2008-04-08

Improving Spreadsheets for Complex Problems

Brian C. Whitmer
Brigham Young University - Provo

IMPROVING SPREADSHEETS FOR COMPLEX PROBLEMS

by

Brian Whitmer

A master's thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a master's thesis submitted by
Brian Whitmer

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

_____	_____
Date	Dan Olsen, Chair
_____	_____
Date	Robert Burton
_____	_____
Date	Kent Seamons
_____	_____
Date	Parris Egbert
	Graduate Coordinator

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Brian C. Whitmer in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

_____ _____

Date

Dan R. Olsen
Chair, Graduate Committee

Accepted for the Department

_____

Parris Egbert
Graduate Coordinator

Accepted for the College

_____

Thomas W. Sederberg
Associate Dean, College of
Physical and Mathematical Sciences

ABSTRACT

IMPROVING SPREADSHEETS FOR COMPLEX PROBLEMS

Brian C. Whitmer

Department of Computer Science

Master of Science

Spreadsheets are one of the most frequently used applications. They are used because they are easy to understand and values can be updated easily. However, many people try to use spreadsheets for problems beyond their intended scope and end up with errors and miscalculations. We present a new spreadsheet system which uses complex-values and equation code reuse to overcome the limitations of spreadsheets for complex problems. We also discuss the features necessary in order to make these enhancements useful and effective.

ACKNOWLEDGEMENTS

I would like to thank my wife Paula for her loving support, and my daughter Becca for her welcome distractions. I would never have finished this without them both.

TABLE OF CONTENTS

List of Figures ..... **viii**
Chapter 1 – Introduction **1**
Spreadsheet Limitations 2
ICE Sheets 6
Chapter 2 – Prior Work **9**
Error Prevention/Detection 9
Spreadsheet Enhancements 11
Matlab 13
Chapter 3 – Complex-Valued Cells **15**
Robust Referencing 15
Complex Value Creation 18
Complex Function Results 20
Viewing Large Complex Values 22
Chapter 4 – Templates **27**
Copy and Paste 28
Using Template Code 30
Linking Templates 33
Chapter 5 – Extensibility **35**
Extensible Functions 35
Extensible Formats 39
Chapter 6 – Conclusion **43**
References **45**

LIST OF FIGURES

Figure 1: Complex-valued cells **2**
Figure 2: Extensible complex-valued functions **4**
Figure 3: Extensible formats **5**
Figure 4: The WYSIWYT system **9**
Figure 5: "Template" spreadsheets using ViTSL **10**
Figure 6: The PrediCalc system **11**
Figure 7: User-Centered Functions **12**
Figure 8: Matlab **13**
Figure 9: Complex referencing **16**
Figure 10: Combining cells **19**
Figure 11: Matrix multiply in ICE Sheets **21**
Figure 12: Matrix multiply in Excel **21**
Figure 13: Tearing off cells **23**
Figure 14: Active/inactive sizes **24**
Figure 15: Active/inactive formats **25**
Figure 16: Data and graphs remain separate in Excel **25**
Figure 17: Template table **28**
Figure 18: Copy and paste can lead to errors **29**
Figure 19: Templates propagate their changes **31**
Figure 20: Data propagation in Excel **32**
Figure 21: The Java interface for functions in ICE Sheets **36**
Figure 22: Naïve Bayes display format **40**
Figure 23: Extensible formats for different displays **41**

CHAPTER 1 – INTRODUCTION

Spreadsheets are one of the most common end-user applications today. Everyone from accountants to baseball coaches takes advantage of the ease of entering data and performing calculations. The basic format of spreadsheets, a two-dimensional grid of scalar values, has not changed much since VisiCalc was created in 1979. Because spreadsheets are so easy to understand and develop, people still use them for simple calculations, but also take advantage of their ease of use for things as complex as stochastic simulation [9] and data warehousing [11]. The simple concept of a grid of numerical values and equations translates well in some situations, but hampers ease of development in many others. Research has shown that up to 90% of professional spreadsheets contain errors [8], and many of those errors come from trying to fit a complex problem into the limited expression available in conventional spreadsheets.

All the functionality available in spreadsheet programs is also available in more advanced environments such as Java, Python or Matlab. The reason end-users continue to prefer spreadsheets over such languages is that spreadsheets offer a visual representation of the data that is easy to understand and modify. Programming languages require more expertise and offer a less-intuitive presentation of results. For this reason users continue to pour their complex problems into spreadsheet systems, even though it often results in errors and misunderstandings.

Spreadsheet Limitations

We see three key limitations in conventional spreadsheets when dealing with complex problems. First, spreadsheet functions and computations must evaluate to a scalar result (some systems like Excel have array results, but these are limited and can be confusing). Second, spreadsheets rely heavily on copy and paste, which is a poor method of code reuse. Third, the limited set of scalar-based visual representations and computations prevents effective formulation of complex problems.

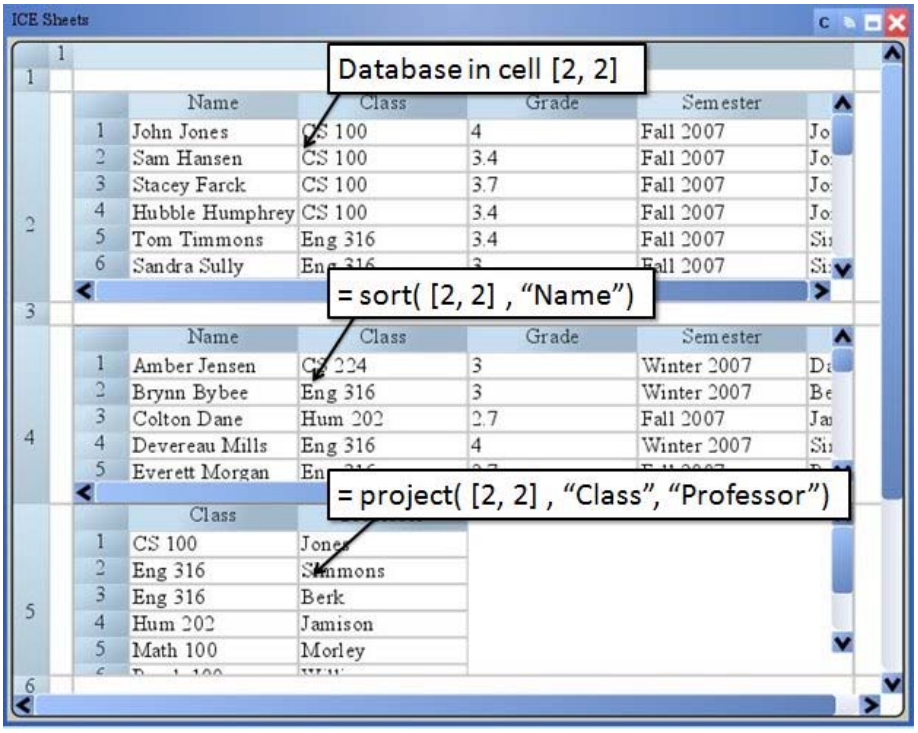


Figure 1: Complex-valued cells and function results are useful for tasks like database representation.

In spreadsheets all values and results must evaluate to scalars. This isn't a problem for adding or multiplying a few numbers, but is a factor in more advanced computations. For example, the product of two matrices is another matrix, but

spreadsheets don't natively allow for matrix results. Also, a large collection of values could function as a database. It would be useful to have functions that resulted in complex values such as select, join, pivot or sort to show summaries of this database. For example, Figure 1 shows a spreadsheet with a database in one cell, a version of that database sorted by name in another cell, and the list of class-professor pairs for that database in a third cell. Functions that return these types of results can't be written for spreadsheets because of the scalar-only restriction.

Spreadsheets also lack code (or equation) reuse and abstraction. A good way to highlight this limitation is to compare spreadsheets to programming languages. If a Java programmer is going to use a block of code repeatedly, he abstracts it out into a separate class or method. In spreadsheets when a user wants to reuse equations, he copies the region of values and pastes it to a new region in the sheet. This leaves him with duplicate copies of the same set of equations to manage instead of abstracted, reusable code like the Java programmer. In addition, values are scattered across a single grid instead of being modularized. This causes referencing problems as users copy and paste regions, because they will often miss cells necessary for the computation. For example, a set of equations may be based on constants defined somewhere else on the sheet. Using default cell referencing, a copied version of the equations will no longer point to the correct constants, and the new results will be incorrect. A more powerful solution would be to allow related values to be combined into a self-contained unit.

Spreadsheets have a small set of possible functions and visual representations, and almost all of these are for scalars only. No fixed set of functions and visualizations can service everyone's needs. Some spreadsheet systems do allow for extensible functions (Excel uses Visual Basic), but these systems can be confusing in their implementation and are still restricted to scalar-only results. As a result, most users rely on the provided set of equations and assume anything beyond that is infeasible. With an extensible system allowing complex results, most of the functionality of, for example, the WEKA machine learning suite [10] could be integrated into a spreadsheet system. Likewise it would be possible to write a function that would query the online BLAST database [2] for related DNA sequences, or to retrieve an up-to-date list of stock quotes using a web-request function (see Figure 2). Such complex-valued extensibility is not readily available in current spreadsheet applications.

