Simple Graph Editing

Richard J. Fateman and John L. Chen

EECS Department, University of California, Berkeley

Abstract: Applications of mathematics occasionally produce models that are best understood and visualized as directed or undirected graphs. Computer aids to mathematicians can contribute to the understanding and communication of this information, especially if well-integrated into the problem-solving working environment of the investigator. This particular exercise was prompted by a paper [1] by a colleague, F. Alberto Grünbaum. The particular need in Grünbaum's paper is to describe a graph relating input and output states via a matrix of probabilities. The paper's illustrative graph is of relatively small size (24 nodes, 60 edges), but its display required careful "manual" rearrangement.

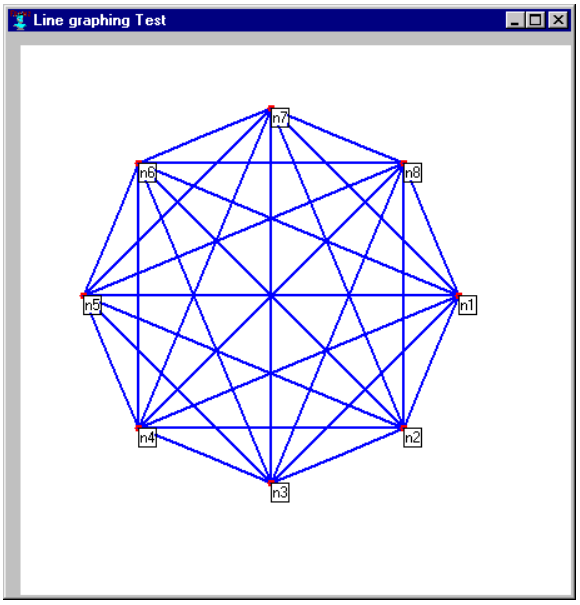

Figure 1

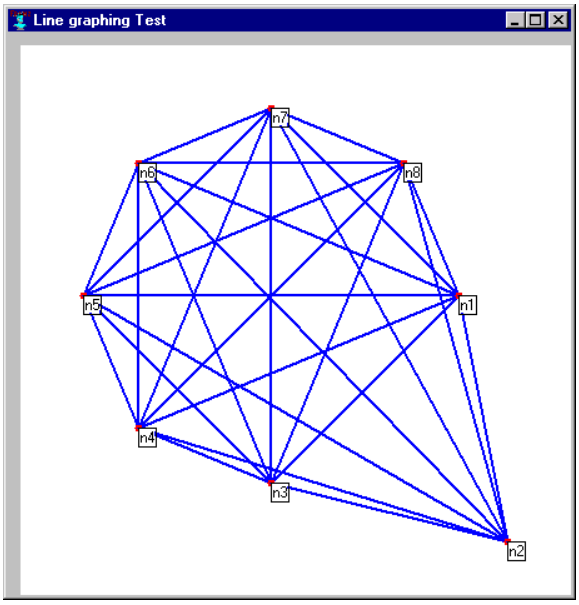

Figure 2

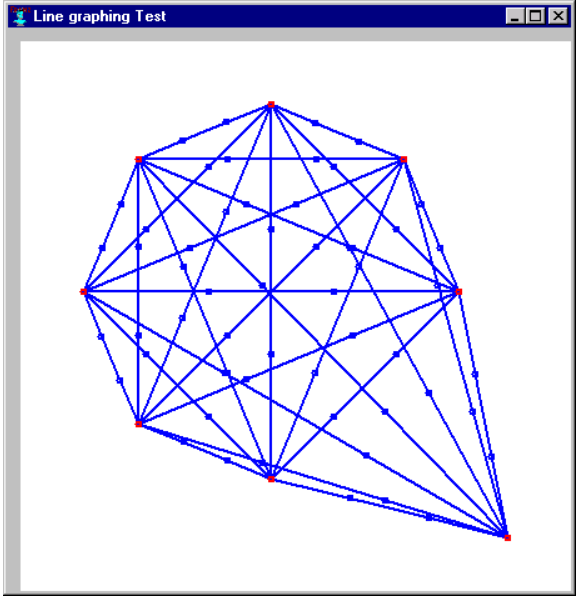

Figure 3

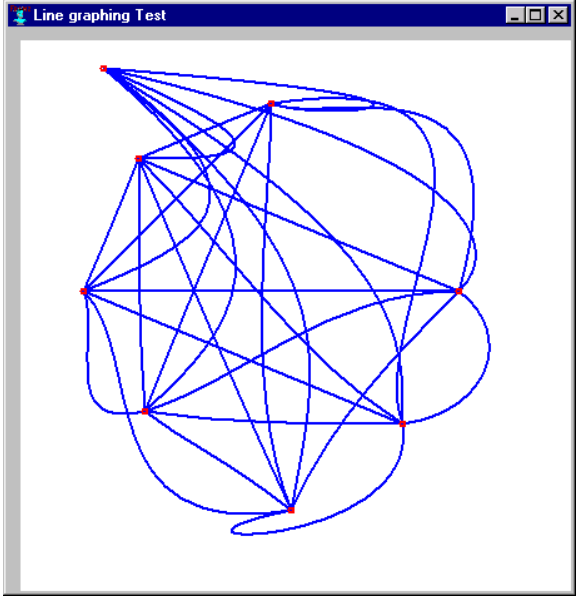

Figure 4

Introduction and Motivation

A key to understanding and interactively modifying complex geometrical/mathematical relationships, can be modifying the display of a graph. To illustrate some of the simple possibilities for display and modification, consider Figures 1—4.

This (even simpler) example graph given in Figure 1 is the complete undirected graph of 8 nodes (28 edges) arranged on a circle. (We have written other simple initial configurations such as a regular $n \times m$ grid layout with links to adjacent nodes.)

Figure 2 illustrates the effect of our first simple program, implementing only one action, moving a node. This is done by moving the mouse near a node, in this case the node labeled n2. Pushing the left-button down has the effect of moving the cursor exactly to n2. While holding the button down, n2 may be moved anywhere on the canvas, with the edges attached to it working like rubber bands. Releasing the mouse button leaves the node in its new location, and updates the calling program's notion of edges and nodes for the graph. The only other activities possible in the program are enlarging or shrinking the window, moving the window on the display, shrinking the window to an icon, or deleting the window, by the conventional mechanisms.

