SODA: an Open-Source Library for Visualizing Biological Sequence Annotation

Jack W. Roddy
The University Of Montana

Travis J. Wheeler
The University Of Montana

SODA - AN OPEN SOURCE LIBRARY FOR VISUALIZING BIOLOGICAL SEQUENCE ANNOTATIONS

By

Jack W Roddy

Bachelor of Science, The University of Montana, Missoula, MT, 2019

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

Autumn 2020

Approved by:

Ashby Kinch Ph.D., Dean
Graduate School

Travis Wheeler Ph.D., Chair
Computer Science

Doug Brinkerhoff Ph.D.
Computer Science

Cory Palmer Ph.D.
Mathematical Sciences

Roddy, Jack W, M.S., January 2021 Computer Science

SODA - an open source library for visualizing biological sequence annotations

Chairperson: Travis Wheeler

Genome annotation is the process of identifying and labeling known genetic sequences or features within a genome. Across the various subfields within modern molecular biology, there is a common need for the visualization of such annotations. Genomic data is often visualized on web browser platforms, providing users with easy access to visualization tools without the need for installing any software or, in many cases, underlying datasets. While there exists a broad range of web-based visualization tools, there is, to my knowledge, no lightweight, modern library tailored towards the visualization of genomic data. Instead, developers charged with the task of producing a novel visualization must either adopt a complex system or fall back on general purpose visualization frameworks. Here, I present SODA, a web-based genomic annotation visualization library implemented in TypeScript as an abstraction over D3. SODA is designed to be lightweight and flexible, empowering developers with the tools to easily create customized and nuanced genomic visualizations.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor and committee chair, Travis Wheeler. He has made my journey through graduate school remarkably enjoyable. I am endlessly grateful to George Lesica for providing guidance on the work presented here. I'd also like to acknowledge the support of my other committee members, Doug Brinkerhoff and Cory Palmer, both of whom provided some of my favorite classes during my time at the University of Montana. It's unlikely that this project would have ever begun if it weren't for the time I spent at the Institute for Systems Biology with Robert Hubley, Arian Smit, and Jeb Rosen. Finally, I would like to thank every other member of the Wheeler Lab for being so awesome and fun to work around.

TABLE OF CONTENTS

CODE LISTINGS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Genome annotation is the process of identifying and labeling known genetic sequences or features within a genome. Across the various sub-fields within modern molecular biology, there is a common need for the visualization of such annotations. Annotation visualizations can be placed into two broad categories: genome browsers, which are designed to support on-demand visualizations of queries made to large-scale annotation databases; and localized visualizations, in which a one-off figure is produced, typically from a local data file. The work presented in this thesis is concerned with visualizing genomes in linear context; many bacterial genomes are circular, and perhaps best visualized in that context, but that is beyond the scope of this work. There is often a great deal of similarity in the visual content necessary to effectively visualize annotations that emerge from distinct datasets. Almost ubiquitously, linear genomic visualizations are plotted in two dimensions. Along the horizontal axis, individual annotations are plotted as glyphs that span the range of genomic coordinates that they characterize. Usually, a difference in the vertical placement of two annotations is used to prevent the visual overlap of features, but in some cases, it can have semantic meaning.

Typically, genomic visualization applications are built on web platforms, making them accessible to users without the need to install software. However, there exists no web-based library tailored for producing genomic visualizations. As such, developers have limited options when tasked with producing a novel visualization: they can either attempt to adapt an existing framework or fall back on general-purpose web visualization tools. Both avenues have a significant drawback in the time cost of development, and the adaptation route suffers from a lack of flexibility and likely requires the adoption of a technology stack. This has left a need for a lightweight and flexible library that equips developers with the tools to easily and efficiently create dynamic and interactive genomic visualizations.

Genome browser services, such as the UCSC Genome Browser [1], the ENSEMBL Genome browser[2], and JBrowse[3] leverage the commonality across annotation data to great effect: they provide a rich, cohesive environment for visualizing and comparing annotations sourced from various databases. Typically, a genome browser implements several data *tracks*, which are essentially a visualization pattern tailored to a specific type of annotation. Since tracks share the same horizontal coordinate space, a group of tracks can be stacked vertically to simultaneously display annotations from different databases that are tied to the same region of the genome. The major genome browsers are all open source, and have varying levels of support for visualizing user data that is not officially provided by the platform. However, the software environments that serve as their foundation are complex (and, in some cases, outdated), and they are not flexible in the face of novel use cases.

There exists also a number of single-purpose tools for the generation of static annotation visualizations, such as DnaPlotLib[4] and DnaFeaturesViewer[5]. Instead of providing an interactive environment that loads and visualizes data in response to database queries, these tools generally produce a localized, one-off figure from user-supplied data. While they can supply a turnkey solution out of the box, tools in this category are inherently limited to their intended use-cases, and figure customization outside a limited scope is often infeasible.

Here, I present SODA, a modern, web-based software library that aims to provide a generalized and modular framework for the generation of dynamic and interactive genomic visualizations. Rather than a tool in and of itself, SODA is a library with which to build visualization tools. SODA provides developers with a toolkit that lends itself to the creation of genome browsers, single-purpose tools, or something in between. The visualizations produced by SODA can be easily integrated with web pages, and it is simple to define interactions between a SODA visualization and other page features. The SODA API is simple and flexible, exposing both high and low-level features. Developers can easily and quickly create dynamic visualizations without a deep understanding of SODA internals. On the other hand, experienced developers can assume a considerable level of control over the fine details of the rendering process.

This purpose of this thesis is to present readers with an understanding of the full scope of what SODA is capable of, and a general idea of the amount and style of code that needs to be written to produce a visualization. It is not intended as a practical developer's guide, and, as such, may lack descriptions of some of the finer details of the SODA API. Readers may want to supplement this document with the comprehensive SODA API documentation, which can be found at `https://sodaviz.readthedocs.io/`. SODA is released as open source software under the BSD-3 licence, and is available for download on the NPM package registry (`https://npmjs.com/package/@traviswheelerlab/soda`), and the full source code can be found at `https://github.com/TravisWheelerLab/soda`.

CHAPTER 2 EXAMPLE APPLICATIONS

Details of SODA's implementation and features are described in Chapters 3 and 4, respectively. To first provide context for those details, this chapter showcases three applications that are implemented using the library. Each tool is an interactive and dynamic visualization of Transposable Element (TE) annotations. Collectively, these tools demonstrate utilization of the entirety of the current SODA feature set. While they are all similar by virtue of visualizing essentially the same type of data, each tool presents that data in a different context. In chapter 5, the implementation of each application is explored in detail.

2.1 Dfam-SODA

Dfam[6] is an open access database of Transposable Elements (TE). One of the features on the Dfam website allows users to view a visualization for the annotation of TEs in a relatively short (up to 100,000 base pairs) range on a chromosome. The previous implementation of the visualization was effective, but simple and static. The maintainers of Dfam expressed an interest in replacing it with a SODA-based visualization to improve its functionality and make the process of updating it more streamlined. I took this as an opportunity to test the integration of SODA-based technologies into a real-world website technology stack. Dfam-SODA has now replaced the previous visualization on the live Dfam website.

2.1.1 Annotation of TEs

The bulk of the annotations underlying the Dfam-SODA visualization are the result of comparing a genome to a database of known TE elements, which are broadly categorized in a familial hierarchy. The annotations are supplemented with annotation of simple tandem repeats (repetitive sequence such as 'atgatgatgatg'). Figure 2.1 depicts the Dfam classification hierarchy. A particular region of the genome will be annotated as either belonging to one of the TE families in the database, belonging to a tandem repeat class, or having no annotation.

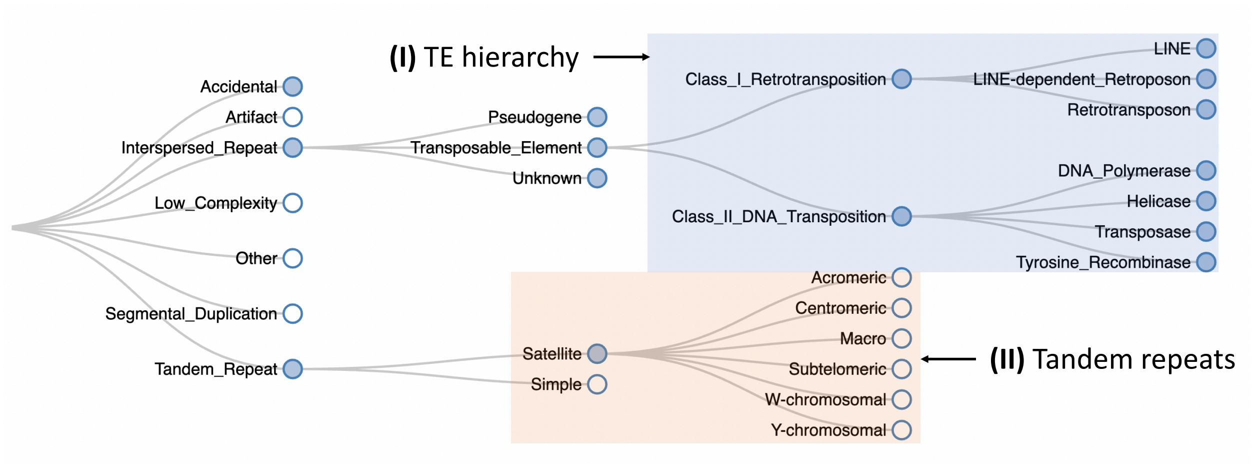


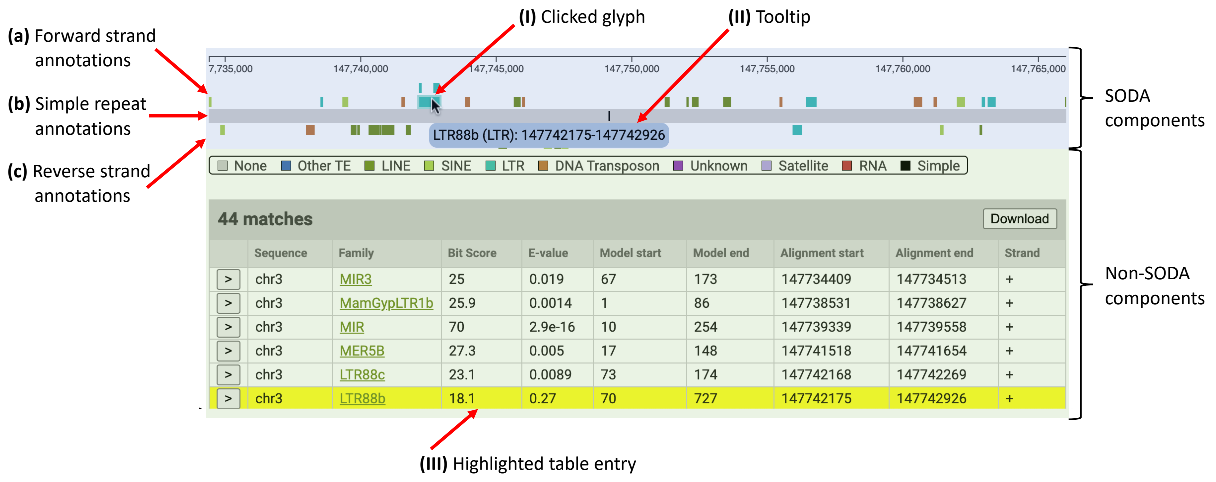
Figure 2.1: A representation of the hierarchy of Dfam classifications. At (I), the families of TEs are shown, and tandem repeats are shown at (II).

