**VSG**: Easy python drawing- You decide where to put shapes and text and when to show or save them, VSG does the rest

VSG ©2009-2022 Andrew Fire and Stanford University. Please send all bug reports to afire<rat>stanford.edu (change the cute rodent reference to '@' before sending)

As of 12/31/22 current implementation of VSG is VSG\_ET, requiring Python v. 2.7+. To write/display raster files, install Python Imaging, GraphicsMagick, or ImageMagick.

Minifesto: the entire description and guide for any software should fit on a single page (2-sided), which you're reading. CurrentVersion: https://github.com/FireLabSoftware/VSG\_Module

**Introduction:** VSG is a package designed to allow rapid visualization of experimental data on a two dimensional canvas. Individual drawings are a combination of shapes (rectangles, circles, polygons, arcs), lines, and text. VSG is not "Photoshop" (i.e., not designed to work with photographic images.) Rather VSG is designed to facilitate abstract visualizations of discrete/analog data. The main VSG interface for creating objects is a series of python routines described below. Each shape or text element is created by a single command such as vtext(text='Hello World'), then vdisplay() is called to show the canvas. VSG can also be used by creating a text file with a series of commands (advanced feature 4 below).

**First:** from VSG\_ModuleET import \* This is needed to use VSG. If you get an error, put a current VSG version (VSG\_Module#.py) in the same directory as your script.

**Quick Start:** To make a picture ("data visualization") with VSG, you will give a list of commands to create objects, then ask VSG to display them. You don't need to give any initial commands to set up the canvas and you don't need to decide how big the canvas will be.

**Prequel: two VSG code examples** (for this doc**, red: VSGprocedures, orange: VSGparameters, blue:user values, magenta: global variables):**

vrect(x1=11, y1=17, width=20, height=5, fill=red, stroke=blue, strokewidth=2); vdisplay()

Displays image of a rectangle (top left= (11,17), width=20, height=5, fill=red, 2-pixel blue boundary)

vcircle(fill=red, xc=11, yc=17, r=10, xlink="http://www.stanford.edu"); vwrite('mycircle.svg')

Write an svg file with a circle (center=(11,17), radius=10, fill=red, live web-link (http://www.stanford.edu))

**Procedures in VSG (after you import VSG, you have a blank canvas, this then gets filled as you add shapes and text):**

**Add Simple Objects**: vline(parameters), vrect(parameters), vcircle(parameters), vellipse(parameters), vsquare(parameters)

**Multipoint Objects**: vpolyline(parameters), vpolygon(parameters), varc(parameters) **vconnect**(list,params,cjust=): join objects in list w. polyline

**Text Objects**: vtext(text=" favorite text", params [see below])-- text can be >1 line. cjust: corner/side of objects to join (**c**,n,e,s,w,ne,sw,se,nw)

**Basic Annotation**: vtitle(<text='..'>), vlegend(<text='..'>) Title (above) or legend (below) current canvas. Omit text for default title/legend

**Make a human-readable grid or colorkey**: vgrid(<parameters>), vcolorkey(<parameters>)

**Display or save a drawing**: vdisplay(), vwrite('myfile.svg'). Some display features can work with versions of **ipython notebook -inline**

**Specifying position and shape: <**Units default to an assumption that 1 unit is 1 pixel (i.e. x1=11 puts the left edge at pixel 11). >

**Horizontal position** and **width** of any simple object are specified with a combination of x1,x2,xc,xr,width:

x1=Left edge, x2=Right edge, xc=horizontal center, width=width, xr=width/2

**Vertical position** and **height** of any simple object are specified with a combination of y1,y2,yc,yr,height:

y1=Top edge, y2=Bottom edge, yc=Vertical center, height=height, yr=height/2

**arc** ('**pieslice**') objects: x and y values (above) define the "whole pie", you need to specify what part of the pie is in the arc.

Two values define this, a0 is the fraction of the pie (clockwise from top) to start; ad is the fraction of the pie included in the arc.

So a0=0, ad=1 gives the whole pie, a0=0.25, a1=0.5 gives the lower right quadrant.

