EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2008)
C. Alvarado and M.- P. Cani (Editors)

AlgoSketch: Algorithm Sketching and Interactive Computation

Chuanjun Li[1], Timothy S. Miller[1], Robert C. Zeleznik[1] and Joseph J. LaViola Jr.[2]

[1]Brown University, Department of Computer Science, Providence, RI USA
[2]University of Central Florida, School of EECS, Orlando, FL USA

Abstract

We present AlgoSketch, a pen-based algorithm sketching prototype with supporting interactive computation. AlgoSketch lets users fluidly enter and edit 2D handwritten mathematical expressions in the form of pseudocode-like descriptions to support the algorithm design and development process. By utilizing a novel 2D algorithmic description language and a pen-based interface, AlgoSketch users need not work with traditional, yet complex 1D programming languages in the early parts of algorithm development. In this paper, we present the details behind AlgoSketch including the design of our 2D algorithmic description language, support for iteration and flow of control constructs and a simple debugging trace tool. We also provide some examples of how AlgoSketch might be used in the context of image analysis and number-theoretic calculation problems found. Based on preliminary user feedback, we believe AlgoSketch has the potential to be used to design and test new algorithms before more efficient code is implemented. In addition, it can support users who may not be familiar with any advanced programming languages.

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [User Interfaces]: Graphical user interfaces

1. Introduction

In traditional algorithm design, pencil and paper is often used in the early stages of the design process to create flow charts or pseudocode. This creativity and conceptual understanding phase is followed by implementation in an advanced language, such as C/C++, Java, C#, Mathematica or Matlab code. Using pencil and paper in the algorithm design process allows for fluidity of thought and clear expression of ideas. However, paper is a static medium and it is not possible to visualize algorithm results and behavior using it in isolation. Unfortunately, the exploration of an algorithm's behavior sketched out on paper requires a transformation to code in a subsequent implementation that can be time consuming and error prone. In addition, any necessary change in the algorithm design could require costly changes in implementation.

Mitigating the transition from pencil and paper algorithm design to programming language implementation requires the ability to go directly from pencil and paper to program execution. The ability to recognize handwritten 2D mathematical expressions using a pen-based computer (see Figure 1) makes this possible [LZ04, ZML07]. Since a series of handwritten 2D mathematical expressions can be considered a simplified algorithm we can bridge the gap between pencil and paper and executable code. Thus, the *idea* behind algorithm sketching not only removes the subsequent implementation, but can also support the display of intermediate results for better understanding of algorithm behavior, algorithm debugging, and hand-drawn diagrams as shown in Figure 2.

In this paper, we present AlgoSketch, a prototype system that focuses on entering algorithms using a mathematically-based 2D algorithmic description language. The AlgoSketch prototype not only introduces flow of control constructs into pen-based computing, but also makes available intermediate results and variable values for execution-tracking and problem-fixing. Intuitive 2D expressions, including matrices, are also supported for algorithmic computing. We also show applications of algorithm sketching to image processing and number theoretic computation, and discuss the prototype's existing limitations and future work.

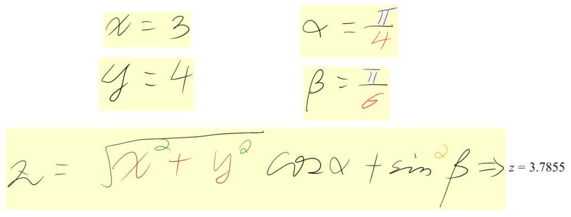


Figure 1: *Computing from 2D mathematical expressions. The system colorizes ink strokes based on the recognized symbols' semantic meanings. Recognition of the handwritten expressions is discussed in Section 4. Variables in the last expression are replaced with the corresponding values, and the result is available after the evaluation gesture, an ending double arrow.*

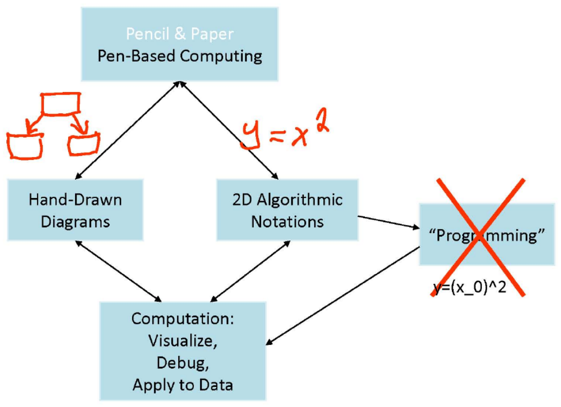


Figure 2: *The concept of pen-based algorithm sketching. Hand-drawn 2D diagrams, notations and visualization results are directly available from sketched notations.*

