Universidad de Guanajuato

División de Ciencias e Ingenierías

Reporte de proyecto final: Generador de objetos astronómicos

Uriel Chávez Flores

Baruch Mejía Martínez

Fernando Emanuel Trujillo Cándido

Asignatura: Programación Orientada a Objetos y Eventos

10 de junio de 2022

Descripción general

Este programa genera un conjunto de objetos astronómicos que simulan las observaciones de observatorios que se dedican a la espectrografía de galaxias y cuásares.

Los objetos se generan de manera homogénea sobre la bóveda celeste con distribuciones en corrimiento al rojo y razón entre cuásares y galaxias basados en datos de observaciones reales del proyecto SDSS. Para cada objeto también se genera un espectro, cuya generación también se basa en datos experimentales reales, se realizó un análisis de componentes principales de tres diferentes tipos de objetos, galaxias en general, cuásares con corrimiento al rojo menor a 2.1 y mayor a 2.1.

En este documento explica el funcionamiento del programa describiendo los 3 archivos principales. El análisis estadístico se realizó en el cuaderno PCA.ipynb, los resultados de este análisis fueron usados en el archivo objast.py en donde se declaran las clases principales necesarias para la creación de los objetos astronómicos, y finalmente en el archivo main.py se crean las interfaces gráficas con las que el usuario interactúa para obtener información de los objetos generados.

Análisis estadístico (cuaderno PCA.ipynb)

Como se mencionó antes, el análisis estadístico se realizó con datos de espectros reales medidos por el SDSS (Sloan Digital Sky Survey). En total se usaron 10 catálogos de 1000 objetos de las carpetas públicas del SDSS previamente descargados (https://data.sdss.org/sas/dr14/eboss/spectro/redux/v5_10_0/). Estos catálogos guardan el espectro de los objetos junto a más información que los identifica.

```
plates_local='../spPlates/'
plate_files = []
plate_files.append('spPlate-3586-55181.fits')
plate_files.append('spPlate-3587-55182.fits')
plate_files.append('spPlate-3588-55184.fits')
plate_files.append('spPlate-3589-55186.fits')
plate_files.append('spPlate-3590-55201.fits')
plate_files.append('spPlate-3606-55182.fits')
plate_files.append('spPlate-3607-55186.fits')
plate_files.append('spPlate-3609-55201.fits')
plate_files.append('spPlate-3615-55179.fits')
plate_files.append('spPlate-3639-55205.fits')

ids = [[3586,55181],
       [3587,55182],
       [3588,55184],
       [3589,55186],
       [3590,55201],
       [3606,55182],
       [3607,55186],
       [3609,55201],
       [3615,55179],
       [3639,55205]]
```

```
# Abrimos los archivos
plates = []
for i in range(len(plate_files)):
    file = plates_local + plate_files[i]
    plates.append(fits.open(file))
plate1 = fits.open(file)
```

De todas las observaciones se seleccionaron aquellas identificadas como cuásares y galaxias, guardándolas en arreglos por separado.

```
lims = []
logwaves = []
fluxes = []
ratios_qso_gal = []
id_qso = np.array([0,0,0,0])
id_gal = np.array([0])
for i in range(len(plate_files)):
    # Obtenemos longitudes de onda y flujos
    fluxes.append(np.array(plates[i][0].data))
    pltheader = plates[i][0].header
    coeff0 = pltheader['COEFF0']
    coeff1 = pltheader['COEFF1']
    logwaves.append(coeff0 + coeff1 * np.arange(plates[i][0].data.shape[1]))
    # También necesitamos sus límites para luego recortarlas
    # a un rango de longitudes de onda estándar
    lims.append([logwaves[i][0],logwaves[i][-1:][0]])

    # Obtenemos las IDS de los objetos identificados como cuásares y galaxias
    plugmap = plates[i]['PLUGMAP'].data
    qso_pm = plugmap['OBJTYPE'] == 'QSO'
    gal_pm = plugmap['OBJTYPE'] == 'GALAXY'
    fbid_pm = plugmap['FIBERID']
    # Los guardamos en una tabla junto a las ids de los archivos
    tableids = np.array([[ids[i][0], ids[i][1], fbid, i] for fbid in fbid_pm[qso_pm]])
    id_qso = np.vstack([id_qso,tableids])
    # De las galaxias guardamos solo el FIBERID + n_archivo * 1000 para recuperar los espectros
    tableids_gal = np.array([fbid + i*1000 for fbid in fbid_pm[gal_pm]])
    id_gal = np.append(id_gal,tableids_gal)

    # Guardamos también la razón entre galaxias y cuásares
    ratios_qso_gal.append(len(tableids)/len(tableids_gal))
id_qso = id_qso[1:]
id_gal = id_gal[1:]
```

Dado que cada archivo tiene diferentes rangos de medición de longitudes de onda, se estandarizó a un único rango, esto es necesario para poder hacer el PCA.

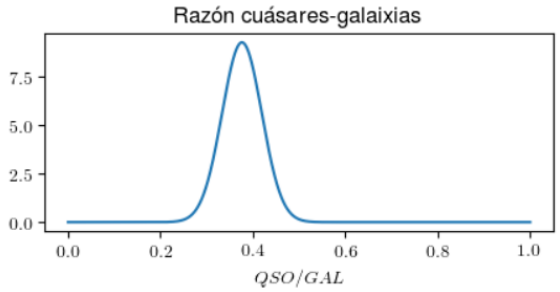
```
# Encontramos el mayor rango de longitudes de onda que aceptan los datos de los 10 archivos
lims = np.array(lims)
lim_inf = lims.T[0].max()
lim_sup = lims.T[1].min()
lim_slices = -((lims - [lim_inf, lim_sup])*10001).astype(int)


# Conociendo los límites, recortamos los flujos
flux = np.zeros(int((lim_sup - lim_inf) * 10000)+1)
logwave = []
for i in range(len(plates)):
    fluxes[i] = fluxes[i][:].T[lim_slices[i][0]:len(fluxes[i][0])+lim_slices[i][1]].T
    logwaves[i] = logwaves[i][lim_slices[i][0]:len(logwaves[i])+lim_slices[i][1]]
    flux = np.vstack([flux, fluxes[i]])
flux = flux[1:]

# Ahora el arreglo de longitudes de onda es el mismo para los flujos recortados
logwave = logwaves[0]
```

