Tarea 16 - Interpolacion, Operadores de Newton

October 26, 2019

0.1 Introducción

Para esta tarea se programaron métodos de interpolación basados en operadores de Newton como son Newton hacia adelante y atrás, Newton centrado hacia adelante y atrás e interpolación con operador promedio. Los códigos están escritos en C en el archivo interp2.c con header interp2.h. A partir de estos archivos se genera una biblioteca de python con la ayuda de la herramienta Swig. Se muestran los resultados en este reporte con la ayuda de numpy para el manejo de arreglos.

Importa numpy para manejo de arreglos, matplotlib para la graficación e interp1 con funciones de interpolación escritas en C. Biblioteca interp2 contiene funciones: * gregory_forward_interp: interpolación de Gregory-Newton hacia adelante * gregory_backward_interp: interpolación de Gregory-Newton hacia atrás * gauss_forward_interp: interpolación de Newton centrado hacia adelante * gauss_backward_interp: interpolación de Newton centrado hacia atrás * stirling_interp: Interpolación de Stirling o con operador promedio

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import interp2
```

0.2 Newton hacia adelante

Este es un método para realizar interpolación sobre puntos que estén en el extremo inferior del intervalo de datos que se tiene, debido a cómo está construido. Y tiene la siguiente forma:

$P_n(x) = f(x_0) + \Delta f(x_0)s + \frac{\Delta^2 f(x_0)}{2!}s(s-1) + ... + \frac{\Delta^n f(x_0)}{n!}s(s-1)...(s-n+1)$

Y donde $s = \frac{x-x_0}{h}$, $h = x_1 - x_0$

0.2.1 Resultados

0.2.2 $f(x) = sin(x)$ con $x\epsilon[-\pi, \pi]$

Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 4$ y $n = 6$.

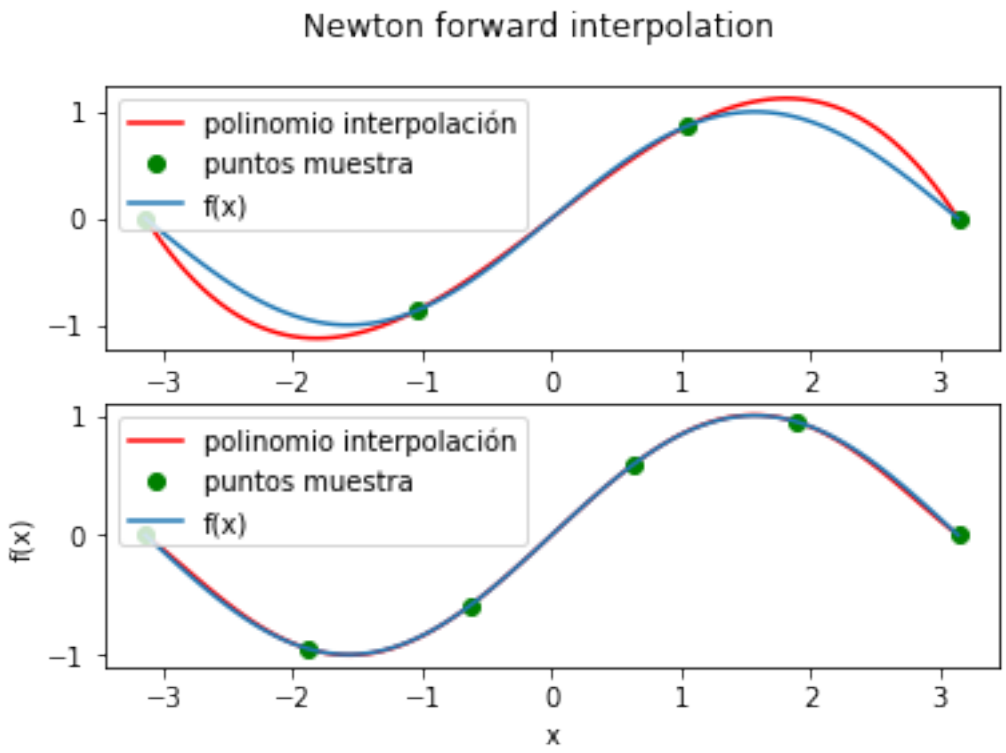
```
In [125]: x1 = np.linspace(-np.pi, np.pi, 4)
          x2 = np.linspace(-np.pi, np.pi, 6)
          p = np.linspace(-np.pi, np.pi, 100)

          print("Tiempo de ejecución: \n")
```

```
        y1 = interp2.gregory_forward_interp(x1, np.sin(x1), p, len(p))
        %time y2 = interp2.gregory_forward_interp(x2, np.sin(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 38 µs, sys: 1e+03 ns, total: 39 µs
Wall time: 49.4 µs
```

Se compara la función $f(x)$ con los valores (p, y) interpolados mediante Newton hacia adelante.

```
In [126]: fig, axs = plt.subplots(2)

        # Interpolation with sample 1
        axs[0].plot(p, y1, 'r', label='polinomio interpolación')

        # Interpolation with sample 1
        axs[1].plot(p, y2, 'r', label='polinomio interpolación')

        # Sample points used for interpolation
        axs[0].plot(x1, np.sin(x1), 'go', label='puntos muestra')
        axs[1].plot(x2, np.sin(x2), 'go', label='puntos muestra')

        # f(x)
        xt = np.linspace(-np.pi, np.pi, 100)
        ft = np.sin(xt)
        axs[0].plot(xt, ft, label='f(x)')
        axs[1].plot(xt, ft, label='f(x)')
        plt.ylabel('f(x)')
        plt.xlabel('x')
        fig.suptitle('Newton forward interpolation')
        axs[0].legend()
        axs[1].legend()
        plt.show()
```



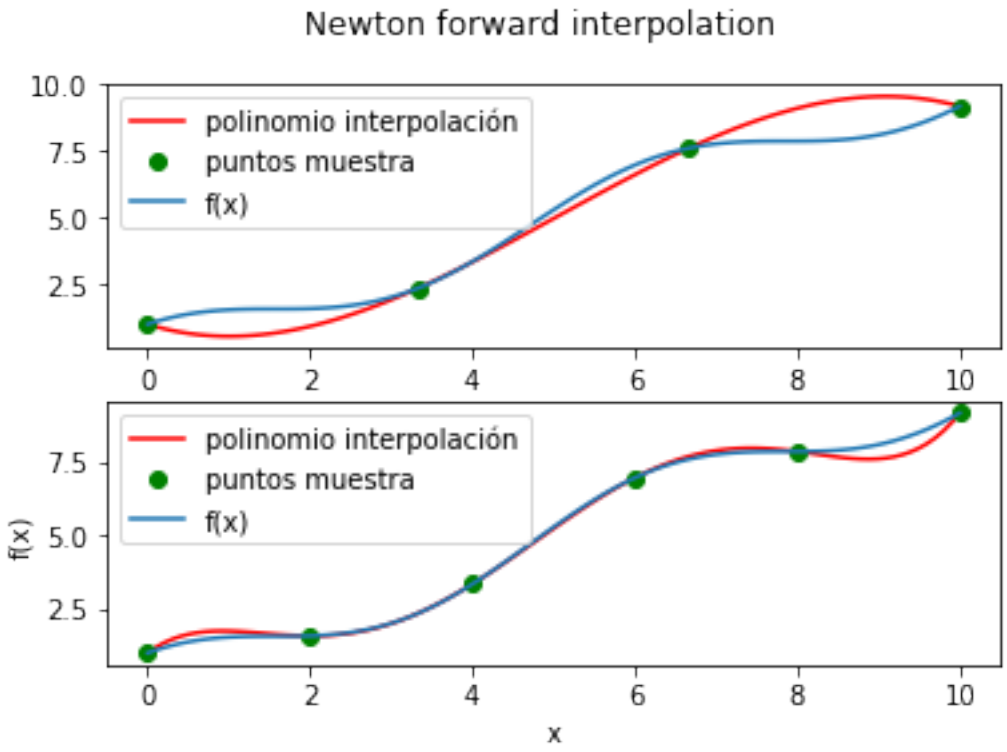
0.2.3 $f(x) = cos(x) + x$ con $x\epsilon[0, 10]$

```
In [6]: def func(x):
            return np.cos(x) + x
```

Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 4$ y $n = 6$.

