

# Using the Recognition Strategy Library to Study the Behavior of a Math Recognition System

Kevin Talmadge, Richard Zanibbi

*Rochester Institute of Technology*

*Rochester, New York*

*1 Lomb Memorial Drive, Rochester*

---

## Abstract

We instrument a system for recognizing handwritten mathematical expressions using the Recognition Strategy Library, a data provenance tool for pattern recognition research, in order to study the behavior of the recognition system and demonstrate the utility of the library. In this case study, the library was used to perform comparison tests to quantitatively analyze several modifications to the recognition system. Several improvements were made to the library as a result of this study, including the ability to obtain the set of unique interpretations, determine the intersection and difference between two interpretation sets, and adding a method to dynamically record interpretations within a decision function.

---

## 1. Introduction

A common problem in developing and evaluating complex recognition systems is the difficulty of examining the intermediate results of a particular subsystem. Typically, the only information produced by these systems is the final result. Producing any other information requires the manual addition of print statements  
5 or other means of output deep within the logic of the code. This method is time consuming, error prone, and is rarely done in a standardized fashion. The recognition strategy library allows researchers to obtain and log intermediate decision outputs and related metrics that will allow them to better understand, debug, and improve their system in comparison to the traditional, manual means.

The recognition strategy library, or RSLib, is a data provenance tool in the form of a Python library for  
10 pattern recognition researchers. Data provenance tools are designed to help researchers and programmers record and analyze data through all transformations, analyses and interpretations. The purpose of the library is to automate and standardize the process of logging the decision history of a system for evaluation. This library provides researchers with the ability to easily save intermediate decisions made by recognition systems for evaluation. These saved interpretations may be reused, either by the same system, saving the time and  
15 effort of rerunning parts of an algorithm, or by a different system, providing a method of interchanging algorithms between systems. These facilities can reduce the effort required to create new prototypes, as well as provide an additional avenue of comparison between different algorithms.

Another benefit RSLib provides is the means to easily swap decision points in and out of a system. Pattern recognition systems are often very large and complicated, and may be slow, especially in the development stage. Often, a researcher is only focused on a particular subsystem within the recognition pipeline at a time. In such a case, it would be advantageous to simply reuse other parts from previous systems to create a full prototype, freeing up development time and resources for the primary area of interest.

RSLib allows for this reuse of previously generated interpretations, which means entire stages of the recognition process can be used from other systems with very little effort. After instrumentation, the resulting interpretations at each decision point defined in the system will be recorded. These interpretations can be saved as is via serialization. Saved interpretations can then be reloaded into a system, where the data may be used immediately instead of rerunning the algorithms that produced it.

The current library is the third iteration of the original concept[1][2][3]. The original language, as well as the more recent SML based version, were both standalone, domain specific languages. Much of the functionality of RSLib was present in one or both of the two languages that preceded it, but the fact that the current version is a Python library rather than a new language with a full compiler to maintain is a distinct advantage.

Despite the track record of RSLibs predecessors, its current form as a Python library has not been thoroughly tested, nor has its utility been demonstrated, in a significant, real world pattern recognition system. The primary objective of this research is to use RSLib in a modern handwritten mathematical expression recognition system in order to test the library's functionality, to demonstrate its utility in evaluating the recognition system and providing meaningful insight, and to determine what changes and additional functionality would help the tool to better serve its purpose.

As a result of this case study, several additional features were added to the library. First, a function was added to dynamically add sub-decision points within a decision function. This addition enables users to add additional granularity to the analysis of the instrumented system. Secondly, a function to obtain the unique set of interpretations was added. Functions to produce the intersection and difference of two sets of interpretations were also added. Finally, functions to save and load interpretation sets as-is were added, allowing easier analysis and reuse of interpretations.

With the addition of the dynamic sub-decision point function, the parsing algorithm was instrumented to record a detailed history of each decision made by the algorithm. Using this history, a process of creating a visualization of the parsing algorithm is demonstrated.

Two comparison studies were performed in this study. Both involve using RSLib in comparing a modified recognition system against a baseline system, in order to quantitatively analyze the benefit of the modifications.

In the first study, the goal was to determine the utility of a particular preprocessing step in the segmentation algorithm in which overlapping strokes are automatically merged into the same symbol. The second study was also intended to analyze the segmentation stage of the system. In this study, a new strategy of

pairing strokes for the segmentation classifier was tested against the original, time ordered pairing method.  
55 In both of these studies, the original system was determined to be superior.

## 2. Background

### 2.1. Language Basics

In the *Handbook of Programming Languages, Volume III*[4], a domain specific language is described as a tool that allows programmers to develop in a domain more quickly and effectively than with a general  
60 purpose language. In addition, an emphasis is placed on abstraction; making operations that are commonly used in the domain, yet complex or time consuming to express in a general purpose programming language, easy and intuitive to express.

The previous incarnations of RSL were standalone domain specific languages. With the change to a Python library, RSLib is more an *embedded* domain specific language. An embedded DSL is written as a  
65 library for a host language and, like a regular DSL, adds domain-specific primitives and functionality that allow for better abstraction in the domain. The library has much the same functionality as the previous versions (and perhaps more), but now it contains all of the power, expressiveness and familiarity of the Python language.

