DESIGN OF A SPREADSHEET INTERFACE FOR S

Richard M. Heiberger and Magnus Mengelbier, Temple University
Richard M. Heiberger, Temple University, Department of Statistics, Philadelphia, PA

Key Words: Splus

Abstract

We have designed and constructed an interactive spreadsheet interface to S (Becker, Chambers, and Wilks 1988) that maintains the complete power and generality of the S language. At user level, the interface behaves like the popular spreadsheet programs available for personal computers: one or more S data objects (matrices or three-way arrays) are displayed on the screen. The user graphically (with mouse or cursor motion) identifies a cell for review, and possible updating, of its contents. The user can graphically identify one of the `spread.frame`'s associated macros, containing an arbitrary S expression, and then either update the macro's definition or execute it.

The current version, available on statlib (send spread from S), uses the classes and methods technology introduced in S Version 3 (Chambers and Hastie, 1992). The spreadsheet is designed in a modular fashion with device-specific methods for the display and updating of spreadsheet objects. We include methods for two devices: the generic S graphics device and a character based device using the emacs environment. The goal is an S Version 4 implementation that takes full advantage of the event-driven readers to be made available.

We discuss the design decisions mandated by the joint constraints of providing complete spreadsheet capability along with the full power of S, and the portability considerations for working with several different screen handling technologies.

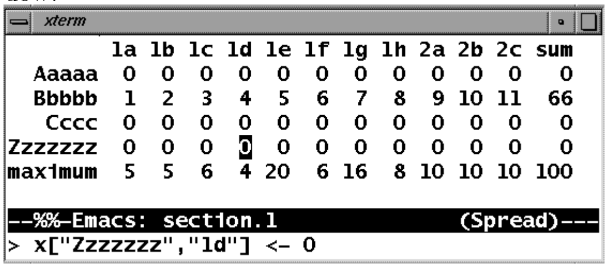
1 Introduction

A spreadsheet interface to S, as in Figure 1, can simplify several situations. We use it for simultaneous data collection, display, and summarization. It is useful for data handling and cleaning operations, expecially when working with multiple linked files. It can provide interactive multiple views of a single data file. It is also useful for statisticians working in environments where other people expect data to be spreadsheet compatible.

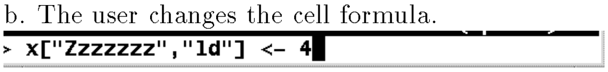
A spreadsheet in S differs in several important ways from an ordinary spreadsheet. Data is entered directly as an S object, hence the data conversion

Figure 1. Cell Update Forces Re-evaluation

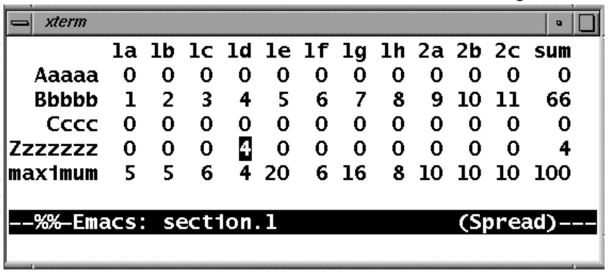
a. The entire `spread.frame` is displayed in a screen buffer. The user identifies one cell for updating, usually by moving a pointing device or cursor. The formula associated with the cell is displayed in a window.



b. The user changes the cell formula.



c. The system evaluates the new cell formula, and then reevaluates the entire `spread.frame`. The `sum` column of row `Zzzzzzz` has been updated.



step can be avoided. Any S function can be associated with the entire spreadsheet or any of its cells. The full analytic and display power of S is available.

We provide spreadsheet behavior by introducing a new class of S objects, the `spread.frame`, and several supporting methods. A `spread.frame` is an object of class `"spread"`, derived from the `"array"` class, with one or more attributes of mode `"expression"`. The default structure maintains four types of expressions: `"expr"`, `"before"`, `"after"`, and `"macro"`. The array `"expr"` and the vectors `"before"` and `"after"` are evaluated every time the `spread.frame` object is updated. When any cell of the `spread.frame` is changed, any other cell whose value is dependent on that cell is automatically updated.

Any valid S expression may be associated with the `spread.frame`. Since the `spread.frame` object is an `array` with additional attributes, it can be used in ordinary calculations.

