Overlay plotting with svplot

Jon F. Claerbout

ABSTRACT

"Svplot" is a plotting utility that runs interactively under Sun Microsystem's "Sunview" and allows hard and soft copy through SEP's traditional vplot utility. Svplot supports *both* hard copy and interactive on-screen plotting of *both* vector and raster information. Further, the vector and raster is handled in a coordinate-system-independent manner that should be readily transportable to future systems. Svplot provides *sv_XXXX* routines that mimic vplot's *vp_XXXX* routines except that the plot appears immediately on the Sun screen and later, on command, in a file for vplot hardcopy. Svplot consists of a header *svplot.h* and four parts, a part *svcoord.c* for coordinate transformations, a vector library *svlib.c*, a raster library *svras.c*, and an overlay management utility *svover.c*.

INTRODUCTION

After investing a great deal of personal time in developing interactive programs to demonstrate hyperbolic overlays, Fourier analysis, and Z-plane filtering, I found I still did not have suitable means of preparing materials for reports and lecture notes. So I have been looking for solutions for myself and the SEP students and staff.

Altogether, I have written about 10,000 lines of C-language code for interactive seismological work on the SUN III color graphics workstation. Of the 10,000 lines written, the last three quarters use the 1000 line svplot package described here to handle all the graphics. Svplot came into existence because although Sunview is interactive it has no way to produce hard copy, hence no way to get plots into reports or lecture notes. SEP's traditional plotting systems, including the most general one, vplot, are not interactive. Svplot is exciting because it handles *both* vector and raster for *both* moving displays and for hardcopy. Commercial systems known to us that come closest to svplot, do not handle raster effectively.

Svplot really was born out of an application program *Balloon* (described elsewhere in this report) in which an interpretive "comic strip" balloon is superposed on a plot (from any of our most important plot formats) The user interactively adjusts the location of the balloon and its pointer. Since then, svplot was used in an application *Zplane* where the operator

moves poles and zeros around in the complex plane and watches the filter, spectrum, and filtered data adjust themselves. Figure 1 is an example of a filter and its complex frequency plane made by *Zplane* and balloon annotation done with *Balloon*. In *ed1D*, the third major application, two one-dimensional functions are displayed which can be edited in various ways. The functions are selectable from Fourier transform pairs, Hilbert pairs, Kolmogoroff spaces, reflection coefficients and impedances or reflection seismograms, etc, or any two of the above functions as well as a few others from one-dimensional seismology. Immediately after the user edits one function, the other is updated consistently.

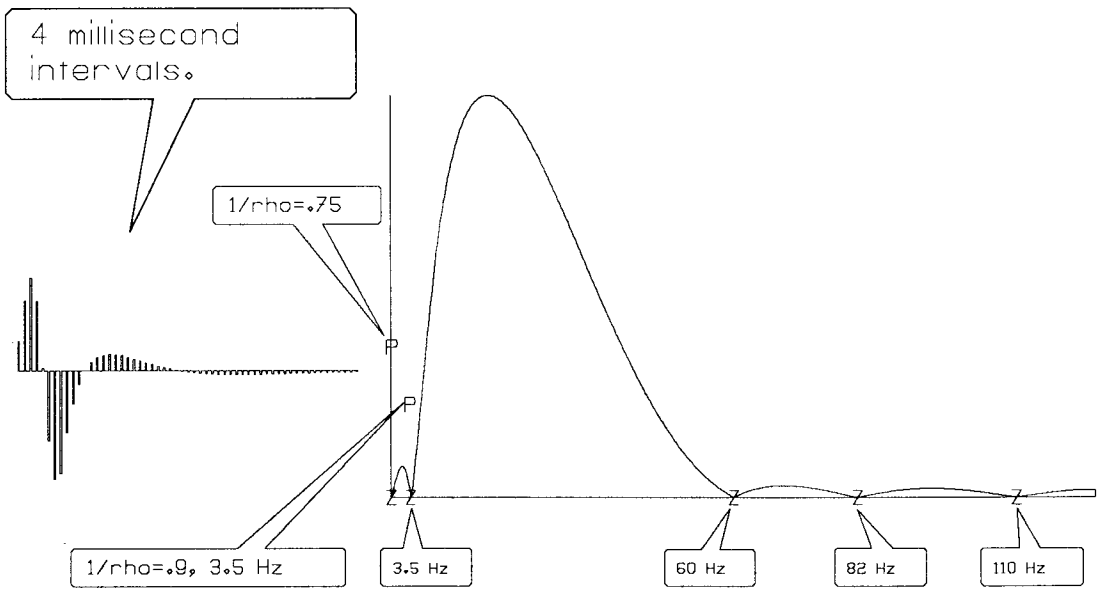

FIG. 1. Repeat of a figure in SEP-56 page 270 with added information enabling readers to duplicate the plot.

