TECHNICAL REPORT NO. 412 A

# EMILY: A Visualization Tool for Large Sparse Matrices

by

R. Bramley (bramley@cs.indiana.edu)

and

T. Loos (tloos@cs.indiana.edu)

July, 1994

All screen shots have been deleted from this version to save space

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

# Contents


**1 Introduction** **1**

**2 Users' Guide** **3**

2.1 What is **emily**? . . . 3

2.2 How to Run **emily** . . . 3

2.3 Window Types . . . 4

2.4 **emily**'s Display Areas . . . 4

2.4.1 Main Display Area . . . 4

2.4.2 Status Areas and Zooming . . . 5

2.5 Menu Options . . . 6

2.5.1 The File Menu . . . 6

2.5.2 The Colors Menu . . . 6

2.5.3 The Background Menu . . . 6

2.5.4 The Scaling Menu . . . 7

2.5.5 The Help Menu . . . 7

**3 Design Overview** **7**

3.1 Overall Control . . . 9

3.2 Read Routine – `ReadHBFile()` and `hbin()` . . . 9

3.3 Compression Routine - `CompressMat()` . . . 10

3.3.1 Purpose . . . 10

3.3.2 Terminology . . . 10

3.3.3 The Compression Algorithm . . . 12

3.4 Scaling Routine – `ScaleMat()` . . . 14

**4 Performance Results** **14**

4.1 Methodology . . . 14

4.2 User Response Time Data . . . . . 15

**5 Application Examples** **15**

5.1 Sedimentary Basin Modeling Problems . . . . . 15

5.2 Radiosity Problem . . . . . 16

5.3 Incomplete Factorization Preconditioners . . . . . 16

5.4 Domain Decomposition . . . . . 18

5.5 Applications Summary . . . . . 19

**6 Possible Extensions** **19**

**7 Future** **21**

7.1 Possible Extensions . . . . . 21

7.2 Conclusions . . . . . 21

**8 Acknowledgments** **22**

**A Data Structures** **23**

A.1 RCASParameters . . . . . 23

A.2 Matrix Data Structures . . . . . 24

**B Pseudo-Code Version of `CompressMat()`** **27**

**C Raw Data for Performance Statistics** **29**

# List of Figures


1 Diagram of **emily**'s High Level Design . . . . . 8

2 High Level View of `CompressMat()` . . . . . 11

3 Example Fractional Area Calculation . . . . . 13

4 . . . . . 32

5 . . . . . 32

6 . . . . . 32

7 . . . . . 32

8 Convergence History vs. Iteration Number . . . . . 33

9 Convergence History vs. CPU Time . . . . . 33

# 1 Introduction

The solution of linear systems of equations is probably the most ubiquitous problem in scientific, engineering, and statistical computations. In spite of the tremendous growth in computer memory sizes and computational rates, many problems remain intractable and currently insoluble because the linear subproblems are too large or have other unfavorable properties. Linear systems can be written as $Ax = b$, where $A$ is an $n \times n$ matrix and $b$ a given $n$-vector. Current research focuses on systems that are *large*, *sparse*, and *unstructured*. The term *large* is a moving target - while a system of 30 unknowns was considered large in the 1940's, modern codes for three dimensional fluid flows routinely have $\mathcal{O}(10^4)$ unknowns. The direct resolution of phenomena like turbulence can lead to systems with $\mathcal{O}(10^6)$ unknowns.

Storage of a dense matrix with $10^6$ unknowns would require $10^{12}$ words, clearly beyond the capabilities of current and projected computers. Fortunately the systems that occur in most applications are *sparse*, consisting mainly of zeros. By storing only the nonzero terms, the memory requirements are usually reduced to $\mathcal{O}(n)$, instead of $\mathcal{O}(n^2)$, well within the capabilities of current computer memories.

The *unstructured* property of modern linear systems usually arises from handling complicated physical domains, such as the interior of automobile engines or nuclear reactors. Simplex methods for linear programming also require solving highly unstructured linear systems, since the coefficient matrices $A$ are commonly created as an arbitrary collection of $n$ columns drawn from a larger $n \times m$ matrix. The lack of structure implies, among other things, the need for a common data structure capable of holding such systems, which can support the operations typically performed on the matrices.

Solving large, sparse, unstructured linear systems by direct factorization such as by Gaussian elimination is impractical, because of the phenenomon of *fill–in*, the creation of new nonzero entries during the factorization. Typically this can cause the number of non-zeros to be stored to grow to $\mathcal{O}(n^{3/2})$ or larger, along with a concomitant growth in the number of operations to be performed. Purely iterative methods are generally ineffective also, converging too slowly to be practical. Virtually all research in the numerical solution of linear systems concentrates on hybrid methods, which usually perform an inexact or *incomplete* factorization of the matrix in order to keep memory requirements acceptable, followed by an iterative method to refine the solution. The incomplete factorization is often called *preconditioning* the linear system, and the factors it produces are collectively called a *preconditioner*.

Crafting a hybrid solution method, especially for nonsymmetric matrices, is still more an art than a science. Techniques include reordering the matrix in order to get a nonzero structure more favorable to the algorithms used, scaling the matrix $A$ to make its underlying eigenvalue distribution more suitable for the iterative solve phase, and deciding where in the matrix to allow fill–in when creating an incomplete factorization preconditioner.

The size of the problems needing solution is also leading researchers to try solving them on parallel processors. Because shared memory machine architectures usually are not scalable (able to grow with the problem size and still maintain high performance) an important problem is the dis-

tribution of the matrix data across distributed memory architectures. This requires partitioning the data, preferably with processors receiving equal amounts of data and the amount of run-time communication between them minimal.

Previous work [7] [10] [1] has shown the utility of being able to visualize the location of the nonzero entries of the matrix and its preconditioners, as well as the magnitude of the entries in the matrix. For example, most iterative solvers converge faster if the matrix is diagonally dominant, that is, the main diagonal entry in each row is larger than the sum of the absolute values of the off–diagonal entries in the same row. This property is also important in assuring the existence of most incomplete factorization preconditioners (the incomplete factorization may fail because of pivots equal to zero, even when the corresponding complete factorization has all nonzero pivots). As another example, a direct inverse preconditioners is an explicitly created matrix $M$ that in some sense approximates the inverse of $A$. The iterative method is then applied to the system $MAx = Mb$. Being able to view the locations of the larger entries in the inverse matrix for smaller instances of the matrix $A$ can give important clues of where to allow non-zeros to appear in the matrix $M$. A third example of the utility of being able to visualize large sparse matrices is probably the most important: the debugging of codes that create and manipulate the data structures used for storing the matrices. A fourth example is the need to see how well partitioning methods work for splitting matrices and their data up amongst parallel processors. As with most scientific visualization tasks, the most important benefits are probably the unforeseen ones; can something be seen, that was not anticipated in advance?

Although there are systems for viewing dense matrices (MatVu), or for viewing the nonzero structure of sparse systems (MatVu, SPY in Matlab, PLTMT in Sparskit, and SHOWMAP in SMMS), none of these are targeted towards extremely large linear systems, and only MatVu can indicate the magnitude of the matrix entries. We have developed a Motif-based program, which can display extremely large sparse linear systems. Using a mouse, the user can select regions of the matrix to "zoom" in on, expanding them to see more detail. The windows opened up also display the matrix coordinates (in terms of row and column numbers) of the region being viewed, allowing the user to identify individual entries if desired. The visualization program, called **emily**, reads matrices stored in standard Harwell/Boeing format files. It can also be called from within Matlab, so that users can perform manipulations on the matrices at a high level and quickly view the results. One of the results of this project is that no single color map for the magnitude of the matrix entries is universally good, and when examining a matrix using **emily** it is useful to be able to change the background color and the scheme that maps magnitudes to colors, switching among several different ones to bring out different features. **emily** provides nine colormaps and seven background colors to choose from, accessible by clicking a mouse. There are two versions of **emily**: a Matlab version implemented using a `cmex` [6] interface and a traditional command-line interface version. Both versions can accept a file name as input; the file is assumed to be in Harwell-Boeing format. The Matlab version can also handle either sparse or dense matrix Matlab input matrices. Both versions have been tested on an IBM RS/6000 computer and the command line version has also been tested on Sun and various SGI hardware platforms.

A zoom feature has been implemented, allowing the user to select out an area with the mouse and show an enlarged view of the selected area. Currently, up to 10 zoom windows can be active at one time.

The program uses its own colormap (which may lead to a "technicolor" flash on some displays when leaving or entering the display window), so **emily** could allocate most of the colormap for displaying matrix values without interference. Status areas show the current position in the matrix (in terms of the row and column matrix indices), a color bar that both shows the range of colors and the numerical maximum and minimum values of the matrix, and the coordinates of the sub-matrix of interest.

Section 2 gives a more detailed description of **emily**, and how to use the package. Section 3 describes the design and details of **emily**, and in particular how extremely large matrices are mapped to a limited display device. One example matrix we have viewed has 214,000 unknowns; mapping a 214,000 $\times$ 214,000 matrix to a 1024 $\times$ 1024 display device requires some kind of compression algorithm, and Section 3 explains how we draw ideas from the graphics community to do this. Section 4 gives some performance figures for **emily**, indicating that for large matrices the time is almost entirely consumed in reading the matrix into the program, not in the compression or actual display. Section 5 gives examples of the utility of **emily**. Section 7 outlines possible extensions and gives conclusions.