The primary focus of the original language and, in turn, all other incarnations since, is the interpretation  
70 type. An interpretation type represents a possible interpretation of the data at a particular point in the system as it relates to the problem domain. As an example, an interpretation of a math recognition system might include information about strokes in an expression, which strokes are grouped together as symbols, the label or classification of those symbols, and the relationship between the symbols in the expression. Each decision point will modify and/or add information to the current interpretation. Each decision point  
75 may produce multiple interpretations. For the recognition system used in this study, however, only one interpretation is produced for each decision.

The language also defines the strategy function, which defines a sequence of decision operations to be performed. A strategy takes the current set of interpretations and parameters as input, and produces a new set of interpretations as output. Pattern recognition systems contain segmentation, classification and parsing  
80 stages. These three stages determine where the objects of interest are (segmentation), identify object types (classification), and determine the structure of the objects (parsing). A simple strategy may be to run those stages as a sequence, while a more complicated strategy may involve those stages interacting based on the current set of interpretations, eg: reclassifying based on uncertainty in the parsing stage.

This design abstracts out the idea of interpretations and decision points, and makes the complex and  
85 time consuming task of adding logging functionality to a recognition system easier and more reliable. The researcher only needs to wrap pieces of the recognition system in RSLib's strategy and interpretation objects, at which point they can use the library to obtain interpretation and trace graph information, which can be saved, analyzed and reused as many times as necessary.

## 2.2. History of RSL

90 The predecessor of RSLib, the recognition strategy *language*, was initially proposed as a standalone, domain specific language (DSL) by Zanibbi et al.[1]

The recognition strategy language was later reimplemented as a different domain specific language, which compiled to Standard ML[2]. Several significant design changes and additions were made between the first version and this second version.

95 In the original version of RSL, decision functions were explicitly broken into three distinct categories: classification, segmentation and parsing. Similarly, interpretations contained only regions, relations and parameters. This second version, however, generalized these facilities.

Reporting functionality was also added. The original RSL saved history to text files, which were examined with other tools. The second version saw the addition of a secondary 'reporting' entry point, which has access 100 to the full history information and program trace. These reporting functions could be written by the user, and would be called at the end of the execution of a strategy.

Annotations were also added to RSL in this second iteration. An annotation allows the user to record additional relevant information at each decision point, unrelated to the interpretation.

These changes, and others, have survived in some form in the current iteration of RSL.

105 B. Holm's master's thesis addressed much the same problem as this project when the tool was still a language[3]. In his thesis, he demonstrated RSL's utility in evaluating one of the two primary computer vision algorithms in a sign language recognition system. The language was found to be valuable for the hand detection algorithm, which had several distinct steps, providing opportunity for meaningful comparison. It proved less useful, however, for the dynamic programming algorithm, due to its brute force nature and many 110 intermediate steps.

Ultimately, the fact that this second iteration of RSL required learning a new language in addition to having a working knowledge of SML greatly limited the tool's potential. It was decided that redesigning the language as a Python library may significantly benefit its ease of use and allow for a much wider user base.

115 While the overall design and abstraction mechanisms remain mostly unchanged in RSLib, the fact that it is fully contained in Python means that it contains all of the power and utility of Python, and only requires knowledge of the one language. In addition, Python is also more widely used than SML, and uses an imperative paradigm that is more familiar to many programmers than the functional paradigm used in SML. These factors may make RSLib more approachable and perhaps easier to learn for new users than the previous versions.

120 An additional benefit of moving to Python is that the implementation of the library is now significantly simpler. Instead of requiring an entire compiler, it is now several hundred lines of Python code. With this change, there are now fewer opportunities for bugs, and maintenance and enhancement of the tool will be much simpler.

The Python library was written by Chris Sasarak as part of his Masters project[5]. This research is an

125 extension of his work.

### 2.3. The Handwritten Mathematical Expression Recognition System

RSLib will be used to evaluate the latest entry to the CROHME competition by the Document and Pattern Recognition Laboratory (DPRL) at the Rochester Institute of Technology in Rochester, New York, USA.

130 The Competition on Recognition of Online Handwritten Mathematical Expressions, or CROHME, is an annual competition for pattern recognition researchers working in this area of research[6, 7, 8]. The *online* part of the title refers to the fact that the stroke information is known. The input data originates from a tablet or other digital input device (as opposed to a scanned image) and is in the form of discrete, digital samples, separated into strokes by pen-up/pen-down events.

135 Consider these stages in the context of a system recognizing handwritten mathematical expressions. The segmentation stage will determine which strokes comprise each individual symbol in the expression. The classification stage will consider each symbol as determined by the segmentation stage, and will decide what each symbol is. Finally, the parsing stage will determine how the expression is formatted and how the symbols are positioned relative to one another (eg. superscript, subscript, etc.)

