```
import matplotlib.pyplot as plt
import numpy as np
import os
import re
from scipy.signal import savgol_filter
from scipy.optimize import curve_fit
import ipywidgets as widgets
from IPython.display import display
import os

class ramanfrom:

def __init__(self, path='./muestras/carpeta', extetion= '.txt'):
if path == '':
raise Exception(" Ingrese una ruta a la carpeta que contiene los espectros:
Ej: './muestras/carpeta' ")
else:
ramanspecs = {}
files = []
ramans = []
for r, d, f in os.walk(path):
for file in f:
if extetion in file:
raman = RamanSpectrum(r+'/'+file)
files.append(r+'/'+file)
ramans.append(raman)
ramanspecs[raman.metadata['Acquired']] = raman

self.path = path
self.ramans = ramans
self.files = files
self.ramanspecs = ramanspecs

def date(self, date):
self.date = date

def pop(self, name=' '):
if name == ' ':
raise Exception("Especifique el nombre del espectro a extraer")
else:
self.ramanspecs.pop(self.date + name)

def randomspec(self):
return self.ramanspecs[np.random.choice(list(self.ramanspecs.keys()))]

def fit(self, method = 'nomethod', window = 30, ord=5,initial_guess = [100,
100, 100]):
if method == 'nomethod':
raise Exception("Especifique el método a utilizar")
else:
if method=='sav_gol':
for k,v in self.ramanspecs.items():
pass
elif method=='poly_fit':
for k,v in self.ramanspecs.items():
pass
elif method=='fit_gauss':


for k,v in self.ramanspecs.items():
pass
else:
raise Exception("Método no disponible")


def gaussian(x, amplitude, mean, stddev):
return amplitude * np.exp(-(x - mean) ** 2 / (2 * stddev ** 2))


# Define the function to fit the entire spectrum
def fit_gaussians(x, *params):
num_peaks = len(params) // 3
```

```
result = np.zeros_like(x)
for i in range(num_peaks):
result += gaussian(x, params[i * 3], params[i * 3 + 1], params[i * 3 + 2])
return result

class RamanSpectrum:

def __repr__(self):
return 'RamanSpectrum (repr): ' + self.filepath

def __str__(self):
return 'RamanSpectrum (str): ' + self.filepath

def __init__(self, filepath, x=None,y=None):
self.filepath = filepath
self.metadata = {}
self.props = {}
self.dictcoords = {}

if (x==None) and (y==None):
"x and y not give, reading a file instead"

with open(filepath, 'r', encoding='iso-8859-1') as f:
lines = f.readlines()

for line in lines:
if line.startswith("#"):
key, value = line.strip().split("=")
self.metadata[key[1:]] = value.replace('\t', '')

self.metakeys = self.metadata.keys()

self.data = np.loadtxt(lines[len(self.metadata):])
self.x = self.data[:,0]
self.y = self.data[:,1]

self.acquired = self.metadata['Acquired']
self.title = self.metadata['Title'].replace(' ','_')

self.sample = self.title + '/' + self.acquired

if not os.path.exists(self.sample):


os.makedirs(self.sample)

for e in self.data:
self.dictcoords[e[0]] = e[1]

"self.metadata"
"self.dictcoords"
"self.metakeys"

def setprops(self,prop, name):
self.props[name] = prop

# Create sliders for frequency, amplitude, and plot interval



def interactive(self, x=[],y=[],mod = False, method='sav_gol',):

proposed_y = []
proposed_x = []

if (x==[]) and (y==[]):
x = self.x
y = self.y
else:
x=x
y=y
```

```
xmin,xmax = min(x),max(x)
ymin,ymax = min(y),max(y)

a_text = widgets.IntText(value=0, description='a:')
b_text = widgets.IntText(value=-1, description='b:')


freq_slider = widgets.IntSlider(min=1, max=50,
value=30,description='Frequency:')
amp_slider = widgets.IntSlider(min=1, max=5, value=3,
description='Amplitude:')

x_min_slider = widgets.FloatSlider(min=xmin, max=xmax, value=xmin,
description='X min:')
