XRD File reader and plotter

El código está diseñado para convertir archivos en formato uxd de una máquina XRD a archivos CSV. Estos archivos se utilizan posteriormente para su representación gráfica y análisis de propiedades como: FWHM, theta y tamaño de cristalito utilizando las fórmulas de Debye-Scherrer y Williamson-Hall, porcentaje de cristalinidad y tipo de esfuerzo. Todo esto se realiza utilizando el lenguaje de programación Python.

Temas: XRD, conversión de archivos, análisis de datos, Debye-Scherrer, Williamson-Hall, Python.

In [1]:
```python
# Importamos librerias necesarias para nuestro programa
import pandas as pd
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
from IPython.display import display, Markdown
from scipy.signal import find_peaks
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from scipy.optimize import curve_fit
```

Lectura del Archivo Crudo

En esta sección del código, se realiza la lectura del archivo crudo proveniente del software del equipo de XRD. El objetivo es procesarlo para generar un archivo de salida en formato CSV.

In [2]:
```python
file = open('../SrTiO3.uxd', mode='r')

content = file.read()

partes_importantes = content.split(';')

tabla_contenido = partes_importantes[7]

titles = tabla_contenido[1:15]

tabla_contenido = tabla_contenido.replace(titles, '2THETA, PSD\n')
tabla_contenido = tabla_contenido.replace('       ', ', ')
tabla_contenido = tabla_contenido.replace('      ', ', ')

file.close()
```

Conversión del Archivo

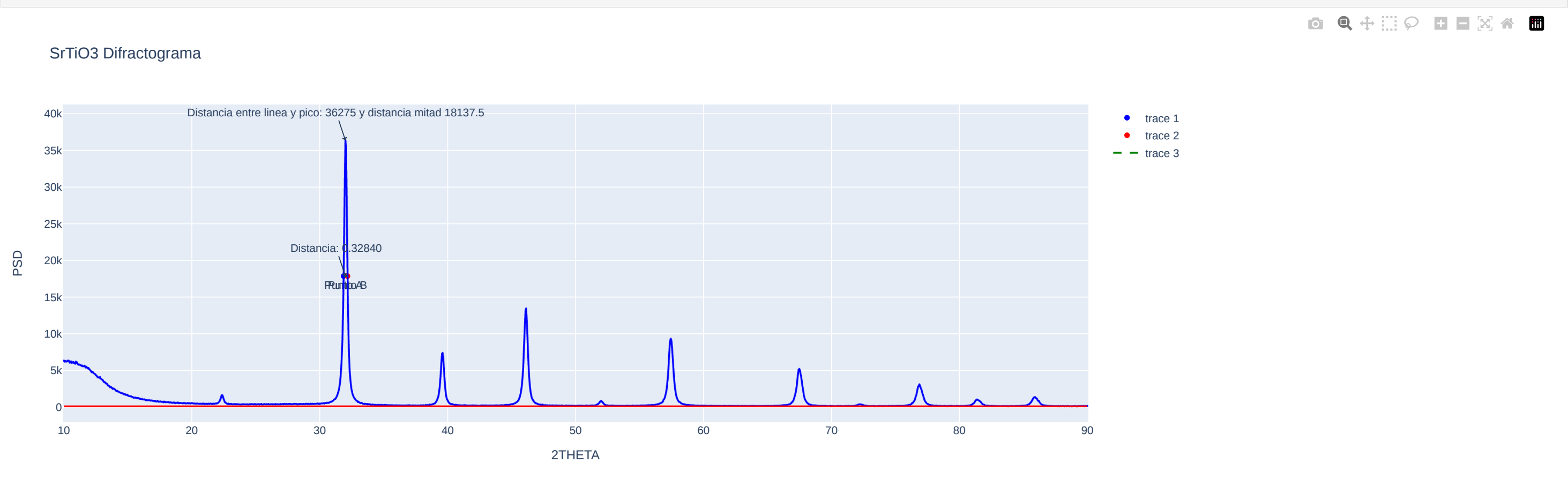
En esta sección del código, el contenido del archivo crudo se transforma en un archivo CSV para luego ser abierto con pandas.

