Introducción a python para computación numérica II

- El algoritmo de Horner
- Ejercicios:
  - Ejercicio 1
  - Ejercicio 2
  - Ejercicio 3
- Ejercicios propuestos

El algoritmo de Horner

Importamos los módulos `matplotlib.pyplot` , `numpy` y las funciones de `numpy` para polinomios.

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.polynomial.polynomial as pol
```

Sea un polinomio

$$P(x) = p_0 + p_1\,x + p_2\,x^2 + p_3\,x^3 + p_4\,x^4 + p_5\,x^5$$

Usaremos como coeficientes de este polinomio

```
p = np.array([1., -1, 2, -3,  5, -2])
p0, p1, p2, p3, p4, p5 = p
```

Por lo tanto el polinomio es

$$P(x) = 1 - x + 2\,x^2 - 3\,x^3 + 5\,x^4 - 2\,x^5$$

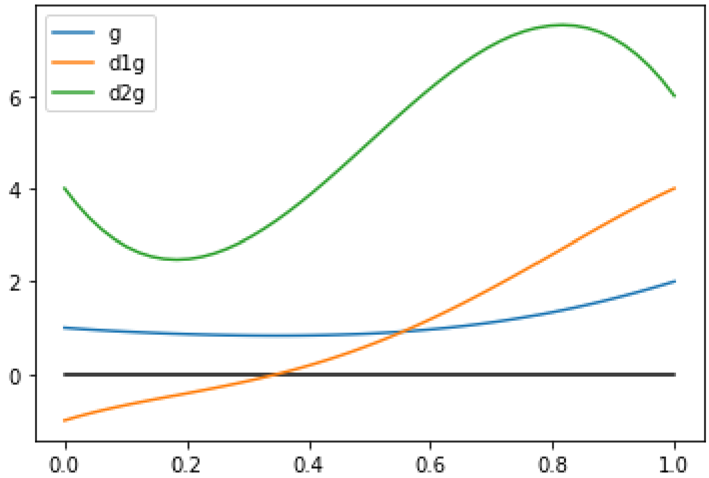
Podemos definir este polinomio y sus derivadas como funciones lambda

```
g = lambda x: p0 + p1*x + p2*x**2 + p3*x**3 + p4*x**4 + p5*x**5
d1g = lambda x: p1 + 2*p2*x + 3*p3*x**2 + 4*p4*x**3 + 5*p5*x**4
d2g = lambda x: 2*p2 + 6*p3*x + 12*p4*x**2 + 20*p5*x**3
```

Y entonces podemos dibujarlos

```
a = 0.; b = 1.
x = np.linspace(a,b)

plt.figure()
plt.plot(x,0*x,'k')
plt.plot(x,g(x), label = 'g')
plt.plot(x,d1g(x), label = 'd1g')
plt.plot(x,d2g(x), label = 'd2g')
plt.legend()
plt.show()
```



Hay una forma mejor de trabajar con polinomios de forma que una vez hemos almacenado sus coeficientes en un array numpy no necesitamos usar funciones lambda.

Usando el comando numpy `polyval` podemos calcular el valor del polinomio en un punto o en varios puntos simultaneamente

```
x0 = -0.5
print('Valor de P en el punto  ', x0)
print('Con polyval:            ', pol.polyval(x0, p))
print('Con la función lambda g:', g(x0))
```

