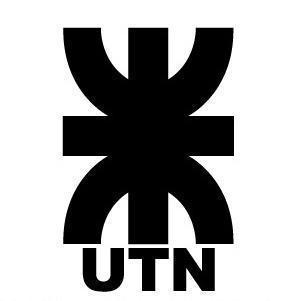
**UNIVERSIDAD TECNOLÓGICA NACIONAL**

**FACULTAD REGIONAL DE SANTA FE**

**TRABAJO PRÁCTICO N° 3**

**Interpolación y Mínimos Cuadrados**

**Cátedra:** Matemática Superior

**Carrera:** Ingeniería en Sistemas de Información

**Comisión:** 3A

**Año:** 2021 - 2° Cuatrimestre

**Integrantes:** Albani, Juan - juanalbani48@gmail.com

Musso, Emilio - emilio.musso2898@gmail.com

Nuñez, Nahuel - nahuelnunezz14@gmail.com

Peiretti, Tomás - tomaspeiretti@gmail.com

Riquelme, Nahuel - nahuelriquelme00@gmail.com

**ÍNDICE**

[**1 - DESARROLLO**](#_afd6d2qsyxmj) **3**

[1.1 - SEÑAL BIOLÓGICA - ECG](#_cqhij6s3ds1r) 3

[1.1.1 - APROXIMACIÓN DE LA SEÑAL](#_dcyd3sm7bgoq) 3

[1.1.1.1 - PLANTEO DE LAS CONDICIONES INICIALES](#_btaldze2utjd) 3

[1.1.1.2 - CALCULO DE MINIMOS CUADRADOS](#_e1vghekh87nf) 4

[1.1.1.3 - PROBLEMAS ENFRENTADOS](#_ezryt3yjq9xx) 5

[1.1.1.4 - TEORIAS FORMULADAS](#_hzo6qu12l2mo) 6

[1.1.1.5 - APROXIMACIONES LOGRADAS](#_vqd1ha9t63ey) 6

[1.1.2 - DETERMINACIÓN DE LA DURACIÓN Y AMPLITUD DE CADA ONDA](#_1ca6r61y0drh) 8

[1.1.3 - ANÁLISIS DEL AJUSTE LOGRADO](#_obv6g0ihx4ke) 9

[1.2 - POLINOMIO DE NEWTON](#_et00hcvlzyl6) 10

[1.2.1 - MODIFICACIÓN DE LA TABLA DE DIFERENCIAS DIVIDIDAS](#_t2lz2wdnpjuf) 10

[1.2.2 - VERIFICACIONES CON EJEMPLOS](#_fog3ktozzrkz) 13

[1.2.2.1 - EJEMPLO 0](#_qwhfs0goi2b9) 13

[1.2.2.2 - EJEMPLO 1](#_8jmjez5ap0b7) 14

[1.2.2.3 - EJEMPLO 2](#_wbbgq0ilc82) 15

[1.2.2.4 - EJEMPLO 3](#_gzrwwhvo2mlq) 16

[1.2.2.5 - EJEMPLO 4](#_8p0j7j5dl9yf) 16

# 1 - DESARROLLO

## 1.1 - SEÑAL BIOLÓGICA - ECG

La señal obtenida de los datos brindados a través del archivo cardio\_03.csv es la siguiente:

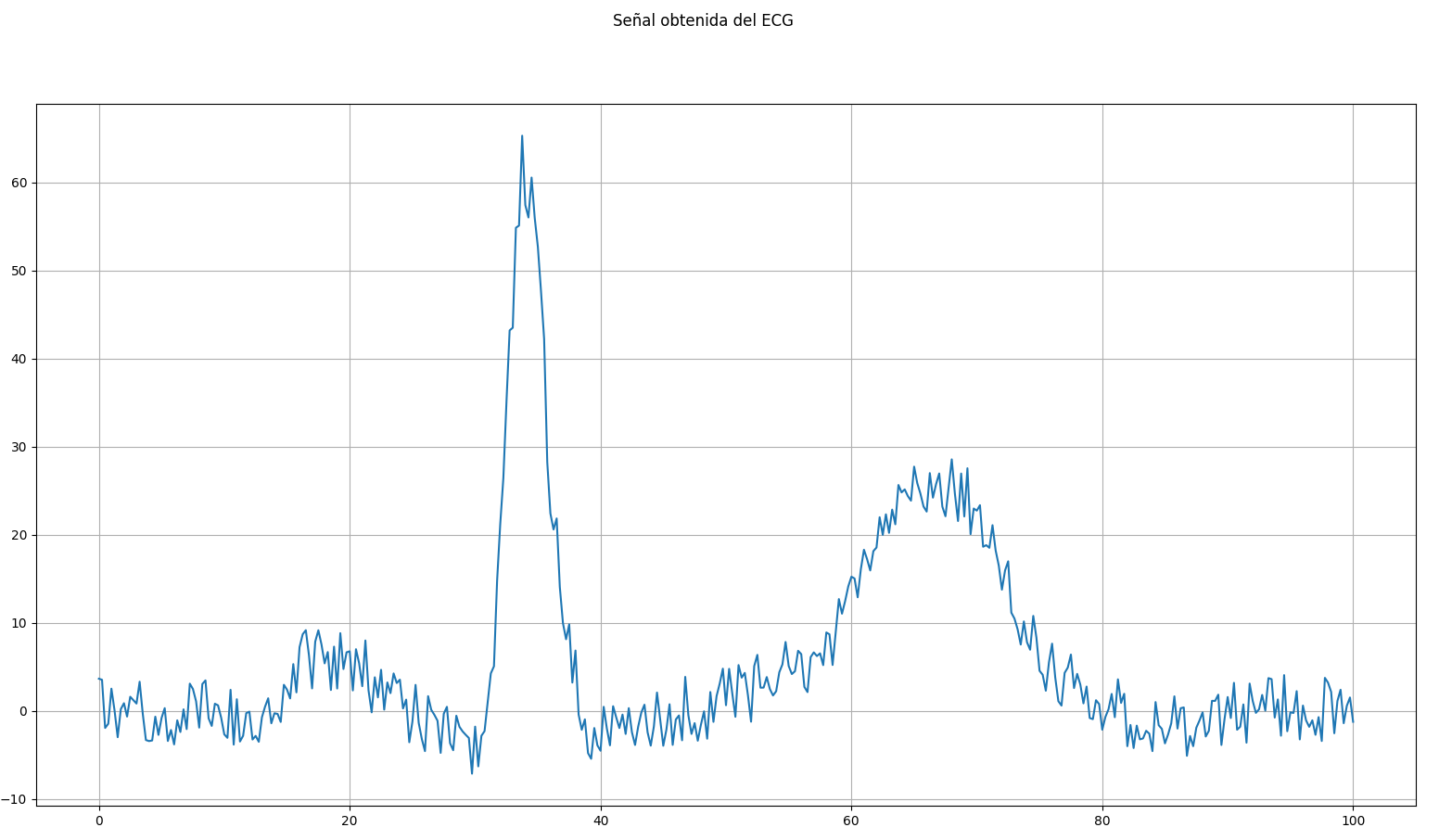


Imagen 1 - Señal dada

### 1.1.1 - APROXIMACIÓN DE LA SEÑAL

Para obtener un modelo que aproxime a toda la señal, se podría aplicar tanto interpolación como mínimos cuadrados. Se evaluaron las siguientes posibilidades:

* Generar un polinomio de newton de grado igual a la cantidad de puntos del archivo menos uno para que pueda pasar pasar por todos los puntos.
* Generar una interpolación por tramos utilizando spline, buscando generar polinomios de menor grado que tiendan a no oscilar tanto pero logrando pasar por todos los puntos.
* Generar un polinomio de un grado suficientemente grande para lograr una buena aproximación, pero encontrando los valores de los coeficientes utilizando mínimos cuadrados.
* Dividir la señal en tramos y utilizar polinomios de menor grado para la aproximación, hallando los coeficientes utilizando mínimos cuadrados condicionados para asegurar la continuidad y suavidad en los extremos de los polinomios.