Figure 2: Allowing for extensible complex-valued functions opens up new possible spreadsheet functions, such as a stock quote retrieval function.

Spreadsheets also lack extensibility for their visual representations, or formats. Current spreadsheets allow restricted conditional formatting and a small set of charts and

graphs. Formatting of a single cell is limited to scalar-based formats only. If instead formatting were opened up and could handle scalar *or* complex values, spreadsheet design would become much more flexible and understandable. Developers could design tree structures to represent a decision tree classifier, a pedigree view, or a binary search tree-sorted representation of data. They could write formats to cater specifically to large or small screens (Figure 3). It would be possible to implement something as complex as an interactive 3-D rendering of a data set or as simple as color-coded text. Traditional spreadsheets don't allow for these kinds of extensible formatting options. This severely limits the range of possible uses for spreadsheets and makes design and comprehension more difficult.

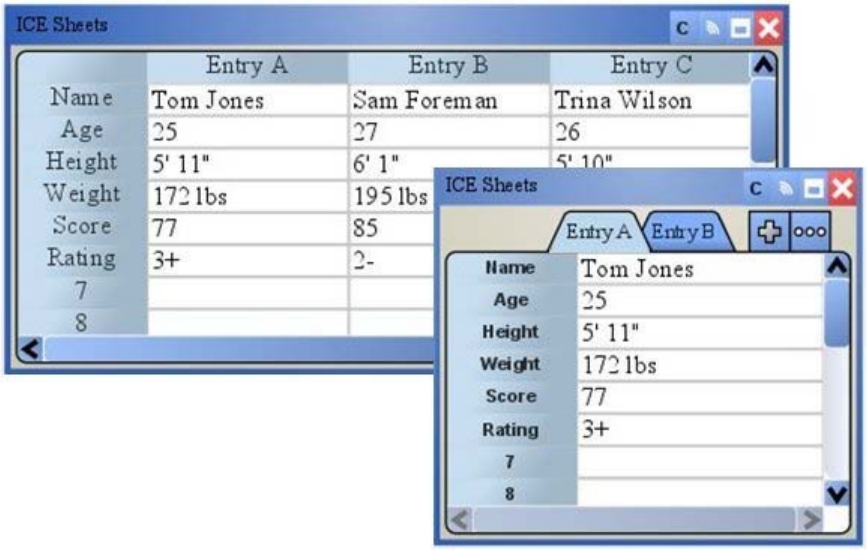


Figure 3: Extensible formats allow for many ways to view data. The front format could be a specialized format for smaller screens such as PDAs.

ICE Sheets

Our solution, ICE Sheets, overcomes these limitations by incorporating complex-valued cells and function results, the separation of data from equation code, and extensible functions and formatting. Together these enhancements remedy the problems just discussed.

First we augment the spreadsheet model by allowing cells to contain whole tables of values, either created or derived, in addition to single scalars. This nesting can go as many levels deep as is needed, opening up many new possibilities for representing data. We also allow complex-valued function results as a way to broaden the range of possible functions.

Second, we allow for code abstraction by separating equation code away from data values. As users create complex values, the properties and equations are pulled out into what we call a template. This template's code can be reused on different sets of values to generate multiple complex cells that are all linked to the same set of equations. In this way we replace the less-effective copy-and-paste paradigm with the programming concept of abstraction.

Finally, we introduce an extensible system for functions and formatting. We implement a plugin architecture that makes it easy to create and use new packages of functions and formats. By implementing this system on top of a set of simplified Java interfaces, we make it possible for developers to create more specialized function and format packages capable of handling or returning arbitrarily-complex values. We build

our formats on top of the XICE architecture, which we will discuss later, to simplify the creation of formats. Because these functions and formats can be based on scalar or complex values, they offer more expression than previous solutions.

These enhancements combined make it possible to more effectively express complex problems in spreadsheets. They will also allow new problems to be solved where previously the spreadsheet environment was too restrictive.

CHAPTER 2 – PRIOR WORK

Because of the prevalence of spreadsheet programs, much research has gone into improving the usage experience. Spreadsheet errors are common, and it is important that we find solutions that are capable of either finding problems or decreasing the likelihood of the problems occurring in the first place. Spreadsheet research generally can be separated into two broad categories: error prevention/detection, and program enhancements.

Error Prevention/Detection

Most spreadsheet research focuses on ways to detect or decrease errors while staying within the limits of traditional spreadsheets. What You See Is What You Test (WYSIWYT) [3] helps users find errors by working backwards. WYSIWYT lets users mark cells as either correct or incorrect and trace back to find the likely source of the error (see Figure 4). This backwards trace can help in discovering existing problems, but does nothing to remedy them. As such it doesn't really solve the problem of complex spreadsheet design.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Student**	**ID**	**HW1**	**HW2**	**HW3**	**HW Average**	**Quiz 1**	**Quiz 2**	**Quiz Average**	**Final**	**Extra Credit**	**Average**	**Grade**
2	Marc	1	95.0	95.0	95.0	95.0	95.0	95.0	95.0	95.0	0.0	95.0	A
3	Joel	2	85.0	85.0	85.0	85.0	85.0	85.0	85.0	85.0	5.0	85.0	B
4	Beth	3	75.0	75.0	75.0	75.0	75.0	75.0	75.0	75.0	5.0	75.0	C
5	Aiden	4	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	0.0	65.0	D
6	Emily	5	85.0	85.0	85.0	85.0	85.0	85.0	85.0	85.0	0.0	85.0	B
7	**Averages**		**81.0**	**81.0**	**81.0**	**81.0**	**81.0**	**81.0**	**81.0**	**81.0**	**2.0**	**81.0**	

Figure 4: The WYSIWYT [3] system uses checks and X-marks to determine testedness.

Type Inference [1] attempts to enforce a stronger typing for cell values. The system notifies the user when a value's type is different than expected. This idea offers some usefulness, but most spreadsheet values are only strings or numbers, so type enforcement won't catch many problems. It also fails to propose an easier method for developing complex spreadsheets.

Another approach to error prevention is spreadsheet modeling. The Visual Template Specification Language (ViTSL) [4] separates design into two distinct steps, equations and data. The goal is to reduce errors by generating equations without being distracted by values (see Figure 5). The notion of code templates is very useful, but designing abstract equations without concrete values works opposite of spreadsheet strengths. As a result, this system can hamper as much as it helps.

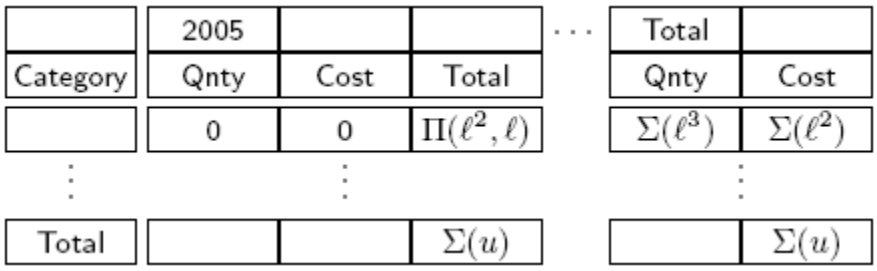


Figure 5: Designing "template" spreadsheets using ViTSL [4]

All of these debugging solutions fall short of our goal to make spreadsheet design more flexible and understandable in that they focus too specifically on traditional spreadsheet design.

Spreadsheet Enhancements

Some studies have also sought to enhance spreadsheets by adding new functionality. PrediCalc [6] makes cell evaluation omni-directional, allowing values to be derived from their related values in any ordering (see Figure 6). This idea is useful in its dynamic solution-finding, but breaks down when considering complex equations such as regions of cells as values because multiple solutions are possible for any single problem.

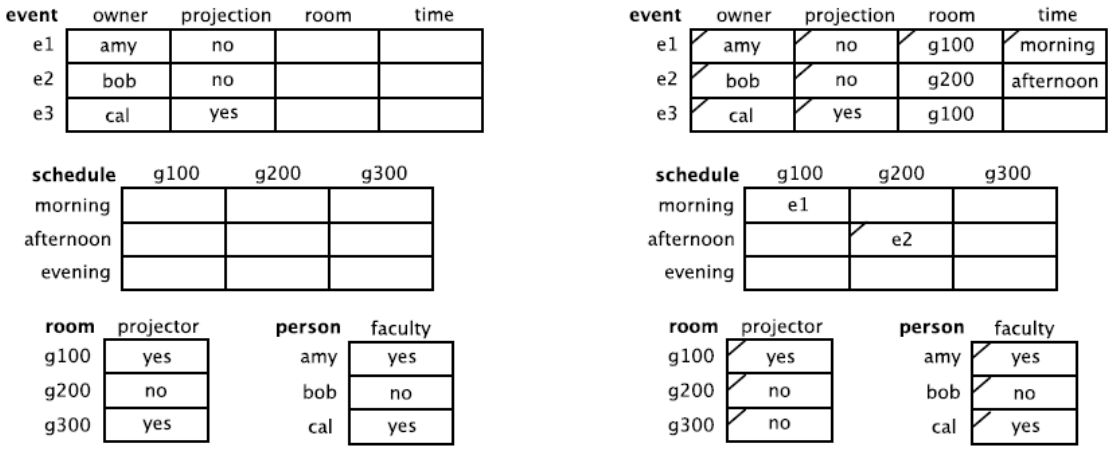


Figure 6: The PrediCalc [6] table of dependencies before and after entering scheduling events e1 and e2. Dependent values are updated automatically.