2.1.2 Description of Dfam-SODA

In the visualization, each annotation record is represented by a rectangle that is color coded by the family of the TE it represents. There are three core components of the Dfam-SODA visualization, stacked vertically in the following order:

1. The annotation of TE's on the forward strand of the chromosome
2. The annotation of simple tandem repeats
3. The annotation of TE's on the reverse strand of the chromosome

Each of the SODA components is configured with an informational tooltip and a highlight effect on hovered glyphs. Immediately following the core of the visualization are two non-SODA based components: a legend describing the colors used in the visualization and a table with a row describing each annotation. Finally, there is interactive behavior between the SODA-based components and the table: when a glyph is clicked by the user, the table scrolls to and highlights the row corresponding to the clicked glyph.



44 matches

	Sequence	Family	Bit Score	E-value	Model start	Model end	Alignment start	Alignment end	Strand
>	chr3	MIR3	25	0.019	67	173	147734409	147734513	+
>	chr3	MamGypLTR1b	25.9	0.0014	1	86	147738531	147738627	+
>	chr3	MIR	70	2.9e-16	10	254	147739339	147739558	+
>	chr3	MER5B	27.3	0.005	17	148	147741518	147741654	+
>	chr3	LTR88c	23.1	0.0089	73	174	147742168	147742269	+
>	chr3	LTR88b	18.1	0.27	70	727	147742175	147742926	+

Figure 2.2: A screenshot of the Dfam-SODA visualization embedded in the Dfam website. The core of the visualization is shown at (a), (b), and (c). In the screenshot, the glyph at (I) was hovered and clicked, resulting in the tooltip at (II), and the highlighted table entry at (III).

2.2 UCSC RepeatMasker Track

The UCSC genome browser[1] is a popular genome browser that houses visualization tracks for dozens of kinds of genomic annotation. The tracks are independently configured and submitted by various groups, and they can vary greatly in visual complexity. While the UCSC genome browser is an effective tool, it is built with dated technologies and can provide a both a frustrating user and developer experience. The UCSC rendering backend is written in C++, and it functions by producing static images on the server side and uploading those images to the client. As a result, the browser is slow to respond to user input and provides no dynamic interaction.

The RepeatMasker track in the UCSC browser visualizes the annotation of TE's and other repetitive DNA features, and is arguably one of the most nuanced and information-dense tracks in the UCSC browser. As a means to test the ease of development and performance of a SODA visualization of complex data, we recreated the visual aspects of the UCSC RepeatMasker track, supplemented with improved dynamic functionality.

2.2.1 Description of RepeatMasker-SODA

While RepeatMasker-SODA largely visualizes the same data as Dfam-SODA, it provides additional visual indicators to represent complex relationships that are not present in the Dfam annotations.

2.2.1.1 RepeatMasker Annotation Glyphs

Like in Dfam-SODA, the annotation of TEs and simple repeats are represented with rectangle glyphs. The rectangles in RepeatMasker-SODA, however, are different in the following three ways:

1. The outline, rather than the entire glyph, is colored according to the TE family color scheme.
2. The interior of the rectangles are shaded in grayscale to indicate the inferred biological age (determined by the quality of the sequence alignment that defines the annotation) of the annotated feature. Younger features appear darker, while older features appear lighter.
3. The interior of the rectangles are textured with a repeating chevron pattern to indicate which chromosome strand the feature was identified on. Annotations on the forward strand have a chevron pattern that points to the right, and the reverse strand patterns point left.

An example of this is shown in Figure 2.3.

2.2.1.2 RepeatMasker Annotation Fragments

Often, a RepeatMasker annotation represents a fragment of a known TE sequence. Two reasons that this can occur are:

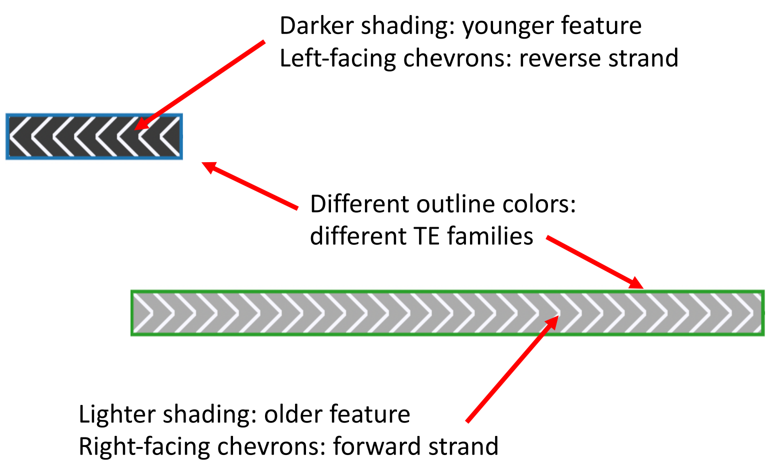

Darker shading: younger feature
Left-facing chevrons: reverse strand
Different outline colors: different TE families
Lighter shading: older feature
Right-facing chevrons: forward strand

Figure 2.3: An example of RepeatMasker-SODA rectangle glyphs.

1. TE features are often copied, and the copy is placed in another location in the genome. Sometimes, this process yields only a fragment of the original.

2. Large-scale deletions can occur, leaving behind fragments of originally full-length features.

When a RepeatMasker-SODA glyph represents a sequence fragment, dashed horizontal lines are rendered around it to project the portion(s) of the TE sequence that is missing from the fragment. An example of this is shown in Figure 2.4.

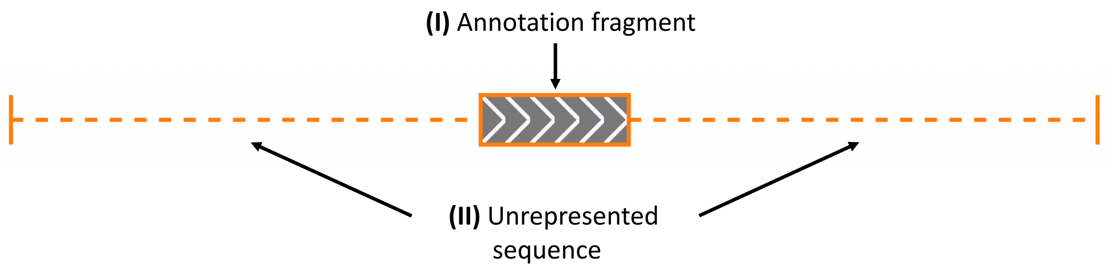


Figure 2.4: An example of an annotation fragment (I) surrounded by the projections of the portion of the known TE sequence that is missing from the fragment at (II).

2.2.1.3 Compact Rendering

In the RepeatMasker-SODA visualization, some sequence projections that flank rectangles are excessively long. To condense the visualization, they are rendered by default in a compact form. However, if a rectangle is clicked by the user, they will expand to their full length. If the rectangle is clicked a second time, the flank will collapse back to its compact length. For an example of this functionality, refer to Figure 2.5.

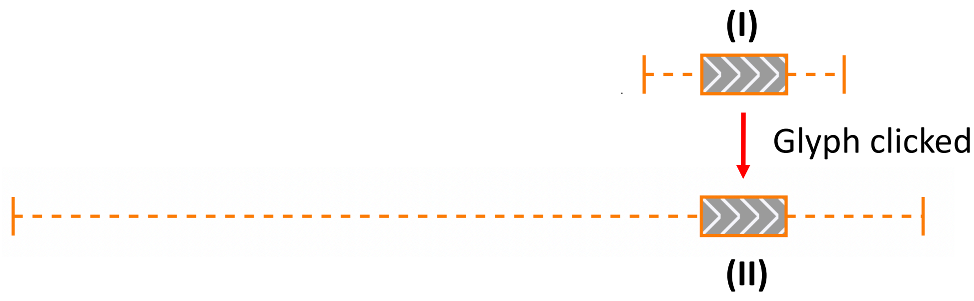


Figure 2.5: An example of the RepeatMasker glyph expansion. A compact glyph is shown at (I). After it is clicked by the user, it expands to the length shown at (II).

2.2.1.4 Annotation Fragments Produced by Insertion

The TE replication process described earlier sometimes results in the placement of a TE copy inside another TE that was already at the insertion site, fragmenting the TE that was already there. These situations add two layers of complexity to the RepeatMakser-SODA visualization:

1. When we infer that a feature has been fragmented by an insertion event, the fragments are joined with angled lines.

2. When we infer that a feature is itself an insertion, the glyph is always placed vertically as close as possible to the feature it fragmented. Inserts are never placed above the features they fragmented.

For an example of this, refer to Figure 2.6.

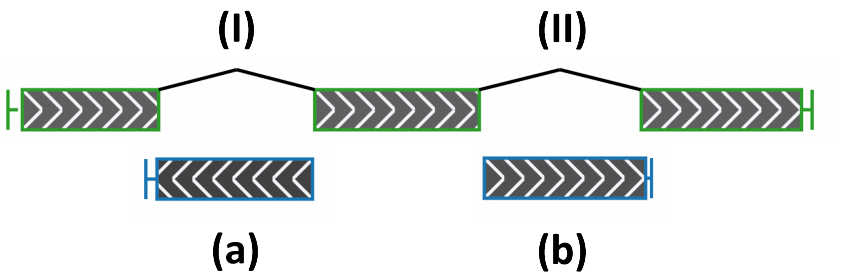


Figure 2.6: A depiction of two insertions into a TE feature. The blue features at (a) and (b) inserted into the green feature. At (I) and (II) are the angled lines to indicate that the three green fragments were originally joined.

2.2.1.5 Dynamic Annotation Labels

A dynamic label is placed immediately to the left of each annotation glyph. As the user zooms in and out, the labels automatically adjust the level of text detail they display depending on how much space is available. For an example of this, refer to Figure 2.7

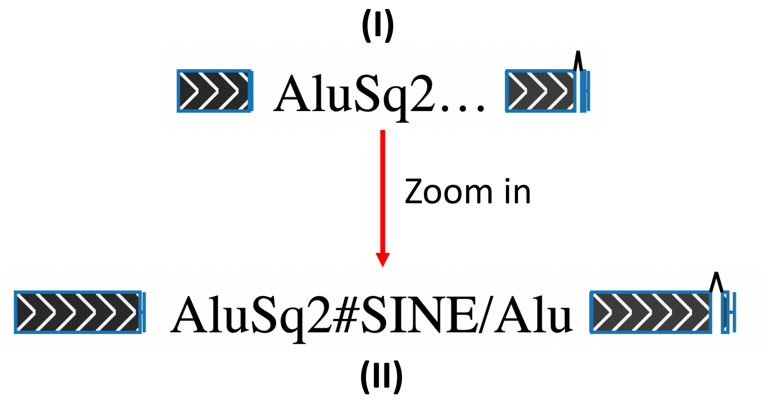


Figure 2.7: An example of the dynamic labels in RepeatMasker-SODA. At (I), the label is restricted by another glyph and displays shorter length text. After zooming in, there is more space and the full label is shown at (II).

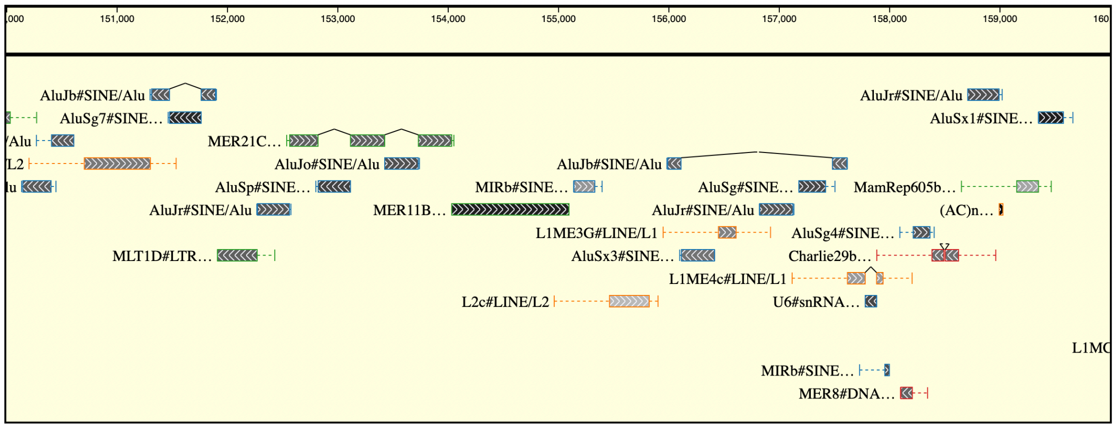


Figure 2.8: A screenshot of the RepeatMasker-SODA visualization.