140 In RSLib, all of this information would make up a final interpretation. Each stage will receive an interpretation from previous computation and will use that interpretation to make a decision, the results of which will be used to update the interpretation. The final result of a math recognition system - the final interpretation - will contain all of the information about which strokes comprise which symbols, what the symbols are, and how they relate. From start to finish, the system works with interpretations, starting with  
145 unlabeled strokes and ending with labeled groups of strokes with defined relationships.

Segmentation, classification and parsing need not be (and often aren't) strictly sequential. Analysis from the parsing stage, for example, may constrain symbol classification based on the structure of the expression, or the segmentation stage may make use of the classifier to validate segmentation candidates. The trace graph produced by RSLib will show which decisions were made and in what order, which is useful in systems where  
150 the decision points are not necessarily fixed. In this system, however, the three stages will be sequential.

### 2.4. Changes to RSLib

Several changes were made to the library based on our experience as end users.

#### 2.4.1. Saving and Loading Interpretation Sets

Because of the way the interpretations are implemented, interpretation objects are not directly serializ-  
155 able, which means they cannot be saved using a serializing method such as the one employed by Python's *Pickle* library. To work around this limitation while maintaining the simplicity of the serialization process, two functions were added: *saveInterp* and *loadInterp*. The *saveInterp* function deconstructs the interpretation object into its vital (and serializable) information and uses *pickle* to save it. The *loadInterp* function

does the opposite; it loads the information from the pickle file and recreates the interpretation object. These  
160 functions allow for interpretations to be stored and re-loaded as-is. An example of this functionality can be  
seen in Figure 5, which demonstrates the process of reusing interpretations via RSLib. This functionality can  
also be used for loading saved interpretations for use in separate scripts for analysis.

#### 2.4.2. Creating Sub-decision Points

An additional function was added to the Strategy class. The *rsl\_post* function allows for the dynamic  
165 addition of sub-decision points, and their interpretation data, within a decision function.

Systems generally have several discrete, overarching stages. These can be instrumented and enqueued  
in the strategy object, as has been demonstrated. A user of RSLib, however, may want to dynamically log  
decision points within these stages, in order to record more detailed intermediate information.

The usage of the post function is very similar to the process of enqueueing a decision function in the  
170 strategy object. The function takes a label string, which is associated with the decision point, as well as  
a function and argument list. The function follows the same format as the main decision point wrapper  
functions, and is similarly called with the given argument list as well as the current set of interpretations  
and the strategy object. The resulting interpretation set is recorded with the decision point.

The label argument need not be unique for each call to the post function. Labels will be appended with  
175 a numerical index corresponding to the number of times the label has been used previously, automatically  
imposing a chronological ordering on the decision points.

This function was added to facilitate a more natural method of recording the intermediate interpretations  
within the parsing algorithm of the CROHME system. Each time the relation tree is updated in the parsing  
algorithm, the post function is used to create a new decision point with the updated interpretations. This  
180 produces a step-by-step history of the creation of the relation tree, from the first decision to the final result.  
Example usage of the post function as well as the decision function used in the parsing algorithm is shown  
in Figure 1.

With each step of the parsing algorithm recorded, a step-by-step visualization of the production of the  
final relation tree can be produced automatically by iterating over the sub-decision points and creating a  
185 graph for each intermediate relation tree. Such a visualization may allow researchers to determine precisely  
where errors occur, and observe how those errors cascade through the rest of the algorithm. An example of  
this visualization can be seen in the case study.

#### 2.4.3. Obtaining the Set of Unique Interpretations

Another addition to the library is the ability to determine the unique set of interpretations within  
190 an interpretation set.

The *getUniqueInterps* method, part of the strategy object, takes a regular expression string specifying  
which decision point(s) are desired. The method returns a dictionary from interpretation objects to the  
list of decision point labels associated with them. This dictionary allows for the determination of which

---

```

def parse_postFn(relation_tree, interps, strategy):
    # Get the current interpretation.
    interp = interps[0]

    # Update the interpretation with the new relation tree.
    interp.relation_tree = copy.deepcopy(relation_tree)

    # Return the updated interpretation.
    return [interp]

# Create a subdecision point with the given label using postFn.
strategy.rsl_post(label, parse_postFn, relation_tree)

```

---

Figure 1: The rsl\_post Function

interpretations are duplicates, but perhaps of more use is the ability to take an interpretation object and easily determine which decision point or points produced it.

This process requires the ability to produce a hash value of an interpretation, so the interpretation object itself can be used to index into a dictionary. This is done by hashing the string representation of the interpretation. The default string method of the interpretation object will produce a string containing a thorough, recursive string representation of all of the interpretation's attributes in alphabetical order.

In order for this to work properly, all objects contained within an interpretation *must* have a defined `__repr()` function, preferably with descriptive content. Without the `__repr()` function defined, an object's string representation will contain memory address information, which is not consistent between objects. This assures that the string is descriptive enough to differentiate between different interpretations, and consistent enough to produce the same hash value between different interpretation objects with the same values.

#### 2.4.4. Intersection and Difference of Interpretation Sets

Two operations that prove very useful in performing comparisons between interpretation sets are intersection and difference. These two operations, which are also methods of the strategy class, can be used to easily determine which interpretations are the same and which interpretations differ between two strategies.