```
In [127]: x1 = np.linspace(0, 10, 4)
          x2 = np.linspace(0, 10, 6)
          p = np.linspace(0, 10, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.gregory_forward_interp(x1, func(x1), p, len(p))
          %time y2 = interp2.gregory_forward_interp(x2, func(x2), p, len(p))

Tiempo de ejecución:

CPU times: user 69 ţs, sys: 2 ţs, total: 71 ţs
Wall time: 82.3 ţs
```

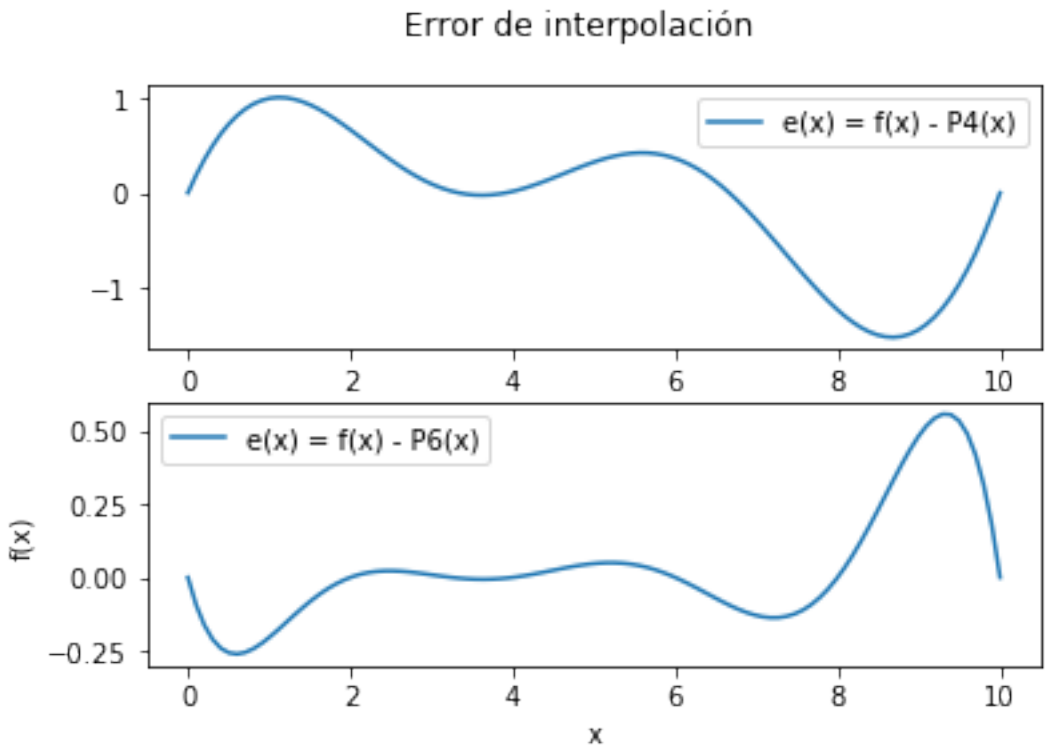
In [128]:
```
fig, axs = plt.subplots(2)

# Interpolation with sample 1
axs[0].plot(p, y1, 'r', label='polinomio interpolación')

# Interpolation with sample 1
axs[1].plot(p, y2, 'r', label='polinomio interpolación')

# Sample points used for interpolation
axs[0].plot(x1, func(x1), 'go', label='puntos muestra')
axs[1].plot(x2, func(x2), 'go', label='puntos muestra')

# f(x)
xt = np.linspace(0, 10, 100)
ft = func(xt)
axs[0].plot(xt, ft, label='f(x)')
axs[1].plot(xt, ft, label='f(x)')
plt.ylabel('f(x)')
plt.xlabel('x')
fig.suptitle('Newton forward interpolation')
axs[0].legend()
axs[1].legend()
plt.show()
```



0.2.4 Análisis Error

Se puede observar que el error de interpolación disminuye considerablemente al pasar de n = 5 a n = 6. También se puede observar que el error tiene un comportamiento oscilatorio. Además de haber un menor error en valores cercanos al extremo izquierdo del intervalo.

```
In [131]: fig, axs = plt.subplots(2)
          xt = np.linspace(0, 10, 100)
          ft = func(xt) - y1
          axs[0].plot(xt, ft, label='e(x) = f(x) - P4(x)')
          ft = func(xt) - y2
          axs[1].plot(xt, ft, label='e(x) = f(x) - P6(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Error de interpolación')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```




```
In [132]: errors = []
          m = 11

          for i in range(2, m):
```

```
            x = np.linspace(0, 10, i)
            p = np.linspace(0, 10, 100)
            y = interp2.gregory_forward_interp(x,func(x),p,len(p))
            errors.append( np.mean( np.abs(func(p) - y) ) )
```

También se puede observar que para n entre 2 y 5 el error se reduce de manera considerable, pero a partir de n = 6, el error deja de disminuir.

```
In [133]: for i in range(2, m):
            print("Error promedio de e(x) con n = {0} es {1}".format(i, errors[i-2]))
```

```
Error promedio de e(x) con n = 2 es 0.6533577440700897
Error promedio de e(x) con n = 3 es 0.6685047375458717
Error promedio de e(x) con n = 4 es 0.5689381813087819
Error promedio de e(x) con n = 5 es 0.40545681682310947
Error promedio de e(x) con n = 6 es 0.12074545250753323
Error promedio de e(x) con n = 7 es 0.08229400026047046
Error promedio de e(x) con n = 8 es 0.014948968653796161
Error promedio de e(x) con n = 9 es 0.00991033079040368
Error promedio de e(x) con n = 10 es 0.0012639638529725384
```

0.3 Newton hacia atrás

Este es un método para realizar interpolación sobre puntos que estén en el extremo superior del intervalo de datos que se tiene, debido a cómo está construido. Y tiene la siguiente forma:

$P_n(x) = f(x_n) + \bigtriangledown f(x_n)s + \frac{\bigtriangledown^2 f(x_n)}{2!}s(s+1) + ... + \frac{\bigtriangledown^n f(x_n)}{n!}s(s+1)...(s+n-1)$

Y donde $s = \frac{x-x_n}{h}$, $h = x_1 - x_0$

0.3.1 Resultados

0.3.2 $f(x) = sin(x)$ con $x\epsilon[-\pi, \pi]$

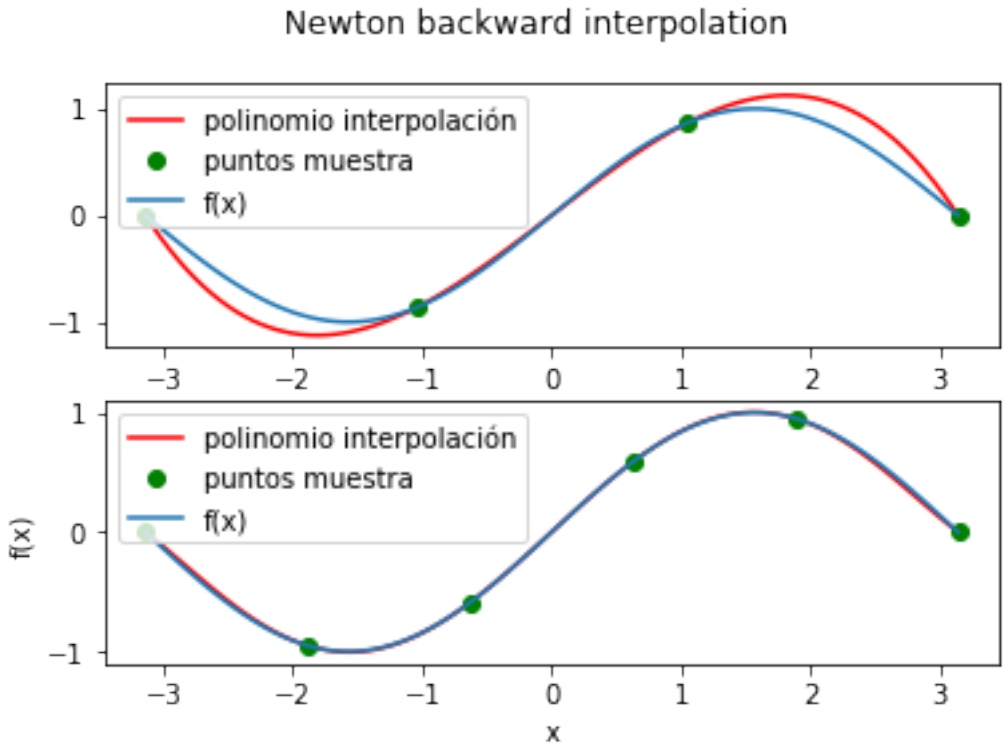
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 4$ y $n = 6$.

```
In [136]: x1 = np.linspace(-np.pi, np.pi, 4)
          x2 = np.linspace(-np.pi, np.pi, 6)
          p = np.linspace(-np.pi, np.pi, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.gregory_backward_interp(x1, np.sin(x1), p, len(p))
          %time y2 = interp2.gregory_backward_interp(x2, np.sin(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 58 ţs, sys: 2 ţs, total: 60 ţs
Wall time: 69.9 ţs
```