x_max_slider = widgets.FloatSlider(min=xmin, max=xmax, value=xmax,
description='X max:')
y_min_slider = widgets.FloatSlider(min=ymin, max=ymax, value=ymin,
description='Y min:')
y_max_slider = widgets.FloatSlider(min=ymin, max=ymax, value=ymax,
description='Y max:')

la = widgets.Text(value='100,100,100', description='list_1:')
lb = widgets.Text(value='b', description='list_2:')

# Function to update the plot based on slider values
def update_plot(freq, amp, x_min, x_max, y_min,
y_max,aa=a_text,bb=b_text,la=la,lb=lb):
if mod:
if method == 'sav_gol':
final_x = x
final_y = y

ny = savgol_filter(y, freq, amp)


proposed_x = x
proposed_y = ny
params=[]
elif method == 'poly_fit':
final_x = x
final_y = y

ny = np.polyfit(x,y, freq)

proposed_y = np.polyval(ny, x)
proposed_x = x
params = []

elif method == 'fit_gauss':
chain = la.replace('[','').replace(']','').split(',')
chain = [int(e) for e in chain]

print(chain)

results = self.fitgaussians(pair='raw',initial_guess=chain,interactive=True)

final_x = x
final_y = y

proposed_x = results[0]
proposed_y = results[1]
params = results[2]
else:
raise Exception("Método no disponible")
else:
raise Exception("No se ha modificado el espectro, añada como parametro mod =
True (method = sav_gol, poly_fit, fit_gauss)")


if params == []:
```

```
pass
else:
# We plot the individual gaussians which parameters are stored in params
print('Gaussian parameters =====')
print(params)
print('Gaussian parameters =====')
for i in range(len(params)//3):
plt.plot(proposed_x,gaussian(proposed_x,params[3*i],params[3*i +
1],params[3*i + 2]))
plt.plot(proposed_x,proposed_y,color='red',label='Proposed')

plt.figure(figsize=(10, 6))
plt.plot(final_x[aa:bb], final_y[aa:bb])
plt.plot(proposed_x, proposed_y, color='red', label='Proposed')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.title(la+' +++ '+lb)
plt.show()

# Create interactive widgets


interactive_plot = widgets.interactive(
update_plot,
freq=freq_slider,
amp=amp_slider,
x_min=x_min_slider,
x_max=x_max_slider,
y_min=y_min_slider,
y_max=y_max_slider,
aa=a_text,
bb=b_text,
)

# Display the widgets
display(interactive_plot)




def plotnsave(self,_dir,_show = True,xy=[],mz = 1, circles = False):
cdir = self.sample+'/'+_dir
if not os.path.exists(cdir):
os.makedirs(cdir)
name = cdir+"/{i}.png".format(i = self.metadata['Date'])

if (xy==[]):
x = self.x
y = self.y
else:
x=xy[0]
y=xy[1]
if circles:
plt.plot(x,y,'o',markersize=mz)
else:
plt.plot(x,y)
plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title(self.metadata['Acquired'])
plt.savefig(name)
if _show:
plt.show()
plt.clf()

def plot(self):
self.plotnsave('raw_plot')
```

```
def acces_value(self, pattern):
matched_keys = set()
regex = re.compile(pattern, re.IGNORECASE)
for key in self.metakeys:
if regex.search(key):
matched_keys.add(key)
return (matched_keys, [self.metadata[key] for key in matched_keys])


def crop(self, lower, upper, show = False):
self.croppedx = []
self.croppedy = []

for e in self.dictcoords.keys():
if e > lower and e < upper:
self.croppedx.append(e)
self.croppedy.append(self.dictcoords[e])
self.plotnsave('crop',_show = show, xy=[self.croppedx,self.croppedy])

def values_close_to(self,ls):
return self.x[np.isclose(self.x[:, None], ls, atol=0.5).any(axis=1)]


def plotpoints(self, intervals, x = []):
if x == []:
x = self.x
y = self.y
else:
y = []
x = self.values_close_to(x)
for e in x:
y.append(self.dictcoords[e])

xp = []
yp = []

for interval in intervals:
xi = self.values_close_to(interval)
for _ in xi:
xp.append(_)
yp.append(self.dictcoords[_])