```
Valor de P en el punto   -0.5
Con polyval:             2.75
Con la función lambda g: 2.75
```

Para practicar operar con arrays numpy unidimensionales vamos a implementar el algoritmo que utiliza `polyval` , el **algorimo de Horner**. Primero, pensemos cuantas sumas y productos necesitaríamos para calcular el polinomio en un punto x_0

$$P(x_0) = p_0 + p_1\,x_0 + p_2\,x_0^2 + p_3\,x_0^3 + p_4\,x_0^4 + p_5\,x_0^5$$

Para el segundo sumando necesitamos 1 multiplicación, 2 para el tercero, 3 para el cuarto, 4 para el quinto y 5 para el sexto. Además hacen falta 5 sumas. Es decir

- $1+2+3+4+5 = 15$ multiplicaciones.
- 5 sumas.

Veamos como calcularíamos lo mismo con el **algoritmo de Horner**.

Si aplicamos Ruffini al polinomio con $x_0 = 1$

$$\begin{array}{c|rrrrrrl} & -2 & 5 & -3 & 2 & -1 & 1 & \\ 1 & & -2 & 3 & 0 & 2 & 1 & \\ \hline & -2 & 3 & 0 & 2 & 1 & \boxed{2} & = P(1) \end{array}$$

Si

$$P(x) = 1 - x + 2x^2 - 3x^3 + 5x^4 - 2x^5$$

y

$$Q(x) = 1 + 2x + 3x^3 - 2x^4,$$

tenemos que

$$P(x) = Q(x)(x-1) + 2$$

por lo que

$$P(1) = Q(1)(1-1) + 2,$$

es decir,

$$P(1) = 2.$$

Y las operaciones realizadas son

$$\begin{array}{c|rrrrrrl} & -2 & 5 & -3 & 2 & -1 & 1 & \\ 1 & & -2 & 3 & 0 & 2 & 1 & \\ \hline & -2 & 3 & 0 & 2 & 1 & \boxed{2} & = P(1) \end{array}$$

Es decir

$$\begin{array}{c|cccccccl} & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 & \\ x_0 & & q_5\,x_0 & q_4\,x_0 & q_3\,x_0 & q_2\,x_0 & q_1\,x_0 & \\ \hline & q_5 & q_4 & q_3 & q_2 & q_1 & \boxed{q_0} & = P(x_0) \end{array}$$

$$
\begin{aligned}
q_5 &= p_5\\
q_4 &= p_4 + q_5\, x_0\\
q_3 &= p_3 + q_4\, x_0\\
q_2 &= p_2 + q_3\, x_0\\
q_1 &= p_1 + q_2\, x_0\\
q_0 &= p_0 + q_1\, x_0
\end{aligned}
$$

Es decir, 5 multiplicaciones y 5 sumas. Y en general, para un polinomio de grado n, se realizarán n multiplicaciones y n sumas.

Ejercicios

Ejercicio 1

Escribir una función que implemente el **algoritmo de Horner**. Las variables de entrada serán un array numpy unidimensional (vector) que contendrá los coeficientes de un polinomio P, y el punto x_0, es decir, **`horner(x0,p)`** y las variables de salida serán un array numpy unidimensional, que contendrá los coeficientes del polinomio Q y que llamaremos **`cociente`** , y una variable real que será $P\left(x_0\right)$ y que llamaremos **`resto`** . Imprimir los coeficientes del polinomio Q y $P\left(x_0\right)$. Comprobar que el valor de $P\left(x_0\right)$ es correcto usando el comando **`pol.polyval(x0,p)`**

Probar con el polinomio P_0 y el punto x_0

```
p0 = np.array([1.,2,1])
x0 = 1.
```

Dar los resultados para el polinomio P_1 y el punto x_1

```
p1 = np.array([1., -1, 2, -3,  5, -2])
x1 = 1.
```

Dar también los resultados para el polinomio P_2 y el punto x_2

```
p2 = np.array([1., -1, -1, 1, -1, 0, -1, 1])
x2 = -1.
```

Notas:

- El número de elementos de un array unidimensional `p` viene dado por `n = len(p)`.
- Podemos inicializar un vector `q` del mismo tamaño que `p` con `q = np.zeros_like(p)` o con `q = np.zeros(n)`.
- `range(n1,n2,-1)` crea una sucesión de enteros consecutivos de `n1` a `n2` (pero sin llegar a `n2`) con paso `-1`.
- El último elemento de un vector `p` es `p[n-1]` o también `p[-1]`.
- Si queremos extraer todos los elementos de `p` menos el primero podemos hacerlo con `p[1:]`.
- Si hay varios parámetros de salida, la última línea de la función puede ser

```
return cociente, resto
```

- Y luego podemos llamar a la función con

```
c, r = horner(x0, p)
```

Para facilitar importar la función `horner` en el **ejercicio 3**, la estructura del programa en el fichero será