# 2 Users' Guide

This guide is broken down into five parts: what emily is, how to run emily, the different types of windows emily uses, the different display areas, and emily's menu options.

## 2.1 What is emily?

emily is a tool for visualizing sparse matrices using either UNIX (UNIX is a trademark of UNIX Systems Laboratories) or Matlab. The UNIX version accepts input in Harwell-Boeing (HB) sparse matrix format. The Matlab version of emily can be used with HB matrix files, or to look at ether sparse or dense Matlab matrices.

## 2.2 How to Run emily

The UNIX command line usage is:

```
% emily [hb_file_name]
```

where `hb_file_name` is a file containing a sparse matrix specified in Harwell-Boeing format [2]. The Harwell-Boeing file is read and the sparse matrix is displayed. If no filename is specified, a dense matrix based on $A_{ij} = i + j$ will be calculated and displayed.

The Matlab version of **emily** is executed as:

```
>> emily(M)
```

where `M` is a matrix. If `M` is a string matrix, it is interpreted as a filename. The file, which must contain a sparse matrix in Harwell-Boeing format, is read in and the sparse matrix is displayed. If `M` is a sparse or dense matrix, its values are displayed.

Both versions use the same user interface and display layout.

## 2.3 Window Types

**emily** has two types of windows: the main window and zoom windows. The main window is created on invocation of **emily**. Zoom windows are created by performing a zoom operation, described in Section 2.4.2.

Generally, a zoom window and the main window are the same. There are some important differences, which will be pointed out as they come up.

## 2.4 emily's Display Areas

**emily** has several display areas and various new windows can be created to help focus in on submatrices and to give some additional information about specific points in the matrix.

### 2.4.1 Main Display Area

The main display area shows the (sub-)matrix of interest using the selected colormap and background color. The colormap is the palette of colors that **emily** chooses from to display the matrix. The background color is the color that is used to represent a non-fill (or zero) entry in the input sparse matrix.

The matrix is displayed with the upper left corner as matrix coordinate $(0, 0)$ and the lower right corner as matrix coordinate $(NR, NC)$, where $NR =$ the number of rows in the input matrix and $NC =$ the number of columns in the input matrix. In other words, the matrix is displayed as if it were in CSR (Compressed Sparse Row) format, even though the Harwell-Boeing format is CSC (Compressed Sparse Column).

If the input matrix consists of solely two values (usually 0 and 1), then **emily** renders the matrix as a bitmap, using the colormap values corresponding to the smallest and largest values therein. Otherwise, **emily** tries to approximate that matrix value's contribution to the displayed value by calculating the area in the display matrix one input matrix value covers and distributing the input value over that area in the display matrix.

**emily** is currently configured to have the window try to conform to the matrix. On resizing or selecting a submatrix, the displayed matrix will keep the selected aspect ratio (ratio of rows to columns) at the expense of wasting potentially usable display space. All windows, however, are initially square, so the easiest way to find the delineation between used and wasted space is to

select a new background color. This has the side effect of only repainting the matrix's background, not the whole window's.

Clicking on a point in the main display window will pop up a small subwindow with some information about the point selected. This subwindow will not be created if the user is in the middle of a zoom operation.

### 2.4.2 Status Areas and Zooming

A matrix status position indicator which displays the current position of the mouse in matrix (row, column) coordinates starting with (0,0) is just below the menu bar on **emily's** left. This can be handy for determining and selecting areas of interest.

Just below the position indicator is the zoom icon, which controls zoom operations. A zoom operation is **emily's** method of allowing user's to focus on a submatrix of the current matrix on display.

To execute a zoom operation:

- Click on the zoom icon with the first mouse button.
- Move the mouse to the main display area.
- Click on the starting point of interest in the main display area, again with the first mouse button.
- Drag the mouse until the area of interest has been selected.

A new window will be created with the submatrix of interest. This process can be repeated either on the main window or any zoom window until there are a total of 9 zoom windows open. The zoom windows are independent of each other. They will be all be closed if either a new matrix is read in or the main window is closed, terminating **emily**.

Underneath the zoom icon is the color bar with the maximum and minimum values of the compression matrix on the top and bottom, respectively, of the color bar. The color bar's color values change with the selected colormap; the maximum and minimum values change with the submatrix of interest.

Beneath the color bar, the maximum and minimum values of the input submatrix and the corresponding display window coordinates are displayed. This allows easy identification of extreme values.

There is a small text status area at the bottom of the window which displays the coordinates of the (sub)matrix displayed in that window and the total size of the matrix. Also, for a sparse matrix, the number of non-zero entries is printed.

## 2.5 Menu Options

At the top of the main window is a menu bar controlling five menus: File, Colors, Background, Scaling, and Help.

### 2.5.1 The File Menu

From the **main window**, the File Menu allows:

- reading of a new file by selecting the "New" option. This also closes all open zoom windows.
- clearing or redrawing the main window by selecting the "Clear" or "Refresh" options, respectively.
- choosing the scaling strategy by selecting the "Scaling" option.
- leaving **emily** by selecting the "Quit" option.

From a **zoom window**, only the "Quit" option can be selected from the File Menu, which closes only that zoom window. That is, if a window is created by zooming on a zoom window, and the original zoom window is closed, the child zoom window will remain open.

### 2.5.2 The Colors Menu

The Colors Menu allows selection of a colormap. Selecting a new colormap from the Colors Menu changes **emily**'s appearance instantaneously. The selected colormap is common for all windows, so changing the colormap on any window changes it for all windows. The default colormap is the "Spectral" colormap, which roughly follows the colors of the spectrum, from violet to red

Most of the colormaps have hopefully intuitive names, but "Pete's" colormap and the "Weather" colormap are representative not of the colormap, but of the origin of the colormap. "Pete's" colormap goes roughly around 2/3rds the HSV color cone, from black, through purple, red, orange, and yellow, to white. The "Weather" colormap goes from green through yellow to red and provides very nice constrast to a black background for all input values.

### 2.5.3 The Background Menu

The Background Menu allows selection of the display matrix's background color. A new background color causes the whole display matrix to be recalculated to insert the new background color. This implies that selecting a new background color will take some time – up to a minute or two in some cases.

The "Natural" background color is the color of the entry in the selected colormap corresponding to the minimum compressed matrix value. For example, for the "Spectral" colormap, the Natural background color is violet. The default background color is black.

Changing the background only affects the background of the selected window. However, all of the windows zoomed from that window will inherit the selected background color, unless changed in the child window.

### 2.5.4 The Scaling Menu

The Scaling Menu allows selection of the scaling strategy. The scaling strategy is a function applied to the value $v$ of each entry in the compression matrix $C$ that determines the displayed color of $v$. The compression matrix $C$ is a dense matrix of the same size as the display area of the window with real valued entries that correspond to the input value(s) that overlap that portion of the screen. See the section on `CompressMat()` for more details on how $C$ and $v$ are calculated. For now, the important fact is that the value $v$ corresponds to one or more input matrix value(s).

To scale $C$, four menu options are available:

- **Linear** The matrix values are scaled within the range of the current colors linearly using the formula: $\frac{(v-\min)}{(\max-\min)}$, where max and min are the minimum and maximum values in the compression matrix.
- **Absolute Linear** The matrix values are scaled linearly, but the formula is: $\frac{(|v|-\min_{|v1|\in C})}{(\max_{|v1|\in C}-\min_{|v1|\in C})}$, where $v1$ is an arbitrary compression matrix value.
- **Abs. Natural Log** The absolute values of the matrix values are scaled logarithmically using natural logarithms. If the minimum absolute value of the matrix is 0, then the next largest value is used. The formula is: $\frac{(\log|v|-\log(\min_{|v1|\in C,|v1|\neq 0}))}{(\log(\max_{|v1|\in C,|v1|\neq 0})-\log(\min_{|v1|\in C,|v1|\neq 0}))}$
- **Abs. Log Base 10** The same formula is used as for the natural log scaling, but the logarithm used is the $\log_{10}$.

### 2.5.5 The Help Menu

The "Help" option on the far right hand side of the menu bar provides a quick reference to **emily**'s features. Help is not available from a zoom window.