plt.plot(x,y)
plt.plot(xp,yp, 'o',color = 'red')
return xp,yp;



def polyfit(self, x=[],y=[], mod = "raw",ord = 3):

'''
The method provides a polyfit from numpy
'''


if x == [] or y == []:
if mod == "cropped":
x_ = self.croppedx
y_ = self.croppedy
elif mod == "raw":
x_ = self.x
y_ = self.y
else:
raise Exception("No cropped data")


else:
x_ = x
y_ = y
```

```
fitted = np.polyfit(x_, y_, ord)

self.fitted = fitted

plt.subplot(2, 1, 1)
plt.plot(x_, y_)
plt.plot(x_, np.polyval(fitted, x_))
plt.xlim(min(x_), max(x_))
ef = abs(max(y_) - min(y_))/20
plt.ylim(min(y_)-ef, max(y_)+ef)
plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title('With baseline _._._: '+self.metadata['Acquired'])
# Plotting the second part
plt.subplot(2, 1, 2) # Create subplot 2
plt.plot(x_, y_ - np.polyval(fitted, x_))
self.polylx = x_
self.polyly = y_ - np.polyval(fitted, x_)
plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title('Without baseline')

# Saving both plots in the same file
if not os.path.exists(self.sample+'/polyfit'):
os.makedirs(self.sample+'/polyfit')
plt.savefig(self.sample+"/polyfit/{i}.png".format(i=self.metadata['Date']))
plt.subplots_adjust(hspace=1)
plt.show()



def sav_gol(self, x=[], y=[], window = 20, order=4, show=True):
if x == [] or y == []:
x = self.x
y = self.y
else:
x = x
y = y

self.denoisedx = []
self.denoisedy = []


self.savgol = savgol_filter(y, window, order)
self.denoisedx = x
self.denoisedy = self.savgol

plt.plot(self.denoisedx, self.denoisedy, 'o', markersize=0.5)


self.basedx = self.denoisedx
self.basedy = self.denoisedy

plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title(self.metadata['Acquired'])

if not os.path.exists(self.sample+'/denoised'):
os.makedirs(self.sample+'/denoised')

# Saving the cropped data as an image
name = self.sample + "/denoised/{i}.png".format(i = self.metadata['Date'])
plt.savefig(name)
if show:
plt.show()
else:
self.plotnsave(_dir='denoised_and_baselined',_show=False,circles=True,mz=1)
plt.clf()
```

```
# Define the function to fit the peaks
# Perform curve fitting
# The * before initial_guess is to unpack the list
# The ** before test is to unpack the dictionary

def fitgaussians(self, pair='raw', initial_guess = [100, 100, 100, 200, 200,
200, 400, 400, 400], interactive = False):

if pair == 'denoised':
x = self.denoisedx
y = self.denoisedy
elif pair == 'cropped':
x = self.croppedx
y = self.croppedy
elif pair == 'raw':
x = self.x
y = self.y
else:
raise Exception("Especifique de donde se obtendrán los datos")

params, _ = curve_fit(fit_gaussians, x, y, p0=initial_guess)

# # Extract individual peak parameters
num_peaks = len(params) // 3
peak_params = []

for i in range(num_peaks):
peak_params.append((params[i * 3], params[i * 3 + 1], params[i * 3 + 2]))

# Lambda function to round a float number to an integer

round2int = lambda x: int(round(x))

# # Print the peak parameters


lss = []
for i, (amplitude, mean, stddev) in enumerate(peak_params):
print(f"Peak {i+1}: Amplitude={amplitude}, Mean={mean}, Stddev={stddev}")

lss.append(round2int(amplitude))
lss.append(round2int(mean))
lss.append(round2int(stddev))

# # Print the fitted function parameters
print("\n")
print(lss)
print("\n")

self.gaussbasedx = x
self.gaussbasedy = y - fit_gaussians(x, *params)

self.multiparams = params
self.fitedparamsx = x
self.fitedparamsy = fit_gaussians(x, *params)

if interactive:

return [x, fit_gaussians(x, *params), params, _]

else:
# # Plot the original spectrum and the fitted curve
plt.figure(figsize=(8, 6))
plt.title(self.metadata['Acquired'])
