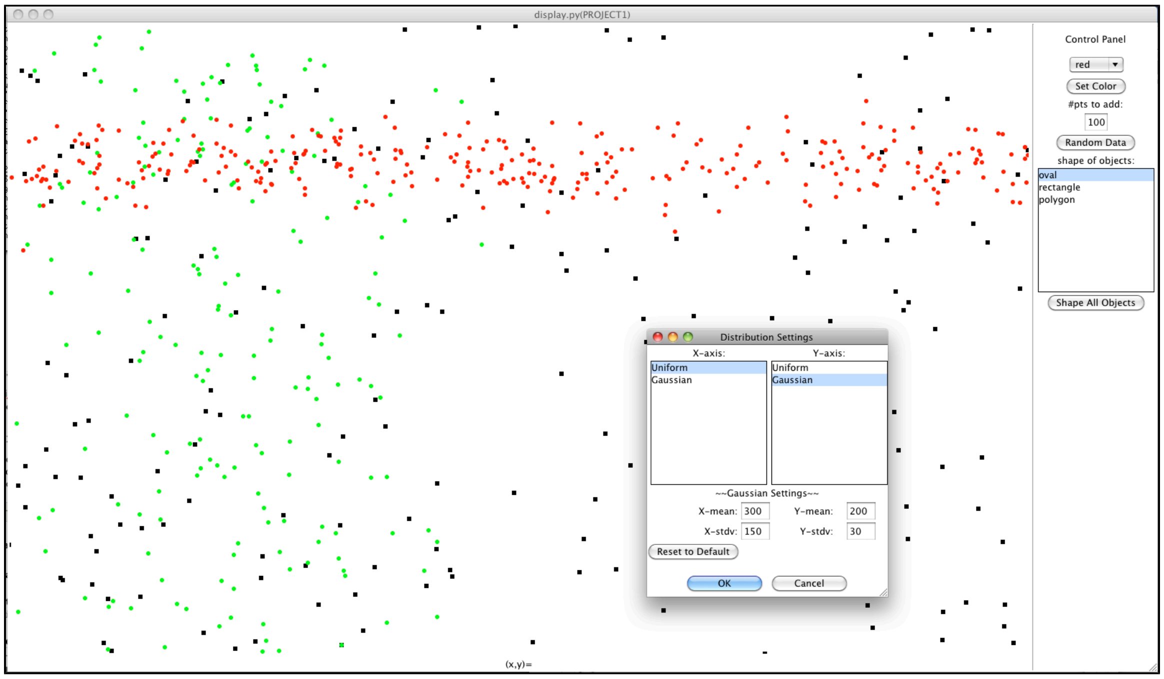
# cs251s14project1

## CS251: Project1

Write-Up: Create a new wiki page, give it a useful title (e.g. Stephanie's CS251 Project 1), and describe the functionality of your GUI. Include at least one screen shot and clearly identify any extensions that you did. As a rule-of-thumb, your write-up should be approximately 2 pages when printed. It is important to be both thorough and concise!



X-axis: [✓ Uniform] Gaussian
Y-axis: Uniform [✓ Gaussian]
~~Gaussian Settings~~
X-mean: 300   Y-mean: 200
X-stdv: 150   Y-stdv: 30

This project has allowed me to familiarize myself with TKinter and become accustomed to the syntax of Python. Below is a summary of some of the features:

**Some Keyboard and Mouse Buttons**

MouseMotion --> if mouse is over an object, the coordinates will be printed at the bottom of the canvas (ext.4)

Double-Click MouseB1 --> adds a point on the canvas at the point the cursor is pointing to (pt.2b)

Drag with MouseB1 --> move canvas view (pt. 2a)

Drag with MouseB2 --> if the cursor is on an object in the canvas, all of the objects on the canvas will scale to a larger size (ext.1)

Shift-Cmmd-MouseB2 --> the point(s) below the cursor will be removed from the canvas (ext.5)

Cmmd-Q --> quit out of the window and program. Command also found in the pull-down menu

Cmmd-N --> this command will clear all the data objects in the canvas. Command also found in the pull-down menu (pt.1b)

Cmmd-D --> opens Dialog window of "Distribution Settings." Choose uniform or Gaussian distribution for each of the x and y axes. Gaussian options below allow the user to set the mean and standard deviation or reset to default settings (mean=1/2 screen axis, std=1/8 screen axis). Command also found in the pull-down menu (pt.3, ext.6, ext.7)

### The Right-Side "Control Panel"

Random Data --> random objects will be placed on the canvas based on the settings displayed in the Control Panel (pt.2a)

Adjust Number --> an Entry widget where the user can type in the number of data objects to generate by the "Random Data" button. Initialized with 10 objects. (ext.2)

Adjust Shape --> a ListBox widget where the selected shape will be the shape used to generate the data objects by "RandomData." Initialized with the "oval" selected. The "polygon" selection produces the same results as the "oval" selection. (ext.3)

Adjust Color --> The pull-down Menu allows the user to select a color for objects generated by "Random Data." The "Set Color" button sets all objects on the canvas to the selected color.