In [3]:
```python
output_file = open('data.csv', 'w')

output_file.write(tabla_contenido)

output_file.close()
```

Generación de Gráfica para la Fórmula de Debye-Scherrer

En esta sección del código, se extrae la altura del pico más prominente de la gráfica para analizar su FWHM y, a continuación, se aplica la fórmula de Debye-Scherrer para calcular el tamaño del cristalito.

In [4]:
```python
df = pd.read_csv('data.csv')

y_position_value_global = 100
beta_constant_value = 0


fig = px.line(
    df, x=' 2THETA',
    y=' PSD',
    labels={'Name': '2theta', 'Value': 'values'},
    title='SrTiO3 Difractograma'
)


fig.update_traces(line=dict(color='blue'))

# Agragamos una linea de referencia
fig.add_shape(
    type='line',
    x0=df[' 2THETA'].min(),
    y0=y_position_value_global,
    x1=df[' 2THETA'].max(),
    y1=y_position_value_global,
    line=dict(color='red', width=2, dash='solid')
)

#-------------------------------------------------
#Se calcula la altura máxima en de la reflexión más grande:
altura_reflexion_max = max(df[' PSD'].values) - y_position_value_global
# indice reflexión más alta
indice_refle_mas_alta = df[df[' PSD'] == max(df[' PSD'].values)].index[0]
#print(indice_refle_mas_alta)
# Obtenemos la mitad de la distancia con de la reflexión mas grande usando el punto de referencia
given_PSD_value = altura_reflexion_max/2

# Se calcula la diferencia absoluta entre los dos valores dados y todos los datos de la columna PSD.
df['Absolute_Difference'] = abs(df[' PSD'] - given_PSD_value)


# Encuentra la fila con la menor diferencia encontrada
closest_index = df['Absolute_Difference'].idxmin()

closest_2THETA_value = df.loc[closest_index, ' 2THETA']
closest_PSD_value = df.loc[closest_index, ' PSD']

# Encontramos el siguiente valor más cercano
next_upper_values = df[df[' PSD'] > given_PSD_value]
if not next_upper_values.empty:
    next_upper_index = next_upper_values[' PSD'].idxmin()
    next_upper_2THETA = df.loc[next_upper_index, ' 2THETA']
    next_upper_PSD = df.loc[next_upper_index, ' PSD']

#-------------------------------------------------

promedio_de_dos_puntos = ((df[' 2THETA'][next_upper_index] + df[' 2THETA'][next_upper_index + 1])/2) + 0.003

# Agregamos dos puntos con las coordenadas de closes index y
# el promedio de 2theta en el next_upper_index y next_upper_index + 1

fig.add_trace(
    go.Scatter(
        x=[df[' 2THETA'][closest_index]],
        y=[df[' PSD'][closest_index]],
        mode='markers+text', text='Punto A',
        textposition='bottom center',
        marker=dict(color='blue')
    )
)


fig.add_trace(
    go.Scatter(
        x=[promedio_de_dos_puntos],
        y=[df[' PSD'][closest_index]],
        mode='markers+text', text='Punto B',
        textposition='bottom center',
        marker=dict(color='red')
    )
)


# Creamos una linea entre esos cos puntos
fig.add_trace(
    go.Scatter(
        x=[df[' 2THETA'][closest_index], promedio_de_dos_puntos],
        y=[df[' PSD'][closest_index], df[' PSD'][closest_index]],
        mode='lines', line=dict(color='green', dash='dash')
    )
)


# Se calcula la distancia con la formula euclidiana
def calculate_fwhm(index):
    half_max = df[' PSD'][index] / 2  # Half of the peak's maximum value
    left_bound = np.where(df[' PSD'][:index] < half_max)[0][-1]  # Left boundary
    right_bound = np.where(df[' PSD'][index:] < half_max)[0][0] + index  # Right boundary
    fwhm = df[' 2THETA'][right_bound] - df[' 2THETA'][left_bound]  # FWHM calculation
    return fwhm

beta_constant_value = calculate_fwhm(indice_refle_mas_alta)  #Se guardan los datos en la variable global
distance_two_points = beta_constant_value
# Se agrega una anotación con esta distancia
fig.add_annotation(
    x=(next_upper_2THETA + closest_2THETA_value)/2,
    y=df[' PSD'][closest_index],
    text=f'Distancia: {distance_two_points:.5f}', # Ponemos la distancia con 3 puntos decimales
    showarrow=True,
    arrowhead=1,
)

fig.add_annotation(
    x=(next_upper_2THETA + closest_2THETA_value)/2,
    y=df[' PSD'][closest_index + 3],
    text=f'Distancia entre linea y pico: {altura_reflexion_max} y distancia mitad {altura_reflexion_max/2}',
    # Ponemos la distancia con 3 puntos decimales
    showarrow=True,
    arrowhead=1,
)

fig.show(renderer='notebook')
```