To position **polylines** and **polygons** need a series of points joined together. Use a list of alternating x and y values or (x,y) tuples

points=[10,20,30,60,60,30] : defines triangle: (10,20) to (30,60) to (60,30). points=[(10,20),(30,60),(60,30)] does same.

To position **text**, you define a single x and y value that will be used as an anchor. This can be where the text starts, ends or is centered.

x1=val: start of text at position x=val | x2=val: end of text at position x=val | xc=val: center of text at position x=val

y1=val: bottom of text at position y=val | y2=val: top of text at position x=val | yc=val: center of text at position y=val

So vtext(text='Do you have any cheese?',xc=12,y1=10) places text centered horizontally at x=12 with vertical alignment of the lower edge at y=10

**Borders:** You can set the border of a shape (rectangle, ellipse, line) to any width**;** strokewidth=3 gives a border width of 3

**Specifying color:** For vrectangle/vellipse/vpolygon, you can specify a fill=color and stroke=color. vline/vtext has only stroke=color

Common colors: orange,black,blue,gray,green,lime,maroon,purple,red,white,yellow,brown,magenta,cyan. Quotes allow broad range of colors (e.g. 'pink', 'olive')

fill=rgb(r,g,b) | stroke=rgb(r,g,b): r, g, and b are red, green, blue integer values from 0 to 255 (dark to light in each color channel)

fill=#rrggbb | stroke=#rrggbb: rr, gg, bb are two digit hex numbers specifying the three channels (also #rgb is shorthand for #rrggbb)

fill=val | stroke=val: 0.0<val<1.0 gives linear heat-map (0.0: black, 1.0:white, rainbow in between). Integers 0-1088 give a log-range.

fill="string"| stroke="string": A random color will be assigned (the same random color will be used for any object with color="string")

**Formatting text**: For vtext, vtitle, vlegend: font="font size weight styles" (e.g. "Times 12 Bold") defines font, size in points, styles: bold, italic

Some usable fonts= Verdana, Courier, Helvetica, Times, DejaVu. VSG will guess if your font is not found. delimiter=<'|' >: line splitter in vlegend

**Label** any object with label='label\_text'. Options: labelcolor='color', labelfont='font', and labeljust='justify position' (can be n,s,e,w,ne,nw,sw,se,c)

**Displaying and Saving**: vdisplay() displays the drawing, vdisplay('myfile.svg') specifies a file name. vwrite: synonym for vdisplay

Files can be saved in several formats based on the extension in filepath (e.g., .svg, .ps, .png, .jpg, .tif, .pdf). File type also sets display mode:

'.svg' pictures displayed using web browser (high resolution, zoom/pan/scroll active, text-rotation functional, partially interactive).

'.html' pictures displayed using web browser ('html5': med. resolution, zoom/pan/scroll, all objects rotatable, mouse responses not yet implemented)

'.ps' pictures displayed using "TK" tool. (high resolution no zoom/pan/scroll, no text-rotation, interactive)

'.jpg','.tif','.png': pictures displayed using standard picture viewer (medium resolution, zoom/pan/scroll, some text rotation)

Some additional features for .jpg/.tif/.png display (e.g, rotated text) are added if you can install free ImageMagick (link) or GraphicsMagick (link) packages on your system.

If no temporary filename is given, file is saved as "temp.svg" and viewed with browser. Call vwrite or vdisplay with <display=False> to save a file without immediate display.

**You don't need to know this, but.. VSG** can use several different packages to render your shapes as images. These are usually chosen automatically but can be engaged specifically if you wish, using

vTKwrite('fff.ps') <displayed using the TK package, saves only .ps files>, vTKwrite(live=True) continues to update a single TK drawing at each call (no mouse actions in this case)

vSVGwrite('fff.svg') <displayed using the web browser, saves only .svg files>, vHTML5write('fff.html') <displayed using the web browser, only .html files>

vMVGwrite('fff.ext') <rendered by GraphicsMagick or ImageMagick, displayed with platform-standard picture viewer; any graphic format (ext=jpg,tif,pdf,png, etc)

vPILwrite('fff.ext') <rendered by Python Imaging Library, displayed with platform-standard picture viewer; almost any graphics format (ext=jpg, tif, pdf, png...)

**Your canvas grows as you add objects:**

The canvas starts out with the first object will grow to accomodate all added objects. Negative numbers are also allowed.