Grünbaum pointed out to us that it is difficult to see the edges, even after the nodes have been rearranged, and so he selectively curved the edges to improve the visibility, and in particular avoided situations in which edges appear to hit a node, but in reality only passed nearby. Figure 3 is the same as Figure 2, and is based on the same data structures, but with two changes: We have instructed the display program to omit node labels. We have also changed the edges representation so that instead of start/end pairs, the edges each have two extra Bezier control points. (The total number of points is 3n+1 where n is an integer, one by default). We have instructed the program to display the Bezier control points. Any of the points (including the nodes themselves) can be moved anywhere, using the same convention as in the Figure 1 transformation. A mouse down-move-release relocates a point. A possible result is illustrated by the somewhat fanciful Figure

4. We have, in this illustration, instructed the program to NOT display the Bezier points since they cease to be very helpful when they deviate far from the line they control in the case of relative sharp bends.

Functions

One of our goals in this design was to make the graph editing program easy to use. That is, we insist that users to be able to use this program "right out of the box" with minimal coaching.

From a programmer's perspective, one is continually tempted to add pieces of code and thus it is relatively difficult to decide when you have "enough" features. It is easy to find excuses for neat hacks to make a program "general." The programmer's internal model is joined to the design and implementation, and so can evolve to an elaborate program without great intellectual effort. It is a battle to keep in mind the goal of the simple and intuitive user's model.

Immediate notions for extensions include the ability to change the color of selected nodes or edges, to add or delete nodes and edges, relabel, resize, or color the component, etc.

With simple/intuitive in mind, how can these other functions be added? At the click of the right mouse button you see the other major functions displayed in a pop-up menu. There is also the need to specify whether an edge is a straight line, a "poly-line" or a bezier curve. In the latter two cases, how many points are needed?

(These figures show a few more features than the earlier screen shots, as well as different default appearances for the window decorations. These are set external to our programs.)

Adding a node

Being able to add to an existing graph seems useful to a graph editor. With the click of the right mouse button, a pop-up menu will appear with the following options, *'Add Node,' 'Delete,' Draw Edge'* and *'Rename'* (Figure 6). By selecting the *'Add Node'* option, you will have effectively added a node at the position that the right mouse button was clicked (Figure 7).