## 3 Design Overview

**emily**'s software was written in C. It is comprised of four major routines (see Figure 1 for their inter-relation):

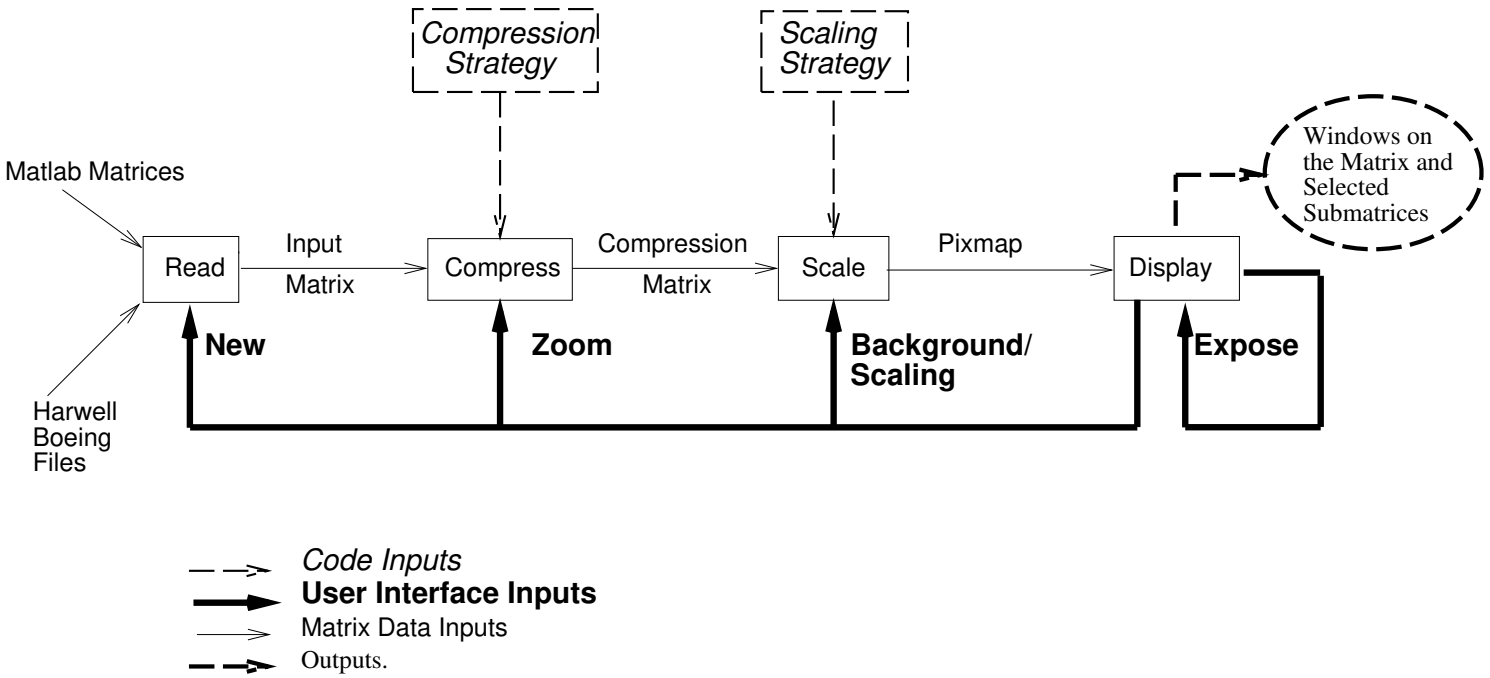



Figure 1: Diagram of **emily**'s High Level Design

1 *Read*: Matrix input, if necessary.

2 *Compress*: Conversion from the input matrix to a matrix of the same size as the display matrix with real valued entries.

3 *Scale*: Scaling from a matrix of real valued entries to a matrix of the same size with integer valued entries that correspond to the displayed colors.

4 *Display*: Viewing of the resultant scaled matrix and user interaction.

This design allows the flexibility to add or modify any routine without affecting the others; for example, during `emily's` development, a new compression routine was written, tested, and utilized, without affecting the reading, scaling, or display functions. This flexibility costs some additional memory for a compression matrix and somewhat reduced performance.

Also, the data structures used – relatively large structures containing all information needed for an operation – were chosen for flexibility. They contain too much information for almost any single function in a routine, but that data is available if needed. Also, function calling conventions don't require changes if a new data item is needed, since all functions are passed a pointer to the common structure and the new item is just a field in that structure. All of `emily's` interface functions try to follow that convention.

### 3.1 Overall Control

**emily**'s overall control is vested in three major areas:

- The primary user interface routine: `DrawMat()`
- The primary sequencing routine: `ReadCompressAndScaleMat()`. This function's name will be abbreviated below as `RCASMat()`, as RCAS stands for **R**ead **C**ompress **A**nd **S**cale.
- The primary display routine: `ShowRCASResults()`

**emily**'s primary internal data structure is `RCASParameters`, which is passed to `RCASMat()` and `ShowRCASResults()`. It contains data about the input matrix, submatrix of interest, the display interface, and the compression and scaling routines; see Appendix A for more details. Each window has its own `RCASParameters` structure – all of which are stored in the global `RCAS[]` array, which is indexed by the internal window number (0 corresponds to the main window, 1 for the first zoom window, 2 for the second, etc.) If a zoom window is destroyed, its `RCAS[]` entry is deallocated.

`DrawMat()` sets and creates up the display window using *Motif* calls (along with necessary *Xlib* and *XToolkit* calls), sets up the initial `RCASParameters` structure for the initial `RCASMat()` and `ShowRCASResults` calls, and starts up the main GUI application loop. `DrawMat()` takes two parameters: a string and a flag. The flag tells `DrawMat()` to treat the string as either a filename or an address of a `DaMatrix()` data structure, which is **emily**'s representation of a generic (dense or sparse) matrix.

`RCASMat()` uses the `action` field in its input to determine why it was called. The reasons why `RCASMat()` could be called are (in descending order): to read in a matrix, to compress (or expand) a matrix, or to scale a matrix. Calling `RCASMat()` for a particular action implies that all subordinate actions will be performed – so matrix compression implies matrix scaling. `RCASMat()` performs its duties by calling the main read, compression, and scaling routines, respectively `ReadHBFile()`, `CompressMat()`, and `ScaleMat()`. Each has its own data structure of primary interest. In the case of `CompressMat()` and `ScaleMat()`, a pointer to their primary data structure (respectively a `CompressionRequest` and `ScaleRequest`) is the only parameter to the routine. `RCASMat()` is an implementation of Figure 1, with the exception of the display routines. Its primary output is a Pixmap (an X data structure that represents the screen) and is stored as the `pix` field of an `RCASParameters` structure.

`ShowRCASResults()` is the simplest of the 3 primary routines – it takes the Pixmap created by `RCASMat()` and updates the display and status areas. It is also called when an X Expose event needs to be processed (i.e. when any part of any of **emily**'s windows is uncovered and needs to be redrawn).

### 3.2 Read Routine – `ReadHBFile()` and `hbin()`

The matrix input routine currently only reads in one type of file: sparse matrices in Harwell-Boeing format. (No dense matrix input routine exists, since there is no standard for dense matrix files.)

The input routine consists of the interface function `ReadHBFile()`, which takes 3 arguments: the filename, a flag that specifies whether C or FORTRAN array indexing conventions are to be followed on the output, and a pointer to a data structure called `HBMatrix`, which is described in Appendix A.2 "Matrix Data Structures" . It reads the header using the `ReadHBHeader()` function and calls `ReadHBBody()` to allocate memory for and read the body of the matrix.

The input routine for Harwell-Boeing files was developed to be used independently of the rest of **emily**. A Matlab interface routine was written to allow direct access to the routine via a Matlab function. The Matlab interface is called `hbin()` (hbin for Harwell-Boeing INput). `hbin()` takes a file name as its only argument and returns from 1 to 4 results, depending on the matrix file – the input matrix, a right hand side vector, an initial guess vector, and a solution vector.

Since the read routine was written in C and the Harwell-Boeing file format has a FORTRAN bias, it is somewhat slower and bigger than corresponding FORTRAN code. The major advantage to using C for `ReadHBBody()` is that it has a much cleaner interface to Matlab, which allows `hbin()` to return data directly to a Matlab user vs. writing data to a `.mat` file and requiring the user to load the `.mat` file.

## 3.3 Compression Routine - `CompressMat()`

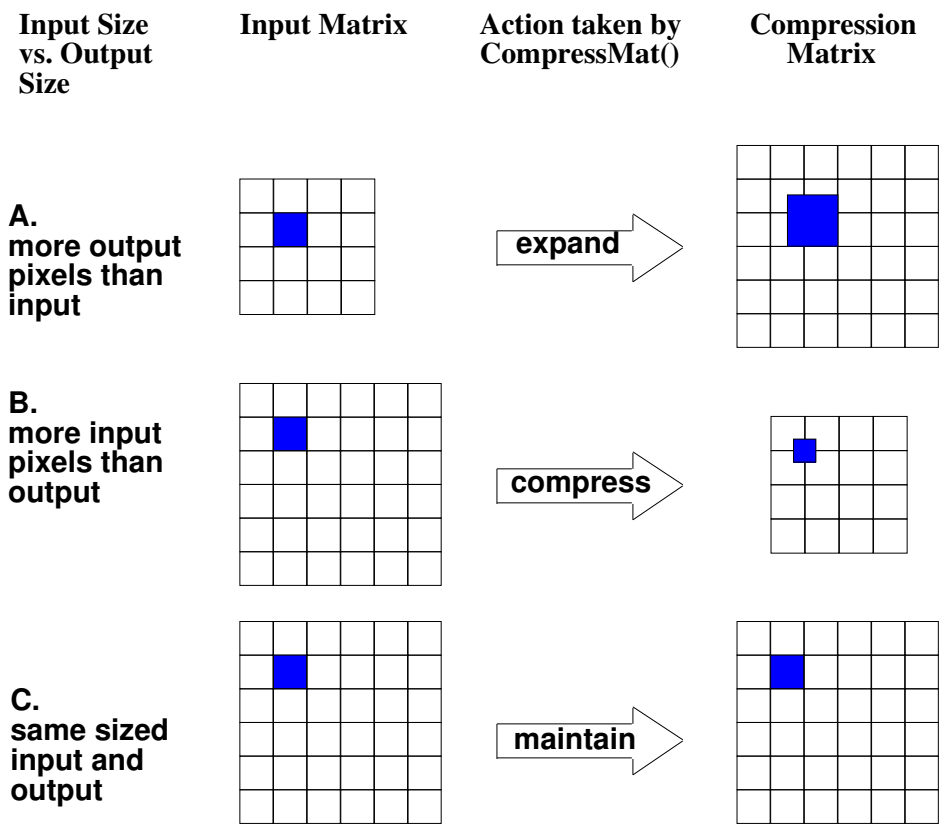
### 3.3.1 Purpose

`CompressMat()` has the task of transforming an input matrix or submatrix into a compression matrix. A compression matrix is a dense matrix of the same size as the main display with real valued entries that correspond to the input value(s) which overlap that portion of the screen (i.e. each compression matrix entry is a real value that corresponds to one output pixel.) So, `CompressMat()` can handle two conversions: sparse to dense matrix conversion and input matrix size to display matrix size. Other conversions are performed by `CompressMat()` to get a common output matrix, such as conversion to C array addressing, but those are not really important here and are not discussed below.

