ATOMICSETS.JL: THE CALCULUS OF SUPPORT FUNCTIONS IN JULIA

by

MIA KRAMER

B.Sc., The University of British Columbia, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

AtomicSets.jl: The Calculus of Support Functions in Julia

submitted by Mia Kramer in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Michael Friedlander, Professor, Computer Science and Mathematics, UBC
Supervisor

Ronald Garcia, Associate Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

Atomic sets are a powerful abstraction of sparsity to more general notions of structure. We present here a software implementation of atomic sets and support functions in the Julia programming language. Previous work in compressed sensing gives us a calculus for support functions under sums and applications of linear maps to atomic sets. We discuss this calculus, how it is reflected in software, and its applications. Emphasis is placed on how the calculus can be represented in a way that allows for both ease of use as a software library and high performance in applications, and how the Julia programming language allows this.

Lay Summary

Working with structure is an essential component of making optimization problems feasible and fast. This work describes a software implementation of a general formulation of structure that one can impose on the solutions of some classes of optimization problems. The method of dealing with structure described here is to assign a set of "atoms", which taken together define a particular notion of structure, to the solution of a problem. The software package works generically with these sets, and makes an effort to reflect the mathematics in the way one uses the package.

Preface

This work builds on previous work by Michael Friedlander, Zhenan Fan, and Babhru Joshi. In particular, it is a rewrite of the AtomicOpt.jl package. AtomicSets.jl is a collaborative software project by Michael Friedlander, Zhenan Fan, Babhru Joshi, and myself. The analysis of embedding the calculus of support functions and the representations of atomic sets in the Julia programming language described here are original works by me.

Table of Contents

Abstract iii

Lay Summary iv

Preface v

Table of Contents vi

List of Figures ix

List of Definitions x

List of Algoithms xi

List of Examples xii

Acknowledgements xiii

Dedication xiv

1 Introduction 1

1.1 Convexity 3

2 Atomic Sets and Decompositions 7

2.1 Atomic Sets 7

2.2 Tools of the Trade 8

2.2.1 Support Functions 9

2.2.2 Exposing Faces 10

2.2.3 Gauge Functions 12

2.2.4 Alignment 13

2.3 The Calculus of Support Functions 14

3 The Level-Set Method and Deconvolutions 17
3.1 Outline of the Algorithm 17
3.2 The Level Set Method 18
3.2.1 Dual Conditional Gradient Method for Projection 20
3.3 Polar Deconvolutions 21
4 The Julia Programming Language 22
4.1 Julia 22
4.1.1 Julia's Type System & Multiple Dispatch 23
4.1.2 Other Performance Considerations 26
4.2 Representing Mathematics in Julia 27
5 AtomicSets.jl: The Calculus of Atomic Sets in Julia 29
5.1 Basic Set Representations 29
5.1.1 Sets 29
5.1.2 Atoms 31
5.1.3 Faces 32
5.1.4 Operations 34
5.2 Composite Set Representations 34
5.2.1 Mapped Sets 34
5.2.2 Sum of Sets 34
5.3 Linear Maps 36
5.3.1 Maps in their Natural Spaces 36
5.3.2 Flattenings 37
5.4 The Level Set Method 37

6 Examples 40

7 Future Directions 49

7.1 Using Value Types To Check Set Sizes 49

7.2 Representing Maps with Tensors 50

7.3 Atom Types as Computation Graphs 51

Bibliography 53

A Index 55

B Notation 56

List of Figures

1.1 An atomic decomposition 2
1.2 A convex set, showing some collections of points and their convex combinations 4
1.3 A convex set and some of its supporting hyperplanes 5
1.4 A set of three points and their convex hull 6
2.1 An atomic decomposition of some data 10
2.2 An exposed face of the one ball 11
2.3 $\mathbb{B}_1$ and its gauge visualized 13
2.4 A set transformed under a linear map, with support and gauge labelled 15
3.1 Illustration of the Level-Set Method 19
5.1 Call Graph of the Level Set Method 39
6.1 Frank-Wolfe in AtomicSets.jl 41
6.2 Linear Programming in AtomicSets.jl 42
6.3 Star-Galaxy Separation Using the Level Set Method 44
6.4 Chessboard Separation Using the Level Set Method 46
6.5 Sparse Signal Multiplexing Results 48

List of Definitions

1.1 Convex Set 3
1.2 Supporting Hyperplane 3
1.3 Convex Hull 4
2.1 Support Function 9
2.2 Exposed Face 10
2.3 Gauge 12
2.4 Dual Norm 13

List of Algoithms

3.1 Level-Set Algorithm 19
3.2 Dual Conditional Gradient Method for Projection 20
6.1 Frank-Wolfe 40

List of Examples

2.1 Sparsity 7
2.2 Low-Rankness 7
2.3 Sparsity in Frequency 8
2.4 Sparsity, Continued 9
2.5 Low-Rankness, Continued 9
2.6 Sparsity, Continued 11
2.7 Low-Rankness, Continued 11
2.8 Sparsity, Continued 12
2.9 Low-Rankness, Continued 12
6.1 Frank-Wolfe 40
6.2 Linear Programming 41
6.3 Star-Galaxy Separation 42
6.4 Chessboard Separation 43
6.5 Sparse Signal Multiplexing 45

Acknowledgements

First, I'm extremely grateful to my advisor Professor Michael Friedlander for all his guidance and enthusiasm. His support and knowledge helped me succeed in doing a master's in a field I was new to, and his excitement made doing it enjoyable.

I would also like to thank Professor Ronald Garcia for agreeing to be in my examining committee and providing valuable feedback and a different perspective on this thesis. I am also grateful to my colleague Zhenan Fan for all the help and many discussions.

Finally, I would never have gotten the opportunity to even start without the years of support, help, and love from my parents Anna Leininger and Greg Kramer, from Jay Lowenstein, and from my friends and family.

Dedication

To Bob Leininger, for always sharing in my excitement for learning.

1.
Introduction

Finding and enforcing structure in optimization has been the primary tool for developing efficient algorithms with good guarantees. Given only the most general form of constrained optimization—given $f : \mathbb{R}^n \to \mathbb{R}$, finding the minimizer x^* of $f(x)$ subject to the constraint $x \in \mathcal{A}$, $\mathcal{A} \subseteq \mathbb{R}^n$—one can do almost nothing. However, by identifying properties of the objective and the constraints, we can make the problem feasible. In this work, we describe a method for exploiting expected structure in the solution in order to accelerate finding said solutions. By finding larger sets of useful properties, we increase the number of practically solvable problems.

Structured optimization applies this principle to the variable x: by identifying and using expected structure in the minimizer in addition to the objective and constraints, we can achieve results which better match this expected structure and sometimes get them faster. The core tool we use for describing structure, atomic decompositions[1,2], generalize earlier work in compressed sensing[3–8] by assigning sets of ATOMS which define a particular desired structure. This formulation of structure and generalization of sparsity provides a convex geometry which allows for efficient algorithms with comprehensible geometric meaning.

The set of atoms from which we draw (the ATOMIC SET) is notated $\mathcal{A}$. Given this, we define an ATOMIC DECOMPOSITION of some data $x \in \mathcal{X}$ relative to $\mathcal{A}$ as a set of coefficients c_a such that

$$\begin{aligned} x &= \sum_{a \in \mathcal{A}} c_a \, a \\ c_a &\geq 0 \quad \forall a \in \mathcal{A}. \end{aligned} \tag{1.1}$$

If few of the coefficients in the decomposition are nonzero, we say that data is sparse relative to $\mathcal{A}$. We are interested in sets where sparsity in this decomposition implies structure in the data being decomposed. For example, sparsity with respect to basis vectors and their negation ($\mathcal{A} = \left\{ \pm e_i \right\}$) gives traditional sparsity (few entries in the vector being decomposed are nonzero).

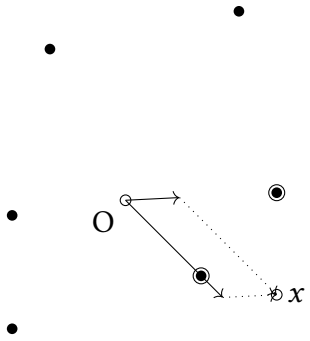


Figure 1.1 An atomic decomposition. Atoms are filled dots, and the data is x. Atoms in the decomposition are circled. Note that generally decompositions are not unique, which is trivially true for this set as there are more atoms than there are dimensions. "O" in this and all following diagrams indicates the origin.

We present here a software implementation of the abstraction of atomic sets and support functions in the Julia programming language[9]. The mathematical structure here is general and supports a common set of tools and so should software implementing it: we want the structure, generality, and efficiency of a mathematical framework to be reflected in software using an implementation of it. Julia is a good implementation language for such a framework for a few reasons: its syntax resembles mathematical notation, its evaluation model reliably produces efficient code under easy-to-achieve conditions, and its core paradigm of multiple dispatch lends itself naturally to dealing with generic operations over a collection of types in an extensible way.

In addition, this software implements the level-set method[10] and the polar deconvolution algorithm[11] using the tools and abstractions described here and also implemented in the package.

In Chapter 2 we discuss the atomic set formulation of structure. Then in Chapter 3 we discuss the two aforementioned algorithms. In Chapter 4 we describe the basics of the Julia programming language and how it is suited to representing mathematics. In Chapter 5 we discuss the implementation of the atomic sets abstraction in Julia and some limitations. In Chapter 6 we show some examples of using the package to perform various optimization tasks. Finally, in Chapter 7 we discuss possible future directions and improvements for the package.

1.1. Convexity

To understand atomic sets and the algorithms built on them, one must understand convexity. This section covers this background material. This section draws heavily from [12], which discusses it in more detail.

Most of us are introduced to the notion of convexity in grade school geometry, where a convex polygon is one without any caves or dents. We can make this definition more rigorous and extend it to much more general spaces:

Definition 1.1 (Convex Set)

A set $\mathcal{X} \subseteq \mathcal{V}$ (for a Euclidean space $\mathcal{V}$) is convex if and only if for every collection of points $(a_1, a_2, ..., a_k) \in \mathcal{X}$, their convex combination

$$\sum_i^k a_i c_i, \quad \text{where } c_i \geq 0 \text{ and } \sum_i^k c_i = 1 \tag{1.2}$$

is also contained within $\mathcal{X}$.

An example of a convex set with some convex combinations of elements is depicted in Figure 1.2.

Definitions for conic and affine sets are achieved by changing the restriction on the coefficients. In particular,

- Conic sets require $c_i \geq 0$, but the sum is allowed to be any positive number.
- Affine sets require $\sum_i c_i = 0$, but coefficients are allowed to be negative.
- Removing all restrictions on coefficients gives linear spaces.

Convex sets can be defined explicitly by describing all points within the set, but they can also be defined implicitly by the set of SUPPORTING HYPERPLANES:

Definition 1.2 (Supporting Hyperplane)

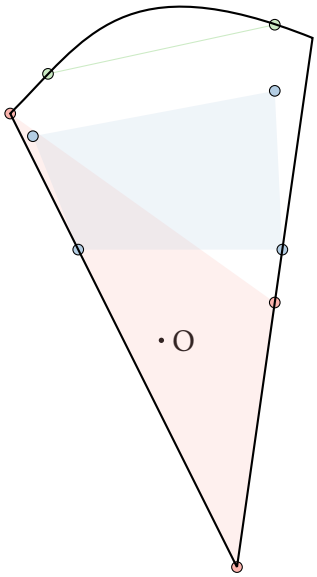


Figure 1.2 A convex set, showing some collections of points and their convex combinations.

A supporting hyperplane to a set $\mathcal{X} \subseteq \mathcal{V}$ is a hyperplane $\mathcal{H} \subseteq \mathcal{V}$ such that:

- $\mathcal{H}$ intersects $\mathcal{X}$ at one or more points, and
- $\mathcal{X}$ is entirely contained in one of the two halves of $\mathcal{V}$ with shared boundary $\mathcal{H}$. This is called a half-space.

See Figure 1.3 for some examples.

Note that half-spaces are convex, and that convex sets are closed under intersection.

Finally, we introduce the concept of a convex hull. This follows naturally from the definition of convexity.

Definition 1.3 (Convex Hull)

The convex hull to a set $\mathcal{X} \subseteq \mathcal{V}$, written conv $\mathcal{X}$, is the set of convex combinations of every point in $\mathcal{X}$. Equivalently, it is the intersection of every half-space in $\mathcal{V}$ which includes all of $\mathcal{X}$.

See Figure 1.4 for an example.