Both operations take as arguments the strategy object to compare against, and a regular expression string to specify the decision points of interest. They both return a dictionary mapping decision labels to the single, matching interpretation in the case of intersection, or to the pair of differing interpretations in the case of difference.

These functions can make comparisons between the results of two different strategies easier. By using the difference function, the mismatches between the two strategies are immediately available for analysis; no additional work is needed. Similarly, the intersect function can produce the similarities between the

strategies for analysis.

### 3. Case Study: Analyzing the Math Recognition System

#### 3.1. Instrumentation of the CROHME System with RSLib

No changes were made to the functionality of the CROHME system<sup>1</sup>, but minor modifications were made to the structure of the program in order to make the decision points contained and easily instrumentable. Two stages, the classification stage and label graph output code, were moved from their original positions to their own separate functions. These changes facilitated the creation of the necessary wrapper functions for RSLib while preserving the function of the system.

The process of instrumenting the decision algorithms is generally straightforward. For each algorithm, a wrapper function is created to handle the utilization and updating of the interpretation object. These functions each take as arguments whatever information is necessary for the decision function, as well as the current set of interpretations and the strategy object. At the most basic level, these wrapper functions consist of two or three steps. First, if necessary, data needed for the decision algorithm is retrieved from the interpretation object. Next, the decision function is called with the appropriate arguments. Finally, the interpretation set is updated with the desired information created by the decision function. As an example, Figure 3 shows the wrapper function for the segmentation algorithm. Three of these decision point wrappers were created for the CROHME system; one each for the segmentation, classification and parsing stages.

In addition to the decision functions, the strategy also contains a reporting function. The purpose of this reporting function is to save out the label graph file for the current equation, as well as to pickle the final interpretation for later use and analysis. The use of this reporting function can be seen near the end of the code in Figure 2. The function `reportFn` is a closure, created by the `writeLgFn` function. Because the `reportInterps` function of the strategy object only passes the interpretation to the given function, the other necessary arguments must be specified beforehand, which makes the closure method a natural choice.

The process of enqueueing the decision points as a strategy is similarly straightforward. First, the interpretation type and initial interpretation are created. Next, the strategy object is created using the interpretation type and initial interpretation, as well as an optional reporting function. The decision functions, their arguments, and a label for the decision point are then queued up in the strategy object using the `append` function. Lastly, the strategy is run. At this point, the reports can be printed or saved, and any other output can be performed, including saving a trace graph of the system or pickling the interpretation. Figure 2 shows this process in code.

The interpretation type specifies all of the fields that will be saved in the interpretation. These fields are later filled with information from the three decision points of the system. Figure 4 explains the information stored in the final interpretation. Truth data to validate against is saved in a separate interpretation.

---

<sup>1</sup>Source code available at [https://github.com/DPRL/CROHME\\_2014](https://github.com/DPRL/CROHME_2014)



This figure demonstrates the creation and execution of a strategy using RSLib. First, the interpretation type for the system is created, specifying a label for the type, as well as a list of attributes an interpretation will contain. A strategy is then created with the interpretation type, as well as an initial interpretation, which contains no data in this example. The main decision functions are then added to the strategy in order of execution using the append function. Each append takes, as an argument, a tuple containing the label to associate with the decision point, the decision function to run, and the list of arguments to pass to the decision function. The *eq* argument in this example is an object used by the CROHME system which contains all of the information about an equation. After running the strategy, a report function is created and passed to the strategy for execution. The reportInterps function takes a regular expression argument specifying which decision point or points upon which to execute the report function.

---

```
# Create the interpretation type.
iType = rsl.makeInterpretationType("crohme", ["strokes", "segments", "labeled_segments",
        "symbol_candidate_list", "relation_tree"])

# Create initial interpretation.
initial_interp = iType()

# Initialize strategy.
strategy = rsl.Strategy(iType, initial_interp)

# Add segmentation, classification and parsing stages.
strategy.append(("segmentation", segmentFn, [eq]))
strategy.append(("classification", classFn, [eq, 0_eq]))
strategy.append(("parsing", parseFn, []))

# Run the strategy.
strategy.run()

# Create the closure for the report function.
reportFn = writeLgFn(eq, filename, output_path, pickle_path + 'results/')

# Write LG and pickle final interpretation via a report.
strategy.reportInterps("parsing", reportFn)
```

---

Figure 2: Creating, Enqueuing and Running the Strategy

This figure shows an example of a decision function used by RSLib. The function takes the single argument *eq* specified when the function was enqueued in Figure 2. Each decision function is also given the current interpretation set, as well as the calling strategy object, as additional arguments. This function performs the intended segmentation, then updates the current interpretation and returns it. This interpretation will now be stored in the final interpretation set, associated with the given label for this decision point.