Se compara la función $f(x)$ con los valores (p, y) interpolados mediante Newton hacia atrás.

```
In [137]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, np.sin(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, np.sin(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(-np.pi, np.pi, 100)
          ft = np.sin(xt)
          axs[0].plot(xt, ft, label='f(x)')
          axs[1].plot(xt, ft, label='f(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Newton backward interpolation')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```



0.3.3 $f(x) = cos(x) + x$ **con** $x\epsilon[0, 10]$

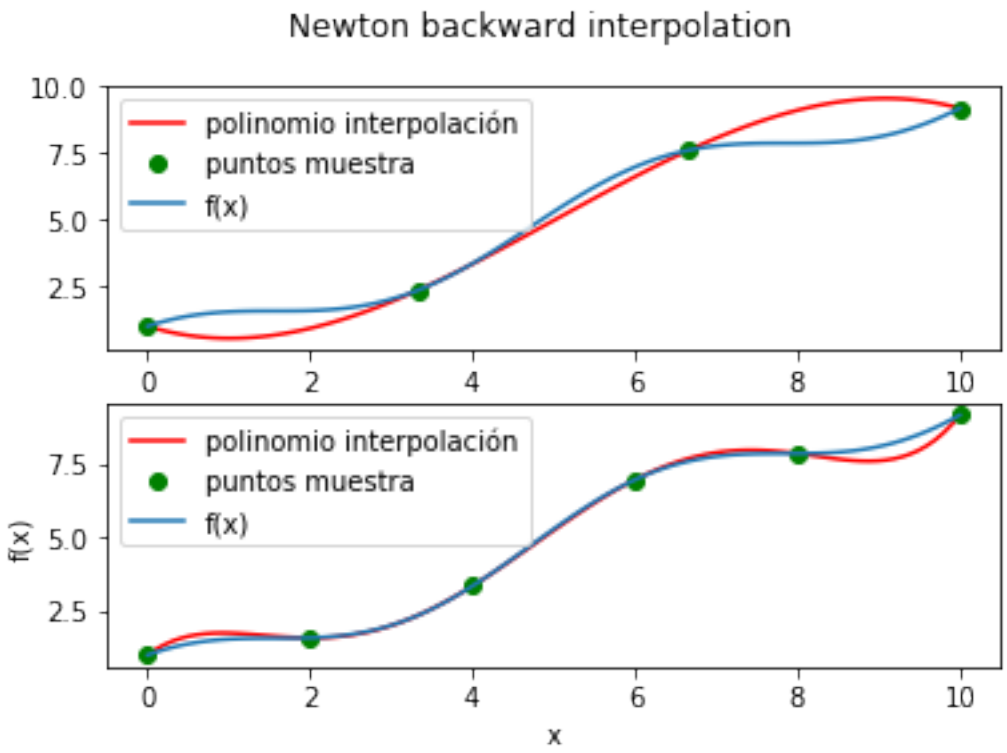
```
In [138]: def func(x):
              return np.cos(x) + x
```

Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 4$ y $n = 6$.

```
In [139]: x1 = np.linspace(0, 10, 4)
          x2 = np.linspace(0, 10, 6)
          p = np.linspace(0, 10, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.gregory_backward_interp(x1, func(x1), p, len(p))
          %time y2 = interp2.gregory_backward_interp(x2, func(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 36 ţs, sys: 1 ţs, total: 37 ţs
Wall time: 46 ţs
```

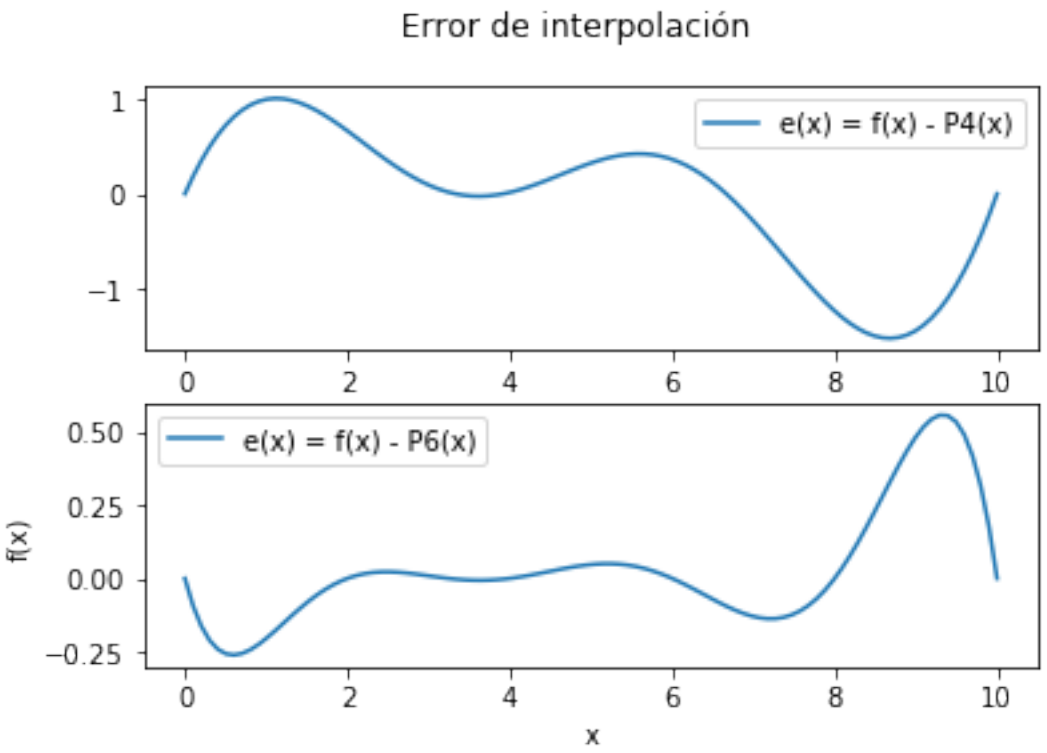
```
In [140]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, func(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, func(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(0, 10, 100)
          ft = func(xt)
          axs[0].plot(xt, ft, label='f(x)')
          axs[1].plot(xt, ft, label='f(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Newton backward interpolation')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```



0.3.4 Análisis Error

Al igual que Newton hacia adelante, se puede observar que el error de interpolación disminuye considerablemente al pasar de n = 5 a n = 6. Aunque en este caso no se notó mucha diferencia en error en el extremo izquierdo del intervalo.

```
In [141]: fig, axs = plt.subplots(2)
          xt = np.linspace(0, 10, 100)
          ft = func(xt) - y1
          axs[0].plot(xt, ft, label='e(x) = f(x) - P4(x)')
          ft = func(xt) - y2
          axs[1].plot(xt, ft, label='e(x) = f(x) - P6(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Error de interpolación')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```




```
In [142]: errors = []
          m = 11

          for i in range(2, m):
              x = np.linspace(0, 10, i)
              p = np.linspace(0, 10, 100)
              y = interp2.gregory_backward_interp(x,func(x),p,len(p))
              errors.append( np.mean( np.abs(func(p) - y) ) )
```

También se puede observar que para n entre 2 y 5 el error se reduce de manera considerable, pero a partir de n = 6, el error deja de disminuir.