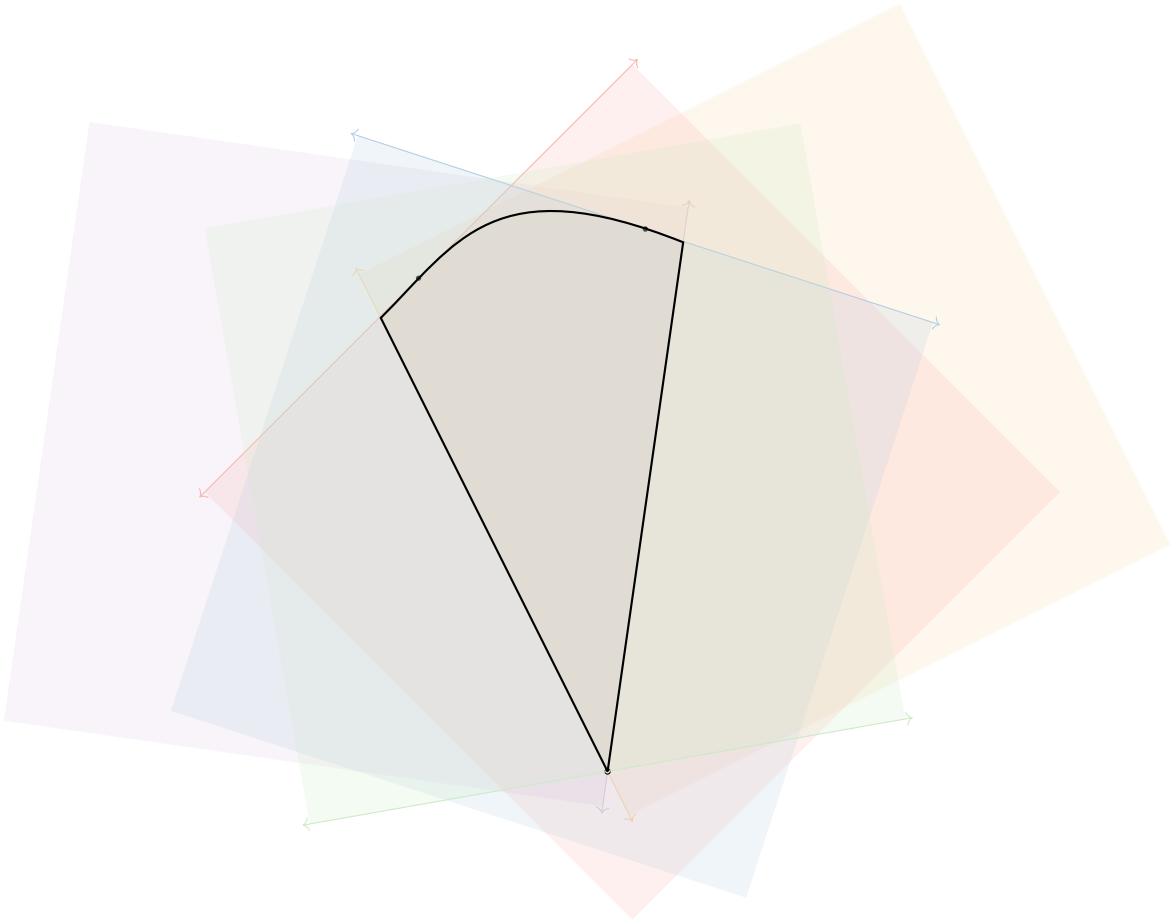


Figure 1.3 A convex set and some of its supporting hyperplanes. The boundary of the half-spaces are depicted as lines, and the space itself as shaded rectangles. Note that these extend infinitely in their respective directions.

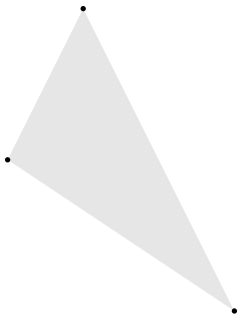


Figure 1.4 A set of three points (black points) and their convex hull (gray shading).

2.
Atomic Sets and Decompositions

In this chapter we discuss in more detail atomic sets and their decompositions, along with the tools which allow us to usefully abstract over these sets. In addition, we cover the rules for the calculus of these tools. This chapter draws from [1, 2, 11, 13] which discuss this subject in more detail.

2.1. Atomic Sets

Let's start by considering sparsity and how one might describe signals which are k-sparse (vectors which have only k nonzero entries) in the language of atomic sets. As we are working with atomic decompositions (Equation 1.1), we want to consider such a vector as a combination of k elementary parts. In this case, we might pick our elementary parts to be the unit basis vectors of our space, scaled appropriately. Since our coefficients are restricted to be nonnegative, we include both e_i and $-e_i$ in our decomposition

$$\begin{aligned} x &= \sum_{a \in \mathcal{A}} c_a a \\ c_a &\geq 0 \\ a &\in \left\{ \pm e_i \right\}, \end{aligned} \tag{2.1}$$

along with the restriction that at most k of c_i can be nonzero.

Example 2.1 (Sparsity)

In our example above, we built our set from positive and negative unit vectors. This gives us the atomic set

$$\mathcal{A} = \mathbb{B}_1 := \left\{ e_i \,\middle|\, i \in [1, n] \right\} \cup \left\{ -e_i \,\middle|\, i \in [1, n] \right\}. \tag{2.2}$$

Example 2.2 (Low-Rankness)

A matrix is low-rank if it has few nonzero singular values relative its size. If an n-by-m matrix is rank-one, for example, it can be written as $u\sigma v^\top$, where σ is its sole nonzero singular value and u, v are unit-norm vectors of dimension n and m, respectively. If it is rank two, then it can be written as the sum of two such outer products. Therefore, if it is low-rank we expect it to be built from a small number of scaled outer products off the form $uv^\top$, and without loss of generality we can have the scalings positive because for every u and v, we also have $-u$ and $-v$.

Therefore, the atoms of interest here are outer products of unit-norm vectors (of appropriate dimension), and

$$\mathcal{A} = \mathbb{B}_* := \left\{ uv^\top \;\middle|\; u \in \mathbb{R}^n, v \in \mathbb{R}^m, \|u\| = \|v\| = 1 \right\}. \tag{2.3}$$

Example 2.3 (Sparsity in Frequency)

Many kinds of data in signal processing are composed of relatively few frequencies, such as a recording of an instrument playing a note where the signal will be primarily composed of the fundamental and harmonics. A single frequency can be represented as the inverse discrete Fourier transform of basis vectors, but for simplicity we consider the discrete cosine transform. As our set of interest is a linear map applied to every element of a set we've already defined ($\mathbb{B}_1$), we can write our set as:

$$\mathcal{A} = \mathtt{DCT}^{-1}\, \mathbb{B}_1, \tag{2.4}$$

where DCT is the discrete cosine transform. Here, the notation $M\mathcal{A}$, where M is a linear map and $\mathcal{A}$ is a set, is the set $\left\{ Ma \;\middle|\; a \in \mathcal{A} \right\}$.

Both [1] and [14] give more examples of atomic sets, along with their convex hulls.

2.2. Tools of the Trade

We wish to have a general framework for dealing with these atomic sets, and so we need a common set of tools which apply to all and have a well-defined meaning. We start by

using the support and expose functions, which allow us to define a notion of alignment with respect to a set.

2.2.1. Support Functions

The first question we might want to ask about an atomic set is about which atoms we'd expect to participate in a decomposition of our data. This naturally leads us to the definition of the support function, which gives us a metric of how aligned our data is with an atom in the set:

Definition 2.1 (Support Function)

The SUPPORT FUNCTION to an atomic set $\mathcal{A}$

$$\sigma_{\mathcal{A}}(z) := \sup\left\{ \langle a, z\rangle \;\middle|\; a \in \mathcal{A} \right\}. \tag{2.5}$$

Intuitively, the support function measures the distance from the origin of the nearest supporting hyperplane of $\mathcal{A}$ in the direction of z.

Example 2.4 (Sparsity, Continued)

Let's define the support function for $\mathbb{B}_1$. We recall from its definition in Equation 2.1, we have only positive and negative coordinate vectors, so the greatest possible inner product with data is always going to be its largest element:

$$\sigma_{\mathbb{B}_1}(z) = \|z\|_\infty. \tag{2.6}$$

Example 2.5 (Low-Rankness, Continued)

For $\mathbb{B}_*$, the supremum will be achieved by the largest singular value of the data. Let the data $Z = U\,\mathrm{Diag}(\mathrm{s})\,\mathrm{V}^\top$, then:

$$\sigma_{\mathbb{B}_*}(Z) := \max_i s_i = \|Z\|_\infty. \tag{2.7}$$

We've intentionally left the support function for $\mathtt{DCT}^{-1}\,\mathbb{B}_1$ until later.

2.2.2. Exposing Faces

The support function gives us a metric of support, but we might also ask which atom(s) achieve this greatest alignment. In other words, which atom in our set is most aligned with our data:

Definition 2.2 (Exposed Face)

The EXPOSED FACE to an atomic set $\mathcal{A}$ in the direction of z is

$$\mathcal{F}_{\mathcal{A}}(z) := \left\{ a \in \mathcal{A} \cup \{0\} \;\middle|\; \langle a, z \rangle = \sigma_{\mathcal{A}}(z) \right\}. \tag{2.8}$$

Note that a face is always convex and always contains at least one point.

Given a method of selecting a single atom from a face consistently, we can also define

$$\begin{aligned} \text{expose}_{\mathcal{A}}(z) = & \arg\sup_{a} \left\{ \langle a, z \rangle \;\middle|\; a \in \mathcal{A} \cup 0 \right\} \\ & \in \mathcal{F}_{\mathcal{A}}(z). \end{aligned} \tag{2.9}$$

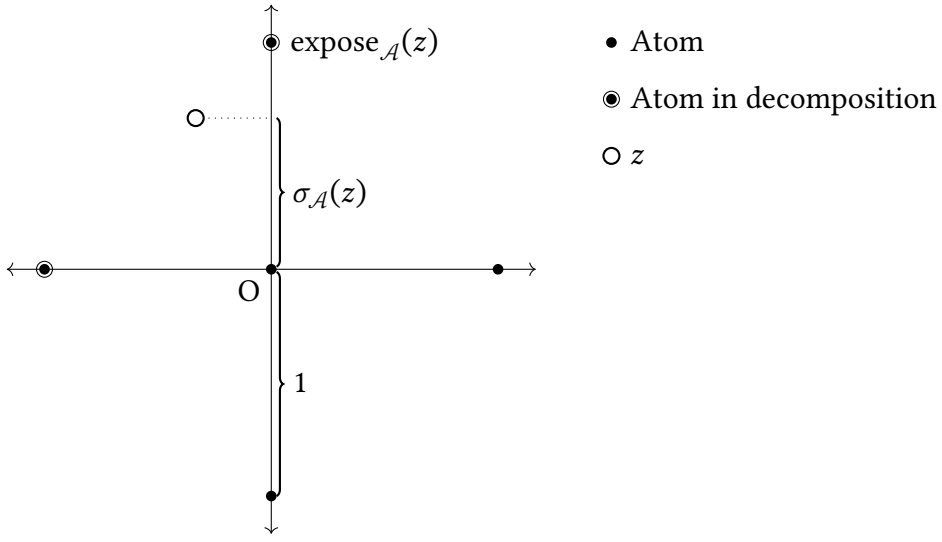


Figure 2.1 An atomic decomposition of some data. Note that we are assuming all the atoms are unit-norm to simplify this diagram, but that is not generally true. In particular, the value of $\sigma_{\mathcal{A}}(z)$ scales with the norm of expose$_{\mathcal{A}}$(z).

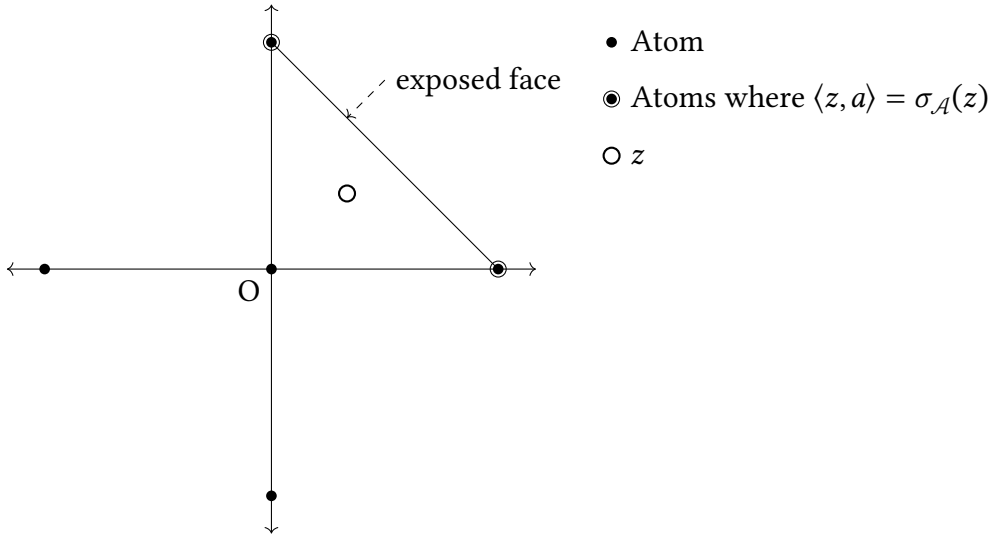


Figure 2.2 An exposed face of the one ball. Note that any point in this face is an acceptable choice for defining the expose operation.

Example 2.6 (Sparsity, Continued)

The definition of the exposed face for $\mathbb{B}_1$ is simplified by noting that this set is the cross-polytope, a regular polytope, in any number of dimensions. When our data is most aligned with the centroid of any k-face, that k-face is the exposed face. Otherwise, only a vertex of the polytope satisfies the supremum for the support function.

We have infinite possible definitions for expose here, but a simple and intuitive one is to take the centroid of $\mathcal{F}(z)$.

Example 2.7 (Low-Rankness, Continued)

If our data Z has multiplicity k for its largest singular value and has a singular value decomposition $\sum_i^k s_i u_i v_i^\top$, then (assuming s is in descending order) the exposed face is

$$\mathcal{F}_{\mathbb{B}_*}(Z) := \sum_i^k s_i u_i v_i^\top. \tag{2.10}$$

The support and expose functions are intuitively revealing the atom closest to our data. This already is enough to define an approximation to data using our set: we can repeatedly build an approximation $\tilde{z}$ to z with respect to an atomic set $\mathcal{A}$ by repeatedly adding to

our approximation $\sigma_{\mathcal{A}}(z-\tilde{z})\cdot \text{expose}_{\mathcal{A}}(z-\tilde{z})$ (with the approximation initialized to zero). The exact meaning of this naturally changes with choice of $\mathcal{A}$; for $\mathbb{B}_1$ we get a low-rank approximation to our data (assuming the number of times we updated the approximation is fewer than the number of nonzero elements in z) and for $\mathbb{B}_*$ we'd get a low rank approximation (assuming, again, the number of atoms we exposed is fewer than the rank of our data).

2.2.3. Gauge Functions

We can see how atomic decompositions might be useful, but they are frequently not unique. How do we learn about how good our decomposition is?

Definition 2.3 (Gauge)

The gauge function to an atomic set

$$\gamma_{\mathcal{A}}(z) := \inf_{c}\left\{\sum_{a\in\mathcal{A}} c_a \;\middle|\; z=\sum_{a\in\mathcal{A}} c_a a, c_a \geq 0\right\}. \tag{2.11}$$

Or, equivalently[2],

$$\gamma_{\mathcal{A}}(z) := \inf_{\lambda}\left\{\lambda \;\middle|\; z\in\lambda\ \text{conv}\left(\mathcal{A}\cup\{0\}\right)\right\}. \tag{2.12}$$

Note that the gauge function's codomain is $\mathbb{R}\cup\{\infty\}$, as there may be no scaling for which the point of interest is contained within the set.