2.3 PolyA Debugging Tool

The Wheeler lab has recently developed PolyA (manuscript in prep), a tool that adjudicates annotations by computing position specific confidence scores for competing alignments to the same region in a genome. For debugging purposes, we found that it was informative to view preexisting annotation (produced by the ProcessRepeats tool in the RepeatMasker [7] software), adjudicated annotation (produced by PolyA), and a heatmap of relevant confidence scores in a multi-track visualization. To aid in the development of PolyA and test the capabilities of SODA in a multi-track visualization context, we built a SODA-based PolyA output visualizer.

2.3.1 Description of PolyA-SODA

The development of PolyA is motivated specifically by the intention to adjudicate TE annotation, so the PolyA-SODA debugging tool depends on the previously described RepeatMasker-SODA to visualize both preexisting and adjudicated annotations. The PolyA-SODA visualization has three tracks:

1. A RepeatMasker-SODA track displaying current annotations from the UCSC RepeatMasker database

2. A RepeatMasker-SODA track displaying the PolyA adjudicated annotations for the same region

3. A track that displays a heatmap of the confidence scores for competing alignments in the same region

In addition, we added a vertical rule that spans the height of the visualization and follows the user's mouse to simplify the comparison of annotation across tracks. A tooltip that indicates the position to the nearest base pair is fixed to the rule.

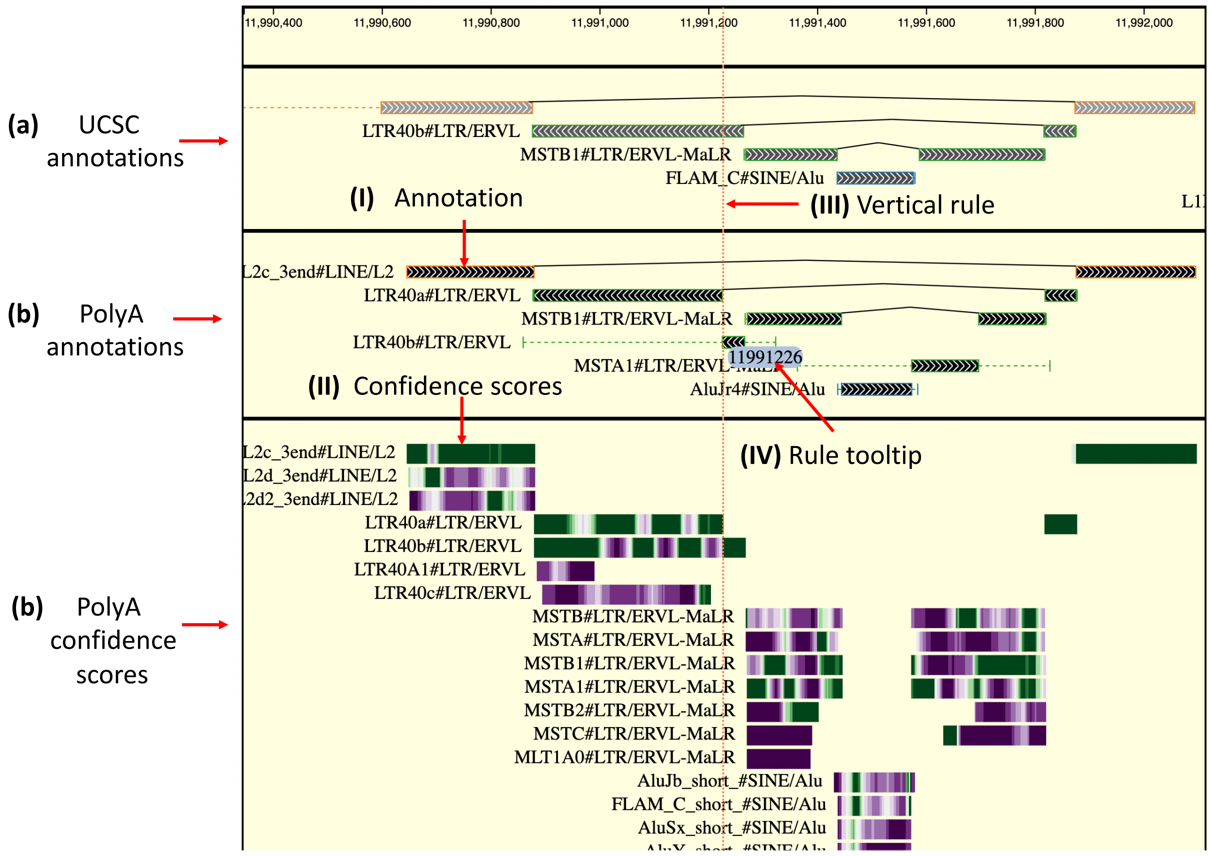


Figure 2.9: A screenshot of the PolyA-SODA debugging visualization. The RepeatMasker-SODA tracks are shown at (a) and (b), and the PolyA confidence score heatmap is shown at (c). An annotation is pointed out at (I), with the corresponding confidence scores shown at (II). The vertical rule is shown at (III), with the tooltip indicating its position in base pairs shown at (IV).

CHAPTER 3 IMPLEMENTATION

This chapter provides a description of SODA's implementation and design details. As mentioned previously, SODA mostly distinguishes itself from other accessible genomic visualization options by virtue of being a library that aims to support the visualization needs of the myriad of genomics-focused web services. SODA provides developers with a modular toolkit to ease the development of visualization applications that may not follow any established use case. SODA makes few assumptions about client data and the way in which it should be visualized, and it leverages that principle to allow a developer fine-grained control over the details of a visualization.

3.1 Design Principles

The main design principles of SODA are as follows:

1. It should be a modular library built on top of modern web technologies to ease development of visualization applications.
2. It should make few assumptions about client data and the way it should be visualized.
3. It should be capable of producing interactive and dynamic visualizations.
4. It should be lightweight, and its use should not require either the understanding or adoption of a complex system.
5. It should be easy to integrate in websites, and should support interaction with non-SODA website features.

6. It should be nonrestrictive, providing flexibility and fine-grained control for extensive customization.

7. It should provide intuitive implementation patterns that can be used as development blueprints and easily adapted to various use cases.

3.2 Technologies

SODA is built on top of modern web technologies, each of which is addressed in the following sections. Each section includes brief descriptions of a technology, the rationale for its use, and an explanation of how it used.

3.2.1 TypeScript

TypeScript is an open source programming language that extends JavaScript (the most widely used web programming language), by providing the addition of static type definitions. Because TypeScript code is compiled to JavaScript for execution, it can be interwoven seamlessly with JavaScript code. However, this also means that it cannot strictly enforce types at runtime. Instead, it provides a safety net during application development by helping to prevent type errors before any code is ever executed. TypeScript type definitions can be written to describe the behavior and structure of arbitrary JavaScript code. This allows the TypeScript compiler to perform type checking and inference on external JavaScript dependencies of a TypeScript project.

In general, type definitions in code can greatly improve both the readability and maintainability of a software product. Because of this, SODA is implemented entirely in TypeScript, and developers are strongly encouraged to use TypeScript when developing SODA-based applications. In particular, SODA makes extensive use of the concept of *type parameters* to propagate information of client-written SODA extensions throughout core SODA features. While this doesn't equip SODA with features that could not be produced with JavaScript code, it reduces the risk of misusing SODA features. Refer to code listing 1 for a simple example of how type parameters work, and to Chapter 5 for examples of actual SODA TypeScript code.

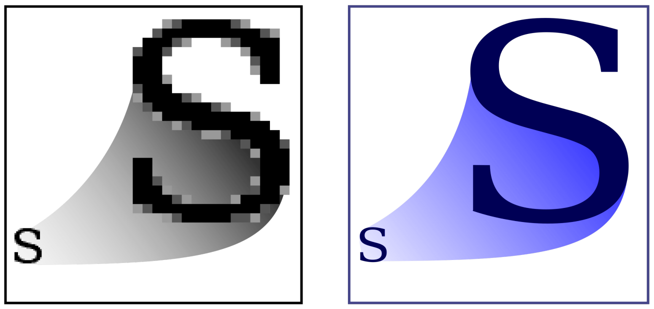
Code Listing 1: An example of how type parameters work in TypeScript

```typescript
class classA {
    propertyA: string;
}

class classB {
    propertyB: number;
}

// An interface with two type parameters, A and B.
interface InterfaceC<A extends classA, B extends classB> {
    // An anonymous callback function property that uses
    // the type arguments to type its own parameters
    callback: (a: A, b: B) => void;
}

// An instantiation of the interface with type parameters.
// Here, we provided classA and classB as the parameters, but,
// in principle, we could supply extensions of those classes
let c: InterfaceC<classA, classB> = {
    // To properly instantiate this interface, we have to
    // supply the callback function property
    callback: (a, b) => {
        // Normally, strict TypeScript would complain about parameters
        // a and b being ambiguously typed.
        // In this case, the TypeScript compiler is smart enough
        // to propagate the type arguments onto the parameters

        // This is valid because the compiler infers that a is of type ClassA
        console.log(a.propertyA);
        // This is valid because the compiler infers that b is of type ClassB
        console.log(b.propertyB);
    }
};
```

3.2.2 SVG – Scalable Vector Graphics

Scalable Vector Graphics (SVG) is a modern vector image format that is designed for use in web browsers. Vector images are defined by points in the Cartesian plane, which are then joined together by lines and curves when the image is actually rendered (see `https://en.wikipedia.org/wiki/Vector_graphics`. This allows SVG images to be programatically defined in Extensible Markup Language (XML) and rendered crisply in a browser at arbitrary zoom levels. Complex SVG based visualizations can be less performant than those based on other web rendering technologies

(Canvas, WebGL). However, they are generally easier to implement and test, and, for this reason, SODA visualizations are rendered entirely as SVG images.



(a) Standard raster image (JPEG, PNG, BMP, TIFF, etc.)

(b) SVG image

Figure 3.1: A scalability comparison between standard image formats and SVG images. (source: `https://en.wikipedia.org/wiki/Scalable_Vector_Graphics`)

3.2.3 D3 – Data-Driven Documents

D3 is a popular data visualization library implemented in JavaScript. At its core, D3 provides a set of tools to make it easy to manipulate and bind data to a browser's DOM elements, which control the visual aspects in a webpage (see `https://en.wikipedia.org/wiki/Document_Object_Model` for more information). D3 is powerful and flexible, but it has a reputation for having a steep learning curve. SODA's rendering module is an abstraction over D3 and it provides a straightforward API for rendering annotations as SVG images.

CHAPTER 4 FEATURES

This chapter presents a description of the core components of SODA and the ways in which they interact with one another. For more detailed information about the SODA API, refer to the library documentation (found at `https://sodaviz.readthedocs.io/`).

4.1 Annotation Objects

Annotation objects hold the data necessary to place the glyphs that represent an annotation within the coordinate space of a Chart (a Chart is an object in which Annotations are rendered, see Section 4.2). The base Annotation implementation stores only positional information and a unique identifier field. SODA assumes that horizontal coordinates are in reference to genomic positions, while vertical coordinates simply place an annotation into different rows/bins in a Chart. Most often, vertical placement of glyphs in a genomic visualization has no semantic meaning, and any intentional choice in vertical placement is motivated by an attempt to condense the visualization or preserve visual clarity. For specialized visualizations that represent auxiliary information (e.g. gene expression, sequence alignment quality) in an annotation, developers are intended to extend the Annotation class to include any data necessary to augment the rendered glyphs. The Annotation classes available in the SODA core are described in Table 4.1. For more detailed descriptions, refer to `https://sodaviz.readthedocs.io/en/latest/annotations.html`.