2. Related Work

The idea of using a sketch-based interface in the early stages of design is not new. For example, Damm et al. used a gestural user interface in their Knight system, a tool for cooperative objected-oriented design [DHT00]. Gross used gestures for creating and editing diagrams for conceptual 2D design tasks [GD96] and, in the 3D domain, Zeleznik et al. used simple sketches for rapid conceptualizing and editing of approximate 3D scenes [ZHH96]. Igarashi et al. also used a sketch-based interface in creating free-form 3D models [IMT99]. In other examples, Forsberg et al. used a sketch-based interface for the rapid creation of musical scores [FDZ98] and Landay and Myers developed a system for prototyping user interfaces [LM95]. Finally, both Alvarado and Kara have developed sketch-based interfaces for the early stages of mechanical design [Alv00, KGS04]. The main difference between these conceptual design systems and our work is that they did not focus the conceptual design of algorithms.

Pen-based interfaces have also been developed for mathematical computation. Chan and Yeung developed PenCalc, a simple pen-based calculator [CY01], while xThink, Inc. developed MathJournal^{TM}, a system designed to solve equations, perform symbolic manipulation, and make graphs. Other pen-based interfaces that support both numeric and symbolic computation include MathBrush [LMM*06] and Microsoft Math^{TM}. These systems are similar to AlgoSketch in that they support computation. However, they do not support flow of control mechanisms and conceptual algorithm design. In addition, these systems do not provide real-time, write anywhere interfaces [ZML07].

The MathPad^2 system [LZ04, LaV07] is the closest in spirit to AlgoSketch. MathPad^2 was designed to let users create dynamic illustrations for exploring mathematical and physical concepts by combining handwritten mathematics and free-form drawings with a pen-based interface. This system let users effectively sketch out small algorithmic descriptions that were used to drive animations. Although MathPad^2 supported flow of control constructs (i.e., iteration and branching), it was limited in the types of algorithms that could be created. In addition, it did not give users real-time feedback when writing mathematical expressions and provided no debugging support. AlgoSketch supports a richer set of possible algorithms in domains such as image processing and number-theoretic computation.

3. AlgoSketch Language Design

In this section, we discuss some specifications and design issues related to algorithm sketching, including support of keywords and flow of control, spatial arrangements of sketches, and scope specification and identification. We also present the use of a trace table construct for facilitating the understanding of the sketched algorithms. The AlgoSketch language specifications are summarized in Table 1.

Flow of control and lexical scoping are an integral part of the algorithm sketching language. AlgoSketch recognizes only the most frequently used constructs including *if, else* for conditionals and *for* for loops. Based on our experience, we believe these constructs are necessary and sufficient for specifying the algorithms AlgoSketch users are likely to attempt. More constructs, such as *do, while*, etc. could be added if the need arises. Scope specification for these constructs utilizes the concept of overloaded notations in conjunction with the 2D spatial layout of the algorithmic statements.

3.1. Algorithm Sketching Language Constructs

The flow of control constructs use shorthand symbolic notations when possible, and resort to keywords otherwise. For

Table 1: *AlgoSketch Language Specifications*

Constructs and Notations		Descriptions
Constructs	if	for conditionals. Keywords are used.
	else	
	for	for *for* loops. $\forall$ is used for it.
Notations	$\longleftarrow$	for function definition and return
	$\forall$	a shortcut for *for*
	$\in$	for argument type specification and loops
	$\sim$	for omitted data. Order is enforced.
	$=$	for assignment and equality
	$//T()$	specifically for trace table
	$\nearrow\searrow\longrightarrow$	set trace point if any one is inside function definition
	$//$	for comments. It can cover multiple lines

example, we define the compact mathematical symbol $\forall$ to mean the *for* loop construct. Alternatively, the keywords *if* and *else* are typically used directly, although shortcut notations using braces (Figure 3) are also available when each conditional expression can fit on a single line. Other shortcut notations are also used, such as an ending left arrow ($\longleftarrow$) for function definition, a left arrow for *return*, and a tilde ($\sim$) as shown in Figure 3. More conventional ellipses ($\cdots$) can be used in place of a tilde ($\sim$), however, both entering and recognizing handwritten ellipses is more difficult.

