CS522 - Drawing Lattices. Handle Graphics.

Tibor Jánosi

March 18, 2005

The appropriate presentation of numerical results is often critical for the proper understanding of various financial phenomena. Certain relationships between numerical quantities might be very difficult to grasp, unless represented graphically. Financial (more generally: scientific) data visualization is a very important domain in itself. One subpart of problem 3 provides the pretext for a more thorough examination of Matlab's graphical capabilities.

0.1 Drawing the Lattice

You must draw recombining lattices, similar to those that you have seen in class (see figure 1 for an example).

At first sight, the task of drawing a lattice might seem non-trivial, primarily because one would have to determine the coordinates of all the component points. If we could find an easy way to think about the coordinates, the task of drawing a lattice would become much simpler. The two methods we present here are very simple; they are of comparable complexity, even though the explanations provided for the first one are more extensive.

0.1.1 The First Approach

If an "up" transition could represent a fixed-length movement along one axis, and a "down" transition a movement along the other axis, the coordinates of all the points in the lattice could be easily drawn. Let us assume that "up" represents a movement along the vertical axis (y-axis), and "down" represents a movement along the horizontal axis (x-axis). A lattice corresponding to 5 time intervals will then look like the one represented in figure 2.

To get the lattice in the final position we will have to rotate it clockwise by $\frac{\pi}{4}$ radians (45 degrees). Before worrying about rotations, however, let us understand how can we actually draw the lattice on this rectangular grid. Let us assume that we are drawing a lattice that corresponds to n subintervals.[1]

We can draw the entire lattice by drawing $2n - 2$ lines, half of them horizontal, and half vertical. The coordinates of the lines to draw are shown in table 1.

The rotation of a point around the origin can be accomplished by multiplying the respective point with a suitably chosen matrix. Using subscripts 0 and 1 to denote the initial and final coordinates, respectively, of a point rotated clockwise about the origin by an angle of φ radians, one can establish the following relationship:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

[1]In practice, given our screen resolutions, n must be small, probably not bigger than 10 or 15.

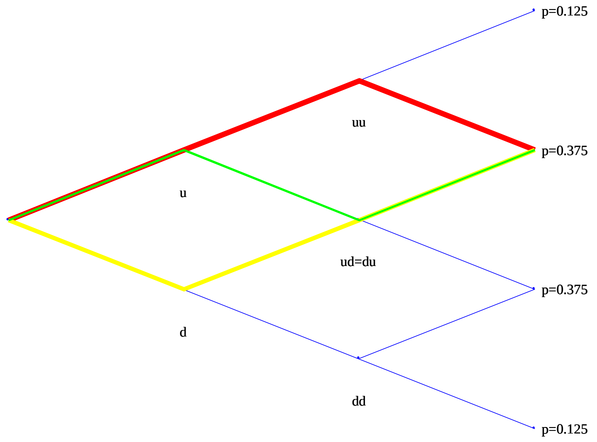

Figure 1: A simple recombining lattice. Intermediate states are labeled with strings showing the number of "up" and "down" transitions from the initial state. Final states are labeled with their respective probabilities, assuming that the transition to the "up" state occurs with a probability of 0.5.

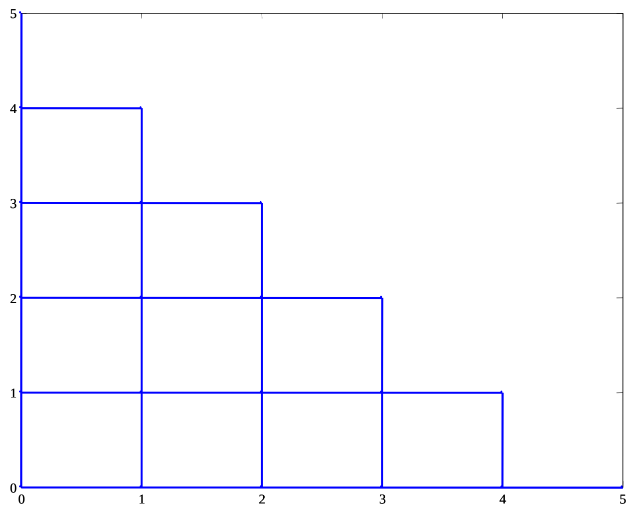

Figure 2: Lattice in original position on a rectangular grid.

Horizontal		Vertical	
$(0,0)$	$(n,0)$	$(0,0)$	$(0,n)$
$(0,1)$	$(n-1,1)$	$(1,0)$	$(1,n-1)$
$(0,2)$	$(n-2,2)$	$(2,0)$	$(2,n-2)$
...	...	...	...
$(0,n-1)$	$(1,n-1)$	$(n-1,0)$	$(n-1,1)$

Table 1: Endpoints of lines that must be drawn to generate a recombining lattice with n subintervals on the rectangular grid.