Ya teniendo separados los cuásares de las galaxias, se calculó la razón entre ellas asumiendo una distribución normal de esta cantidad.

```
# Para obtener la razón entre cuásares y galaxias asumimos una
# distribución normal, por lo tanto,  calculamos la desviación
# estándar y la media de la razón en los 10 archivos usados
# Estos datos los copiamos maualmente al módulo objast
ratio_avg = np.average(ratios_qso_gal)
ratio_std = np.std(ratios_qso_gal)
print("Media: ", ratio_avg)
print("Desviación estándar: ", ratio_std)
```

✓ 0.7s

```
Media:  0.37625819972845664
Desviación estándar:  0.042980063642444125
```



En los catálogos anteriores no se almacena el corrimiento al rojo del objeto en cuestión, para obtenerlo se necesita otro catálogo de cuásares en donde sí

podemos encontrar el dato, en este podemos identificar los cuásares del filtrado anterior por medio de los dos números de identificación “MJD” y “PLATE”.

```
# Se buscan los cuásares que se filtraron de los archivos anteriores en el
# catálogo DR14Q_v4_4.fits, en este buscaremos los corrimientos al rojo
catalog_full = Table.read('../DR14Q_v4_4.fits');

cat = catalog_full[((catalog_full['MJD']==55181) & (catalog_full['PLATE']==3586)) |
                   ((catalog_full['MJD']==55182) & (catalog_full['PLATE']==3587)) |
                   ((catalog_full['MJD']==55184) & (catalog_full['PLATE']==3588)) |
                   ((catalog_full['MJD']==55186) & (catalog_full['PLATE']==3589)) |
                   ((catalog_full['MJD']==55201) & (catalog_full['PLATE']==3590)) |
                   ((catalog_full['MJD']==55182) & (catalog_full['PLATE']==3606)) |
                   ((catalog_full['MJD']==55186) & (catalog_full['PLATE']==3607)) |
                   ((catalog_full['MJD']==55201) & (catalog_full['PLATE']==3609)) |
                   ((catalog_full['MJD']==55179) & (catalog_full['PLATE']==3615)) |
                   ((catalog_full['MJD']==55205) & (catalog_full['PLATE']==3639))]
```

Una vez filtrados, almacenamos en un arreglo la información del filtrado anterior y el corrimiento al rojo.

```
# Guardaremos en un arreglo la siguiente información de cada cuasar
# [0] PLATE
# [1] MJD
# [2] FIBERID
# [3] Z
idz_cat = np.array([np.array(cat["PLATE"]),np.array(cat["MJD"]),np.array(cat["FIBERID"]),np.array(cat["Z"])]).T
full_id_cat = [str(int(i[0]))+str(int(i[1]))+str(int(i[2])) for i in idz_cat]
full_id_qso = [str(int(i[0]))+str(int(i[1]))+str(int(i[2])) for i in id_qso]
idz_qso = idz_cat[np.in1d(full_id_cat,full_id_qso)]

z = idz_qso.T[3]

# [4] N_PLATE
n_plate_id_qso = id_qso[np.in1d(full_id_qso,full_id_cat)].T[3]
idz_qso = np.append(idz_qso.T,[n_plate_id_qso], axis = 0).T

# [5] INDEX IN FLUX
index_in_flux = idz_qso.T[2] + 1000 * idz_qso.T[4] + 1
idz_qso = np.append(idz_qso.T,[index_in_flux], axis = 0).T
```

De todos los objetos filtrados se separaron en tres categorías para la generación de espectros de dichas tres categorías, las cuales son cuásar con z<2.1, cuásar con z>2.1 y galaxias.

```
# Separamos en aquellos con z>2.1 y z<2.1
idzmenos_qso = idz_qso[idz_qso.T[3]<2.1]
idzmas_qso = idz_qso[idz_qso.T[3]>=2.1]
```

Para la graficación correcta de la longitud de onda de los objetos observados es necesario el corrimiento al rojo. Se almacenaron los espectros de cada categoría en arreglos separados, para sobre estos hacer el PCA.

```
# Con el corrimiento al rojo podemos recorrer la longitud de onda para graficarlos
lrf = [10**logwave/(1+z[i]) for i in range(len(z))]
spectra = np.array([flux[idz_qso.T[5][i].astype(int)] for i in range(len(z))])
spectramas = np.array([flux[idzmenos_qso.T[5][i].astype(int)] for i in range(len(idzmenos_qso.T[5]))])
spectramenos = np.array([flux[idzmas_qso.T[5][i].astype(int)] for i in range(len(idzmas_qso.T[5]))])
spectragalaxia = np.array([flux[id_gal[i]] for i in range(len(id_gal))])
```

Las cantidades totales de objetos de cada categoría son las siguientes.

```
    # Mas información de nuestros datos
    print("Cuásares totales: ", len(idz_qso))
    print("Cuásares z<2.1: ", len(spectramenos))
    print("Cuásares z>2.1: ", len(spectramas))
    print("\nGalaxias totales: ", len(spectragalaxia))
```

✓ 0.4s

```
Cuásares totales:  282
Cuásares z<2.1:  189
Cuásares z>2.1:  93

Galaxias totales:  5670
```

