

```
In [1]: 1 %pylab inline
        2 #
        3 # conda install -c conda-forge uncertainties
        4 #
```

Populating the interactive namespace from numpy and matplotlib

## 1 Ejercicio

En la Tabla 1 se reportan valores medidos ( $X$ ,  $Y$ ).

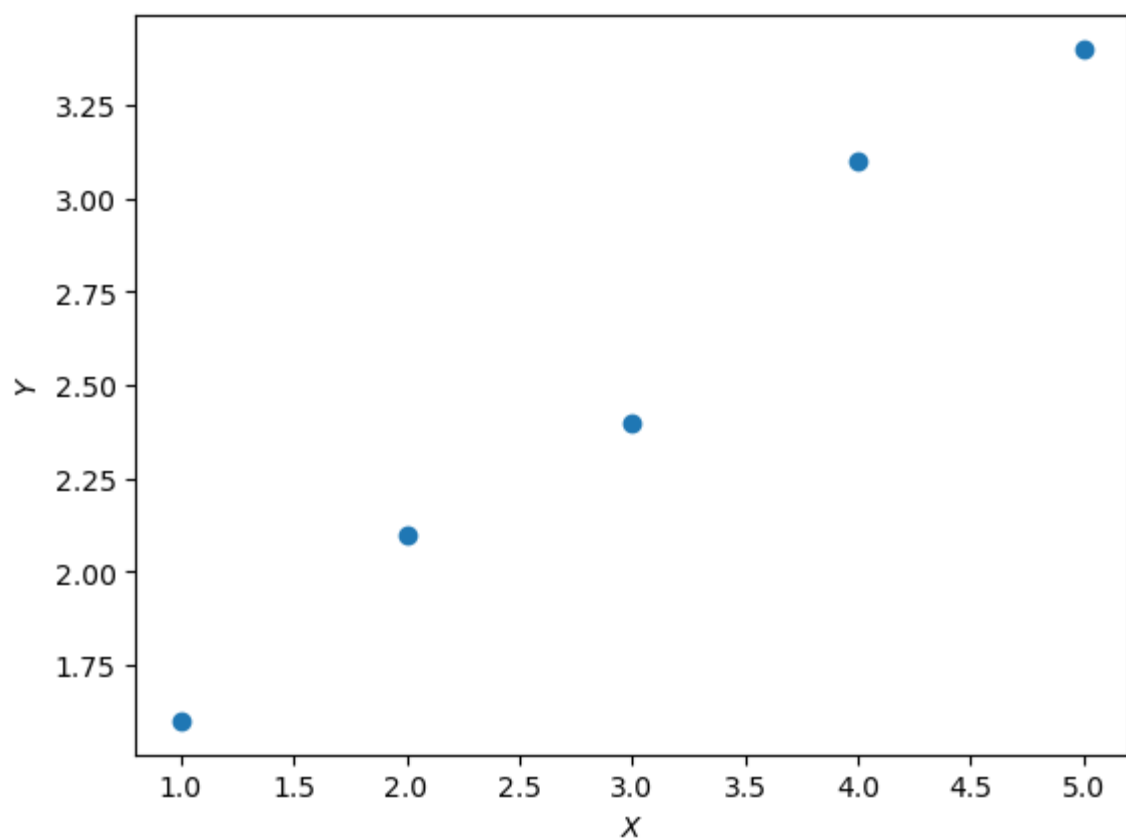
1. Graficar los valores, hacer el ajuste lineal y reportar la pendiente y su incerteza.
2. Calcular los valores  $Y_i/X_i$ , con  $i = 1$  a 5. Tomar el promedio de ellos y calcular su incerteza.
3. Calcular los valores  $(Y_{i+1} - Y_i)/(X_{i+1} - X_i)$ , con  $i = 1$  a 4. Tomar el promedio de ellos y calcular su incerteza.
4. Dibujar las pendientes que resultan en los casos 2 y 3 sobre la figura hecha en 1.

X	Y
1	1.6
2	2.1
3	2.4
4	3.1
5	3.4

```
In [2]: 1 from scipy.optimize import curve_fit
        2 import uncertainties as unc
        3 import uncertainties.unumpy as unp
        4
        5 x = array([1, 2, 3, 4, 5])
        6 y = array([1.6, 2.1, 2.4, 3.1, 3.4])
```

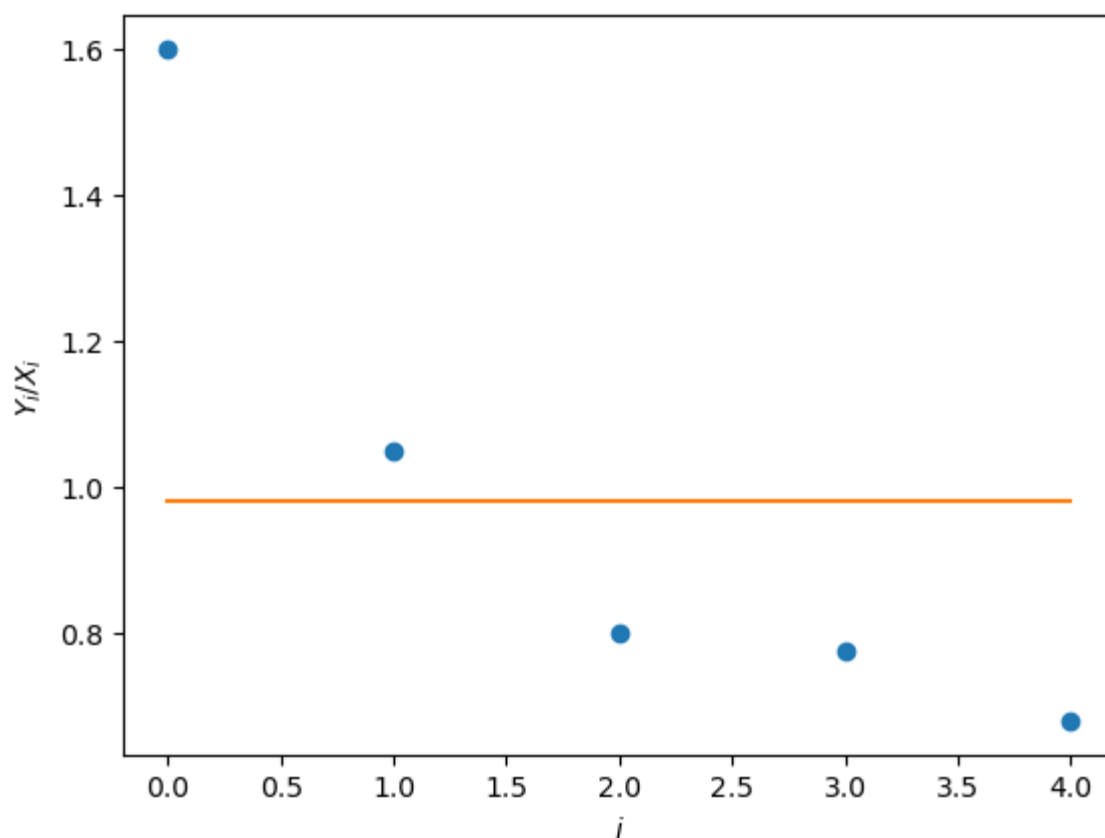
```
In [3]: 1 plot(x, y, "o")  
        2 ylabel("$Y$")  
        3 xlabel("$X$")
```

```
Out[3]: Text(0.5, 0, '$X$')
```