The variables VSG.xmin, VSG.xmax, VSG.ymin, VSG.ymax give the extremes of the canvas at any given time.

You can specify these values to have a minimum canvas size. Setting VSG.autogrow=False (or stet=True for single objects) turns off the autogrow feature

**Adding weblinks and info-display to objects** (works only in '.svg' or '.ps' mode, with .ps mode giving the most interactive display)

popup="my popup text" clicking object pops up some text, e.g., popup="hey you clicked on my object"

detail="my details" control-clicking object pops up some text, e.g., detail="here are more details"

xlink ="http:my\_link" shift-clicking object opens a url site: e.g., xlink=http://firelab.stanford.edu/

mouseover="my mouseover text" mouse over object pops up some text, e.g., mouseover ="hey you moused my object"

**Some additional things that can be set for SVG files** (these will also work on some systems using GraphicsMagick)

strokeopacity=opa fillopacity=opa opa is a real number between 0.0 (transparent) & 1.0 (opaque)

rotate=angle rotate text by angle: uses the positioning anchor (e.g. upper center for xc,y2) as the center for rotation)

**Advanced Feature 1: Multiple canvases:** Individual drawings are called canvases. One way to make several drawings in a script

Make a first drawing, display it, call vclear(), make a second drawing, etc. Alternatively you can have several active canvases at once

When you import VSG, a single canvas (called "VSG") is created allowing you to place shapes and text. The vxxx() procedures above

(vtext(),vrect(), etc) add individual elements to the primordial "VSG" canvas, this can also be done by VSG.vtext(),VSG.vrect(), etc

To create an additional canvas named "mycv", use the following 'constructor' syntax: mycv=VSGcanvas()

To, e.g., add a rectangle to mycv: mycv.vrect(x1=12,x2=24,yc=10,yr=4); To write/display "mycv": mycv.vdisplay('mycv.svg')

mycv1+mycv2: new canvas overlaying mycv2 over mycv1 | mycv.transform(xs=0.5,ys=1.1,xd=9,yd=-8) yields a copy of mycv scaled by xs,yx, then translated by xd,yd

VSGmontage([cv1, cv2,cv3,cv4], optional params): returns a montage of canvas list; Params:columns=n; align='UR', 'LR', etc (Up,Mid,Low; Left,Right,Cen);PanelBuffer=n (pixels between canvases)

**Advanced Feature 2: Shortcuts to specify position and shape**

r=r sets both the x and y radii for an object (e.g. for squares or circles)

dx=dx, or dy=dy sets the x or y position of the current object relative to the previous object (with distance dx or dy)

packabove=cl | packbelow=cl move object up/down until the nearest other object is cl pixels away. cl=real: precise; cl=int: rough

packleft=cl | packright=cl move object left/right until the nearest other object is cl pixels away.

If you don't specify an x position (as either x1, xc, or x2) the object will go just to the right of the last object

If you specify an x position but not a y position, the object will extend the bottom left of the drawing

If you specify x and/or y position but not shape (ie no radius, width, etc) the object will take its width/height/shape from the last object

Partial specification of objects often succeeds, but be aware that cavases inherit stroke,fill,strokewidth,font,..from the previous object unless stet was set to True for that object

If you add an object with unusual properties (font,fill, etc), you may need to respecify properties for subsequent objects so they are not also altered.

**Advanced Feature 3: Changing global settings on your canvas** (Several canvas parameters can be changed manually with vset(key=value))

Commands below change the main VSG canvas (e.g., vset(bg=2)), while mc.vset(bg=2) changes user canvas 'mc'

vset(bg='red') : Sets background color (default=gray) | vset(drawtop=False) : True (default)=new items on top, False=on bottom.

vset(margin=9) : border on edges of drawing (default=10) | vset(bd=1) : width of a border on tk application rendering (default is 2)

vset(scale=4.0) : Set drawing scale at 4 pixels in final image per x and y unit (default scale=1.0 for vector (SVG and ps) and 3.0 for raster)

vset(xmin=10) (or xmax,ymin,ymax): set min/max values for x, y. These values still adjust automatically, so a canvas includes all elements that have been added. autogrowh and autogrowv control growth specifically in the horizontal and vertical directions (or set stet=True for individual objects to prevent canvas growth).