Example 2.8 (Sparsity, Continued)

For $\mathbb{B}_1$, it's easier to consider the definition in Equation 2.12 when we note that the convex hull of our set is the unit level set for the l_1 norm. As such, the gauge to this set is just that norm.

Example 2.9 (Low-Rankness, Continued)

We note for $\mathbb{B}_*$ that the coefficients in a decomposition with respect to this set are the singular values of the input data. The sum of the singular values is the Schatten 1-norm, so the gauge to this set is again that norm.

For both of these examples, the gauge function w.r.t. our sets was just a norm. This isn't always true, but is when the set is bounded, symmetric, and contains the origin. Gauge functions generalize norms.

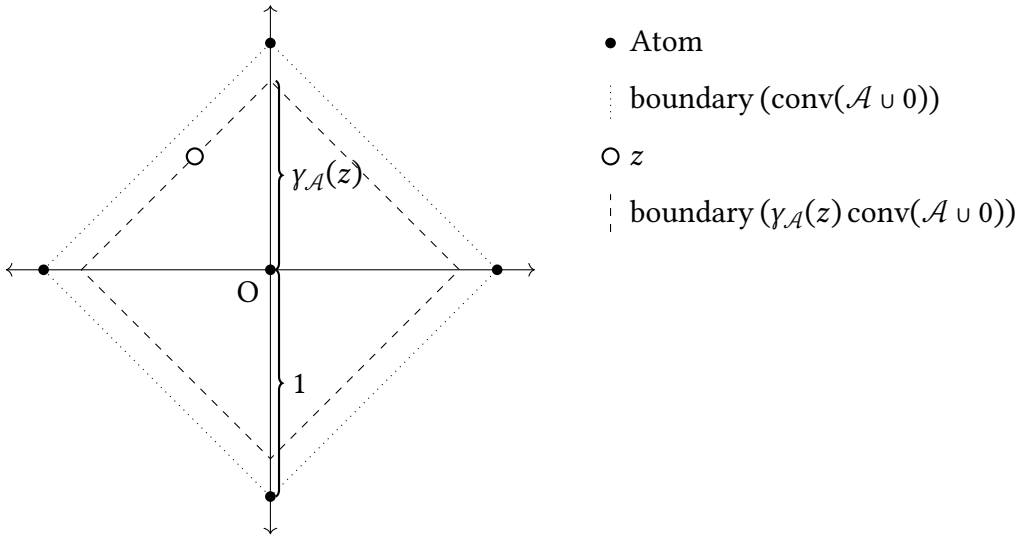


Figure 2.3 $\mathbb{B}_1$ and its gauge visualized. Note that as in Figure 2.1, all atoms are unit-norm to simplify the diagram. The gauge function also scales with the norm of the atoms in the set.

2.2.4. Alignment

We've made the claim that gauge functions generalize norms, but in what specific sense is that true? We've seen that gauges with respect to some sets are norms, but what about when that isn't true?

We start by examining the case where both $\sigma_{\mathcal{A}}$ and $\gamma_{\mathcal{A}}$ are norms.

Notice that, for example, $\sigma_{\mathbb{B}_1}(z) = \|z\|_\infty$ and $\gamma_{\mathbb{B}_1}(x) = \|x\|_1$. If we build the set $\mathbb{B}_2 := \left\{ a \;\middle|\; \|a\|_2 = 1 \right\}$, we find its gauge and support to be the same; $\sigma_{\mathbb{B}_2}(z) = \gamma_{\mathbb{B}_2}(z) = \|z\|_2$.

Definition 2.4 (Dual Norm)

Given a norm $\|\cdot\|$, its dual is

$$\|\cdot\|^* = \sup\left\{ \langle z, x\rangle \;\middle|\; \|x\| \leq 1 \right\}. \tag{2.13}$$

Duals always obey

$$\langle z, x \rangle \leq \|x\| \cdot \|z\|^*. \tag{2.14}$$

When $\gamma_{\mathcal{A}}$ is a norm, $\sigma_{\mathcal{A}}$ is its dual.

However, as previously noted, $\gamma_{\mathcal{A}}$ is not necessarily a norm, and may be one of a large family of convex functions. Still, as shown in [12] and [2], these functions share a similar relationship

$$\langle x, z \rangle \leq \gamma_{\mathcal{A}}(x)\, \sigma_{\mathcal{A}}(z), \tag{2.15}$$

which is called the polar inequality. We say that vectors z and x are ALIGNED with respect to $\mathcal{A}$ if Equation 2.15 is an equality.

2.3. The Calculus of Support Functions

So far we've considered sets on their own, and we've defined a common set of operations we wish to have on any atomic set (support functions and exposed faces), and a quantity of interest (the gauge).

We've defined one set another way, though: when discussing signals that are sparse in frequency, we were working with the set $\mathrm{DCT}^{-1}\mathbb{B}_1$. While we could define this set explicitly, it's much more convenient to work with a set already defined.

Assuming we have our common tools defined for some atomic set $\mathcal{A}$, we can work them out for the set $M\mathcal{A}$, where M is a linear operator:

$$\begin{aligned}
\sigma_{M\mathcal{A}}(z) &= \sup_a \left\{ \langle a, z \rangle \;\middle|\; a \in M\mathcal{A} \cup \{0\} \right\} \\
&= \sup_a \left\{ \langle Ma, z \rangle \;\middle|\; a \in \mathcal{A} \cup \{0\} \right\} \\
&= \sup_a \left\{ \langle a, M^\top z \rangle \;\middle|\; a \in \mathcal{A} \cup \{0\} \right\} \\
&= \sigma_{\mathcal{A}}(M^\top z).
\end{aligned} \tag{2.16}$$

Similarly,

$$
\begin{aligned}
\gamma_{M\mathcal{A}}(z) &= \inf_c \left\{ \sum_a c_a \;\middle|\; z = \sum_a c_a\, a,\ c_a \geq 0,\ a \in M\mathcal{A} \right\} \\
&= \inf_c \left\{ \sum_a c_a \;\middle|\; z = \sum_a c_a M a,\ c_a \geq 0,\ a \in \mathcal{A} \right\} \\
&= \inf_c \left\{ \sum_a c_a \;\middle|\; M\backslash z = \sum_a c_a a,\ c_a \geq 0,\ a \in \mathcal{A} \right\} \\
&= \gamma_{\mathcal{A}}(M\backslash z).
\end{aligned}
\tag{2.17}
$$

Finally,

$$
\begin{aligned}
\mathcal{F}_{M\mathcal{A}}(z) &= \left\{ a \in M\mathcal{A} \cup \{0\} \;\middle|\; \langle a, z\rangle = \sigma_{M\mathcal{A}}(z) \right\} \\
&= \left\{ a \in M\mathcal{A} \cup \{0\} \;\middle|\; \langle a, z\rangle = \sigma_{\mathcal{A}}(M^\top z) \right\} \\
&= \left\{ Ma \in \mathcal{A} \cup \{0\} \;\middle|\; \langle a, M^\top z\rangle = \sigma_{\mathcal{A}}(M^\top z) \right\} \\
&= M\mathcal{F}_{\mathcal{A}}(M^\top z).
\end{aligned}
\tag{2.18}
$$

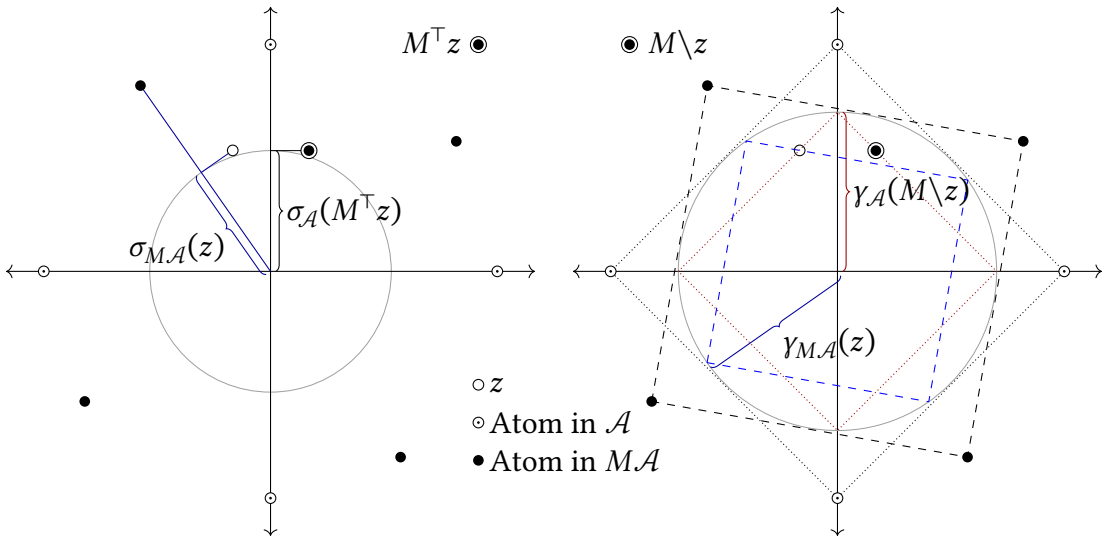


Figure 2.4 $\mathbb{B}_1$ transformed under a linear map, with support and gauge labelled. For visual purposes, the transformation is unitary so that all atoms remain unit-norm and the support and gauge are trivially drawable.

Additionally, we can get some results for the Minkowski sum of sets. Given $\mathcal{A}_1$ and $\mathcal{A}_2$ and $\mathcal{A} = \mathcal{A}_1 + \mathcal{A}_2 = \left\{ a_1 + a_2 \;\middle|\; a_1 \in \mathcal{A}_1, a_2 \in \mathcal{A}_2 \right\}$, and noting that given an $a \in \mathcal{A}$ there must exist $a_1 \in \mathcal{A}_1, a_2 \in \mathcal{A}_2$ such that $a = a_1 + a_2$:

$$
\begin{aligned}
\sigma_{\mathcal{A}_1+\mathcal{A}_2}(z) &= \sup_{a} \left\{ \langle z, a \rangle \;\middle|\; a \in (\mathcal{A}_1 + \mathcal{A}_2) \right\} \\
&= \sup_{a_1+a_2} \left\{ \langle z, a_1 + a_2 \rangle \;\middle|\; a_1 \in \mathcal{A}_1,\ a_2 \in \mathcal{A}_2 \right\} \\
&= \sup_{a_1+a_2} \left\{ \langle z, a_1 \rangle + \langle z, a_2 \rangle \;\middle|\; a_1 \in \mathcal{A}_1,\ a_2 \in \mathcal{A}_2 \right\} \\
&= \sup_{a_1} \left\{ \langle z, a_1 \rangle \;\middle|\; a_1 \in \mathcal{A}_1 \right\} + \sup_{a_2} \left\{ \langle z, a_2 \rangle \;\middle|\; a_2 \in \mathcal{A}_2 \right\} \\
&= \sigma_{\mathcal{A}_1}(z) + \sigma_{\mathcal{A}_2}(z),
\end{aligned}
\tag{2.19}
$$

and

$$
\begin{aligned}
\mathcal{F}_{\mathcal{A}_1+\mathcal{A}_2}(z) &= \left\{ a \;\middle|\; a \in (\mathcal{A}_1 + \mathcal{A}_2),\ \langle a, z \rangle = \sigma_{\mathcal{A}_1+\mathcal{A}_2}(z) \right\} \\
&= \left\{ a_1 + a_2 \;\middle|\; a_1 \in \mathcal{A}_1,\ a_2 \in \mathcal{A}_2,\ \langle a_1 + a_2, z \rangle = \left(\sigma_{\mathcal{A}_1}(z) + \sigma_{\mathcal{A}_2}(z) \right) \right\} \\
&= \left\{ a_1 \;\middle|\; a_1 \in \mathcal{A}_1,\ \langle a_1, z \rangle = \sigma_{\mathcal{A}_1}(z) \right\} + \left\{ a_2 \;\middle|\; a_2 \in \mathcal{A}_2,\ \langle a_2, z \rangle = \sigma_{\mathcal{A}_2}(z) \right\} \\
&= \mathcal{F}_{\mathcal{A}_1}(z) + \mathcal{F}_{\mathcal{A}_2}(z).
\end{aligned}
\tag{2.20}
$$

Note that we cannot get an analytic form for $\gamma_{\mathcal{A}_1+\mathcal{A}_2}$. See Chapter 3 for a discussion.

3.
The Level-Set Method and Deconvolutions

We now have a tool, atomic decompositions, for dealing with data which is sparse or simple relative to some abstract set which defines the notion of sparsity. We are here interested in solving inverse problems of the form

$$b = Mx_s^\natural + \eta \quad \text{where} \quad x_s^\natural := \sum_{i=1}^{k} x_i^\natural, \tag{3.1}$$

where b is our measurement, $x^\natural$ are true signals, M is a linear map, and η is some noise. The goal here is to find some $x_i \approx x_i^\natural$ under the assumption that the true signals have structure; or, more specifically, that they are sparse relative to a known atomic set.

Chandrasekaran et. al.[1] examine this in the case of all signals being drawn from a single set, while Fan et. al.[11] consider this in the case where signals may be drawn from multiple atomic sets $\mathcal{A}_i$. This chapter follows [11], which makes the assumption that each of the k signals $x_i^\natural$ is sparse relative to the set $\mathcal{A}_i$.

First, we note that each $x_i^\natural$ being $\mathcal{A}_i$-sparse implies that $x_s^\natural = \sum_i x_i$ is sparse relative to $\mathcal{A}_s = \sum_i \lambda_i \mathcal{A}_i$, where λ_i are real coefficients which define the relative contribution of each set to the whole signal, defined so that $\gamma_{\lambda_i \mathcal{A}_i}(x_i^\natural) = \gamma_{\mathcal{A}_1}(x_1^\natural)$. See Chapter 6 for examples of computing these coefficients.