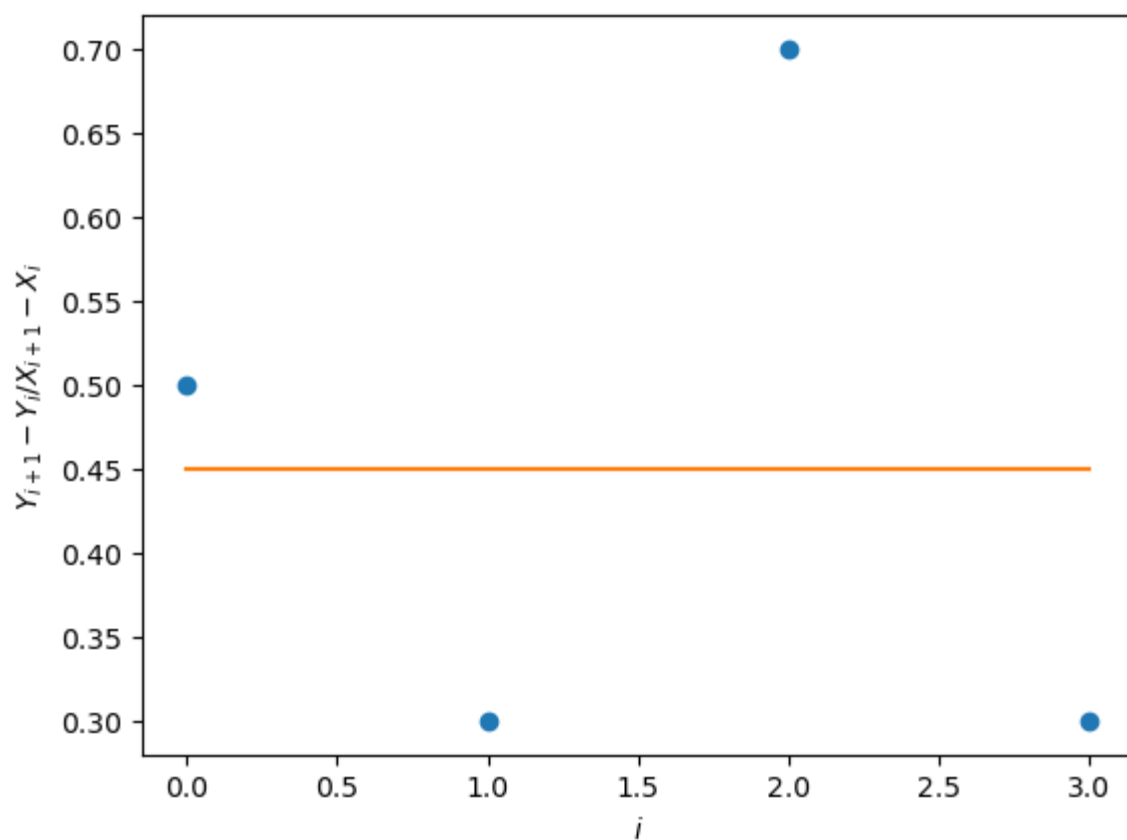
```
In [4]: 1 yx = y / x
2 plot(yx, "o")
3 plot(ones_like(yx) * yx.mean())
4 ylabel("$Y_i/X_i$")
5 xlabel("$i$")
6
7 print("Pendientes      =", yx)
8 print("Promedio       =", yx.mean())
9 print("Dispersion      =", yx.std())
10 print("Valor reportado =", unc.ufloat(yx.mean(), yx.std()))
```

```
Pendientes      = [1.6  1.05  0.8  0.775 0.68 ]
Promedio        = 0.9810000000000001
Dispersion      = 0.33278221106303146
Valor reportado = 0.98+/-0.33
```



```
In [5]: 1 dydx = (y[1:] - y[:-1]) / (x[1:] - x[:-1])
2
3 plot(dydx, "o")
4 plot(ones_like(dydx) * dydx.mean())
5 ylabel("$Y_{i+1}-Y_i/X_{i+1}-X_i$")
6 xlabel("$i$")
7
8 print("Pendientes      =", dydx)
9 print("Promedio       =", dydx.mean())
10 print("Dispersion     =", dydx.std())
11 print("Valor reportado =", unc.ufloat(dydx.mean(), dydx.std()))
```

```
Pendientes      = [0.5 0.3 0.7 0.3]
Promedio       = 0.44999999999999996
Dispersion     = 0.16583123951777015
Valor reportado = 0.45+/-0.17
```



```

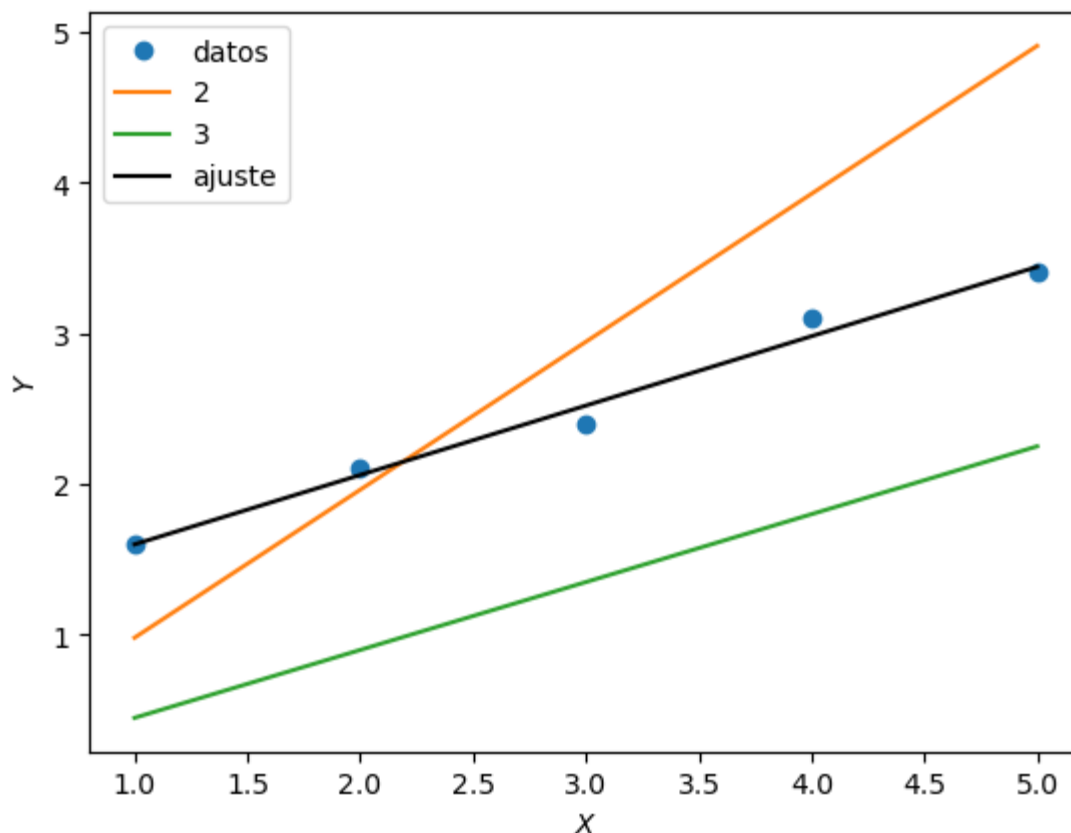
In [6]: 1 def lineal(x, a0, a1):
        2     return a1 * x + a0
        3
        4 popt, pcov = curve_fit(lineal, x, y)
        5
        6 aa0, aa1 = unc.correlated_values(popt, pcov)
        7
        8 plot(x, y, "o", label="datos")
        9
       10 plot(x, x * yx.mean(), "-", label="2")
       11 plot(x, x * dydx.mean(), "-", label="3")
       12
       13 plot(x, lineal(x, *popt), "-", color="black", label="ajuste")
       14
       15 ylabel("$Y$")
       16 xlabel("$X$")
       17 legend()
       18
       19 print("          aa0 =", aa0)
       20 print("          aa1 =", aa1)
       21 print("Valor reportado =", aa1)

```

aa0 = 1.14+/-0.11

aa1 = 0.460+/-0.033

Valor reportado = 0.460+/-0.033



## 2 Ejercicio

En la Tabla 2 se reportan los valores medidos para determinar una resistencia comercial. La corriente se midió al 5% y la tensión al 1%.