```
import numpy as np
import numpy.polynomial.polynomial as pol

def horner(x0,p):
    ...  # <--- escribe aquí tu código
    return cociente, resto

def main():
    p0 = np.array([1.,2,1])
    x0 = 1
    c, r = horner(x0,p0)
    rp   = pol.polyval(x0,p0)

    print('Coeficientes de Q = ', c)
    print('P0(1)        = ', r)
    print('Con polyval = ', rp)

    ... # <--- ejemplos con polinomios p1 y p2

if __name__ == "__main__":
    main()
```

De esta forma, si queremos importar la función `horner` en futuros ejercicios, el código en la función `main` no se ejecutará al importarlo, solo al ejecutar el fichero `Ejercicio1.py`.

```
%run Ejercicio1.py
```

```
Coeficientes de Q =  [3. 1.]
P0(1)        =  4.0
Con polyval =  4.0


Coeficientes de Q =  [ 1.  2.  0.  3. -2.]
P1(1)        =  2.0
Con polyval =  2.0


Coeficientes de Q =  [ 4. -5.  4. -3.  2. -2.  1.]
P2(-1)       =  -3.0
Con polyval =  -3.0
```

Ejercicio 2

Modificar la función del **Ejercicio 1** de forma que admita, en lugar del punto `x0` , como variable de entrada, un vector del tipo `x = np.linspace(-1,1)` .

La función será `HornerV(x,p)` y, ahora, la variable de salida serán los valores `y` del polinomio P en los puntos contenidos en el vector `x` . Construir este programa añadiendo un bucle externo que repita el proceso para cada punto contenido en `x` y que guarde los resultados en un vector `y` .

Utilizando esta función, dibujar el polinomio P

```
p = np.array([1., -1, 2, -3, 5, -2])
```

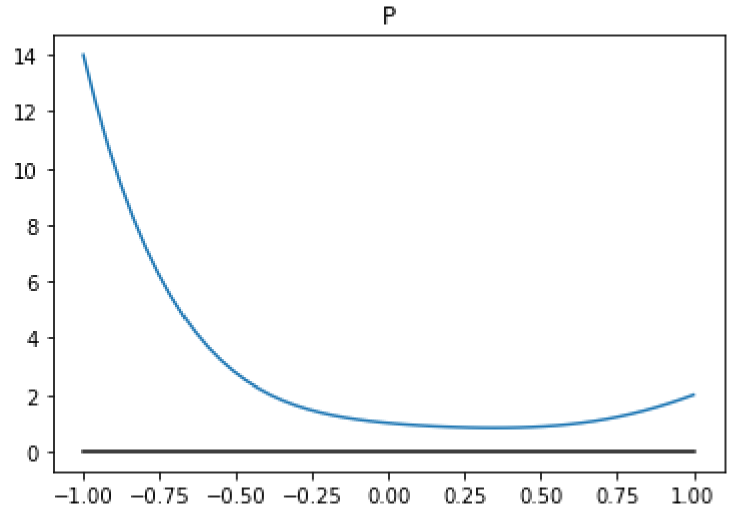
Dibujar también el polinomio R

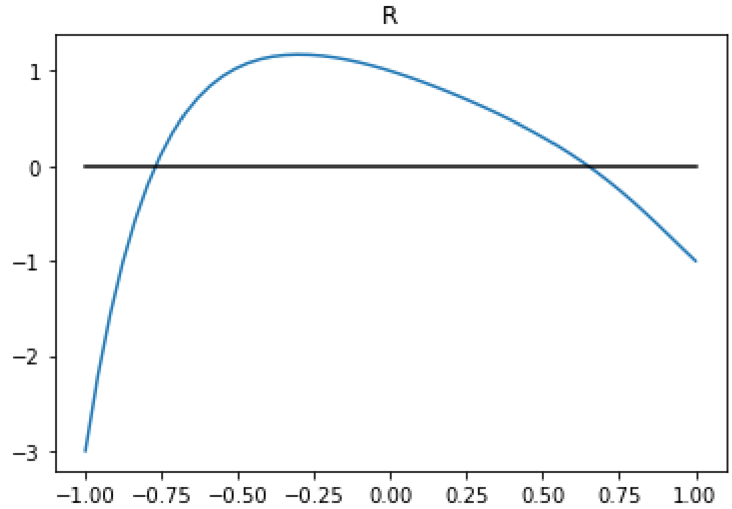
```
r = np.array([1., -1, -1, 1, -1, 0, -1, 1])
```

Nota:

- Se puede inicializar el vector `y` con un vector del mismo tamaño que `x` con `y = np.zeros_like(x)` .