```
In [143]: for i in range(2, m):
              print("Error promedio de e(x) con n = {0} es {1}".format(i, errors[i-2]))
```

```
Error promedio de e(x) con n = 2 es 0.6533577440700895
Error promedio de e(x) con n = 3 es 0.6685047375458717
Error promedio de e(x) con n = 4 es 0.5689381813087817
Error promedio de e(x) con n = 5 es 0.4054568168231094
Error promedio de e(x) con n = 6 es 0.12074545250753342
Error promedio de e(x) con n = 7 es 0.08229400026047023
Error promedio de e(x) con n = 8 es 0.014948968653796304
Error promedio de e(x) con n = 9 es 0.009910330790403944
Error promedio de e(x) con n = 10 es 0.0012639638529717456
```

0.4 Newton centrado hacia adelante (Gauss hacia adelante)

Este es un método para realizar interpolación derivado de Newton hacia adelante y se usa sobre puntos que estén en la mitad del intervalo de datos que se tiene, debido a cómo está construido. Y tiene la siguiente forma:

$P_n(x) = f(x_0) + \triangle f(x_0)s + \frac{\triangle^2 f(x_{-1})}{2!}s(s-1) + \frac{\triangle^3 f(x_{-1})}{3!}(s+1)s(s-1) + \frac{\triangle^4 f(x_{-2})}{4!}(s+1)s(s-1)(s-2) + ...$

Y donde $s = \frac{x-x_0}{h}$, $h = x_1 - x_0$

0.4.1 Resultados

0.4.2 $f(x) = sin(x)$ con $x\epsilon[-\pi, \pi]$

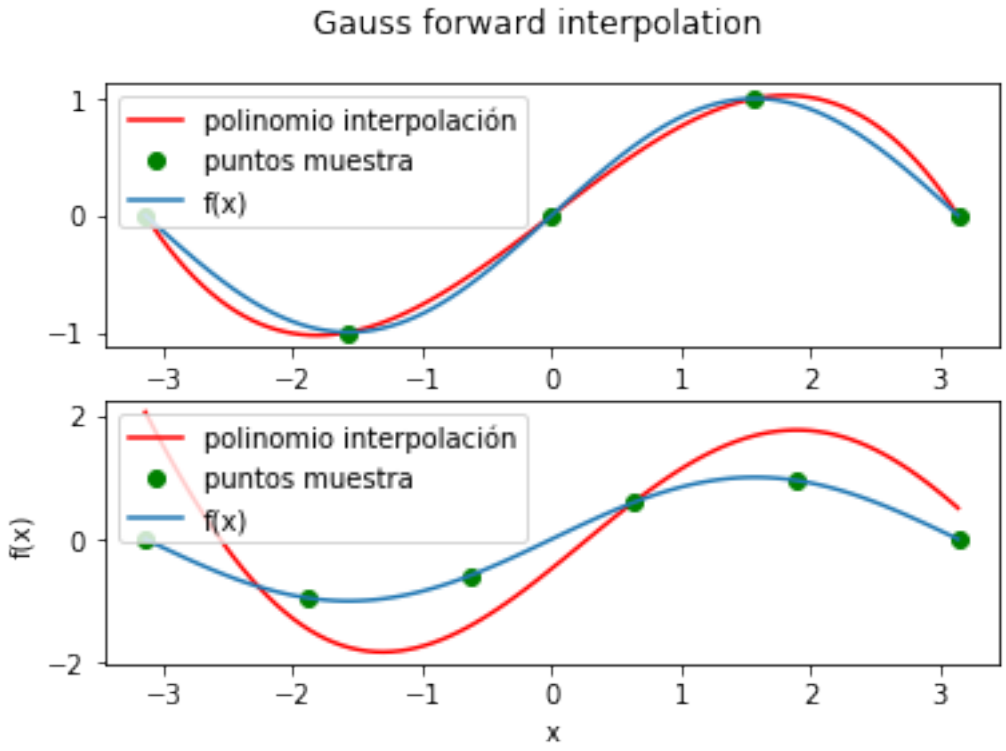
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [166]: x1 = np.linspace(-np.pi, np.pi, 5)
          x2 = np.linspace(-np.pi, np.pi, 6)
          p = np.linspace(-np.pi, np.pi, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.gauss_forward_interp(x1, np.sin(x1), p, len(p))
          %time y2 = interp2.gauss_forward_interp(x2, np.sin(x2), p, len(p))

Tiempo de ejecución:

CPU times: user 61 ţs, sys: 2 ţs, total: 63 ţs
Wall time: 72.7 ţs
```

Se compara la función $f(x)$ con los valores (p, y) interpolados mediante Newton centrado hacia adelante.

```
In [167]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, np.sin(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, np.sin(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(-np.pi, np.pi, 100)
          ft = np.sin(xt)
```

```
axs[0].plot(xt, ft, label='f(x)')
axs[1].plot(xt, ft, label='f(x)')
plt.ylabel('f(x)')
plt.xlabel('x')
fig.suptitle('Gauss forward interpolation')
axs[0].legend()
axs[1].legend()
plt.show()
```



0.4.3 $f(x) = cos(x) + x$ con $x \epsilon [0, 10]$

```
In [49]: def func(x):
             return np.cos(x) + x
```

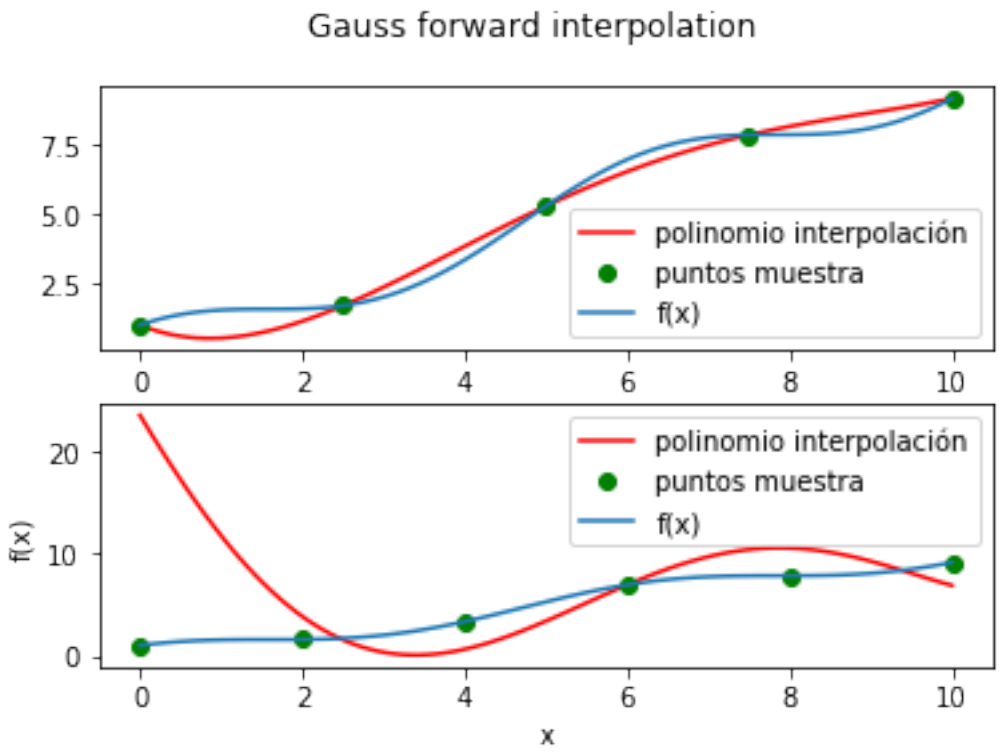
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [172]: x1 = np.linspace(0, 10, 5)
          x2 = np.linspace(0, 10, 6)
          p = np.linspace(0, 10, 100)