3.1. Outline of the Algorithm

Now, we'll outline the algorithm. We introduce an accuracy parameter α, which defines the maximum level of misfit between the mapped solution Mx and the data b. Intuitively, this is the expected size of the error. There are two stages to the algorithm: stage one solves the problem

$$\min_y \gamma_{M\mathcal{A}_s}(x) \quad \text{s.t.} \quad \|y - b\|_2 \leq \alpha. \tag{3.2}$$

This is the *decompression* stage, which recovers the superposition of the signals approximating $x_s^\natural$ in Problem 3.1. Stage two solves

$$\min_{x_1,\ldots,x_k} \max_{i\in[1,k]} \gamma_{\lambda_i \mathcal{A}_i}(x_i) \quad \text{s.t.} \quad \sum_i^k x_i = x_s^*, \tag{3.3}$$

where x_s^* is the solution to Problem 3.2. This is the *deconvolution* problem, where given a solution to the decompression problem we recover the individual signals $x_i^\natural$.

3.2. The Level Set Method

In this section, we describe stage I of the algorithm (solving Problem 3.2).

As shown in [1], finding the gauge gives us information about the ideal atomic decomposition. This is partially apparent from its definition (Equation 2.11). However, it is not immediately obvious how to compute the gauge, as the naive method would require enumerating every possible decomposition and taking the minimum. However, the level set method described in [10, 15, 16] provides a root-finding algorithm: we transform the problem

$$\inf_\lambda \lambda \quad \text{s.t.} \quad b \in \lambda \mathcal{A}, \tag{3.4}$$

which uses the alternate definition for the gauge function from Equation 2.12, into

$$\begin{aligned} &\inf_\tau \left\{ \tau \;\middle|\; v(\tau) = 0 \right\} \quad \text{s.t.} \quad \gamma_{\mathcal{A}}(x) \le \tau, \\ &\text{where} \quad v(\tau) = \min_x \frac{1}{2}\|x - b\|_2^2, \end{aligned} \tag{3.5}$$

which can be solved with a root-finding method.

To see this, we can get an upper bound to the problem using the norm squared of the primal residual $r = x - b$. The dual problem (solved approximately in our specific case in Subsection 3.2.1) gives us a duality gap, which allows also an underestimate of the true iterate. Using this true iterate, we can project onto the ideal solution $\frac{1}{2}\alpha^2$ (using our error parameter α). For an illustration, see Figure 3.1.

The specific application of the general level-set method to solving the decompression problem follows. Note here that this method is generic over atomic sets $\mathcal{A}$, which in our case will be the set $M\mathcal{A}_s$. However, as we'll discuss in Section 5.4, it is useful to consider a general set $\mathcal{A}$ which is notated here.

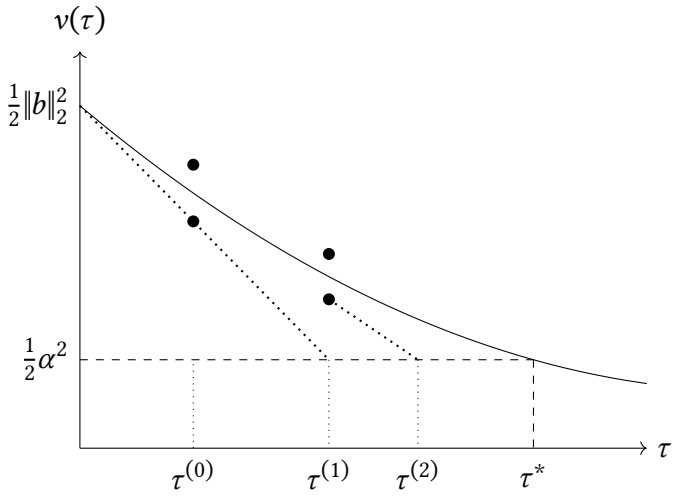


Figure 3.1 Illustration of the Level-Set Method. Dots above the $v(\tau)$ line are overestimates and those under are underestimates. The distance between these points is the duality gap g. Illustrated are two iterations. Note the first iteration is free and exact, which is why there are no over- or under-estimates at $\tau = 0$. Also note the scale here is not exact, and the free first iterate would have made more progress than illustrated.

Algorithm 3.1 (Level-Set Algorithm)

(1) **Input:** noise level α, accuracy ϵ, projection tolerance ϵ_p
(2) $\tau_0 \leftarrow 0$
(3) **For** $t \leftarrow 0, 1, \ldots,$ **do:**
(4) $\quad \left(y_\tau^t, g^t\right) \leftarrow \text{project}_{\tau^{(t)}\mathcal{A}}\left(b, y^{(t-1)}, \epsilon_p\right)$
(5) $\quad r^{(t)} \leftarrow b - y_\tau^{(t)}$
(6) $\quad l^{(t)} \leftarrow \max\left(\alpha, \|r^{(t)}\|_2^2 - g^{(t)}\right)$
(7) $\quad$ **If** $\|r^{(t)}\| < \sqrt{\alpha^2 + \epsilon}$, **then break**
(8) $\quad \tau^{(t+1)} \leftarrow \tau^{(t)} + \left(l^{(t)} - \alpha^2/2\right)/\sigma_{\mathcal{A}}(r^{(t)})$
(9) $\quad y^{(t)} \leftarrow y_\tau^{(t)} \cdot \tau^{(t+1)}/\tau^{(t)}$
(10) $(x_1, \ldots, x_k) \leftarrow \text{deconvolve}(r^{(t)})$
(11) **Return** $(x_1, \ldots, x_k)$

At each iteration, we project the measurement onto the atomic set $\tau^{(t)}M\mathcal{A}_s$, starting at the previous estimate for $Mx_s^\natural$ using the dual conditional gradient method described in Subsection 3.2.1, which returns an underestimate for $\tau^{(t)}Mx_s^\natural$ and the duality gap g (line

4). We then update the residual (line 5) and the lower minorant $l^{(t)}$ for the $v(\tau)$ update (line 6). If the problem is (super-)optimal, we stop. Otherwise, we update the iterate τ (line 8) and the estimate for $Mx_s^\natural$ (line 9). Finally, we solve the deconvolution problem as described in Section 3.3 (line 10).

3.2.1. Dual Conditional Gradient Method for Projection

Note here we are again writing our algorithm generically over a set $\mathcal{A}$. In the case of solving the decompression problem we are projecting onto the set $\tau^{(t)} M\mathcal{A}_s$.

The linesearch step on line 8 above depends on the input set $\mathcal{A}$. When $\mathcal{A}$ is of the form $\sum_i M\mathcal{A}_i$ (or equivalently $M\sum_i \mathcal{A}_i$, which is first transformed into the prior form), we keep separate iterates for y and r, and perform separate search directions for each of them (Equation 3.6). In this case, $A = \sum_i^k \mathcal{A}_i$. Otherwise, there is only a single iterate and we perform a simple line search (Equation 3.7).

Algorithm 3.2 (Dual Conditional Gradient Method for Projection)

For a single set $\mathcal{A}$, the algorithm is:

(1) **Input:** b, estimate y, accuracy ϵ_p
(2) $r^{(0)} \leftarrow b - y$
(3) **For** $t \leftarrow 0, 1, \ldots,$ **do:**
(4) $\quad p^{(t)} \leftarrow \text{expose}_{\mathcal{A}}(r^{(t)})$
(5) $\quad \Delta r^{(t)} \leftarrow p^{(t)} - y^{(t)}$
(6) $\quad g^{(t)} \leftarrow \left\langle r^{(t)}, \Delta r^{(t)} \right\rangle$
(7) $\quad$ **If** $g^{(t)} < \epsilon_p$, **then break**
(8) $\quad \theta^{(t)} \leftarrow \text{linesearch}\left(r^t, \Delta r^{(t)}, g^{(t)}\right)$
(9) $\quad r^{(t+1)} \leftarrow r^{(t)} - \theta^{(t)} \Delta r^{(t)}$
(10) $\quad y^{(t+1)} \leftarrow y^{(t)} + \theta^{(t)} \Delta r^{(t)}$
(11) **Return** $\left(y^{(t)}, g^{(t)}\right)$

When we know that $A = \sum_i^k \mathcal{A}_i$, we can accelerate by tracking multiple iterates. The input changes to a set of estimates y_i, one for every $\mathcal{A}_i$. Similarly, we have one exposed atom p_i and search direction Δr_i for each set. We take the sum on line 6 and the products on lines 9 and 10 become elementwise with the set of coefficients θ

returned from Equation 3.6. This allows a separate search direction for each of the sets, while still remaining within the bounds of $\mathcal{A}$ for the next iterate.

In the case of multiple iterates, we solve the problem:

$$\theta = \underset{\theta_1,\ldots,\theta_k \in [0,1]}{\arg\min} \left\| r^{(t)} - \sum_{i=1}^{k} \theta_i \Delta r_i^{(t)} \right\|. \tag{3.6}$$

Otherwise, we use the simpler linesearch

$$\theta^{(t)} = \min\left(1, g^{(t)} / \|\Delta r^{(t)}\|_2^2\right). \tag{3.7}$$

3.3. Polar Deconvolutions

In this section, we describe stage II of the algorithm (solving Problem 3.3). At line 10 of Algorithm 3.1, the residual r contains information about which atoms are in the support for each of the approximations x_i to $x_i^\natural$. From [2], we know that $x_i^* \in \operatorname{cone} \mathcal{F}_{M,\mathcal{A}_i}(r^*)$, where $\operatorname{cone} \mathcal{X}$ is the set $\left\{ x \;\middle|\; \exists \lambda \in \mathbb{R}, \lambda \geq 0, x \in \lambda \mathcal{X} \right\}$. Therefore, we can solve the deconvolution problem by solving

$$\min_{x_1,\ldots,x_k} \frac{1}{2} \left\| M \sum_{i}^{k} x_i - (b - \bar{r}) \right\|_2^2 \quad \text{s.t.} \quad x_i \in \operatorname{cone} \mathcal{F}_{MF_i}(\bar{r}), \tag{3.8}$$

where $\bar{r}$ is the approximation to r^* from line 10 of Algorithm 3.1. This problem can be solved as a linear least-squares problem with box constraints.

In [2], analysis of recovery conditions for this step is done under the assumption that each signal is drawn from a set which is sufficiently incoherent; that is, the descent cones are sufficiently distant in angle and have only a trivial intersection. Intuitively, if two sets are too similar (for example, the one-ball and the simplex in the same direction), the signals cannot be deconvolved.

4.
The Julia Programming Language

Julia[9] is a high-level programming language designed for mathematical and scientific computing. It has a unique evaluation system that allows for high-level, dynamic code with high performance. Its core principle is that when TYPE INFERENCE is able to determine exact types for all expressions, we can compile a specialized version of the code using the optimizations that LLVM offers.

Julia's main programming paradigm, MULTIPLE DISPATCH, lends itself to intuitive representations of mathematics. That, combined with its evaluation system, allows for representing mathematics in a way which is both performant and expressive. We start by describing Julia's operation at a high level, then go over its type system and evaluation model in more detail. Then, we discuss techniques for representing mathematics in Julia, and how they can be used in combination with static analysers to detect some kinds of errors ahead-of-time.

4.1. Julia

Briefly, the core programming paradigm Julia uses is multiple dispatch. Functions are generic objects which have zero or more callable METHODS. When a function is called with some arguments, the most specific method is used. For example,

Julia
```
foo(x::Number)  = 2x
foo(x::Integer) = 3x
foo(x::Int8)    = 4x
```

where `x::T` indicates that `x` is a value of type `T`, and `T <: U` indicates `T` is a subtype of `U`. Here,

- `foo(1.5)` uses a version of the topmost method compiled for `x::Float64`, as `Float64` is a subtype of `Number` (`Float64 <: Number`) but not of `Integer` or `Int8`.
- `foo(25)` uses a version of the middle method compiled for `x::Int64`, as `Int64 <: Number` and `Int64 <: Integer`, but `Integer <: Number`, making the second method more specific.
- `foo(Int8(42))` uses the last method.

This system allows for a large amount of expressiveness with relatively few lines of code while still getting high performance. To define addition for a custom type, for example, we need only add a method to the built-in + function.

Another example is adding custom array types. Here is an example of a fill-matrix which stores only a single value:

Julia
```julia
# declaring a parametric type, which is a subtype of another
# also, restrict that parameter to be a subtype of `Number`
struct FillMatrix{T <: Number} <: AbstractMatrix{T}
    fill::T
    size::Tuple{Int, Int}
end

# adding a method to the generic `size` function
size(a::FillMatrix) = a.size

# adding a method to the generic `getindex` function
function getindex(a::FillMatrix, i::Integer, j::Integer)
    # check bounds
    if i ≤ 0 || j ≤ 0 || i > a.size[1] || j > a.size[2]
        throw(BoundsError(a, (i, j)))
    else
        a.fill
    end
end
```

With only those two methods defined, we get matrix-vector and matrix-matrix products. In fact, `FillMatrix` now works for much of the standard library linear algebra functions, all of which will generate code specialized on `T`.

We now discuss Julia in more detail.

4.1.1. Julia's Type System & Multiple Dispatch

There are four kinds of types in Julia: `DataTypes`, `Unions`, `UnionAlls` and `Bottom`. We'll discuss them in order.

`DataTypes` form a tree. They are either abstract or concrete. Only concrete types may have values, and only abstract types may have subtypes. They are nominal and parametric, and type parameters may be values of types which are "bitstypes"—types which are immutable and contain no references. In Julia parlance, types which have values as parameters are

referred to as VALUE TYPES[1]. `DataTypes` may be mutable or immutable, which changes the mutability of any value of the type. Mutability is not hereditary; an immutable type may contain a mutable type which retains its mutability.

`Unions` express "one-of" relationships: if a variable can be either an `Int64` or a `Float64`, it is of type `Union{Float64, Int64}`. We also have that `Int64 <: Union{Float64, Int64}` and that `Float64 <: Union{Float64, Int64}`, making `Unions` useful for controlling method dispatch in different manners than through the `DataType` hierarchy. Finally, `Union{}` is the bottom type and it is uninhabitable.