Svplot now stands on two narrow foundations, Sunview and Vplot. I would like to broaden these foundations to encompass more devices and more widely available plotting languages. So within a year or so we may expect either or both of these foundations to change, though the direction is not clear, NeWS? Xwindows? Postscript? As a consequence, care is being taken to protect svplot's users from unnecessary future disruption. This mainly means designing complete coordinate-system independance.

Left handed systems, etc.

Consider the placement of European language text. The origin is in the upper left corner. Applying mathematics to the placement of text, the 1-axis is the horizontal placement of letters on a line, and the 2-axis gives the lines down the page. This coordinate system is "left handed". The television scan line system is likewise, and hence are Sunview, MacIntosh, and most video systems. A mathematical matrix always has A_{11} at the upper left, and the 1-axis (first subscript) points down. In reflection seismology, the 1-axis generally points down from the surface of the earth. Despite these important systems with the origin in the upper left, the preponderance of applications have origins in the lower left. I do not expect the ideological struggle between these systems to be resolved in my lifetime. In self-defense we should vigilantly avoid writing code that is coordinate-system dependent.

(Besides the two dominating systems described in the previous paragraph, is another right handed system with the origin in the upper left hand corner and the 1-axis pointing down. That system (called "rotated" at SEP) is a natural one for maintaining consistency between paper devices that are infinite in one dimension (paper on a roll) and terminals that scroll the screen upward. Since the advent of laser printers and nonscrolling workstation screens, the "rotated" system is on the decline, though it is still with us in valuable older software.)

A plot utility need not dogmatically choose a coordinate system and force users to adapt to it. A plot utility should *be* a utility and serve the people by finding out what coordinate system they want to use. In *svplot* the user chooses the coordinate system, and can change it dynamically.

Need for floats

On a raster system such as Sunview it is tempting to do the graphics work with integers, it should be faster, the cursor location is presented to you as integers, and all the drawing routines take integer arguments. But integers are a trap. When time comes to make a hard copy of your plot, you have lost the possibility of the higher precision of laser printers. So the plot utility *svplot* uses all floating point arguments for locations.

COORDINATES

It is important to distinguish between coordinate transformations and coordinate systems themselves. If you have N coordinate systems, then there are $N(N-1)$ transformations among them. For large N it is obvious that you want to define the coordinate systems, and then derive the coordinate transformations. This is simplest and ensures consistency (thereby easing maintenance). But this is not the philosophy of Postscript or Vplot or GKS, or HyperCard. They give you a default coordinate system and, if you are lucky, they tell you how to design a *transformation* of your application coordinates to fit theirs.

Definition of coordinate triad

Given cylindrical coordinates (r,θ), algebraically r is the 1-axis and θ is the 2-axis. *Algebraic coordinates* are defined by the order that components are presented. Algebraic coordinates have no geometric properties. Further, algebraic coordinates are programmers' coordinates. They are often positive integers such as those that range over loops, or range over indices of arrays.

Consider two-dimensional linear coordinate systems. To convert a vector from one such coordinate system to another you need a 2×2 matrix for rotation and deformation and you need a two-component vector for translation. In all, the transformation requires six numbers.

(Some users might wish to restrict these six numbers in various ways, to ensure isotropy, etc, but not all users should be forced to make isotropic plots. In fact few plots need be close to isotropy and for some plots, the seismic (t,x)-plane for example, there is no concept of isotropy. There should be a straightforward way to enable some users to have isotropy without requiring all users to draw isotropic plots. Svplot ignores the isotropy issue.)

We all know what we mean by *coordinate transformation,* but what do we mean by

a coordinate system? Since we are talking about plotting, not mathematics, we mean coordinates on a rectangular screen or page where the *boundedness* of the rectangle is a central aspect.

A *coordinate triad* ties together the algebraic concepts of coordinates with the geometrical concepts. Presumably, a coordinate triad is defined at the beginning of a program (or large block of code) and all required information about geometry is drawn from that triad and from nowhere else.