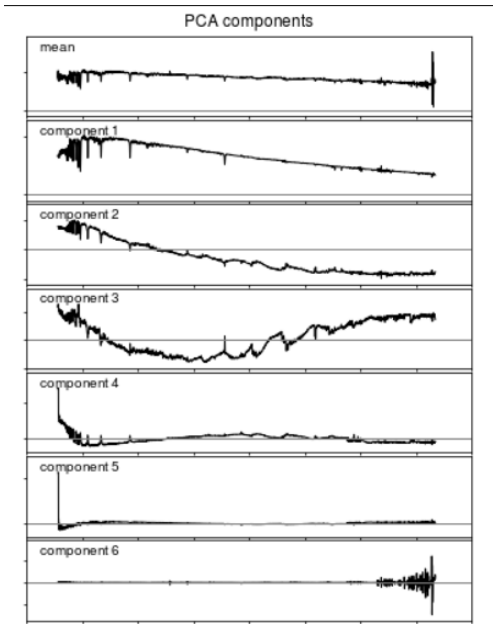
El PCA se realizó usando la librería sklearn a la cual se le dio como argumento los arreglos de espectros.

```
# En total usaremos 50 componentes principales más la media para generar los espectros
n_components = 51
#decompositions = compute_PCA(spectra,n_components)
#decompositions = compute_PCA(spectramenos,n_components)
#decompositions = compute_PCA(spectramas,n_components)
decompositions = compute_PCA(spectragalaxia,n_components)
```

```
#Se crea función para hacer la descomposición
@pickle_results('spec_decompositions.pkl')
def compute_PCA(spectra,n_components=5):
    spec_mean = spectra.mean(0)
    # PCA: use randomized PCA for speed
    pca = PCA(n_components - 1, random_state=0, svd_solver='randomized')
    pca.fit(spectra)
    pca_comp = np.vstack([spec_mean,pca.components_])
    return pca_comp
```

Esta función regresa las componentes principales en un arreglo, las cuales graficamos con la siguiente función

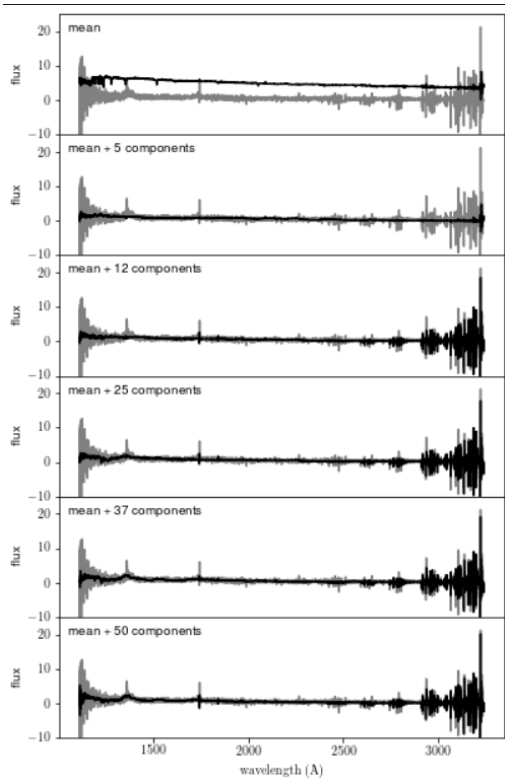
```
def plotdecom(decompositions):
    global logwave
    n_components = len(decompositions)
    fig = plt.figure(figsize=(5, n_components))
    fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05,
                        bottom=0.1, top=0.95, hspace=0.05)
    titles='PCA components'
    i=1
    for j in range(n_components):
        ax = fig.add_subplot(n_components,1,j+1)
        ax.yaxis.set_major_formatter(plt.NullFormatter())
        ax.xaxis.set_major_locator(plt.MultipleLocator(1000))
        if j < n_components - 1:
            ax.xaxis.set_major_formatter(plt.NullFormatter())
        else:
            ax.xaxis.set_major_locator(
                plt.FixedLocator(list(range(3000, 11999, 1000))))
            ax.set_xlabel(r'wavelength ${\rm (\AA)}$')
        ax.plot(10**logwave, decompositions[j], '-k', lw=1)
        # plot zero line
        xlim = [3000, 11000]
        ax.plot(xlim, [0, 0], '-', c='gray', lw=1)
        if j == 0:
            ax.set_title(titles)
        if j == 0:
            label = 'mean'
        else:
            label = 'component %i' % j
        ax.text(0.03, 0.94, label, transform=ax.transAxes,
                ha='left', va='top')
        for l in ax.get_xticklines() + ax.get_yticklines():
            l.set_markersize(2)
        # adjust y limits
        ylim = plt.ylim()
        dy = 0.05 * (ylim[1] - ylim[0])
        ax.set_ylim(ylim[0] - dy, ylim[1] + 4 * dy)
        ax.set_xlim(xlim)
    plt.show()
```



.
.
.

Para visualizar cómo las componentes principales reproducen los espectros experimentales se usó la siguiente función