2 Spreadsheet Kernel

2.1 `spread.frame` Object

The `spread.frame` object inherits from an ordinary matrix, data.frame, or three-way array. Any pre-existing object can be coerced to a `spread.frame` by the S statement `x <- as.spread(x)`. The `spread.frame` object `x` has the following four new attributes (any or all of which may be empty) in which executable expressions are stored.

`attr(x,"expr")` An array of expressions with shape identical to that of the data array within the `spread.frame x` itself. Each element of `"expr"` may be an arbitrary S expression that evaluates to a scalar. The expression may be a constant, or be dependent on other cells of the `spread.frame`, or be dependent on other objects in the `search()` path. The value of each expression is assigned to the matching cell of the data array. The entire `"expr"` is evaluated every time any cell of the `spread.frame` object is modified.

`attr(x,"before")` A vector of arbitrary S expressions. The entire vector is evaluated every time any cell of the `spread.frame` is modified and before the cell-by-cell expressions in the `"expr"` array. A typical application would be to modify/update one or more values of the `spread.frame` in response to changes in an associated variable .

`attr(x,"after")` A vector of arbitrary S expressions. The entire vector is evaluated every time any cell of the `spread.frame` is modified and after the cell-by-cell expressions in `"expr"`. A typical application would be to summarize information in the updated `spread.frame`. It could also be used to update another S object or a graphical display.

`attr(x,"macro")` A vector of arbitrary S expressions. Any individual expression is evaluated on request from the user.

2.2 Infrastructure: `"spread"` Methods

The fundamental method that drives the process is subscripted assignment `[←`. The simplest case, scalar assignment to a single cell, is

```
"[←.spread" ←
   function(object, ..., value)
{
   x ← object[,]
   e ← attr(object,"expr")
   e[[...]] ← substitute(value)
   x[[...]] ← eval(e[[...]])
   attributes(x) ← attributes(object)
   attr(x,"expr") ← e
   update(x)
}
```

The `"expr"` attribute is removed from the object and updated with the *formula* of the assignment, not the *value*. Then the formula for the updated cell is evaluated in the frame of the `"[←.spread"` function. This has the implication that other cells in the `spread.frame` must be referenced by the name `x[,]`. Finally, the `update()` function is called to evaluate the `"before"` expressions, cycle through the entire set of `"expr"` expressions, and then evaluate the `"after"` expressions.

The behavior is illustrated in Figure 2.

2.3 Design Decisions

Our initial design of the `"spread"` class included just two types of expressions, the `"expr"` and `"macro"`. These two types are all we need to obtain the behavior of the popular PC spreadsheets. We added the other two types for two reasons: execution efficiency and simplicity.

Simple matrix summaries, such as a set of row sums, are efficient in S when executed as a single statement. The identical arithmetic executed on a cell-by-cell basis does not use the generic array structure of S and hence bears a heavy performance penalty. For example, the gradesheet application in Section 4.1 has essentially one summary operation, computing a set of row sums. In an array language such as S, doing this with a single statement in the `"after"` vector makes more sense and is faster than the cell-by-cell method with individual summations for each cell in the `"sum"` column.

Having both `"before"` and `"after"` vectors allows symmetry in linking `spread.frame`s to other S objects or external information resources such as databases and real-time sensor arrays.

Figure 2. Simple `spread.frame` with Formula Assigned to Cells

```
> tmp                                # ordinary matrix
      1 2 3 4 sum
Aaaaa 0 0 0 0   0
Bbbbb 0 0 0 0   0
 Cccc 0 0 0 0   0

> x <- as.spread(tmp)                # turn it into a spread.frame
> x[1,"sum"] <- sum(x[1,-5])         # assign formula to cells
> expr(x)[[1,"sum"]]                 # formula is stored
sum(x[1, -5])

> x[1,1:4] <- c(10, 20, 30, 40)  # assign values
> x
       1  2  3  4 sum
Aaaaa 10 20 30 40 100                # sum column uses formula
Bbbbb  0  0  0  0   0
 Cccc  0  0  0  0   0

> names(attributes(x))
[1] "dim"       "dimnames" "expr"      "macro"
[5] "before"    "after"    "class"
```