Last of all, we have `UnionAll` types. For a parametric type, these represent a union over every possible parameter in a parametric type. For example, the Julia standard library has the `Vector{T}` type, which is used both as a growable list of `T`s and as a vector of `T` in the linear algebra sense. `Vector` is a `UnionAll`, giving us e.g. `Vector{Int} <: Vector`.

As previously stated, functions are generic objects which have zero or more callable methods. These are declared similarly to function overloading from languages like C++. When a function is called, the types of the arguments are used to look up the appropriate method, which is always the most specific[2].

Types themselves are first-class, which is a critical component of Julia's PROMOTION system, where values of different types are "widened" to an appropriate common type before a binary operation. For example, when calling `+` on a `Float32` and a `Float64`, no specific method for those two argument types exists. However, there is a method which takes two `Numbers` and promotes them to a common type, in this case `Float64`. This is chosen because it can represent, with a possible loss of accuracy, all values of both input types. These rules are themselves simply written in Julia using multiple dispatch on the `promote_rule` function. As types are first class values, this can be expressed and extended naturally.

All of Julia's typing is semantically dynamic, and values are statically unityped as the top type `Any`. Variables do not need type annotations, and methods can be looked up at runtime. However, the Julia compiler runs type inference on every function when given input types. If it is able to determine concrete types for expressions, then it can take advantage of the LLVM compilation pipeline. Otherwise, for values that have not been assigned concrete types boxing and dynamic dispatch is used. This process is called SPECIALIZATION. For example, let's declare a function:

[1] They are closely related to dependent types in other languages, non-type template parameters in C++, and const generics in Rust.

[2] This may be ambiguous, in which case an error is thrown.

Julia
```
bar(x, y, z) = (x + y) / z
```

Julia gives us many tools for examining intermediate results. Let's use the `@code_warntype` macro to examine a typed intermediate representation when called with all integers:

Julia
```
julia> @code_warntype bar(1, 2, 3)
MethodInstance for bar(::Int64, ::Int64, ::Int64)
  from bar(x, y, z) in Main at REPL[1]:1
Arguments
  #self#::Core.Const(bar)
  x::Int64
  y::Int64
  z::Int64
Body::Float64
1 — %1 = (x + y)::Int64
│    %2 = (%1 / z)::Float64
└──      return %2
```

This internal representation of code is in static single assignment (SSA) form[17], where intermediate steps are assigned to numbered variables which are assigned a value once. Type inference is applied to it, and we see the results annotated using the `x::T` syntax. In the body, we can see that intermediate values have all been inferred, and `x + y` has been inferred as another 64-bit integer and the result of that divided by `z` as a 64-bit float. This intermediate representation can now be compiled to LLVM IR, and we can use its full optimization pipeline to get efficient machine code for this particular combination of input types.

This is the fundamental building block of how Julia achieves good execution speed. Whenever a method is called with a new set of input types, a specialized version is compiled after type inference is run. When type inference fails to assign a concrete type to an expression (perhaps a variable is assigned values of multiple types or perhaps there is not enough information), Julia falls back to dynamic dispatch. This is referred to in the Julia community as "Just Ahead-Of-Time compilation" as it technically falls under the definition of Just-In-Time compilation but makes full advantage of the tools available for ahead of time compilation (and is not a tracing JIT). Since functions can also be inlined during this process, very high performance code can be achieved using high-level constructs such as higher-order functions.

4.1.2. Other Performance Considerations

There are performance considerations to mutability of user-defined types. In particular, mutable types (types declared with `multable struct`) typically require boxing. Let's examine the following two type definitions:

Julia
```
mutable struct MutableWrapped
    x::Float64
end

struct ImmutableWrapped
    x::Float64
end
```

Due to Julia's semantics, we require the following:

Julia
```
x = MutableWrapped(1.0)
a = [x, x] # create two-element vector of MutableWrapped
a[1].x = 0. # set the wrapped value of the first item to second
a[2].x == 0. # will be true
```

The only practical way to implement this is to have any `MutableWrapped` be heap-allocated and only store references. However, for `ImmutableWrapped`, we have:

Julia
```
x = ImmutableWrapped(1.0)
a = [x, x]
# we can't overwrite a[1].x since it is immutable, so we replace
a[1] = ImmutableWrapped(0.)
a[2].x  # will be 1.
```

and so the compiler is free to store `ImmutableWrapped` on the stack (or directly in a vector in the above case).

The other performance consideration we want to examine here is surrounding `Unions`. As they show up frequently in inferred code, including every time there is iteration, their performance is critical. If we have a `Union` of only bitstypes (for example `Union{Int32, Float32}`), it is stored as a tagged union (and therefore can be stack-allocated). Dispatch on it can be reduced from full dynamic dispatch to branching or a lookup table.

4.2. Representing Mathematics in Julia

We've already touched on how Julia's dispatch allows easy extension of concepts such as addition to new types, and even in such a way that can be combined easily with existing types.

Typical goals when writing a software library for representing mathematics would be:

- Ergonomics of use—how intuitive the behaviour of your representations are and how close your code is to a reasonable notation for the math.
- Performance—when executing high level representations, how close to a hand-tuned implementation one can get both in time and space utilization.
- Correctness—beyond the obvious goal of having results of operations be correct, errors should ideally be statically identifiable and invalid states should be unrepresentable.

In terms of ergonomics, Julia allows familiar syntax for declaring and applying functions. Most users are uncomfortable with prefix or s-expression syntaxes, and so using the familiar `verb(objects)` notation is a reasonable choice for notation. In addition, Julia eschews the object-oriented `subject.verb(objects)` format of single dispatch. While many programmers are familiar with it, it has a significant disadvantage in ergonomics. Consider the following example in Python of doing geometric algebra computations, which computes the wedge product on either a full multivector or k-vector of only a single grade.

Python

```
class KVector(GeometricAlgebraElement):
    ...

class Multivector(GeometricAlgebraElement):
    ...

a, b, c = ...

outer = a.wedge(b).wedge(c)
```

The wedge product is a binary operation, and it is not at all clear why one of the operands should be the "subject" and the other the "object". Julia also allows declaring infix operators (from a preselected list), so we get the much more natural result of:

Julia

```julia
struct KVector <: GeometricAlgebraElement
    ...
end

struct Multivector <: GeometricAlgebraElement
    ...
end

wedge(x::GeometricAlgebraElement, y::GeometricAlgebraElement) = ...
wedge(x::KVector, y::KVector) = ...
const ∧ = wedge

outer = a ∧ b ∧ c
```

Prefix notation, such as most Lisps' `(verb objects)` also solves this problem, but is unfamiliar to most programmers and does not resemble traditional math notation.

In addition, in Python (and many other languages) it is difficult to add type-based optimizations. For example, suppose we have either full multivectors or single-grade k-vectors, both of which implement a generic interface. In the case of computing the wedge product for two k-vectors, the result will always be another k-vector and we can skip a large amount of computation. In Julia adding this special case is easy using multiple dispatch, but in Python it requires a special check in the `wedge` method of `KVector`. If we had an optimization involving two different types, Python would require adding a check to both methods while in Julia we need only add two methods (one of which is trivial if the operation is commutative).

Correctness is an ongoing area of experimentation in Julia. Its system of dispatch is flexible enough that totally unrelated packages can be freely combined, revealing sometimes mismatched assumptions about core types, which are therefore underspecified. However, it shows promise in terms of static analysis compared to other dynamically typed languages since after successful type inference it has a statically typed IR. As this is also desirable for performance reasons, Julia already has built-in tools for checking and improving the results of type inference. JET.jl[18] builds on top of this system to perform static analysis (even interactively in the REPL).

Julia also has value types which could be used to enforce sizing through definitions—see Section 7.1.

5.
AtomicSets.jl: The Calculus of Atomic Sets in Julia

Here we describe a representation of the calculus of atomic sets and the level set method in Julia. An emphasis has been placed on efficient execution within the limits of good ergonomics. We have made certain assumptions in this model; in particular that in general the use case is that sets will be constructed infrequently compared to the number of times they are used. In particular, sets should correspond one-to-one to types as much as possible so that dispatch on generic functions like `support` or `expose` can be compiled independently for every set. This provides highly specialized code, however in the use case where sets are frequently dynamically created (perhaps from data), this creates significant overhead in compilation and dispatch.

5.1. Basic Set Representations

5.1.1. Sets

Every set is a subtype of

Julia
```
abstract type AtomicSet
end
```

As previously stated, every set would ideally have its own type. For basic sets (not sums or maps), this is relatively easy to accomplish. For the l_1 norm ball for example (see Equation 2.2), the only information that we need about the set is that it

- is the l_1-norm-ball,
- is in dimension N.

As such, we've chosen to represent it like so:

Julia
```
struct OneBall{N} <: AtomicSet
end
```

However, as Julia currently supports no option for predicates or restrictions on value-types (`N` in this case), we have a constructor and check that it is positive and integral:

Julia
```
struct OneBall{N} <: AtomicSet
    function OneBall{N}() where {N}
        N > 0 || throw(DomainError("Dimension $N not positive"))
        N isa Integer || throw(TypeError(:Set,"", Integer, typeof(N)))
        new{N}()
    end
end
```

While it would appear from this that constructing a `OneBall{N}` requires run-time checks, this is not the case. Since every function in Julia is compiled for every set of input types, the constructor is compiled for every value of `N`. Constant folding is able to elide the checks, which we can verify by examining lowered code:

Julia
```
julia> @code_typed OneBall{5}()
CodeInfo(
1 —      nothing::Nothing
│        nothing::Nothing
│   %3 = %new(OneBall{5})::OneBall{5}
└──      return %3
) => OneBall{5}

julia> @code_typed OneBall{-1}()
CodeInfo(
1 —      goto #3 if not false
2 —      nothing::Nothing
3 ┄ %3 = Base.string("Dimension ", $(Expr(:static_parameter, 1)), " not positive")::Any
│   %4 = AtomicSets.DomainError(%3)::Any
│        AtomicSets.throw(%4)::Union{}
└──      unreachable
) => Union{}
```

This is in the same SSA form as described in Subsection 4.1.1.

In the first case, the compiler has correctly identified that the exceptions can never be thrown and that the function is infallible for `N = 5::Int`. In the second, it's correctly identified that the function will always throw (since `N < 1`), and the return type has been identified as `Union{}` (the bottom type).

The expected performance benefit of eilding the `N isa Integer` and `N > 0` checks here is almost nothing, especially since we've stated that the assumption under which we're working is that sets are constructed infrequently relative to their use. However, this is a

good demonstration of the advantages of using value types: eliding checks like this may have greater impact on hot loops, and they make static analysers more useful.

Other sets are represented similarly, for example the `NucBall` ($\mathbb{B}_*$):

Julia
```
struct NucBall{M, N} <: AtomicSet
    function NucBall{M, N}() where {M, N}
        N isa Integer || throw(TypeError(:NucBall, "", Integer, typeof(N)))
        M isa Integer || throw(TypeError(:NucBall, "", Integer, typeof(M)))
        N ≥ 1 || throw(DomainError(N, "N must be a positive integer"))
        M ≥ 1 || throw(DomainError(M, "M must be a positive integer"))
        new{M, N}()
    end
end
```

As a set of matrices, we store the number of rows (`M`) and columns (`N`).

Other sets are defined similarly,

- Vector sets:
  - `OneBall{N}`: $\mathbb{B}_1 := \left\{ \pm e_i \,\middle|\, i \in [1, N] \right\}$
  - `Simplex{N}`: $S_N := \left\{ e_i \,\middle|\, i \in [1, N] \right\}$
  - `TwoBall{N}`: $\mathbb{B}_2 := \left\{ a \in \mathbb{R}^N \,\middle|\, \|a\|_2 = 1 \right\}$
- Matrix sets:
  - `NucBall{M, N}`: $\mathbb{B}_* := \left\{ uv^\top \,\middle|\, u \in \mathbb{R}^M,\ v \in \mathbb{R}^M,\ \|u\|_2 = \|v\|_2 = 1 \right\}$
  - `Spectraplex{N}`: $S_{N,N} := \left\{ uu^\top \,\middle|\, u \in \mathbb{R}^N,\ \|u\|_2 = 1 \right\}$
  - `MatrixOneBall{M, N}`: $\mathbb{B}_1^{M,N} := \left\{ e_{ij} \,\middle|\, i \in [1, M],\ j \in [1, N] \right\}$
  - `BlockNucBall{M, N, BM, BN}` is the set of $M \times N$ matrices where one $BM \times BN$ block is a `NucBall{BM, BN}` and all other entries are zero.

Note that all of these basic sets are zero-sized. They carry no runtime information and largely exist as markers for dispatch, and given the type of a set the only value of that type can always be trivially constructed (e.g. given `T::Type = OneBall{N}`, we can always get the only value `x` such that `x::OneBall{N}`).

5.1.2. Atoms

Atom types are a subtype of

```julia
abstract type Atom{Set <: AtomicSet}
end
```

where `Set` is the type of the set to which the atom belongs.

Atoms are parameterized by their array type, for example with the two-ball:

```julia
struct TwoBallAtom{Set <: TwoBall, Vt <: AbstractVector} <: Atom{Set}
    z::Vt
end
```

Some sets have further restrictions on the type of representation, such as the atom for the spectraplex, where atoms are all matrices are of the form $uu^\top$ (implemented here in the `SymVecOuterProduct` type):

```julia
struct SpectraplexAtom{Set <: Spectraplex, Repr <: SymVecOuterProduct} <: Atom{Set}
    repr::Repr
    function SpectraplexAtom(::Set, mat::SymVecOuterProduct) where
            {N, Set <: Spectraplex{N}}
        ...
    end
end
```

To get atoms as a tensor in their space (i.e., as a subtype of `AbstractArray`), we have the `materialize` function.

```julia
a = expose(A, z)
a isa Atom
x = materialize(a)
x isa AbstractArray
```

5.1.3. Faces

A face abstraction must be able to represent an infinite number of points, but are always a convex combination of k atoms. Therefore, we represent them as a map from k coefficients to a tensor in the space of the set.