### 3.3.2 Terminology

An input matrix value is assumed to cover some amount of area on the display area – an input value can cover less than, exactly one, or more than one entry in the compression matrix. See Figure 2 for illustrations. Since one entry in the compression matrix directly corresponds to one screen pixel, entries in the compression matrix will be referred to below as pixels. Also, rectangular matrices may cover proportionally more (or less) area over the rows than over the columns in the compression matrix, so potentially different weights are used for the rows and columns. The row weight is the ratio between the number of rows in the output matrix to the number of rows in the input matrix. The column weight is defined in the same manner for columns.

Fractional pixels may be covered around the edges of the area covered by an input value, so those areas are calculated and their contribution is passed on to a compression strategy to update the




For all three cases, the one input value will contribute to all output pixels overlapped by the shaded area in the compression matrix.

The input value's contribution to each output pixel is proportional to the fractional area the input value covers in the output pixel.

Figure 2: High Level View of `CompressMat()`

pixel. A compression strategy is a function that takes the the current pixel value and an update to that value and combines the inputs to arrive at a new pixel value. **emily** comes with 4 compression strategies, which can only be chosen by modifying **emily**. The current strategy sums the input values for each pixel.

The other three extant strategies calculate:

- the square of the input values
- 1 for every non-zero entry (this creates bitmaps)
- the number of input values that overlap the pixel (an estimate of the density at that pixel for really large matrices)

### 3.3.3 The Compression Algorithm

The basic idea of the compression algorithm is a combination of unweighted and weighted area sampling [4, pp. 132-137]. In unweighted area sampling, the weights are the proportional area of the source image covered in the destination pixel.

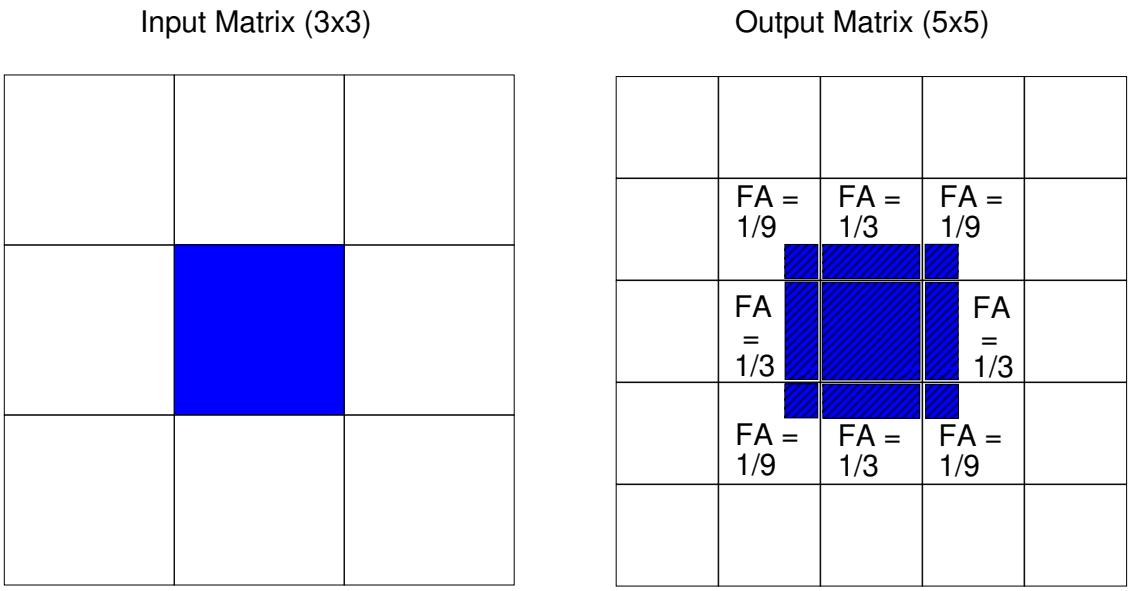
With weighted area sampling, these weights are calculated by some function $W(x, y)$, which gives a weight based on the distance $x$ and $y$ are from the center of the position on the destination grid corresponding to the source point. $W(x, y)$ is usually chosen to give preferential weight to the center of the pixel vs. the corners; for example $W(x, y) = 1 - \sqrt{(x - 0.5)^2 + (y - 0.5)^2}$

This is a useful formulation of $W$ if it is reasonable to assume pixel centers are more important than the corners, which is the case for anti-aliasing lines (the subject of the citation above). With a matrix, there is no reason to assume that the center of the input value is more important than the corners – a matrix value can be assumed to evenly cover the entire area of its entry. This allows the use of unweighted area sampling for calculating the contribution of a matrix value to a pixel. This contribution can then be calculated as: the fractional area of the pixel covered by the input value $\times$ the input value. See Figure 3 for an example calculation.

But, it is possible that the user may wish to run some different calculation on the weighted area other than the current compression strategy. The role of the compression strategy $CS(o, i)$ is shown below:

$$
\begin{aligned}
out[out_row][out_col] &= CS(out[out_row][out_col], contribution), \text{ where} \\
out[out_row][out_col] &= \text{the output pixel} \\
CS(o, i) &= \text{the compression strategy, currently } o + i
\end{aligned}
$$

Allowing $CS(o, i)$ to be specified changes the compression algorithm from being solely unweighted area sampling to a hybrid between unweighted area sampling and weighted area sampling. It is not




CompressMat for a 3x3 input to a 5x5 output matrix:
RowWeight = ColumnWeight = 5 output pixels/3 input values.

FA (Fractional Area) will be 1/3 * 1/3 = 1/9 on corners,
1/3 * 1 along the sides, and 1 in the center of the output
matrix.

Figure 3: Example Fractional Area Calculation

purely weighted area sampling in the sense of [4], since the updates to the pixel are based on the area being sampled.

Pseudo-C for `CompressMat()`'s algorithm is in Appendix 2. Not included in that pseudo-code is a small amount of code that recognizes if the input data items are bi-valued. If so, a flag stating that the input is a bitmap is set so that a scaling strategy could potentially take advantage of the fact. The actual compression routine in **emily** was modified to move the comparisons out of the inner most loops to improve **emily**'s performance.

### 3.4 Scaling Routine – `ScaleMat()`

`ScaleMat()` takes the compression matrix that was created by `CompressMat()` and scales it into integer values in the range $[0, \texttt{USER_COLORS} - 1]$. `USER_COLORS` is now defined to be 240 – this allows for the majority of the standard 256 color colormap to be used for matrix display, with 16 colors reserved for background colors and further development.

Like `CompressMat()`, `ScaleMat()` provides a loop to do the scaling and calls an input function (known here as a "scale strategy") to do the pixel scaling. The current scaling strategies are described above in the User's Guide in the "Scaling Menu" paragraph.

`ScaleMat()`'s output matrix is a Pixmap of the same size as the screen. A Pixmap is an area of off-screen memory that serves as a screen buffer (i.e. one can perform operations like "draw a point","draw a green line", etc. to a Pixmap). In some sense, this makes `ScaleMat()` the interface between matrix calculations and the user interface.

## 4 Performance Results

**emily**'s primary design goals did not emphasize speed; rather, they emphasized generality and flexibility at the expense of execution speed. However, some timing statistics are given below for comparison.

### 4.1 Methodology

All data below were generated from runs made with **emily** in standalone mode on an IBM RS/6000 computer model S-520H with 64 Mb of main memory and a 32 Kb cache. Each test involved invoking **emily** from the command line and closing the display window as soon as possible – this gives a fair estimate of the amount of time it takes to display a particular matrix. All time values were generated using the `time` UNIX$^{TM}$ command. For all data points, **emily** was run five times, the high and low time values were discarded, and the mean of the remaining three values is presented. All time values are given in seconds and all file size values are given in bytes.

The seven point difference operator matrix was run with various sized meshes from $3x3x3$ to $50x50x80$. Also, several matrices from the HB collection were tested – both real, unsymmetric, and

assembled (RUA) and pattern matrices were tested.

### 4.2 User Response Time Data

Approximate response time (wall clock) data for the slowest matrices in each class of matrices tested (seven point difference operator, RUA, Pattern, and the Maxwell matrix) was also collected. For visualization programs, such as **emily**, wall clock time may be misleading, but the times given below are about the time it takes to wait for **emily** to display the main window with each matrix.

Note: the Maxwell matrix is not part of the HB collection, but is derived from geochemical data provided by the Indiana University Chemistry Department. It has just shy of 4,000,000 non-zero entries stored in a 214,020 by 214,020 matrix.

| Approximate Real Time Values for Relatively Slow Matrices | | |
|---|---|---|
| Name | File Size | Real Time |
| 50x50x80 Mesh | 34965312 | 124 |
| psmigr3 (RUA) | 11574900 | 79 |
| bcsstk32 (Pattern) | 8701911 | 33 |
| Maxwell (RUA) | 118532229 | 506 |

Some statistics on its performance are given graphically in Figures 4 through 7 and the raw data can be found in Appendix C. From these tests, it appears that **emily** runs in time proportional to the input file size for each matrix of a given type (RUA, Pattern, etc.). This is not surprising, given **emily**'s basic RCAS design.