Pero, al observar que la señal dada posee ruido, es decir, que los datos brindados no son exactos, se optó por el método de mínimos cuadrados condicionado (para lograr que la aproximación sea suave) dividiendo la señal en diferentes tramos.

#### 1.1.1.1 - PLANTEO DE LAS CONDICIONES INICIALES

Se tomaron las siguientes consideraciones al inicio del planteo del método:

* Los tramos iniciales y finales de la función de aproximación serán rectas horizontales y = 0.
* Cualquier recta horizontal colocada en un tramo de la aproximación siempre será de la forma y = 0 o y = a (siendo a un coeficiente).
* Siendo dos tramos contiguos de la aproximación y (siendo sus funciones f1 y f2 respectivamente) cuyo punto de unión posee abscisa , para asegurar la suavidad de la aproximación se debe plantear que:

El planteo de las ecuaciones descritas anteriormente reduce los grados de libertad (eliminan coeficientes) del sistema de ecuaciones formado por todos los polinomios de los tramos de aproximación. Esto no resultó un problema durante el proceso del trabajo práctico, pero se lo aclara en este apartado porque implica la existencia de un “límite inferior” en los grados de los polinomios elegidos a partir del cual se empiezan a tener coeficientes libres para aplicar mínimos cuadrados.

def get\_condiciones\_iniciales(f1,f2,p):

x = sym.symbols('x')

if f1 != 0 and f2!=0:

c1 = f1.subs(x,p) - f2.subs(x,p)

c2 = sym.diff(f1,x).subs(x,p) - sym.diff(f2,x).subs(x,p)

if f1 == 0:

c1 = f2.subs(x,p)

c2 = sym.diff(f2,x).subs(x,p)

if f2 == 0:

c1 = f1.subs(x,p)

c2 = sym.diff(f1,x).subs(x,p)

return [c1, c2]

Una vez planteadas todas las condiciones iniciales con el código anterior utilizamos el método solve de la biblioteca sympy para despejar los coeficientes y reemplazarlas en las funciones originales.

#### 

#### 1.1.1.2 - CALCULO DE MINIMOS CUADRADOS

A continuación, fue necesario armar la matriz y el vector de términos independientes para hallar los valores de los coeficientes por mínimos cuadrados. Para esto, se debió obtener las funciones (phi) vistas en clase, que multiplican a cada uno de los coeficientes en los distintos tramos. La estrategia utilizada fue obtenerlas a partir de evaluar cada coeficiente de la expresión, fijando su valor en uno, y colocando en cero el resto de los coeficientes.

def calcularSistemaMinimosCuadrados2(coef,phi,intervalos\_x,intervalos\_y):

x = sym.symbols('x')

A=[]

b=[]

for c in coef:

fila = []

for c2 in coef:

sum=0.0

for i in range(len(intervalos\_x)):

for j in range(len(intervalos\_x[i])):

try:

sum+=phi[i][c].subs(x,intervalos\_x[i][j])\*phi[i][c2].subs(x,intervalos\_x[i][j])

except KeyError:

pass

fila.append(float(sum))

sum=0.0

for i in range(len(intervalos\_x)):

for j in range(len(intervalos\_x[i])):

try:

sum+=phi[i][c].subs(x,intervalos\_x[i][j])\*intervalos\_y[i][j]

except KeyError:

pass

b.append(float(sum))

A.append(fila)

return A,b

Una vez hallados estos dos elementos, se utilizó el método linalg.solve de numpy para resolver el sistema.