Query By Excel (QBX) [11] ties large spreadsheet tables to a relational database to let users more easily generate query-like calculations. The system allows for select, union and join operations, but is based on Excel's PivotTable structure, which uses dynamically-sized regions of cell. If the size of the PivotTable grows then it can overwrite other data on the same table. QBX also provides only a limited set of possible

summary-type complex results based on PivotTables. Computations like average and total are useful, but do not provide the more general solution we are after.

The User-Centered Functions system [5] simplifies custom function generation for spreadsheets. They system allows users to generate custom functions by extracting equations from a region of related cells into a separate sub-sheet (see Figure 7). This notion of modularity is a useful contribution which we also leverage in our solution. However, the system is limited to a single scalar final result. The authors point out the importance of complex-valued parameters, but only implement vector-valued inputs, not results.

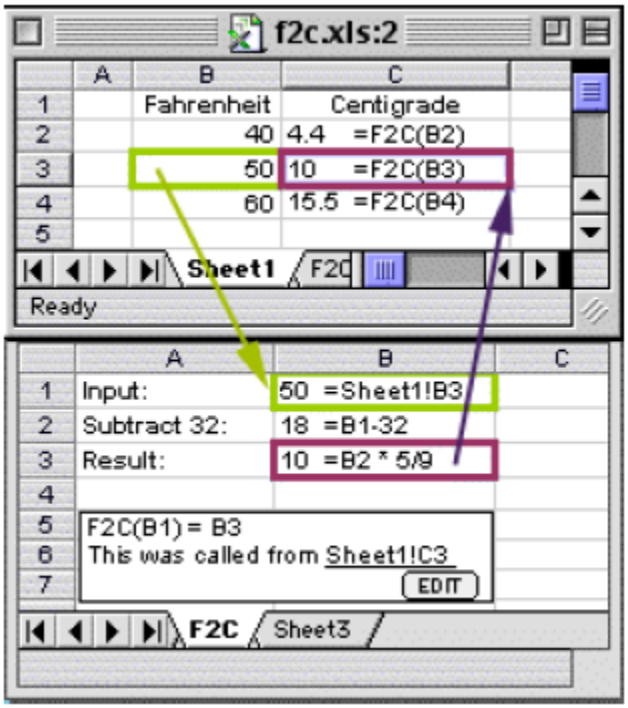


Figure 7: User-Centered Functions [5] let users create their own functions by pulling calculations out onto separate sheets.

All of these studies enhance development in small ways, but the more general problem of how conventional spreadsheets limit development has not been addressed in a complete solution like the one we propose.

Matlab

Complex spreadsheet tasks can also be computed using Matlab [7]. Matlab allows for scalar- or complex-valued calculations and results (see Figure 8). However, since Matlab is a command prompt environment that requires working knowledge of the system and syntax, it is much harder to learn and use than spreadsheets. As such, it does not resolve the usability concerns we wish to address. ICE Sheets combines a power comparable to Matlab with the user interface of spreadsheets.

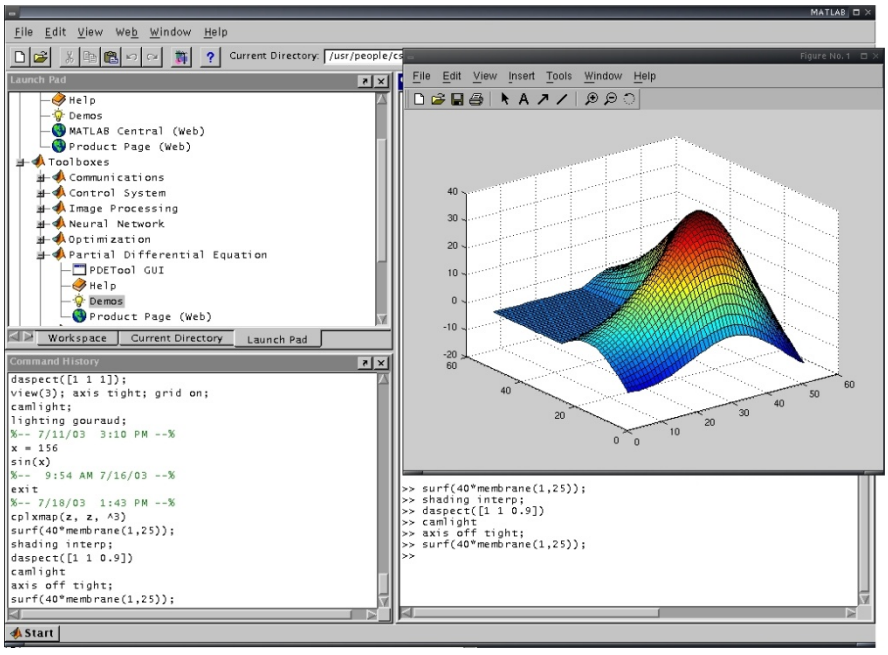


Figure 8: Matlab can display complex data representations, but is managed from the command prompt.

CHAPTER 3 – COMPLEX-VALUED CELLS

Our first enhancement is to augment our spreadsheet program with complex-valued cells. In ICE Sheets complex values are represented as a table of values contained within a cell. This enhancement is potentially too complex to be effective. We need to address how to effectively reference within complex cells, create complex cells, and interact with very large complex cells.

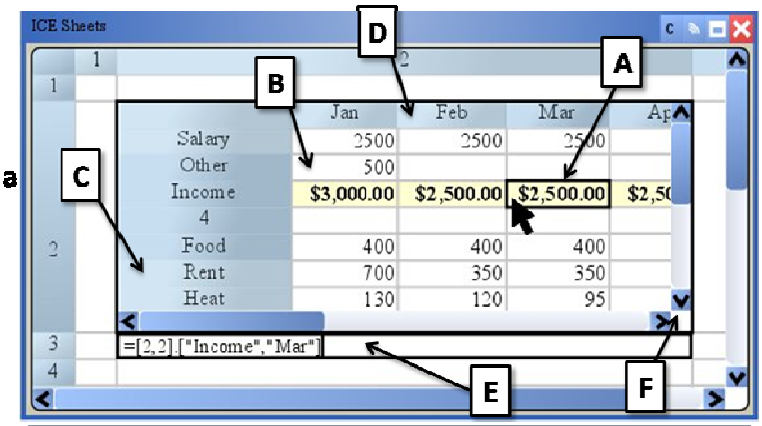
Robust Referencing

Spreadsheets need a uniform referencing mechanism. For example, consider the complex budget cell in Figure 9a. It would be useful to retrieve specific values, such as the total income for March. To allow for such inner-value referencing we introduce the dot (".") notation to access inner values, and the carat notation "^" which lets cells access values in their parent table. To access cell A from cell E, the user types the reference of the cell, a dot, and the reference within the cell: =[2,2].[3,3]. He could similarly select regions of cells using bracket notation {} (to select the region from B to A he would type =[2, 2].{2, 1: 3, 3}).

However, numerical indexing is unsatisfactory because it doesn't *explain* what is referenced. Spreadsheet systems have named regions, but this solution becomes unwieldy with too many or overlapping regions. We instead use named rows and columns. A cell is referenced by its row and column numbers or names, as in Figure 9b. We allow some special referencing as well, letting users leave values blank or use the star ("*") operator. If the row or column index is left blank then it is assumed to be the value

1 (cell B is referenced as [“Other”] instead of [“Other”, 1]), and a star value selects all values in the given row or column. Named references make it clear what an expression is pointing to, reducing ambiguity and easing spreadsheet development

a



b

Region	Reference String (from cell E)
Cell A	=[2, 2].[“Income”, “Mar”]
Cell B	=[2, 2]. [“Other”]
Region B to A	=[2, 2]. {“Other” : “Income”, “Mar”}
Row C	=[2, 2]. [“Rent”, *]
Column D	=[2, 2]. [*, “Feb”]
Cell F	=[2, 2]

Figure 9: Even complex cells can be understandably referenced using their row and column names.

By using named row and column references we allow spreadsheets to easily model other data structures. For example, the expression [*, “Feb”] selects the cells in column D of Figure 9a. The result is a table with named rows and a single column, which is a record for the month of February. Values in the record can be accessed like fields ([“Food”], [“Rent”], etc.). The record data structure is useful in many contexts. A large cell can also be a database, where the column names serve as a schema. The user can sort a database cell by column name, project only specific columns, or select rows

that match some criteria (Figure 1). These data structures provide simple but effective models for organizing data.