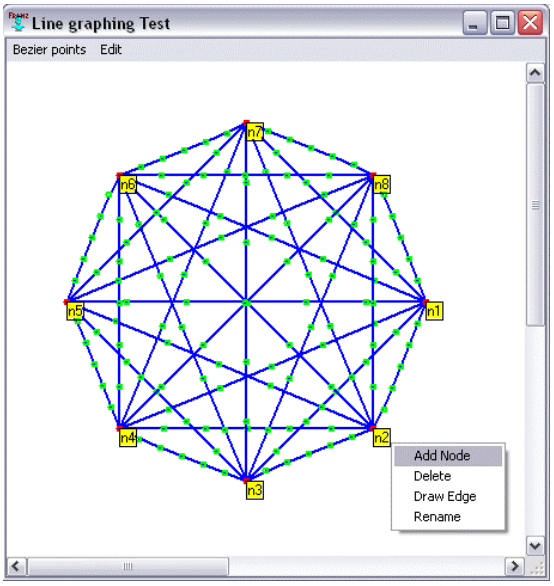

Figure 6

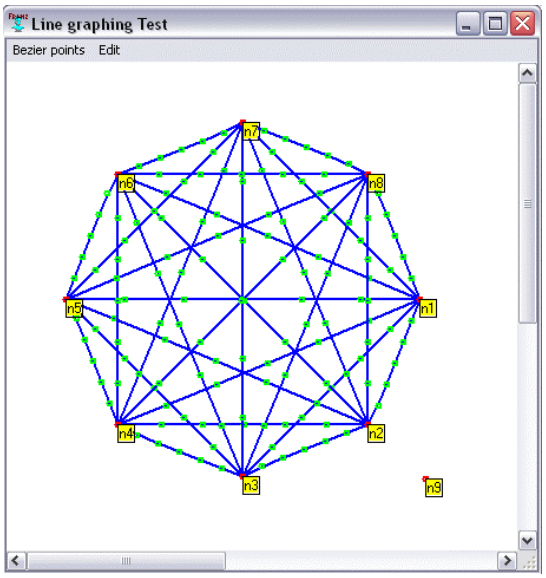

Figure 7

Renaming a node

The default name of a newly created node will be generated in sequence n1,…nk, but you may wish to set it to a more meaningful name. To do so, you right click on the chosen node (Figure 8), and then select the Rename option. This opens a box in which a new name can be typed, terminated by a newline (Figure 9).

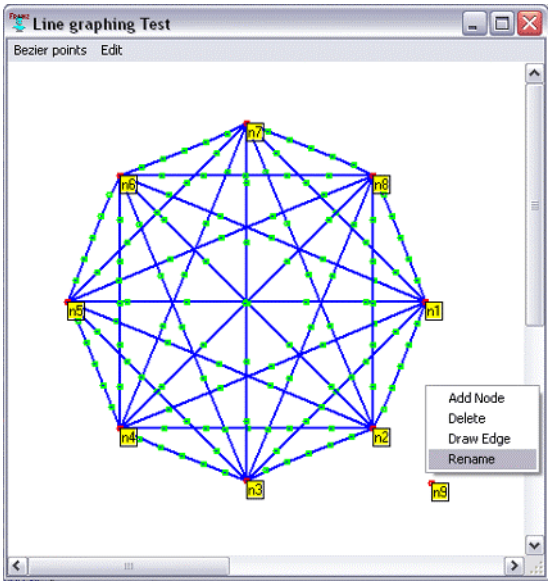

Figure 8

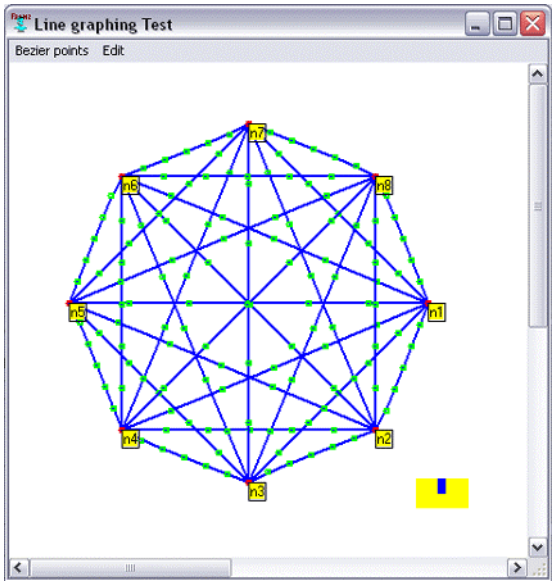

Figure 9

Adding edges

There are currently two ways you can add edges to the graph: (1) you can add edges from node A to all the other nodes on the graph or (2) you can add a new edge from node A to any other node (excluding nodes to which it is already connected). To do so you right-click on the starting node (Figure 10). Then select *'Draw Edge'* to see another menu consisting of '*None,' 'All'* and a list of all possible nodes you can connect to (Figure 11). Select the one you wish to connect to and you're done! (options can be selected by mouse click or the keyboard shortcut letters listed).