---

```
def segmentFn(eq, interps, strategy):
    # Run the segmenter.
    eq.lei_CROHME2013_segment()

    # Get the interpretation
    interp = list(interps)[0]

    # Gather the segmentation results from the equation object.
    # Format: [ [0], [1, 2], [3], [4, 5, 6], .... ]
    seg_data = []

    for seg in eq.segments:
        seg_data.append(list(seg.strokes))

    # Make sure the strokes are in time order.
    interp.segments = sorted(seg_data)

    # Return the updated interpretation.
    return [interp]
```

---

Figure 3: Wrapper Function for Segmentation

---

```

# Reload the interpretation.
oldInterp = rsl.loadInterp(filename)

# Pull out the saved segmentation.
saved_segmentation = oldInterp.segments

# Add the segmentation information to the equation object.
eq.segments = saved_segmentation

# Continue with classification and parsing stages...

```

---

Figure 5: Reusing interpretations

After saving an interpretation, it can  
 250 be reloaded and reused in lieu of re-  
 running parts of the system. Figure 5  
 shows an example of an interpretation  
 being reused to determine segmentation  
 instead of segmenting the data again.  
 255 This method can save a lot of time if  
 only the later stages of the system need  
 to be rerun.

### 3.2. Studies

#### 3.2.1. Step-by-Step Recording and Visu- 260 alization of the Parsing Algorithm

With the addition of the *rsl\_post*  
 function, decision points and their inter-  
 pretations can be recorded dynamically.  
 With this functionality, it is possible to  
 265 instrument the parsing algorithm to save  
 a step-by-step record of the relation tree  
 as it is built.

Instrumentation of the algorithm required the modification of the parsing function itself, rather than simply wrapping the function. The process, however, remained fairly straightforward.

270 First, the decision function to be called on each update was created. This function simply saved a copy of the relation tree in the current interpretation.

#### **strokes**

The digital stroke data of the equation. Each stroke is a list of (X, Y) coordinates.

#### **segments**

The symbols (groups of strokes) as produced by the segmentation algorithm.

#### **labeled\_segments**

The segmentation results along with the top three symbol labels from the classification algorithm.

#### **symbol\_candidate\_list**

The output of the classification algorithm. Required as input for the parsing algorithm.

#### **relation\_tree**

The results of the parsing algorithm. A tree structure describing the relationship between symbols in the expression.

**Figure 4:** Interpretation Fields for Math Recognition System

With the decision function created, a call to the `rsl_post` function was added immediately after each modification to the relation tree within the parsing algorithm. In total, seven positions in the code required the function call.

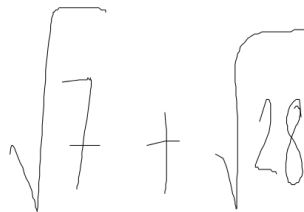
A handwritten mathematical equation showing the sum of two square roots:  $\sqrt{7} + \sqrt{28}$ . The numbers 7 and 28 are written in a simple, slightly irregular hand.

Figure 6: Test Equation for Parsing Visualization

275 The result of the instrumentation is the addition of a number of sub-decision points within the interpretation set, each corresponding to a single update of the relation tree. Each of these decision points are associated with a given label, as specified in the calls to `rsl_post`. The set of decisions can be selected using a regular expression, and due to the automatically appended numbering, can be sorted in chronological order.

280 The parsing stage of the math recognition system was instrumented using this technique. At each update to the relation tree in the parsing algorithm, a post was performed to save the interpretation with the new tree. As a result, the final interpretation produced by the system contains an ordered history of each step in the parsing algorithm.

Using this history, a reporting function was created to step through the parsing history and create a visualization of the relation tree. These visualizations, as well as visualization of the input equation, are then saved as a PDF. The result of this visualization process can be seen in Table 1, which displays the stages of the parsing algorithm as it operated on the equation in Figure 6

A visualization process like this can help researchers see precisely where an algorithm makes errors, and how those errors cascade through the rest of the process.

### 3.2.2. Analysis of Segmentation Preprocessing Feature via Comparison

290 One of the benefits of RSLib is the ability to repeatedly produce uniform, well-structured interpretation results after instrumenting a system. This makes the process of performing side-by-side comparisons of resulting interpretations easier. By comparing the results of a modified system against a baseline, quantitative data about the real utility of those modifications can be obtained.

295 Within the segmentation stage, there is a small preprocessing step in which all touching or overlapping strokes are automatically merged. A comparison can be performed between the results of the original system against those of the system with this feature disabled in order to quantitatively determine the benefit of this feature.

With the CROHME system instrumented with RSLib, the system can produce interpretations from the segmentation stage both with and without this stage. The two resulting interpretations can then be analyzed