The star notation is especially significant for complex-valued cells, as it is no longer necessary to know exactly how many cells are in the region. If a region were defined in Excel as A1:A200, then inserting a cell at row 10 would update the reference to be A1:A201, but adding a value to the end of the region would *not* update the reference and the new value would be ignored. Traditional spreadsheets allow for referencing an entire row or column, but when all the data is on a single grid, columns and rows often contain more than one region of data (see Figure 10a, where the columns run over two sets of data). Only when data sets can be separated out into distinct complex cells does whole-row and whole-column referencing become useful. For example, in Figure 10c the user could easily find the average GPA for all students by typing =average([*, "GPA"]). This approach would not work in Figure 10a, since the region would have to be explicitly defined.

Adding named rows and columns also makes cell referencing more robust to changes. In traditional spreadsheets, if the user inserts or deletes a row or column then cell references have to be updated in order to stay accurate. This approach, though inconvenient, works because of the limited size of conventional spreadsheets. Once spreadsheets can contain any-sized complex values, the number of possible references to update grows very quickly. If named references are used instead of numerical references then insertions, re-orderings and deletions will have no effect on cell references (the

reference [“Income”, “Mar”] is not affected by row and column index changes). Referencing is robust to changes in the spreadsheet structure.

In traditional spreadsheets it is always clear which cell the user is selecting because all cells are separate. Once we allow for complex-valued cells, ambiguity arises. If a user clicks on a matrix, did they mean to select the entire matrix, or a value in the matrix? In Figure 9a is the cursor selecting the entire budget table =[2,2], or just the cell =[2,2].[“Income”,”Mar”]? For ICE Sheets we assume selection goes as “deep” as it can, and in the above case would select the cell =[2,2].[“Income”,”Mar”]. If the user wanted to select the whole table she could click outside the grid of values but still inside the table cell itself (point F in Figure 9a). By always providing a region of the format that is not part of any inner cell, we make either type of selection possible.

Complex Value Creation

To let users create a complex cell we add a “Create Table” option. This option creates a new complex cell and shows a table view of that cell that can then be edited. However, since complex values are essentially a collection of inner values, the obvious question is “how many values do I need?” If the complex value is a database, for example, it may not be clear at first how many entries are needed. Our answer is to make complex value sizes dynamic, just like spreadsheet tables. By viewing complex values as a nested table, users can scroll down or right to add as many additional values as they need, eliminating the size problem.

However, solution formulation often occurs during development, not before. Many users will enter a region of values, and *then* realize those values should be combined into a single complex value. Our system implements this functionality with the "Create As Table" option. Figure 10 shows this for two separate regions of data, one above the other. The regions are combined into complex values which are placed in the top-left cell of the selected region.

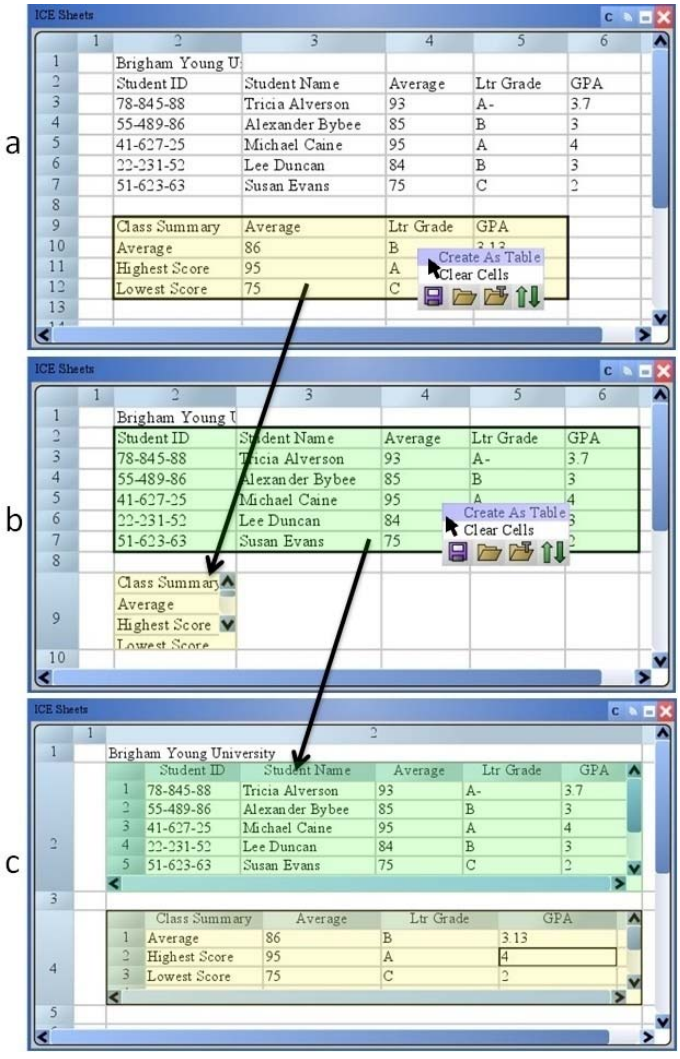


Figure 10: Users can select regions of cells and combine them into independent complex cells.

The modularization of complex values has the added benefit of separating regions of values away from one another. In traditional spreadsheets, inserting into one region of values affects other regions, as does resizing or moving rows and columns. In Figure 10a, the "Student Name" column needs to be very wide, but this forces the "Average" column below to be much wider than necessary, taking up precious screen space. Once regions are combined into complex cells (Figure 10c), the "Student Name" column in the first complex cell can be sized without resizing the other complex cell.

Complex Function Results

Conventional spreadsheets don't offer complex-valued results. Some array-type results are possible, but these are still spread across regions of cells. By allowing more native support for complex results, ICE Sheets makes complex computations more feasible and usable. For example, since spreadsheets can now return tables of values, it is easy to create functions that return standard 2-D geometry matrices (scale, rotate, translate) as in Figure 11. This takes half as many equations as the same set of calculations in Excel (Figure 12). Functions can also be written for database-like functionality including select, project, sort and join statements as in Figure 1. Likewise, functions can be written to take in a set of data instances and return an array of coefficients for a linear perceptron or a least squares approximation. Functions can even return the list of books by a given author on Amazon.com, or the list of web sites matching a given query on Google. Many new possibilities open up when functions can return complex results.

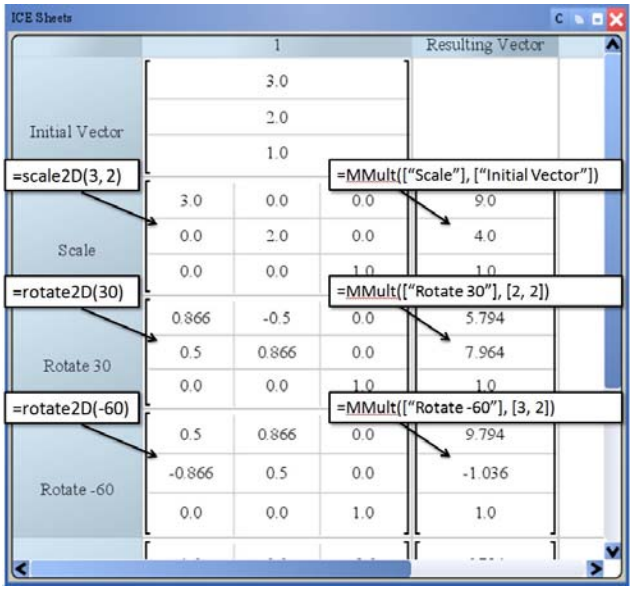


Figure 11: 2-D geometry matrices are a good example of function set that returns complex results.

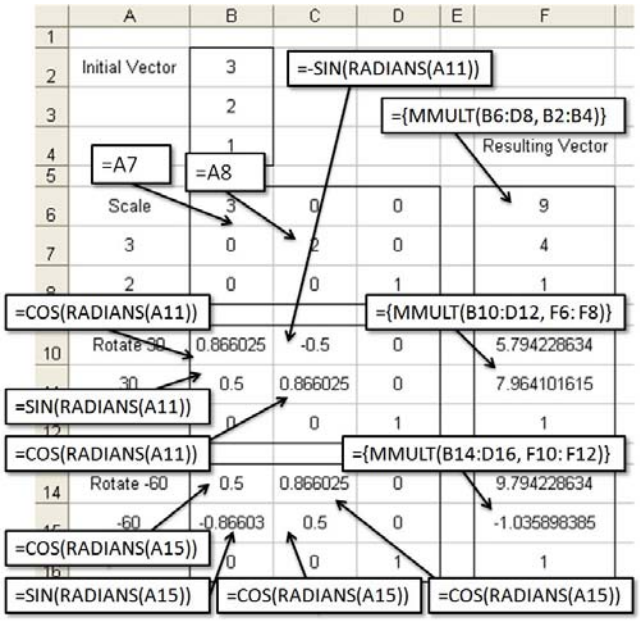


Figure 12: The example from Figure 11 written in Excel takes twice as many equations, some special key combinations, and is not as clear.