Drawing order can be forced by setting 'priority=n' for any object. Default is 0 (draw in order of creation). priority<0 are drawn earlier (bottom layers), priority>0 drawn later.

**Advanced Feature 4: Using a batch file instead of python commands to generate a drawing**

You don't need to program python to use VSG: Any file with a list of entries with format "!VSG<command param1=val1,param2=val2...>" can be used to build a VSG canvas.

To draw two rectangles on a canvas and show them using svg, !VSG<rect x1=10,y1=10,xr=5,yr=3> !VSG<rect x1=20,y1=20,xr=3,yr=5> !VSG<vwrite 'temp.svg'>

To operate on a canvas other than the primordial canvas VSG, put the name of that canvas after the ! (it will be created if it doesn't already exist).

VSG commands in a file on the same line or separated. Parameter assignments can be separated by spaces or commas. Avoid "=<>'," in strings to prevent ambiguity.

Running "python VSG\_Module#.py F1path" from command line renders VSG commands from file F1path, then exits. To run in true batch mode, set display=False in vwrite()

**Experimental Feature 1: vgrid()** : a basic one-size-fits all procedure for annotating drawings and graphs.

Calling vgrid() after placing a set of objects draws axes and labels that are a 'best' guess at what you are plotting.

The default for vgrid() is to draw axes based on pixel coordinants used to specify individual objects

If your pixel coordinants are related by a scale (or log) factor to real world measurements, you need to tell vgrid this.

This is done by adding xg= and/or yg= definition to the points in the code where you define objects.

As a default, an object drawn as vellipse(xc=10,yc=20,height=5,width=5) will be assigned axis values x=10,y=20.

If this point corresponds to real-world x=100 and a y=2, use vellipse(xc=10,yc=20,height=5,width=5,xg=100,yg=2)

vgrid() parameters, all optional: at minimum I recommend providing x, y axis labels (gxlabel,gylabel), a graph title (gtitle), and choosing log vs linear scales (gxlog,gylog)

gxlabel="My X Axis": label for x axis; gylabel="My Y Axis": label for y axis

gtitle="My title" title of graph (default label will give information about open files, todays date and time, and the name of your script)

gxlog=True: log scale on x axis (=False for a linear x axis) | gylog=True: log scale on y axis (=False for a linear y axis)

You can also specify various color/font/width parameters with vgrid. Assignment uses same syntax as with simple objects (e.g. glabelfont="Times 12", glabelxcolor=red) ]

gaxisfont,gaxisfontsize: for drawing numbers or other divisions on axes; gticklength: length of ticks on x and y axis

glabelfont,glabelfontsize,glabelcolor: for drawing labels on axes | gborderwidth,gbordercolor: outer border rectangle width/color

glabelxcolor,glabelycolor: colors for x and y axis; gtitlefont,gtitlefontsize,gtitlecolor: for drawing graph title

gmajorwidth,gmajorcolor:width and color of major grid lines; gminorwidth,gminorcolor: width and color of minor grid lines

gxnumrotate, gynumrotate: angle (degrees) to rotate axis numbers. Works with svg, not ps. Will work with jpg,tif,png (only) if graphicsmagick is installed.

gxmajor=True draw major x grid lines (=False : just ticks); gxminor=True draw major x grid lines (=False : just ticks)

gymajor=True draw major y grid lines (=False : just ticks); gyminor=True draw major y grid lines (=False : just ticks)

gxgrid=False: skip the X axis; gygrid=False: skip the Y axis; gxmin=(xminValue, xminCanvas) Specify Min x value, Min x canvas position (similar: gymin=)

gxdom: parameter that identifies the descriptive "x value" for any given object (gxdom="xc" for middle, "x1" for left, "x2" for right).

gydom: parameter that identifies the descriptive "y value" for any given object (gydom="yc" for middle, "y1" for bottom, "y2" for top)

There are two possible components (or modes) to a colorkey.