Cálculo de tamaño de cristalito a partir de la formula de Debye-Scherrer

$$D = \frac{K\lambda}{\beta cos(\theta)}$$

Donde:

- K: Factor de estructura (0.9 para perovskitas [2])
- λ: Longitud de onda CuKα (1.5406 Å)
- β: Distancia entre **Punto A** y **Punto B**
- D: Tamaño de cristalito

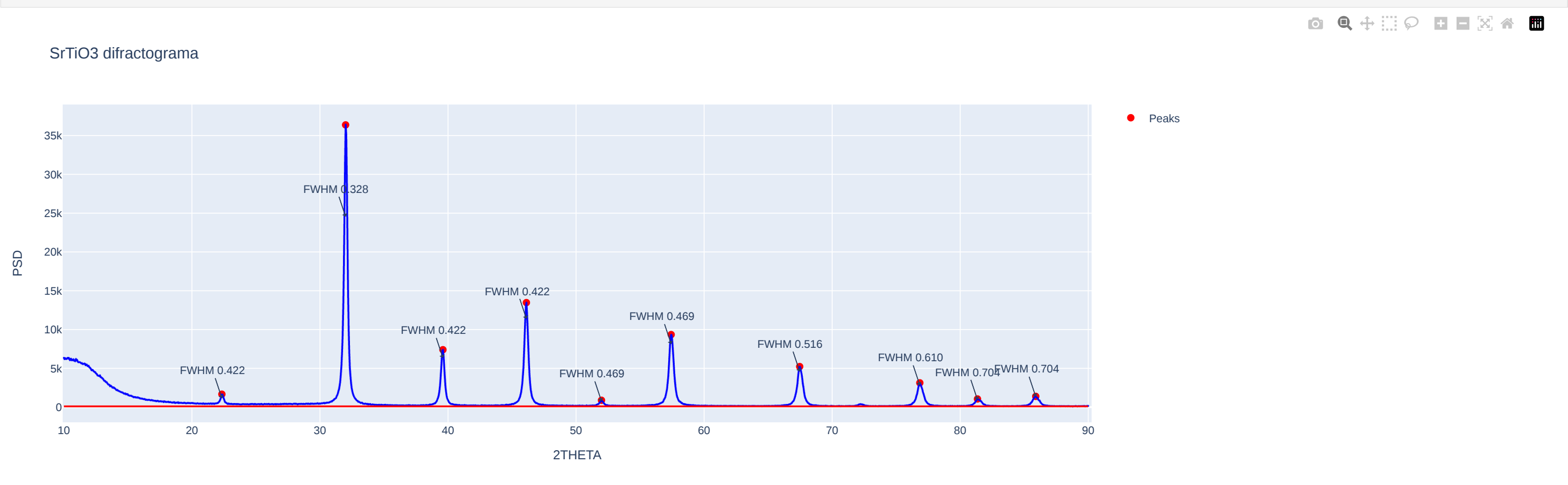
In [5]:
```python
def calc_tamanio_cristalito_scherrer (beta, theta):
    K = 0.89 # Para cúbicas según la referencia
    LAMBDA = 1.5406 # Longitud de onda Cobre K alfa

    #print(theta)
    theta_rads = np.deg2rad(theta)
    angulo = np.cos(theta_rads)
    #print(angulo)
    cristalito_size = (K * LAMBDA)/(beta * angulo)
    #print(f'Tamaño de cristalito calculado: {cristalito_size.values[0]} Å')
    return cristalito_size.values[0] * 0.1 # Se multiplica por 0.1 para convertir A a nm

theta_scherrer = df[df[' PSD'] == max(df[' PSD'].values)][' 2THETA']
#print(theta_scherrer/2)
display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:5px;" >
Tamaño de cristalito calculado:
    <b> {calc_tamanio_cristalito_scherrer(np.deg2rad(distance_two_points),(theta_scherrer/2))} nm </b>
</div>

&nbsp;

'''))
```