## 5 Application Examples

This section presents four different uses for viewing a sparse matrix. The first is a structured matrix from a sedimentary basin modeling problem, the second shows a relatively dense matrix that comes from a radiosity method in computer graphics, the third the effects of incomplete factorization preconditioners, and the fourth shows results of a graph partitioning code. These were chosen to give an idea of the range of applications for matrix viewing, and its advantages.

### 5.1 Sedimentary Basin Modeling Problems

The first example is from a linear system that occurs in modeling sedimentary basin problems. The matrix, contributed by J. M. Miles and P. Ortoleva, models complicated geochemical phenenoma on a three dimensional computation mesh that spans several orders of magnitude in the different coordinate directions. The system has 214010 rows and columns, and nearly four million nonzero entries. Figure **??** shows the overall matrix. Note the small bands of nonzeros in the upper right and lower left corners, indicating periodic boundary conditions on one of the coordinate directions.

Figure ?? shows a zoom window of part of the diagonal of the same matrix, indicating that the diagonal entries vary by as much as 11 orders of magnitude in just a few consecutive unknowns. Such drastic variation in a system indicates serious problems of scaling, and either the physical model needs to be changed to account for this, or the linear system solver used must implement sophisticated preconditioning strategies to account for it.

The most important use of **emily** on this matrix, however, is for debugging. The matrix was extracted from a program that only implicitly defined the linear system, and the ability to use **emily** to zoom in on individual entries allowed spot checks for questionable entries. This allowed fast debugging of the matrix extraction and data structure conversion routines.

### 5.2 Radiosity Problem

The image in Figure ?? shows a relatively dense matrix from a radiosity problem in computer graphics, supplied by G. Baranos and P. Shirley. The system comes from a scene with four walls, which together with the floor and ceiling give the six by six block structure. Each block in turn consists of $13 \times 13$ patches, and the $i, j$ entry is the fraction of light from patch $j$ to patch $i$. The main diagonal is the fraction of incoming light to a patch that gets re-emitted, normalized to 1. The rest of each diagonal block is zero, because there is no light directly emitted from a patch on a wall to another patch on the same wall - the angle between them is 180 degrees. Further information about the application can be extracted from the matrix image. Darker parts of the matrix indicate little light being received from patches with numbers corresponding to those column numbers, showing they can probably be neglected. The strong block structure and diagonal dominance of the system indicates block solvers are called for, and virtually any iterative method will converge for this problem.

The matrix has the form $I - F$, and frequently a Neumann series expansion $(I - F)^{-1} = I + F + F^2 + \cdots$ is used to approximate its inverse. Using **emily** can help see how rapid the convergence of the Neumann series is, by explicitly viewing the partial sums.

### 5.3 Incomplete Factorization Preconditioners

As mentioned in the introduction, iterative methods for solving systems $Ax = b$ usually rely on preconditioning to accelerate their convergence. A common class of preconditioners, ILU($s$), is based on carrying out the process of Gaussian elimination on $A$, but limiting the propogation of fill–in elements. For ILU(1), for example, original entries from the matrix are allowed to create new nonzeros in the factor, but those newly created entries are not. Instead, any fill-in they might cause is simply discarded during the factorization. In general, $s$ levels of allowed fill means that original entries in the matrix are allowed to create nonzeros (level 1 elements), those in turn can cause fill entries (level 2 elements), and so forth up to level $s$ elements, which are *not* allowed to create new nonzeros in the factors.

This process gives an approximate factorization $A \approx LU$ with $L$ and $U$ lower and upper triangular matrices, but helps limit the amount of additional storage below that which regular Gaus-

sian elimination would incur. The iterative method is then applied to the preconditioned system $(LU)^{-1}Ax = (LU)^{-1}b$; the better the factorization approximates the true factorization of $A$, the closer the preconditioned coefficient matrix $(LU)^{-1}A$ will approximate the identity matrix. It should be noted that in practice the preconditioned coefficient matrix is never explicitly formed, but whenever the iteration requires a matrix–vector product $w = (LU)^{-1}Ad$, it is computed in three steps:

1. $v = Ad$
2. Solve the lower triangular system $Lu = v$
3. Solve the upper triangular system $Uw = u$

This is because although $A$ is sparse and the incomplete factorization keeps $L$ and $U$ sparse, in general $(LU)^{-1}A$ and $(LU)^{-1}$ will be dense matrices.

As the third example application of **emily**, we examine ILU($s$) preconditioning on a small test matrix. It is still an active area of research to determine how effective ILU($s$) is, both in terms of the amount of storage and how "close" the preconditioned system is to the identity matrix. The first example explores this for the transpose of the Harwell/Boeing matrix FS_183_1, a matrix of order 183 with 1069 nonzeros coming from chemical kinetics applications. This small of a problem was chosen so that the preconditioned matrix $(LU)^{-1}A$ could be explicitly created and examined with **emily**. Figure **??** shows the ILU(0) factors for $A$, with both $L$ and $U$ displayed in the same matrix (since they are lower/upper triangular, they can be overlapped and displayed in the same array). Because $s = 0$ levels of fill are used, the nonzero structure of the combined L/U factors is the same as that of A. Note that in Figure **??**, which uses the spectral colormap with linear scaling, all the entries are red except for two near the diagonal around row 135. Figure **??** shows a zoom of the same matrix, displaying rows 133 to 142 and columns 133 to 143. This indicates one extremely small entry (around $-7.6 \times 10^8$) is causing all the other entries to appear relatively large and so have the same color. Figure **??** shows the ILU(0) factors, but now using the hot colormap and scaling based on the logarithm of the absolute value of the entries. This reveals more of the magnitude structure of the matrix, and in particular shows that the diagonal entries are usually larger than the off–diagonal entries. One potential source of numerical instability in incomplete factorizations are small pivot elements, which are stored as the diagonal entries of the combined $L$ and $U$ factors. This Figure shows that the pivots tend to stay relatively large for this problem, and the ILU(0) factors will probably not introduce numerical instabilities.

Figure **??** shows the ILU(1) factors. Many more nonzeros have been introduced, resulting primarily from the line of entries in row 40; a single row of nonzeros in the original matrix can cause the LU factors to become dense because every entry below them is a potential level 1 fill-in. In this case, there are 8386 nonzeros in $L$ and $U$, 8 times as many as were needed for ILU(0). Does this explosion in the number of nonzeros pay off in effectiveness of the preconditioner? **emily** can help answer this: Figures **??** and **??** show $(LU)^{-1}A$ for ILU(0) and ILU(1), resp., using the hot colormap with linear scaling. Note that for both $(LU)^{-1}A$ has diagonal entries near 1, and most of the off-diagonal entries are near 0. Furthermore, although both $LU$ and $A$ have entries of size $\mathcal{O}(10^8)$, preconditioning has reduced that range to $\mathcal{O}(10^0)$. This indicates that ILU(0) is probably a

good preconditioner for this problem. Although for the ILU(1) preconditioner the diagonal entries of $(LU)^{-1}A$ are closer to 1 and the off–diagonal entries are closer to 0, the improvement is not as dramatic as that obtained in going from $A$ to the ILU(0) preconditioned system. This indicates that the eight–fold increase in storage (and additional computation) needed to go from ILU(0) to ILU(1) will probably not pay off in this case.

Figure 8 plots the norm of the preconditioned residual vector $r_k = (LU)^{-1}(b - Ax_k)$ versus iteration number $k$ for CGNE, a common iterative method ([3]). Without preconditioning, the method does not converge within 100 iterations and the residual norm oscillates wildly. ILU(0) preconditioning, however, helps the method converge within 41 iterations, and ILU(1) preconditioning causes convergence within 19 iterations, both results consistent with what **emily** shows about the preconditioned systems. However, there is a much higher price to pay for the ILU(1) preconditioner because of its high fill-in costs both additional storage and computations. Figure 9 plots the residual norm versus CPU time, and shows that in the time it takes to compute the ILU(1) preconditioner, the ILU(0) preconditioned system has completely solved the problem, as was suggested from the examination using **emily**.

### 5.4 Domain Decomposition

An important problem in parallel computing is how to divide data up amongst distributed memory processors. The two primary goals are to have *load balancing*, a roughly equal amount of work on each processor, and to limit the amount of interprocessor communication required. For problems resulting from the discretization of partial differential equations (PDEs) on a regular mesh, this partitioning problem is relatively straightforward. Increasingly, however, engineers are using irregular meshes to better model geometrically complicated regions and to capture localized physical phenemona. In this case, dividing up the data can be expressed as a graph partitioning problem. Unfortunately, such a problem is NP-hard and so algorithms based on heuristics are frequently used.

Measuring the effectiveness of such partitionings is nontrivial, in part because there are several, possibly conflicting, goals. Load balancing and reduced communication requirements are only two; another could be balancing the number of boundary condition nodes. Generally users want a way to visually measure the effectiveness of the graph partitioning. The usual way of doing this is by coloring the nodes of the physical mesh with colors corresponding to the processor assignment. There are several problems with this. For a large number of processors it is difficult to distinguish between similar colors. For meshes with large differences in the sizes of the elements (ones with $\mathcal{O}(10^4)$ or more ratios in the areas of the elements often occur in adaptive mesh refinement), it is difficult to pick out the partitioning within crowded areas. Finally, for three or four dimensional meshes, all the problems of higher dimensional visualization occur.