          print("Tiempo de ejecución: \n")
```

```
        y1 = interp2.gauss_forward_interp(x1, func(x1), p, len(p))
        %time y2 = interp2.gauss_forward_interp(x2, func(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 38 ţs, sys: 1 ţs, total: 39 ţs
Wall time: 48.2 ţs
```

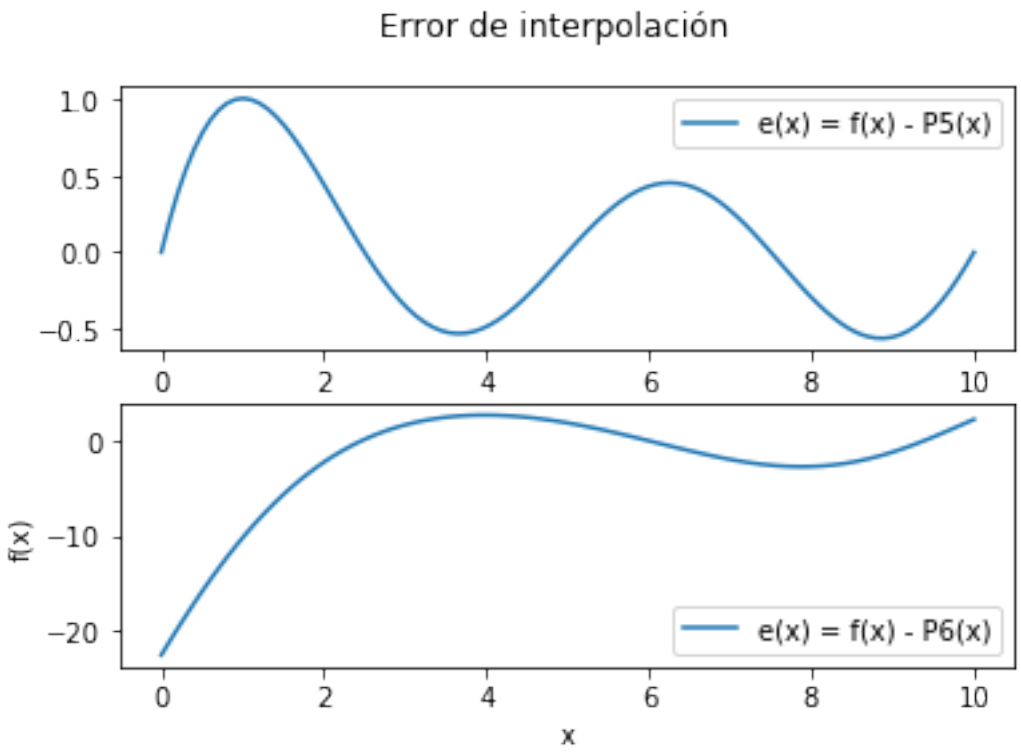
In [173]:
```
fig, axs = plt.subplots(2)

        # Interpolation with sample 1
        axs[0].plot(p, y1, 'r', label='polinomio interpolación')

        # Interpolation with sample 1
        axs[1].plot(p, y2, 'r', label='polinomio interpolación')

        # Sample points used for interpolation
        axs[0].plot(x1, func(x1), 'go', label='puntos muestra')
        axs[1].plot(x2, func(x2), 'go', label='puntos muestra')

        # f(x)
        xt = np.linspace(0, 10, 100)
        ft = func(xt)
        axs[0].plot(xt, ft, label='f(x)')
        axs[1].plot(xt, ft, label='f(x)')
        plt.ylabel('f(x)')
        plt.xlabel('x')
        fig.suptitle('Gauss forward interpolation')
        axs[0].legend()
        axs[1].legend()
        plt.show()
```



0.4.4 Análisis Error

Para este caso se puede observar que el error siempre es considerablemente mayor para n par, esto debido a que no se tiene un elemento central, sino 2 elementos centrales causando que el calculo sea erroneo, aunque aún así el error en valores intermedio tiende a ser menor. Al contrario cuando la cantidad de muestras es impar la interpolación funciona bastante bien.

```
In [175]: fig, axs = plt.subplots(2)
          xt = np.linspace(0, 10, 100)
          ft = func(xt) - y1
          axs[0].plot(xt, ft, label='e(x) = f(x) - P5(x)')
          ft = func(xt) - y2
          axs[1].plot(xt, ft, label='e(x) = f(x) - P6(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Error de interpolación')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```




```
In [176]: errors = []
          m = 11

          for i in range(2, m):
              x = np.linspace(0, 10, i)
              p = np.linspace(0, 10, 100)
              y = interp2.gauss_forward_interp(x,func(x),p,len(p))
              errors.append( np.mean( np.abs(func(p) - y) ) )
```

También se puede observar que para n entre 2 y 5 el error se reduce de manera considerable, pero a partir de n = 6, el error deja de disminuir.

```
In [177]: for i in range(2, m):
              print("Error promedio de e(x) con n = {0} es {1}".format(i, errors[i-2]))
```

```
Error promedio de e(x) con n = 2 es 0.6533577440700895
Error promedio de e(x) con n = 3 es 0.6685047375458717
Error promedio de e(x) con n = 4 es 3.654456287695191
Error promedio de e(x) con n = 5 es 0.4054568168231094
Error promedio de e(x) con n = 6 es 3.6329565134896513
Error promedio de e(x) con n = 7 es 0.08229400026047026
Error promedio de e(x) con n = 8 es 1.0760476019974563
Error promedio de e(x) con n = 9 es 0.00991033079040349
Error promedio de e(x) con n = 10 es 0.6483010785243715
```

0.5 Newton centrado hacia atrás (Gauss hacia atrás)

Este es un método para realizar interpolación derivado de Newton hacia atrás y se usa sobre puntos que estén en la mitad del intervalo de datos que se tiene, debido a cómo está construido. Y tiene la siguiente forma:

$P_n(x) = f(x_0) + \triangle f(x_{-1})s + \frac{\triangle^2 f(x_{-1})}{2!}s(s+1) + \frac{\triangle^3 f(x_{-2})}{3!}(s+1)s(s-1) + \frac{\triangle^4 f(x_{-2})}{4!}(s+2)(s+1)s(s-1) + ...$

Y donde $s = \frac{x-x_0}{h}$, $h = x_1 - x_0$

0.5.1 Resultados

0.5.2 $f(x) = sin(x)$ con $x\epsilon[-\pi, \pi]$

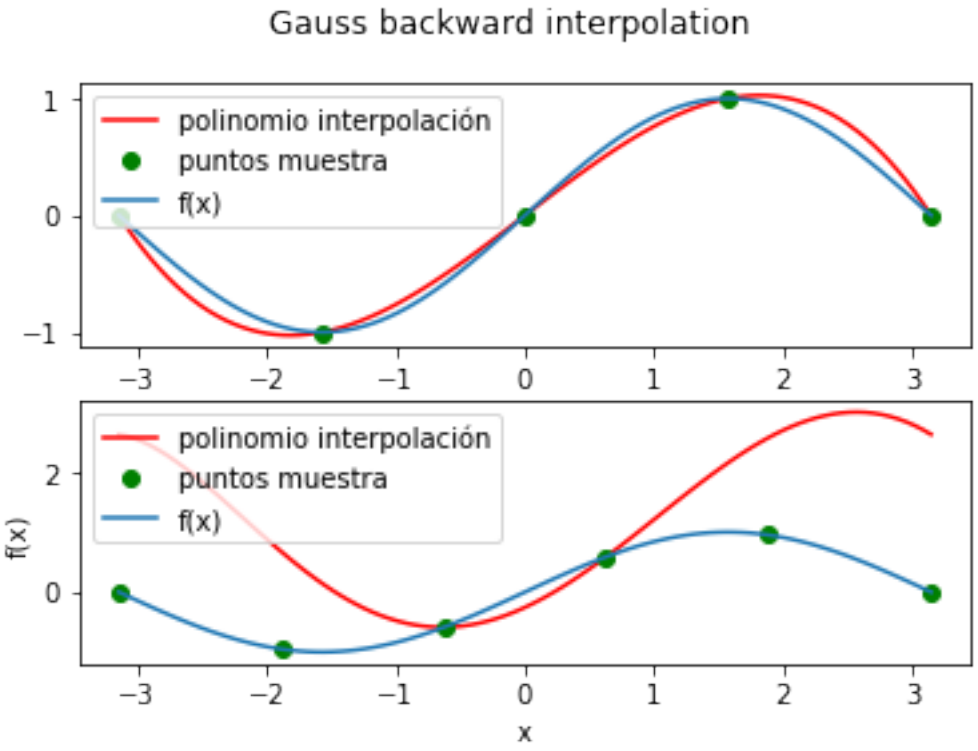
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [179]: x1 = np.linspace(-np.pi, np.pi, 5)
          x2 = np.linspace(-np.pi, np.pi, 6)
          p = np.linspace(-np.pi, np.pi, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.gauss_backward_interp(x1, np.sin(x1), p, len(p))
          %time y2 = interp2.gauss_backward_interp(x2, np.sin(x2), p, len(p))

