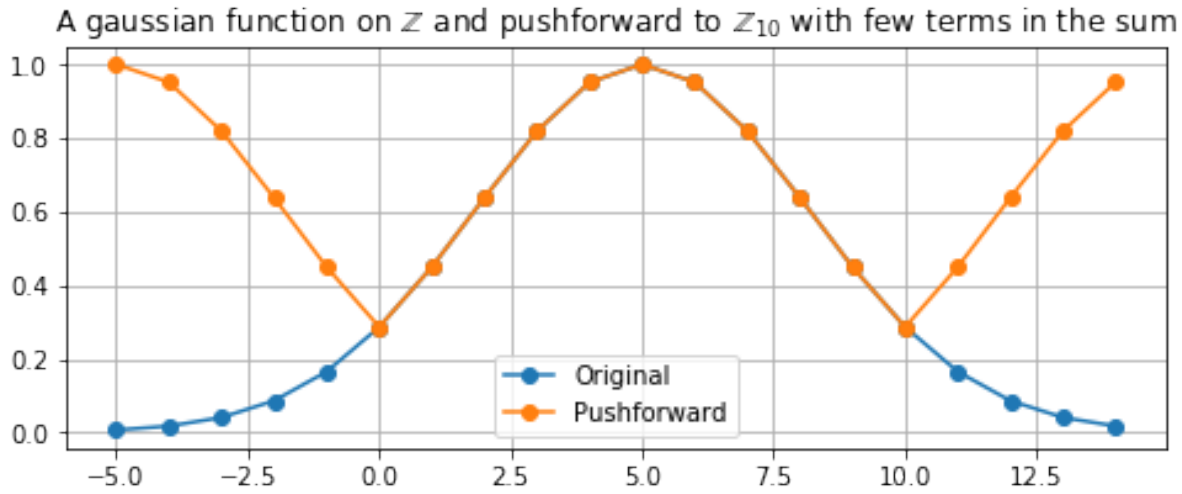



```
In [10]: # Use a pushforward to periodize the function
         phi = HomLCA([1], target = [10])
         show(phi)
```

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

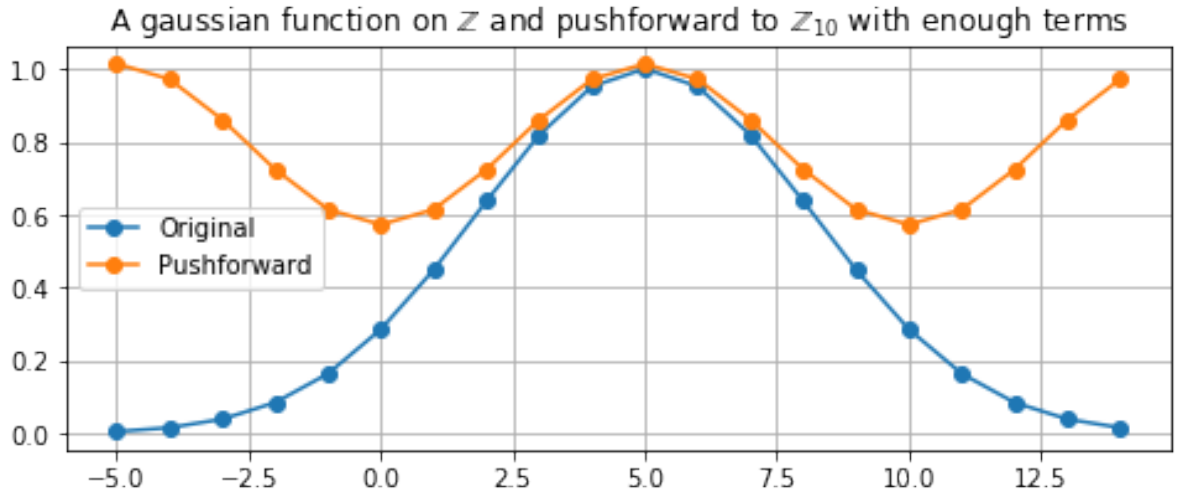
First we do a pushforward with only one term. **Not enough terms are present** in the sum to capture what the pushforward would look like if the sum went to infinity.

```
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)
```

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

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

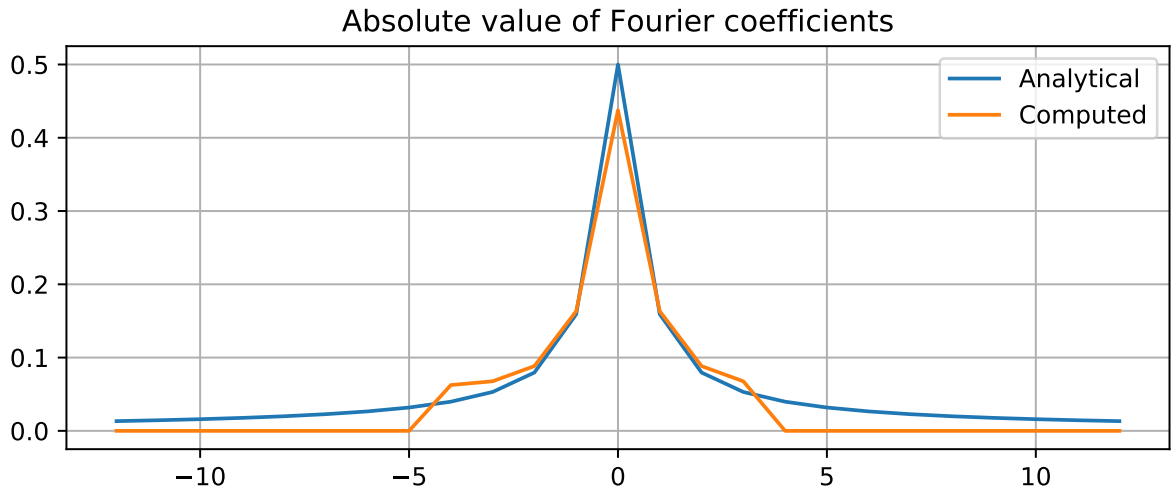
Let us compare this result with the analytical solution, which is

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

```
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.

This concludes Appendix B. Every class, method and function is documented with examples. Due to the length, this part of the documentation is omitted. It can be found online at abelian.readthedocs.io/, and the `.pdf` documentation is found at media.readthedocs.org/pdf/abelian/latest/abelian.pdf.