4.2 Chart Objects

When rendering, the glyphs that represent annotations are plotted inside of a Chart object. The core components of a Chart are the DOM element in which annotations are drawn, and

Object	Description
Annotation	The base Annotation class from which all other Annotation classes are derived
OrientedAnnotation	Adds an orientation field, which is most likely used to indicate chromosome strand
CompactAnnotation	Supports two sets of horizontal positioning fields so that the glyph can be dynamically expanded and compacted
TextAnnotation	Used for textual glyphs

Table 4.1: A description of Annotation objects in the SODA core.

a scale function that translates semantic coordinates into the relative coordinate space of the Chart. SODA provides an abstract base Chart class, along with some functional extended charts to facilitate the straightforward generation of common genomic visualization patterns. For heavily customized visualizations, developers are intended to extend the Chart class. The Chart classes available in the SODA core are described in table 4.2. For more detailed descriptions, refer to `https://sodaviz.readthedocs.io/en/latest/charts.html`.

Object	Description
ChartBase	The abstract base Chart class from which all other charts should be derived. It implements the functionality to create, embed in the browser, and manage the SVG viewport in which glyphs are rendered.
TrackChart	A Chart class that aims to mirror the basic functionality of a genome browser track. In particular, it comes configured to interface with the zooming and resizing SODA modules.
AxisChart	A Chart class that is used to render a horizontal Axis. It is intended to be added alongside and synchronized with another Chart object to display the genomic coordinates of a visualization. For examples of how this is done in practice, refer to chapter 2.

Table 4.2: A description of Chart objects in the SODA core.

4.3 Modules

4.3.1 Glyph Rendering

SODA provides a module that can be used to render glyphs to represent Annotation objects. The module can be used to generate rectangles, lines, arrows and text. Rendered glyphs can be customized with a configuration API in which glyph attributes can either be set directly, or by callback functions that implicitly receive references to the represented Annotation object and the target Chart. The glyphs available in the SODA rendering module are described in Table 4.3. For a more detailed description of the rendering API, refer to the documentation appendix.

Internally, SODA maintains a data structure that maps each Annotation to its representative glyph's DOM element. Whenever an Annotation is used to render a glyph, SODA updates the internal map automatically. The internal map is exposed to developers to make it easy to access the DOM elements.

4.3.2 Zooming

The zoom module allows developers to easily add interactive horizontal zooming and panning to a chart. When configuring a visualization with multiple charts, a ZoomController object automatically handles the synchronization of zoom level across all of its registered charts. Upon a zoom event, a zoom behavior function, which utilizes a scaling function and the coordinate data stored within a glyph's bound Annotation object, is applied to each glyph to transform and re-render the glyph. Any glyphs rendered with the primitive rendering API will automatically be assigned a default zoom behavior. Alternatively, the API allows the default zoom behaviors to be overwritten with arbitrary client-defined functions.

4.3.3 Resizing

The resize module allows developers to easily configure how a Chart will be re-rendered in response to resize event. When configuring a visualization with multiple charts, a ResizeController can be configured to handle the synchronized resizing of all of its registered charts. The TrackChart

Glyph	Description	Example
Rectangle	A simple rectangle. By default, it will cover the full width of an Annotation and the full height of a row in a Chart.	
Line	A simple line. There are convenience functions to render vertical and horizontal lines, but an arbitrary line may be rendered if the endpoint coordinates are provided.	
Chevron Rectangle	A rectangle with a chevron pattern background. By default, it will cover the full width of an Annotation and the full height of a row in a Chart.	
Chevron Line	A horizontal line with a chevron pattern affixed to it. By default, it will cover the full width of an Annotation and the full height of a row in a Chart.	
Text	A dynamic text label. It can be configured to display different length text depending on how much room it has in the coordidnate space of the Chart it's rendered in.	Some long text Some long... Some...

Table 4.3: A description of the glyphs available in the rendering module.

class provides a default resize behavior that maintains the original view range in terms of semantic coordinates, but a developer is free to implement custom behavior.

4.3.4 Layout Management

In general, the coordinate spaces of genomic annotations are one-dimensional. Often times, a dataset will contain collisions in this space, which would result in a visual overlap in the figure. A developer is free to manually or programmatically define their own layout, but SODA also provides a module to optimize the layout of rendered annotations to conserve vertical screen real estate while preventing any horizontal overlap.

4.3.4.1 Graph Coloring Based Layout

The layout module achieves this outcome by casting layout in terms of the classic computational problem of "graph coloring". The annotation overlap problem can be reduced to graph coloring in the following way:

1. Let each annotation be represented as a vertex in a notional graph G.
2. For each pair of vertices (v,w) in G, add an edge connecting v and w if the annotations associated with v and w are overlapping.

With such a graph in place, a proper coloring of the graph (in which no two vertices share the same color) indicates a non-overlapping layout of the annotations (all vertices sharing a color will see their corresponding annotations placed at the same vertical position in the visualization). Since edges are defined between overlapping annotation vertices, no two annotations that overlap will be placed at the same vertical position. A layout defined in this way will use a number of rows equal to to the number of colors.

Graph coloring is a well known NP-complete problem, and, as such, is computationally difficult to solve optimally. However, the graphs defined in this case are a special type of graph known as *interval graphs* (see `https://en.wikipedia.org/wiki/Interval_graph`), which are colorable in polynomial time. SODA's layout module uses a simple algorithm (see Algorithm 1) that is designed to color interval graphs.

Algorithm 1: The interval graph coloring algorithm

```
verts = vertices sorted by annotation start coordinate;
colors = 0;
while $length(verts) > 0$ do
    $v = verts.pop()$;
    $vColor = 0$;
    for $c = 0..colors$ do
        for each $w \in verts$ that has been colored with $c$ do
            if $v$ overlaps with $w$ then
                vColor++;
                break;
            end
        end
        break;
    end
    $colors = max(colors, vColor)$;
    $v.color = vColor$;
end
```

4.4 Plugins

4.4.1 Click and Hover Plugins

The Click and Hover plugins allow developers to bind any number of arbitrary callback functions to be executed whenever a glyph is clicked or hovered. The callback functions implicitly receive references both to the glyph's DOM element and its respective Annotation object. This way, the callback functions can easily be defined to modify the fields on the Annotation object and also to instigate some sort of visual change in the glyph.

4.4.2 Tooltip Plugin

The Tooltip plugin makes use of the Hover plugin to allow developers to cause a text tooltip to appear next to a glyph. The Tooltip configuration API allows developers to define callback functions to dynamically specify the style and text for a tooltip from fields on the glyph's underlying Annotation object.

4.4.3 Rule Plugin

The rule plugin allows developers to add a sliding vertical rule to any Chart. By default, the rule position is bound to the location of the mouse. Optionally, a tooltip displaying the semantic coordinate position of the rule can be attached to the rule. If there are multiple Charts with rules in a visualization, the rule positions can optionally be synchronized.

CHAPTER 5 EXAMPLE IMPLEMENTATIONS

In this chapter, I provide an in-depth exploration of the implementations of the examples shown in Chapter 2. This is intended to showcase SODA's feature set and present the design patterns that emerged during its development. Each of the applications builds upon the TrackChart class and the design pattern that it encourages, and collectively they provide a complete demonstration of all of SODA's features.

5.1 TrackChart Implementation

In this section, we will describe the implementation of the TrackChart class, which each of the example SODA applications is built on top of. The TrackChart class is largely designed to fit the basic needs of a typical genomic visualization track. It is configured to automatically support horizontal zooming, panning, and resizing, and it provides a blueprint for the canonical SODA glyph rendering pattern. The TrackChart can be instantiated directly and used to render simple visualizations, but this is a somewhat awkward approach that is probably best used for experimentation or rough prototyping. Instead, developers building a practical SODA application are encouraged to implement an extension of the TrackChart class.

5.1.1 Instantiation

The TrackChart class itself extends the abstract ChartBase class, and, as a result, inherits the basic functionality to generate and manage an SVG viewport in the browser. For a detailed description of the API for the ChartBase and TrackChart classes, refer to the documentation at `https://sodaviz.readthedocs.io/en/latest/charts.html`. A TrackChart is initialized with a TrackChartConfig object (see code listing 2), which houses all initial configuration options. The TrackChart constructor uses the *selector* property from the config to locate the DOM container for the visualization, and then creates the SVG viewport. Once a TrackChart has been initialized, glyphs may be rendered inside of its SVG viewport.

Code Listing 2: The TrackChartConfig interface definition.

```
export interface TrackChartConfig extends ChartConfig {
    // these properties are inherited from ChartConfig
    // and define the SVG viewport attributes
    selector: string;
    binHeight?: number;
    height?: number;
    width?: number;
    // these properties are unique to TrackChartConfig
    // and define the zooming and panning constraints
    scaleExtent?: [number, number];
    translateExtent?: (chart: TrackChart<any>) =>
        [[number, number], [number, number]];
}
```

5.1.2 Rendering

The TrackChart class inherits a render() method from the abstract ChartBase class which calls three other required methods:

- preRender() – A method that adjusts Chart properties to accommodate a new render. In the TrackChart, this updates the coordinate translation scale and adjusts the SVG viewport height to fit the glyphs that will be rendered.
- inRender() – An abstract method that should use the SODA rendering module to draw glyphs.
- postRender() – A method that runs any routines that need to be executed after the rendering takes place. In the TrackChart, this simply alerts the ZoomController and plugins if they are registered to the Chart.

The default render() implementation expects a TrackChartRenderParams object (see Code Listing 3) as an argument.

Code Listing 3: The TrackChartRenderParams object definition.

```
export interface TrackChartRenderParams extends ChartRenderParams {
    // these properties are inherited from ChartRenderParams
    // and they define the semantic width of the current render
    queryStart: number;
    queryEnd: number;
    // this property is unique to TrackChartRenderParams
    // and it defines the height of the visualization
    maxY?: number;
}
```

5.1.3 The Canonical Rendering Pattern

In the canonical rendering pattern, a developer implements an extension of the TrackChart class, in which a custom rendering routine is defined within the inRender() method. The extended class can retain the default render() function parameters, but it will more than likely be prudent for the developer to similarly extend the default TrackChartRenderParams class to include annotation and auxiliary data. An implementation of the inRender() method is necessary for the pattern, but the developer can optionally overwrite the preRender() and postRender() methods as well. Detailed examples of the canonical rendering pattern can be found in the following sections describing the implementations of each of the example SODA applications.

5.1.4 The Inverted Rendering Pattern

In the inverted rendering pattern (which is more suited for prototyping, experimentation, or possibly very simple practical applications), the developer defines a rendering routine external to the TrackChart class. This pattern is slightly awkward, but has advantage of not requiring the implementation of an extension of the TrackChart class. First, the developer instantiates a base TrackChart object. Then, the TrackChart is prepared by calling the render() method with arguments to appropriately adjust the dimensions of the SVG viewport and the coordinate translation scale. Finally, the developer can use the glyph rendering module to render Annotation objects inside the TrackChart. Refer to the following code listing for an example of this process.