It is more effective to view the partitioning in the adjacency matrix of the partitioned graph, ordering the nodes by first listing those in partition set 1, then those in partition set 2, and so forth. Row and column dividers can then be drawn in the matrix to show where the partitioning occurs. Figure **??** shows a three dimensional object partitioned into 16 sets, from which it is not apparent how good the partitioning is. Figure **??** shows the corresponding adjacency matrix, sorted

by partition number with mesh lines added for clarity. Immediately it can be seen that the number of nodes assigned to each processor are roughly equal, because the diagonal blocks are of about the same size. Furthermore, communications requirements between the processors will correspond to edges of the mesh that cross partition lines, which in turn correspond to nonzeros in the adjacency matrix that lie in off-diagonal blocks.

**emily** can be used to provide even more information about the partitioning problem. On most distributed memory machines, the "distance" between processors varies depending on the interconnection topology of the computer hardware. Some processors are adjacent, with communications taking one "hop", while others are more distant and messages between them can require several hops. The values assigned to the partitioned adjacency matrix can be given values to reflect this underlying topology, so that both the amount and type of interprocessor communication induced by the partitioning are immediately and visually apparent.

### 5.5 Applications Summary

A large sparse matrix viewer is clearly of great utility in a variety of computing areas. In scientific and engineering computing, many codes simply set and solve large sparse linear systems, with small amounts of code interspersed for relatively minor tasks. Although an applications scientist will want to see results in the physical domain she is examining, almost all the other phases of scientific computing can draw more and better information by viewing the underlying linear systems.

The images were obtained by interactively experimenting with the available color maps and scaling strategies available in **emily**. One of our conclusions is that large sparse matrices are best examined using many different viewpoints, and it is necessary to cycle through different colormaps and scaling strategies, and to zoom in on several regions, to bring out features of interest. This paper can only give brief static snapshots of what is best done dynamically and interactively.

## 6 Possible Extensions

Additional dialogs could be created to display the degree of diagonal dominance of the rows/columns. One method would be to calculate: $\frac{M_{ii}}{\sum_{j=1}^{n} M_{ij}}$ for row diagonal dominance or $\frac{M_{ii}}{\sum_{j=1}^{n} M_{ji}}$ for column diagonal dominance which leads to a value between 0 and 1 and use the current colormap to display the vector of dominance values. The corresponding column and row vectors of dominance values would be displayed to the right and bottom of `emily`'s main display window.

A slider bar could be added to select drop tolerance. The drop tolerance range is the range of input values which are considered to be zero entries. This could be done for the absolute value of the input value or separate upper and lower drop tolerance values could be stored and the value considered zero if `upper` $\geq$ `input_value` $\geq$ `lower`.

Many things are possible/designed in the software that aren't available to the user. For two examples, `emily` was designed to either stretch the matrix to fill the whole screen or to maintain

the selected aspect ratio and waste screen space (the latter is the current selection). This could be made a user option, either with another menu option, possibly along with support for a `emily` control resources file.

File output routines could be added. Potentially all three internally used matrices be written – the current selected input matrix could be written in either Harwell-Boeing format or some dense matrix format, the compression matrix could be dumped in the same dense matrix format, and the output matrix could be saved in a graphics (e.g. PPM or GIF) or PostScript format. With the Matlab version, if the input was from a file, the matrix could be returned to the user.

A closer approximation to weighted area sampling could be tried as a compression routine, with more data being passed to a compression strategy (the `CompressionStrategyParameters` data structure only has the old value and the update to that value). This may lead to a more realistic visual approximation of the input data.

# 7 Future

## 7.1 Possible Extensions

Additional dialogs could be created to display the degree of diagonal dominance of the rows/columns. One method would be to calculate: $\frac{M_{ii}}{\sum_{j=1}^{n} M_{ij}}$ for row diagonal dominance or $\frac{M_{ii}}{\sum_{j=1}^{n} M_{ji}}$ for column diagonal dominance which leads to a value between 0 and 1 and use the current colormap to display the vector of dominance values. The corresponding column and row vectors of dominance values would be displayed to the right and bottom of **emily**'s main display window.

A slider bar could be added to select drop tolerance. The drop tolerance range is the range of input values which are considered to be zero entries. This could be done for the absolute value of the input value or separate upper and lower drop tolerance values could be stored and the value considered zero if `upper` $\geq$ `input_value` $\geq$ `lower`.

Many things are possible/designed in the software that aren't available to the user. For two examples, **emily** was designed to either stretch the matrix to fill the whole screen or to maintain the selected aspect ratio and waste screen space (the latter is the current selection). This could be made a user option, either by adding another menu option or by adding support for a **emily** control resources file.

File output routines could be added. Potentially all three internally used matrices be written – the current selected input matrix could be written in either Harwell-Boeing format or some dense matrix format, the compression matrix could be dumped in the same dense matrix format, and the output matrix could be saved in a graphics (e.g. PPM or GIF) or PostScript format. With the Matlab version, if the input was from a file, the matrix could be returned to the user.

A closer approximation to weighted area sampling could be tried as a compression routine, with more data being passed to a compression strategy (the `CompressionStrategyParameters` data structure only has the old value and the update to that value). This may lead to a more realistic visual approximation of the input data.

## 7.2 Conclusions

**emily** and its spinoff **hbin** provide a way to view and files that contain sparse matrices specified in Harwell-Boeing format. **emily** allows a user a fair amount of control over their view of the matrix by allowing changes in the colormap and the submatrices of interest. Both were designed to work as either a stand-alone UNIX™ [1] application or run under Matlab.

A variation on unweighted area sampling was used to convert the input to a more "graphically friendly" form (called a "compression matrix") and the same idea – calling a user-specified function to do the weighting – was used to convert the real valued compression matrix to a displayable integer-valued form. This proved to be successful in allowing potential flexibility and display of

[1] UNIX is a trademark of UNIX System Laboratories, Inc.

necessary values, but there is probably still room for improvement.

# 8 Acknowledgments

Thanks to Ken Chiu for letting us steal his source code to use as a basis for the user interface and using his vast library, Marc VanHeyningen for his general software knowledge, Pete Shirley for teaching us about anti-aliasing and giving us the colors for "Pete's" colormap, the IU Chemistry Department for a REALLY BIG matrix to experiment with, and Emily Meyers for donating her name to the project.

The following books helped immensely for the user interface design, with X Windows[9], overall Motif programming[5], and really **tough** Motif programming [8].

# A Data Structures

## A.1 RCASParameters

The core data structure of `emily`, `RCASParameters` contains arguably too much information, but it is all the information required to keep and draw one window. One note: the input matrix pointer is shared among all windows and the original data are allocated by the main window.

```
typedef struct _RCASParameters
{
    int used; /* is this RCAS currently in use? */
    RCASAction action; /* what to do */
    char *file;        /* input file */
    DaMatrix *input;   /* input matrix */
    DenseMatrix *comp; /* compression matrix */
    CompressionStrategy *cstrat; /* how to compress? */
    ScaleStrategy *sstrat;       /* how to scale? */
    int conformal; /* conform to matrix? or screen? */

    /* sub matrix support */
    int input_startrow; /* starting input row of interest */
    int input_startcol; /* starting input column of interest */
    int input_endrow; /* final input row of interest */
    int input_endcol; /* final input column of interest */

    /* returned parameters */
    double min; /* minimum value in the compression matrix */
    double max; /* maximum value in the compression matrix */
    double absmin; /* minimum value wrt absolute value in the comp. matrix */
    double absmax; /* maximum value wrt absolute value in the comp. matrix */
    int zero_in_input; /* was zero seen in input? */

    /* data about the input */
    double input_min; /* minimum value in source matrix */
    int input_min_col;
    int input_min_row;

    double input_max; /* maximum value in source matrix */
    int input_max_col;
    int input_max_row;

    double input_absmin; /* |min value| in source matrix */
    int input_absmin_col;
    int input_absmin_row;
```

```
    double input_absmax; /* |max value| in source matrix */
    int input_absmax_col;
    int input_absmax_row;

    /* X/Motif data */
    int mapped;  /* has the current window been mapped to the display? */
    int bitmap; /* are there only 2 values in the input */
    int pixalloc; /* has a pixmap been allocated? */
    Pixmap pix;        /* output pixmap */
    int startcolor; /* starting color */
    int colormap; /* colormap chosen */
    int numcolors; /* colors are in the range [startcolor,startcolor+numcolors-1] */
    int force_black; /* if > 0, force scaling 0 values to be black */
    int what_is_black; /* "black" color if force_black chosen */
    int screen_width; /* cols */
    int screen_height; /* rows */
    Window window; /* the window to draw to */
    Display *display; /* display that holds pixmap */
    GC *gcs; /* array of GCs used for drawing output matrix, one per color */
    WidStruct wids;    /* widgets needed */
} RCASParameters;
```

## A.2 Matrix Data Structures

`DaMatrix` is the generic matrix type for `emily`. As can be seen below, it's a union of a dense and a sparse matrix.

```
typedef struct
{
    MatrixType type; /* type of matrix: sparse(HB) or dense */
    union { /* the actual matrix */
        HBMatrix *hb;
        DenseMatrix *d;
    } data;
} DaMatrix;
```

The HBMatrix type is based on the specification of the Harwell-Boeing matrix format given in Duff, et. al [2]. It is a sparse matrix format that stores the $MxN$ matrix with $NZ$ non-zero entries in 3 arrays: a $N+1$ length integer vector (`col_inds[]`) of column pointers, a length $NZ$ integer vector (`row_inds[]`) of row indices, and a length $NZ$ numerical vector (`values[]`) of data values (usually doubles). Two adjacent column pointers `col_inds[c]` and `col_inds[c+1] - 1` contain the indices into the row index vector and values vector that hold the entire column.