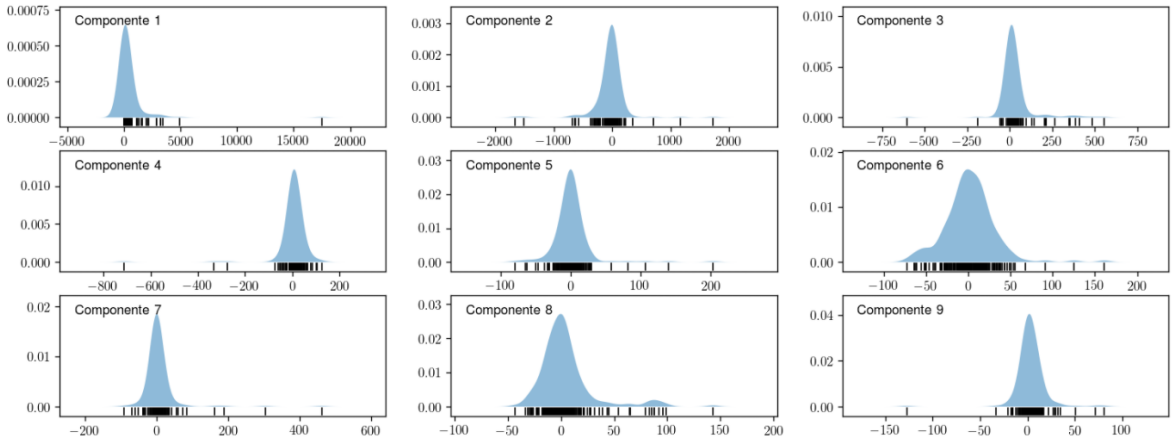
```python
# Plot the sequence of reconstructions
def plotrecons(evecs,spec,spec_mean,wavelengths):
    coeff = np.dot(evecs, spec - spec_mean)
    n_components = len(evecs)
    iters = (np.array([0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1])*n_components).astype(int)
    fig = plt.figure(figsize=(6, 14))
    fig.subplots_adjust(hspace=0, top=0.95, bottom=0.1, left=0.12, right=0.93)
    for i, n in enumerate(iters):
        ax = fig.add_subplot(911 + i)
        ax.plot(wavelengths, spec, '-', c='gray')
        ax.plot(wavelengths, spec_mean + np.dot(coeff[:n], evecs[:n]), '-k')
        if i < 3:
            ax.xaxis.set_major_formatter(plt.NullFormatter())
        ax.set_ylim(-10, 25)
        ax.set_ylabel('flux')
        if n == 0:
            text = "mean"
        elif n == 1:
            text = "mean + 1 component\n"
        else:
            text = "mean + %i components\n" % n
        ax.text(0.02, 0.93, text, ha='left', va='top', transform=ax.transAxes)
    fig.axes[-1].set_xlabel(r'${\rm wavelength\ (\AA)}$')
    plt.show()
```



Las componentes principales de cada tipo de espectro fueron guardadas en archivos de texto para usarlas después en la generación de espectros. Para generar los espectros usamos la distribución de los coeficientes de los espectros experimentales en la base de los componentes principales, estos coeficientes también fueron guardados en archivos de texto.

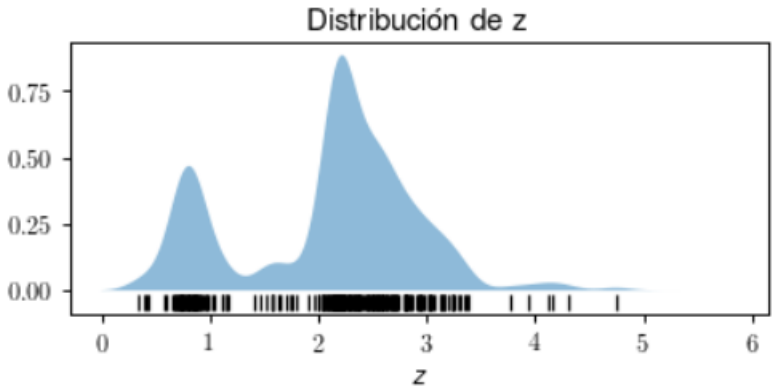
```python
# Para generar los espectros usaremos la distribución de los coeficientes
# de los datos origianles en la base de PCA
# Esta distribución la aproximaremos con KDE (kernel density estimation)
# que para obtener la distribución final, sustituye cada coeficiente por
# una distribución normal

# Calculamos los coeficientes
n_components = 50
coeffs = np.dot(spectramenos,evecs[:n_components].T).T

# Lo mismo haremos para obtener una distribución de los corrimientos
# al rojo y con dicha distribución generarlos
z = idz_qso.T[3]

# Líneas para guardar los archivos necesarios para la librería oast
# np.savetxt('Distribution_GAL.txt',coeffs, fmt = '%f', delimiter=" ")
# np.savetxt('evecs_GAL.txt',evecs, fmt = '%f', delimiter=" ")
# np.savetxt('spec_mean_GAL.txt',spectramas.mean(0), fmt = '%f', delimiter=" ")
# np.savetxt('Distribution_z.txt',z, fmt = '%f', delimiter=" ")
```

Con estos coeficientes se generaron distribuciones de probabilidad para la generación aleatoria de dichos coeficientes para nuevos espectros, esto se hizo mediante KDE (Kernel Distribution Estimation), que lo que hace es sumar distribuciones normales colocadas en cada posición de los datos experimentales.



La imagen anterior muestra solamente las distribuciones de las primeras 9 componentes, en total se usaron las distribuciones de 50 componentes. Muestreando estas distribuciones se generan los nuevos espectros, que son vectores en la base de las primeras 50 componentes principales.

La misma idea fue aplicada para la generación del corrimiento al rojo, usando los datos de corrimiento al rojo se generó una distribución de la cual se muestrea para generar nuevos corrimientos al rojo.



Generación de objetos (librería objast)

Como se describió en la sección de análisis estadístico, los espectros se generaron usando los KDE obtenidos de los PCA y los coeficientes de los datos experimentales en la base de los primeros 50 PCA.

En esta librería se cargan los archivos de texto generados por el análisis estadístico y construye las distribuciones de las que muestrearemos para generar espectros y corrimientos al rojo.

```
# VARIABLES GLOBALES Y CARGA DE ARCHIVOS
# Estos datos son necesarios para la generación de espectros
# y corrimientos al rojo (z)
dist_qso_mas = np.loadtxt("archivos/Distribution_QSO+2.1.txt")
spec_mean_qso_mas = np.loadtxt("archivos/spec_mean_QSO+2.1.txt")
evecs_qso_mas = np.loadtxt("archivos/evecs_QSO+2.1.txt")

dist_qso_menos = np.loadtxt("archivos/Distribution_QSO-2.1.txt")
spec_mean_qso_menos = np.loadtxt("archivos/spec_mean_QSO-2.1.txt")
evecs_qso_menos = np.loadtxt("archivos/evecs_QSO-2.1.txt")

dist_gal = np.loadtxt("archivos/Distribution_GAL.txt")
spec_mean_gal = np.loadtxt("archivos/spec_mean_GAL.txt")
evecs_gal = np.loadtxt("archivos/evecs_GAL.txt")

dist_z = np.loadtxt("archivos/Distribution_z.txt")

logwave = np.loadtxt("archivos/logwave.txt")

n_components = 50
```

```
# Los datos cargados se almacenan en un Dataframe al que podemos
# acceder fácilmente con los índices y nombre de columnas.
generadores = pd.DataFrame([[loadKdes(dist_qso_menos),evecs_qso_menos,spec_mean_qso_menos],
                            [loadKdes(dist_qso_mas),evecs_qso_mas,spec_mean_qso_mas],
                            [loadKdes(dist_gal),evecs_gal,spec_mean_gal]],
                           index=["QSO-","QSO+","GAL"], columns=["kdes","evecs","spec_mean"]).T

# Cargamos también la distribución de z
kde_z = loadKde_z()
```