plt.plot(x, y, label='Original Spectrum')
# plt.plot(x, fit_gaussians(x, *params), color='red',label='Fitted Curve')

# # Plot the individual peaks
for i, (amplitude, mean, stddev) in enumerate(peak_params):
plt.plot(x, gaussian(x, amplitude, mean, stddev), label=f'Peak {i+1}')
```

```
plt.plot(x, fit_gaussians(x, *params), color='red',label='Fitted Curve')
plt.xlabel('X')
plt.ylabel('Intensity')
plt.legend()
plt.show()



def baseline(self,degree = 1, show = False, before=False):
# Fit polynomial baseline

xfit = self.denoisedx[:5] + self.denoisedx[-5:]
self.a = list(self.denoisedy[:5])
self.b = list(self.denoisedy[-5:])

print(self.a)
print(self.b)

yfit = self.a + self.b


print(type(self.denoisedx),type(self.denoisedy))
print('The lenghts',len(xfit),len(yfit))

coefficients = np.polyfit(xfit, yfit, degree)
baseline = np.polyval(coefficients, self.denoisedx)

# Plot the original signal and the baseline
# plt.plot(self.denoisedx, self.denoisedy, label='Original Signal')
new_zero = abs(min(self.denoisedy - baseline))
if before:
plt.plot(self.denoisedx, (self.croppedy + new_zero) , label='baselined')
plt.plot(self.denoisedx, baseline, label='Baseline')
else:
plt.plot(self.denoisedx, (self.denoisedy + new_zero), label='baselined')
plt.plot(self.denoisedx, baseline, label='Baseline')
self.basedx = self.denoisedx
self.basedy = (self.denoisedy + new_zero) - baseline
# plt.plot(self.denoisedx, baseline, label='Baseline')
plt.legend()
plt.xlabel('wavenumber (cm$^{-1}$)')
plt.ylabel('Intensity (counts)')

if not os.path.exists(self.sample+'/baseline'):
os.makedirs(self.sample+'/baseline')

plt.title(self.metadata['Date'] + ' - Baseline')
if show:
plt.show()
name = self.sample + "/baseline/{i}.png".format(i = self.metadata['Date'])
plt.savefig(name)
plt.clf()




# Define the function as the sum of three Gaussian curves
def gaussian(self, x, amplitude, center, sigma):
return amplitude * np.exp(-(x - center)**2 / (2 * sigma**2))

def multi_peak_fit(self, x, *params):
num_peaks = len(params) // 3
y_fit = np.zeros_like(x)

for i in range(num_peaks):
amplitude, center, sigma = params[i*3 : (i+1)*3]
y_fit += self.gaussian(x, amplitude, center, sigma)
```

```
return y_fit


def get_fitting(self, f2 = 0.5, f3 = 0.3 , c1 = 520, c2 = 500, c3 = 480, s1 =
10, s2 = 20, s3 = 40 ,show = False):
x = np.array(self.basedx)
y = np.array(self.basedy)


yspec = max(y)
yspec2 = yspec*f2
yspec3 = yspec*f3

# Perform the multi-peak fitting
# initial_guess = [yspec3, 450, 100, yspec2, 510, 10, yspec, 520, 10]
initial_guess = [yspec3, c3, s3, yspec2, c2, s2, yspec, c1, s1] # Initial
guess for parameters: [amplitude1, center1, sigma1, amplitude2, center2,
sigma2, amplitude3, center3, sigma3]

if not os.path.exists(self.sample+'/fit'):
os.makedirs(self.sample+'/fit')

popt, pcov = curve_fit(self.multi_peak_fit, x, y, p0=initial_guess)

# Extract the optimized parameters
amplitudes = popt[0::3]
centers = popt[1::3]
sigmas = popt[2::3]

for i in range(0,3):
amp,cen,sig = popt[i*3:(i+1)*3]

self.fit_props = popt

# Print the results

print('Amplitudes: {}'.format(amplitudes))
print('Centers: {}'.format(centers))
print('Sigmas: {}'.format(sigmas))

# Generate the fitted curve
x_fit = np.linspace(x.min(), x.max(), 1000)
y_fit = self.multi_peak_fit(x_fit, *popt)
# Plot the original data and the fitted curve
plt.plot(x, y, 'bo', label='Original Data')
plt.plot(x_fit, y_fit, 'r-', label='Fitted Curve')
self.x_fit = x_fit
self.y_fit = y_fit
plt.legend()
plt.xlabel('wavenumber (cm$^{-1}$)')
plt.ylabel('Counts (a.u.)')