Tamaño de cristalito calculado: **24.886226598966275 nm**

Generación de Gráfica para la Fórmula de Williamson-Hall

En esta sección del código, se identifican todos los picos de la gráfica para luego determinar el FWHM de cada uno, permitiendo la capacidad de discriminar entre picos para mejorar la precisión de los datos. Posteriormente, se crea la gráfica de $\beta \cdot \cos(\theta)$ vs $\sin(\theta)$ y se aplica una regresión lineal para determinar el tipo de esfuerzo, lo que facilita el cálculo del tamaño de cristalito mediante la fórmula de Williamson-Hall.

In [7]:
```python
# Encontramos los picos más altos en la gráfica
peaks, _ = find_peaks(df[' PSD'], prominence=50 , distance= 90)  # Adjust prominence as needed
peaks = np.delete(peaks, 0)
peaks = np.delete(peaks, 1)
peaks = np.delete(peaks, 7)

y_position_value_global = 100
beta_constant_value = 0

lista_picos_index = []

for i in peaks:
    lista_picos_index.append(i)

fig = px.line(
    df,
    x=' 2THETA',
    y=' PSD',
    labels={'Name': '2theta', 'Value': 'values'},
    title='SrTiO3 difractograma'
)

fig.add_scatter(
    x=df[' 2THETA'][peaks],
    y=df[' PSD'][peaks],
    mode='markers',
    marker=dict(color='red', size=8),
    name='Peaks'
)

fig.update_traces(line=dict(color='blue'))



# Agregamos una linea horizontal que sirve como punto de referencia
fig.add_shape(
    type='line',
    x0=df[' 2THETA'].min(),
    y0=y_position_value_global,
    x1=df[' 2THETA'].max(),
    y1=y_position_value_global,
    line=dict(color='red', width=2, dash='solid')
)

#-------------------------------------------------


def calculate_fwhm(index):
    half_max = df[' PSD'][index] / 2  # Altura media de valor máximo
    left_bound = np.where(df[' PSD'][:index] < half_max)[0][-1]  # Lado izquierdo
    right_bound = np.where(df[' PSD'][index:] < half_max)[0][0] + index  # Lado derecho
    fwhm = df[' 2THETA'][right_bound] - df[' 2THETA'][left_bound]  # FWHM calculo
    return fwhm

def largo_pico(distancia, df_f, peaks):

    fig.add_annotation(
        x=df_f[' 2THETA'][peaks],
        y=df_f[' PSD'][peaks - 2],
        text=f'FWHM {distancia:.3f}', # Ponemos la distancia con 3 puntos decimales
        showarrow=True,
        arrowhead=1,
    )


fwhm_values = []
angulos_peaks = []

for peak_index in peaks:
    fwhm = calculate_fwhm(peak_index)
    angulos_peaks.append(df[' 2THETA'][peak_index])
    fwhm_values.append(fwhm)

for i in range(len(lista_picos_index)):
    largo_pico(fwhm_values[i], df, lista_picos_index[i])


fig.show(renderer='notebook')
```