## Part 3: creating a dialog window to manage distribution settings

DistributionDialog is derived from the Dialog class which initializes dialog window with only the **buttonbox** set of widgets executing the classic "ok" and "cancel" methods. I only overwrote the **body** and **apply** methods while adding my own helper method **handleGssD**.

## Extension1: enlarging objects in the canvas

Most of the code for this extension can be found in the **handleMouseButton2** and **handleMouseButton2Motion** methods. It is important to save the initial location and size information of the objects in the initial click so that the objects can grow with respect to their original sizes and they can grow based off the total movement of the mouse rather than the speed of the mouse.

```
def handleMouseButton2(self, event):
 # save location of initial click
 self.baseClick = (event.x, event.y)

 # save the initial coordinates of the objects
 for obj in self.objects:
  i = self.objects.index(obj)
  loc = self.canvas.coords(obj)
  if i == len(self.objCurLoc):
   self.objCurLoc.append(loc)
  else:
   self.objCurLoc[i][:] = loc
 self.baseClick = (event.x, event.y) self.objCurLoc[i][:] = lo
```

The **if-else** statement allows us to check if the 2-D array of coordinates for all the objects needs to be expanded to accommodate new objects or if the information needs to be overwritten. We could just append the information onto the 2-D array and it would work for the first expansion, but if we wanted to adjust the size again, the initial coordinates being called from Motion would be that of the original sizes from before the first size-adjustment because the updated data would not be found in the corresponding indices of the object array.

Once the initial click and coordinate location has been saved into field variables, we can write the motion method to adjust the size of the objects and update their new coordinates.

```
def handleMouseButton2Motion(self, event):
 # calc num of pix to incr/decr coords by, based off dist mouse travels up/down
 diff = ( event.x - self.baseClick[0], event.y - self.baseClick[1] )
 tenpercscr = int( self.canvas.winfo_height()/10 ) # type-cast in case canvas not
mult. of 10
 addpix = ( (dif[1] % tenperscr) - diff[1]] )/tenperscr

 # update size of all objects
 for obj in self.objects:
  i = self.objects.index(obj)
  loc = self.objCurLoc[i][:]
  self.canvas.coords(obj, loc[0]-addpix,
     loc[1]-addpix,
     loc[2]+addpix,
     loc[3]+addpix )
```

Each oval increases in size by 2 pixels (one on each side) for every tenth of the screen up/down the mouse travels from the initial click. First I calculated the pixel difference between the initial click and the current mouse location (the difference along x is not used) and stored it in **diff**. Then I calculated the pixels of a tenth of the screen (being sure to round to the nearest pixel) as **tenperscr**. Finally, the number of pixels to add to each coordinate is stored in **addpix**. The **for** loop simply takes the original coordinates and adds or subtracts as appropriate by the value stored in **addpix** to each coordinate.

## Extension 2: widget to control how many points to add when plotting random points

I used the **Entry** widget for obtaining user input for the number of objects to add to the canvas. The default is set at 10 when the window is initially generated. I started with code under the **buildControls** method just below the set color button.

```
# create label for user's benefit
label = tkLabel( rightcntlframe, width=3 )
label.pack()


# create Entry box initialized with "10"
self.txtwid = tk.Entry(rightcntlframe, width=3)
self.txtwid.insert(0,"10")
self.txtwid.pack()
```

I created a label for the **Entry** box, and then I created the **Entry** box with a starting value of 10. Then under the **randomDataPoints** method I created in Lab1 I retrieved the value from the field widget **self.txtwid** to set the **points** variable.

## Extension 3: controlling the shape of the data objects

There are two aspects to this extension: setting the shape of subsequent objects placed by the **randomDataPoints** method and a button which sets all objects on the canvas to the selected shape. The shape used is determined by a **ListBox** in the right-side control frame with the default shape selected as an oval. The only shapes currently available are the "oval" (circle) and the "rectangle" (square) because I have not decided yet how to use the polygon method. If I add a triangle ora any other shape, I must consider how I would scale it when I use the **handleMouseButton 2Motion** in extension 1. "Oval" and "rectangle" both use four coordinates so it is easy to call either **create_oval** or **create_rectangle** without having to change the code in the other extensions.

In my code, I create a simple **ListBox** in the right side control frame with the oval and rectangle options ("polygon" is set to plot ovals just the same as the "oval" selection). I also add a button labeled with "Shape All Objects" which calls the **handleButton3** method. Then I added a couple lines to the **randomDataPoints** method which calls the index of the selected string from the list (0=oval, 1=rectangle, 2=polygon) and stores it in the **shp** variable. Based on the value of the **shp** variable, the **pt** object to be appended to the canvas is created as either an oval or a rectangle. Whenever **randomDataPoints** is called, it will check the **shp** the controls have selected to determine what kind of object it will add to the canvas.