(i) a text-based "color list" indicating associations of individual colors each with one "real world" feature (e.g. **o** Apples , **o** Oranges , **o** Grapes), and

(ii) a graphic spectrum (e.g. a heat map) with a continuum of colors corresponding to a range of numerical values (e.g. -10**ooooo**0**ooooo**+10).

You can have both modes (i) and (ii) in the same drawing (one will be placed just above the other)

For either mode, the main requirement is to specify during the generation of objects which color to associate with what name or value range.

For a text-based color key, you can specify a text tag using the assignment colorkey='<string>' when making objects.

(e.g., to assign "ET" to the color of a new rectangle, we add the clause colorkey="ET" to the creating command: vrect(x1=10,width=20,yc=0,yr=15,fill=red,colorkey="ET")

For objects assigned two colors (a stroke and a fill) when created, you'll need to specify which is associated with the key. Use strokekey='ET' and/or fillkey='ET' in this case.

To make a colorkey spectrum you'll need to specify how a range of numerical values are being mapped into color space.

If you use VSG's ability to define a color for each number (with commands like fill=0.5), then vcolorkey() will default to a colorscale based on number/colors that are used.

If you have scaled your numeric range to fit the built in range, use colorindex=value to specify a 'real world' number associated with the assigned color during object creation.

e.g. for a situation where values are mapped in a 10:1 ratio onto the built in integer (1-1088) color list, use vrect(x1=10,width=20,yc=0,yr=15,fill=100,colorindex=10)

If both fill and stroke are specified in a statement, use fillindex=value or strokeindex=number to explicitly make clear which is being associated.

You can also use your own function to associate numbers with colorspace. In this case, you can pass the function to the program vcolorkey as the parameter "colorvalue"

So if mycolor() is a function that you use to convert numbers (integer or floating) into colors (or anything that VSG can recognize in a color assignment), use

vcolorkey(colorvalue=mycolor) to apply mycolor() in making a color key.

With a colorvalue function assigned, you still need to have assigned colorindex=value (or fillindex=number or strokeindex=number) when making the appropriate objects.

Advanced: You can also use python's lambda feature to specify the colorkey function: e.g., vcolorkey(colorvalue=(lambda x:rgb(2,2,2))) [maps an x of 2 to the color '#222']

Some additional (optional) parameters for vcolorkey | spectrumtitle=Title to sit above colorkey spectrum; gsorted (or nsorted)=True:sort keys by text (number)

logmode=True: log x axis (default:False) | mincolorindex=, maxcolorindex= sets start/end of color range | inclusive=True explicitly label all integers in range (default:False)

ckfont= The font used for labels in the colorkey (e.g. "courier 12 Bold"). If you don't specify a size, colorkey makes an (educated) guess | keyheight= Set line height of colorkey

ckfontcolor= The font color for text labels in colorkey (default: default font color for the canvas, VSG.fontcolor) | lastplus=True : Add "+" to end of spectrum number range

x1, x2= Set left,right edges of colorkey (default: 1/7 to 4/7 from VSG.xmin to VSG.xmax) | y2= Sets upper edge of the colorkey (default: current bottom edge VSG.ymin)

**Experimental Feature 3:** version EA (02-27-22) adds some primative interface and file handling tools:

vCommand() takes all previous assignment statements (eg, x=1) and allows variables to be reset from the command line. Thus putting the vommand vCommand() in your code after a set of assignment statements allows any of the assigned variables to be reset from the command line by name. Here is a short example program that counts all "G" characters in the file mySequence.fasta

## MySimpleCountingScript.py

## Choose file and which base to count

InputFileName = 'mySequence.fasta'

BaseToCount = 'G'

## Get all data in file by opening the file and reading

InputData = open(InputFileName, mode='rt').read()

CountOfBases = InputData.count('G')

print("Counting result for ",InputFileName,BaseToCount,CountOfBases)

Now lets make this into a command line flexible program by allowing the user to adjust the first variables. Note that all we do is import the vCommand function and call it after the relevant variables have been declared in the original program (two lines of code).