```
%run Ejercicio2.py
```

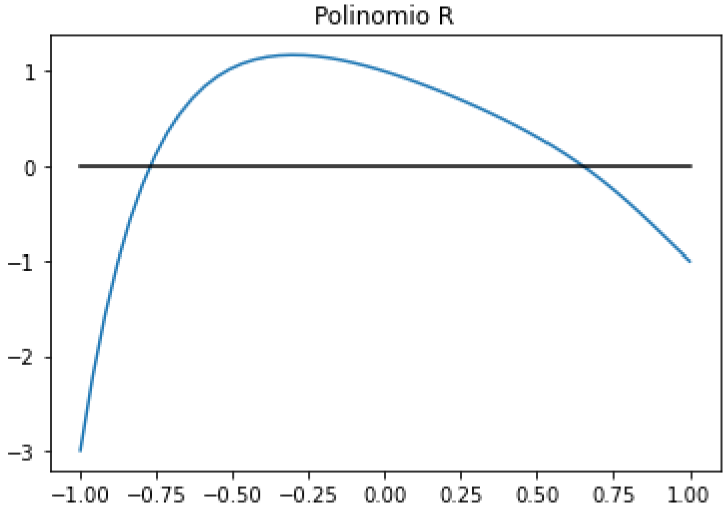




El comando `polyval` también puede obtener los valores del polinomio para los elementos de un vector.

```
r = np.array([1., -1, -1, 1, -1, 0, -1, 1])
x = np.linspace(-1,1)

plt.figure()
plt.plot(x,pol.polyval(x,r))
plt.plot(x,0*x,'k')
plt.title('Polinomio R')
plt.show()
```



Fórmula de Taylor para un polinomio de grado 5

$$P(x) = p_0 + p_1 x + p_2 x^2 + p_3 x^3 + p_4 x^4 + p_5 x^5$$

$$P(x) = P(x_0) + \frac{P'(x_0)}{1!}(x - x_0) + \frac{P''(x_0)}{2!}(x - x_0)^2 + \frac{P'''(x_0)}{3!}(x - x_0)^3 + \frac{P^{(4)}(x_0)}{4!}(x$$

$$P(x) = P(x_0)$$

$$+ (x - x_0)\underbrace{\left[\frac{P'(x_0)}{1!} + (x - x_0)\underbrace{\left[\frac{P''(x_0)}{2!} + (x - x_0)\underbrace{\left[\frac{P'''(x_0)}{3!} + (x - x_0)\underbrace{\left[\frac{P^{(4)}(x_0)}{4!} + \right.}\right.}_{Q_3(x)}\right.}_{Q_2(x)}\right.}_{Q_1(x)}$$

Así que si aplicamos el algoritmo de Horner sucesivamente, podemos calcular el valor de las derivadas sucesivas del polinomio en un punto. En el siguiente ejemplo para $x_0 = 1$ y $P(x) = 1 - x + 2x^2 - 3x^3 + 5x^4 - 2x^5$

$$\begin{array}{c|rrrrrrl}
 & -2 & 5 & -3 & 2 & -1 & 1 & \\
1 & & -2 & 3 & 0 & 2 & 1 & \\
\hline
 & -2 & 3 & 0 & 2 & 1 & \boxed{2} & = P(1)\,/\,0! \\
1 & & -2 & 1 & 1 & 3 & & \\
\hline
 & -2 & 1 & 1 & 3 & \boxed{4} & & = P'(1)\,/\,1! \\
1 & & -2 & -1 & 0 & & & \\
\hline
 & -2 & -1 & 0 & \boxed{3} & & & = P''(1)\,/\,2! \\
1 & & -2 & -3 & & & & \\
\hline
 & -2 & -3 & \boxed{-3} & & & & = P'''(1)\,/\,3! \\
1 & & -2 & & & & & \\
\hline
 & -2 & \boxed{-5} & & & & & = P^{(4)}(1)\,/\,4! \\
1 & & & & & & & \\
\hline
 & \boxed{-2} & & & & & & = P^{(5)}(1)\,/\,5!
\end{array}$$

Ejercicio 3

Utilizando la función definida en el ejercicio 1, calcular e imprimir las derivadas sucesivas del polinomio $P(x)$ en el punto x_0 definiendo la función `dersuc(x0,p)` .

(a) Escribir los restos de dividir P una y otra vez por $(x - x_0)$. Lo mismo para R y x_1.