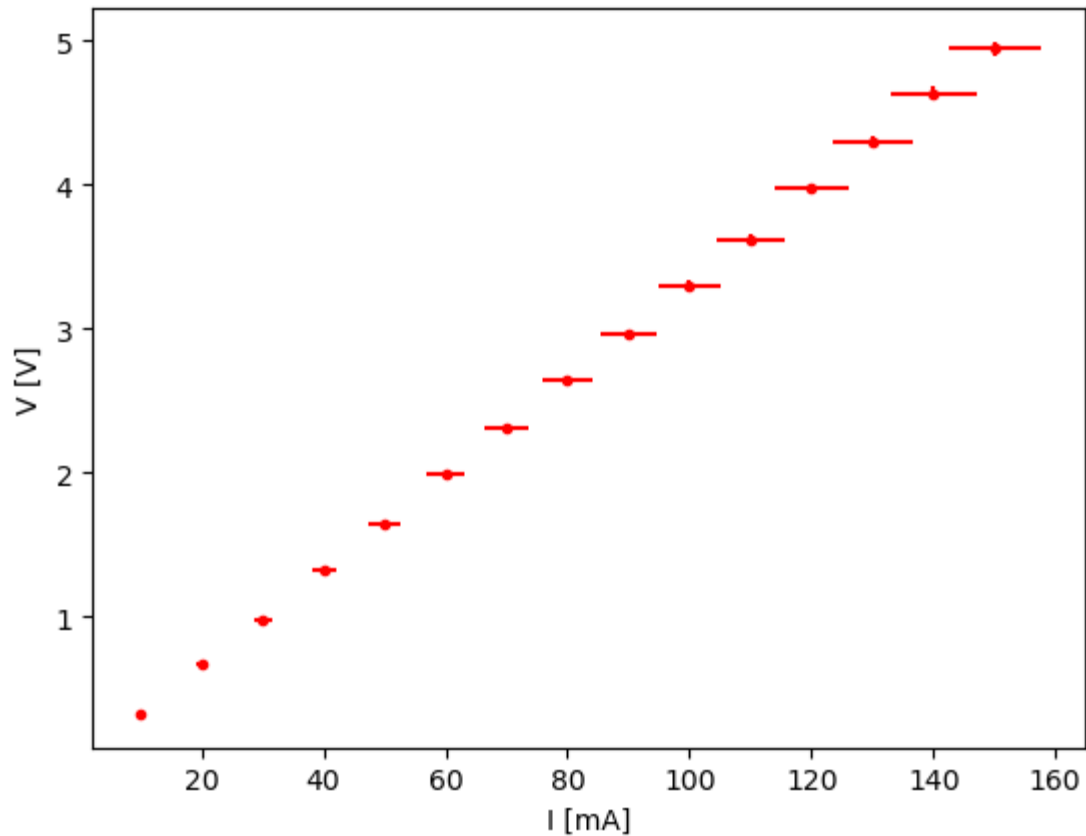
1. Graficar los valores y hacer el ajuste lineal.
2. Reportar el valor de la resistencia con su incerteza

Corriente [mA]	Tensión [V]
10	0.33
20	0.67
30	0.98
40	1.32
50	1.64
60	1.99
70	2.31
80	2.64
90	2.96
100	3.3
110	3.62
120	3.97
130	4.29
140	4.63
150	4.94

```
In [7]: 1 from scipy.optimize import curve_fit
2 import uncertainties as unc
3 import uncertainties.unumpy as unp
4
5 I = array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130,
6 V = array([
7     0.33, 0.67, 0.98, 1.32, 1.64, 1.99, 2.31, 2.64, 2.96, 3.3, 3.62
8     4.29, 4.63, 4.94
9 ])
```

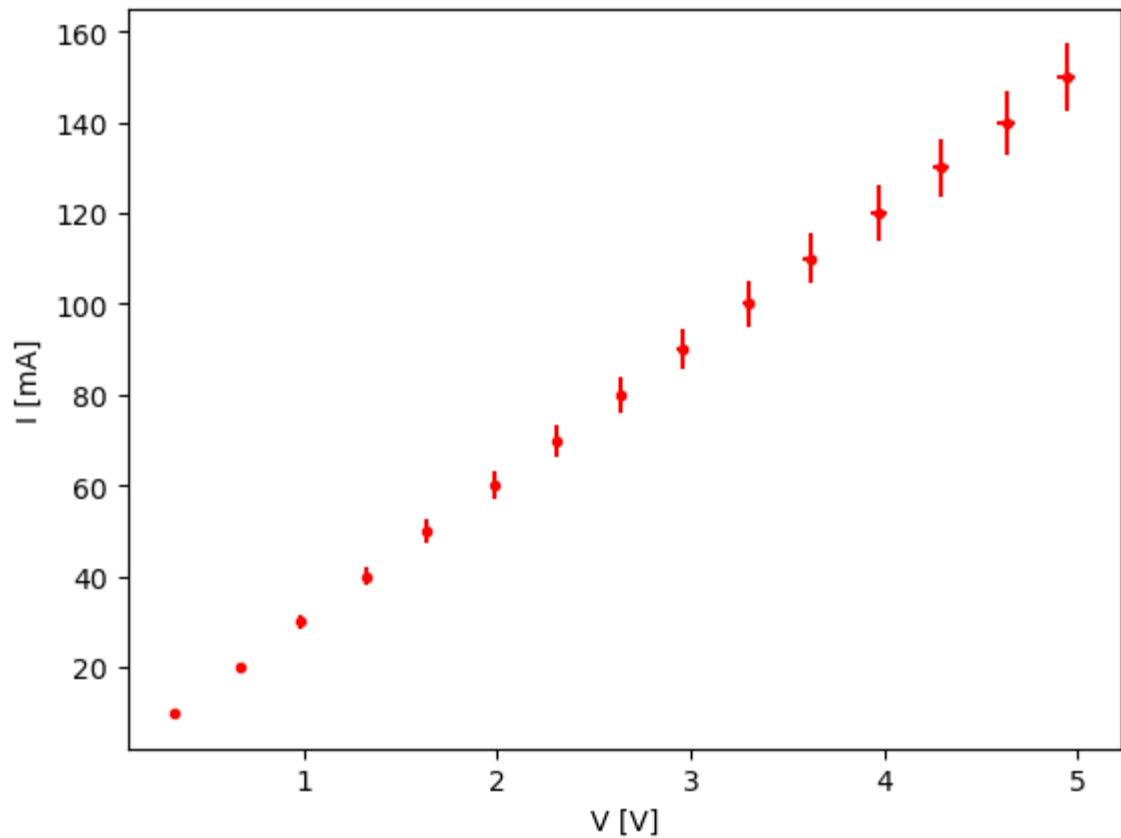
```
In [8]: 1 errorbar(I, V, yerr=V * 0.01, xerr=I * 0.05, linestyle='none', colo
2 plot(I, V, ".", color="red")
3 ylabel("V [V]")
4 xlabel("I [mA]")
5 # legend()
```