Viewing Large Complex Values

A cell can contain any number of possible values, but the user cannot understandably interact with large numbers of values when they are contained in a small spreadsheet cell. We introduce the notion of "tearing off" cells, where a new window pops up containing a larger view of the cell's data. This new window is connected to the same model, but can be changed to whatever size is convenient for the user. For example, a table could contain a complex value that served as a database of university students. The entire database could be held in one cell, and cells below could be used to write simple functional queries with results of only a few columns (say, the average GPA per semester, or the list of all classes per semester as in Figure 13). It would be a waste of space to expand the entire column since the queries have only a few columns, but the actual database is too large for the column width. Instead of widening all the cells, the user could tear off the database cell and interact with it in a separate window (Figure 13). This notion of "tearing off" solves the problem of dealing with exceptionally large complex values by letting the user pull out key items to view in detail.

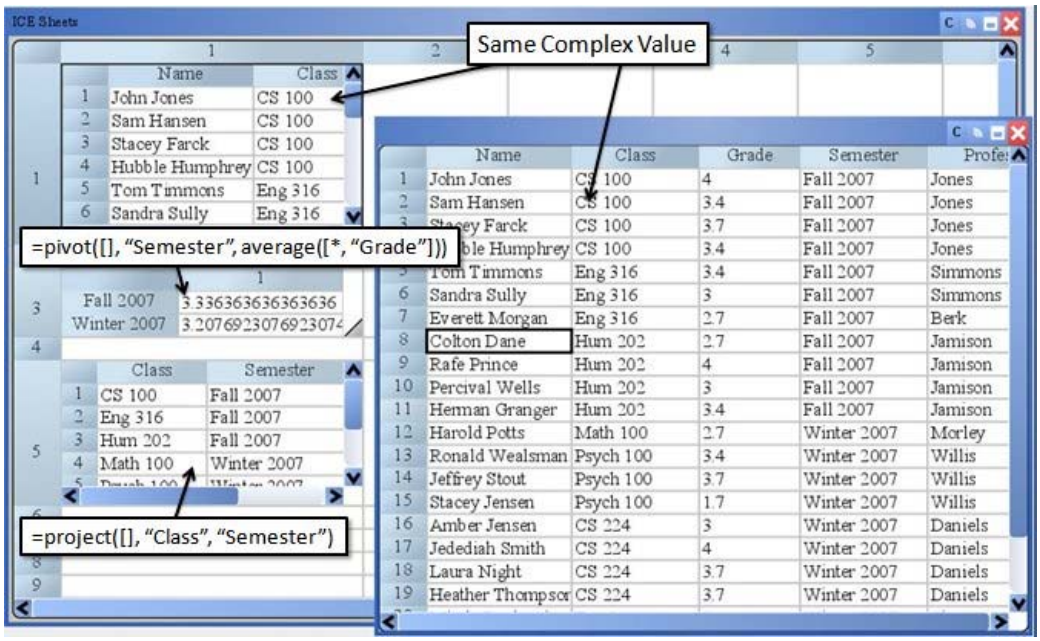


Figure 13: Complex cells can be "torn off", allowing a larger view of the same complex data.

However, tearing off cells lets the user expand only one cell at a time. It does not provide an easy way to rapidly expand multiple cells. In Figure 14a the user has a series of matrix multiplications. He wants to review the cells one at a time, but there is not enough room to show them all at once. It would be inconvenient to tear off all the cells, to keep resizing rows and columns to view the cells one at a time, or to enlarge all the cells and keep scrolling up and down within the table. Instead we introduce active and inactive sizes for rows and columns. When the user clicks a cell, its row and column assume their active size (see Figure 14a). When he clicks another cell the row and column resume their inactive sizes and the new active cell expands its row and column (Figure 14b). When a row or column is resized, it updates its active size if it contains the active cell, otherwise it updates its inactive size. For consistency, the inactive size must always be equal to or less than the active size.

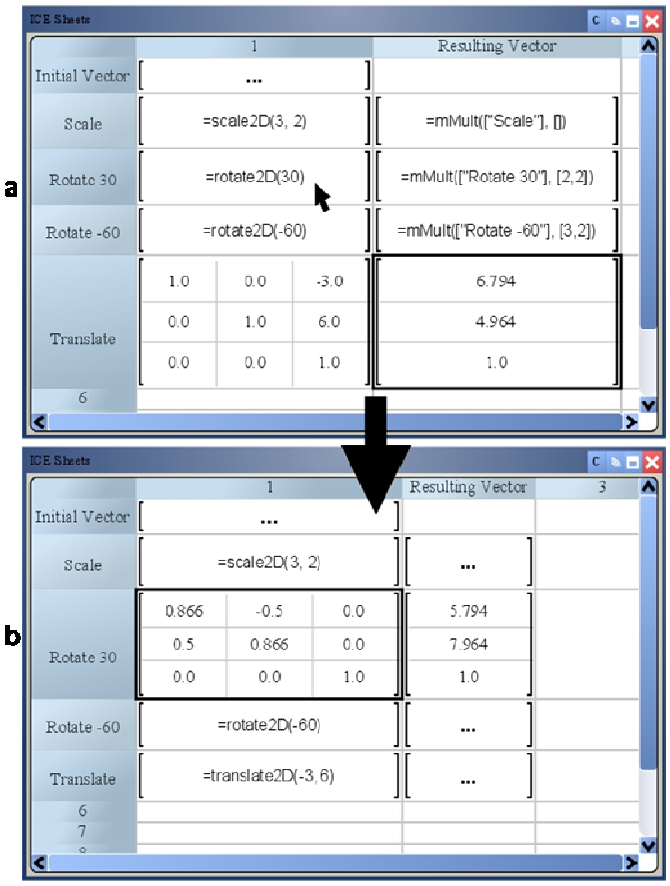


Figure 14: Active/inactive sizes. The "Translate" row is active, while others are inactive. The active row and column have enough size to be interactive.

In addition to active and inactive sizes, each cell can be assigned an active and inactive format. This lets the cell display usable information whether it is small or large, active or inactive. For example, a series of point plots could show a line graph when inactive, but show the table of values when active (Figure 15). This clears up screen space taken in traditional spreadsheets, which keep visible both the data values and the graph for each data set (Figure 16). The pairing of active and inactive formats lets the user define for each cell a summary type of functionality *and* an interactive functionality, providing a more effective use of screen space.

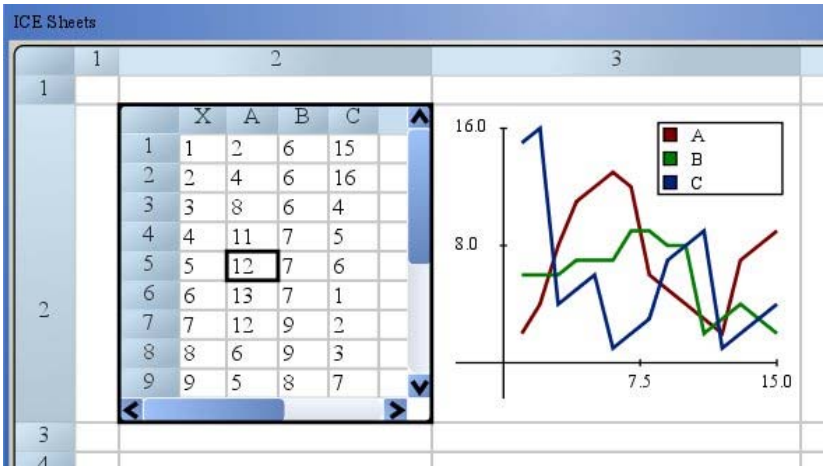


Figure 15: These two complex values contain similar amounts of data. The left cell is active and can be edited, while the right cell is inactive and displays a summary of its contents.

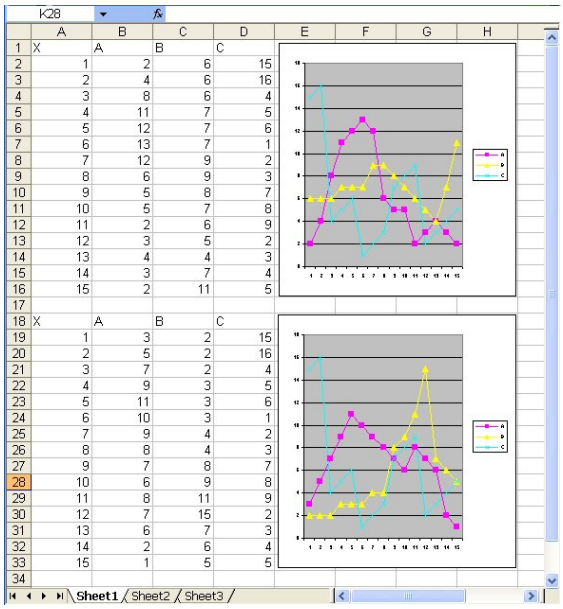


Figure 16: In traditional spreadsheets, data and graphs must remain separate. This takes up twice as much space as ICE Sheets.