2.4 Multiple `spread.frames`

The current implementation supports multiple `spread.frames` and multiple views of the same `spread.frame` through a single S session. We have introduced several objects to simplify the switching between `spread.frames`. The `spread.frame` kernel assumes the `spread.frame` is an S object named `x` as described in Section 2.2. The switching task is essentially a mechanism for keeping track of the user's name for the active `spread.frame`. We do this with a foreground function `fg()` and two frame-0 variables `.Active` and `.Active.buffer`. `.Active` contains the name of the active `spread.frame` object and `.Active.buffer` contains the name of the current two-dimensional slice. When the name of a `spread.frame`, say `"section1"`, is assigned to `.Active`, its value is duplicated in the S variable `x` and the screen display is updated. Upon completion of the update, the resulting object is assigned back into the original name stored in `.Active`.

The object `.Registry` is a data structure used to keep track of the `spread.frames` in the current `.Data`. In the current version `.Registry` is itself a `spread.frame` whose macro names are the names of the `spread.frames`. Our intent is to make `.Registry` a more general nested list. Its function will be to keep track of `spread.frame`-specific format information. Currently, the display appearance is that of the default `print()` command. Eventually, the display will be formatted according to the information stored in `.Registry`. The object `.Registry` is retained across login sessions.

All this occurs transparent to the user and, ideally, the user will work entirely through spreadsheet style interfaces, or similar buffers, and will never need to interact directly with an S session.

3 Device Handler Structure

The simplicity of the spreadsheet is that the user graphically identifies a cell for review, and possible updating, of its formula. For discussion purposes we will use the terminology of *placing the cursor on the cell*. When the cursor is placed on a cell entry, the cell formula is displayed for update. The underlying structure of a `spread.frame` is specified by the definition of the class `"spread"`. Details of identifying the cell and of displaying a formula for update are device specific. The idea of defining a series of `spread.frame` devices is similar to S supporting a series of graphics devices.

The current emacs device works by constructing S command sequences and appending them to the end of the S session buffer, essentially the same as if a user had typed it. Future development will make the behavior more like that of a smart graphics device with a spreadsheet mode. The end result will be a spreadsheet style interface which is a natural extension of the parser environment. An event-based interface is under construction to be used with Version 4 of S.

3.1 `emacs` device: `Spread` mode

We are working with Emacs 19.28. In this implementation of the `emacs` interface, the emacs environment is in charge and S is a subordinate process. The details are based on the `comint` (compiler interface) mode introduced with Emacs 19. We have defined a new read-only major mode, `Spread` mode, which permits cursor movement (cursor arrows and page-up and page-down commands), the `ENTER` key, and several `^C`-prefix keys. On X-windows terminals `Spread` mode accepts mouse clicks.

The effect of the `ENTER` key is to:

1. place in the minibuffer an S assignment statement that will re-create the current cell value from the current cell expression.
2. wait for a minibuffer entry.

Once the user has modified the minibuffer entry with a replacement expression, emacs will

3. print the modified statement in the `*S*` buffer, tell S to execute it, and finally print the updated `spread.frame` to the spreadsheet buffer.

The emacs interface is built on the `comint` mode. The user must first create a `.Data` directory in the appropriate directory. Then `Spread` mode is entered by loading an emacs lisp file `sprd-int.el`. The lisp file creates a new buffer `*Splus*` and runs `Splus` in it. If `.Registry` already exists in `.Data` it is displayed in `Spread` mode, otherwise a new `.Registry` is created and displayed. Existing `spread.frame`s are displayed by typing `"f"` or `[mouse-2]` on the name of the `spread.frame` in `.Registry`. Once a `spread.frame` has been displayed, all further review or change can be made entirely in `Spread` mode. Cursor or mouse motion is used to select a cell. The `ENTER` key or `[mouse-3]` is used to open a cell's formula for display or revision. Macros associated with the `spread.frame` can be displayed, changed, or executed by typing or clicking.

The mechanics of the interface are simple in principle. Each key or mouse action causes emacs to generate, type, and execute one or more S commands in the `*Splus*` buffer. Execution is initiated by the emacs `(comint-send-input)` function. Emacs waits for the S command to complete with the `(accept-process-output)` function. The emacs `(comint-output-filter)` function detects any errors in the excution of the S command. When an S error is detected in one of a series of S commands, all further generated S commands are cancelled and the display shifts to the `*Splus*` buffer.