```
p = np.array([1., -1, 2, -3,  5, -2])
x0 = 1.

r = np.array([1., -1, -1, 1, -1, 0, -1, 1])
x1 = -1.
```

(b)

Escribir las derivadas sucesivas para el polinomio P y el punto x_0. Lo mismo para R y x_1. Para ello habremos de multiplicar cada resto por el factorial correspondiente porque

$$r_i = \frac{P^{(i)}(x_0)}{i!} \quad \Longrightarrow \quad P^{(i)}(x_0) = r_i \times i!$$

Nota:

- No usar función para el factorial: ir generándolo de forma similar a como se hizo en la práctica anterior para los polinomios de McLaurin.
- Si queremos duplicar un array numpy `v2 = np.copy(v1)` porque si escribimos `v2 = v1` los dos arrays se almacenan en la misma posición de memoria y si modificamos uno el otro queda modificado.
- Para imprimir los resultados, se puede usar `np.set_printoptions(suppress = True)` para evitar la notación exponencial.
- Si queremos importar la función `horner` del **ejercicio 1**, podemos hacerlo escribiendo al principio del fichero

```
from Ejercicio1 import horner
```

```
%run Ejercicio3a.py
```

```
Restos de dividir P una y otra vez por (x-x0)
[ 2.  4.  3. -3. -5. -2.]


Restos de dividir R una y otra vez por (x-x1)
[ -3.  21. -46.  60. -51.  27.  -8.   1.]
```

```
%run Ejercicio3b.py
```

```
Derivadas sucesivas de P en x0 = 1
[   2.    4.    6.  -18. -120. -240.]


Derivadas sucesivas de R en x1 = -1
[   -3.    21.   -92.   360. -1224.  3240. -5760.  5040.]
```

Ejercicios propuestos

Sea un polinomio mónico ($a_n = 1$) con el término independiente distinto de cero y con todas sus raíces enteras y distintas.

Si aplicamos el algoritmo de Horner sucesivamente, y vamos probando con diferentes divisores del término independiente, podremos encontrar todas las raíces del polinomio.

Por ejemplo, el término independiente del polinomio

$$P(x) = 18 + 9x - 20x^2 - 10x^3 + 2x^4 + x^5$$

es 18. Sus divisores son $\{1, 2, 3, 6, 9, 18\}$ y posibles raíces enteras son $\{1, -1, 2, -2, 3, -3, 6, -6, 9, -9, 18, -18\}$.

Aplicando Horner reiteradamente

$$\begin{array}{r|rrrrrr}
 & 1 & 2 & -10 & -20 & 9 & 18 \\
1 & & 1 & 3 & -7 & -27 & -18 \\ \hline
 & 1 & 3 & -7 & -27 & -18 & \boxed{0} \\
-1 & & -1 & -2 & 9 & 18 & \\ \hline
 & 1 & 2 & -9 & -18 & \boxed{0} & \\
-2 & & -2 & 0 & 18 & & \\ \hline
 & 1 & 0 & -9 & \boxed{0} & & \\
3 & & 3 & 9 & & & \\ \hline
 & 1 & 3 & \boxed{0} & & & \\
-3 & & -3 & & & & \\ \hline
 & 1 & \boxed{0} & & & &
\end{array}$$

llegamos a la conclusión de que

$$P(x) = (x-1)(x+1)(x+2)(x-3)(x+3)$$

Ejercicio 4

(a)

Escribir una función **`divisores(m)`** que calcule los divisores enteros de un número positivo entero **`m`** . La variable de salida será un array numpy que contenga sus divisores y sus opuestos. Para el ejemplo anterior **`m = 18`** y la salida sería el array numpy **`div = [1,-1,2,-2,3,-3,6,-6,9,-9,18,-18]`** .

Probar la función calculando los divisores de 6, 18 y 20.

Nota:

- Necesitamos que los índices sean enteros. Además consideramos **`m`** positivo. Por ello, inicializamos el vector que contendrá los divisores de la siguiente manera