Para cada tipo de objeto que generaremos se creó una clase, pero antes se definió una clase padre “objast” (objeto astronómico) que tiene como atributo su posición en coordenadas (RA, DEC) (ascensión recta y declinación), también tiene como método la generación de posiciones aleatorias homogéneas sobre la cúpula celeste.

```
# Clase madre objast (objeto astronómico)
# Solo tiene como atributo la posición en coordenadas (RA, DEC)
class objast():
    def __init__(self, RA_DEC = np.nan, *args, **kwargs):
        if(np.isnan(RA_DEC)):
            self.RA_DEC = self.rand_sphere()
        super(objast,self).__init__()

    def rand_sphere(self, npt=1):
        phi=np.random.uniform(-np.pi,np.pi,npt)
        theta=np.arccos(np.random.uniform(-1,1,npt))-np.pi/2
        pos = np.vstack((phi,theta)).T
        if(npt == 1):
            pos = pos[0]
            self.RA_DEC = pos
        return pos
```

Después, se definen las clases “oamas” y “oamenos” (objeto astronómico con z>2.1 y z<2.1 respectivamente), estas clases heredan de la clase objast y tienen como

atributo propio el corrimiento al rojo “z” para el cual tienen un método que lo genera de acuerdo con la distribución de corrimiento al rojo que se obtuvo del análisis estadístico.

```python
class oamas(objast):
    def __init__(self, z = np.nan, *args, **kwargs):
        if(np.isnan(z)):
            z = self.generarZ()
        else:
            self.z=z
        super(oamas,self).__init__(*args, **kwargs)

    def generarZ(self):
        z = 2.0
        while (z < 2.1):
            z = generarZ()
        self.z = z
        return z
```

```python
class oamenos(objast):
    def __init__(self, z = np.nan, *args, **kwargs):
        if(np.isnan(z)):
            z = self.generarZ()
        else:
            self.z=z
        super(oamenos,self).__init__(*args, **kwargs)

    def generarZ(self):
        z = 2.2
        while (z > 2.1):
            z = generarZ()
        self.z = z
        return z
```

Y finalmente se declaran las clases los tipos de objetos que se generarán en el programa final que son:

- Cuásar con z<2.1
- Cuásar con z>2.1
- Galaxia con z<2.1
- Galaxia con z>2.1

Que tienen como atributos su espectro, tipo de objeto y tipo de espectro, aparte de tener el método que genera su espectro en función de su atributo tipo de espectro. Notar que la generación de espectros de galaxias es la misma para z<2.1 y z>2.1, la única diferencia entre estas dos clases es la herencia del método de generación de corrimiento al rojo, uno hereda de “oamas” y otro de “oamenos”.

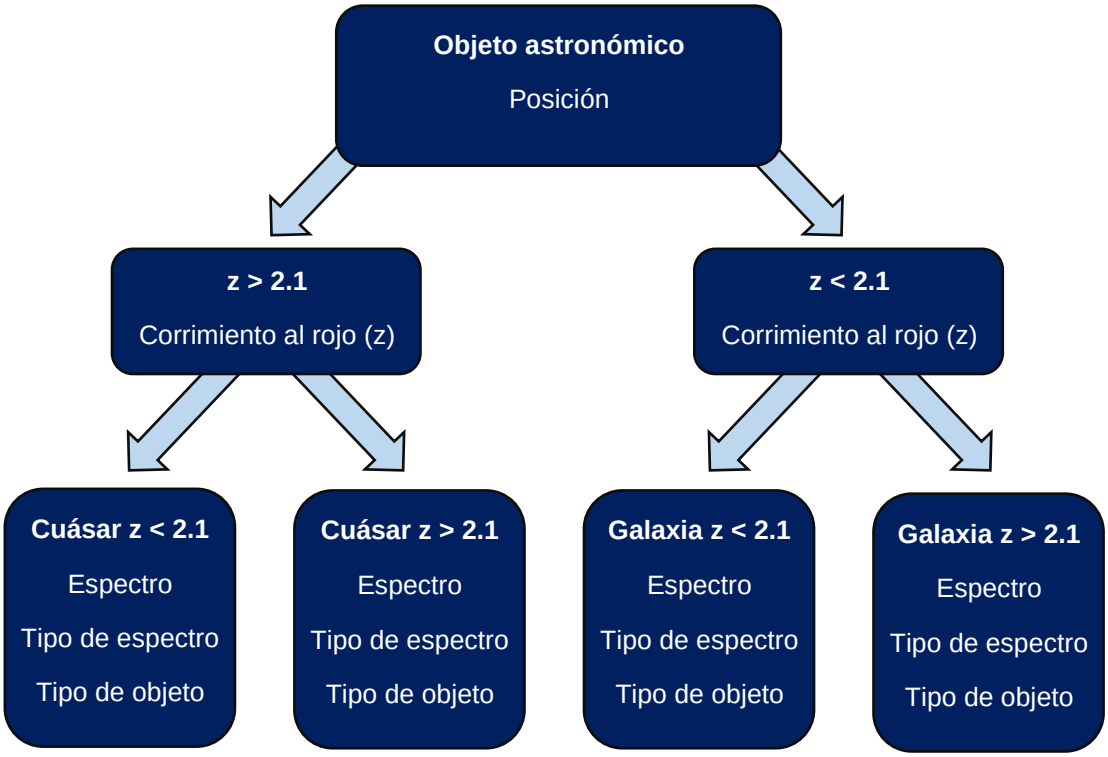
```python
class qsomas(oamas):
    def __init__(self, *args, **kwargs):
        self.espectro = self.generarEspectro()
        self.tipo_objeto = "QSO+"
        self.tipo_espectro = "QSO+"
        super(qsomas,self).__init__(*args, **kwargs)

    def generarEspectro(self,n_espectros=1):
        espectro_generado = generarEspectroKdes(generadores["QSO+"]["kdes"], n_espectros)
        self.espectro = espectro_generado
        return espectro_generado
```

```python
class qsomenos(oamenos):
    def __init__(self, *args, **kwargs):
        self.espectro = self.generarEspectro()
        self.tipo_objeto = "QSO-"
        self.tipo_espectro = "QSO-"
        super(qsomenos,self).__init__(*args, **kwargs)

    def generarEspectro(self,n_espectros=1):
        espectro_generado = generarEspectroKdes(generadores["QSO-"]["kdes"], n_espectros)
        self.espectro = espectro_generado
        return espectro_generado
```

```python
class galmas(oamas):
    def __init__(self, *args, **kwargs):
        self.espectro = self.generarEspectro()
        self.tipo_objeto = "GAL+"
        self.tipo_espectro = "GAL"
        super(galmas,self).__init__(*args, **kwargs)

    def generarEspectro(self,n_espectros=1):
        espectro_generado = generarEspectroKdes(generadores["GAL"]["kdes"], n_espectros)
        self.espectro = espectro_generado
        return espectro_generado
```

```python
class galmenos(oamenos):
    def __init__(self, *args, **kwargs):
        self.espectro = self.generarEspectro()
        self.tipo_objeto = "GAL-"
        self.tipo_espectro = "GAL"
        super(galmenos,self).__init__(*args, **kwargs)

    def generarEspectro(self,n_espectros=1):
        espectro_generado = generarEspectroKdes(generadores["GAL"]["kdes"], n_espectros)
        self.espectro = espectro_generado
        return espectro_generado
```