Code Listing 4: An example of using the inverted rendering pattern with the TrackChart.

```
let n = 10;
let exampleWidth = 1000;

// first, we'll make some simple Annotation objects
let ann: soda.Annotation[] = [];
for (let i = 0; i < n; i++) {
    let id = i.toString();
    let annConf: soda.AnnotationConfig = {
        id: id,
        w: (exampleWidth/n),
        x: i * (exampleWidth/n),
        y: i,
        h: 0,
    };
    ann.push(new soda.Annotation(annConf));
}

// create an AxisChart and a TrackChart
let axis = new soda.AxisChart({selector: '#axis-chart'});
let chart = new soda.TrackChart({selector: '#track-chart'});

// define simple render parameters
let renderParams: soda.TrackChartRenderParams = {
    queryStart: 0,
    queryEnd: exampleWidth,
    maxY: n
};

// call render() on each Chart to prepare it for the glyphs
axis.render(renderParams);
chart.render(renderParams);

// we'll use a d3 scale to help us pick the rectangle colors
let colorScale = d3.scaleOrdinal(d3.schemeCategory10);

// define a simple rectangle config
let rectConf: soda.RectangleConfig<soda.Annotation, soda.TrackChart> = {
    selector: 'ann',
    // we'll use a callback to set the rectangle colors
    fillColor: (d: soda.Annotation) => colorScale(d.id)
};

// finally, we'll use the glyph module to draw the rectangles
soda.rectangleGlyph(chart, ann, rectConf);
```

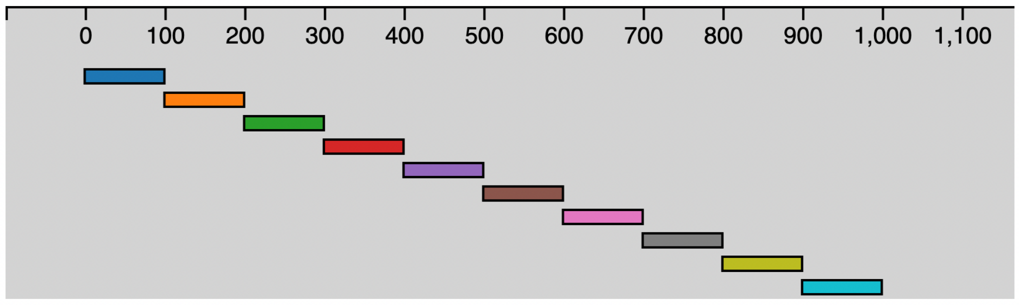


Figure 5.1: The resulting visualization using the inverted rendering pattern with the TrackChart.

5.2 Dfam-SODA Implementation

Dfam-SODA is made up of three components, each of which is an instantiation of a Dfam-TrackChart, an extended TrackChart class. The three components are encapsulated in a driver class, DfamAnnotationsGraphic, which instantiates a ZoomController, a ResizeController, an AxisChart, and a DfamTrackChart for each of the core components. Then, it translates the results of a query to the Dfam API (`https://www.dfam.org/help/api`) into SODA Annotation objects and feeds the Annotations into the correct DfamTrackChart's rendering routine.

Dfam-SODA is an open source application, and its full source code can be found at `https://github.com/TravisWheelerLab/dfam-soda`.

5.2.1 Dfam Annotation Records as SODA Annotation Objects

The Dfam API returns the results of a query to the TE annotation database as a JSON string. Since the Dfam Annotation records do not conform to a common annotation format, a simple parsing routine was written to translate the records into an extended SODA Annotation object. For details on the definition of the DfamAnnotation class, refer to Code Listing 5.

Code Listing 5: The custom Annotation object for Dfam records

```
export class DfamAnnotation extends Annotation implements OrientedAnnotation {
    // these fields are part of the base Annotation class
    // a unique identifier for this Annotation
    readonly id: string;
    // the semantic x coordinate of the annotation
    readonly x: number;
    // the semantic width of the annotation
    readonly w: number;
    // the y coordinate of the annotation
    y: number;

    // this is where the DfamAnnotation specific fields start
    // the family/type of TE this object represents
    readonly type: string;
    // the fine-grained classification of the TE
    readonly modelName: string;
    // the divergence score of the TE
    readonly score: number;
    // the orientation of the alignment
    readonly orientation: string;
    // a string identifier that provides us a means to find
    // the TE's row in the table below the visualization
    readonly rowId: string;
}
```

5.2.2 DfamTrackChart Overview

The three core components of the Dfam-SODA visualization are the forward strand annotations, the simple repeat annotations, and the reverse strand annotations. In this case, the vertical placement of the distinct groups of annotations has semantic meaning, but the vertical positioning of annotations relative to others in the same group does not. The forward and reverse strand charts have a variable height, defined by the number of rows necessary to render all of the annotations without visual overlap. The forward strand chart clusters annotations toward the bottom of the chart, moving them up as necessary to avoid overlap, while the reverse strand inverts that behavior. The simple repeat chart is fixed to a single row, as simple repeats do not to overlap.

To simplify the rendering logic, we implemented a single class capable of rendering each component in isolation. The DfamTrackChart class extends the base SODA TrackChart class, and inherits from it a considerable amount of functionality.

Code Listing 6: An overview of the definition of the DfamTrackChart class

```typescript
export class DfamTrackChart extends TrackChart<DfamChartRenderParams> {
    // d3 scale to map class to color
    colorScale: d3.ScaleOrdinal<string, string>;
    // if the chart is inverted, we invert the layout in the y direction
    inverted?: boolean;

    constructor(config: DfamChartConfig) {
        super(config);
        this.colorScale = d3.scaleOrdinal(REPEAT_COLORS)
            .domain(REPEAT_TYPES);
        this.inverted = config.inverted;
    }

    protected preRender(params: DfamChartRenderParams): void {}

    protected inRender(params: DfamChartRenderParams): void {}

    protected renderAnnotations(annotations: DfamAnnotation[]) {}

    protected setGlyphDynamics(annotations: DfamAnnotation[]): void {}

    protected bindHover(ann: DfamAnnotation): void {}

    protected bindTooltip(ann: DfamAnnotation): void {}

    protected bindClick(ann: DfamAnnotation): void {}
}
```

5.2.3 DfamChartRenderParams

The DfamChartRenderParmams extends the TrackChartRenderParams by adding one property that holds an array of DfamAnnotation objects.

Code Listing 7: An overview of the definition of the DfamChartRenderParams

```typescript
export interface DfamChartRenderParams extends TrackChartRenderParams {
    // these properties are inherited from TrackChartParams
    queryStart: number;
    queryEnd: number;
    maxY?: number;
    // this property holds the DfamAnnotation objects that will be rendered in the chart
    ann: DfamAnnotation[];
}
```

5.2.4 DfamTrackChart Rendering Routine

The DfamTrackChart class makes a small addition to the base preRender() implementation, implements inRender(), and uses the base postRender().

5.2.4.1 DfamTrackChart.preRender()

The base preRender() method assumes that the Annotation objects passed to it already have some sort of layout information, so it sets the height of the Chart based on the largest y–coordinate it finds among the Annotations. Since there is no external layout definition in the Dfam-SODA visualization, we use the layout module to assign a y–coordinate to each annotation and supply a maximum y value. Once this has been done, we can pass the adjusted render parameters off to the base preRender() method, which will finish preparing the Chart for rendering.

Code Listing 8: The DfamTrackChart preRender() routine

```
protected preRender(params: DfamChartRenderParams): void {
    params.maxY = Math.max(1, soda.greedyGraphLayout(params.ann));
    super.preRender(params);
}
```

5.2.4.2 DfamTrackChart.inRender()

The DfamTrackChart inRender() implementation calls two subroutines:

- renderAnnotations() – This renders the rectangles
- setGlyphDynamics() – This uses SODA plugins to bind dynamic functionality to each rendered glyph

Code Listing 9: The DfamTrackChart inRender() routine

```
protected inRender(params: DfamChartRenderParams): void {
    this.renderAnnotations(params.ann);
    this.setGlyphDynamics(params.ann);
}
```

5.2.4.3 DfamTrackChart.renderAnnotations()

The DfamTrackChart uses the rectangle glyph module along with a simple configuration to draw a rectangle for each Annotation. A callback function is provided that will dynamically set each rectangle to the appropriate color that indicates the type of TE it represents. Additionally, if the DfamTrackChart has been configured to be *inverted* (forward versus reverse strand), a y–coordinate callback is provided, which simply inverts the logic of the default y coordinate callback.

Code Listing 10: The DfamTrackChart renderAnnotations() routine

```
protected renderAnnotations(annotations: DfamAnnotation[]) {
    const conf : RectangleConfig<DfamAnnotation, DfamTrackChart> =  {
        selector: 'dfam-ann',
        strokeWidth: () => 4,
        strokeOpacity: () => 0,
        strokeColor: (a, c) => c.colorScale(a.type),
        fillColor: (a, c) => c.colorScale(a.type),
    };

    if (this.inverted) {
        // if the chart has been inverted,
        // we invert the y-coordinate calculation here
        conf.y = (a: DfamAnnotation): number =>
            (this.binCount - a.y - 1) * this.binHeight + 2
    }
    soda.rectangleGlyph(this, annotations, conf);
}
```

5.2.4.4 DfamTrackChart Dynamic Functionality

The DfamTrackChart loops over all of the rendered Annotation objects and uses a few simple subroutines that use SODA plugins to add dynamic functionality to each glyph.

Code Listing 11: The DfamTrackChart setGlyphDynamics() routine

```
protected setGlyphDynamics(annotations: DfamAnnotation[]): void {
    for (const ann of annotations) {
        this.bindHover(ann);
        this.bindClick(ann);
        this.bindTooltip(ann);
    }
}
```

5.2.4.5 DfamTrackChart Hover Behavior

A SODA hover configuration holds a reference to the Annotation to which the functionality will be bound and two callback functions. The mouseover() callback will be called when the representative glyph is hovered with the mouse, and the mouseout() callback will be run when the mouse is moved off of the glyph (assuming it was already being hovered). The callbacks can be defined such that they receive both a reference to the Annotation object and a selection of the glyph's DOM element.

The Dfam-SODA hover behavior changes the *stroke-opacity* property during hover events to achieve a highlighting effect on glyphs that are hovered. In this case (shown in Code Listing 12), only a reference to the DOM element was needed.

Code Listing 12: The DfamTrackChart bindHover() routine

```
protected bindHover(ann: DfamAnnotation): void {
    const hoverConf: soda.HoverConfig<DfamAnnotation> = {
        ann: ann,
        mouseout: (s, a) => {
            s.style('stroke-opacity', 0);
        },
        mouseover: (s, a) => {
            s.style('stroke-opacity', 0.5);
        },
    };
    soda.addHoverBehavior(hoverConf);
}
```

5.2.4.6 DfamTrackChart Click Behavior

A SODA click configuration holds a reference to the Annotation to which the functionality will be bound, along with a callback function. The click() callback will be called whenever the representative glyph is clicked with the mouse. The callback can be defined such that it receives both a reference to the Annotation object and a selection of the glyph's DOM element.

The Dfam-SODA click behavior uses the *DfamAnnotation.rowId* property along with D3 to get access to the annotation's corresponding row in the table below the visualization, highlights it, and scrolls the browser to its position. In this case (shown in Code Listing 13, only a reference to the Annotation was needed.

Code Listing 13: The DfamTrackChart bindClick() routine

```
protected bindClick(ann: DfamAnnotation): void {
    const clickConf: soda.ClickConfig<DfamAnnotation> = {
        ann: ann,
        click: (s, a) => {
            const rowSelection = d3.select<HTMLElement, any>(`#${a.rowId}`);
            const rowElement = rowSelection
                .node();
            if (rowElement == undefined) {
                throw(`Table row element on ${a.id} is null or undefined`);
            }
            else {
                // scroll the page to the table row
                rowElement
                    .scrollIntoView(false);
                // temporarily highlight the row
                rowSelection
                    .style('background-color', 'yellow');
                // fade back to the original color
                rowSelection
                    .transition()
                    .duration(2000)
                    .style('background-color', null);
            }
        }
    };
    soda.addClickBehavior(clickConf);
}
```

5.2.4.7 DfamTrackChart Tooltips

A SODA tooltip configuration holds a reference to the Annotation to which the functionality will be bound, a callback function to define the tooltip text, and several optional style parameters. The text callback function can be defined such that it receives a reference to the Annotation object.