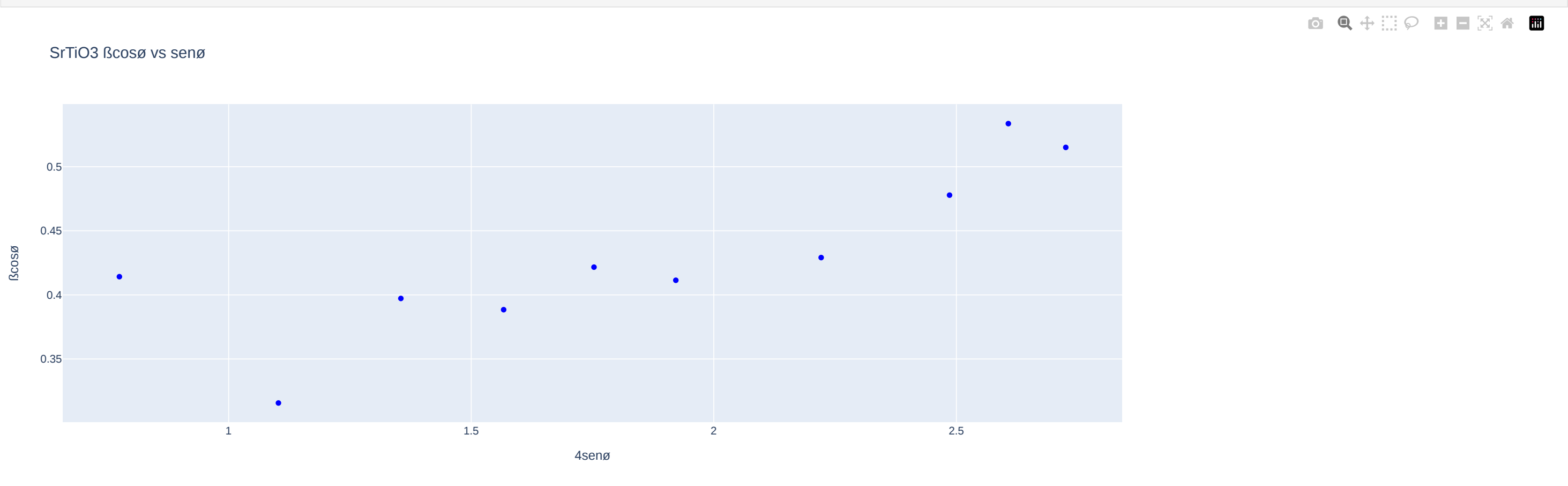
Se obtienen los datos necesarios para hacer la gráfica $\beta cos\theta$ vs $sen\theta$

In [8]:
```python
lista_de_angulos_2theta =  angulos_peaks
lista_de_angulos_theta = []
for angulo in lista_de_angulos_2theta:
    lista_de_angulos_theta.append(angulo/2)
```

In [9]:
```python
sen_angulos_plot = []
cos_angulos_plot = []
contador = 0
for angulo in lista_de_angulos_theta:

    # Convertimos de grados a radianes
    deg_angle_sen = np.sin(np.deg2rad(angulo))
    deg_angle_cos =  np.cos(np.deg2rad(angulo))

    sen_angulos_plot.append(4 * deg_angle_sen)
    cos_angulos_plot.append(fwhm_values[contador] * deg_angle_cos)
    contador += 1
```

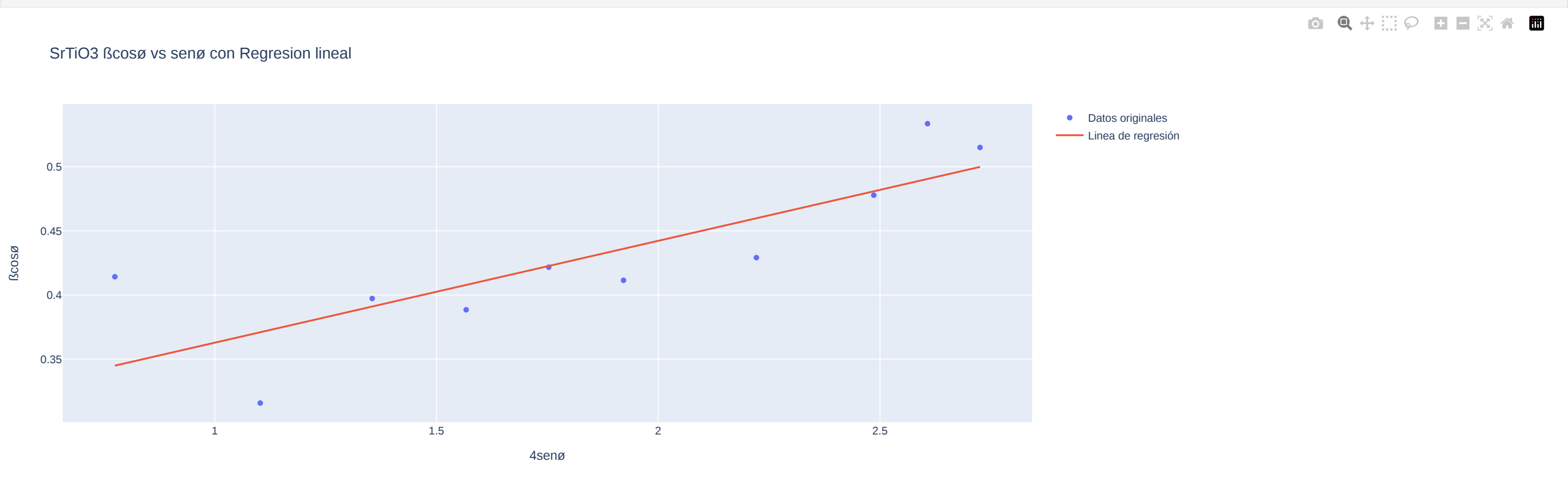
Se grafíca $\beta cos\theta$ vs $sen\theta$