Tiempo de ejecución:

CPU times: user 62 ţs, sys: 1e+03 ns, total: 63 ţs
Wall time: 75.6 ţs
```

Se compara la función $f(x)$ con los valores (p, y) interpolados mediante Newton centrado hacia atrás.

```
In [180]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, np.sin(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, np.sin(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(-np.pi, np.pi, 100)
          ft = np.sin(xt)
```

```
axs[0].plot(xt, ft, label='f(x)')
axs[1].plot(xt, ft, label='f(x)')
plt.ylabel('f(x)')
plt.xlabel('x')
fig.suptitle('Gauss backward interpolation')
axs[0].legend()
axs[1].legend()
plt.show()
```



0.5.3 $f(x) = cos(x) + x$ con $x\epsilon[0, 10]$

```
In [49]: def func(x):
             return np.cos(x) + x
```

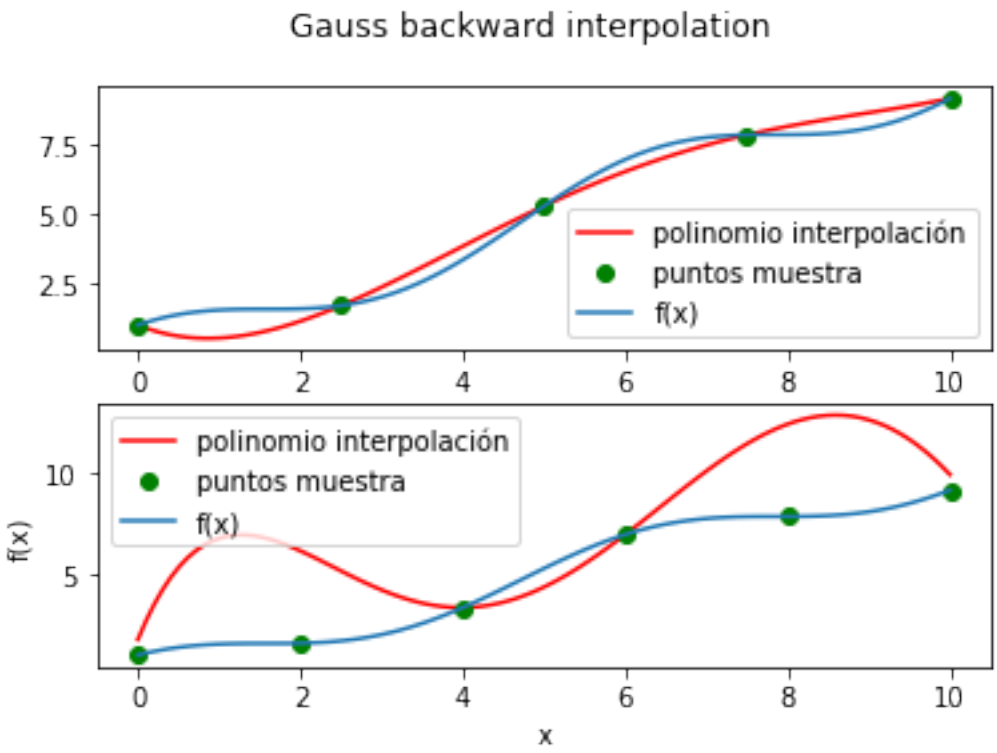
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [183]: x1 = np.linspace(0, 10, 5)
          x2 = np.linspace(0, 10, 6)
          p = np.linspace(0, 10, 100)

          print("Tiempo de ejecución: \n")
```

```
        y1 = interp2.gauss_backward_interp(x1, func(x1), p, len(p))
        %time y2 = interp2.gauss_backward_interp(x2, func(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 47 µs, sys: 1e+03 ns, total: 48 µs
Wall time: 57.2 µs
```

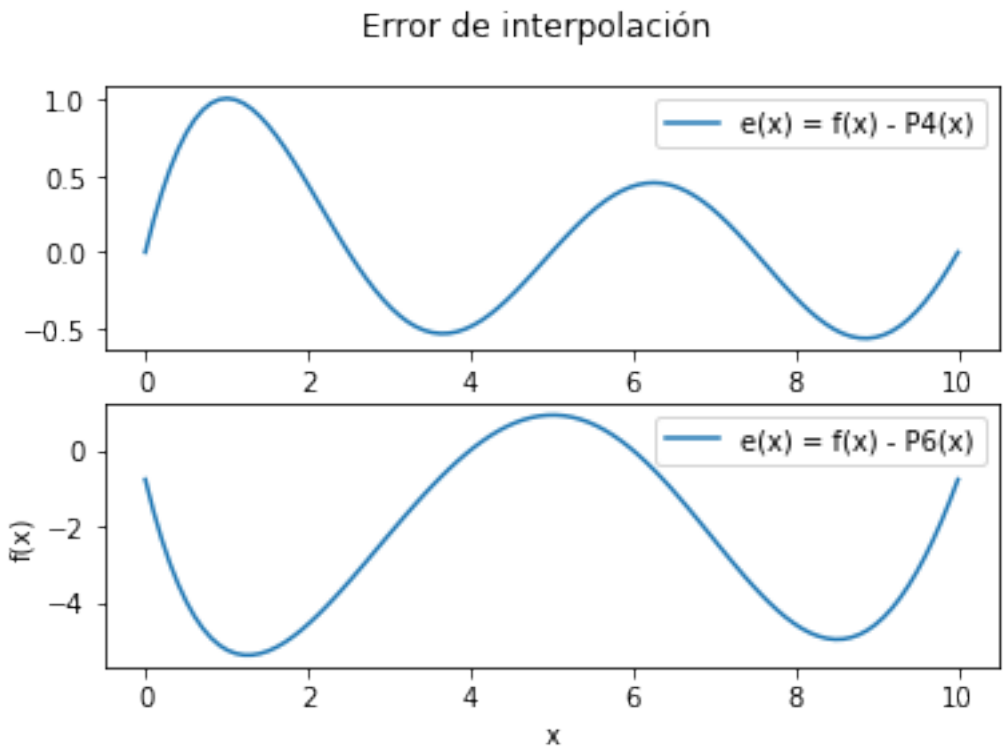
```
In [184]: fig, axs = plt.subplots(2)

        # Interpolation with sample 1
        axs[0].plot(p, y1, 'r', label='polinomio interpolación')

        # Interpolation with sample 1
        axs[1].plot(p, y2, 'r', label='polinomio interpolación')

        # Sample points used for interpolation
        axs[0].plot(x1, func(x1), 'go', label='puntos muestra')
        axs[1].plot(x2, func(x2), 'go', label='puntos muestra')

        # f(x)
        xt = np.linspace(0, 10, 100)
        ft = func(xt)
        axs[0].plot(xt, ft, label='f(x)')
        axs[1].plot(xt, ft, label='f(x)')
        plt.ylabel('f(x)')
        plt.xlabel('x')
        fig.suptitle('Gauss backward interpolation')
        axs[0].legend()
        axs[1].legend()
        plt.show()
```



0.5.4 Análisis Error

Al igual que con Gauss hacia adelante, debido a que el método asume que hay un elemento central en las muestras, esto implica que n debe ser impar para que funcione bien. Por lo que se nota que el error es mucho mayor para n par, aunque en general menor en las partes intermedias del intervalo.

```
In [185]: fig, axs = plt.subplots(2)
          xt = np.linspace(0, 10, 100)
          ft = func(xt) - y1
          axs[0].plot(xt, ft, label='e(x) = f(x) - P4(x)')
          ft = func(xt) - y2
          axs[1].plot(xt, ft, label='e(x) = f(x) - P6(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Error de interpolación')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```




```
In [186]: errors = []
          m = 11

          for i in range(2, m):
              x = np.linspace(0, 10, i)
              p = np.linspace(0, 10, 100)
              y = interp2.gauss_backward_interp(x,func(x),p,len(p))
              errors.append( np.mean( np.abs(func(p) - y) ) )
```

También se puede observar que para n entre 2 y 5 el error se reduce de manera considerable, pero a partir de n = 6, el error deja de disminuir.