We now define a *coordinate triad.* A coordinate triad is defined by three geometrical points, the top left corner, the bottom left corner, and the bottom right corner (named t, o and r). There is no presumption that the origin is the bottom left corner! To specify a coordinate triad, each of these corners must be given in algebraic coordinates, i.e. a value must be given for the corner's 1-axis and another value for the corner's 2-axis. We saw that the coordinate transformation took six numbers to specify and now we see that the coordinate triad is also specified by six numbers.

Given two triads (twelve numbers) the program *coordinate.r* finds the transformation (six numbers) between the triads. The method is this: Let one triad be three (x, y) pairs and the other triad be three (u, v) pairs. The transformation for the triads, indeed, for the whole space is

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} A_{xx} & A_{xy} \\ A_{yx} & A_{yy} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} s_x \\ s_y \end{bmatrix} \tag{1}$$

Take the transpose

$$[x\ y] = [\text{u v 1}] \begin{bmatrix} A_{xx} & A_{yx} \\ A_{xy} & A_{yy} \\ s_x & s_y \end{bmatrix} \tag{2}$$

Stacking the row vectors in (2) on top of each other for three (x, y) and (u, v) pairs gives two sets of 3×3 equations solvable for the transformation, namely for (A_{xx}, A_{xy}, s_x) and for (A_{yx}, A_{yy}, s_y). These equations are:

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} = \begin{bmatrix} u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \\ u_3 & v_3 & 1 \end{bmatrix} \begin{bmatrix} A_{xx} & A_{yx} \\ A_{xy} & A_{yy} \\ s_x & s_y \end{bmatrix} \tag{3}$$

Before using any svplot plot routines, you must notify the svplot package of your screen dimensions by means of a call to *svcoord_init(n1,n2)* where $n1$ and $n1$ are frame dimensions. This initialization routine first defines triads from some important systems, namely:

```
Cvplot,     /* vplot's "STANDARD" coordinate system */
Crotated,   /* vplot's "ROTATED" coordinate system */
Cportrait,  /* vplot in portrait mode */
Csunv,      /* Sunview coordinates */
```

Then defaults are prepared for two essential coordinate transformations, *Ltube* and *Lhard.* *Ltube* is the transformation from algebraic coordinates to the screen. The default is for

Sunview programmers, so for Sunview programmers it is an identity transformation that *svcoord_init(n1,n2)* makes with

get_transform(&Csunv, &Csunv, &Ltube)

The other necessary coordinate transformation *Lhard* is the one from algebraic coordinates to the output vplot file. It is made with

get_transform(&Csunv, &Cvplot, &Lhard).

The *Balloon* application program allows for hardcopy in portrait mode by making the output transformation by

get_transform(&Csunv, &Cportrait, &Lhard)

The source code in *svcoord.c* contains various other routines for handling coordinate transforms and triads. About the only one of general use is *svcoord_iso_init(nx, ny)* which ensures isotropy of your final plot when *ny* is not 75% of *nx*.

This package needs more work to give you a coordinate system for all seasons. For example, you may interactively move a box around a screen and want an automatic means of getting various coordinate systems that refer to that box. At the present time much of the required code is buried in balloon/box.c but it is not yet been extracted for public use.

RASTER

Initially, my raster package could support all angles, but in the interest of efficiency only 90 degree alignments are currently supported.

Cube indexing

Cube indexing is a subject slightly outside the realm of plotting, but it suggests that the array accessed by the raster plot routine should not be required to be "tight packed" or "stored by columns," (or by rows) but stored more generally. I interpret the *ordinates* of a cubic array by:

$$A(i1,i2,i3) = A[\ s0 + s1*i1 + s2*i2 + s3*i3]$$

The array operations: transpose, rotate, slice, reflect, subcube or window, can all be accomplished by manipulations on the seven parameters (n1,n2,n3,s0,s1,s2,s3). The data itself need not be touched. For the two-dimensional raster plot routine I reduce the seven parameters to (n1,n2,s1,s2). Sequential access uses incrementation instead of fixed point multiplications, so besides being general, and economical of storage, this method is fast. Notice that when the k-axis (k = 1, 2, or 3) is reversed, the value of sk (s1,s2, or s3) is made negative after s0 is incremented by sk(nk-1). Further details are in the *gapaxis.c* source with the *Balloon* application.