There is a single `Face` type which represents faces from any set:

Julia
```
struct Face{Set <: AtomicSet, Map <: LinearMap, Rank <: Union{Int, Tuple, Array}}
    set::Set
    map::Map
    rank::Rank

    ...
end
```

The map implements the mapping from coefficients to tensors and the RANK is the number of atoms in the face. Its meaning is as follows:

- For `SumSet`, it is either an `Int` (in which case all of the summed sets have the same rank) or a `Tuple` (in which case there is an entry in the tuple for each set in the sum).
- For `BlockNucBall`, it is either an `Int` (in which case each of the blocks has the same rank) or an `Array` (in which case the size of the array should match the number of blocks).
- For `MappedSet`, it's whatever the inner set uses.
- For other sets, it's always an `Int`.

Note that `map::LinearMap`. For atoms, we are always working in the natural spaces of sets (see Subsection 5.3.1 for a description of these maps). However, due to requirements on faces for `SumSet`, for faces we currently need to work in a flattened space—using an isomorphism between the natural space and vectors. See Subsection 5.3.2 for a discussion of the implementation of those maps.

This has the unfortunate consequence that requesting an output from the map for a matrix set yields a vector:

Julia
```
A = NucBall{5, 6}()
z = rand(5, 6)      # our data
size(z) == size(A)  # true
F = face(A, rand(5, 6), maxrank=2)
x = F * [0.5, 0.5]   # this represents a matrix in the same space as A, but:
x isa AbstractVector # true
size(x) == size(A)   # false
```

The `naturalspace` function is provided to use the inverse of the isomorphism above to get data back into its native space:

Julia
```
X = naturalspace(A, x)
X isa AbstractMatrix  # true
size(X) == size(A)    # true
```

5.1.4. Operations

Our basic tools (expose, support, gauge, face) are, idiomatically for Julia, generic functions which use the sets for dispatch. For example, our library implements:

```julia
expose(A::OneBall, z::AbstractVector) = ...
expose(A::NucBall, z::AbstractMatrix) = ...
```

and so on. This makes it very easy to be generic over any atomic set.

All the basic sets have a closed form expression for the gauge function.

5.2. Composite Set Representations

5.2.1. Mapped Sets

Mapped sets store two components: the set that has been mapped (the "inner" set) and the map. Unlike for the basic sets, atoms are lazily evaluated in a sense, as they store an atom of the inner set. The map is actually applied to the atom when `materialize` is called.

Mapped sets are built in the same way as maps are applied in the rest of Julia: through use of the `*` operator. Atoms can be mapped as well:

```julia
A  = OneBall{5}()
MA = M * A              # build a mapped set
a  = M * expose(A, x)   # map an atom
ma = expose(MA, y)

set(a) == set(ma)       # true
```

`MappedSet` has a closed-form gauge function if the inner set does. See Section 2.3 for an explanation.

5.2.2. Sum of Sets

In terms of implementation, the Minkowski sum of sets is the most complex set type. As we must be able to store sets of different types, we require heterogeneous lists. The first

thing that one might try is to store the sets in an `Vector{AtomicSet}`, which does work. However, doing so prevents static dispatch on any of the elements in the sum, in e.g., the `expose` function. In addition, this stores elements on the heap which, as we'll see, is unnecessary and adds a pointer indirection, which may slow down computations.

We're dealing with a heterogeneous list, but we don't require mutability. So, a next natural choice would be to use a `Tuple`, like so:

Julia
```
struct SumSet{K, Sets <: NTuple{K, AtomicSet}}
    sets::Sets
end
```

This works better and can allow type inference to succeed, at least in construction of sets. This works because tuples are covariant with respect to their type parameters, unlike other types in Julia. However, this too fails to be type-stable in some situations which require iteration over sets in the sum. So, we need a representation that satisfies these conditions:

- Type-stable in every situation we address
- Stack-allocatable
- Has associative structural equality—`A + (B + C)` should have identical structure (and type) to `(A + B) + C` to make equality comparisons trivial

We accomplish this using a structure that is conceptually a type-level cons list (but which compiles to essentially the same layout as a `Tuple`):

Julia
```
struct SumSet{Head <: AtomicSet, Tail <: AtomicSet} <: AtomicSet
    head::Head
    tail::Tail
end
```

Here, we add the restriction that `Head` may never be a subtype of `SumSet`, but `Tail` may. This way `head` is always the first item in the sum, and `tail` is the rest of the sum. We enforce this using four rules for addition:

Julia
```julia
# default case
+(A::AtomicSet, B::AtomicSet) = SumSet(A, B)

# this is the same case as above, but we need this method to disambiguate
# (this would be the first `+` in A + (B + C))
+(A::AtomicSet, B::SumSet) = SumSet(A, B)

# adding a new set to the end of a sum expression
# (this would be the second `+` in (A + B) + C)
# note: this one's recursively calling + on the rest of the expression
+(A::SumSet, B::AtomicSet) = SumSet(head(A), tail(A) + B)

# finally, merging two sum expressions
# (this would be the middle `+` in (A + B) + (C + D))
# this one's also recursive
+(A::SumSet, B::SumSet) = SumSet(head(A), tail(A) + B)
```

Now, we can define functions which operate on every element in the sum recursively. The base case is always `Tail <: AtomicSet`, and other cases always have `Tail <: SumSet`. Consider the implementation of `foreach` as an example, which calls `f` on each element in the sum and discards the result:

Julia
```julia
# recursive case
function foreach(f, a::SumSet{Head, Tail}) where
        {Head <: AtomicSet, Tail <: SumSet}
    f(head(a))
    foreach(f, tail(a))
end

# base case
function foreach(f, a::SumSet{Head, Tail}) where
        {Head <: AtomicSet, Tail <: AtomicSet}
    f(head(a))
    f(tail(a))
    nothing
end
```

5.3. Linear Maps

5.3.1. Maps in their Natural Spaces

Previous versions of AtomicSets.jl used the Julia package `LinearMaps.jl`[19] to represent all maps. In `LinearMaps.jl`, all linear maps are in $\mathbb{C}^n \to \mathbb{C}^m$; that is, they map vectors in

$\mathbb{R}^m$ (or $\mathbb{C}^m$) to vectors in $\mathbb{R}^n$ (or $\mathbb{C}^n$). They essentially act as matrices that we don't want to explicitly instantiate (and so do not implement indexing methods), but in particular are required to have a size and act on vectors. We have maps that act on different spaces: for example the `TraceMap` which maps symmetric $k \times k$ matrices to $\mathbb{R}^{m \times n}$, or the `ScatterMap` which maps length-k vectors to $m \times n$ matrices. These cannot be easily represented in the `LinearMaps.jl` framework, in part because they do not have a size which is a 2-tuple.

So, we provide our own map types that mostly behaves similarly. Instead of having a `size`, they have explicit `codomainsize` and `domainsizes` (which correspond to covariant and contravariant indices when treating them as tensors). They are required to have `adjoint` defined, as well as in-place multiplication with tensors of the correct sizes. They also have an element type for type-promotion purposes, but this is allowed to be `Union{}` for e.g. mask types, which should not change the input element type.

5.3.2. Flattenings

Due to constraints on input types for the implementation of the box-constrained least squares solver[20] in the linesearch (Equation 3.6), we require the maps implementing faces to be essentially `<: LinearMap` from the LinearMaps.jl package. This directly contradicts the goal above and the larger goal of representing objects in their natural spaces as much as possible, but due to time constraints this was the only available solution. As such, we provide a `FlattenLinearMap <: LinearMap`, which wraps maps that have the above methods implemented and internally applies an isomorphism to and from vectors for both the codomain and domain.

5.4. The Level Set Method

The level set algorithm, described in Chapter 3, is relatively straightforward to implement using the primitives we have already established. It is divided into several important functions.

Firstly, we have a function `level_set` which computes all of Algorithm 3.1. However, it is a trivial wrapper over the `gauge!`[3] function, which provides an abstraction for computing using the more general level-set form. Deconvolution, for example (Problem 3.8), is

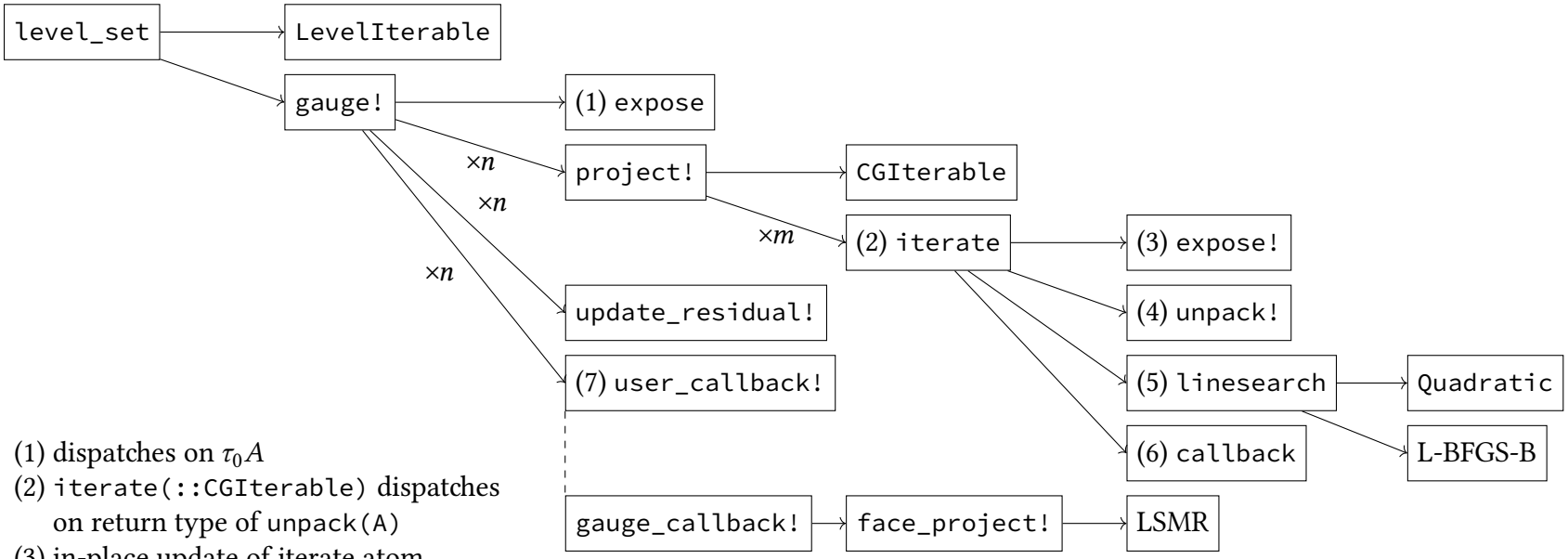
[3] By convention, procedures in Julia which mutate their arguments have bang (`!`) in their names. In this case, as in `project!`, we are mutating a struct which stores iterate information.

implemented in a callback. This way, `gauge!` could be modified to, for example, simply return an underestimate for the $\gamma_{\mathcal{A}}(x)$ for sets which have no closed form expression for the gauge.

Secondly, we have the `project!` function which computes approximations to projections onto atomic sets using only `expose` and `support`. As discussed in Subsection 3.2.1, there are two possible cases for project: those where we have a single set, single iterate, and simple linesearch; or where we have a sum of sets, a set of iterates, and a complex linesearch. These are accomplished using an `unpack` function. This takes an atomic set and returns either the set back or a tuple of sets (in the second case). This works in the same way for atoms. In this way, `gauge!` is agnostic to the number of iterates, but `project` can dispatch on the type of the iterate (whether it is an array or a tuple of arrays). The multiple line search problem (Equation 3.6) is solved using LBFGSB.jl[21], which is a wrapper around the L-BFGS-B algorithm implementation from [22].

Finally, the deconvolution problem is solved in a callback. Problem 3.8 is solved using the LSMR[23] method from the IterativeSolvers.jl[20] package.

The call graph for the level-set method is shown in Figure 5.1. The bulk of the algorithm is implemented in the `gauge!` method, which updates an iterator in-place and has its behaviour modifiable through a callback which updates the iterate. The name is somewhat misleading; at the moment it computes a suboptimal lower bound to the gauge function. However, it could be simply modified to compute the gauge properly. The projection code is mainly in the `iterate` method for a `CGIterable` type.



(1) dispatches on $\tau_0 A$
(2) `iterate(::CGIterable)` dispatches on return type of `unpack(A)`
(3) in-place update of iterate atom
(4) only called for unpackable method of `iterate`
(5) dispatches on return type of `unpack`
(6) user-defined callback for early termination of project
(7) user-defined callback for modification of `LevelIterable`, using `recover_callback!` for `level_set`

Figure 5.1 Call Graph of the Level Set Method.

6.
Examples

Here we present some examples of using AtomicSets and the level set method.

Example 6.1 (Frank-Wolfe)

The Frank-Wolfe algorithm[14,24] finds the minimum of a convex function f in a convex domain $\mathcal{D}$. The algorithm is:

Algorithm 6.1 (Frank-Wolfe)

(1) **Inputs:** ∇f, $\mathcal{D}$
(2) **Let** $x^{(0)} \in \mathcal{D}$
(3) **For** $k = [0, K)$, **do:**
(4) $\quad s \leftarrow \arg\max_{s \in \mathcal{D}} \langle s, \nabla f(x^{(k)}) \rangle$
(5) $\quad \gamma \leftarrow \frac{2}{k+2}$
(6) $\quad x^{(k+1)} \leftarrow (1-\gamma)x^{(k)} + \gamma s$

Noting that $\arg\max_{s \in \mathcal{D}} \langle s, \nabla f(x) \rangle \equiv \arg\min_{s \in \mathcal{D}} \langle s, -\nabla f(x) \rangle$, and when $\mathcal{D} = \operatorname{conv}(\mathcal{A} \cup \{0\})$ we have it equivalent to $\operatorname{expose}_{\mathcal{A}}(-\nabla f(x))$. So, we can implement it in that case using AtomicSets.jl. The following finds the minimum of f given its gradient within the domain $\operatorname{conv}(\mathcal{A} \cup \{0\})$:

Julia
```julia
function frank_wolfe(∇f, domain::AtomicSet, x0::AbstractArray, K::Integer)
    x = copy(x0)
    for k in 1:K
        a  = expose(domain, -∇f(x)) |> materialize
        γ  = 2 / (k + 1)
        x .= ((1 - γ) .* x) .+ (γ .* a)
    end
    x
end
```

See an example plot of the iterates for this algorithm in Figure 6.1.