```
In [187]: for i in range(2, m):
              print("Error promedio de e(x) con n = {0} es {1}".format(i, errors[i-2]))
```

```
Error promedio de e(x) con n = 2 es 0.6533577440700895
Error promedio de e(x) con n = 3 es 0.6685047375458718
Error promedio de e(x) con n = 4 es 2.5537555593521075
Error promedio de e(x) con n = 5 es 0.4054568168231094
Error promedio de e(x) con n = 6 es 2.6771250408033613
Error promedio de e(x) con n = 7 es 0.08229400026047022
Error promedio de e(x) con n = 8 es 1.6151108120594193
Error promedio de e(x) con n = 9 es 0.009910330790403215
Error promedio de e(x) con n = 10 es 1.2617364125253612
```

0.6 Operador promedio (Stirling)

Este es un método para realizar interpolación derivado de promediar Newton hacia adelante y Newton hacia atrás y se usa sobre puntos que estén en la mitad del intervalo de datos que se tiene, debido a cómo está construido. Y tiene la siguiente forma:

$P_n(x) = f(x_0) + [\frac{\triangle f(x_0)+\triangle f(x_{-1})}{2}]s + \frac{\triangle^2 f(x_{-1})}{2!}s^2 + [\frac{\triangle^3 f(x_{-1})+\triangle^3 f(x_{-2})}{2}]\frac{s(s^2-1)}{3!} + \triangle^4 f(x_{-2})\frac{s^2(s^2-1)}{4!} +$ ...

Y donde $s = \frac{x-x_0}{h}$, $h = x_1 - x_0$

0.6.1 Resultados

0.6.2 $f(x) = sin(x)$ con $x\epsilon[-\pi, \pi]$

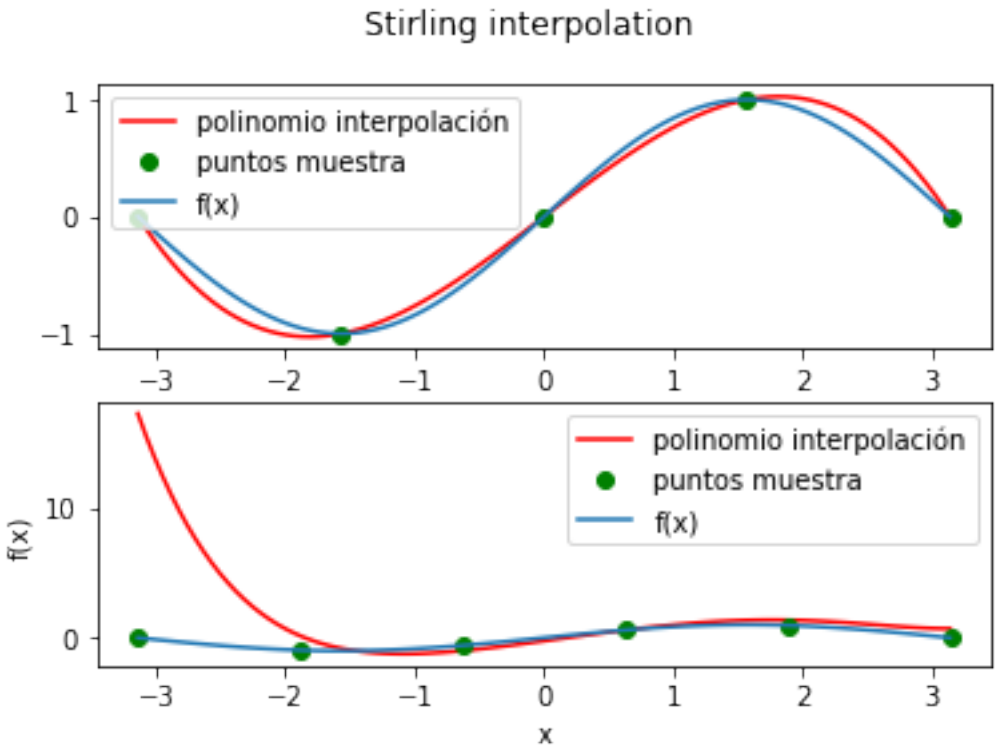
Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [190]: x1 = np.linspace(-np.pi, np.pi, 5)
          x2 = np.linspace(-np.pi, np.pi, 6)
          p = np.linspace(-np.pi, np.pi, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.stirling_interp(x1, np.sin(x1), p, len(p))
          %time y2 = interp2.stirling_interp(x2, np.sin(x2), p, len(p))

Tiempo de ejecución:

CPU times: user 110 µs, sys: 3 µs, total: 113 µs
Wall time: 124 µs
```

Se compara la función $f(x)$ con los valores (p, y) interpolados mediante Stirling.

```
In [191]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, np.sin(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, np.sin(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(-np.pi, np.pi, 100)
          ft = np.sin(xt)
          axs[0].plot(xt, ft, label='f(x)')
```

```
axs[1].plot(xt, ft, label='f(x)')
plt.ylabel('f(x)')
plt.xlabel('x')
fig.suptitle('Stirling interpolation')
axs[0].legend()
axs[1].legend()
plt.show()
```



0.6.3 $f(x) = cos(x) + x$ con $x\epsilon[0, 10]$

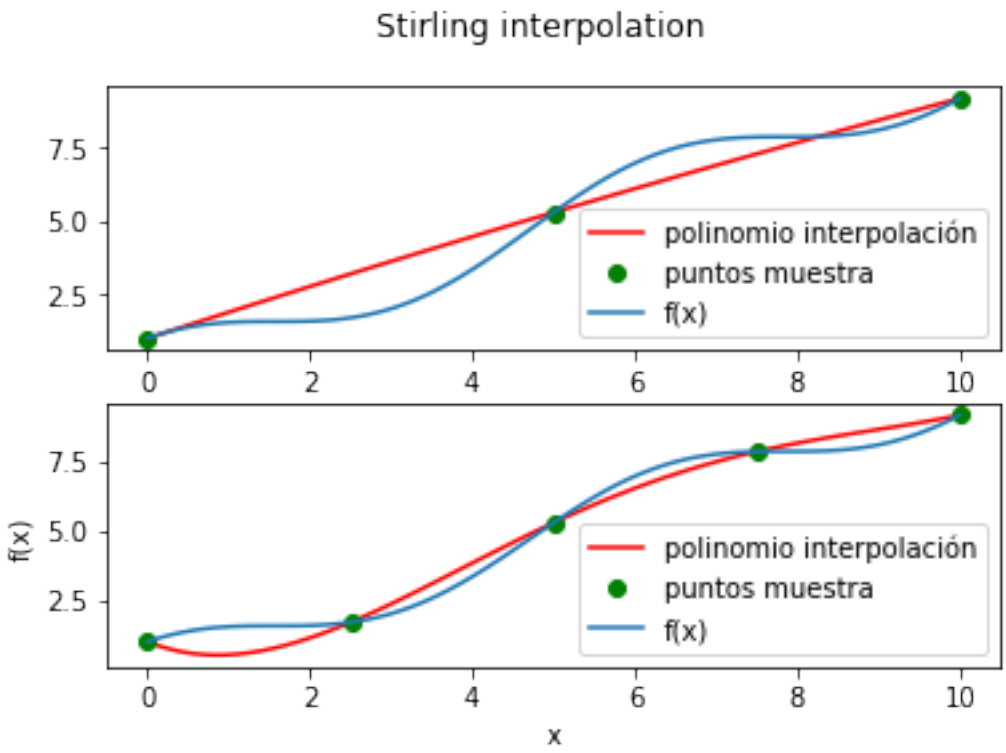
```
In [192]: def func(x):
              return np.cos(x) + x
```