These notations can be overloaded, as they are in general mathematics notations, to have different meanings in different contexts as shown in Figure 3. The example on the left side of the figure defines an integer from a sequence of digits, whereas the tilde surrounded by commas defines a sequence of integers, with the sequence number being the input integer, and the values of the integers being specified by the ending expression (x^2) as shown by the result of the function call $g(9)$. The right example computes the sum of two equal digit integers. The tilde in $\forall i \in 0 \sim n$ denotes the integer range $[0,n]$, while the tilde in $\longleftarrow (w_{n+1}w_n \sim w_1w_0)$ denotes the omitted digits in an integer value. Notice that no matter what contexts a tilde might be in, it enforces an order on the data it represents. An equal sign ($=$) is also overloaded for both equality and assignment depending on its context. Likewise, different notations can be overloaded according to the usage domain.

Notations can still be clear even though they have different meanings in different contexts. For example, in mathematical notation associated with cryptanalysis, $x, y \in P[n+1]_b$ can denote that x and y are positive integers of $n+1$ digits with base b, while x_i, y_i, and w_i denote the ith digit of x, y, and w, respectively. Some notations can cover multiple "lines" as shown in Figure 3 for the *if-else* comment.

Although the execution order of algorithmic statements plays a key role in making a program function as expected, statements can be positioned in different ways so long as they can be reached in the required order. For example, sequential statements can be positioned sequentially from top to bottom, or they can be arranged as in Figure 1. For the flow of control constructs, such as the *if*, *else*, or *for* statements, starting on new lines would make the sketches look neat and make reading and understanding easier. Hence, we require each flow of control construct to start on a new line, but any other statements can follow each other horizontally. Since we associate no baseline with the algorithmic sketches, a new line refers to the lower space that does not overlap with previous statements vertically.

The scope of flow of control constructs is based on visual indentation. This not only improves the visual clarity of the algorithms but also avoids the problems associated with recognizing more subtle punctuation or verbose lexemes, such as the various scope identifiers/tags in any HTML or LaTeX file. Identification of horizontal indentations depends on the size of the indented space and the sizes of the drawn symbols. Computing the size of the drawn symbols can be inefficient if there are many symbols and poorly defined, if there are very wide symbols such as $\surd$ and division lines. Thus, we use the difference between the horizontal starting position of flow of control constructs and those of following statements to determine the presence of an indentation. If the difference in the horizontal starting positions is larger than the height of the construct statement, there is an indentation and the lower statement is within the scope of the construct. Otherwise, it is outside of the scope of the construct.

3.2. Variable Tracing

AlgoSketch not only offers the flexibility and fluidity of entering and manipulating sketched/handwritten expressions or statements, but also provides a tool for understanding the execution of sketched algorithms. For example, a trace table of the run-time values of indicated variables at a specific lexical position in the algorithm can be displayed at a target location. To produce such a trace, an arrow ($\nearrow$, $\searrow$, or $\longrightarrow$) is drawn from the trace point to where the trace display