Formats themselves also can easily adapt to different sizes and situations. For example, when the matrix format is sized too small to show its contents, it instead shows its underlying equation (as in Figure 14) or a "…" if even smaller. Such adaptive formats add to the overall flexibility of our solution.

CHAPTER 4 – TEMPLATES

In ICE Sheets we separate data from equation code. Every cell has a value (scalar or complex) and a table of important properties. This property table stores the cell's equation, its formatting parameters (font, color, etc.) and any additional parameters used by the cell. Having property tables makes it possible for a cell to reference the code of another cell by linking to the other cell's property table, while keeping its own concrete values. For example, in Figure 17 the two lower sheets are linked to the first sheet's property table, and are all using the same equation, A + B. If we update that equation in the first sheet (say, to A + 2B), the change will be propagated immediately to the lower two sheets as well. This kind of linking introduces the notion of code reuse, which we call a "template." We replace the prior notion of copy and paste with a more robust template system. However, we also have to address issues with the creation and use of template code.

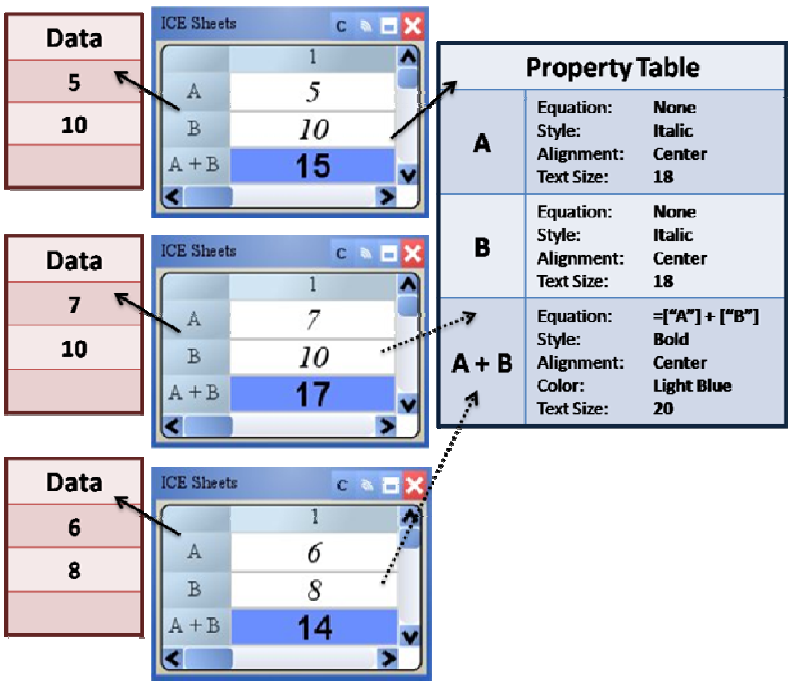


Figure 17: In this example, the lower two sheets are linked to the first sheet's property table, allowing for reuse of the "code" in that sheet.

Copy and Paste

Code reuse in spreadsheets is traditionally accomplished by copy and paste. To reuse an equation or region of values, the user copies it and pastes it to a new location. For template code in ICE Sheets to be effective, it needs to be as usable and understandable as copy and paste. We address this issue with the property tables already mentioned. To link a cell to a template, the user right-clicks on the cell and selects "Create Table". The user can create the table "From Template" which lets them select a file to be the template, or "From Cell" which lets them select another cell in the sheet to be the template. Any change to the original template will be propagated to all cells that use that template (see Figure 1 7). This process is comparable in ease to copying and pasting, but has the added abstraction and modularity benefits of template code.

Using templates offers a more powerful method of code reuse. Traditional copy and paste requires an intimate understanding of spreadsheet syntax (for example, the difference between “A4” and “A4”). It is also likely that the user will fail to select an important cell, causing the pasted region to work incorrectly. For example, a research team may write a spreadsheet with a sequence of computations and function calls in separate cells (Figure 18). They would like to view this equation with different initial values, so they copy the region and paste it into the two rows below. However, the original sequence relied on an unnoticed value in C2, and the pasted sequences do not compute properly. Finding and correcting these types of errors takes unnecessary time and effort. In ICE Sheets the problem is avoided by using complex cells. Instead of spreading values all over a large grid, related values are held in a single complex cell. The user can reuse this cell as a template without having to worry about missing necessary variables.

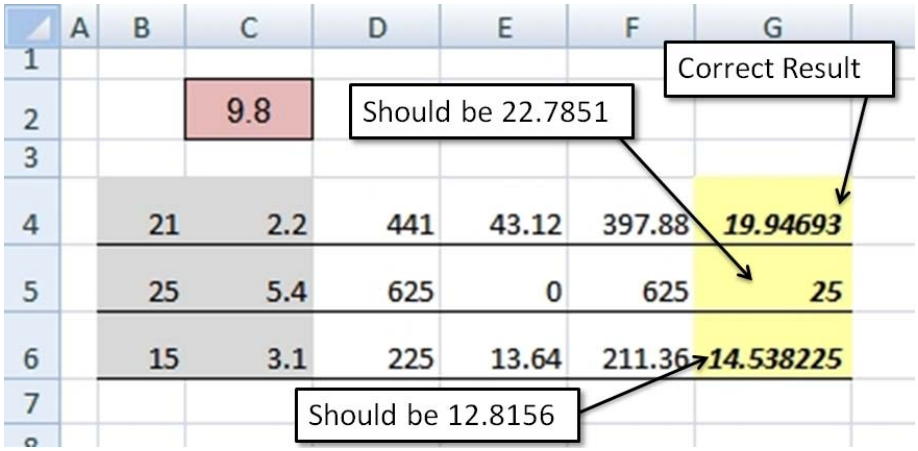


Figure 18: Copy and paste can easily lead to errors. The lower two rows are copies of the first row, but pasting led to an incorrect reference.

Problems also exist when duplicating whole files. To reuse a spreadsheet file, the user makes a copy of the file and changes the necessary values. The problem with file copying is that a change to the original file will not be propagated to the sheets based on that file. Instead the user has to manually find and correct every pasted copy of the errored equations. A good example of this is a set of financial reports with a shared miscalculation. In conventional spreadsheets, the problem would have to be corrected in every instance of the report instead of just one. ICE Sheets prevents this concern through the template system. Updating the template file will propagate the change to all sheets based on that file.

Using Template Code

Because all complex cells have property tables, any complex cell is a potential template. This is in contrast to previous research [4], where the user must intentionally create a template by specifying an abstract set of equations which is later populated with discrete values. The prior step of creating abstract code is no longer necessary since property tables let the user define the template in the same phase as the concrete data. There is still the additional concern of knowing what concrete values to use when creating a complex cell based on a template. We initialize the new complex cell with the same concrete values as the template, and then let the user override those values as needed. This solves the problem of understanding cells created from templates.

Template code also makes it possible to augment all instances of a template at once. Figure 19 shows a mortgage payment calculation template. The template is being

used to compare potential mortgages side by side. Originally this table only had four values: Mortgage, Years, Rate and Monthly Payment. Later on the user wanted to add a fifth row, Total Interest, as another means of comparison. To do this she entered a new equation in the next row of the template and the rest of the instances instantly added this equation as well. This kind of after-the-fact template updating is a very compelling feature. To accomplish the same thing in traditional spreadsheets would take multiple copy and paste commands and possibly a row insert as well (see Figure 20).

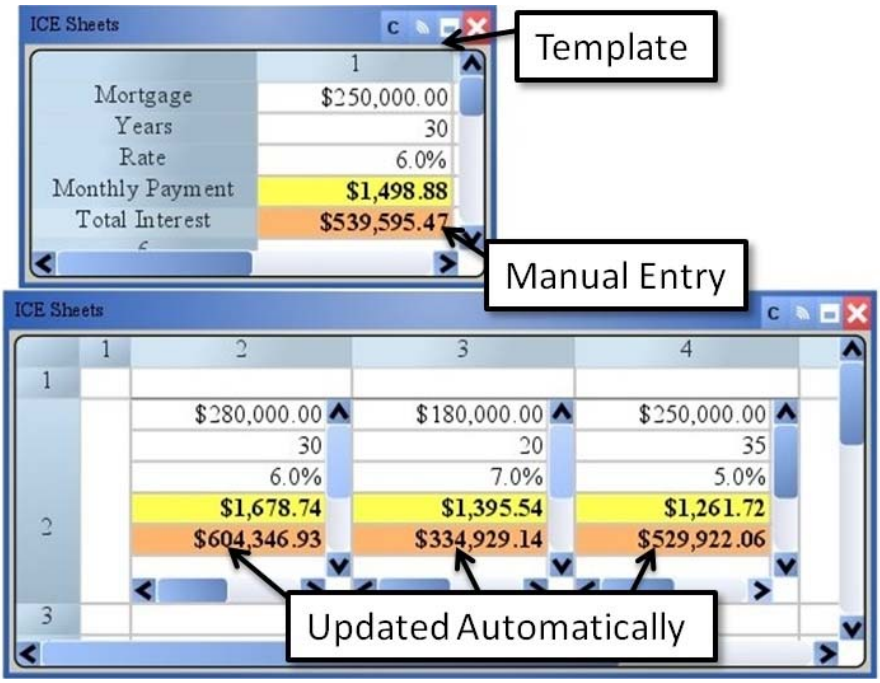


Figure 19: Templates propagate their changes to all cells based on the template, so users can add new data to all linked cells at once.