Out[8]: Text(0.5, 0, 'I [mA]')



```
In [9]: 1 errorbar(V, I, yerr=I * 0.05, xerr=V * 0.01, linestyle='none', colo
2 plot( V, I, ".", color="red")
3 ylabel("I [mA]")
4 xlabel("V [V]")
5
6 #legend()
```

Out[9]: Text(0.5, 0, 'V [V]')



## 2.1 Con cuadrados minimos con error en el eje Y

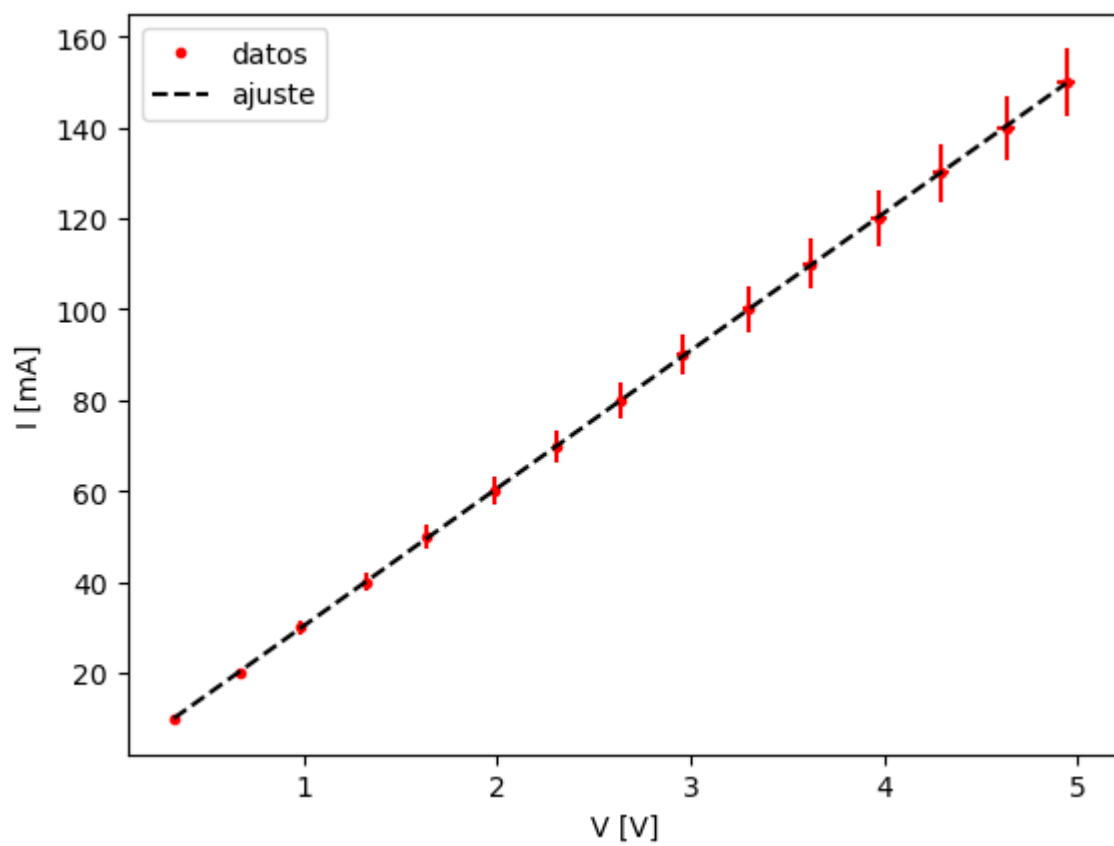


```
In [10]: 1 def func(x, R):
2         return x / R
3
4
5 semilla = [10.0]
6
7 popt, pcov = curve_fit(func, V, I, p0=semilla)
8
9 aa0 = unc.correlated_values(popt, pcov)
10 print("          aa0 =", aa0[0]*1000)
11
12 popt, pcov = curve_fit(func,
13                         V,
14                         I,
15                         p0=semilla,
16                         sigma=I * 0.05,
17                         absolute_sigma=True)
18
19 aa0 = unc.correlated_values(popt, pcov)
20 print("          aa0 =", aa0[0]*1000)
21
22 errorbar(V, I, yerr=I * 0.05, xerr=V * 0.01, linestyle='none', color='red')
23 plot(V, I, ".", color="red", label="datos")
24 plot(V, func(V, *popt), linestyle="dashed", color="black", label="a")
25 ylabel("I [mA]")
26 xlabel("V [V]")
27
28 legend()
```

aa0 = 32.993+/-0.023

aa0 = 33.0+/-0.4

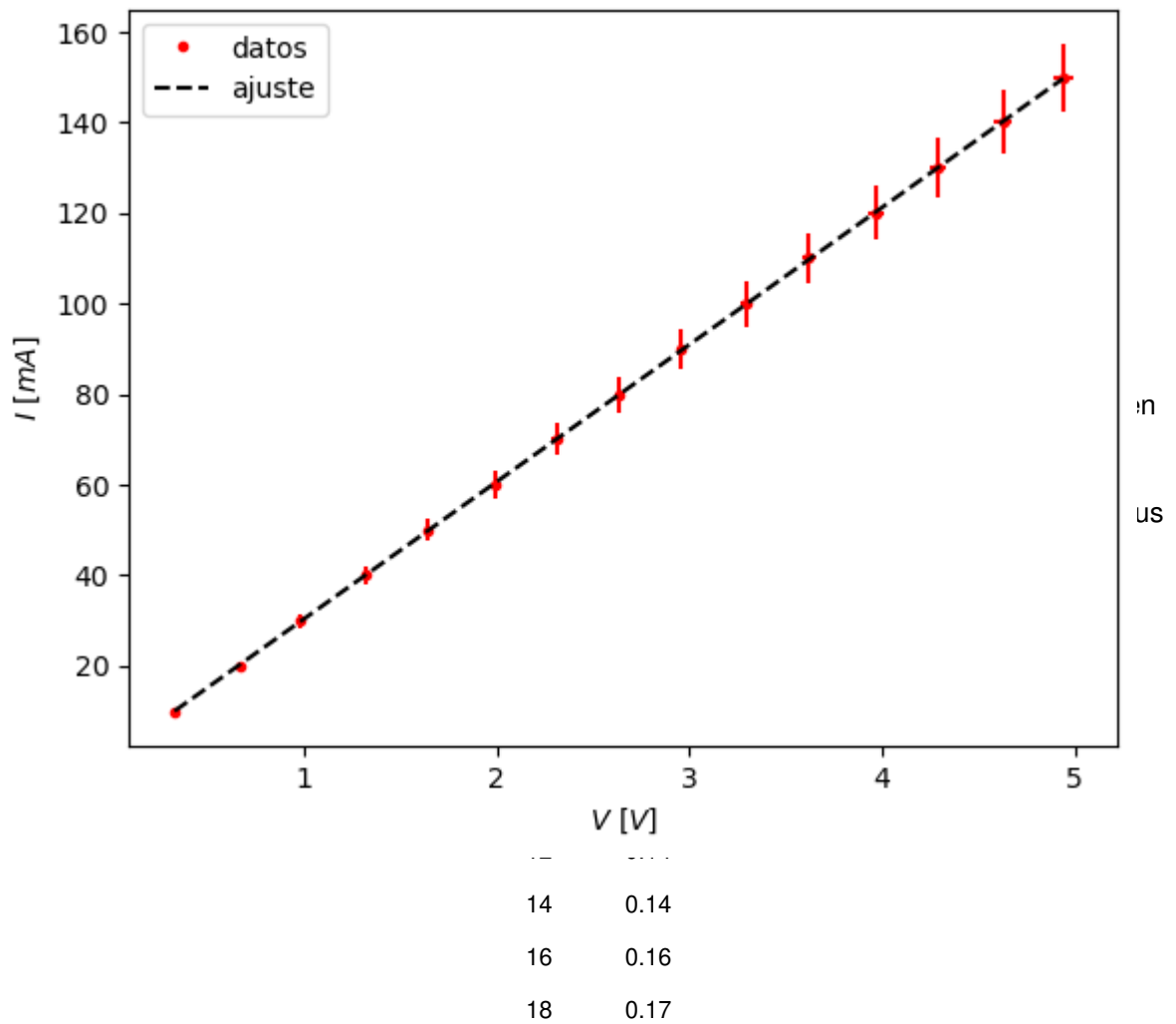
Out[10]: <matplotlib.legend.Legend at 0x7f0ae993ebb0>