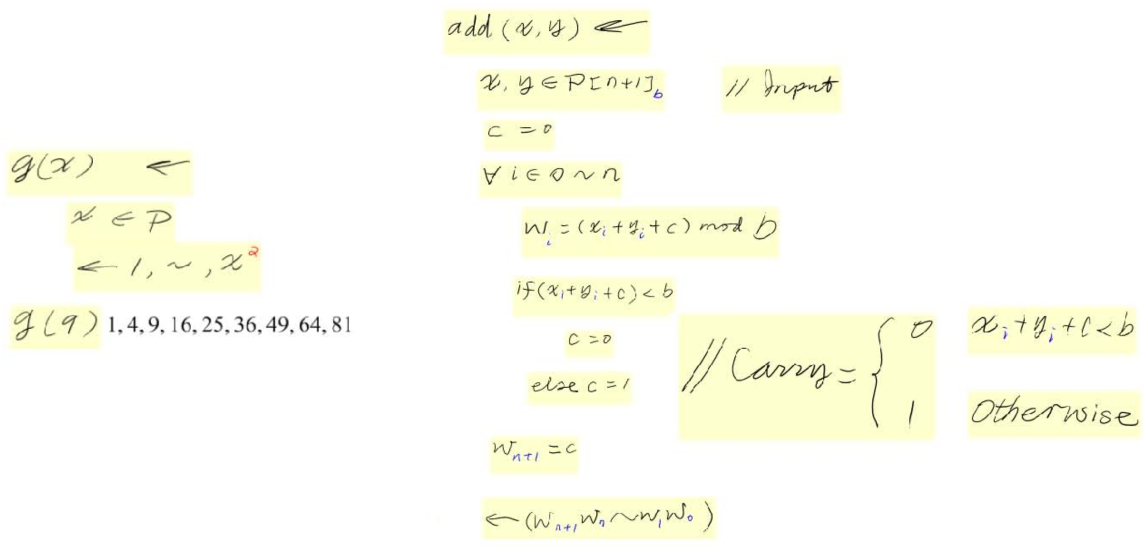


Figure 3: *Notation extension for algorithm sketching. The same notation (∼) can have different meanings in different contexts.*

should be shown. We will use an example in Section 6 to further illustrate these trace tables.

4. Segmentation and Recognition of Mathematical Expressions

AlgoSketch is built in the context of MathPaper, a pen-based mathematical expression recognition and computation system. In this section, we briefly summarize how MathPaper turns strokes into mathematical expressions. More details can be found in [ZML07].

Figure 4 shows the steps required to recognize a mathematical expression as strokes are entered. When a new ink stroke is input, it is first tested against a set of command gesture templates. Command gestures are ink strokes that are distinct from mathematical symbols which immediately perform upon input, such as lasso selection, symbol dragging, stroke deletion, menu selections, etc. If a stroke is recognized as a command gesture, its action is performed and the stroke is deleted without ever being sent to the mathematics recognition engine. All other strokes are passed to the symbol recognizer and then the expression recognizer.

The symbol recognizer is a large rule-base of ad hoc boolean algorithms each finely tuned to recognize a symbol allograph (e.g., different ways of writing the same character). Each allograph is mapped by default to a mathematical symbol, although users can override this mapping if they wish. There are no formal guidelines or restrictions for writing these algorithmic rules, we have found that finding cusp-based features is often helpful. If none of the rule-based recognizers match an input stroke(s), we fall back on the recognition label assigned by the Microsoft handwriting recognizer.

Based on the symbol recognition labeling and additional ad-hoc spatial tests, strokes are grouped into ranges. A range is a group of symbols that collectively constitute a single, complete expression or line of an algorithm. When new strokes are drawn or deleted, only the ranges that are directly affected by the action are parsed.

The parsing of the recognized symbols in one range includes two stages, called Parse 1 and Parse 2. Parse 1 examines all symbols in one range and collects symbols for a common baseline, and stores the common baseline symbols in a Line object. Depending on the types of symbols in a Line object, each symbol may have one associated superscript Line object and/or one associated subscript Line object. Here a super/subscript Line records the geometric relationship between the parent symbol and its child symbols. The output of Parse 1 is a tree of Line objects, with the root being the common baseline. Parse 2 converts Parse 1's geometric representation into a semantic mathematical expression tree. During this process, a language model can be used to coerce changes to either the symbolic or geometric parse structure; for instance, the input sequence 'c''0''5' will be converted into "cos" as long as a single stroke allograph for '5' was recognized .

This process of symbol recognition and parsing is executed in real-time after each stroke is input. The result of the recognition is displayed using one of several different visualization strategies [LLMZ08]. In addition, the mathematical expression that is output can be exported to Mathematica or any one of many symbolic or computational engines.