Raster calling sequence

The vplot raster routine that *svplot* builds upon has a blatant coordinate system de-

pendence in the argument list. Its argument list specifies (x,y) coordinates for the *lower left* and *the upper right* and an *orientation* and an *axis reversal* parameter. (It is confusing too). This is six numbers in all. Instead of that six, *svplot* uses six others, the coordinates of three points, namely, the location for the Fortran array element A(1,1), the location for the array element A(n1,1), and for A(1,n2). I should point out a minor disadvantage of svplot—the user could specify a nonrectangular region. So I interpret the given arguments as the outer bounding rectangle of whatever is given.

The available routines in the raster package *svras.c* are:

sv_ras(x0,y0, x1,y1, x2,y2, n1,n2, s1,s2, raster)

sv_wiggle(x0,y0, x1,y1, x2,y2, n1,n2, s1,s2, raster, dir)

The three points (x0,y0), (x1,y1), (x2,y2) bound a rectangular region that will contain the raster bytes. The raster matrix is accessed as "raster[s1*i1 + s2*i2]." The three points correspond with the raster as follows:

raster[s1*i1] for $0 \leq i1 < n1$ plots from (x0,y0) to (x1,y1).

raster[s2*i2] for $0 \leq i2 < n2$ plots from (x0,y0) to (x2,y2).

As stated earlier, s1 and s2 may be negative. The wiggle-trace routine is like the raster routine with the additional argument "dir" where

if(dir == 1) traces run on 1-axis.

if(dir == 2) traces run on 2-axis.

An example is in Figure 2.

VECTOR LIBRARY

In *svlib.c* are user callable subroutines named *sv_XXXX* that mimic a few of SEP's traditional vplot routines vp_XXXX. Internally each *sv_XXXX* subroutine fits the pattern:

- Convert coordinates to Sunview integers.
- mimic spen, ie put the image on the screen.
- if either of the global variables *sv_save* or *sv_animate* are set, make hardcopy by calling *vp_XXXX*

The available subroutines are few, namely:

svlib_init(pixwin)

sv_vector(x0, y0, x1, y1)

sv_move(x, y)

sv_draw(x, y)

sv_text(x, y, size, orient, textstring)