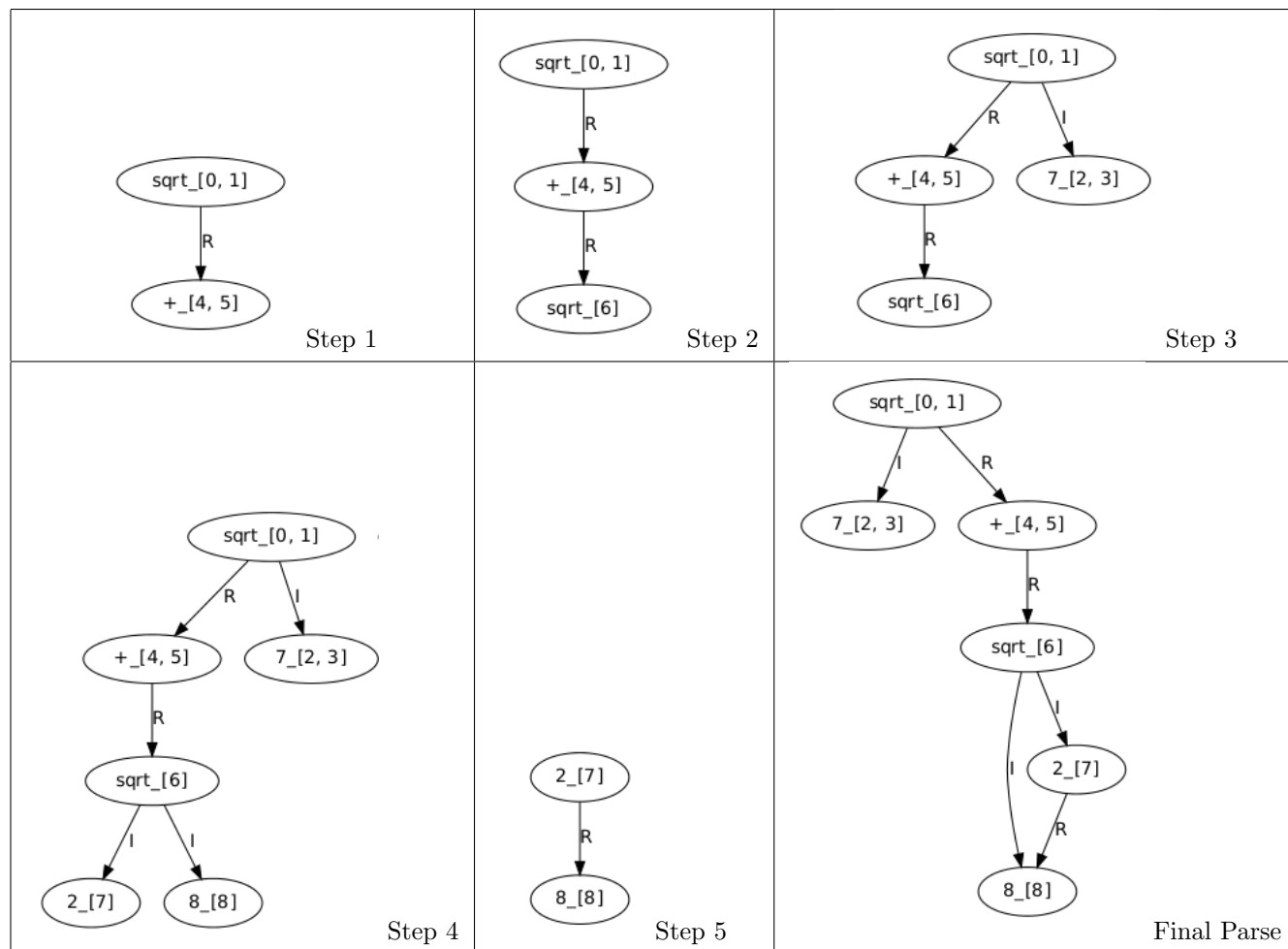


Table 1: Step-by-Step Parsing Visualization of the Equation in Figure 6

300 against the ground truth, and the results can be compared to determine not only whether the merge step is beneficial, but precisely to what extent.

This process was accomplished using RSLib’s facilities. Two separate strategies were created and run: one performing the merge preprocessing step, and the other forgoing it.

305 With the final interpretations from each strategy, as well as the ground truth, a reporting function was used to calculate segmentation rates for the two interpretations and compare the results.

The results in Figure 8 show a clear, albeit slight, benefit in performing the merge preprocessing. With the addition of ground truth data for symbol labels, a more detailed symbol level analysis of the results can be performed. Looking at the specific types of symbols the segmentation stage has issues with, as well as considering the differences between the merge and no merge systems, may provide useful insight about the  
310 segmentation algorithm.

Figure 7 shows a comparison of missegmentations of the top ten most missegmented symbols between the merge and no merge systems. The graph shows the number of missegmentations for each symbol for

each method, in red and blue, and the total number of *correct* segmentations for comparison, in green. This graph shows that, in terms of raw numbers, the two systems are nearly identical in performance, as was shown in the statistics in Figure 8.