In [11]:

```
1 from scipy.odr import *
2
3 def funcODR(R, x):
4     return x / R
5
6 # Create a model for fitting.
7 quad_model = Model(funcODR)
8
9 # Create a RealData object using our initiated data from above.
10 data = RealData(V, I, sx=V * 0.01, sy=I * 0.05)
11
12 # Set up ODR with the model and data.
13 odr = ODR(data, quad_model, beta0=semilla)
14
15 # Run the regression.
16 out = odr.run()
17
18 # Use the in-built pprint method to give us results.
19 # out.pprint()
20
21 errorbar(V, I, yerr=I * 0.05, xerr=V * 0.01, linestyle='none', color='red')
22 plot(V, I, ".", color="red", label="datos")
23 plot(V,
24       funcODR(out.beta, V),
25       linestyle="dashed",
26       color="black",
27       label="ajuste")
28
29 ylabel("$I\ [mA]$")
30 xlabel("$V\ [V]$")
31
32 legend()
33
34 print("aa0 =", unc.ufloat(out.beta, out.sd_beta)*1000)
```

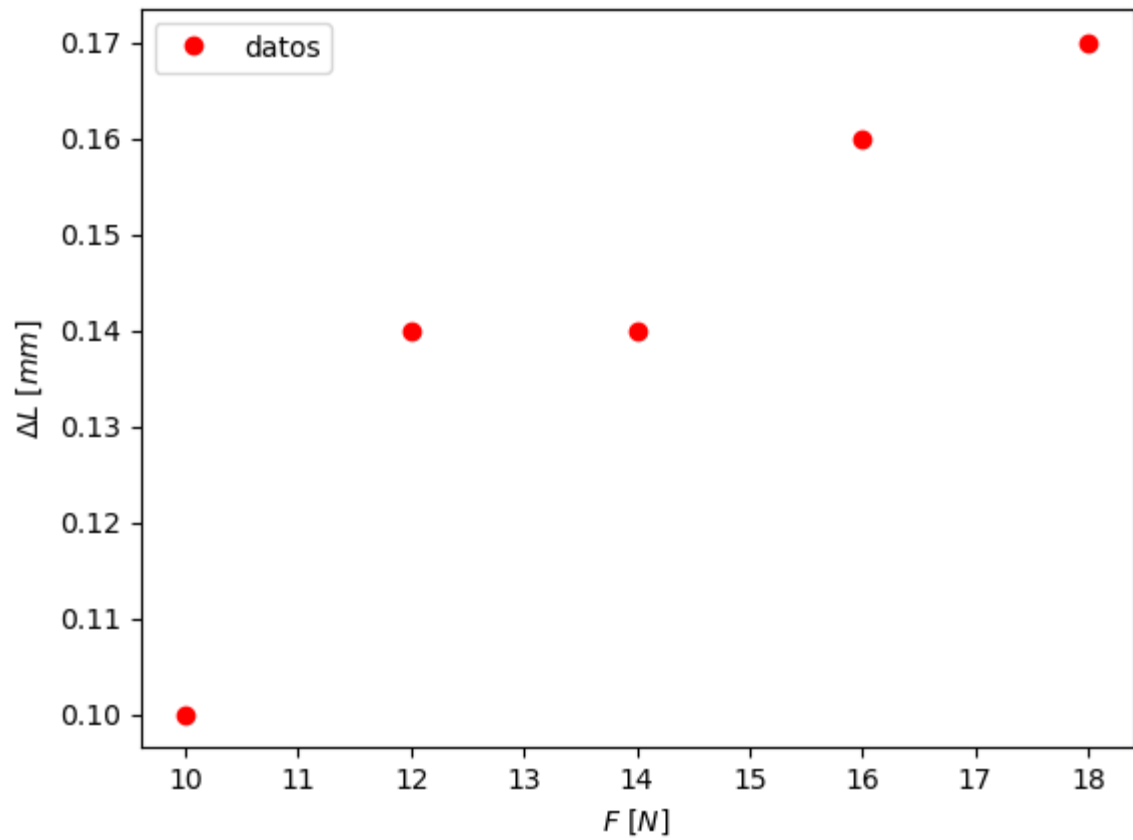
aa0 = 33.00+/-0.05



```
In [12]: 1 from scipy.optimize import curve_fit
          2 import uncertainties as unc
          3 import uncertainties.unumpy as unp
          4
          5 F = array([10, 12, 14, 16, 18])
          6 DL = array([0.10, 0.14, 0.14, 0.16, 0.17])
```

```
In [13]: 1 plot(F, DL, "o", color="red", label="datos")
          2
          3 xlabel("$F$ [N]")
          4 ylabel("$\Delta L$ [mm]")
          5
          6 legend()
```