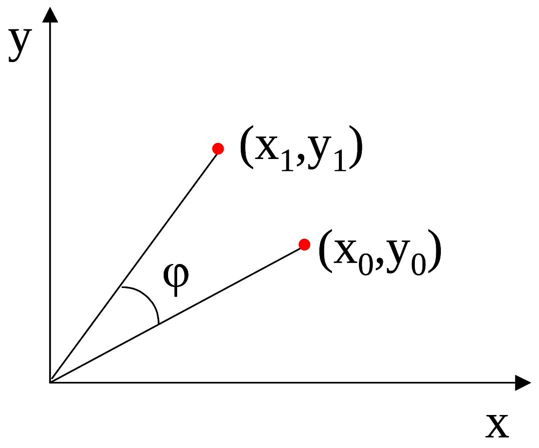

Figure 3: Initial and final position of a point of initial coordinates (x_0, y_0) rotated counter-clockwise around the origin by an angle of φ radians.

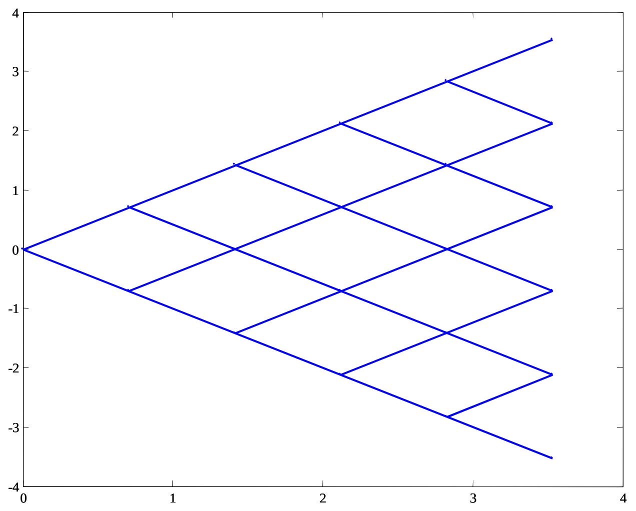

Figure 4: Lattice rotated clockwise by an angle of $\frac{\pi}{4}$; axes not turned off yet.

Since our rotation will be in the clockwise direction, angle $\varphi = -\frac{\pi}{4}$. We thus obtain:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \underbrace{\frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}}_{\mathbf{R}} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

Multiplying the endpoints of lines in table 1 by matrix **R**, we obtain the graphical representation of the lattice, show in figure 4.

The axes are not relevant when representing lattices; we might as well turn them off using Matlab's **axis off** command.

0.1.2 The Second Approach

We will briefly touch upon an alternative approach to drawing the lattice.

The main idea is illustrated in figure 5. As illustrated there, one can move from an "initial" state to the corresponding "up" and "down" state by incrementing the x coordinate by one unit, and the y coordinate by 1 or -1, respectively.

0.1.3 Highlighting Paths

As we saw above, drawing the lattice is simple; highlighting paths will be only slightly more challenging.

We could choose to draw the lines marking various paths so that new lines overwrite lines that have been drawn previously. In other words, the lines drawn later, on top of preexisting lines, would hide older lines. If lines were only pixels on a screen, this might be an acceptable way to highlight paths. As we will see later, Matlab associates

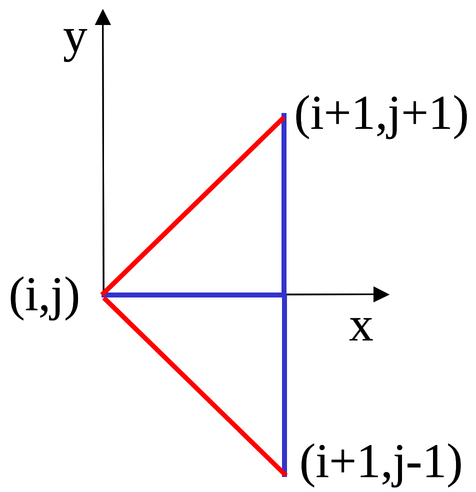

Figure 5: Relationship between the coordinates of the initial state and those of its corresponding "up" and "down" states.

certain internal datastructures to each individual line (and to almost all other graphical elements). If lines had no associated memory structures, then drawing a large number of them would take - perhaps - a very long time, but no resources would be exhausted, as the pixels on the screen would be reused over and over. In Matlab's case, however, creating a large number of lines (or other graphical elements) leads to the consumption of an amount of memory roughly proportional with the number of objects created; sooner or later all available memory will be used up.