```
m = abs(int(m))
div = np.zeros(2*m)
```

  y después de rellenarlo, lo recortamos a su tamaño final. Si contiene **`n`** elementos distintos de cero, se puede recortar con **`div = div[:n]`** .
- Para calcular el resto de la división se puede usar **`np.remainder(x1,x2)`** que calcula el resto de la división entera y es equivalente al operador `python` **`x1 % x2`**

(b)

Escribir una función **`raices(p)`** que calcule las raíces enteras **simples** de un polinomio y las devuelva en un array numpy. Utilizar las funciones **`divisores`** y **`horner`** .

Teniendo en cuenta que todas las raíces de los siguientes polinomios son enteras y simples, calcular las raíces de

```
p0 = np.array([-1.,0,1])
p1 = np.array([8., -6, -3, 1])
p2 = np.array([15., -2, -16, 2, 1])
p3 = np.array([60.,53, -13, -5, 1])
p4 = np.array([490., 343, -206, -56, 4, 1])
```

Nota:

- Tener en cuenta que el número de raíces de un polinomio es el grado del polinomio.
- Una vez que hemos encontrado una raíz (el **`resto`** es cero)

  1. Almacenamos **`x0`** (el divisor del término independiente), como raíz.
  2. En el siguiente paso utilizamos el último **`cociente`** como el siguiente polinomio a dividir usando Ruffini.
  3. Probamos con el siguiente divisor del término independiente.

```
%run Ejercicio4.py
```

```
(a)


Divisores de 6
[ 1. -1.  2. -2.  3. -3.  6. -6.]

Divisores de 18
[  1.  -1.   2.  -2.   3.  -3.   6.  -6.   9.  -9.  18. -18.]

Divisores de 20
[  1.  -1.   2.  -2.   4.  -4.   5.  -5.  10. -10.  20. -20.]


(b)


Raíces de p0

[ 1. -1.]

Raíces de p1

[ 1. -2.  4.]

Raíces de p2

[ 1. -1.  3. -5.]

Raíces de p3

[-1. -3.  4.  5.]

Raíces de p4

[-1.  2. -5.  7. -7.]
```

Dado el polinomio mónico

$$P(x) = 8 - 22x + 17x^2 + x^3 - 5x^4 + x^5$$

Aplicando Horner reiteradamente

$$\begin{array}{r|rrrrrr}
 & 1 & -5 & 1 & 17 & -22 & 8 \\
1 & & 1 & -4 & -3 & 14 & -8 \\
\hline
 & 1 & -4 & -3 & 14 & -8 & \boxed{0} \\
1 & & 1 & -3 & -6 & 8 & \\
\hline
 & 1 & -3 & -6 & 8 & \boxed{0} & \\
1 & & 1 & -2 & -8 & & \\
\hline
 & 1 & -2 & -8 & \boxed{0} & & \\
-2 & & -2 & 8 & & & \\
\hline
 & 1 & -4 & \boxed{0} & & & \\
4 & & 4 & & & & \\
\hline
 & 1 & \boxed{0} & & & &
\end{array}$$

llegamos a la conclusión de que

$$P(x) = (x-1)(x-1)(x-1)(x+2)(x-4)$$

es decir

$$P(x) = (x-1)^3(x+2)(x-4)$$

Ejercicio 5

Modificar el **Ejercicio 4** de forma que ahora permita obtener raíces múltiples enteras de un polinomio. Si una raíz es múltiple, aparecerá en el array de salida tantas veces como su multiplicidad.

Calcular las raíces enteras de