## Choose file and which base to count

InputFileName = 'mySequence.fasta'

BaseToCount = 'G'

## enable the program to detect user-specified values for the preceeding variables in the command line

from VSB\_ModuleEA import vCommand

vCommand()

## Get all data in file by opening the file and reading

InputData = open(InputFileName, mode='rt').read()

CountOfBases = InputData.count('G')

print("Counting result for ",InputFileName,BaseToCount,CountOfBases)

Now when the file runs, you'll be able to specify parameters in the command line, e.g.,

python MySimpleCountingScript My InputFileName='MyNewSequenceData.fasta' BaseToCount='A'

vOpen("MyFileName", mode='xx') and vClose(FileObject) will open/close files for reading or writing text and automatically detects and handles gzip (.gz) and most zip (.zip) archives. This avoids any need to write separate code to handle zip and gzip archives full of text.

vLog("MyNote") opens up a time-stamped log file and writes the note+time to this file

vFastAToDict("MyFastAFileName") returns a dictionary where keys are sequence names and values are sequences

vtdvToDict("MyTabDelimitedValueFileName"). Returns a list of headers (text list) and a dictionary where

keys are row ids (tuple taken from the first few columns in each row)

values are lists of column entries for the individual rows (converted to integer if possible, float, or kept as text)

vantisense(Sequence). Returns antisense sequence for any input DNA

**Experimental Feature 4:** version EO (09-05-22) adds a command line option and python to convert individual files of groups of files from .svg to vector .pdf formats. Some fonts are adjusted in this conversion but in general the product is a high resolution file easily viewed by a standard PDF viewer such as AcrobatReader of Apple Preview:

Saving a vector PDF directly from Python script simply needs a .pdf extension as long as reportlab and svglib are installed:

e.g., vwrite("MyFile.pdf") or vdisplay("MyFile.pdf")

Calling converter from python uses the function vrenderPDF. This routine takes a single parameter, which is the file specification and can be either a single file name or a group of files specified with a wildcard (e.g. vrenderPDF ("MyFile.pdf") or vrenderPDF ("MyFile.pdf")

Conversion can also be done from the command line, with thc command Line Syntax

python VSG\_ModuleET.py vRenderPDF <file specification>

If no file specification is provided, this command line converts all '.svg' files in current directory to .pdf

Alternatively the user can add a file or wildcard specification (name or wildcard) to instruct VSG which files to convert

e.g., python VSG\_ModuleET.py vRenderPDF MyNewFiles\*.svg or python VSG\_ModuleET.py vRenderPDF MyFile.svg

Note that installation of the packages reportlab and svglib can be somewhat challenging. On OSX, it seemed that the location of a folder "FreeType" may have been change in newer versions of the operating system so that is is now placed inside a new folder called FreeType2. Making a copy of the "FreeType" folder and putting this into the folder that now contains FreeType2 seemed to provide the needed resources for the installer (here is a reference on StackExchange [https://stackoverflow.com/questions/36400099/missing-header-file-for-reportlab-dependency]. There may be several instances of FreeType and FreeType2 on any system, knowing which needs to be corrected by copying the FreeType folder to the parent directory may require some trial and error. Althernatively this may be evident from the error message indicated by the ReportLab installer. In some cases, the copy command may also need to be done with sudo priveleges.

**Experimental Feature 5:** Version ET (12-31-22) adds ability to create line and objects that have a variable stroke-width that scales as the graphic is zoomed in and out. The default for VSG is to create objects that zoom proportionallhy when either the display is magnified (e.g. zooming in with the browser or a picture viewer) or when canvases are saved using a vwrite or vdisplay command that has a scale command in it vdisplay('mydrawing.png', scale=3.0). There are two different modes where this can be overridden. The first is to specify svgstrokezoom=True when the object is created. This can also be set globally with vset(svgstrokezoom=True). Importantly, only objects that are created with svgstrokezoom=True in the original object creation statement or that were created after a vset(svgstrokezoom=True) command will have the "fixed stroke-width-on-zoom" property, and this is only relevant for display as svg files (not for pdf, jpg, etc).

A second means to allow objects to retain a fixed size on scaling is accessed by by prepending 'r' to the strokewidth or font size when the object is defined. (e.g. vcircle(xc=0,r=5,strokewidth='r3'); vdisplay('mydrawing.png', scale=3.0). Once an drawing file has been written, the size will remain proportional to other elements in the drawing, so this option only affects the scaling of objects that VSG does in response to the scale=#.# command.