The Dfam-SODA tooltips display the detailed name of the represented TE and its coordinates in the genome. In this case (shown in Code Listing 14), the callback uses a reference to the Annotation to extract the necessary data for the tooltip string.

Code Listing 14: The DfamTrackChart bindTooltip() routine

```
protected bindTooltip(ann: DfamAnnotation): void {
    const tooltipConf = {
        ann: ann,
        text: (a: DfamAnnotation) =>
            `${a.modelName} (${a.type}): ${a.x}-${a.x + a.w}`,
        opacity: () => 1.0,
    };
    soda.tooltip(this, tooltipConf);
}
```

5.2.5 DfamAnnotationsGraphic

The DfamAnnotationsGraphic class is a convenience driver class that initializes, configures, and renders the entire Dfam-SODA visualization. When instantiated, it creates DOM containers for each of its components in the target webpage, then in turn instantiates each of the SODA Charts that will be placed in the containers. Next, it instantiates a ZoomController and a ResizeController, and registers each component to both. Finally, if it was initially provided with Annotations to render, it will render the visualization.

Code Listing 15: An overview of the definition of the DfamAnnotationsGraphic class

```
export class DfamAnnotationsGraphic {
    // a class that unifies the soda components
    // that make up the Dfam-SODA visualization
    data?:                DfamSearchResults;
    zoomController:       ZoomController;
    resizeController:     ResizeController;
    axis:                 AxisChart;
    forwardChart:         DfamTrackChart;
    reverseChart:         DfamTrackChart;
    simpleChart:          DfamTrackChart;

    constructor(config: DfamAnnotationGraphicConfig) {
        this.createContainers(config.target);

        this.axis = new soda.AxisChart({selector: '#soda-axis'});

        this.forwardChart = new DfamTrackChart({selector: '#soda-fwd',
            scaleExtent: config.scaleExtent,
            translateExtent: config.translateExtent,
            binHeight: config.binHeight});
        this.forwardChart.inverted = true;

        this.reverseChart = new DfamTrackChart({selector: '#soda-rev',
            scaleExtent: config.scaleExtent,
            translateExtent: config.translateExtent,
            binHeight: config.binHeight});

        this.simpleChart = new DfamTrackChart({selector: '#soda-smp',
            scaleExtent: config.scaleExtent,
            translateExtent: config.translateExtent,
            binHeight: config.binHeight});

        this.zoomController = new ZoomController();
        this.resizeController = new ResizeController();

        this.zoomController.addComponents([this.axis,
            this.forwardChart,
            this.reverseChart,
            this.simpleChart]);

        this.resizeController.addComponents([this.axis,
            this.forwardChart,
            this.reverseChart,
            this.simpleChart]);

        if (config.data) {
            this.render(config.data);
        }
    }

    protected createContainers(target: any): void {}

    public render(data: DfamSearchResults): void {}

    public drawLegend(): void {}
}
```

Code Listing 16: An overview of the definition of the DfamAnnotationsGraphicConfig

```
export interface DfamAnnotationGraphicConfig {
    // a css selector to locate the
    // target DOM container for the graphic
    target?: string;
    // the result of the Dfam search API,
    //which will be used to render the visualization
    data?: DfamSearchResults;
    binHeight?: number;
    // controls the extent to which a user can zoom the graphic
    scaleExtent?: [number, number];
    // controls the extent to which a user can pan the graphic
    translateExtent?: (chart: TrackChart<any>) => [[number, number], [number, number]];
}
```

5.2.5.1 DfamAnnotationsGraphic.render()

The rendering routine initially triggers the ResizeController, as the Dfam site calls render() whenever the browser is resized. Next, it checks if the rendering data has changed since the previous render took place. If the data hasn't changed, it avoids unnecessary re-rendering by doing nothing. If the data has changed, it parses the new data into DfamAnnotation objects and passes the correct subset to each of its components.

Code Listing 17: The DfamAnnotationsGraphic rendering routine.

```
public render(data: DfamSearchResults): void {
    this.resizeController.trigger();
    if (data !== this.data) {
        this.data = data;
        // parse the search results for Annotation objects,
        // which are returned grouped by their target chart
        let parsedResults = parseDfamSearchResults(data);

        this.axis.render({
            queryStart: parsedResults.queryStart,
            queryEnd: parsedResults.queryEnd
        });

        this.forwardChart.render({
            ann: parsedResults.forward,
            queryStart: parsedResults.queryStart,
            queryEnd: parsedResults.queryEnd
        });

        this.reverseChart.render({
            ann: parsedResults.reverse,
            queryStart: parsedResults.queryStart,
            queryEnd: parsedResults.queryEnd
        });

        this.simpleChart.render({
            ann: parsedResults.simple,
            queryStart: parsedResults.queryStart,
            queryEnd: parsedResults.queryEnd
        });
        this.drawLegend();
    }
}
```

5.2.5.2 DfamAnnotationsGraphic usage

The usage of the DfamAnnotationsGraphic is simple: after a Dfam website user submits a search query, the page stores the resulting API response JSON string (see `https://en.wikipedia.org/wiki/JSON` for more information), and passes it to the graphic's rendering routine.

Code Listing 18: An example of how the DfamAnnotationsGraphic is used in the Dfam webcode

```
redraw() {
    if (!this.graphic) {
        // create the chart from scratch only the first time
        const el = this.graph.nativeElement;
        el.innerHTML = '';

        const graphicConf: DfamAnnotationGraphicConfig = {
            target: el,
            data: this.data,
            scaleExtent: [1, 10],
            translateExtent: (chart) => [[0, 0], [chart.width, chart.height]],
        };
        this.graphic = new DfamAnnotationsGraphic(graphicConf);
    }
    this.graphic.render(this.data);
}
```

5.3 RepeatMasker-SODA Implementation

RepeatMasker-SODA is implemented as an extension of the TrackChart class. The layout of the visualization has a nuanced, semantic meaning that cannot be properly defined using the default SODA layout module. With that being the case, the layout is determined by the SODA client prior to parsing RepeatMasker records into SODA objects. The external layout engine is a complex, rule-based system that is outside the scope of the work presented here.

RepeatMasker-SODA is an open source application, and its full source code can be found at `https://github.com/TravisWheelerLab/rmsk-soda`.

5.3.1 RepeatMasker Annotation Blocks

As shown in Chapter 2, one RepeatMasker annotation is represented by a combination of rectangles, lines, and a dynamic label. Each record in the RepeatMasker database contains one or more annotation *blocks* that collectively describe the annotation of one TE instance. There are three main types of blocks, two of which are further sub-typed. For reference, Table 5.1 describes the blocks and their sub-types, and Figure 5.2 shows how each block type is visualized. The primary block types are:

1. Aligned blocks, which represent alignments between the chromosome and the TE sequence.

2. Unaligned blocks, which represent portions of the TE sequence missing from the annotation.

3. Joining blocks, which represent positional relationships between the other blocks.

Unaligned and joining blocks are sub-typed by positional context, and each sub-type is rendered differently depending on that context. Unaligned blocks can be left-flanking, right-flanking, or internal to the annotation (henceforth referred to as ‘inner unaligned’). Joining blocks are always rendered in pairs, and we distinguish between the left and right members of each pair. In addition to the blocks explicitly represented in the RepeatMasker data, we consider the annotation label as a block in the genomic coordinate space.

Block type	Description	Representative Glyph
Aligned	Represents alignments between the chromosome and the TE sequence	Rectangle with a chevron pattern indicating the chromosome strand
Left unaligned	Represents missing portions of the TE sequence on the left flank of the annotation	Dashed horizontal line with a vertical endpoint line on the left
Right unaligned	Represents missing portions of the TE sequence on the right flank of the annotation	Dashed horizontal line with a vertical endpoint line on the left
Inner unaligned	Represents missing portions of the TE sequence in between aligned blocks	Dashed horizontal line with a vertical endpoint line on the right
Left joining	Represents positional relationships between aligned blocks, unaligned blocks, and repeat models	Solid angled line pointing upwards to the right
Right joining	Represents positional relationships between aligned blocks, unaligned blocks, and repeat models	Solid angled line pointing upwards to the left
Label	Labels the annotation	Text

Table 5.1: A description of the annotation block types in RepeatMasker-SODA.

5.3.2 RepeatMasker Records as SODA Annotation Objects

The blocks in each record are parsed into a collection of RMSKAnnotation objects (refer to Code Listing 19). In addition to the blocks that are explicitly represented in a record, we generate an extra RMSKAnnotation object that is used to render the label next to each annotation.

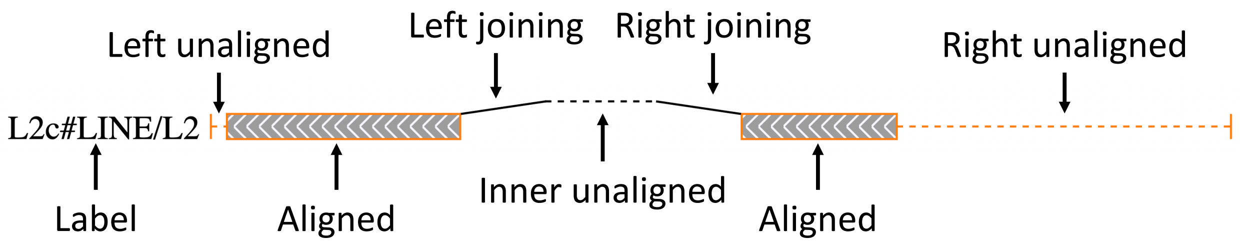


Figure 5.2: An example of a RepeatMasker-SODA glyph that represents unaligned, aligned, and joining blocks.

Code Listing 19: An overview of the RMSKAnnotation class implementation.

```typescript
// the custom Annotation object for joined RepeatMasker records
export class RMSKAnnotation extends Annotation
    implements TextAnnotation, CompactAnnotation, OrientedAnnotation {
    // these properties are inherited from the base Annotation class
    readonly id: string;
    readonly x: number;
    y: number;
    readonly w: number;
    readonly h: number;

    // this is from OrientedAnnotation
    readonly orientation: string;

    // these properties are from TextAnnotation
    text: string[];
    drawThresholds: number[];

    // these properties are from CompactAnnotation
    compacted: boolean;
    compactX: number;
    compactW: number;

    // these properties are unique to the RMSKAnnotation
    // the type of annotations we are drawing
    readonly type: string;
    // TE classification names
    readonly className: string;
    readonly familyName: string;
    readonly subfamilyName: string;
    // the divergence score
    readonly score: number;

    // these methods are from
    public getX(): number {}
    public getW(): number {}
}
```

5.3.3 RMSKTrackChart Overview

The RMSKTrackChart is an extension of the TrackChart class with most of the extended implementation existing as the rendering routine.