In [10]:
```python
fig = go.Figure()
fig.add_trace(go.Scatter(x=sen_angulos_plot, y=cos_angulos_plot, mode='markers', marker=dict(color='blue')))

fig.update_layout(title='SrTiO3 ßcosø vs senø', xaxis_title='4senø', yaxis_title='ßcosø')

fig.show(renderer='notebook')
```



Regresión lineal de los datos

In [11]:
```python
# Se crean arreglos de numpy con las listas de los angulos senos y cosenos ya tratados
x = np.array(sen_angulos_plot)
y = np.array(cos_angulos_plot)

# Se hace la regresión lineal con numpy
slope, intercept = np.polyfit(x, y, 1)  # 1 for linear regression

# Creamos la función de la linea de regresión con la variable slope e intercept
regression_line = slope * x + intercept

# Creamos una gráfica
fig = go.Figure()

# Graficamos los puntos originales
fig.add_trace(go.Scatter(x=x, y=y, mode='markers', name='Datos originales'))

# Graficamos la linea de regresión
fig.add_trace(go.Scatter(x=x, y=regression_line, mode='lines', name='Linea de regresión'))

# Agregamos las etiquetas
fig.update_layout(
    title='SrTiO3 ßcosø vs senø con Regresion lineal',
    xaxis_title='4senø',
    yaxis_title='ßcosø'
)
equation = f'y = {slope}x + {intercept}'

# Calculamos R cuadrada
r_squared = r2_score(y, regression_line)
# Mostramos la gráfica
fig.show(renderer='notebook')

display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px;" >
Ecuación obtenidad: {f'y = {slope}'}<b> x </b>+ {f'{intercept}'} </b>
</div>
<div align="center" style="background-color:yellow; padding-top:3px;" >
$R^2$= {r_squared} </b>
</div>

&nbsp;

'''))
```



Ecuación obtenidad: y = 0.07948605932788505 **x** + 0.2832693357361943

R^2= 0.6759802329862871

Calculo del cristalito utilizando la fórmula de Williamson-hall

Para calcular el cristralito a través de este método tenemos que utilizar la siguiente expresión

$$\beta_{hkl}cos\theta = \frac{K\lambda}{D} + 4\epsilon sen\theta$$

si observamos bien podemos darnos cuenta que la fórmula adopta la forma:

$$y = mx + b$$

Haciendo la regresión lineal de nuestros datos podemos obtener la siguiente expresión

$$y = 0.11535040977386593x + 0.24479779320473088$$

por lo que podemos decir que:

$$\frac{K\lambda}{D} = 0.24479779320473088$$

Asumiendo que:

- K = 0.9
- λ = 1.5406 Å

Podemos despejar y obtener el resultado, tal que:

In [12]:
```python
K = 0.9 # Para cúbicas según la referencia
LAMBDA = 1.5406 # Longitud de onda Cobre K alfa

wh_cristalito = (K * LAMBDA)/intercept


display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:5px;" >
    Tamaño de cristalito calculado por Williamson-hall : <b> {wh_cristalito} nm </b>
</div>

'''))



if slope > 0:
    display(Markdown(f'''
<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:5px;" >
    La pendiente es positiva por lo tanto podemos argumentar que el sistema tiene tiene una <b> tensión </b>
</div>
    '''))
else:
    display(Markdown(f'''
<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:5px;" >
    La pendiente es positiva por lo tanto podemos argumentar que el sistema tiene tiene una <b> compresión </b>
</div>

    '''))
```

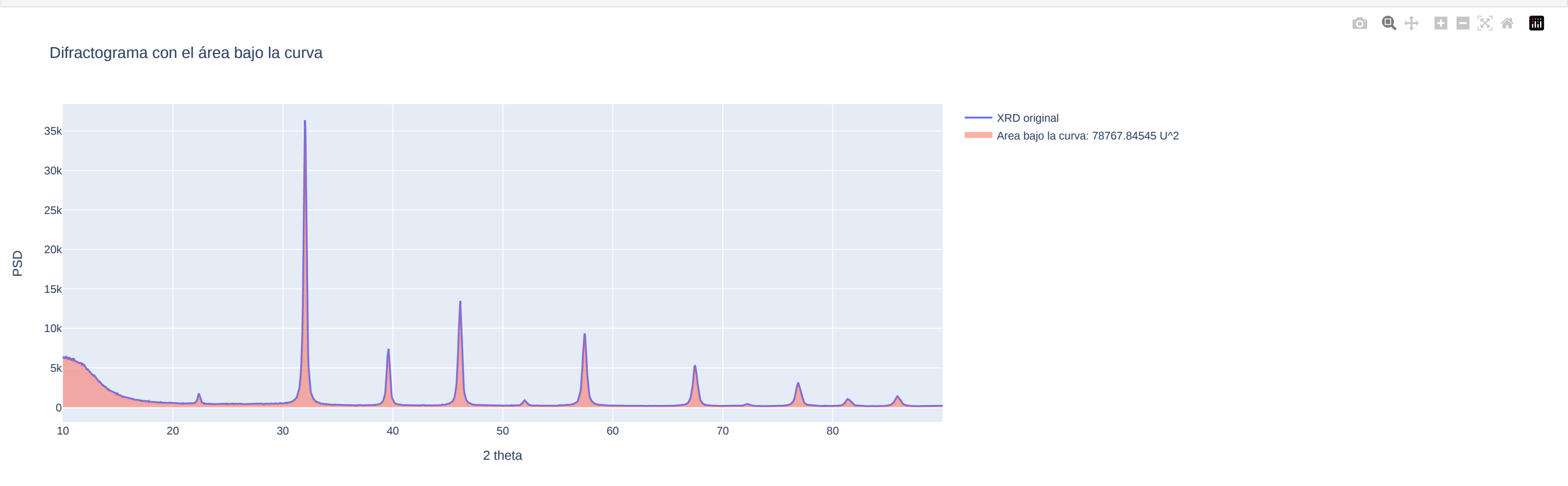
Tamaño de cristalito calculado por Williamson-hall : **4.894776190287218 nm**

La pendiente es positiva por lo tanto podemos argumentar que el sistema tiene tiene una **tensión**

Cálculo del Porcentaje de Cristalinidad

Para determinar el porcentaje de cristalinidad, el primer paso consiste en calcular el área bajo la curva de nuestra gráfica. Gracias a la facilidad que ofrece la librería NumPy, este proceso se simplifica utilizando la función `np.trapz()`.

In [13]:
```python
x_values = df[' 2THETA'].values
y_values = df[' PSD'].values


# Se calcula el área bajo la curva (regla trapezoidal)
area_total = np.trapz(y_values, x=x_values)

# Se grafíca la señal del difractograma
fig = go.Figure()
fig.add_trace(go.Scatter(x=x_values, y=y_values, mode='lines', name='XRD original'))

# Creamos un poligono que dibuje el área bajo la curva
x_polygon = np.concatenate([x_values, x_values[::-1]])
y_polygon = np.concatenate([y_values, np.zeros_like(y_values[::-1])])

fig.add_trace(go.Scatter(
    x=x_polygon,
    y=y_polygon,
    fill='tozeroy',
    mode='none',
    fillcolor='rgba(250, 100, 80, 0.5)',
    name=f'Area bajo la curva: {area_total} U^2'
))

fig.update_layout(
    title='Difractograma con el área bajo la curva',
    xaxis_title='2 theta',
    yaxis_title='PSD'
)

# Show the plot
fig.show(renderer='notebook')

display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:16px;" >
    <h3>
        Area bajo la curva obtenida: {area_total} $U^2$
    </h3>
</div>


&nbsp;

'''))
```