For example, take the following matrix $\begin{pmatrix} -7 & 0 & 0 & 0 & 2 \\ 0 & 15 & 2 & 0 & 4 \\ 0 & 0 & 0 & -6 & 0 \\ 1.3 & 7 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 8 \end{pmatrix}$

Using the FORTRAN array indexing convention (start with 1), the sparse version of the matrix would look like:

| Subscripts | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| col_inds[] | 1 | 4 | 6 | 8 | 9 | 12 | | | | | |
| row_inds[] | 1 | 4 | 5 | 2 | 4 | 2 | 5 | 3 | 1 | 2 | 5 |
| values[] | -7 | 1.3 | 4 | 15 | 7 | 2 | 1 | -6 | 2 | 4 | 8 |

Also, there is a rather detailed header of the file that describes how to read in the matrix, how many elements it has, etc. For `emily`, this header is defined in the `HBHeader` structure. This header is just about unintelligible without reading Duff, et. al. [2] and would not shed much additional light, so it was excluded.

```
typedef struct
{
    /* header */
    HBHeader head;

    /* data */
    int *col_inds; /* column pointers */
    int *row_inds; /* row indices */
    double *values; /* data values */
    double *RHS; /* RHS's */
    double *guess; /* starting guesses */
    double *exact; /* exact solutions */
}  HBMatrix;
```

A dense matrix is a much simpler matter: all that's needed is the number of rows, the number of columns, and the actual data items, stored as doubles. `emily`'s representation is slightly more complex, as it supports both one-dimensional (a vector) and two-dimensional (an array or matrix) representations of a dense matrix. Also, the data could be in row major (C representation where the columns vary fastest) or in column major (FORTRAN representation where the rows vary fastest) order. The definition of a `DenseMatrix` is below:

```
typedef struct
{
    int NumRows;   /* number of rows in the matrix */
```

```
    int NumCols;   /* number of cols in the matrix */
    int one_d;     /* is matrix stored in one-d format */
    int row_major; /* is matrix stored in row major format? */
    union
    {
        double *d1;
        double **d2;
    } data;
} DenseMatrix;
```

## B Pseudo-Code Version of `CompressMat()`

Note: this is an idealized version of the code used in `emily`. The actual code was written to remove conditional statements, to handle bitmaps, pattern matrices, and various methods of ordering matrices (row vs. column major, etc.), and to determine the maximum and minimum values of `CompressMat()`'s input.

```
row_weight = number_of_output_rows / number_of_input_rows
col_weight = number_of_output_columns / number_of_input_columns
for each row R in the input
   for each column C in the input
      if (input[R][C] is non-zero)
             upper_left_row = R * row_weight
             upper_left_col = C * col_weight
             lower_right_row = (R+1) * row_weight
             lower_right_col = (C+1) * col_weight

             // for top and left, use 1 - fractional_part to get the
             // amount spilling off the area... for example, if the
             // left most column is calculated to be column 57.2
             // the actual starting column is 57 and its weight
             // is 1 - 0.2 = 0.8 (the proportion of the fractional
             // pixel area covered by the input)

             top_fractional_area = 1 - fractional_part(upper_left_row);
             left_fractional_area = 1 - fractional_part(upper_left_col);
             bottom_fractional_area = fractional_part(lower_right_row);
             right_fractional_area = fractional_part(lower_right_col);

             for each row R1 from upper_left_row-1 to lower_right_row
                for each column C1 from upper_left_col-1 to lower_right_col
                    if (R1 < upper_left_row) // if above pixel area covered
                       if (C1 < upper_left_col)  // if to the left
                           // update with top,left fractional areas
                           update = top_fractional_area *
                                    left_fractional_area;
                       // else if off to the right, use top, left frac. areas
                       else if (C1 == lower_right_col)
                           update = top_fractional_area *
                                    right_fractional_area;
                       else // else just the row should be off
                           update = top_fractional_area;
                       endif
                    else if (R1 == lower_right_row) // if below area covered
```

```
            if (C1 < upper_left_col)
                update = bottom_fractional_area *
                         left_fractional_area;
            else if (C1 == lower_right_col)
                update = bottom_fractional_area *
                         right_fractional_area;
            else
                update = bottom_fractional_area;
            endif
          endif
          // if here, we know that the row is in range
          // but not so sure about the column.
          if (C1 < upper_left_col)
              update = left_fractional_area;
          else if (C1 == lower_right_col)
             update = right_fractional_area;
          else
             update = 1; // no fractional areas needed!
          endif

          // update the update by multiplying it by the input
          update = update * input[R][C];
          old_value = output[R1][C1];
          output[R1][C1] = compression_strategy(old_value, update);
        end // of C1 for loop
      end // of R1 for loop
   end // of C for loop
end // of R for loop
```

# C Raw Data for Performance Statistics

For all data below, all time values are given in seconds, all file size values are given in bytes, and all the matrices tested were square. See the "Methodology" section for more details on the data collection techniques.

These are the numbers for the HB collection RUA matrices, including the Maxwell matrix. Real time has been rounded to the nearest second.

| Data for HB Collection (RUA) Matrices | | | | | | |
|---|---|---|---|---|---|---|
| Name | N | NNZ | File Size | User Time | System Time | Real Time |
| sherman1 | 1000 | 3750 | 115911 | 3.92 | 0.30 | 12 |
| sherman2 | 1080 | 23094 | 587979 | 5.47 | 0.35 | 14 |
| sherman4 | 1104 | 3786 | 119394 | 3.84 | 0.33 | 11 |
| mahindas | 1258 | 7682 | 201933 | 3.74 | 0.25 | 7 |
| orani678 | 2529 | 90158 | 2219805 | 10.06 | 0.47 | 15 |
| psmigr1 | 3140 | 543162 | 8275203 | 38.88 | 1.41 | 77 |
| psmigr2 | 3140 | 540022 | 11508156 | 36.46 | 1.51 | 77 |
| psmigr3 | 3140 | 543162 | 11574900 | 36.92 | 1.62 | 79 |
| sherman5 | 3312 | 20793 | 586359 | 4.27 | 0.28 | 15 |
| sherman3 | 5005 | 20033 | 608958 | 5.13 | 0.36 | 14 |
| Maxwell | 214020 | 3950494 | 118532229 | 353.84 | 12.04 | 506 |

These are the data values collected for pattern matrices, with real time values rounded to the nearest second.

| Data for HB Collection (Pattern) Matrices | | | | | | |
|---|---|---|---|---|---|---|
| Name | N | NNZ | File Size | User Time | System Time | Real Time |
| bcsstk33 | 8738 | 300321 | 2503791 | 9.55 | 0.42 | 14 |
| bcsstk29 | 13392 | 316940 | 2679318 | 9.95 | 0.49 | 15 |
| bcsstk30 | 28294 | 1036208 | 8627958 | 27.38 | 0.91 | 33 |
| bcsstk31 | 35588 | 608502 | 5217534 | 20.26 | 0.80 | 48 |
| bcsstk32 | 44609 | 1029655 | 8701911 | 27.53 | 0.93 | 33 |

This is the table for the seven point difference operator mesh matrices. Again, real time has been rounded to the nearest second.

| Data for Seven Point Difference Operator Matrices | | | | | | |
|---|---|---|---|---|---|---|
| Mesh Size | N | NNZ | File Size | User Time | System Time | Real Time |
| 3x3x3 | 27 | 135 | 3120 | 4.73 | 0.23 | 8 |
| 6x6x6 | 216 | 1296 | 28765 | 3.33 | 0.23 | 8 |
| 6x6x10 | 360 | 2208 | 48764 | 3.30 | 0.27 | 8 |
| 6x10x10 | 600 | 3760 | 82776 | 3.40 | 0.23 | 8 |
| 10x10x10 | 1000 | 6400 | 147071 | 3.58 | 0.27 | 8 |
| 10x10x20 | 2000 | 13000 | 300327 | 4.04 | 0.27 | 8 |
| 10x20x20 | 4000 | 26400 | 609272 | 4.95 | 0.27 | 8 |
| 20x20x20 | 8000 | 53600 | 1236082 | 6.91 | 0.32 | 10 |
| 20x20x30 | 12000 | 80800 | 1944072 | 8.92 | 0.38 | 13 |
| 20x25x30 | 15000 | 101300 | 2453013 | 10.43 | 0.44 | 15 |
| 20x25x40 | 20000 | 135400 | 3278324 | 12.91 | 0.49 | 18 |
| 20x30x40 | 24000 | 168200 | 3941424 | 14.84 | 0.54 | 20 |
| 20x32x40 | 25600 | 173760 | 4206652 | 15.15 | 0.56 | 21 |
| 20x40x40 | 32000 | 217600 | 5267564 | 18.21 | 0.68 | 25 |
| 25x40x40 | 40000 | 272800 | 6603029 | 22.83 | 0.75 | 28 |
| 30x40x40 | 48000 | 328000 | 7938495 | 26.88 | 0.86 | 33 |
| 35x40x40 | 56000 | 383200 | 9273960 | 30.88 | 0.94 | 37 |
| 40x40x40 | 64000 | 438400 | 10609426 | 34.89 | 0.98 | 41 |
| 45x40x40 | 72000 | 493600 | 11944891 | 38.90 | 1.06 | 45 |
| 50x40x40 | 80000 | 548800 | 13280356 | 42.94 | 1.08 | 49 |
| 50x45x40 | 90000 | 617900 | 14952020 | 47.93 | 1.19 | 54 |
| 50x50x40 | 100000 | 687000 | 17320592 | 53.82 | 1.25 | 59 |
| 50x50x45 | 112500 | 773500 | 19500831 | 60.26 | 1.35 | 67 |
| 50x50x50 | 125000 | 860000 | 21681072 | 66.66 | 1.42 | 73 |
| 50x50x60 | 150000 | 1033000 | 26193652 | 79.60 | 1.57 | 87 |
| 50x50x70 | 175000 | 1206000 | 30579483 | 92.60 | 1.83 | 101 |
| 50x50x80 | 200000 | 1379000 | 34965312 | 105.78 | 2.81 | 124 |

## References


[1] Fernando L. Alvarado. "The Sparse Matrix Manipulation System: User and Reference Manual". Technical report, The University of Wisconsin, May 1993.