Out[13]: <matplotlib.legend.Legend at 0x7f0ae91905e0>



In [14]:

```

1 # https://stackoverflow.com/questions/24633664/confidence-interval-
2
3
4 def lineal(x, a0, a1):
5     return a1 * x + a0
6
7
8 popt, pcov = curve_fit(lineal, F, DL)
9
10 plot(F, DL, "o", color="red", label="datos")
11
12 px = linspace(9, 21, 100)
13
14 plot(px, lineal(px, *popt), linestyle="dashed", color="black", label="lineal")
15
16 aa0, aa1 = unc.correlated_values(popt, pcov)
17
18 py = aa1 * px + aa0
19
20 nom = unp.nominal_values(py)
21 std = unp.std_devs(py)
22
23 plt.plot(px,
24          nom - 1 * std,
25          c='c',
26          linestyle="dotted",
27          label="$\pm 1 \sigma$")
28 plt.plot(px, nom + 1 * std, c='c', linestyle="dotted")
29
30 xlabel("$F \ [N]$")
31 ylabel("$\Delta L \ [mm]$")
32
33 vlines(13, 0.1, 0.21)
34 vlines(20, 0.1, 0.21)
35
36 legend()
37
38 print("aa0 =", aa0)
39 print("aa1 =", aa1)
40 print("DeltaL con 13 N =", aa1 * 13 + aa0)
41 print("DeltaL con 20 N =", aa1 * 20 + aa0)

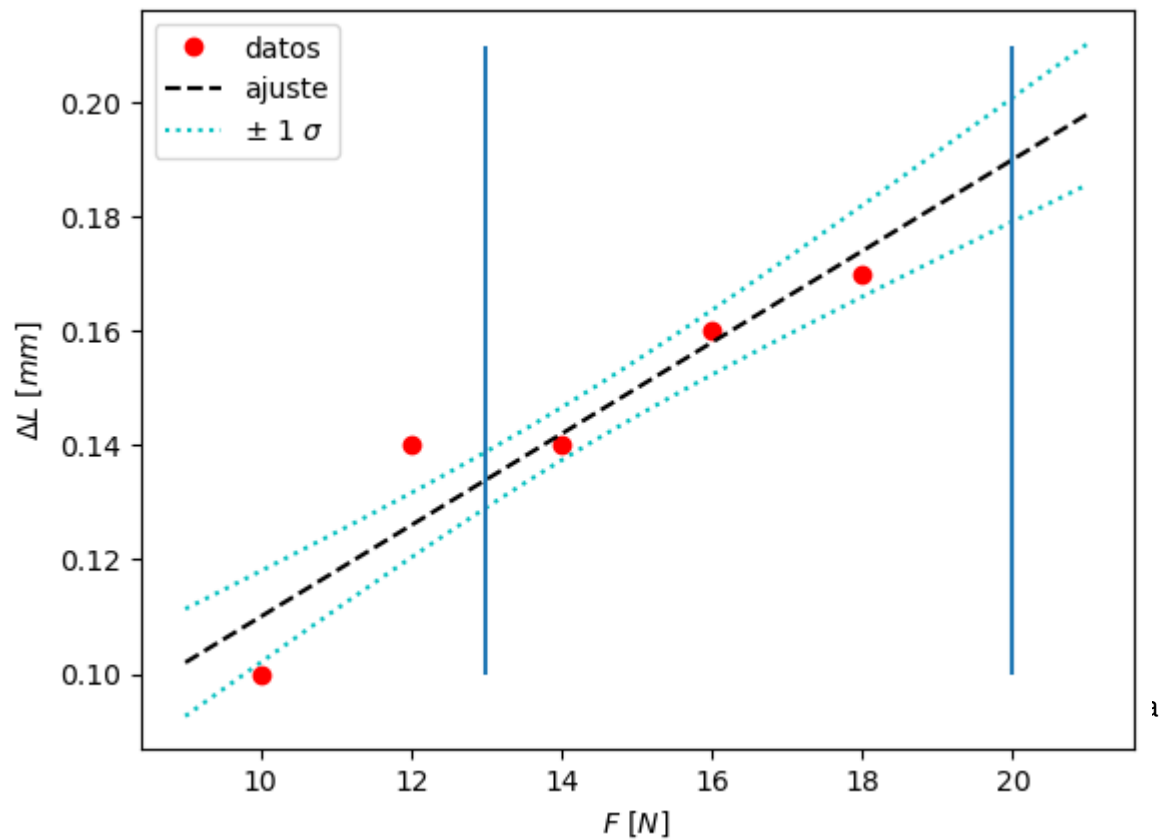
```

aa0 = 0.030+/-0.023

aa1 = 0.0080+/-0.0016

DeltaL con 13 N = 0.134+/-0.005

DeltaL con 20 N = 0.190+/-0.011



El valor de ajuste cuadrático entre 10.1 mm y 10.6 mm.

3. Reportar el valor de  $X$  y el valor de  $\alpha$ , con sus incertezas, que corresponden al mínimo de la curva.

$X$ [mm]	$\alpha$ [°]
12.53	149.28
12.70	149.05
12.88	148.84
13.05	148.64
13.22	148.46
13.40	148.31
13.57	148.18
13.75	148.07
13.92	147.99
14.09	147.94
14.27	147.93
14.44	147.95
14.62	148.02
14.79	148.13
14.96	148.30
15.14	148.52

$X$ [mm]	$\alpha$ [°]
15.31	148.82
15.49	149.19
15.66	149.66
15.83	150.23

## 4.1 Recocrto los datos y hago un grafiquito

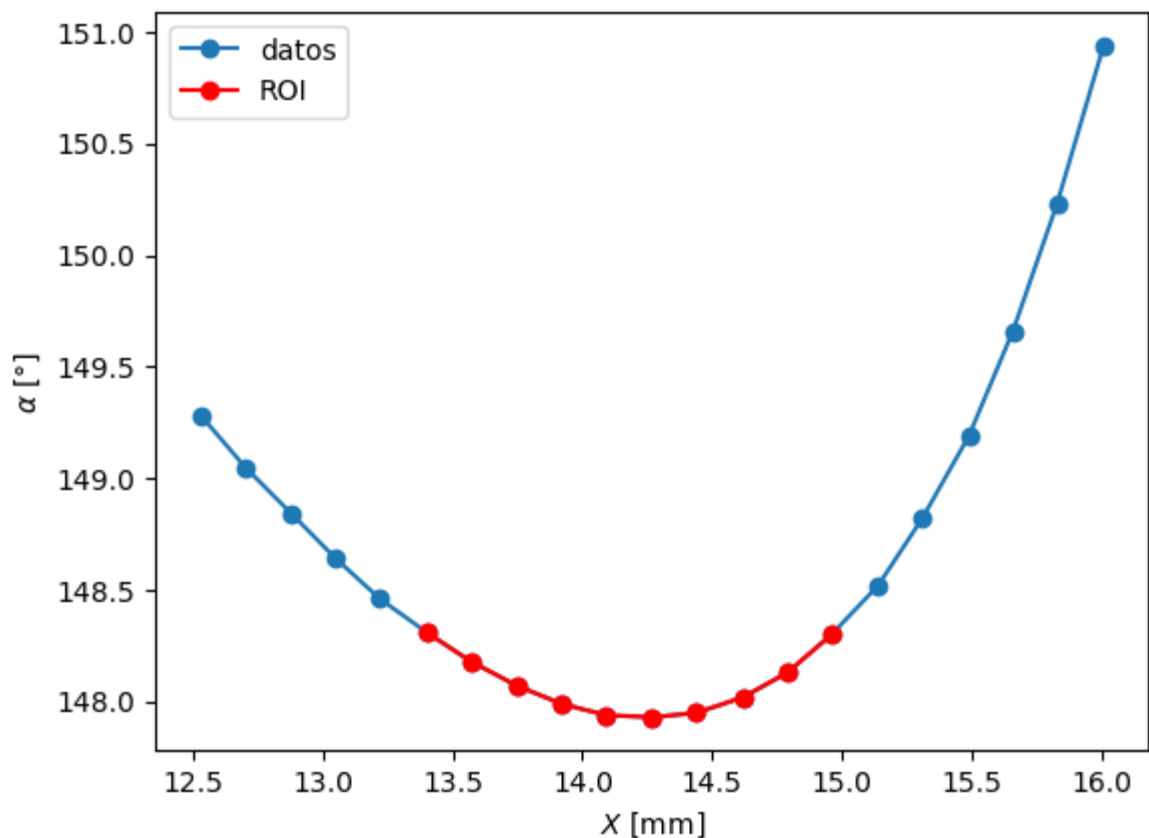
In [15]:

```
1 from scipy.optimize import curve_fit
2 import uncertainties as unc
3 import uncertainties.unumpy as unp
4
5 x = array([
6     12.53, 12.7, 12.88, 13.05, 13.22, 13.4, 13.57, 13.75, 13.92, 14
7     14.44, 14.62, 14.79, 14.96, 15.14, 15.31, 15.49, 15.66, 15.83,
8 ])
9
10 alpha = array([
11     149.28, 149.05, 148.84, 148.64, 148.46, 148.31, 148.18, 148.07,
12     147.94, 147.93, 147.95, 148.02, 148.13, 148.3, 148.52, 148.82,
13     149.66, 150.23, 150.94
14 ])
```



```
In [35]: 1 plot(x, alpha, "o-", label="datos")
2
3 filtro = (x >= 13.4) & (x < 15.0)
4
5 xr, alphas = x[filtro], alpha[filtro]
6
7 plot(xr, alphas, "o-", c='red', label="ROI")
8
9 xlabel(r'$X$ [mm]')
10 ylabel(r'$\alpha$ [°]')
11
12 legend()
```