Code Listing 20: The custom TrackChart extension class

```
export class RMSKTrackChart extends TrackChart<RMSKTrackChartRenderParams> {
    // d3 scale to map divergence score to a color
    divergenceColorScale: d3.ScaleSequential<string>;
    // d3 scale to map class to color outline
    classColorScale: d3.ScaleOrdinal<string, string>;

    constructor(config: ChartConfig) {
        super(config);
        this.divergenceColorScale = d3.scaleSequential(d3.interpolateGreys)
            .domain(SCORE_RANGE);
        this.classColorScale = d3.scaleOrdinal(d3.schemeCategory10)
            .domain(REPEAT_CLASSES);
    }

    protected inRender(params: RMSKTrackChartRenderParams): void {}

    protected renderAligned(aligned: RMSKAnnotation[]) {}

    protected renderLeftUnaligned(leftUnaligned: RMSKAnnotation[]): void {}

    protected renderRightUnaligned(rightUnaligned: RMSKAnnotation[]): void {}

    protected renderInnerUnaligned(innerUnaligned: RMSKAnnotation[]): void {}

    protected renderLeftJoining(leftJoining: RMSKAnnotation[]): void {}

    protected renderRightJoining(rightJoining: RMSKAnnotation[]): void {}

    protected renderLabels(labels: RMSKAnnotation[]): void {}

    protected bindClick(aligned: RMSKAnnotation[]): void {}

    protected bindTooltips(aligned: RMSKAnnotation[]): void {}
}
```

5.3.4 RMSKTrackChartRenderParams

Once the records have been parsed into RMSKAnnotation objects, they are filtered and grouped by block type. The RMSKTrackChartRenderParams interface (see Code Listing 21) contains a property for each the list of RMSKAnnotations.

Code Listing 21: The implementation of the RMSKTrackChartRenderParams.

```
export interface RMSKTrackChartRenderParams extends TrackChartRenderParams {
    // these properties are inherited from TrackChartParams
    queryStart: number;
    queryEnd: number;
    maxY?: number;
    // these are Annotation objects that represent different block types
    aligned:          RMSKAnnotation[];
    leftUnaligned:    RMSKAnnotation[];
    rightUnaligned:   RMSKAnnotation[];
    innerUnaligned:   RMSKAnnotation[];
    leftJoining:      RMSKAnnotation[];
    rightJoining:     RMSKAnnotation[];
    labels:           RMSKAnnotation[];
}
```

5.3.5 RMSKTrackChart Rendering Routine

The RMSKTrackChart uses the default preRender() and postRender(), and it implements inRender().

5.3.5.1 RMSKTrackChart.inRender()

The RMSKTrackChart inRender() implementation calls several subroutines:

- renderAligned() – This renders the aligned blocks
- renderLeftUnaligned() – This renders the left flanking unaligned blocks
- renderRightUnaligned() – This renders the right flanking unaligned blocks
- renderInnerUnaligned() – This renders the inner unaligned blocks

- renderLeftJoining() – This renders the left joining blocks
- renderRightJoining() – This renders the right joining blocks
- renderLabels() – This renders the dynamic labels next to each glyph
- setGlyphDynamics() – This uses SODA plugins to bind dynamic functionality to each rendered glyph

The implementations of each of these subroutines are similar to each other, so, for the sake of brevity, we have omitted several of their descriptions. For the full code, refer to the RepeatMasker-SODA Github repository (`https://github.com/TravisWheelerLab/rmsk-soda`).

Code Listing 22: The RMSKTrackChart inRender() implementation

```
protected inRender(params: RMSKTrackChartRenderParams): void {
    this.renderAligned(params.aligned);
    this.renderLeftUnaligned(params.leftUnaligned);
    this.renderRightUnaligned(params.rightUnaligned);
    this.renderInnerUnaligned(params.innerUnaligned);
    this.renderLeftJoining(params.leftJoining);
    this.renderRightJoining(params.rightJoining);
    this.renderLabels(params.labels);
    this.bindTooltips(params.aligned);
    this.bindClick(params.aligned);
}
```

5.3.5.2 RMSKTrackChart.renderAligned()

Here, we make use of the chevron rectangle glyph module along with a substantial configuration to draw a chevron rectangle for each aligned block. Callback functions in the configuration are used to determine the outline and fill colors of each rectangle, similarly to the Dfam-SODA implementation. By default, SODA glyphs fill the height of a row in the Chart they are rendered in. To override this behavior we supply callback functions to set the y–coordinate and the height of the rectangles so that they are rendered at half of the height of a row.

Code Listing 23: The RMSKTrackChart renderAligned() implementation

```
protected renderAligned(aligned: RMSKAnnotation[]) {
    const rectConf: soda.ChevronRectangleConfig<RMSKAnnotation, RMSKTrackChart> = {
        selector: 'aligned',
        strokeColor: (a, c) => c.classColorScale(a.className),
        fillColor: (a, c) => c.divergenceColorScale(7000 - a.score),
        y: (a, c) => a.y * c.binHeight + c.binHeight/2,
        h: (a, c) => c.binHeight/2,
        chevronSpacing: () => 10,
    };
    soda.chevronRectangleGlyph(this, aligned, rectConf);
}
```

5.3.5.3 RMSKTrackChart.renderLeftUnaligned()

Here, we make use of both the horizontal and vertical line glyph modules to draw the left-flanking unaligned blocks. First, the dashed horizontal lines are rendered with callback functions to position them at center-height relative to the aligned rectangles. Next, the vertical line endpoints are rendered with callback functions to position them at the left end of the block region and so that they are rendered at half the height of a row.

Code Listing 24: The RMSKTrackChart renderLeftUnaligned() implementation

```
protected renderLeftUnaligned(leftUnaligned: RMSKAnnotation[]): void {
    const horizontalConf: soda.HorizontalLineConfig<RMSKAnnotation, RMSKTrackChart> = {
        selector: 'left-unaligned',
        strokeDashArray: () => "3, 3",
        y: (a, c) => (a.y + 1) * c.binHeight - c.binHeight/4,
        strokeColor: (a, c) => c.classColorScale(a.className),
    };
    soda.horizontalLine(this, leftUnaligned, horizontalConf);

    const verticalConf: soda.VerticalLineConfig<RMSKAnnotation, RMSKTrackChart> = {
        selector: 'left-endpoint',
        x: (a) => a.getX(),
        y1: (a, c) => (a.y + 1) * c.binHeight,
        y2: (a, c) => (a.y + 1) * c.binHeight - c.binHeight/2,
        strokeColor: (a, c) => c.classColorScale(a.className),
    };
    soda.verticalLine(this, leftUnaligned, verticalConf);
}
```

5.3.5.4 RMSKTrackChart.renderLeftJoining()

Here, we make use of the generic line glyph module to draw the upward-angled left joining line blocks. In this case, we supply callback functions that define the start and end coordinates of each line.

Code Listing 25: The RMSKTrackChart renderAligned() implementation

```
protected renderLeftJoining(leftJoining: RMSKAnnotation[]): void {
    const lineConf: soda.LineConfig<RMSKAnnotation, RMSKTrackChart> = {
    selector: 'left-join',
        x1: (a) => a.getX(),
        x2: (a) => a.getX() + a.getW(),
        y1: (a, c) => a.y * c.binHeight + c.binHeight/2,
        y2: (a, c) => a.y * c.binHeight + c.binHeight/4,
    };
    soda.lineGlyph(this, leftJoining, lineConf);
}
```

5.3.5.5 RMSKTrackChart.renderLabels()

Here, we make use of the text glyph module to draw the dynamic labels. A callback function is used to place the text at the right end of the space allotted for the label. Another callback function is provided to dynamically generate three levels of text detail from the Annotation objects.

Code Listing 26: The RMSKTrackChart renderLabels() implementation

```
protected renderLabels(labels: RMSKAnnotation[]): void {
    const textConf: soda.TextConfig<RMSKAnnotation, RMSKTrackChart> = {
        selector: 'label',
        x: (a) => a.getX() + a.getW(),
        textPad: 5,
        text: (a) => [`${a.subfamilyName}#${a.className}/${a.familyName}`,
            `${a.subfamilyName}#${a.className}...`,
            `${a.subfamilyName}...`]
    };
    soda.textGlyph(this, labels, textConf);
}
```

5.3.6 RMSKTrackChart Dynamic Functionality

The RMSKTrackChart loops over all of the rendered Annotation objects and uses a few simple subroutines that use SODA plugins to add dynamic functionality to each glyph.

Code Listing 27: The RMSKTrackChart setGlyphDynamics() implementation

```
protected setGlyphDynamics(aligned: RMSKAnnotation[]) {
    for (const ann of aligned) {
        this.bindClick(ann);
        this.bindTooltip(ann);
    }
}
```

5.3.6.1 RMSKTrackChart.bindClick()

Here, a callback function is defined that provides the unaligned flank expansion feature. The click callback first finds each RMSKAnnotation object associated with the one represented by the glyph that was clicked. Then, if any of the objects found are compactable (see Figure 2.5), the *compacted* flag is toggled. Finally, the ZoomController is called to re-render the visualization over a duration of 1000 milliseconds, which results in an animation effect during the expansion or collapsing of the relevant flanks.

Code Listing 28: The RMSKTrackChart bindClick() implementation

```
protected bindClick(ann: RMSKAnnotation): void {
    const clickConf: soda.ClickConfig<RMSKAnnotation> = {
    ann: ann,
    click: (s, a) => {
        let prefix = a.id.split('-')[0];
        for (const key of soda.getAllIds()) {
            if (key.startsWith(prefix)) {
                let ann = soda.getAnnotationById(key);
                if (isCompactAnn(ann)) {
                    ann.compacted = !ann.compacted;
                }
            }
        }
        this.getZoomController().zoomedRenderDuration(1000);
    }
};
soda.addClickBehavior(clickConf);
}
```

5.3.6.2 RMSKTrackChart.bindTooltip()

Here, we define a tooltip configuration with a callback that provides a string containing the detailed name of the hovered annotation.

Code Listing 29: The RMSKTrackChart bindTooltip() implementation

```
protected bindTooltip(ann: RMSKAnnotation): void {
    const conf: soda.TooltipConfig<RMSKAnnotation, RMSKTrackChart> = {
        ann: ann,
        text: (a) => `${a.subfamilyName}#${a.className}/${a.familyName}`,
    };
    soda.tooltip(this, conf);
}
```

5.3.7 RMSKTrackChart Usage

Code Listing 30 provides an example of how RepeatMasker-SODA is used. First, the controller objects are instantiated. Next, the Chart objects are instantiated and added to the controllers. Finally, data is provided to the render() function, which in turn renders the AxisChart and RMSKTrackChart.

Code Listing 30: Example usage of the RMSKTrackChart

```
let zoomController = new ZoomController();
let resizeController = new ResizeController();

window.onresize = () => resizeController.trigger();

const axis = new AxisChart({selector: '#axis'});
const rmskChart = new RMSKTrackChart({selector: '#rmsk-chart', binHeight: 24});

zoomController.addComponent(axis);
zoomController.addComponent(rmskChart);

resizeController.addComponent(axis);
resizeController.addComponent(rmskChart);

// this function uses its query parameter arguments to query the
// UCSC API and render the results in the RMSKTrackChart
function render(chr: string, queryStart: number, queryEnd: number): void {
    // we make an asynchronous query to the UCSC api and parse the
    // results into RMSKTrackChartParams, then render the Charts
    getRMSKRenderParamsFromQuery(chr, queryStart, queryEnd)
        .then((params: RMSKTrackChartRenderParams) => {
            rmskChart.render(params);
            // the RMSKTrackChartParams are also valid AxisRenderParams
            axis.render(params);
        });
}
```

5.4 PolyA-SODA Implementation

PolyA-SODA is made up of three components in a multi-track visualization format. Two of the components are instantiations of the RMSKTrackChart class–one visualizes the existing annotation of TE's from the UCSC genome browser, and the other visualizes the PolyA adjudicated annotations. The third component visualizes the PolyA confidence scores for all competing alignments in the region as a heatmap.

PolyA-SODA is an open source application, and its full source code can be found at `https://github.com/TravisWheelerLab/polya-soda`.

5.4.1 PolyA Confidence Scores as SODA Annotation Objects

The PolyA debugging output provides a sparse two-dimensional array of position specific confidence scores for each competing alignment in the adjudicated region. Each row represents the confidence scores for the alignments of a particular TE family. The confidence scores are rounded to the nearest tenth and encoded with a run length encoding (see Figure 5.3). Null values in a row, which represent a position for which an alignment to a family was not present, are discarded. Each run in a row is represented as a PolyAHeatmapCell object, an extended Annotation object that is used to render the heatmap. For each row, a corresponding PolyAHeatmapLabel object, an extended TextAnnotation object, is created for the purpose of rendering the TE family name next to each row.