All generated command sequences end with an S command to print the revised `spread.frame` to a temporary file, followed by an emacs command to read that file and display it in `Spread` mode.

The `*Splus*` buffer is always available for the user to enter S commands manually. Two common uses are for the initial creation of a `spread.frame` and to investigate S execution errors.

A major advantage of the emacs interface is the lack of dependence on workstation graphics display technology. It will will work well on any character-based terminal for which a `termcap` definition exists. In particular, the emacs device interface can be used over slow telephone connections.

3.2 Graphics Device

The graphics device interface is still in the existence demonstration phase. The screen is divided with the `split.screen()` functions into small pieces, one per cell of the `spread.frame`. The entire `spread.frame` is plotted with the `text()` command. `spread.frame` cells are identified by a control function based on the `locator()` function. Once the cell has been located with the mouse, the function will:

1. place in the S window an S assignment statement that will re-create the current cell value from the current cell expression.
2. wait for an S entry.

Once the user has modified the constructed assignment statement, using workstation-specific screen editing, and entered the modified statement, the control function will

3. execute the modified statement and print the updated `spread.frame` to the graphics device.

Potential advantages of the graphics device include the ability to place one or more graphs in other windows of the same graphics device in which the `spread.frame` itself is displayed.

3.3 Future Development

Version 4 of S has facilities for event monitoring. The `spread.frame`s will be able to send their updating information directly to a FIFO running in the background of the S session. Currently this information is sent to the `tty` in which S is running.

We will extend the infrastructure to include higher-dimensional arrays and, more generally, arbitrary subsets of data objects. We will support Import/Export of popular spreadsheet file formats. We will have SQL capability and be able to display arbitrary views of database formats.

We have outlined a tcl/tk interface that will work on both Windows and Unix version of S-Plus. The current emacs interface requires a Unix host. Although the graphics device interface works on both versions, it is still very rudimentary.

4 Examples

4.1 Gradesheet

I frequently collect data on several variables for a group of students and need to construct a summary of the results (for example, a grade sheet showing the points earned by each student on each exam question, and the total score). I construct a `spread.frame` with the student names as the row identifier, the question numbers as the column identifier. I construct an `"after"` expression

```
after(x)["sum"] <- expression(
  x[, 12] <- apply(x[, -12], 1, sum) )
```

that sums the values for the questions and places them in the `"sum"` column.

The `spread.frame` after one student's scores have been entered appears in Figure 1a. Note that I have placed the maximum scores for each sub-question in the `spread.frame`. This functions both as an aid when entering the data and as a permanent record when the filled out `spread.frame` is printed for my class notes book.

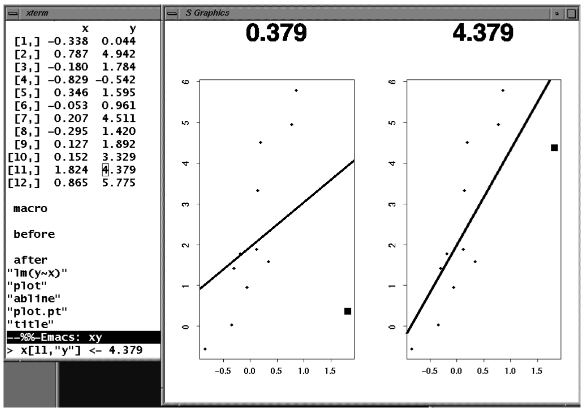
I find it easiest to record grades by entering a student's entire set of question scores at once. I therefore created a sub-class `"grade"` that inherits from `"spread"`. The `"grade"` class has one defining characteristic, when the cursor is placed on the student's name, the entire row except for the `"sum"` column is opened for data entry. (Normally, when a row label is entered, the whole row is opened.) For example, when I enter student `Bbbbb` by clicking on the `Bbbbb` cell in Figure 1a, the entry panel displays

```
x["Bbbbb",-12] ←
        c(0,0,0,0,0,0,0,0,0,0,0)
```

I modify this to the appropriate values from the marked exam:

```
x["Cccc",-12] ←
       c(1,2,3,4,5,6,7,8,9,10,11)
```

Figure 3. `after()` Changes Graphs as Point is Moved. Changing the value of a single point automatically changes the graph of the data. On the left graph, the point `x[11,"y"] = 0.379`. On the right graph, the point `x[11,"y"] = 4.379`.