Out[35]: <matplotlib.legend.Legend at 0x7f0ae88e76d0>



## 4.2 Usando un polinomio en su forma canonica (AKA: Fuerza Bruta)

```

In [17]: 1 def poli(x, a0, a1, a2):
          2     return a0 + a1 * x + a2 * x**2
          3
          4
          5 xe = linspace(13.3, 15.1, 100)
          6
          7 popt, pcov = curve_fit(poli, xr, alphas)
          8
          9 plot(x, alpha, "o", label="datos")
         10 plot(xr, alphas, "o", c='red', label="ROI")
         11
         12 plot(xe, poli(xe, *popt), linestyle="dashed", color="black", label=
         13
         14 fbu_aa0 = unc.ufloat(popt[0], sqrt(diag(pcov))[0])
         15 fbu_aa1 = unc.ufloat(popt[1], sqrt(diag(pcov))[1])
         16 fbu_aa2 = unc.ufloat(popt[2], sqrt(diag(pcov))[2])
         17
         18 fb_aa0, fb_aa1, fb_aa2 = unc.correlated_values(popt, pcov)
         19
         20 pcov_poli = pcov
         21 print("    fb_aa0 =", fb_aa0)
         22 print("    fb_aa1 =", fb_aa1)
         23 print("    fb_aa2 =", fb_aa2)
         24
         25 alphas = fb_aa0 + fb_aa1 * xe + fb_aa2 * xe**2
         26
         27 nom = unp.nominal_values(alphas)
         28 std = unp.std_devs(alphas)
         29
         30 plot(xe, nom - 1 * std, c='green', label="$\pm 1\ \sigma$", linewidth=
         31 plot(xe, nom + 1 * std, c='green', linewidth=0.5)
         32
         33 xlabel("$X$ [mm]")
         34 ylabel("$\alpha$ [°]")
         35
         36 xlim(13.3, 15.1)
         37 ylim(147.9, 148.4)
         38
         39 legend()
         40
         41 print()
         42 print(popt)
         43 print(sqrt(diag(pcov)))
         44 print(pcov)

```

```

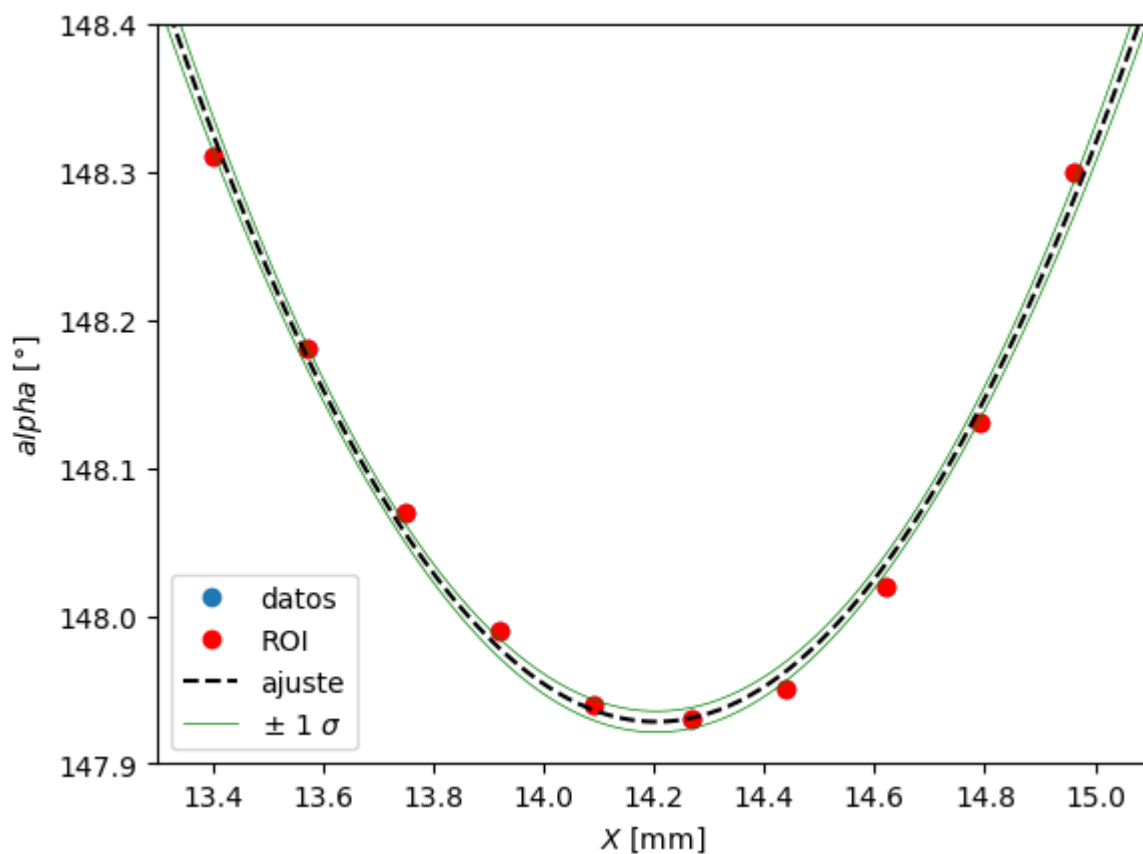
fb_aa0 = 272+/-4
fb_aa1 = -17.5+/-0.6
fb_aa2 = 0.615+/-0.021

```

```

[272.10716767 -17.48501101  0.61549474]
[4.29514327 0.60630916 0.02137551]
[[ 1.84482557e+01 -2.60387151e+00  9.17669815e-02]
 [-2.60387151e+00  3.67610801e-01 -1.29586252e-02]
 [ 9.17669815e-02 -1.29586252e-02  4.56912596e-04]]