```
p1 = np.array([8., -22, 17, 1, -5, 1])
p2 = np.array([-135., 378, -369, 140, -9, -6, 1])
p3 = np.array([96., 320, 366, 135, -30, -24, 0, 1])
p4 = np.array([280., 156, -350, -59, 148, -26, -6, 1])
```

```
%run Ejercicio5.py
```

```
Raíces de p1

[ 1.  1.  1. -2.  4.]

Raíces de p2

[ 1.  1.  3.  3.  3. -5.]

Raíces de p3

[-1. -1. -1. -2. -3.  4.  4.]

Raíces de p4

[-1. -1.  2.  2.  2. -5.  7.]
```

Si aplicamos Horner

$$
\begin{array}{c|cccccc}
 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
x_0 & & q_5\,x_0 & q_4\,x_0 & q_3\,x_0 & q_2\,x_0 & q_1\,x_0 \\
\hline
 & q_5 & q_4 & q_3 & q_2 & q_1 & \boxed{q_0} & = P(x_0)
\end{array}
$$

$$
\begin{aligned}
q_5 &= p_5 \\
q_4 &= p_4 + q_5\,x_0 \\
q_3 &= p_3 + q_4\,x_0 \\
q_2 &= p_2 + q_3\,x_0 \\
q_1 &= p_1 + q_2\,x_0 \\
q_0 &= p_0 + q_1\,x_0
\end{aligned}
$$

Si no necesitamos conservar los valores intermedios, sino solo el final, podemos utilizar una única variable

$$
\begin{array}{c|cccccc}
 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
x_0 & & y_0\,x_0 & y_0\,x_0 & y_0\,x_0 & y_0\,x_0 & y_0\,x_0 \\
\hline
 & y_0 & y_0 & y_0 & y_0 & y_0 & \boxed{y_0} & = P(x_0)
\end{array}
$$

$$
\begin{aligned}
y_0 &= p_5 \\
y_0 &= p_4 + y_0\,x_0 \\
y_0 &= p_3 + y_0\,x_0 \\
y_0 &= p_2 + y_0\,x_0 \\
y_0 &= p_1 + y_0\,x_0 \\
y_0 &= p_0 + y_0\,x_0
\end{aligned}
$$

con la ventaja de que este esquema funciona en `numpy` tanto para valores `float` x_0, y_0 como para vectores (`numpy array`) x, y. Es decir, también podemos hacer

$$
\begin{aligned}
y &= p_5 \\
y &= p_4 + y\,x \\
y &= p_3 + y\,x \\
y &= p_2 + y\,x \\
y &= p_1 + y\,x \\
y &= p_0 + y\,x
\end{aligned}
$$

Ejercicio 6

Modificar la función del **Ejercicio 2** pero utilizando vectorización. Tener en cuenta lo siguiente:

- No añadiremos ningún bucle a la función `horner(x0,p)`. La función será `hornerVect(x,p)`.
- Utilizaremos un vector `y` que en el paso final contendrá los valores del polinomio para los puntos del vector `x`.
- El primer valor de `y` es el mismo para todos los puntos, `p[n-1]`, y podríamos hacer `y = p[n-1]` (con `n = len(p)`) o `y = p[-1]`.

Probar la función, sin dibujar, para el polinomio `p0` y los puntos contenidos en `x0` e imprimir `y`, siendo

```
p0 = np.array([1.,2,1])
x0 = np.array([1.,-1])
```

Dibujar el polinomio P en $[-1,1]$.