if show:
plt.show()
plt.savefig(self.sample+'/fit/{}.png'.format(self.metadata['Date']))
self.popt = popt
plt.clf()

def getgaussfit(self, x,y):
self.gx = x
self.gy = y


def get_2_fitting(self,sca = 30, f2 = 0.5,c1 = 520,c2 = 500,s1 = 10,s2 =
20,show = False, case = 'crop'):
if case == 'crop':
x = np.array(self.croppedx)
y = np.array(np.array(self.croppedy)-sca)


else:
raise AssertionError("Must select c x and y")
```

```
yspec = max(y)
yspec2 = yspec*f2

# Perform the multi-peak fitting
# initial_guess = [yspec3, 450, 100, yspec2, 510, 10, yspec, 520, 10]
initial_guess = [yspec2, c2, s2, yspec, c1, s1] # Initial guess for
parameters: [amplitude1, center1, sigma1, amplitude2, center2, sigma2,
amplitude3, center3, sigma3]

if not os.path.exists(self.sample+'/fit'):
os.makedirs(self.sample+'/fit')

popt, pcov = curve_fit(self.multi_peak_fit, x, y, p0=initial_guess)

# Extract the optimized parameters
amplitudes = popt[0::2]
centers = popt[1::2]
sigmas = popt[2::2]

for i in range(0,2):
amp,cen,sig = popt[i*3:(i+1)*3]

self.fit_props = popt

# Print the results

print('Amplitudes: {}'.format(amplitudes))
print('Centers: {}'.format(centers))
print('Sigmas: {}'.format(sigmas))

# Generate the fitted curve
x_fit = np.linspace(x.min(), x.max(), 1000)
y_fit = self.multi_peak_fit(x_fit, *popt)
# Plot the original data and the fitted curve
plt.plot(x, y, 'bo', label='Original Data')
plt.plot(x_fit, y_fit, 'r-', label='Fitted Curve')
self.x_fit = x_fit
self.y_fit = y_fit
plt.legend()
plt.xlabel('wavenumber (cm$^{-1}$)')
plt.ylabel('Counts (a.u.)')
if show:
plt.show()
plt.savefig(self.sample+'/fit/{}.png'.format(self.metadata['Date']))
self.popt = popt
plt.clf()


def fit_intervals(self,lss,od):
x,y = self.plotpoints(lss)
fitted = np.polyfit(x, y, od)
self.chis = np.polyfit(x, y, od, full=True)
self.fitted = fitted
# Plotting the first part


plt.subplot(2, 1, 1) # Create subplot 1
plt.plot(self.croppedx, self.croppedy)
plt.plot(self.croppedx, np.polyval(fitted, self.croppedx))
plt.xlim(min(self.croppedx), max(self.croppedx))
ef = abs(max(self.croppedy) - min(self.croppedy))/20
plt.ylim(min(self.croppedy)-ef, max(self.croppedy)+ef)
plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title('With baseline')
# Plotting the second part
plt.subplot(2, 1, 2) # Create subplot 2
self.polylx = self.croppedx
fitedcurve = np.polyval(fitted, self.croppedx)
self.fitedcurve = fitedcurve
```

```
self.polyly = (self.croppedy - fitedcurve) + abs(min(self.croppedy -
fitedcurve))
plt.plot(self.polylx,self.polyly)
print(min(self.polyly))
plt.xlabel("Wavenumber (cm$^{-1}$)")
plt.ylabel("Intensity (counts)")
plt.title('Without baseline')

# Saving both plots in the same file
if not os.path.exists(self.sample+'/polyfit'):
os.makedirs(self.sample+'/polyfit')
plt.savefig(self.sample+"/polyfit/{i}.png".format(i=self.metadata['Date']))
plt.subplots_adjust(hspace=1)
plt.show()
```