Our lattice-drawing problem does not require us to draw so many lines that Matlab would run out of memory, even with path highlighting. We will, however, use the problem of path highlighting to emphasize an alternative that uses a fixed number of lines, and changes the properties of these lines as needed. The specific techniques to achieve this will be discussed in the section on handle graphics (see below).

Each graphical element in Matlab is characterized by a number of properties, among which is color or line thickness, for example. One line can only have one color.

When using the rectangular grid we have drawn our lattice using the longest possible horizontal and vertical lines (see table 1). This makes drawing easy, as we only have to deal with a small number of lines ($2n-2$, for n subintervals), but it makes it impossible to highlight paths by changing the line properties (e.g. color), as in general a path will overlap, if at all, with only a part of the long lines.

We can solve this problem by breaking up the set of horizontal and vertical lines given in table 1 into segments of length 1. For example, the line with endpoints $(0,0)$ and $(0,n)$, will be broken up into n segments with endpoints at $(0,0)$, $(0,1)$, $(0,2)$, ...,$(0,n)$. Any small segment will either be entirely contained in an arbitrary path, or will not intersect the path (except, perhaps, at its endpoints). Thus one can highlight a path by appropriately changing the properties of the small segments that result from the breakup of the long horizontal and vertical lines.

Before we finish talking about paths, let us note that you have to highlight all paths from the initial state to an intermediate or final state, depending on what state you are processing at a certain moment in time. Let us assume that the current state has coordinates (l, k) in the lattice before rotation. Based on our conventions, this node was reached by k "up" transitions and l "down" transitions (note that $k+l \leq n$). All paths to the current node will be part of the "rectangle" with corner points at $(0,0)$, $(0,k)$, (k,l), $(l,0)$. Consider now a point (i,j) in, or on the border of this rectangle. Is this point on the path?

0.2 Handle Graphics

All graphical elements (e.g. lines, axes, figures) in Matlab have an associated handle, which can be used to identify and manipulate the respective elements. From the perspective of the user a handle is just an opaque bit pattern. For uniformity, Matlab interprets these bit patterns as real numbers when displaying them, but one should not try to attribute any specific meaning to these numbers.

We present handle graphics by example, as at this level the main issues are mostly self-explanatory. We will add comments when necessary. The reader is advised to browse the Matlab documentation (“help graphics” is a good starting point) for full details.

```
>>x = 1:10;

% Draw a line, capture handle.
>> h = plot(x, x);

>> h
h = 3.00268554687500

% Retrieve the line’s properties.
>> get(h)
Color = [0 0 1]
EraseMode = normal
LineStyle = -
LineWidth = [0.5]
Marker = none
MarkerSize = [6]
MarkerEdgeColor = auto
MarkerFaceColor = none
XData = [ (1 by 10) double array]
YData = [ (1 by 10) double array]
```

```
ZData = []

BeingDeleted = off
ButtonDownFcn =
Children = []
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [100.001]
Selected = off
SelectionHighlight = on
Tag =
Type = line
UIContextMenu = []
UserData = []
Visible = on

% Get the value of the color property.
% The color is an RGB (red-green-blue) triplet. Each value is
% between 0 and 1, and represents the relative intensity of
% the respective color in the composite color of the line.
% Our line is blue.

>> get(h, 'Color')

ans = 0     0     1

% Change the color of our line to pure green, and increase
% thickness to 3 points.

>> set(h, 'Color', [0 1 0], 'LineWidth', 3);


% Make sure that the new line does not overwrite the old one.

>> hold on


% Plot another line, "forget" to capture handle.
% The new line will be blue, the default color in Matlab.

>> plot(x, 10-x);


% Retrieve all objects of type line that Matlab knows about.
% An array of handles will be returned.

>> harr = findobj('Type', 'line')

harr = 1.0e+002 *
         1.01004028320313
         0.03002685546875


% Retrieve all lines (in this case we only have one) that are
% still blue. This way we can recover directly the 'lost' handle.

>> h = findobj('Type', 'line', 'Color', [0 0 1])

h = 1.010040283203125e+002


% The "tag" of a graphical object is a string that can be used
% to uniquely identity a graphical element. For example, one could
% create tags that represent the coordinates of a line's endpoints.
% The tag can be reused to retrieved the handle when needed, if the
% endpoints of a line are known.

>> set(h, 'Tag', '0 9 10 0');
```

0.3 Useful Matlab Functions

You might find it useful to take a look at the following functions:

```
figure
subplot
print
gca
gcf
plot
plotyy
```

```
line

num2str
str2num

pause
```