[2] Iain S. Duff, Roger G. Grimes, and John G. Lewis. "Users Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)". Technical report, Cedex and Boeing Computer Services, October 1992.

[3] Barrett et. al. *Templates for the Solution of Linear Systems: Buildings Blocks for Iterative Methods.* SIAM, Philadelphia PA, first edition, 1994.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, second edition, 1990.

[5] Dan Heller. *Volume Six: Motif Programming Manual for OSF/Motif Version 1.1.* O' Reilly and Associates, Sebastopol, CA, first edition, 1991.

[6] The MathWorks Inc. Matlab External Interface Guide. Source for cmex information. This reference is for Matlab Version 4.0a., August 1992.

[7] The MathWorks Inc. Matlab User's Guide. General purpose guide to Matlab, August 1992.

[8] Eric F. Johnson and Kevin Reichard. *Professional Graphics Programming in the X Window System.* MIS:Press, New York, NY, first edition, 1993.

[9] Adrian Nye. *Volume One: Xlib Programming Manual for Version 11 of the X Window System.* O' Reilly and Associates, Sebastopol, CA, seventh edition, 1991.

[10] Allan M. Tuchman and Michael W. Berry. "Matrix Visualization in the Design of Numerical Algorithms". *ORSA Journal on Computing*, 2(1):84–92, Winter 1990.

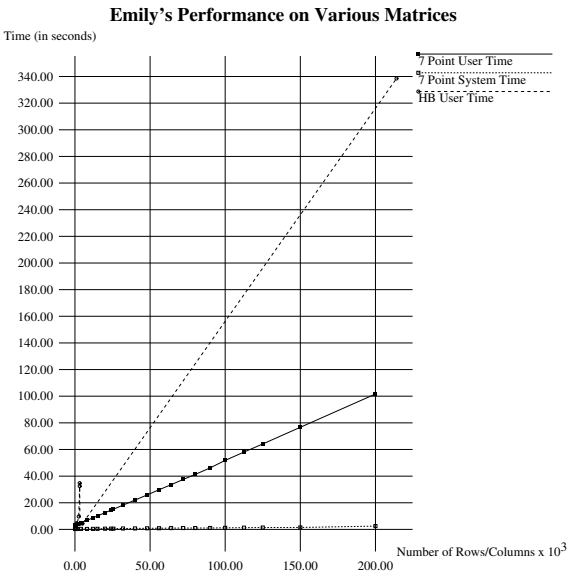



Figure 4:

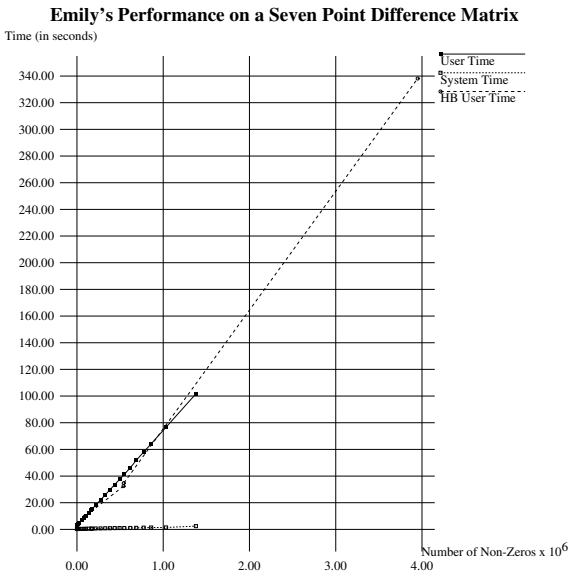



Figure 5:

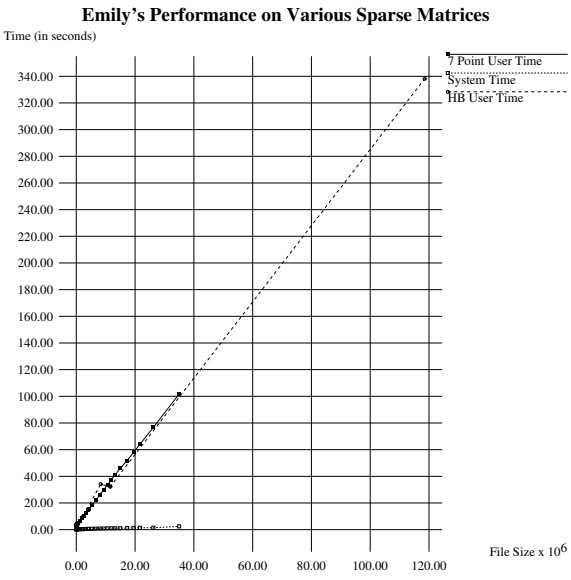



Figure 6:

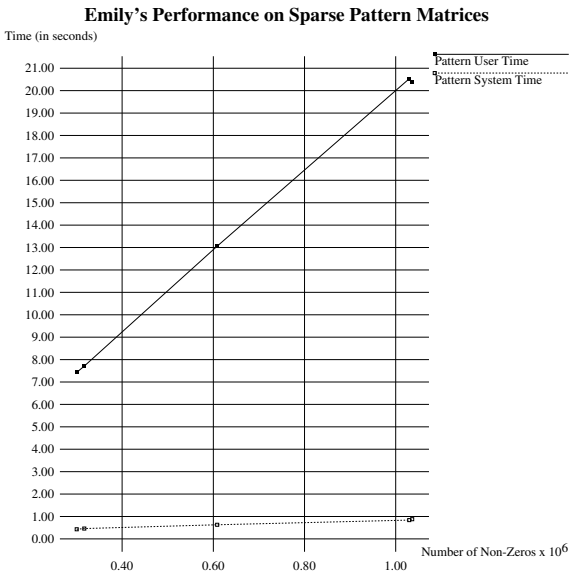



Figure 7:

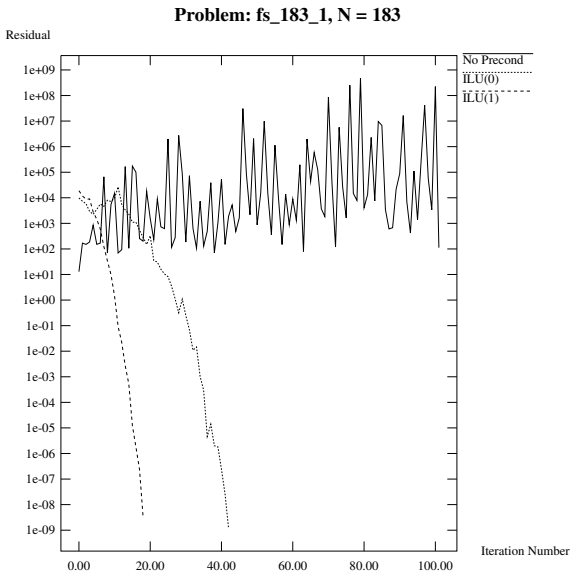



Figure 8: Convergence History vs. Iteration Number

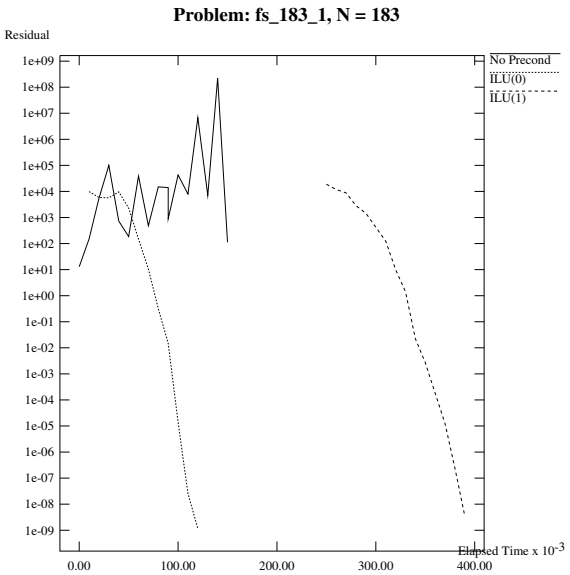



Figure 9: Convergence History vs. CPU Time