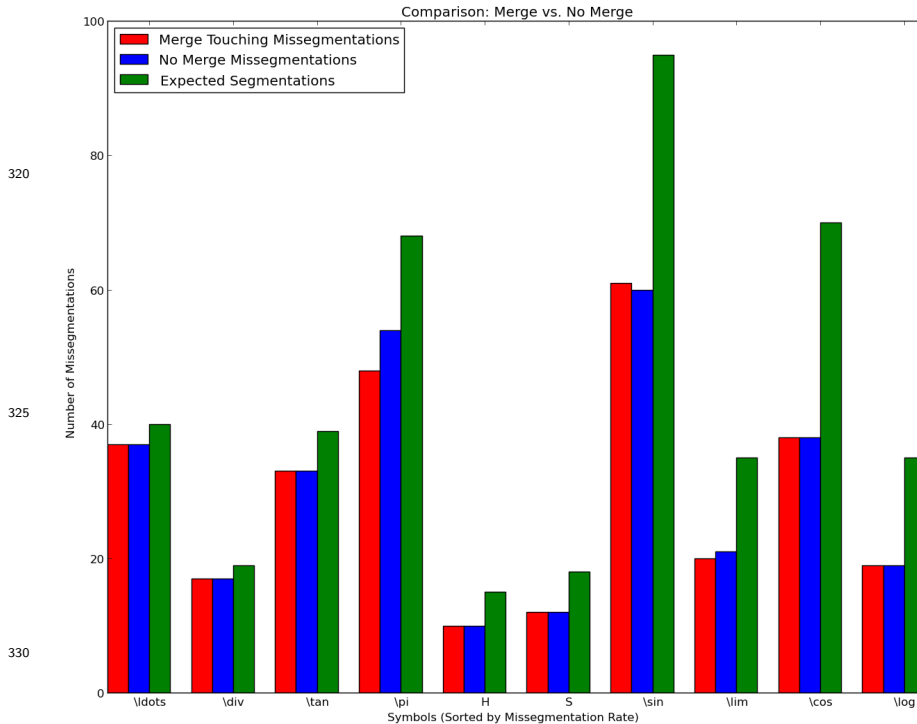


Figure 7: Merge vs. No Merge - Missegmentation Rates by Symbol  
 ing preprocessing step, affirming its inclusion in the algorithm. It also demonstrates the benefit of RSLib  
 in making the process of performing this sort of side by side comparison, and producing useful, quantitative  
 analysis on features straightforward.

### 3.2.3. Comparison of Different Stroke Pairing Methods in Segmentation Algorithm

The previous study has demonstrated the utility of RSLib in performing side by side comparisons of different methods. In this study, a more significant modification will be evaluated using the library.

The math recognition system uses a segmentation algorithm proposed by Lei Hu et al.[9] This algorithm works by stepping through the strokes in time order, evaluating the current stroke and the next stroke in the sequence, and deciding, using an AdaBoost classifier, whether the next stroke should be merged with

System	Recall	Precision	F-Measure
Original (Merge)	85.53%	86.06%	85.79%
Modified (No Merge)	85.64%	85.24%	85.44%

Figure 8: Merge vs. No Merge Segmentation Comparison

For the most part, the systems perform very similarly. The differences, however, are apparent in a few symbols. The merge system performs better on symbols with multiple strokes, such as *pi* and *lim*, symbols where the strokes are likely to be touching. On the other hand, the merge system performs worse on several single stroke symbols, where errant strokes may have overlapped other unrelated symbols.

This comparison clearly and quantitatively demonstrates the benefit of the merge touch-

System	Recall	Precision	F-Measure
Sequential	85.53%	86.06%	85.79%
1NN	81.90%	74.34%	77.94%
2NN	81.18%	77.95%	79.53%

Figure 9: Segmentation using Different Pairing Strategies

the current stroke and incorporated within the current symbol, or split from the current symbol to form the next.

345 This algorithm, though effective, may benefit from the use of a different pairing scheme. Because the algorithm only pairs strokes in sequential order, it is inherently ineffective when considering expressions where strokes within a symbol are not in sequential order; for example, a tittle over an  $i$  or  $j$ , or the cross on a  $t$  added after the rest of the expression has been finished. Another potential shortcoming is the fact that only one stroke, the last stroke in the current symbol, is considered when evaluating the next stroke.  
 350 Considering additional strokes may be beneficial.

The altered algorithm, which will be compared against the original, will choose stroke pairings based upon bounding box center distances, rather than time sequential ordering. In addition, two variations will be tested. The first will pair each stroke with its nearest neighbor, and the second will pair strokes with their two nearest neighbors. The 1NN variation will allow for a reasonable comparison between the two pairing  
 355 strategies, while the 2NN variation will explore the benefit of adding additional pairings.

The expectation is that the nearest neighbor strategy should result in improvements in cases where sequential ordering is not appropriate. Additionally, the 2NN variation will provide extra opportunities for mistakenly split strokes to be merged with the correct symbol. It may, however, result in a significant increase in incorrectly merged strokes, as it places more emphasis on the merge operation by providing more  
 360 opportunities to merge for each stroke.

The segmentation results in Figure 9 clearly show that the original time sequential pairing strategy performs significantly better than both nearest neighbor strategies. Of the two nearest neighbor strategies, considering additional neighbors provided a significant boost in performance, though not nearly enough to compete with the sequential system.

365 Figure 10 shows the comparison in the same format as Figure 7 in the previous study. Here, the top ten missegmented symbols (sorted by rate) of the original sequential algorithm are compared against the 1NN and 2NN pairing strategies. The total number of segmentations is also displayed for comparison. This graph shows a clear pattern of better performance by the original algorithm as compared to the nearest neighbor variants. In general, the two nearest neighbor algorithms display similar performance, but the 1NN variant  
 370 performs better on these particular symbols.