On entry, the `spread.frame` updates itself and calculates the sum for that student. It would be easy to update the stem-and-leaf display or other summary graphic of the sums, as each individual student is added to the `spread.frame`.

4.2 Linking S Objects

The names of the macros of the `spread.frame .Registry` (described in section 2.4) are the names of other `spread.frames` (for example, `section.1`, `section.2`, `section.3` for an instructor teaching three sections of a course). Execution of one of the macros causes the named `spread.frame` to be displayed. The screen display of `.Registry` is therefore a menu into the other `spread.frames`.

4.3 Outliers in Regression

In Figure 3 we have created a `spread.frame` that contains two columns of data, `X` and `Y`. We have created an `"after"` expression that regresses `Y` $\sim$ `X`, plots the data with emphasis on the first point, and then draws the regression line. Every time a data point is changed the plot is redrawn. The application is to illustrate the effect of changing the value of one point on the regression line. We create an array of plots and then systematically change the value of the first point in the data. The resulting series of plots shows that a data point on the edge of the `X` range has a stronger effect on the slope than a data point in the center of the range.

Figure 4. Three-way `spread.frame`: Gradesheet with `sum` and `course` macros.

a. *sec3[,,"midterm"]*

```
           1  2  3  4 sum
  Aaaaa 19 20 18 21  78
  Bbbbb 24 24 19 21  88
   Cccc 13 46 83  0 142
maximum 25 30 20 25 100
```

b. *sec3[,,"final"]*

```
           1  2  3  4 sum
  Aaaaa 20 18 30 15  83
  Bbbbb  0  0  0  0   0
   Cccc  0  0  0  0   0
maximum 25 25 35 15 100
```

c. *sec3[,"sum",]*

```
        midterm final course
  Aaaaa      78    83    161
  Bbbbb      88     0     88
   Cccc     142     0    142
maximum     100   100    200
```

4.4 Multiple Views of a `spread.frame`

The interface can display multiple coordinated views of a single `spread.frame`. Figure 4 shows three views of a three-way grade sheet. Two slices correspond to two exams and a third slice, orthogonal to the first two, shows the course summary. The `spread.frame` has two `after` macros to caclulate the row sums for each exam and the course sums across exams. On an X-terminal, each slice might appear in its own window by taking advantage of emacs 19's ability to control multiple windows in a single emacs session. On a text-based terminal, the different buffers appear in the same emacs window.

The following commands are generated from each graphical interaction:

1. The commands
```
> attach(
+   '/disk5/rmh/spread/sprd3d/.Data')
> emacs.start('/tmp/spra00792')
```
are issued when the `spread.frame` interface is started. They attach the S library containing the `spread.frame` functions and call `emacs.start()` with the name of the directory where temporary files will be stored.

2. The command
```
> print.find.emacs('sec3[,,"final"]',
+  update.Registry=F)
```
is generated when the user places the cursor on the `sec3[,,"final"]` line in `.Registry`. This line prints all known views of the `sec3` object to temporary files and then reads them into emacs buffers.

3. The command
```
> emacs.cell('sec3[,,"final"]',2,1,1)
```
is generated when the user places the cursor on a row label in the `sec3[,,"final"]` buffer. The S function places in the minibuffer an S assignment statement that will re-create the current cell value from the current cell expression.

4. The commands
```
> x[ "Aaaaa",-5,"final" ] <-
+                   c(20, 18, 30, 15)
> invisible(assign(.Active, x))
```
are generated when the user edits the minibuffer. The execution of the revised assignment updates the entire `spread.frame` and causes all views of the updated `spread.frame` to be displayed in their emacs buffers. The assignment of the value of the working `spread.frame` `x` to the `spread.frame` named in `.Active` keeps S's idea of the `spread.frame`'s value synchronized with the user's idea.

4.5 Your Favorite Spreadsheet Application

Any spreadsheet should be able to be written in the S setting. This gives the double advantage of the familiar and powerful S analysis and graphics operations coupled with the ease of data entry from working in spreadsheet mode.

References

Becker, R. A., J. M. Chambers, and A. R. Wilks (1988), *The New S Language: A Programming Environment for Data Analysis and Graphics*, Wadsworth, Monterey, CA.

Chambers, J. M., and T. J. Hastie (1992), *Statistical Models in S*, Wadsworth, Monterey, CA.