Se calcula polinomio de interpolación a partir de los puntos $x_0, x_1, ..., x_n$ y los valores de $f(x_0), f(x_1), ..., f(x_n)$ con $n = 5$ y $n = 6$.

```
In [193]: x1 = np.linspace(0, 10, 3)
          x2 = np.linspace(0, 10, 5)
          p = np.linspace(0, 10, 100)

          print("Tiempo de ejecución: \n")

          y1 = interp2.stirling_interp(x1, func(x1), p, len(p))
          %time y2 = interp2.stirling_interp(x2, func(x2), p, len(p))
```

```
Tiempo de ejecución:

CPU times: user 38 µs, sys: 1 µs, total: 39 µs
Wall time: 47.2 µs
```

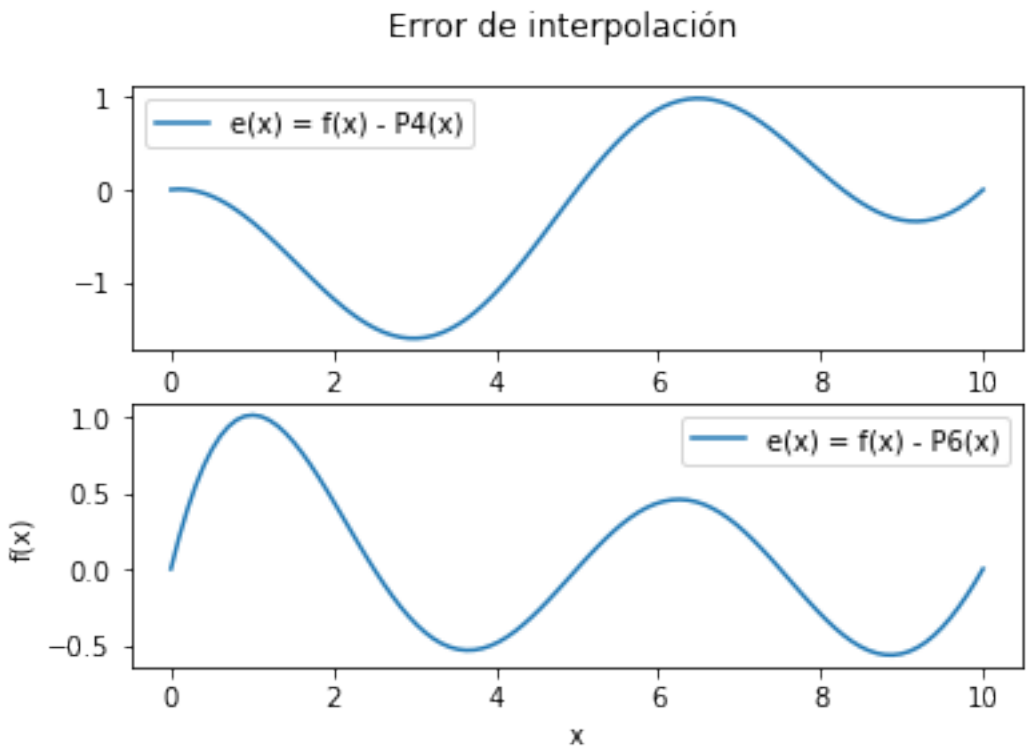
```
In [194]: fig, axs = plt.subplots(2)

          # Interpolation with sample 1
          axs[0].plot(p, y1, 'r', label='polinomio interpolación')

          # Interpolation with sample 1
          axs[1].plot(p, y2, 'r', label='polinomio interpolación')

          # Sample points used for interpolation
          axs[0].plot(x1, func(x1), 'go', label='puntos muestra')
          axs[1].plot(x2, func(x2), 'go', label='puntos muestra')

          # f(x)
          xt = np.linspace(0, 10, 100)
          ft = func(xt)
          axs[0].plot(xt, ft, label='f(x)')
          axs[1].plot(xt, ft, label='f(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Stirling interpolation')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```



0.6.4 Análisis Error

Al igual que con Gauss hacia adelante y atrás, debido a que el método asume que hay un elemento central en las muestras, esto implica que n debe ser impar para que funcione bien.

```
In [195]: fig, axs = plt.subplots(2)
          xt = np.linspace(0, 10, 100)
          ft = func(xt) - y1
          axs[0].plot(xt, ft, label='e(x) = f(x) - P4(x)')
          ft = func(xt) - y2
          axs[1].plot(xt, ft, label='e(x) = f(x) - P6(x)')
          plt.ylabel('f(x)')
          plt.xlabel('x')
          fig.suptitle('Error de interpolación')
          axs[0].legend()
          axs[1].legend()
          plt.show()
```




```
In [196]: errors = []
          m = 11

          for i in range(2, m):
              x = np.linspace(0, 10, i)
              p = np.linspace(0, 10, 100)
              y = interp2.stirling_interp(x,func(x),p,len(p))
              errors.append( np.mean( np.abs(func(p) - y) ) )
```

También se puede observar que para n entre 2 y 5 el error se reduce de manera considerable, pero a partir de n = 6, el error deja de disminuir.

```
In [197]: for i in range(2, m):
              print("Error promedio de e(x) con n = {0} es {1}".format(i, errors[i-2]))
```

```
Error promedio de e(x) con n = 2 es 0.6533577440700895
Error promedio de e(x) con n = 3 es 0.6685047375458717
Error promedio de e(x) con n = 4 es 2.110754292941608
Error promedio de e(x) con n = 5 es 0.40545681682310936
Error promedio de e(x) con n = 6 es 10.013048821092973
Error promedio de e(x) con n = 7 es 3.597141622999521
Error promedio de e(x) con n = 8 es 60.317222450715875
Error promedio de e(x) con n = 9 es 34.908075944483016
Error promedio de e(x) con n = 10 es 5302.91184020994
```

0.6.5 Conclusiones

Estos métodos están basados en operadores de Newton y como se observó en la tarea anterior de métodos de interpolación, el método de Newton fue de los más rápidos, lo que hace que estos métodos tengan la misma rapidez. Todos los métodos se lograron ejecutar en tiempos de entre 40 a 70 μs.

Teóricamente el polinomio de interpolación es único para un conjunto de puntos dados y todos los métodos producen el mismo valor de la función en cualquier punto intermedio. Pero en la práctica surgen diferentes tipos de errores y por motivos diferentes.

Por ejemplo, si se usan demasiados datos, el polinomio generado podría sufrir de muchas oscilaciones haciendo que en ciertos puntos no se tenga una estimación correcta. Como se observa en las gráficas de error, este tiende a tener un comportamiento oscilatorio debido a la forma que tienen los polinomios.

También el error de redondeo se verá afectado dependiendo del rango de valores que queremos interpolar y el método que usemos. Por ejemplo, para Newton hacia adelante debido a que $s = \frac{x-x_0}{h}$ si el punto x donde evaluamos el polinomio es muy cercano a x_0, entonces s estará en el intervalo $(0, 1)$ y como el tener valores con valor absoluto menor a 1 ayuda a reducir el error de redondeo, nuestra interpolación será más precisa. De ahí la importacia de conocer cómo funcionan estos métodos y las implicaciones que tienen a la hora de usarlos en para interpolar valores en diferentes intervalos.

Por último, también se observó que por la forma en que se implementaron Gauss hacia adelante y hacia atrás y Stirling, se asume que la muestra es de tamaño impar, de forma que siempre hay un sólo elemento central que divide a la muestra en partes iguales.