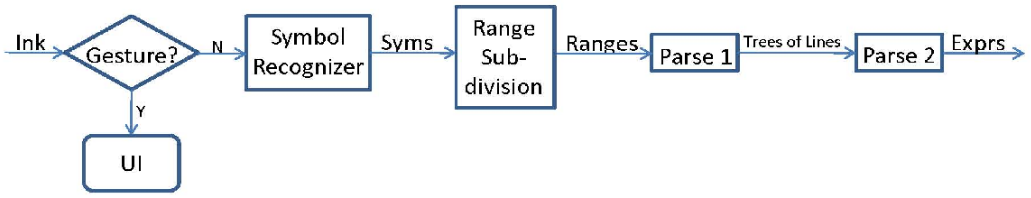


Figure 4: *Dataflow of mathematical expression recognition from ink strokes. Multiple expressions can be recognized, each in one range determined by largely spatial tests of recognized symbols. A tree of Line objects is generated for each range by Parse 1 which determines the geometric structure of symbols in each range, followed by a semantics representation Expr generated by Parser 2 (Figure adapted from [ZML07]).*

5. Parsing and Computation of Sketched Algorithm

In AlgoSketch, mathematical expressions are recognized, parsed and translated for export to Mathematica as described above. Since algorithms involve more notations, syntax and semantics, more work is required to take these descriptions and execute them.

Due to our overloaded notation scheme, precedence needs to be considered for the various symbols or keywords used in algorithm sketching. For parsing convenience, we assign *if, else* and $\forall$ with the lowest precedence such that these flow of control constructs can be easily detected in our parsed expression tree. The $\in$ keyword has lower precedence than a comma, and also lower precedence than multiplication to enable correct parsing of expressions such as $x, y \in P[n+1]_b$, since $P[n+1]_b$ is parsed as the multiplication of P and $[n+1]_b$ by Parse 2. The *mod* operator has higher precedence than multiplication, yet lower precedence than parentheses. The $\sim$ symbol is not treated as an operator, and expressions containing it are parsed in another stage, called Parse 3 as shown in Figure 5.

Parse 3 sorts the expression trees output from Parse 2 into algorithm statements which have the right order and scope information. Expressions are first sorted according to the vertical coordinates of their bounding box's left top corner. After testing whether more than one expression overlaps horizontally, each set of horizontally overlapped expressions is sorted by the horizontal coordinates of the associated bounding box's left top corner. Expressions starting with double slashes (//) are treated as comments and skipped during parsing.

If the algorithm or function name has more than one symbol, Parse 2 will parse them as the multiplication of these symbols, multiplied by the argument list if any. For instance, the function name *add* in Figure 3 would be the product 'a"d"d' before calling Parse 3. Parse 3, will detect the definition of a function and will convert the product into a single word consisting of the individual symbols. Parse 3 will also convert the function definition to be a single expression, with one field being the function name, *add* in the case of the right example in Figure 3, and additional argument fields. Similar parsing is done for function calls.

Before exporting to Mathematica, Parse 3 determines the scope for each flow of control construct by using the indentation information. If the indentation of a lower expression statement is greater than a threshold, i.e., the height of the bounding box of the expression, the lower expression is in the scope of the flow of control construct. Otherwise the scope ends above the expression. Each outermost construct and the expressions in its scope will be converted to form one input statement for Mathematica. The statement is hierarchical and will contain all the nested statements in its scope, and these statements will contain statements for any inner constructs within the scope.

When generating Mathematica input, overloaded notations are disambiguated according to their context. For example, exporting a tilde ($\sim$) to Mathematica does not make sense, so the tilde needs to be replaced with data available at run-time. Hence Mathematica input is generated only when there is a function call as illustrated in Figure 5. When a function call is complete, argument pre-processing is done for value assignment: loading data from a file if the file provides data for any argument, testing if the number and types of arguments are correct, and assigning argument-related data to associate variables, etc. All the pre-processing is exported to Mathematica to improve run-time performance.

The output processing step in Figure 5 handles output issues, such as displaying computation results and the populated trace table if available.

6. Example Applications

This section shows two case study applications for AlgoSketch. The first one illustrates the computation of the factorial of the larger argument from a function call with two arguments, and outputs half of the factorial as shown in Figure 6, while the second one shows the application of AlgoSketch to image processing as shown in Figure 7.