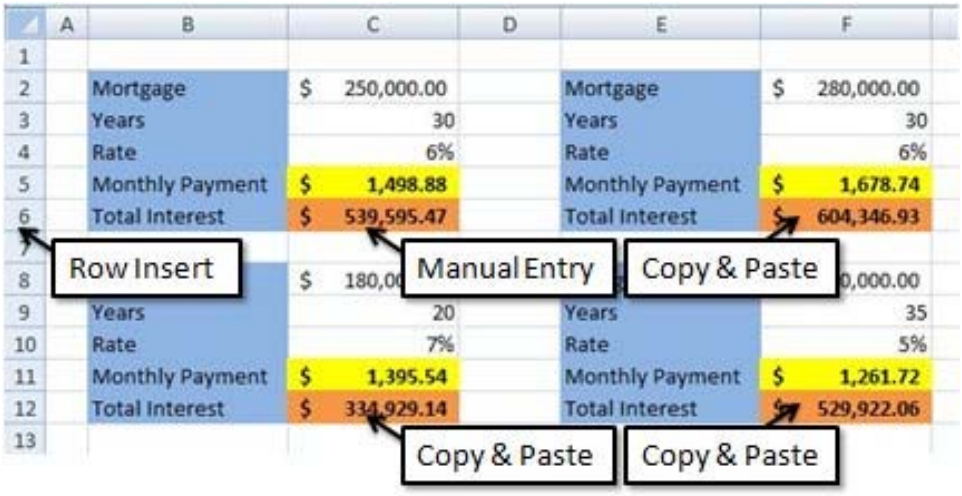


Figure 20: In traditional spreadsheets it is much more difficult to add or update similar equation code in separate cells.

When creating cells based on a template, there may be times when it is useful to propagate changes to concrete values from the template to all cells using that template. For example, a mortgage company may keep a standard interest rate across all calculations. Normally numbers are considered concrete values, and so a change would not be propagated. However, the user can simply make the constant into an equation (instead of the value ".046", the user could type "=.046") which will then allow a change to be propagated to any cells using that template (by changing the equation in the template to say, "=.049").

Users can create powerful and understandable templates by making complex cells with concrete values and computations based on those values. When a complex cell is used as a template, the user can edit the concrete values in the new cell and see the computational results without rewriting or copying equations. Similar code reuse is

discussed in the User-Centered Functions research [5], and can be a useful and simple way for end-users to create reusable functions for their spreadsheets.

Linking Templates

The linking of spreadsheet templates is simple and meaningful, but it raises synchronization concerns. For example, what happens if a user deletes the template that is the basis for another spreadsheet? What happens if a user emails their spreadsheet to someone else, but not the associated template file? Does the cell lose all of its equation and formatting information, or is there some recovery mechanism? Our solution is to allow templates to be *any* complex value, even within the same table file. If there are concerns about losing connectivity to the original templates, then those templates can be housed within the same spreadsheet file. This makes it harder to use templates, but at least addresses the synchronization problem somewhat. The broader question of how to handle broken template links in general we leave for future research.

CHAPTER 5 – EXTENSIBILITY

In ICE Sheets we implement an easy extensibility of both functions and formats. With the added flexibility introduced by complex-valued cells, many new uses will be found for spreadsheet applications. As users find increasingly specialized uses for spreadsheets, the set of included functions will become less complete. No standard set of functions and formats can satisfy all the possible uses of spreadsheet programs, so some mechanism needs to be in place to allow for development of custom functions and formats. Function (but not format) design has been a common piece of spreadsheet programs for many years, but allowing for complex-valued cells brings up new issues that need to be considered.

Extensible Functions

Extensible functions in ICE Sheets are capable of returning complex values. This is a very useful feature in many cases. For example, matrix multiply and database select and join functions all return complex results that would be difficult to retrieve using functions in traditional spreadsheets. This added functionality has the potential to make function design too complicated to be useful. In ICE Sheets we partly address the problem by basing all complex values on a table structure. Tables with named rows and columns are capable of modeling many powerful and useful types of structures, while still being generally understandable.

There is still the more general concern of creating functions with complex results, however. We use a Java interface to help developers write functions for ICE Sheets (see

Figure 21). Most of the interface's methods (*getHelp*, *parameterName*, *parameterHelp*) are for end-user help, giving argument explanations to the user. In general, all the developer needs to specify is how many arguments the function takes in *handlesNParameters*, the type of each argument (String, Integer, Boolean, Table, Any, etc.) in *parameterType*, and a *compute* method which receives the specified arguments and returns some Object as a result. This is comparable in simplicity to Excel's Visual Basic for Applications (VBA) solution.

```
 * <p>Interface defining functions for use in ICESheets.
public interface SheetFunction extends SheetExtension
{
     * The number of parameters the function accepts.
    public boolean handlesNParameters(int nParameters);

     * General help on what the function does.
    public String getHelp();

     * Specific name for the parameter at the given index.
    public String parameterName(int idx);

     * Help explanation for what the parameter at the
    public String parameterHelp(int idx);

     * Required type for the parameter at the given index.
    public ExpressionType parameterType(int idx);

     * Performs the actual computation and returns a result
    public Object compute(Parameter[] parameters, SheetTable table,
            CellIndex currentCell);

}
```

Figure 21: The Java interface used to create functions for ICE Sheets.

Functions are written as Java classes, and can be added to ICE Sheets by clicking the "Manage Extensions" menu option. To add a function, the developer adds their package name to the list of extension packages. Each package in the list must contain an "ICESheet.ext" text file with a list of extension class names in the package. The functions are loaded at runtime.

This Java interface brings ICE Sheets functions to the same level as Excel, but additional functionality is possible because of complex-valued cells. For example, one common need will be to perform a simple calculation on all values in a table (say, the absolute value of an array of numbers, or the upper case version of an array of strings) or on a set of equally-sized complex values (the squared difference between value pairs in two tables). We implement an abstract class called *SheetFunctionScalar* for this very purpose. The *SheetFunctionScalar* class overwrites the *compute* method to handle both scalar and complex arguments. The developer writes a new method, *computeValue*, which computes a result for scalars only. The *SheetFunctionScalar compute* method takes the actual arguments and if any of them are complex values, it breaks them up into separate scalar values. Then the *computeValue* method is called once for each possible value combination.

For instance, a simple function called “difference” could have a *compute* method that takes two scalars and returns the difference. If this function extended the *SheetFunctionScalar* class then the user would simply change the *compute* method to *computeValue*. If the modified function received two scalars as inputs, then it would just call *computeValue* and return as normal. However, if the function received a table and a scalar, then the *SheetFunctionScalar compute* method would take all the values in the table, call *computeValue* for each scalar pair, and return the table of results. The same thing would happen for two tables of values. The actual change to the function was trivial, but it made the function much more powerful. Similar functions could trivially be

written to return say, the absolute value of an array of numbers, the sum squared distance between arrays, etc.

A second commonly-used functionality is to take a list of arbitrary inputs and perform some computation based on the entire list of inputs. This is different than the previous example, since it agglomerates all arguments into a single list instead of keeping arguments separate. For example, an "average" function could take a collection of numbers and arrays of numbers, in any order. This function would be difficult to write because the argument types are not specific. We implement an abstract class called *SheetFunctionArray* to accomplish the task. The *SheetFunctionArray* class has a special *compute* method which takes in the list of scalar and complex arguments and passes the list of all extracted scalars to an abstract *computeValue* method.

If we had an "average" function that took a list of scalars and returned their mean, we could improve the function's power by extending the *SheetFunctionArray* class and renaming the *compute* method to *computeValue*. This would let the function take in scalars *and* arrays of scalars. The *compute* method would extract out the list of all scalars and pass this list to the *computeValue* method, which would return the average. Enhancing functions is this way is simple, but powerful.

Function extensions like SheetFunctionArray and SheetFunctionScalar are not difficult to write, and make possible all sorts of powerful capabilities for spreadsheet functions. Most importantly, they make writing complex-valued functions the same as writing scalar-valued functions.

An additional example of the benefit of easily-extensible functions comes when using spreadsheet programs on a computationally-weak computer. In this case, functions could be designed to gather the arguments and pass overly complicated problems across a network to some more powerful computer. This lets even a handheld device leverage very powerful and complicated systems of computations. Such a framework could also be used to retrieve information such as stock quotes or to query large online repositories such as the BLAST database [2].