La estructura de herencia es la siguiente:

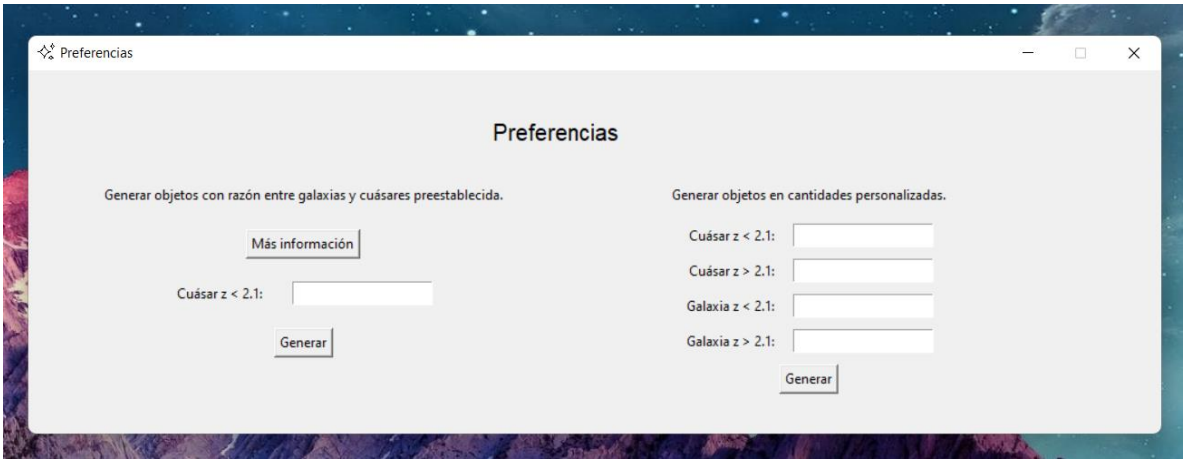


Después hay funciones secundarias que sirven para el tratamiento matemático de puntos y la generación de objetos dada una cantidad. Estas funciones son usadas en el módulo de la interfaz gráfica para interactuar con el modulo objast y para tratar los objetos que se obtienen de este.

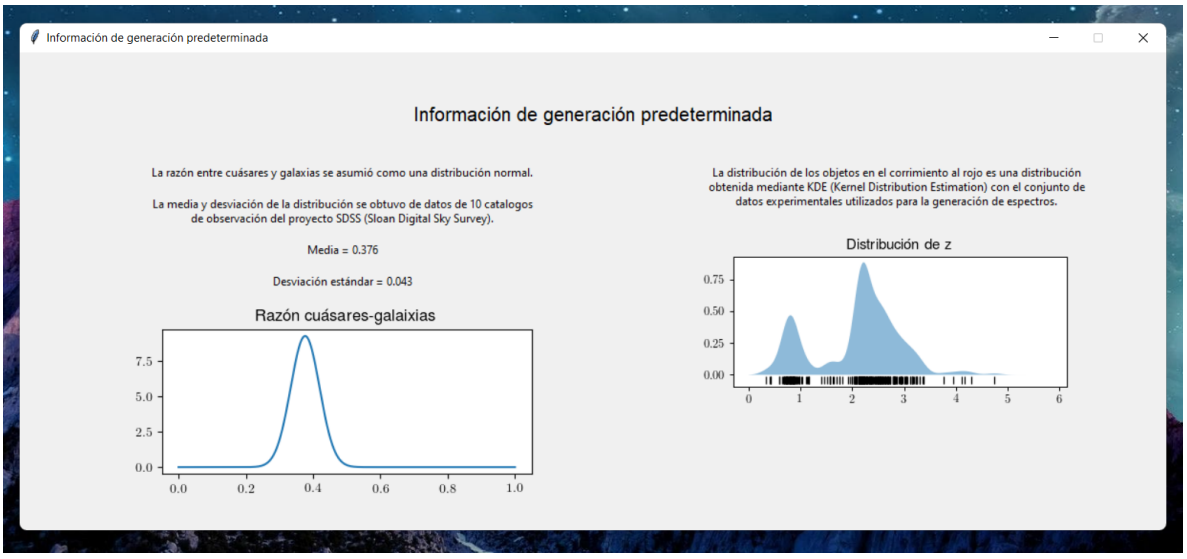
Las funciones más relevantes y el funcionamiento de este módulo se explican y ejemplifican en cuaderno Ejemplos_modulos.py.