Figure 6 shows the support of flow of control con-

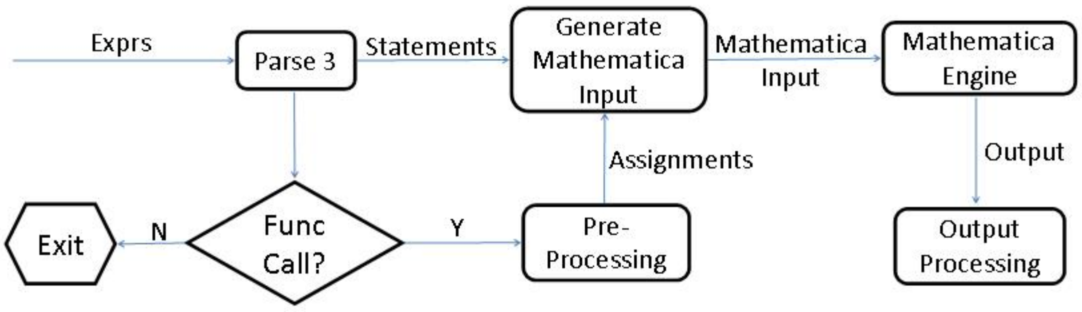


Figure 5: *Data flow of AlgoSketch after Parse 2.*

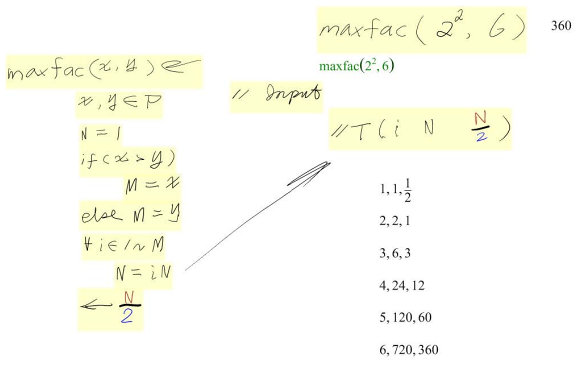


Figure 6: *Application of AlgoSketch to computation. Flow of control and trace table are illustrated in this application.*

structs, including *if*, *else* and a *for* loop. The function call $maxfac(2^2,6)$ can be anywhere outside of function definition, and the function call result is displayed right after the function call. Notice the first argument 2^2 is actually a 2D expression itself. The left arrow ($\longleftarrow$) at the end of $maxfac(x,y)$ is for function definition, specifying that $maxfac$ is a function with two parameters. The left arrow in front of $\frac{N}{2}$ is for returning a value or a 2D expression to a function call.

Indentation is used for the scopes of the *if* and *for* constructs, while the statement $M = y$ is right after the *else* keyword. The statement $\forall i \in 1 \sim M$ is equivalent to *for each increasing integer i between 1 and M*.

The statement in front of $//Input$ specifies the argument types for the function call, in this specific example, two positive integers. We use $//T(...)$ to display trace variables, with variables separated by blank space as shown in this example. Again, the variables can be 2D expressions as shown for $\frac{N}{2}$. This can be very convenient for displaying both individual variables and expressions when values of expressions are expected. The trace table can be at any blank space, and can be lassoed and dragged if needed.

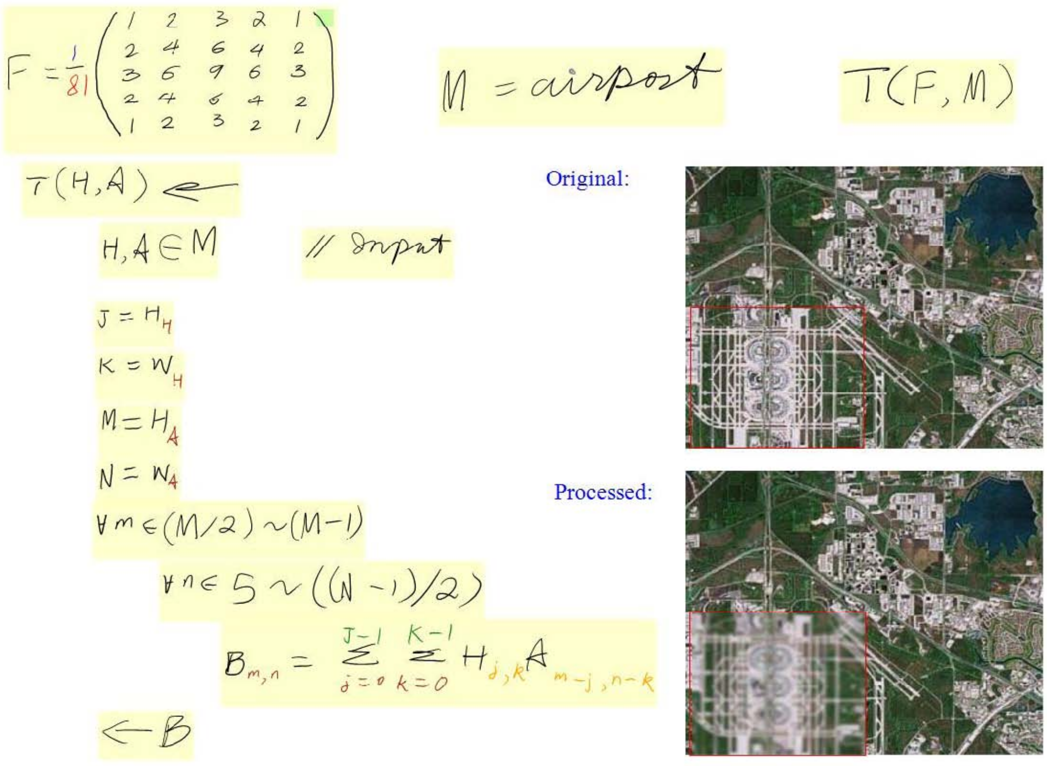
We use an arrow to specify a trace point where variables are to be displayed below the trace table head $//T(...)$. The tail end of the arrow is located at the right of the target statement, indicating that after the statement is executed, the values of the variables are to be traced.