In order to change the shape of all the current objects on the canvas, we have created a button with the **handleButton3** command assigned to it. The code for this method is below:

```
#reset all shapes to currently selected shape
def handleButton3(self)
 shp = int( self.listBox1.curselection()[0] )
 for obj in range( len(self.objects) ):
  loc = self.canvas.coords(self.objects[0])
  # handle oval selection
  if shp == 0:
   pt = self.canvas.create_oval( loc[0], loc[1], loc[2], loc[3],
fill=self.colorOption.get(), outline='' )
  # handle rectangle selection
  if shp == 1:
   pt = self.canvas.create_rectangle( loc[0], loc[1], loc[2], loc[3],
fill=self.colorOption.get(), outline='' )
  # handle polygon selection
  if shp ==2:
   pt = self.canvas.create_oval( loc[0], loc[1], loc[2], loc[3],
fill=self.colorOption.get(), outline='' )
  # delete object from canvas and from front of list
  self.canvas.delete(self.objects[0])
  del self.objects[0]
  # add new object to end of list
  self.objects.append(pt)
```

Because I was running into trouble with the methods that create shapes not directly replacing their objects in the list at their index, I decided to use a loop based on indices instead of a loop that went through each object in a list. To determine the shape, I simply saved the shape-type variable into **shp** and built new objects based on its value. Because the loop I used is based off indices, I deleted objects from the front of the list (notice all the 0 indices used when calling an object from **self.object**) and I appended new objects to the end of **self.object**. All shapes are set to either an oval (the polygon, shp=2, is set to be created as an oval), or a rectangle (shp=1).

## Extension 4: mouse-over reports position of data point underneath it

The code for this extension is a little more scattered. The **Frame** in which the coordinates are reported was labeled **statusarea** and created with the other frames and controls. The **Frame** only has the field **Label botlbl**. Under the **setBindings** method, I bind **<Motion>** to the **handleMouse Motion**, which updates the label in the **statusarea** frame as the mouse moves.

```
#update the bottom label if the mouse is hovering over an object
def handleMouseMotion(self, event):
 thisX = -1
 thisY = -1
 for obj in self.objects:
  loc = self.canvas.coords(obj)
  # if mouse is over object, save centerpoint of x and y
  if loc[0]<=event.x<=loc[2] and loc[1]<=event.y<=loc[3]:
   thisX = (loc[0]+loc[2]) / 2
   thisY = self.canvas.winfo_height() - ((loc[1]+loc[3]) /2)
 # update coords on label if mouse over object
 if thisX > -1:
  coords = '(x,y)=( ' + str(thisX) + ', ' + str(thisY) + ')'
  self.botlbl.config(text=coords)
 # update to default if not over object
 else:
  self.botlbl.config(text='(x,y)=')
```

First it saves the current x and y coordinates of the mouse and goes through every object to see if it is hovering over an object. If it is, the point the object is centered on (avg. each of the x-coord and y-coord values) is saved into **thisX** and **thisY**(note that the Y coord was subtracted from the height to get the position). If not, the **thisX** and **this** remain as -1. By using an **if-else** statement, we can set the label to its default string or update it to show the coordinates of the points.

By taking the average instead of directly printing the mouse position, we give a more accurate position coordinate for the object. This takes a little more code but it is also more precise.

## Extension 5: shift-cmd-mouse-button1 will delete point(s) underneath it

Any object or objects under the mouse when it is clicked while holding the shift and command buttons will be removed from the canvas. This is actually a very simple extension where **<Shift-Command-Button-2>** is bound to the handle method below:

```
# remove the objects containing the event coordinates
def handleShiftCmmdMouseButton2(self, event):
 self.baseClick = (event.x, event.y)
 for obj in self.objects:
  loc = self.canvas.coords(obj)
  if loc[0]<=event.x<=loc[2] and loc[1]<=event.y<=loc[3]:
   i = self.objects.index(obj)
   del self.objects[i]
   self.canvas.delete(obj)
```

The code simply loops through all the objects on the canvas and if the mouse is found within its coordinates, it is deleted from the canvas and the **objects** list. Note that if there is more than one object under the mouse, both will be removed from the canvas.

## Extension 6: distribution dialog box begins with current x and y distributions as the initial selections

This was also a relatively simple extension. I simply passed the distribution information, a field tuple called **dist**, into the constructor of **DistributionDialog** under the name **curDistr**. Once these two variables are linked, I added a couple lines in the **body** method to set the initial selection to the indices found in **curDistr** (a 0 indicates "uniform" distribution and a 1 indicates "gaussian" distribution).

–

Here's the site with the most JavaDoc-like set-up:

http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html