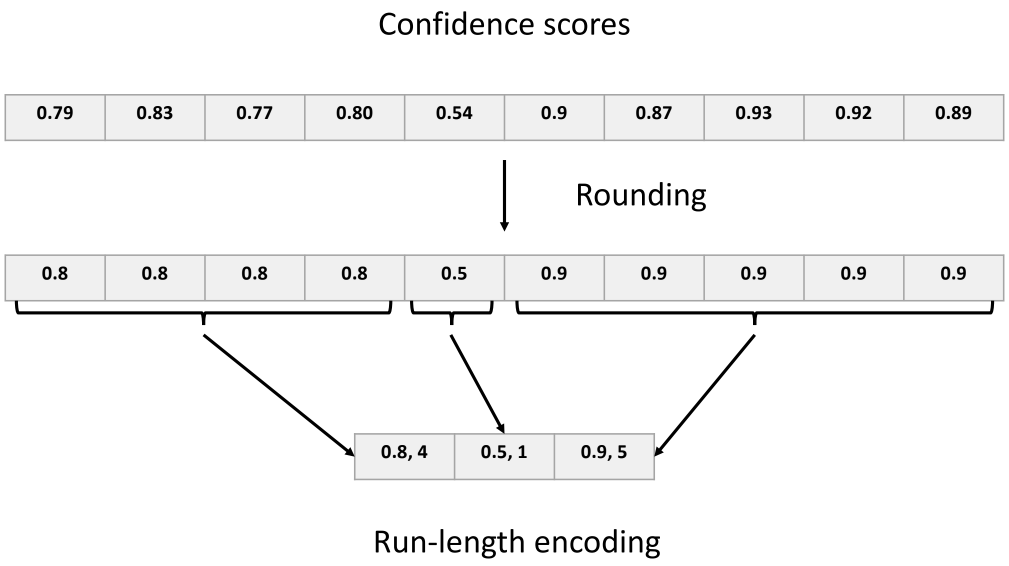


Figure 5.3: An example of how PolyA confidence scores are rounded and encoded with a run-length encoding.

Code Listing 31: The definition of the PolyAHeatmapCell class.

```
export class PolyAHeatmapCell extends Annotation {
    // the confidence score value that we will
    // use to color the heatmap cell
    public value: number;
}
```

Code Listing 32: The definition of the PolyAHeatmapLabel class

```
export class PolyAHeatmapLabel extends Annotation implements TextAnnotation {
    // the name of the TE family that the label is for
    name: string;
    // these properties are required by TextAnnotation
    text: string[] = [];
    drawThresholds: number[] = [];
}
```

5.4.2 PolyAHeatmapChart Overview

The PolyAHeatmapChart is a simple extension of the TrackChart class.

Code Listing 33: The definition of the PolyaHeatmapChart class

```
export class PolyAHeatmapChart extends TrackChart<PolyAHeatmapRenderParams> {
    colorScale: d3.ScaleSequential<string>;

    constructor(config: ChartConfig) {
        super(config);
        this.colorScale = d3.scaleSequential(d3.interpolatePRGn)
            .domain([0, 1]);
    }

    public inRender(params: PolyAHeatmapRenderParams) {}

    public renderHeatmap(rows: PolyAHeatmapCell[]): void {}

    public renderLabels(rowLabels: PolyAHeatmapLabel[]): void {}
}
```

5.4.3 PolyAHeatmapRenderParams

The PolyAHeatmapRenderParams extends the TrackChartRenderParams by adding properties that hold arrays of PolyaHeatmapCells and PolyAHeatmapLabels.

Code Listing 34: The definition of the PolyaHeatmapChart class

```
export interface PolyAHeatmapRenderParams extends TrackChartRenderParams {
    // these properties are inherited from TrackChartParams
    queryStart: number;
    queryEnd: number;
    maxY?: number;
    // these properties hold the Annotation objects for the heatmap labels and rectangles
    rows: PolyAHeatmapCell[];
    rowLabels: PolyAHeatmapLabel[];
}
```

5.4.4 PolyAHeatmapChart Rendering Routine

The PolyAHeatmapChart uses the default preRender() and postRender() and implements inRender().

5.4.5 PolyAHeatmapChart.inRender()

The PolyAHeatmapChart inRender() calls two subroutines:

- renderHeatmap()—This renders the rectangles that visualize the heatmap
- renderLabels()—This renders the TE family name labels next to each row in the heatmap

Code Listing 35: The PolyAHeatmapChart inRender() implementation

```
public inRender(params: PolyAHeatmapRenderParams) {
    this.renderHeatmap(params.rows);
    this.renderLabels(params.rowLabels);
}
```

5.4.6 PolyAHeatmapChart.renderHeatmap()

The PolyAHeatmapChart uses the rectangle glyph module to draw a rectangle for each run of similar confidence scores.

Code Listing 36: The PolyAHeatmapChart renderHeatmap() implementation

```
public renderHeatmap(rows: PolyAHeatmapCell[]): void {
    const rectConf: soda.RectangleConfig < PolyAHeatmapCell, PolyAHeatmapChart > = {
        selector: 'heatmap-cell',
        strokeColor: (a) => this.colorScale(a.value),
        fillColor: (a) => this.colorScale(a.value),
    };
    soda.rectangleGlyph(this, rows, rectConf);
}
```

5.4.7 PolyAHeatmapChart.renderLabels()

The PolyAHeatmapChart uses the text glyph module to draw a label next to each row in the heatmap.

Code Listing 37: The PolyAHeatmapChart renderLabels() implementation

```
public renderLabels(rowLabels: PolyAHeatmapLabel[]): void {
    const textConf: soda.TextConfig<PolyAHeatmapLabel, PolyAHeatmapChart> = {
        selector: 'label',
        textPad: 10,
        x: (a) => a.getX() + a.getW(),
        y: (a) => (a.y + 0.5) * this.binHeight,
        text: (a) => [a.name]
    };
    soda.textGlyph(this, rowLabels, textConf);
}
```

5.4.8 PolyA-SODA Usage

Code Listing 38 provides an example of how PolyA-SODA is used. First, the controller objects are instantiated. Next, the Chart objects are instantiated and added to the controllers. Finally, data is provided to the render() function, which in turn renders each chart.

Code Listing 38: An example of how the entire PolyA-SODA debugging tool is used

```
const zoomController = new ZoomController();
const resizeController = new ResizeController();
const ruleController = new RuleController();

window.onresize = () => resizeController.trigger();

const axis = new AxisChart({selector: '.axis'});
const ucscChart = new RMSKTrackChart({selector: '.ucsc-chart', binHeight: 20});
const polyaChart = new RMSKTrackChart({selector: '.polya-chart', binHeight: 20});
const heatmap = new PolyAHeatmapChart({selector: '.heatmap-chart', binHeight: 20});

const components = [axis, ucscChart, polyaChart, heatmap];

zoomController.addComponents(components);
resizeController.addComponents(components);
// by creating a RuleController and adding the components, the
// vertical rule that spans the visualization is automatically added
ruleController.addComponents(components);

// this function accepts the output of the PolyA debugging tool,
// two JSON strings which encode adjudicated annotation information
// and the relevant confidence scores
function render(polyaAnn: string, polyaConfidence: string): void {
    // the heatmapParams, which include the Annotation objects,
    // are first parsed from the polyA confidence scores
    const heatmapParams = parseConfidenceData(polyaConfidence);

    // the heatmap rendering paramters are inherently also valid
    // axis rendering paramters
    axis.render(heatmapParams);
    heatmap.render(heatmapParams);

    // the relevant chromosome from the polyA data is parsed out
    // so that we can make a query to the UCSC api
    const chr = parseInt(ann.split(/\s+/)[1].replace('chr', ''));

    // we query the range of the polyA data and use the RMSK-SODA
    // parser to get RMSKTrackChartRenderParams for the existing
    // annotations
    getRMSKRenderParamsFromQuery(chr, heatmapParams.queryStart,
                                 heatmapParams.queryEnd)
        .then((params: RMSKTrackChartRenderParams) => {
            ucscChart.render(params);
        });

    // we use a JSON of adjudicated PolyA annotations to get
    // RMSKTrackChartRenderParams for the PolyA annotations
    getRMSKRenderParamsFromJson(ann)
        .then((params: RMSKTrackChartRenderParam) => {
            polyaChart.render(params);
        });
}
```

CHAPTER 6 DISCUSSION

6.1 Future Work

While SODA currently has the functionality to render many of the glyph shapes that are commonly used across various genomic visualizations, I will continue to add new glyph shapes as I become aware of the need for them.

I would also like to implement more chart classes to cater to use cases that do not easily fit the mould of the TrackChart class. For example, I plan to add chart classes that are tailored towards creating line charts and bar charts.

Currently, SODA makes use of the popular JavaScript library D3 to render visualizations in SVG format. This makes the rendering process simple, but has a large impact on performance. I will experiment with using WebGL as a rendering backend. This may improve performance and make SODA visualizations more scalable.

SODA has been initially developed as a web framework that embeds visualizations in the web browser. While this is probably the way in which SODA will most often be used, I would like to develop Python bindings to make it easy to integrate SODA visualizations into command line applications.

6.2 Conclusion

SODA provides a simple, flexible, and modular framework that can be used to easily generate custom, dynamic visualization for arbitrary genomic data. It aims to fill the void between specialized visualization tools and large-scale genome browsers by providing developers with a toolkit that simplifies the process of creating novel genomic visualizations.

BIBLIOGRAPHY

[1] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler, "The human genome browser at ucsc," *Genome research*, Jun 2002. [Online]. Available: https://genome.cshlp.org/content/12/6/996.abstract/

[2] A. D. Yates *et al.*, "Ensembl 2020," *Nucleic Acids Research*, vol. 48, no. D1, pp. D682–D688, 11 2019. [Online]. Available: https://doi.org/10.1093/nar/gkz966

[3] R. Buels *et al.*, "JBrowse: a dynamic web platform for genome visualization and analysis," *Genome Biology*, vol. 17, no. 1, p. 66, Apr 2016. [Online]. Available: https://doi.org/10.1186/s13059-016-0924-1

[4] B. S. Der, E. Glassey, B. A. Bartley, C. Enghuus, D. B. Goodman, D. B. Gordon, C. A. Voigt, and T. E. Gorochowski, "Dnaplotlib: Programmable visualization of genetic designs and associated data," *ACS Synthetic Biology*, vol. 6, no. 7, pp. 1115–1119, 2017, pMID: 27744689. [Online]. Available: https://doi.org/10.1021/acssynbio.6b00252

[5] V. Zulkower and S. Rosser, "DNA Features Viewer: a sequence annotation formatting and plotting library for Python," *Bioinformatics*, vol. 36, no. 15, pp. 4350–4352, 07 2020. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa213

[6] R. Hubley, R. D. Finn, J. Clements, S. R. Eddy, T. A. Jones, W. Bao, A. F. Smit, and T. J. Wheeler, "The Dfam database of repetitive DNA families," *Nucleic Acids Research*, vol. 44, no. D1, pp. D81–D89, 11 2015. [Online]. Available: https://doi.org/10.1093/nar/gkv1272

[7] A. Smit, R. Hubley, and P. Green. (2013-2015) Repeatmasker open-4.0. [Online]. Available: http://www.repeatmasker.org

[8] A. Yousif, N. Drou, J. Rowe, M. Khalfan, and K. C. Gunsalus, “Nasqar: a web-based platform for high-throughput sequencing data analysis and visualization,” *BMC Bioinformatics*, vol. 21, no. 1, p. 267, Jun 2020. [Online]. Available: https://doi.org/10.1186/s12859-020-03577-4