Interfaces gráficas (main.py)

Finalmente, el archivo principal en el que se usan las funciones del módulo anterior es main.py. Este archivo abre la interfaz gráfica en la que se muestran los datos generados. Primero se abre un menú de preferencias en donde se requiere que el usuario decida cuántos objetos generar y si quiere generarlos en cantidades personalizadas o con las distribuciones estándar obtenidas del análisis estadístico.

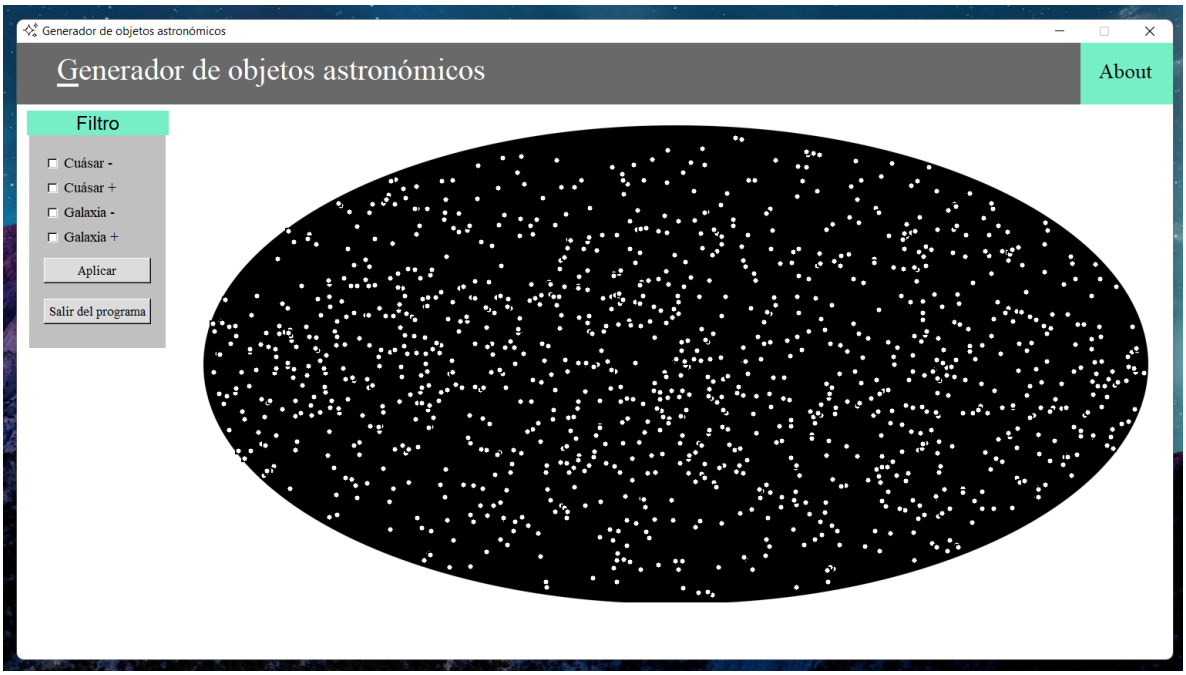


El botón de “Más información” muestra una ventana en la que se explican las distribuciones preestablecidas.

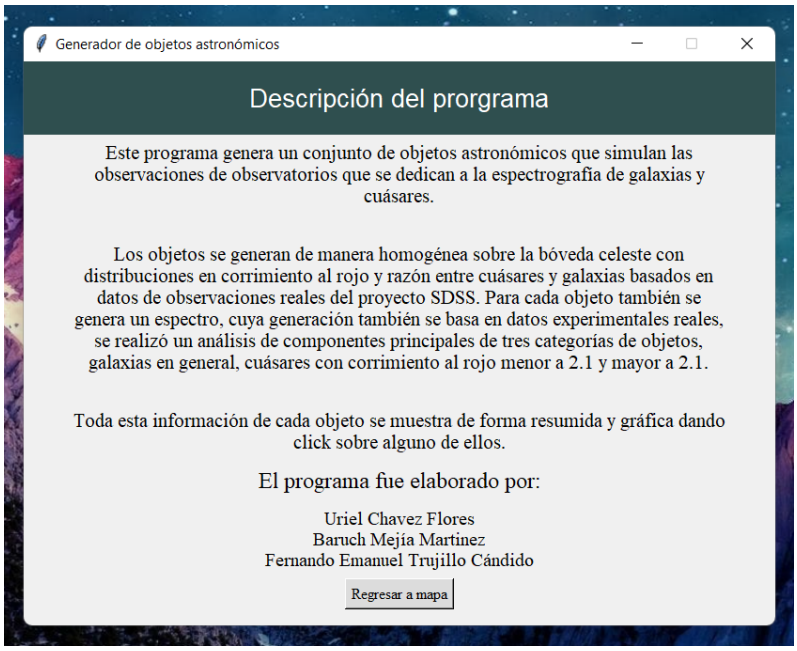


Los espacios donde se ingresan las cantidades están limitados a poder generar a lo más 2000 objetos astronómicos. Una vez se decida la cantidad de objetos en la distribución preestablecida o las cantidades de cada tipo de objeto en la distribución