```



### 4.3 Usando una función inteligente

In [18]:

```
1 def cuadratica(x, a0, a1, a2):
2     return a0 + a2 * (x - a1)**2
3
4
5 xe = linspace(13.3, 15.1, 100)
6
7 semilla = [140, 14, 1]
8
9 popt, pcov = curve_fit(cuadratica, xr, alphas, p0=semilla)
10
11 plot(x, alphas, "o", label="datos")
12 plot(xr, alphas, "o", c='red', label="ROI")
13
14 plot(xe,
15      cuadratica(xe, *popt),
16      linestyle="dashed",
17      color="black",
18      label="ajuste")
19
20 fi_aa0, fi_aa1, fi_aa2 = unc.correlated_values(popt, pcov)
21
22 pcov_lineal = pcov
23 print("    fi_aa0 =", fi_aa0)
24 print("    fi_aa1 =", fi_aa1)
25 print("    fi_aa2 =", fi_aa2)
26
27 alphas = fi_aa0 + fi_aa2 * (xe - fi_aa1)**2
28
29 nom = unp.nominal_values(alphas)
30 std = unp.std_devs(alphas)
31
32 plot(xe, nom - 1 * std, c='green', label="$\pm 1\ \sigma$", linewidth=0.5)
33 plot(xe, nom + 1 * std, c='green', linewidth=0.5)
34
35 xlabel("$X$ [mm]")
36 ylabel("$\alpha$ [°]")
37
38 xlim(13.3, 15.1)
39 ylim(147.9, 148.4)
40
41 legend()
42
43 print()
44 print(popt)
45 print(sqrt(diag(pcov)))
46 print(pcov)
```

```

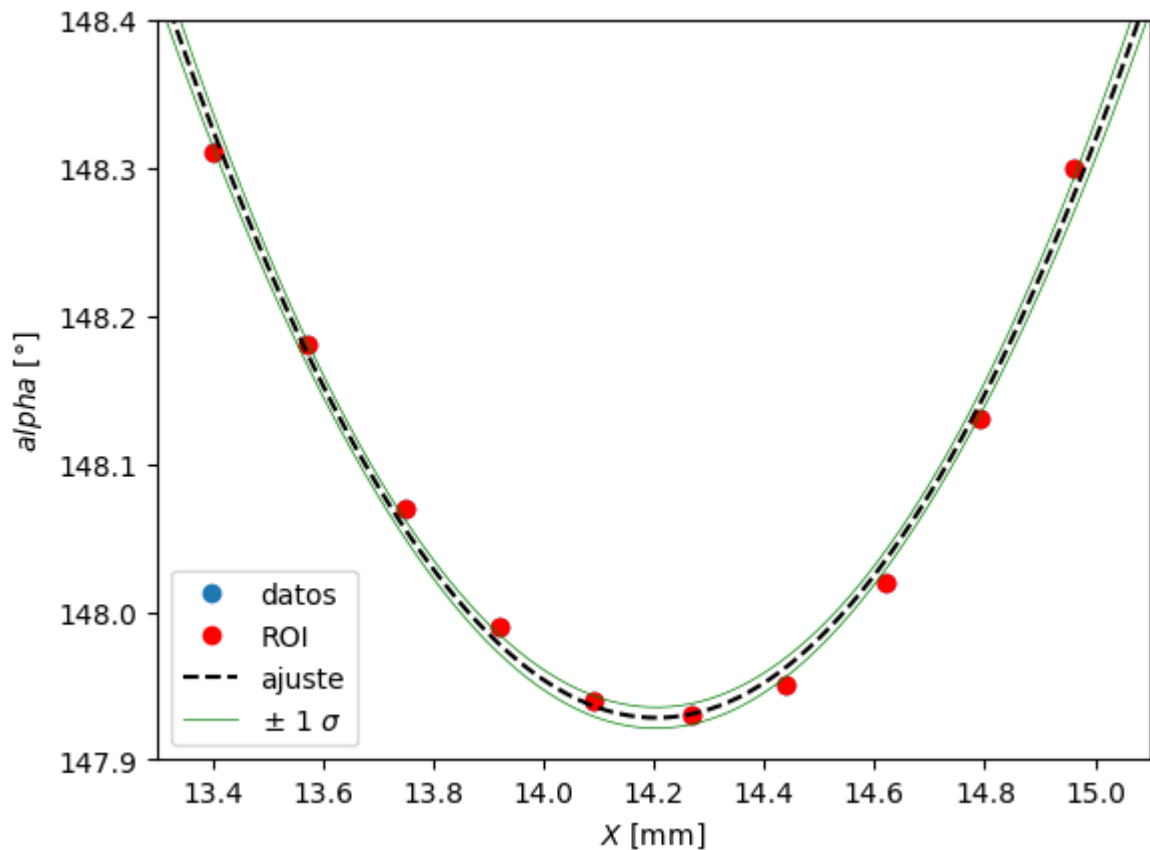
fi_aa0 = 147.928+/-0.007
fi_aa1 = 14.204+/-0.008
fi_aa2 = 0.615+/-0.021

```

```

[147.9283606  14.20402961  0.6154947 ]
[0.00707801 0.00764761 0.0213756 ]
[[ 5.00982182e-05  2.67622575e-06 -1.13562797e-04]
 [ 2.67622575e-06  5.84859458e-05 -1.73635020e-05]
 [-1.13562797e-04 -1.73635020e-05  4.56916195e-04]]

```



**4.4 Calculemos lo que realmente queremos: donde esta el minimo y sus errores!**

```

In [19]: 1 print()
          2 print()
          3 print("Funcion inteligente:")
          4 print("-----")
          5 print("alpha min =", fi_aa0)
          6 print("      x min =", fi_aa1)
          7 print("      p =", fi_aa2)
          8 print()
          9 print()
         10 print("Fuerza Bruta (sin correlaciones):")
         11 print("-----")
         12 print("alpha min =", fbu_aa0 - fbu_aa1**2 / (4 * fbu_aa2))
         13 print("      x min =", -fbu_aa1 / (2 * fbu_aa2))
         14 print("      p =", fbu_aa2)
         15 print()
         16 print()
         17 print("Fuerza Bruta (con correlaciones):")
         18 print("-----")
         19 print("alpha min =", fb_aa0 - fb_aa1**2 / (4 * fb_aa2))
         20 print("      x min =", -fb_aa1 / (2 * fb_aa2))
         21 print("      p =", fb_aa2)
         22 print()
         23 print()
         24 #
         25 # https://en.wikipedia.org/wiki/Propagation\_of\_uncertainty
         26 #

```

Funcion inteligente:

```

-----
alpha min = 147.928+/-0.007
      x min = 14.204+/-0.008
      p = 0.615+/-0.021

```

Fuerza Bruta (sin correlaciones):

```

-----
alpha min = 148+/-11
      x min = 14.2+/-0.7
      p = 0.615+/-0.021

```

Fuerza Bruta (con correlaciones):

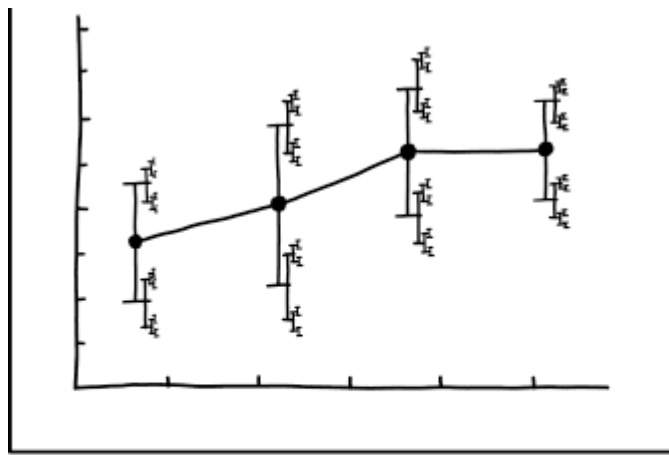
```

-----
alpha min = 147.928+/-0.007
      x min = 14.204+/-0.008
      p = 0.615+/-0.021

```

Hermoso no?





I DON'T KNOW HOW TO PROPAGATE  
ERROR CORRECTLY, SO I JUST PUT  
ERROR BARS ON ALL MY ERROR BARS.

In [ ]:

1