Extensible Formats

In addition to extensible functions, ICE Sheets also allows for development of custom visual representations of data, called formats. Formats can be interactive or static, and can be designed to work on scalar values (a number slider, a currency formatter, etc.) or on complex values (a collapsible tree view, a bar graph, a matrix view, etc.). Multiple formats may link to the same cell, and formats are capable of reading and assigning or updating values to their associated cell. Previous conditional formatting solutions provide a limited set of display options such as different colored backgrounds or icons based on threshold values. These can be useful in some settings, but not all. For example, a research team could write a Naïve Bayes classifier that takes data instances as input and returns a complex value of the resulting probability tables. In this case it would be useful to design a format that clearly displays the contents of those probability tables (see Figure 22). This kind of specialized format is not possible in traditional spreadsheets because there is no notion of extensibility. Instead of providing only a few

formatting options, we allow for an unlimited, unrestricted set of display formats through extensibility.

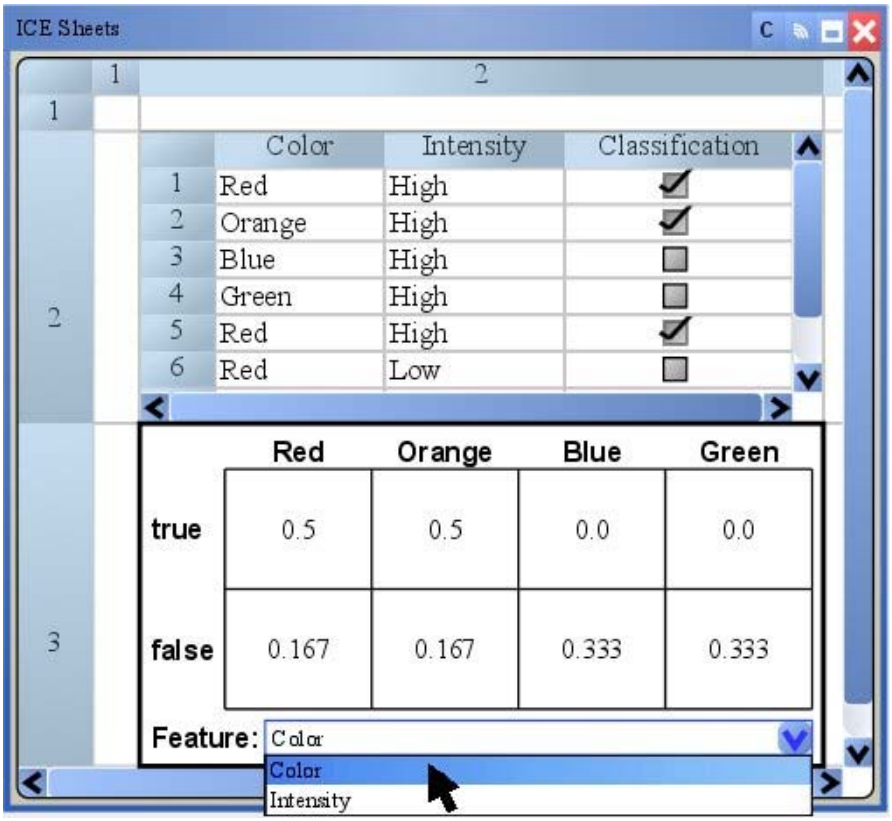


Figure 22: Many new display possibilities open up when formats are extensible. The lower cell shows Naïve Bayes probabilities for data in the upper cell.

Another compelling format example is in Figure 23. The user has a spreadsheet to track their finances, and he wants to edit this spreadsheet on a widescreen desktop and also on a smaller handheld device. The spreadsheet has one row and three columns, and the three cells hold large sub-sheets that show a summary, a log and a budget, respectively. On the desktop screen there is enough room to view and edit the sub-sheets side by side, but the smaller device does not have as much screen space. We could write a special tabbed format as shown that puts each column in a different tab. This tabbed

view is very similar to the view seen in traditional spreadsheets, and is more effective on small screens. By allowing extensible formats, sheet layout can now be more flexible to screen size.

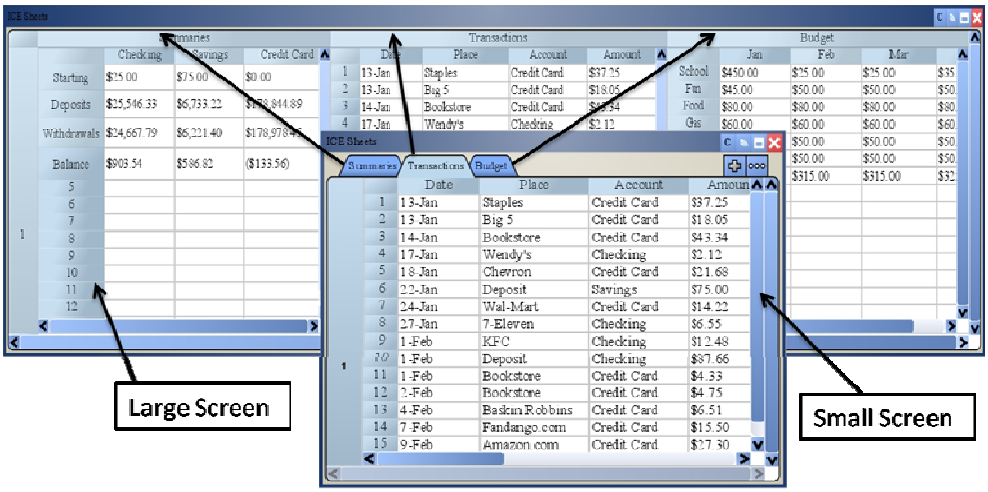


Figure 23: Extensible formats allow users to interact with data effectively on different devices.

The main concern when developing formats is that interactive software design can be complicated and non-intuitive. Many developers would not feel comfortable designing their own formats because they don't feel capable of designing interactive interfaces. We solve this problem by implementing our solution in the XICE architecture.

XICE (eXtended Interactive Computing Everywhere) is a development architecture for creating interactive components. It is easier to use than traditional architectures because it simplifies both the creation of widgets and handling of events. Widgets no longer have to implement a "paint" method to paint themselves to the screen. Instead they organize a scene graph, or tree of sub-widgets, in a layout which is

automatically painted when needed. Additionally, event handling is simplified by a default set of actions that pass events downward through this presentation tree until some object in the tree processes the event. This greatly simplifies development, in our case making it easier to develop customized spreadsheet display formats.

CHAPTER 6 – CONCLUSION

Our ICE Sheets system adds to traditional spreadsheets the concept of complex-valued cells. By including complex values, template code and extensibility we allow for easier representation and manipulation of complex problems within the spreadsheet system. This has the potential to reduce errors during development and opens up many new uses for spreadsheet programs.

When compared to traditional spreadsheet programs, ICE Sheets accomplishes the same tasks with fewer cell entries (Figure 11), more understandable references (Figure 9), and greater flexibility of expression. Users can also replace fragile copy and paste with abstract code reuse through the use of complex cell templates (Figure 17). Developers can write much more powerful functions and display formats than were previously possible, and can focus these solutions toward specific problem sets. This lets users more effectively solve their complex problems by leveraging solutions custom tailored to those problems (Figure 22). In all, these enhancements make complex spreadsheet development a more powerful and understandable experience.

REFERENCES

1.Abraham, R. and Erwig, M. (2006). Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming '06*, 73-94.

2.BLAST (2007). BLAST: Basic Local Alignment Search Tool. http://www.ncbi.nlm.nih.gov/BLAST/.

3.Carver, J., Fisher, M., and Rothermel, G. (2006). An empirical evaluation of a testing and debugging methodology for Excel. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering '06*, 278-287.

4.Erwig, M., Abraham, R., Cooperstein, I., and Kollmansberger, S. (2005). Automatic generation and maintenance of correct spreadsheets. In *Proceedings of the 27th international Conference on Software Engineering* '05, 136-145.

5.Jones, S. P., Blackwell, A., and Burnett, M. (2003). A user-centred approach to functions in excel. In *Proceedings of the Eighth ACM SIGPLAN international Conference on Functional Programming '03*, 165-176.

6.Kassoff, M., Zen, L., Garg, A., and Genesereth, M. (2005). PrediCalc: a logical spreadsheet management system. In *Proceedings of the 31st international Conference on Very Large Data Base* '05, 1274-1250.

7.Matlab (2007). MATLAB – The Language of Technical Computing. http://www.mathworks.com/products/matlab/.

8.Panko, R. R. (2005). What We Know About Spreadsheet Errors. http://panko.cba.hawaii.edu/ssr/Mypapers/whatknow.htm.

9.Seila, A. F. (2006). Spreadsheet simulation. In *Proceedings of the 37th Conference on Winter Simulation '06*, 11-18.

10.Weka (2007). Weka 3 – Data Mining with Open Source Machine Learning Software in Java. http://www.cs.waikato.ac.nz/ml/weka/

11.Witkowski, A., Bellamkonda, S., Bozkaya, T., Naimat, A., Sheng, L., Subramanian, S., and Waingold, A. (2005). Query by Excel. In *Proceedings of the 31st international Conference on Very Large Data Bases* '05, 1204-1215.