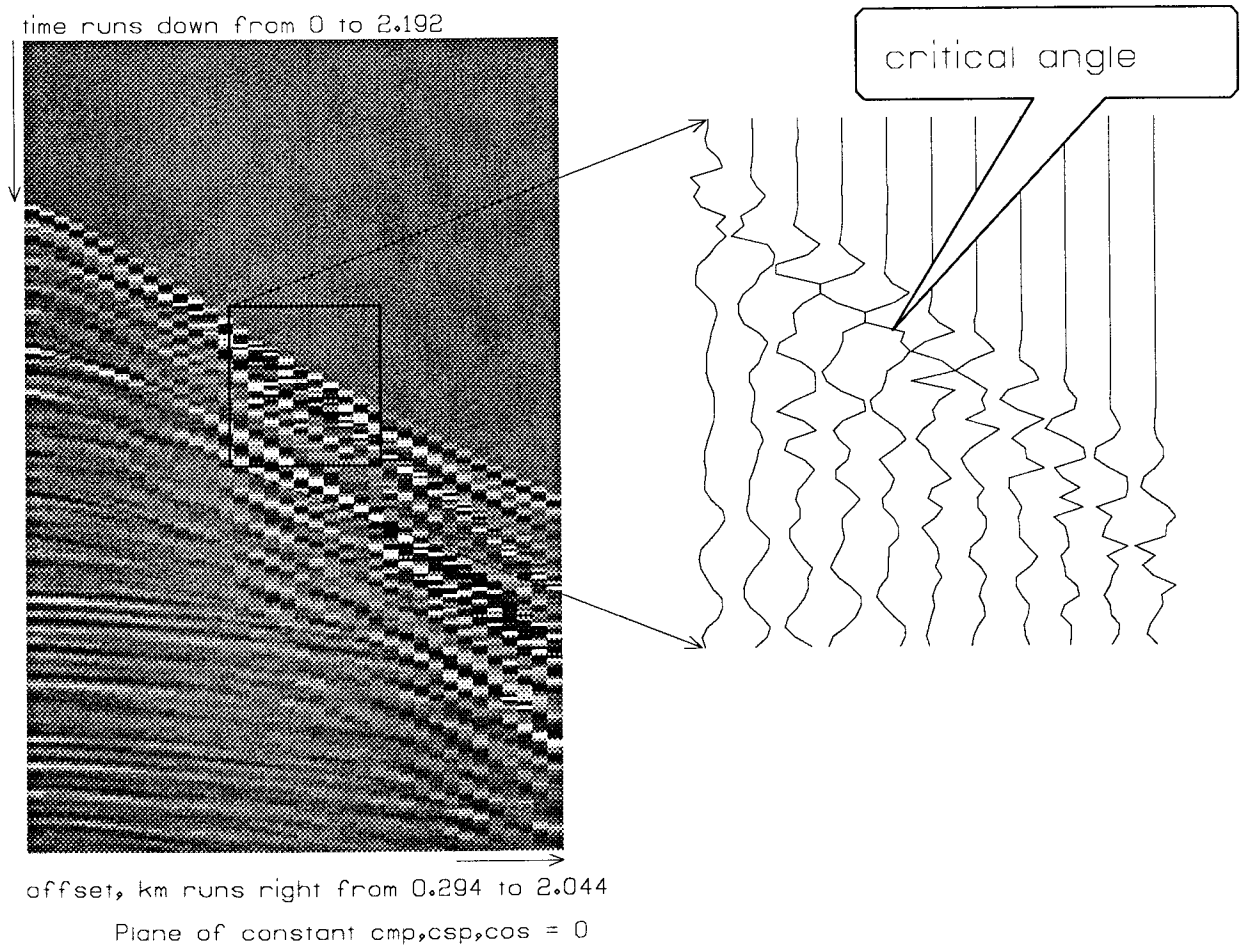

FIG. 2. An example of raster displayed in both typical variable density form and in wiggle trace form. The wiggle trace form is most practical in a zoom where the number of traces is few.

sv_fat(ifat)

sv_area_kludge(x4, y4, n, x, y)

sv_color(my_color)

The routine *svlib_init* must be called first to specify the window on which the plot is drawn.

Vector, move, and draw routines work exactly as you expect.

Sunview doesn't have various sizes and orientations of text, so those arguments of *sv_text()* are ignored on the screen, but honored on the vplot hardcopy. Ideally, I would eliminate the orientation angle parameter from the text routine and instead specify direction with a vector. Then plot files would rotate and translate more sensibly. Unfortunately, neither Sunview nor vplot gave me adequate hooks to do this.

Vector fattening applies to the hardcopy only.

Sunview doesn't have an area command, so I kludged one up by drawing lines. It only works for four-cornered polygons and triangles, so *sv_area_kludge()* has limited ability on the sun screen, but the hard copy is pure vplot. The arguments x4 and y4 are four-component vectors that define the four cornered polygon, (or triangle if one corner is repeated). The arguments x and y are the usual arrays defining all the n corners of a general polygon.

Color handling is more complicated as described in the section on overlays. The extra complexity has to do with making it possible to move plots by redrawing them with color number zero.

OVERLAY UTILITY

An overlay is any plot that can be changed by user interaction. So it is a plot that is repeatedly drawn and erased. I have written about fifty overlays in four large packages, and the common threads are in *svover.c*. The application program *Balloon* had 21 overlays at last count. The file *svover.c* is considerably shorter than this explanation of its function, so any questions you may have are perhaps best answered by reference to the code itself.

User programs invoke *sv_show()* to see an overlay and *sv_hide()* to make it to go away. User code contains sprinklings of *sv_hide()* and *sv_show()* and the pattern (hide; change parameters; show). The programmer's discipline is to avoid the pattern (show; change parameters; show) because multiple copies of an overlay are considered an error.

Successive calls to *sv_show()* or *sv_hide()* are permitted, indeed they are hard to avoid with the operator of an interactive program being permitted to push buttons and move the mouse at will. What happens then is this: If *sv_show()* is called several times in succession, only the first call draws a plot, likewise with *sv_hide()*, only the first call does the actual erase. This design was partly for efficiency, but mainly to allow draw and erase with the *exclusive-or* function.

Both *sv_show()* and *sv_hide()* invoke the same user routine "hit". When the user initializes the overlay structure he/she loads into "hit" a pointer to his/her overlay routine (which calls *sv_draw()*, etc). The user's hit routine can find out if it was called by *sv_hide()* or *sv_show()*, but I found that my hit routines almost never bothered to do so because the process of erasing is usually the same as that of drawing.

Most hit routines do not contain color calls (rainbow=0). Draw and erase colors are specified in the overlay structure and they are taken care of by *sv_show()* and *sv_hide()* if the overlay is monochromatic (not rainbow). The hit routine of a rainbow overlay may call *sv_color()*. If so, colors are overridden automatically during *sv_hide()* which erases by writing with background color. In the *Balloon* application program, many hit routines start off by defining *Ltube* and *Lhard*. Then they call *sv_draw()*, *sv_text()*, etc.