Area bajo la curva obtenida: 78767.84545 U^2

Se obtiene el area bajo la curva pero solo de los picos de la gráfica

In [14]:
```python
# Calculamos el area de cada pico usando np.trapz
peak_areas = []

for i in range(len(peaks) - 1):
    peak_start = peaks[i]  # Se calcula el inicio del pico
    peak_end = peaks[i + 1] if i + 1 < len(peaks) else len(df)  # Se calcula el final del pico

    # Se obtienen los valores para x y y
    x_peak = df[' 2THETA'][peak_start:peak_end].values
    y_peak = df[' PSD'][peak_start:peak_end].values

    # Calcula el area usando la función np.trapz
    area = np.trapz(y_peak, x=x_peak)
    peak_areas.append(area)

# Imprime el area calculada de cada pico
for idx, area in enumerate(peak_areas, start=1):
    display(Markdown(f'Area bajo el pico {idx}: {area}  $U^2$'))

display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:16px;" >
    <h3>
        Suma del area te todos los picos: {sum(peak_areas)} $U^2$
    </h3>
</div>


&nbsp;

'''))
```

Area bajo el pico 1: 9736.148850000009 U^2

Area bajo el pico 2: 11060.014950000117 U^2

Area bajo el pico 3: 5688.808999999912 U^2

Area bajo el pico 4: 4244.044500000017 U^2

Area bajo el pico 5: 3367.6622500000008 U^2

Area bajo el pico 6: 5740.697700000029 U^2

Area bajo el pico 7: 4104.956550000022 U^2

Area bajo el pico 8: 1971.0915000000175 U^2

Area bajo el pico 9: 1454.2565000000009 U^2

Suma del area te todos los picos: 47367.68180000013 U^2

Cálculo del % cristalinidad

Con estos valores podemos calcular la cristalinidad de nuestro material con la siguiente formula:

$$\% \text{ de cristalinidad} = \frac{\text{area de los picos cristalinos}}{\text{area total de la curva}} \cdot 100$$

Haciendo la sustituciones necesarias podemos llegar al siguiente resutlado:

In [15]:
```python
# Calculamos el porcentaje de cristalinidad
porcent_crystallinity = (sum(peak_areas)/area_total) * 100

display(Markdown(f'''
&nbsp;

<div align="center" style="background-color:yellow; padding-top:3px; padding-bottom:16px;"">
    <h3>
        % de cristalinidad obtenida: <b> {round(porcent_crystallinity, 3)} %</b>
    </h3>
</div>


&nbsp;

'''))
```

% de cristalinidad obtenida: **60.136 %**

Conclusión

El uso de Jupyter con el lenguaje de programación **python** representa un recurso invaluable para el análisis de datos científicos, especialmente en el campo de la cristalografía y la caracterización de materiales. Python, con su amplia gama de bibliotecas especializadas como NumPy, Pandas y herramientas de visualización como Plotly, se convierte en una plataforma potente y flexible para realizar estos análisis.

La estructura del cuaderno, combinando explicaciones detalladas con código ejecutable, facilita la comprensión y aplicación de técnicas complejas. Por ejemplo, al utilizar la función `np.trapz()` de NumPy para calcular el área bajo la curva en la determinación del porcentaje de cristalinidad, se demuestra cómo la combinación de herramientas específicas simplifica tareas que de otro modo podrían ser más complicadas.

Además, el proceso de leer archivos, convertir datos, identificar picos en gráficas y aplicar fórmulas para obtener información valiosa sobre los materiales estudiados muestra la versatilidad de Python en el ámbito científico. Esto se traduce en la capacidad de obtener resultados significativos y cálculos valiosos para comprender mejor las propiedades y comportamientos de los materiales.