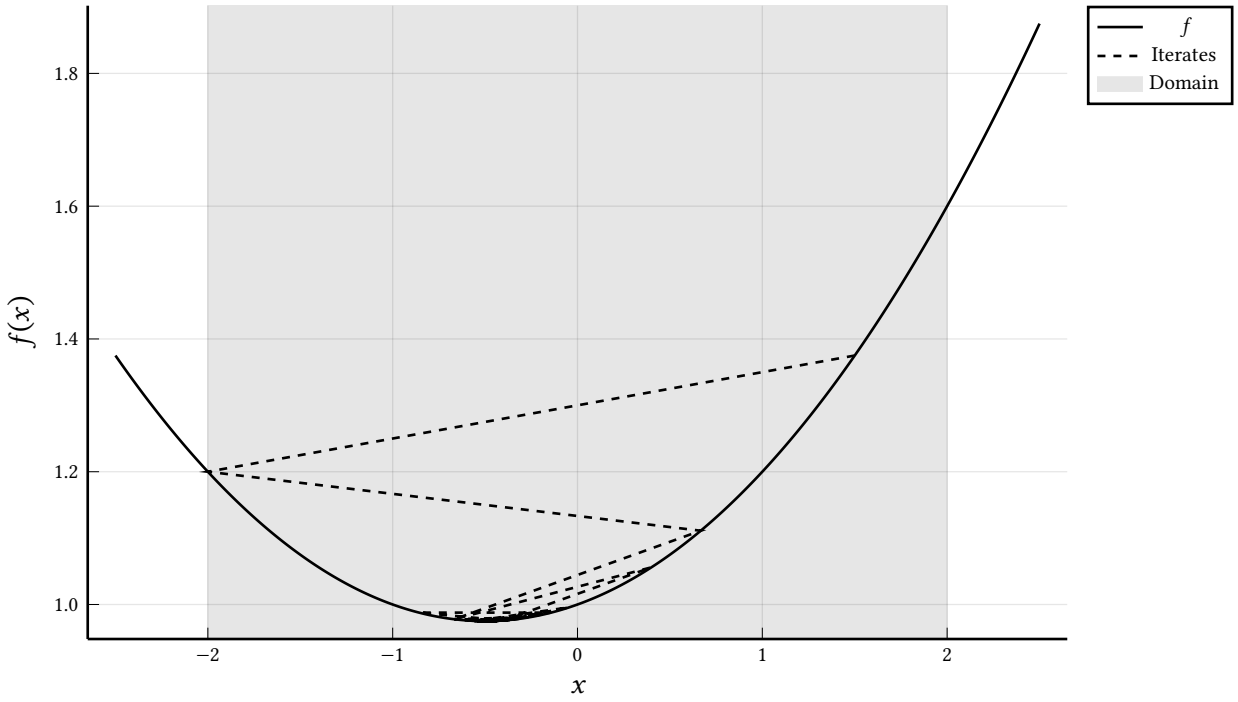


Figure 6.1 Frank-Wolfe in AtomicSets.jl. This example uses a quadratic objective in one dimension, with simple box constraints. The region within the domain has been highlighted.

Example 6.2 (Linear Programming)

We'll start with the standard linear programming problem:

$$\min_x c^\top x + \delta_{\geq 0}(x) \quad \text{s.t.} \quad Mx = b,$$

where c is a positive cost vector $\in \mathbb{R}^n_+$, $M \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$ and $\delta_{\geq 0}$ is the indicator function for non-negatives. We can note that this objective is the gauge to an atomic set, namely $\text{Diag}(1/c)\, S_n$ (where S_n is the simplex in $\mathbb{R}^n$). So, we can rewrite our problem as

$$\min_x \gamma_{\mathcal{A}}(x) \quad \text{s.t.} \quad \|Mx - b\| \leq 0.$$

Now, we can solve it using `level_set`:

Julia
```
function solve_linear_program(M, b, cost, ::Val{N}; kwargs...) where {N}
    A = Diagonal(1 ./ cost) * Simplex{N}()
    solution = level_set(M, A, b, 0.; kwargs...)
    primal(solution)
end
```

A plot of the norm of the residual (for a random 500 × 250 problem) against iteration number for this algorithm is plotted in Figure 6.2.

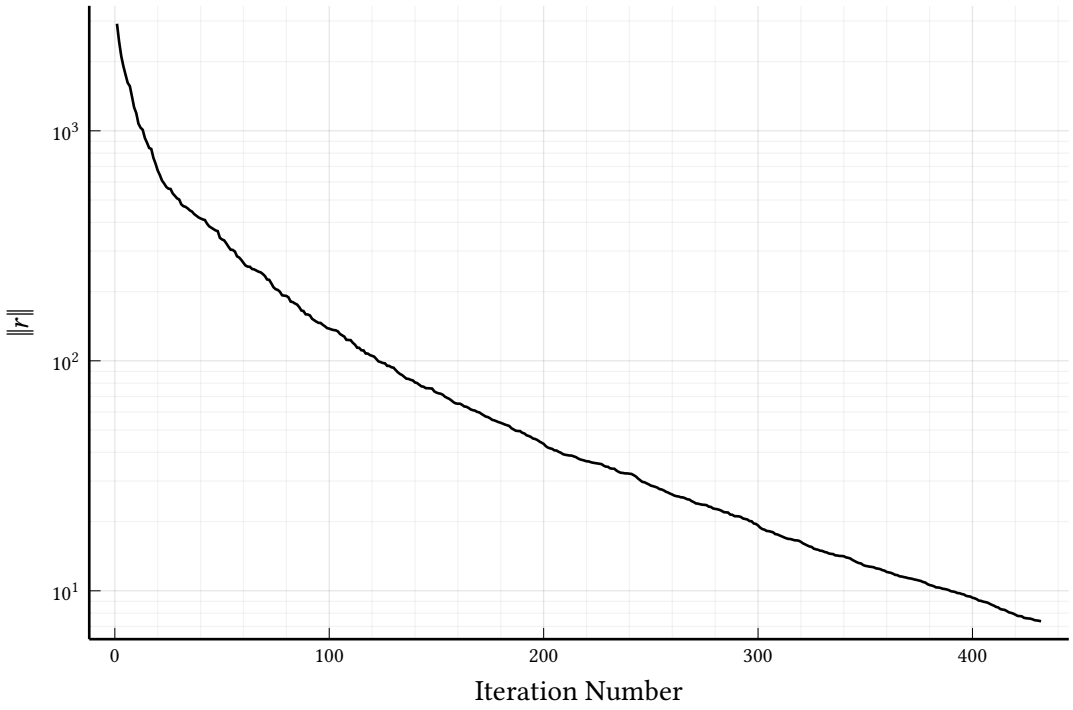


Figure 6.2 Linear Programming in AtomicSets.jl. Plotted here is the iteration number versus norm of the residual (plotted logarithmically).

Example 6.3 (Star-Galaxy Separation)

This example demonstrates the algorithm in [11]. The data (Figure 6.3) is a picture of a galaxy with stars in front of it. The stars are relatively sparse in position, but the galaxy is sparse in frequency[4]. We are able to separate them using the following code (this separation is also shown in Figure 6.3):

[4] In other words, it is sparse in the traditional sense under the inverse discrete cosine transform.

Julia

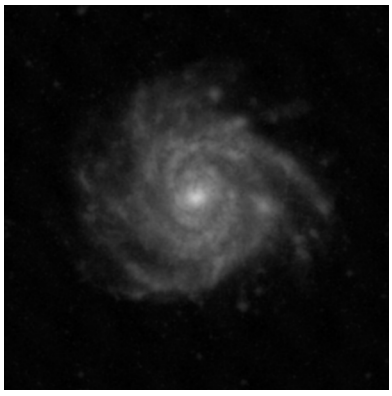
```
b      = get_star_galaxy_image()
M, N   = size(b)
ks, kd = get_star_galaxy_counts()

# set for sparsity in position
As = let A = OneBall{N * M}(), τs = gauge(A, vec(b))
  τs * A
end

# linear map for performing the (I)DCT
Q = LinearMap(idct, dct, n * m, n * m)

# set for sparsity under IDCT
Ad = let A = OneBall{N * M}(), τd = gauge(A, Q'vec(b))
  τd * Q * A
end

# set for whole signal
A = As + Ad

# solve
solution = level_set(I(m * n), A, vec(b), 0.0, maxrank=(ks, kd); kwargs...)
stars, galaxy = primal(solution)
```

Example 6.4 (Chessboard Separation)

Similar to the previous example, we can separate images which are low-rank and those that are sparse in position. In this example, we'll separate an image of a chessboard from the chess pieces on it. Note that unlike in the previous example, we'll deal with the image as a matrix rather than as a vector. The separation is displayed in Figure 6.4.

The original signal b, with both stars and galaxy.

The portion of the signal which is sparse in position-space.



The portion of the signal which is sparse in frequency-space.

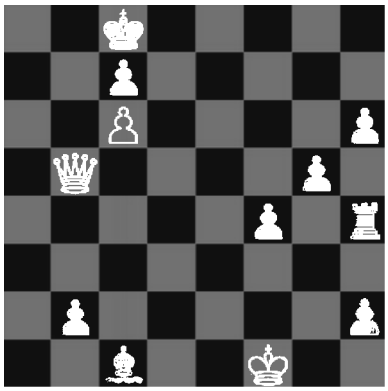
Figure 6.3 Star-Galaxy Separation using the level-set method. Above is the original signal. Below are the separated signals, showing the two parts.

Julia

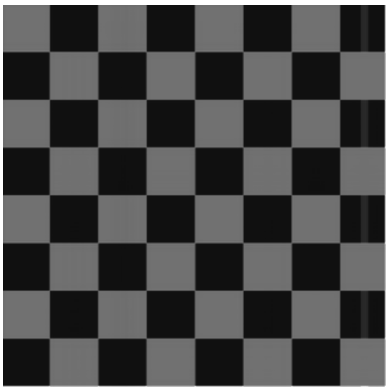
```julia
b      = get_chessboard_image()
M, N   = size(b)
ks, kr = get_chessboard_counts()

# set for sparsity in position
As = let A = MatrixOneBall{M, N}()
    gauge(A, xs) * A
end

# set for low-rankness
Ar = let A = NucBall{M, N}()
    gauge(A, xr) * A
end

# set for whole signal
A = As + Ar

# solve
solution = level_set(SizedIdentity((M, N)), A, b, 0.0, maxrank=(ks, kr); kwargs...)
board, pieces = primal(solution)
```

Example 6.5 (Sparse Signal Multiplexing)

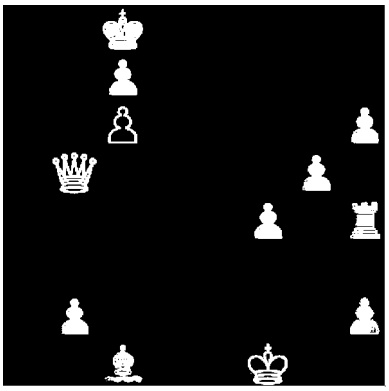
Finally, we'll show the (de-)multiplexing of several sparse signals. l signals of length n are generated, each of which are k-sparse. They are then each randomly rotated. They can be recovered using the level-set algorithm if the rotations are known, as we then have a corresponding set to which each of the messages are sparse.



The original signal b, with both stars and galaxy.



The portion of the signal which is low-rank.



The portion of the signal which is sparse in position.

Figure 6.4 Chessboard separation using the level-set method. Above is the original signal. Below are the separated signals, showing the two parts.

Julia

```julia
k = 8     # messages are k-sparse
l = 3     # there are l messages
n = 1024  # of length n

# generate random unitary transform
gen_rotation() = Matrix(qr(rand(n, n)).Q)
# generate random k-sparse message of length n
gen_message()  = let msg = zeros(n)
    msg[randperm(n)[1:k]] .= rand(k) .+ 0.5
    msg
end

# generate messages and corresponding rotations
messages  = [gen_message() for _ in 1:l]
rotations = [gen_rotation() for _ in 1:l]

# multiplex (sum after rotation)
multiplexed = sum(rotations .* messages)

# build sets for demxuxing
sets    = [norm(messages[i], 1) * rotations[i] * OneBall{n}() for i in 1:l]
sum_set = sum(sets)

# perform the demuxing
solution  = level_set(I(n), sum_set, multiplexed, 0.0, maxrank=(k, k, k); kwargs...)
demuxed   = primal(solution)
recovered = adjoint.(rotations) .* demuxed
```

The original and recovered signals are plotted in Figure 6.5.