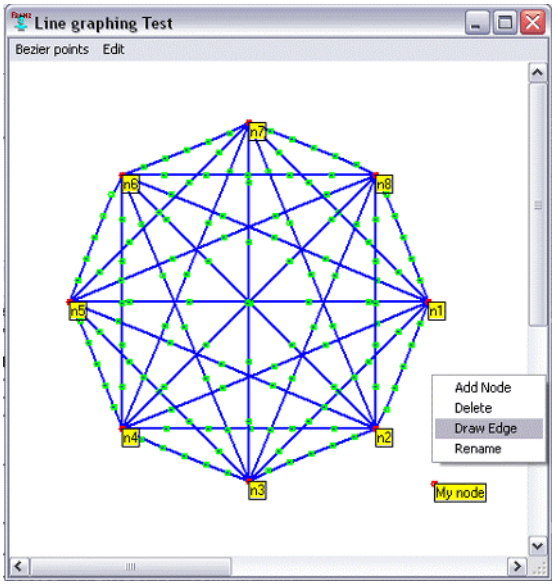

Figure 10

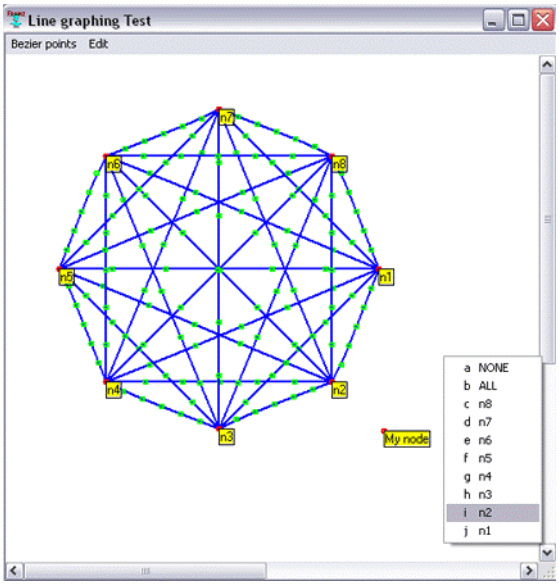

Figure 11

Deleting

If you wish to delete a node or an edge, start by right clicking on the item that you wish to delete and then selecting the *'Delete'* option. As a result, you will be giving a choice of either a node or an edge that may be deleted. If you are close to both a node and an edge, then you will be given a choice (Figure 12). However, if you are by far closer to a node then you will only have that option (Figure 13).