Notice the typeset expression below the function call. Hovering over any sketched expression would display its recognized typeset below the sketch, helping correct recognition error if any.

Figure 7 shows how to process an image using AlgoSketch. The matrix F is a filter to be applied to each selected group of pixels in an airfield image. The image data is stored in a file named airport.jpg. The image file can of be any major type, such as bitmap, jpeg, tiff, and PNG etc, and the file extension can be omitted for input simplicity.

The function takes two matrix parameters, one matrix as a filter, and the other for the original image to be processed. It outputs a processed image with RGB values of selected pixels averaged by applying the filter to each pixel and its neighboring pixels. The four assignments above the outer *for* loop assign the dimensions of the two matrix arguments to the respective variables, which are used for specifying filter values and pixels to be processed. H_A is the height of matrix A, or number of rows in A. W_A is the width of, or number of columns in A. $B_{m,n}$ is for the m,nth pixel of output image B. Our current implementation uses two *for* loops to specify the ranges of pixels to be processed, and processes all RGB values individually for each pixel.

The original image and the processed image can be displayed together with the processing function as shown in Figure 7, giving a direct comparison between the two images, and allowing for revisions to the function. A shortcut is taken by having the sizes and locations of the displayed images hard-coded. They can be easily adjusted by specifying two windows at different locations. The images can be annotated as the red boxes show.



$$F = \frac{1}{81}\begin{pmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{pmatrix}$$

```
M = airport

T(F, M)

T(H,A) ←
    H,A ∈ M        // Input
    J = H_H
    K = W_H
    M = H_A
    N = W_A
    ∀ m ∈ (M/2) ~ (M−1)
        ∀ n ∈ 5 ~ ((N−1)/2)
            $B_{m,n} = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} H_{j,k} A_{m-j,n-k}$
    ← B
```

Figure 7: *Application of AlgoSketch to image processing. 2D input, including matrix input, and 2D expressions, as well as for loops are illustrated in this application.*

7. Limitations and Future Work

AlgoSketch has been proposed for facilitating algorithm design and data analysis or data processing, such as image processing, and for promoting quick understanding of new algorithms. It takes advantage of the flexibility and fluidity of making modifications with little effort, real-time computing and responsiveness, the capability for listing values at any trace point, and for adding annotations using pen-based interaction. The work we have done so far explores mostly the feasibility of algorithm sketching and provides a proof-of-concept prototype. Only a single function definition on a single page is supported. To make it more robust, multiple functions on one or more pages and function calls in functions need to be supported. More data types and flow of control constructs also need to be supported. For example, introduction of recursive calls or even the class concept into AlgoSketch would make it more applicable.

AlgoSketch is built in the context of MathPaper, which has a very friendly user interface for fluid recognition error detection and correction [ZML07]. Hence symbol recognition errors can be conveniently corrected, and debugging can be focused on sketch syntax errors only. If there is a syntax error, AlgoSketch simply does not do any computation, and does not report where the syntax error occurs. More work needs to be done to have sketch syntax errors reported.