To get started you load a structure defining the overlay type. I usually start with the routine *generic_overlay_init()* in which are found

```
struct Overlay *ov; {
ov->hit     = nohit; /* pointer to activator routine */
ov->on      = 0;   /* image on screen now ? */
ov->id      = 0;   /* integer identifier (so hit() can know who called) */
ov->rainbow = 0;   /* contains color calls?  no*/
ov->vphot   = 6;   /* vplot desired foreground color */
ov->vpcold  = 0;   /* vplot desired background (erase) color */
ov->sunhot  = XOR; /* sun foreground color, HOT or XOR */
ov->suncold = XOR; /* sun background color, COLD or XOR */
ov->sunsave = HOT; /* usually HOT, sometimes COLD, never XOR*/
ov->raster  = 0;   /* is the overlay vector or raster? */
}
```

For most of my overlays, I use all the default parameters except that the "hit" routine obviously must be specified. Several different overlays could share the same hit routine which is the reason for the 'id' variable. For example *Balloon* has three overlays that are deformable boxes. All share the same hit routine, so theoretically all three boxes could be on the screen at the same time.

HOT and COLD refer to the Sunview representation of SET and RESET.

The way you get a hard copy is by setting the global variable *sv_save* nonzero and then invoking *sv_show()*. Then set *sv_save=0* so all subsequent moving overlays do not go to the hardcopy file too! A value *sv_save=1* denotes a copy goes to the offline file. A value *sv_save=2* denotes an additional indelible copy is left on the screen.

The exclusive-or function XOR

Overlays often come in families. For example the text filled balloon is written in three parts, the area fill behind the outline, the outline, and the text. Another example is a vplot with a resizing box around it. You need to think about the interference of family members. A problem with SET/RESET overlays is that when you take one off, you also remove anything underneath that was overlapped. This doesn't happen with the exclusive-or function (called XOR). XOR overlays can be put up in any order and taken off in any order and any original image remains perfectly preserved. Rick says that Xwindows doesn't support XOR and there you need to draw overlays with SET and erase them with RESET.

A family of SET/RESET overlays can also be drawn in any order and erased in any order. But if you wish to mix XOR overlays with SET/RESET overlays, then the SET/RESET overlays must be put down before the XOR overlays. Taking them off must be done in the reverse order. Within the SET/RESET group and within the XOR group, order is irrelevant.

Because SET/RESET overlays tend to mutual destruction, things like wire frames (set on a complicated overlay to show it can be moved or resized) must be XOR or they must lie totally outside the object or else they must be kept on a separate plane that can somehow be simultaneously viewed. My application programs choose overlay order and positioning so as to minimize aggravation should we ever convert it to Xwindows. They do not use extra planes for overlays but I expect to change this if I convert to Xwindows.

When the time comes to make hardcopy plots, the philosophy of overlay ordering changes. Instead of choosing the order for the cleanest move, the order is chosen for the best looking final copy.

When it isn't an overlay — tweaking global variables

Of my applications 90% fit the pattern described above. Sometimes I fool *sv_show()* and *sv_hide()* by tweaking global variables. For example a pole-zero plane contains many small letters 'p' and 'z'. The overlay package is designed to prevent the screen from filling up with multiple copies of an overlay, so my Zplane *program* overrides "on". Retrospectively, I notice that an overlay is really just a dozen or so values in a C-language structure and I think I should have made a vector of such structures to handle the many little letters.

Sometimes you are editing a tiny part of a complicated plot. It is too slow to redraw the whole plot after each minor change, so again, you don't have what I formally call an overlay. Maybe you can split your complicated plot into overlays, or you can forget about my overlay package, mess around in pure Sunview, do a global screen erase and then redraw as an overlay (to solve your hard copy problem). This happened with my "push" tool in the one-dimensional signal edit program.

Raster is a bit slow and it happens that when time comes to erase, you have already prepared a massive panel that needn't be prepared again. So Balloon's hit_raster() looks to see if it was called by *sv_hide()*. It does a test "if(sv_caller == SV_HIDE)". To speed raster, I added another global variable *sv_movie* which when set tells the overlay package that raster erases will be done by the subsequent *sv_show()*. Besides speed, an advantage of erasing with a subsequent *sv_show()* is to avoid an annoying screen blink.