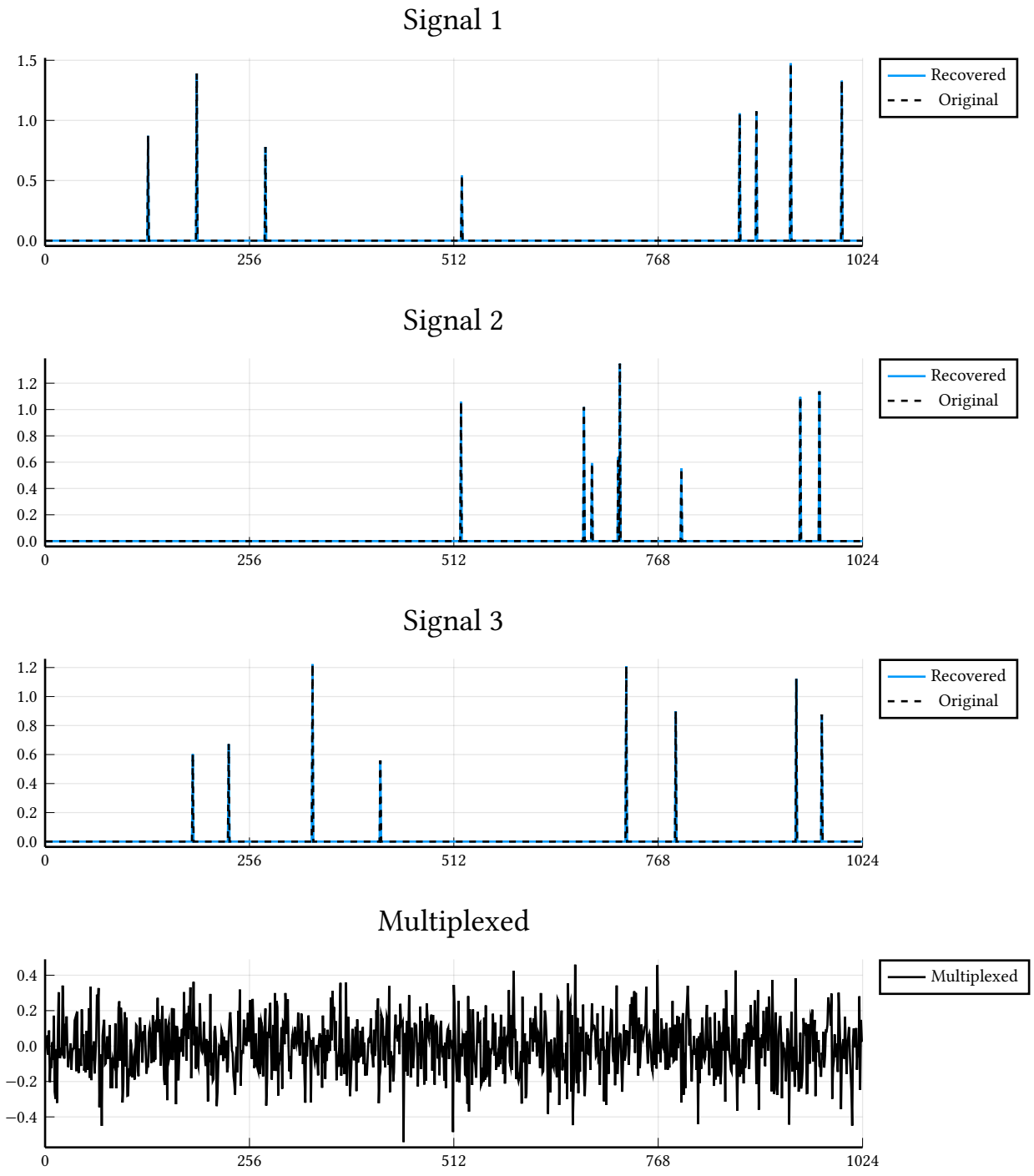


Figure 6.5 Sparse Signal Multiplexing Results

7.
Future Directions

7.1. Using Value Types To Check Set Sizes

We've already shown there are some advantages to having the size of sets encoded in their type (such as the `OneBall{N}`), but we're currently missing a major part of that: the `MappedSet` and `SumSet` do not have sizing as parts of their type. This means that:

Julia
```
A1 = rand(5, 5) * OneBall{5}()
A2 = rand(4, 5) * OneBall{5}()
typeof(A1) == typeof(A2)  # true, but
size(A1) == size(A2)      # false
```

Having every single map be a separate type is impractical (and therefore so is `typeof(T) == typeof(U)` → `T == U` for atomic sets), but sizes could be included in the definition of atomic sets like so:

Julia
```
abstract type AtomicSet{Size <: Tuple}
end

size(::AtomicSet{Tuple{N}}) where {N} = (N,)
size(::AtomicSet{Tuple{M, N}}) where {M, N} = (M, N)
# etc. for every order tensor we need

struct OneBall{N} <: AtomicSet{Tuple{N}}
    ...
end

struct NuclearBall{M, N} <: AtomicSet{Tuple{M, N}}
    ...
end
```

This would mean first that the sum set could also be statically sized, and that rather than checking sizes in the constructor we could only define addition for same-sized sets:

Julia
```
struct SumSet{S, Head <: AtomicSet{S}, Tail <: AtomicSet{S}} <: AtomicSet{S}
    head::Head  # never <: SumSet
    tail::Tail  # maybe <: SumSet
end

+(A::AtomicSet{S}, B::AtomicSet{S}) where {S} =
    SumSet{S, typeof(A), typeof(B)}(A, B)
+(A::SumSet{S}, B::AtomicSet{S}) where {S} =
    head(A) + (tail(A) + B)
+(A::AtomicSet{S}, B::SumSet{S}) where {S} =
    SumSet{S, typeof(A), typeof(B)}(A, B)
+(A::SumSet{S}, B::SumSet{S}) where {S} = let h = head(A), t = tail(A) + B
    SumSet{S, typeof(h), typeof(t)}(h, t)
end

OneBall{5}() + OneBall{5}()  # ok, no checks needed
OneBall{6}() + OneBall{5}()  # undefined
```

Now, instead of relying on compiler optimizations to disallow undefined operations and elide checks when correct, we ensure that only possible operations are defined in the first place. This would also work with static analysers like JET.jl (as discussed in Section 4.2): when type inference succeeds, we would then be able to statically check sizes.

To use this same technique for `MappedSet`, we require the next section.

7.2. Representing Maps with Tensors

As discussed previously, the existing technique of treating linear maps as essentially matrices that we don't want to instantiate fails for cases where we want to work in the natural space of maps like $M : \mathbb{R}^{m,n} \to \mathbb{R}^{k}$. We solved this problem by removing the requirement for `size`, and instead adding `domainsize` and `codomainsize`. These are hinting at a better possible formulation of our maps: as tensors with separate contravariant and covariant indices.

This representation builds on that achieved by [25]. However, that package stores the dimension of spaces as runtime information only, and for this example we want to consider the space as compile-time information (as before). We might then add the supertype of these maps like so:

Julia
```
abstract type TensorMap{Codomain <: Tuple, Domain <: Tuple}
end

const Tensor{Size <: Tuple} = TensorMap{Size, Tuple{}}
```

Then, one would define application of the maps like:

```julia
mul!(
    out::Tensor{Codomain},
    map::TensorMap{Codomain, Domain},
    x::Tensor{Domain}
) where {Codomain, Domain} = ...
```

If any maps were defined over the complex numbers (or anything besides product spaces of real vector spaces), care would have to be taken as the coindices are in the dual space. For real vector spaces, we can identify the dual space with the primal and not worry about the distinction, and contraction with the metric is trivial.

Some of the functionality of `LinearMaps.jl` would have to be replaced, namely:

- Concatenation (for the face of the sum set)
- Efficient uniform scaling
- Composition

This could allow for representing faces in their natural domain without use of flattenings, but it would not solve the problem of the interface to LSMR, which also assumes maps are essentially matrices.

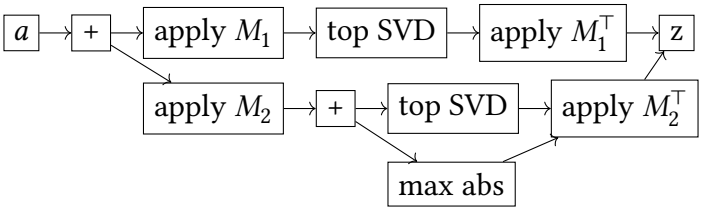
7.3. Atom Types as Computation Graphs

We've discussed already how operations like `expose` and `face` are already partially lazy for the sum and mapped sets, but essentially eager for every basic set (`materialize` is usually trivial for them). If this could be made fully lazy, then `Atoms` would represent a computation graph. `materialize` could then be modified to, for example, compute an `expose` operation on a GPU. We can be sure that atoms are DAGs simply because they are all immutable (`structs`, not `mutable structs`).

In essence, this atom:

```julia
A = (M1 * NucBall{M, N}()) + (M2 * (NucBall{M, N}() + MatrixOneBall{M, N}()))
a = expose(A, z)
```

would be represented by this dependency graph:



A similar graph could be attained for `gauge`, but it would require adding something along the lines of an `Atom` type in that a lazy representation of the operations is needed.

Bibliography

[1] V. Chandrasekaran, B. Recht, P.A. Parrilo, and A.S. Willsky, The Convex Geometry of Linear Inverse Problems, (2010).

[2] Z. Fan, H. Jeong, Y. Sun, and M.P. Friedlander, Atomic Decomposition via Polar Alignment: The Geometry of Structured Optimization, *Foundations and Trends in Optimization* **3**(4), 280-366 (2020).

[3] E.J. Candes, J. Romberg, and T. Tao, Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information, *IEEE Transactions on Information Theory* **52**(2), 489-509 (2006).

[4] S.S. Chen, D.L. Donoho, and M.A. Saunders, Atomic Decomposition by Basis Pursuit, *SIAM Journal on Scientific Computing* **20**(1), 33-61 (1998).

[5] D.L. Donoho, For most large underdetermined systems of linear equations the minimal 1-norm solution is also the sparsest solution, *Communications on Pure and Applied Mathematics* **59**(6), 797-829 (2006).

[6] D.L. Donoho, Compressed sensing, *IEEE Transactions on Information Theory* **52**(4), 1289-1306 (2006).

[7] B. Recht, M. Fazel, and P. Parrilo, Guaranteed Minimum-Rank Solutions of Linear Matrix Equations via Nuclear Norm Minimization, *SIAM Review* **52** (2007).

[8] E.J. Candès and B. Recht, Exact matrix completion via convex optimization, *Found. Comut. Math.* **9**(6), 717–772 (2009).

[9] J. Bezanson, S. Karpinski, V.B. Shah, and A. Edelman, Julia: A Fast Dynamic Language for Technical Computing, (2012).

[10] A. Y. Aravkin, J. V. Burke, D. Drusvyatskiy, Friedlander M. P., and S. Roy, Level-set methods for convex optimization, *Mathematical Programming* **174**(1-2), 359–390 (2018).

[11] Z. Fan, H. Jeong, B. Joshi, and M.P. Friedlander, Polar deconvolution of mixed signal, *IEEE Transactions on Signal Processing* **70** 2713–2727 (2022).

[12] R.T. Rockafellar, *Convex Analysis* (Princeton University Press, 1970).

[13] Z. Fan, Bundle-type methods for dual atomic pursuit, (2019).

[14] M. Jaggi, Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization, (2013).

[15] E. van den Berg and M.P. Friedlander, Probing the Pareto frontier for basis pursuit solutions, *SIAM Journal on Scientific Computing* **31**(2), 890-912 (2008).

[16] E. van den Berg and M.P. Friedlander, Sparse Optimization with Least-Squares Constraints, *SIAM Journal on Optimization* **21**(4), 1201-1229 (2011).

[17] B.K. Rosen, M.N. Wegman, and F.K. Zadeck, Global value numbers and redundant computations, In POPL '88 (1988).

[18] 門脇 宗平, 言語Julia型推論用型静的検出, (2020).

[19] LinearMaps.jl, *https://github.com/JuliaLinearAlgebra/LinearMaps.jl* (2014).

[20] IterativeSolvers.jl, *https://github.com/JuliaLinearAlgebra/IterativeSolvers.jl* (2013).

[21] LBFGSB.jl, *https://github.com/Gnimuc/LBFGSB.jl* (2018).

[22] C. Zhu, R.H. Byrd, P. Lu, and J. Nocedal, Algorithm 778: L-BFGS-B, *ACM Transactions on Mathematical Software* **23**(4), 550–560 (1997).

[23] D.C.L. Fong and M. Saunders, LSMR: An Iterative Algorithm for Sparse Least-Squares Problems, *SIAM Journal on Scientific Computing* **33**(5), 2950-2971 (2011).

[24] M. Frank and P. Wolfe, An algorithm for quadratic programming, *Naval Research Logistics Quarterly* **3**(1-2), 95-110 (1956).

[25] TensorKit.jl, *https://github.com/Jutho/TensorKit.jl* (2014).

A.
Index

a
abstract type **23**
alignment **14**
atomic decomposition **1**, **7**, **10**, **17**, **18**
atomic set **1**, **29**
atoms **1**, **31**

c
concrete type **23**
convex combination **3**, **4**
convex set **3**

e
expose **10**, **15**, **16**, **34**

f
face **10**, **32**
flattening **33**

g
gauge **12**, **13**, **14**, **18**, **34**

h
half-space **4**, **5**

l
level-set method **18**, **19**, **37**

m
methods **22**
multiple dispatch **22**

p
promotion **24**

r
rank **33**

s
specialization **24**
support function **9**, **14**, **16**, **34**
support set **18**
supporting hyperplanes **3**, **5**

t
type inference **22**
type union **24**

v
value type **24**, **30**, **49**

B.
Notation

Vectors are written u, v, scalars are written u, v, and matrices are written U, V. Scalars can will be lowercase unless it is clear from context. Sets are written $\mathcal{U}, \mathcal{V}$, except for the reals and complex numbers which are written $\mathbb{R}, \mathbb{C}$ respectively. If a set could be either the reals or the complex numbers, we write that $\mathbb{K}$.

Some important functions:

- $\|x\|_p : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ are l_p norms, and $\|X\|_p : \mathbb{R}^{n \times m} \to \mathbb{R}_{\geq 0}$ are the Schatten p-norms. $\|X\|_* = \|X\|_1$ is the nuclear norm.
- $\mathbb{B}_\square$ is the unit-ball of a norm $\|\cdot\|_\square$.
- Diag : $\mathbb{K}^n \to \mathbb{K}^{n \times n}$: takes a vector to a diagonal matrix whose elements match the vector.
- diag : $\mathbb{K}^{n \times n} \to \mathbb{K}^n$: inverse of Diag.

Code items are typed in `monospaced` font, and follow Julia conventions: type names are `UpperCamelCase`, variable names are `snake_case`, constants are `SCREAMING_SNAKE_CASE`. If a variable `x` has type `T`, we write that `x::T`. If type `T` is a subtype of `U`, we write that `T <: U`.