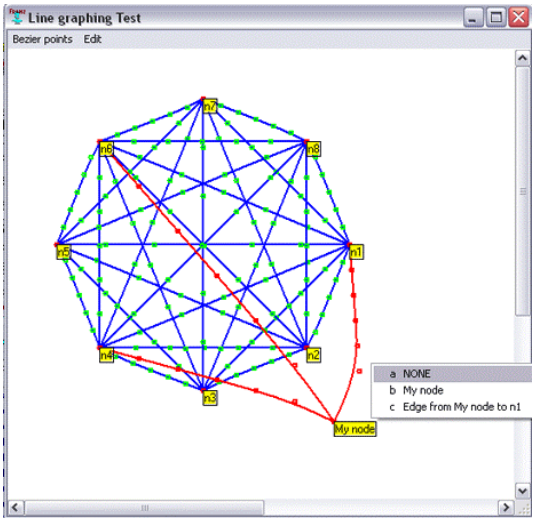

Figure 12

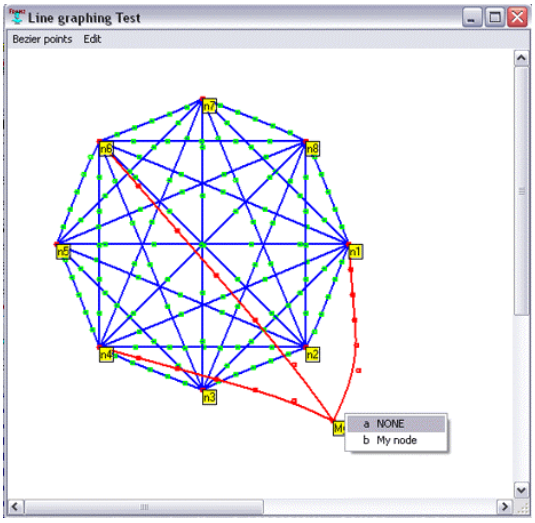

Figure 13

Menu Options

Currently, there are two fixed menu options, (1) *Bezier Points* and (2) *Undo*. Bezier points are the control points that are present on each edge. Under this option, the default setting is on *"Show them!"* (Figure 14) However, you can change the setting to *"Don't Show Them ..."* by clicking on the corresponding option or by pressing CRTL-D, which will effectively remove all the control points from being displayed in the window (Figure 15). If you wish to bring the Bezier points back, just select that option from the menu or press CRTL-S. Bezier or polyline control points are always shown on a selected edge.

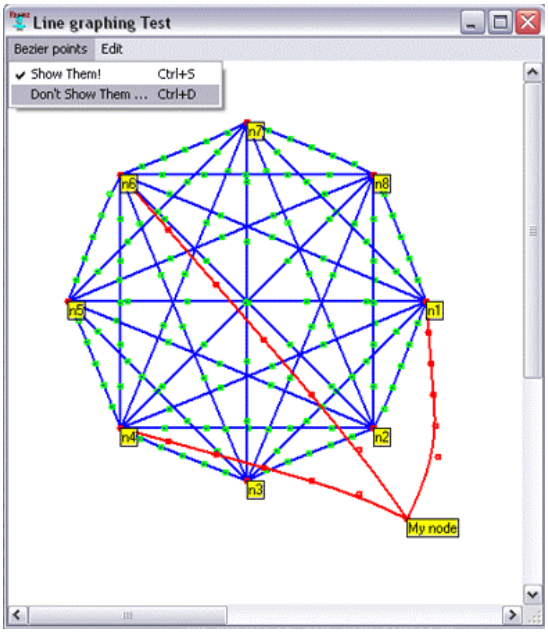

Figure 14

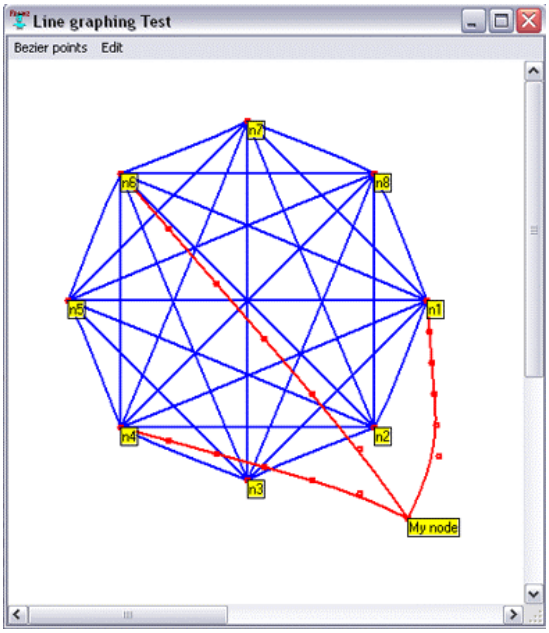

Figure 15

Another menu option that is available is *'Undo,'* which restores the graph the way it was before the most recent change. To do this you just click on the corresponding option under the menu item *'Undo'* or by pressing CRTL-U.

As we have stated before, we wanted to keep this editor clean and simple. Something that people can learn within a few minutes after being exposed to the controls. It is trivial for our purposes to add more functionality but we hesitate to go further without a user-review. We think there are advantages in making the code available in a relatively high-level language (object-oriented Lisp, with calls to standard graphics and windows routines) and in an interactive debugging environment). It makes it somewhat plausible that other people can add to the facilities. The biggest hurdle, in our experience, is to get past the initial barrier to get the first display, and thus the example given is much of the battle.

Nevertheless, what other programs can we anticipate writing?

- Programs to write labels on the edges, and keep the labels close to the edges. This is trivial if the edges are straight lines. In the case of Bezier or polylines, a point along the edge must be designated as the location for the label.
- Programs to select particular edges to be displayed as polyline or Bezier, and to set the number of control points on a per-edge basis.
- Variations on these programs for directed graphs.
- Conversions to/from graph file formats proposed by others.
- Front-ends to other programs to provide initial data for graphs in various additional default layouts.