```
p = np.array([1., -1, 2, -3, 5, -2])
```

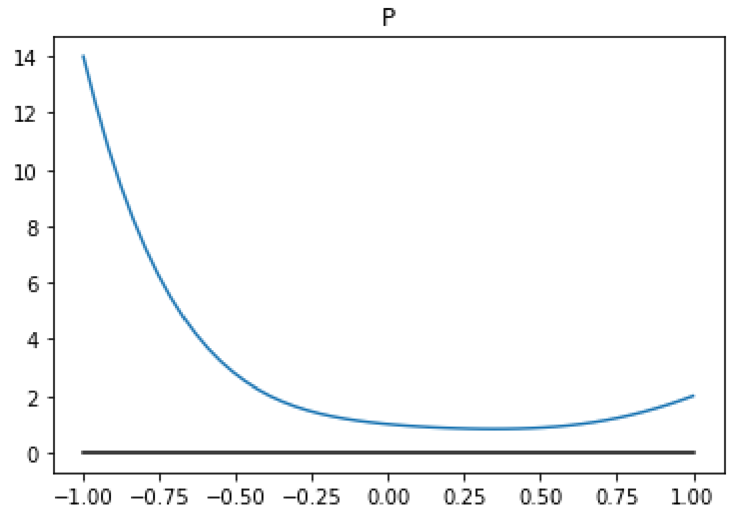
Dibujar también el polinomio R

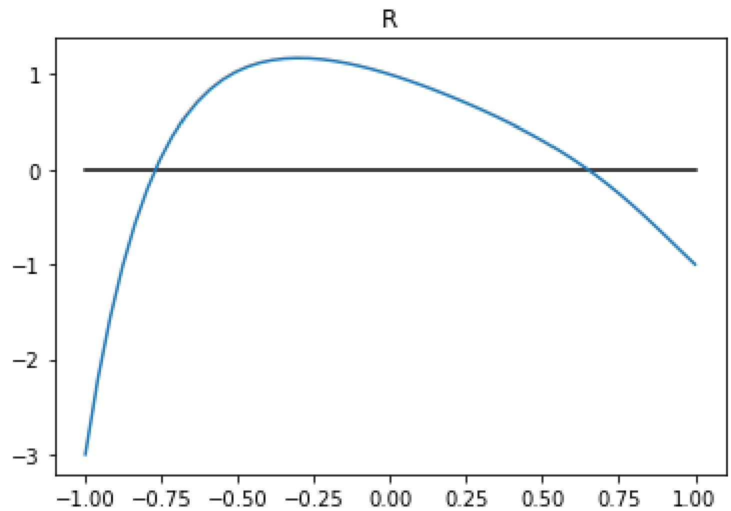
```
r = np.array([1., -1, -1, 1, -1, 0, -1, 1])
```

Utilizando un vector `x = np.linspace(-1,1,1000000)` comparar los tiempos de ejecución con este código y con el del **Ejercicio 2** para dibujar el polinomio `p`.

```
%run Ejercicio6.py
```

```
y =  [4. 0.]
```






```
Tiempo sin vectorización =  3.6953649520874023
Tiempo con vectorización =  0.04923391342163086
```

Ejercicio 7

Modificar el programa del **Ejercicio 3**:

- Ahora debe admitir, en lugar del punto `x0` como variable de entrada, un vector del tipo `x = np.linspace(0,1)` . La función será de la forma `derivadasSuc(p,x)` .
- La variable de salida será una matriz `Y` que contendrá en la primera columna, los valores de P en los puntos de `x` , en la segunda columna, los valores de P' en los puntos de `x` y así sucesivamente hasta la derivada n-ésima, siendo n el grado de P.
- Utilizando esta matriz `Y` , dibujar el polinomio y sus derivadas primera y segunda.

Notas:

- Recordar que la matriz `Y` se puede inicializar `Y = np.zeros((m,n))` donde `m = len(x)` y `n = len(p)` .
- Para extraer una columna `k` de la matriz `Y` sería `Y[:,k]` .
- Si queremos asignar un array numpy hacer `v2 = np.copy(v1)` porque si hacemos `v2 = v1` los dos arrays ocupan el mismo espacio de memoria y si se modifica uno, se modifica también el otro.

Dibujar los polinomios P, P' y P''.

```
p = np.array([1., -1, 2, -3, 5, -2])
```

```
%run Ejercicio7.py
```