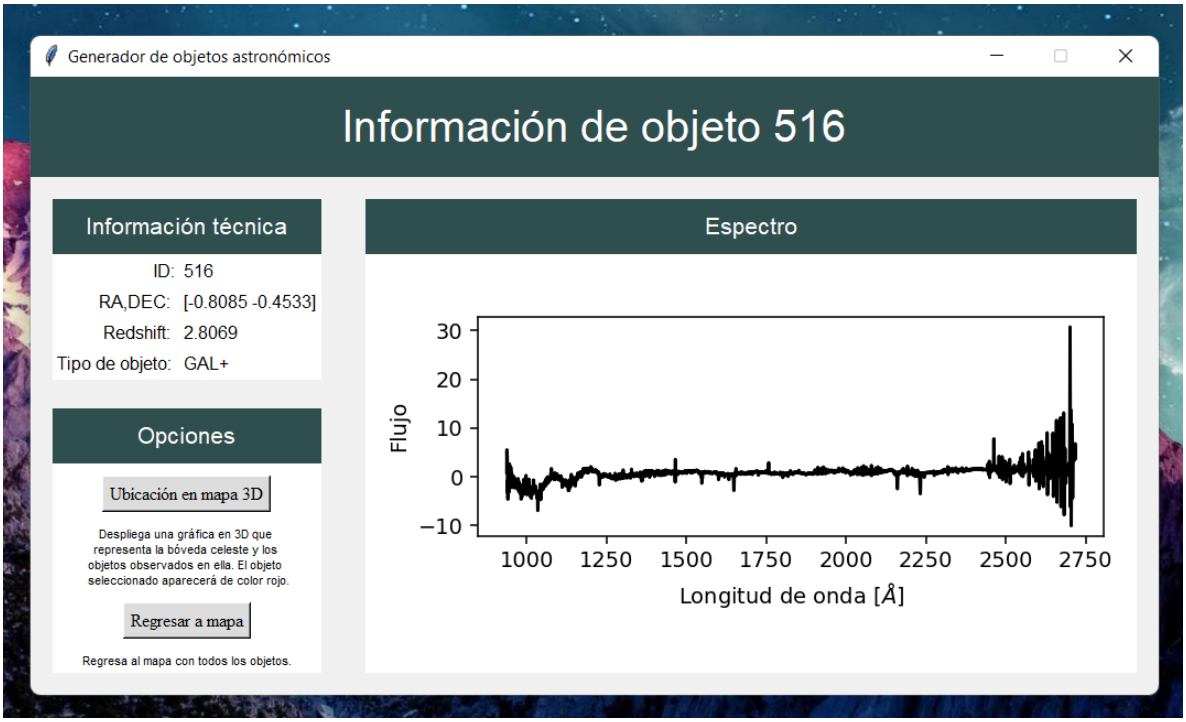
personalizada, el usuario debe dar click en generar, esto cerrará el menú de preferencias y abrirá una interfaz que muestra los objetos en un mapa bidimensional de la bóveda celeste llamado proyección Mollweide. A la izquierda del mapa hay un filtro que permite seleccionar los tipos de objetos que queremos visualizar en el mapa.



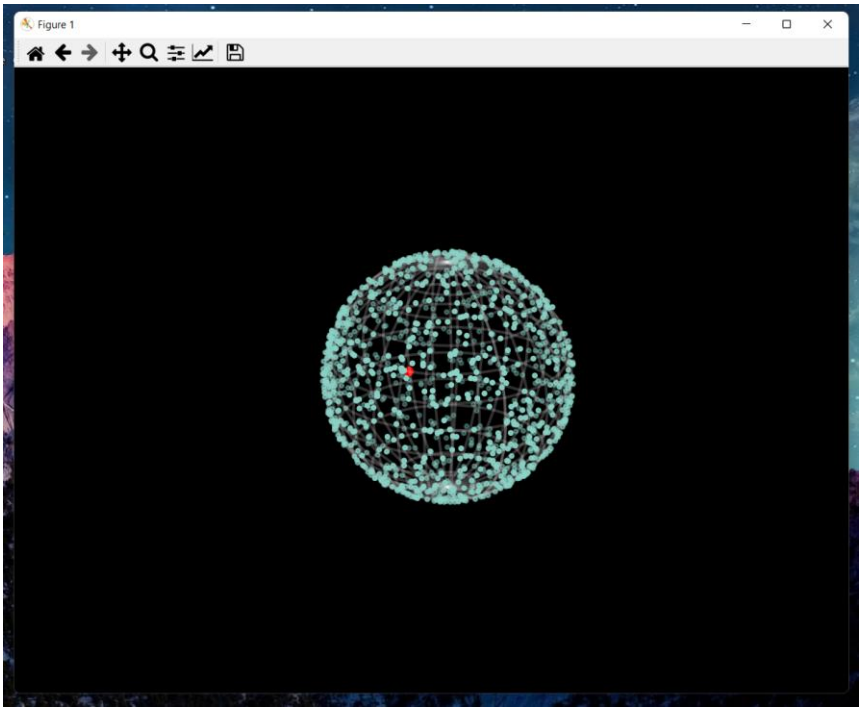
En la esquina derecha superior hay un botón que muestra una breve explicación del programa y la lista de los autores del programa.



El usuario puede dar click sobre los objetos, esto mostrará una ventana con la información del objeto, y una gráfica del espectro generado para dicho objeto.



También da la opción a visualizar el objeto en un mapa 3D interactivo mostrándolo en color rojo haciéndolo resaltar entre los otros puntos.



Todos estos archivos están almacenados en un repositorio de GitHub, en el que se estuvo trabajando durante el proceso de creación por parte de todo el equipo. Todos los archivos necesarios para correr el programa se encuentran en el repositorio.

Enlace al repositorio:

ChavezUriel/ProyectoPOOE (github.com)