There are a substantial number of interesting algorithms that operate on graphs to automate their geometric layout based on various constraints. We do not propose to do this since such an attempt would necessarily be duplicative, and we would hope to take advantage of existing programs written by others. Such issues in graph drawing are explored at http://www.graphdrawing.org a URL with numerous links, including references to two full-length monographs on the topic, an on-line tutorial for GDTools and BLAG, a batch layout program. A brief perusal shows that this latter package includes some 10 layout algorithms. This website includes a number of sets of benchmarks (for example, 388 undirected graphs, with up to a few hundred nodes), information on annual conferences, and competitions.

We expect our programs to be used through a computer algebra system (CAS). A rather large one (Macsyma, especially its open-source version Maxima) is conveniently already written in the same Lisp system in which these programs are written, and was a host system for Grünbaum's calculations. We are exploring options for a suitable "high level" CAS mathematical descriptive language for specifying these graphs, or at least roughing them out, so that a human can easily remedy shortcomings in the design. Another possibility which may be appealing is to lay out a graph in a spreadsheet, either as a connection matrix, or probably more compactly as a collection of nodes and edges. We

have already written links between Lisp and Excel, and there are also links to other CAS. Of course Excel does not really supply "symbolic" data except as strings.

It should not be difficult to provide auxiliary programs to convert these graphs to printer forms (postscript), or to standard file formats.

A natural question to ask is how many nodes and edges can be handled by these programs? Clearly the amount of computation to display any of these graphs increases with the complexity of the graph. The actual "screen real estate" occupied by the graph is not a severe limit since both horizontal and vertical scrolling can be used. We know that some of the operations could be made more efficient by "caching" unchanging parts of images instead of recomputing everything in the graph during "rubber-banding." A version of caching using bitmaps backing store is provided in the graphics system we are using; we may convert to an alternative model suggested by Ken Cheetham of Franz Inc. This avoids certain blinking artifacts when rapidly moving a mouse and dragging curves.

It is unclear how large a graph could be usefully edited by a human in an attempt to aid visual comprehension. A purely symbolic computational representation, say as a sparse connection matrix, may be more useful than an image if there are thousands of points and edges.

The advantage of these programs is that for modest size graphs, it may be possible to rearrange nodes to display attractive symmetries, useful clustering, or other properties. We expect that algorithms which are intended to automatically transform the visual appearance of graphs can be added.

Disclaimer and What About Other Programs?

As far as we know, none of these ideas are new; the initial draft of the program was a few pages of code. The novelty, if there is any, is in making them available as a suite of functions to be called by a CAS. A search on the internet reveals a number of programs, free and commercial with elaborate interfaces. In each of these cases there is a burden on the user to learn the menus and controls, and more significantly, to learn how to convey information to the "home base". Placing a diagram on a clipboard, or writing it out to a file in some "standard" form, only to require the CAS to re-parse the data, is likely to be require expertise beyond that of the casual user.

Interfacing to a Computer Algebra System

Assuming the CAS has a reasonable representation for a graph, perhaps as a list of nodes with their names and locations, plus a list of edges with any relevant information on bends, we can say that a graph G is a pair [nodes, edges]. We can extract from our programs "single minded" functional-style commands like MoveNode(n2,newposition) a situation which takes us from Figure 1 to Figure 2. Or BendEdges(data) going from Figure 3 to Figure 4. While it is sometimes a good idea to use "more powerful tools", this functional approach may be better because it can give command control to the computational engine, and doesn't require the human touch at each decision.

The programs used to produce the Figures 1—4 are (all together) 217 lines of Lisp code, often consisting of declarations of methods associated with windows and interaction, as well as calls to a Common Graphics library, a set of object-oriented tools in Allegro Common Lisp. The latest version (August 12, 2002) is about 860 lines, including many comments.

The free "trial" version of Allegro Common Lisp [2] is quite sufficient to run these examples and develop extensions.

Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. John Chen was supported as an undergraduate NSF research student. We also are grateful for comments from Ken Cheetham of Franz Inc.

Bibliography

1. F. Alberto Grünbaum, "Nonlinear inverse problem inspired by three-dimensional diffuse tomography", *Inverse Problems* **17** (2001) 1907-1922.
2. http://www.franz.com/
3. http://www.graphdrawing.org