One confounding issue in this comparison that may have affected the performance of the nearest neigh-

bor strategies is that the AdaBoost classifier used in the merge/split decision was not retrained for these strategies. The classifier was trained using the original sequential pairing, which gives an advantage to that strategy over the nearest neighbor strategies. The classifier was not retrained due to time constraints.

#### 375 4. Conclusion

While no potential improvements were discovered for the CROHME system, the library proved useful in evaluating alternate strategies and producing quantitative data demonstrating the superiority of the original methods. The straightforward and repeatable process of side-by-side comparisons demonstrated in this case study can be performed again and again for each addition or modification to the system, providing objective feedback about the benefits.

Based upon the history of success of RSLib’s predecessors,

in addition to the utility demonstrated in this case study, it is safe to say that RSLib may prove to be a valuable tool to pattern recognition researchers.

#### References

- [1] R. Zanibbi, D. Blostein, J. R. Cordy, Decision-based specification and comparison of table recognition algorithms., in: S. Marinai, H. Fujisawa (Eds.), Machine Learning in Document Analysis and Recognition, Vol. 90 of Studies in Computational Intelligence, Springer, 2008, pp. 71–103.
- [2] R. Zanibbi, Programmer’s Guide to the Recognition Strategy Language (RSL) (2011).
- [3] B. Holm, Evaluation of RSL History as a Tool for Assistance in the Development and Evaluation of Computer Vision Algorithms, Master’s thesis, Rochester Institute of Technology (2011).

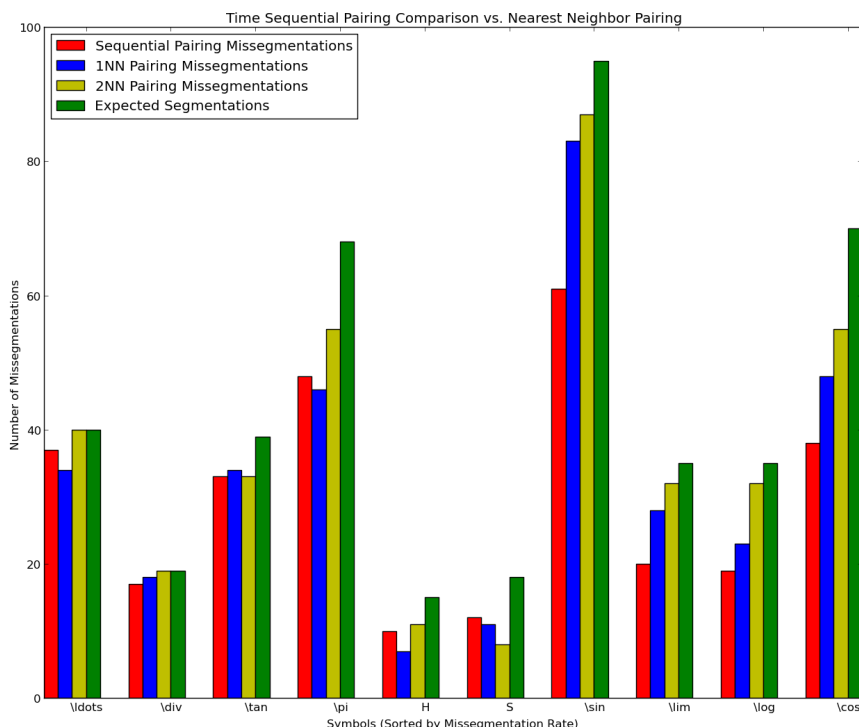


Figure 10: Missegmentations in Different Stroke Pairing Methods



- [4] P. Hudak, Domain Specific languages, in: Handbook of Programming Languages, Vol. III: Little Languages and Tools, MacMillan, Indianapolis, 1998, Ch. 3, pp. 39–60.
- [5] C. Sasarak, RSLib Programmer’s Manual (2014).
- [6] H. Mouchere, C. Viard-Gaudin, D. H. Kim, J. H. Kim, U. Garain, Crohme2011: Competition on Recognition of Online Handwritten Mathematical Expressions, in: Document Analysis and Recognition (ICDAR), 2011 International Conference on, 2011, pp. 1497–1500. doi:10.1109/ICDAR.2011.297.
- [7] H. Mouchre, C. Viard-Gaudin, D. H. Kim, J. H. Kim, U. Garain, ICFHR 2012 Competition on Recognition of On-Line Mathematical Expressions (CROHME 2012)., in: ICFHR, 2012, pp. 811–816.
- [8] H. Mouchre, C. Viard-Gaudin, R. Zanibbi, U. Garain, D. H. Kim, ICDAR 2013 CROHME: Third International Competition on Recognition of Online Handwritten Mathematical Expressions., in: ICDAR, IEEE, 2013, pp. 1428–1432.
- [9] L. Hu, R. Zanibbi, Segmenting Handwritten Math Symbols Using AdaBoost and Multi-scale Shape Context Features, in: Proceedings of the 2013 12th International Conference on Document Analysis and Recognition, ICDAR '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1180–1184. doi:10.1109/ICDAR.2013.239.