For the image processing application, our current implementation algorithmically specifies a rectangular region to be processed. It can be extended such that an arbitrary region can be specified by using the pen. In addition to entering the filtering matrix, it can also be extended for a filtering function to generate the data to be applied to the selected region. Finally, multiple region selection should also be supported.

The trace table shows values of the listed variables at any specific trace point. It would be helpful to be able to trace the value changes of some variables at multiple trace points, and this can be done easily by associating one trace table with multiple trace points. In addition to trace tables, other visualization approaches such as graphs would also be helpful in showing algorithm output

The computation of the sketched algorithm is enabled by translating the recognized algorithm into Mathematica notation. Translating the Mathematica code, or the recognized sketches into other programming languages, such as C/C++, Java or C#, etc. is possible and would boost run-time speed. Support for 2D hand-drawn diagrams for computation and visualization as illustrated in Fig 2 needs to be explored. Currently 2D diagrams are uninterpreted and are for annotations only. Finally, usability studies are needed to evaluate how AlgoSketch performs for real algorithm design, especially if applied to long or computationally intensive algorithms.

8. Conclusion

We have presented AlgoSketch, a pen-based, prototype application for sketching algorithms and interactive computation. Using a novel 2D algorithmic description language based on traditional mathematical notation and special overloaded operators, we support flow of control constructs such as *if, else* and *for* as well as a trace table for tracing algorithm execution and facilitating algorithm debugging. In addition, indentation is used for specifying the scope of the supported constructs. We have also shown two example scenarios for image processing and number-theoretic computation with AlgoSketch, illustrating the feasibility and potential of algorithm sketching. Although there is more work to do on AlgoSketch, we believe our prototype is a good starting point for letting users design and test new algorithms before more efficient code needs to be implemented.

Acknowledgements

This work is supported, in part, with grants from Microsoft Research and IARPA.

References

[Alv00] ALVARADO C.: *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Tech. rep., Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2000.

[CY01] CHAN K.-F., YEUNG D.-Y.: Pencalc: A novel application of on-line mathematical expression recognition technology. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition* (September 2001), pp. 774–778.

[DHT00] DAMM C. H., HANSEN K. M., THOMSEN M.: Tool support for cooperative object-oriented design: gesture based modeling on an electronic whiteboard. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2000), ACM, pp. 518–525.

[FDZ98] FORSBERG A., DIETERICH M., ZELEZNIK R.: The music notepad. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1998), ACM, pp. 203–210.

[GD96] GROSS M. D., DO E. Y.-L.: Ambiguous intentions: a paper-like interface for creative design. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1996), ACM, pp. 183–192.

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: a sketching interface for 3d freeform design. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), ACM Press/Addison-Wesley Publishing Co., pp. 409–416.

[KGS04] KARA L. B., GENNARI L., STAHOVICH T. F.: A sketch-based interface for the design and analysis of simple vibratory mechanical systems. In *Proceedings of ASME International Design Engineering Technical Conferences* (2004).

[LaV07] LAVIOLA J.: Advances in mathematical sketching: Moving toward the paradigm's full potential. *IEEE Computer Graphics and Applications 27*, 1 (2007), 38–48.

[LLMZ08] LAVIOLA J., LEAL A., MILLER T., ZELEZNIK R.: Evaluation of techniques for visualizing mathematical expression recognition results. In *To appear in Graphics Interface 2008* (May 2008).

[LM95] LANDAY J. A., MYERS B. A.: Interactive sketching for the early stages of user interface design. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1995), ACM Press/Addison-Wesley Publishing Co., pp. 43–50.

[LMM*06] LABAHN G., MACLEAN S., MIRETTE M., RUTHERFORD I., TAUSKY D.: Mathbrush: An experimental pen-based math system. In *Challenges in Symbolic Computation Software* (2006), Decker W., Dewar M., Kaltofen E., Watt S., (Eds.), no. 06271 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[LZ04] LAVIOLA J., ZELEZNIK R.: Mathpad^2: A system for the creation and exploration of mathematical sketches. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 432–440. (Proceedings of SIGGRAPH 2004).

[ZHH96] ZELEZNIK R. C., HERNDON K. P., HUGHES J. F.: Sketch: an interface for sketching 3d scenes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 163–170.

[ZML07] ZELEZNIK R., MILLER T., LI C.: Designing UI techniques for handwritten mathematics. In *Proceedings of the 4th EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (SBIM 2007)* (Aug. 2007).