#### 1.1.1.3 - PROBLEMAS ENFRENTADOS

Uno de los puntos principales que causó problemas fue el hecho de que al estar la función definida a tramos, las funciones phi que multiplicaban a los coeficientes también lo están. Por esto, se evaluaron dos posibilidades inicialmente para saber cuál función utilizar a la hora del cálculo:

* Hallar la matriz de mínimos cuadrados y el vector de términos independientes para cada uno de los tramos, despejar los coeficientes y compararlos entre sí.
* Hallar la matriz y el vector para toda la señal pero utilizando el phi correspondiente al tramo al que cada punto pertenece.

En retrospectiva, la primera opción es obviamente ilógica porque los coeficientes obtenidos en cada instancia aproximan lo mejor posible al tramo en el que se calculó pero no aseguran nada respecto al resto de la señal.

Otro de los problemas, fue el hecho de decidir qué hacer si luego de resolver las condiciones iniciales del sistema, en las funciones nos quedaban expresiones no multiplicadas por un coeficiente. Por lo visto en clase, se decidió que sería necesario restar dichas expresiones a las ordenadas de la función. Esto no fue necesario realizarlo por lo que se desarrolla en el siguiente punto.

El algoritmo generado funciona correctamente, sin embargo, posee una complejidad bastante alta lo cual ocasiona que se tengan tiempos de procesamientos largos al aumentar mucho la cantidad de tramos. Se podrían realizar mejoras para disminuir la complejidad, pero no se considero necesario en este caso para ilustrar el entendimiento del concepto.

#### 1.1.1.4 - TEORIAS FORMULADAS

Algo interesante observado durante la experimentación, es que con las consideraciones planteadas al inicio y planteando distintos ejemplos, en ninguno de ellos se dio un caso en que una expresión no quedará multiplicada por un coeficiente.

Esto se tomó como una propiedad válida durante la codificación, logrando que podamos evitar preocuparnos por el segundo problema descrito en el inciso anterior. La propiedad requiere el ejercicio de generar una demostración formal, pero se deja para la posteridad interesada a futuro.

#### 1.1.1.5 - APROXIMACIONES LOGRADAS

Para la primera aproximación, se tomaron las siguientes funciones y tramos:

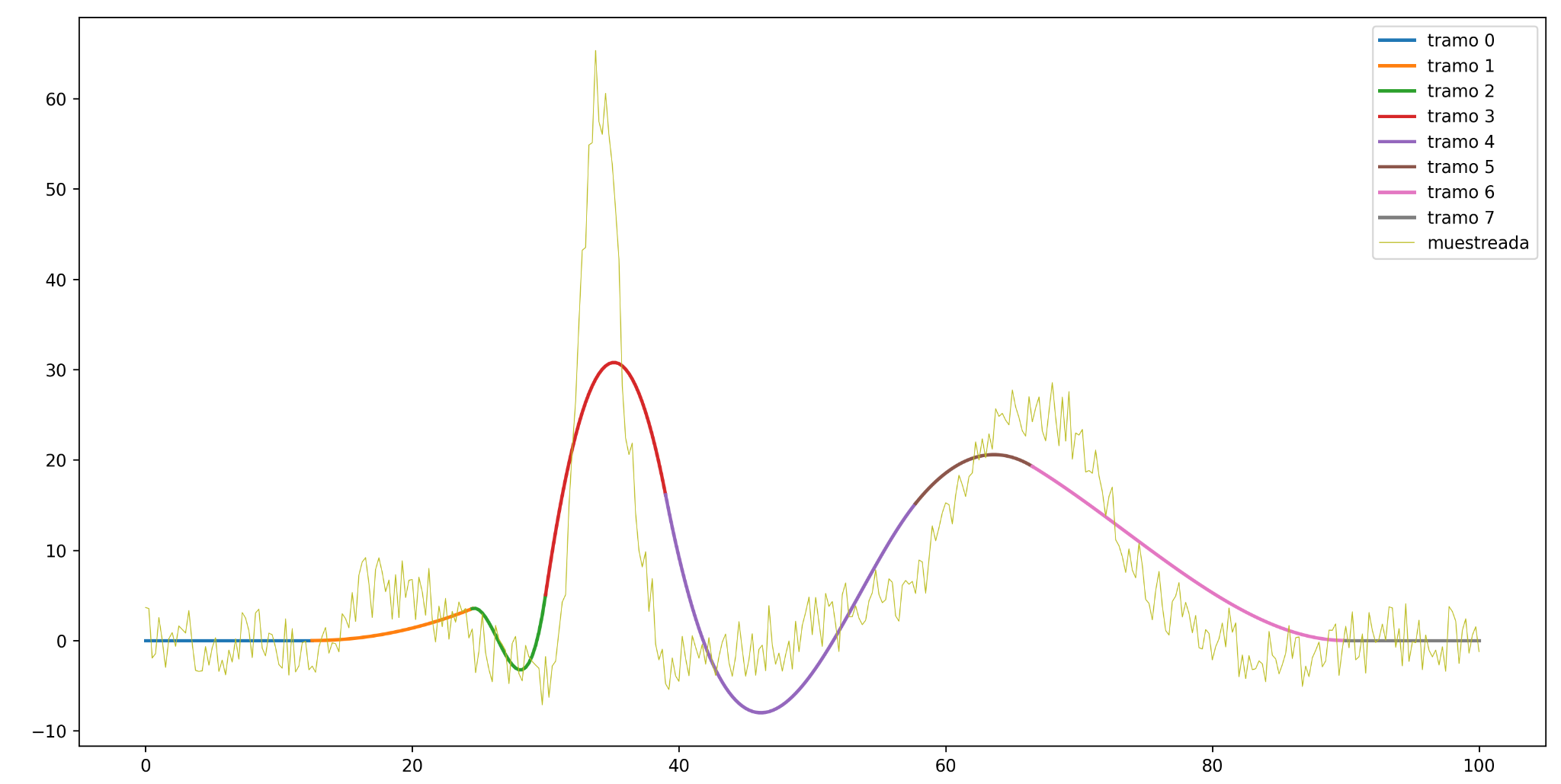


Imagen 1 - Comparación entre la Primera Aproximación y la Señal Real

Para la segunda aproximación, se tomaron las siguientes funciones y tramos:

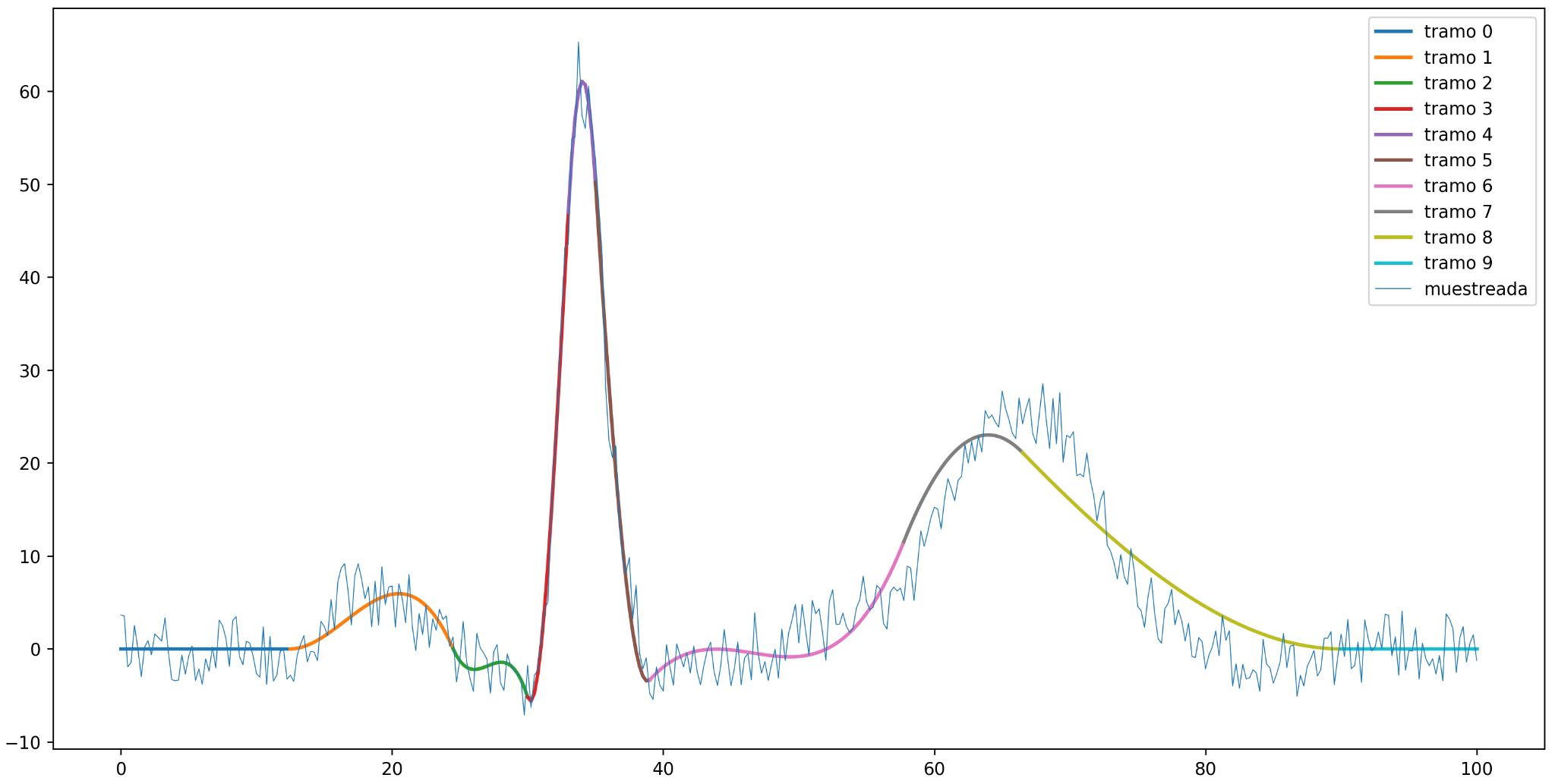


Imagen 2 - Comparación entre la Segunda Aproximación y la Señal Real

Para la segunda aproximación, se tomaron las siguientes funciones y tramos:

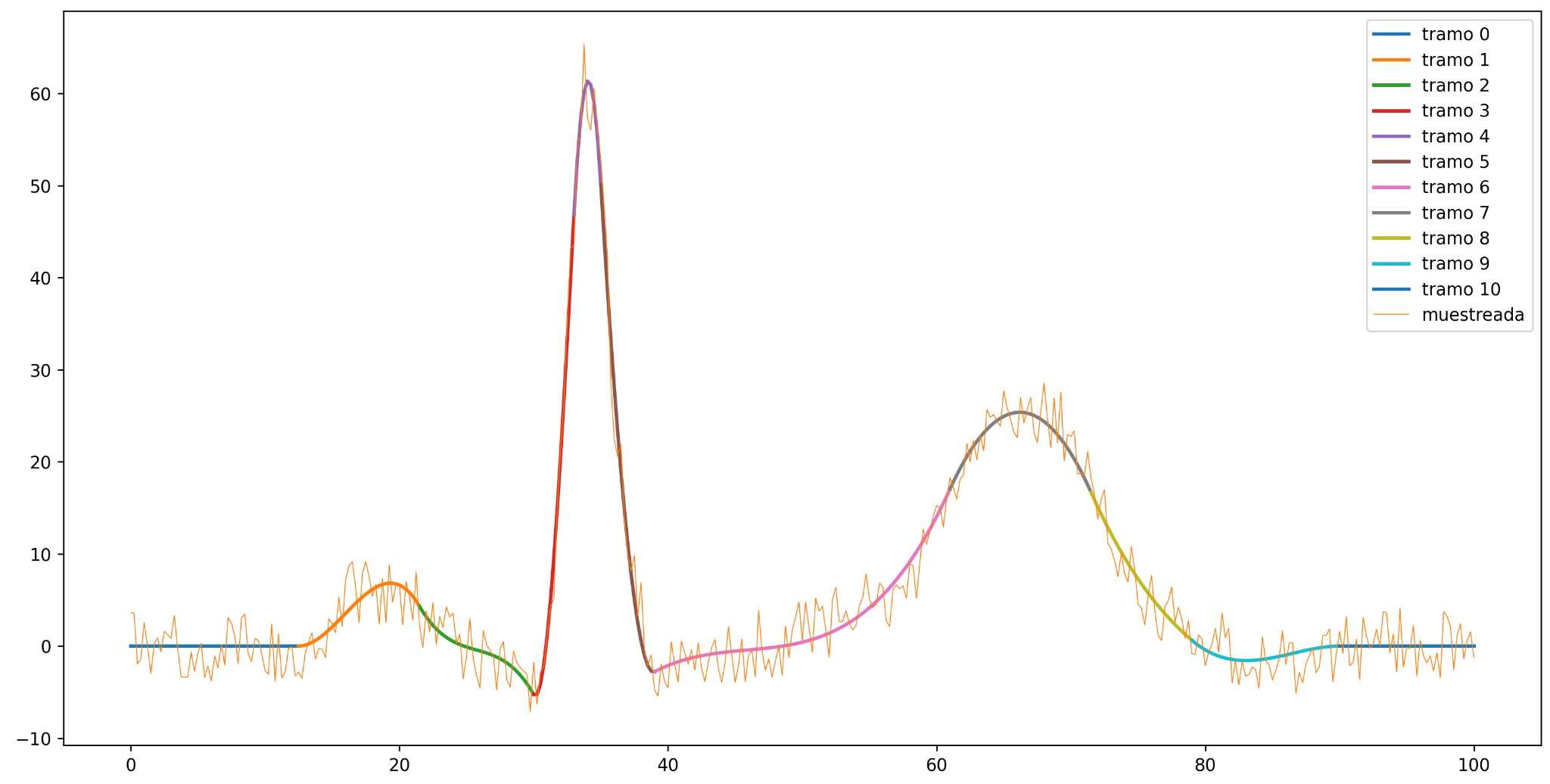


Imagen 3 - Comparación entre la Tercera Aproximación y la Señal Real

Esta última aproximación parece ser un buen ajuste de la señal, con lo cual, es la que utilizaremos de ahora en adelante para el resto de incisos.

### 1.1.2 - DETERMINACIÓN DE LA DURACIÓN Y AMPLITUD DE CADA ONDA

La duración de cada onda se determinó observando gráficamente la aproximación obtenida. Aprovechando que los datos brindados se encuentran en función del tiempo, la duración de cada onda se pudo calcular con la diferencia entre el límite superior y el límite inferior del intervalo en el cual se encuentra definida la onda, obteniéndose los siguientes resultados:

* **Onda P:** se encuentra en el intervalo ***[12.5 ; 25]***, teniendo una duración de **12.5** unidades de tiempo.
* **Onda QRS:** se encuentra en el intervalo ***[25 ; 40.5]***, teniendo una duración de **15.5** unidades de tiempo.
* **Onda T:** se encuentra en el intervalo ***[49 ; 85.5]***, teniendo una duración de **36.5** unidades de tiempo.

Para las amplitudes, se calcula el máximo y el mínimo de cada intervalo, seleccionando gráficamente el tramo que contiene al máximo y el tramo que contiene al mínimo para cada onda, y luego se calcula la diferencia entre ellos. En particular para la onda P, se considera 0 como el mínimo ya que gráficamente se logra ver que el mínimo ronda cerca de ese valor:

# el tramo 1 es el que contiene al maximo de la P

max\_p = tramos[1].subs(x,sym.solvers.solve(sym.diff(tramos[1],x),x)[1])

print('Maximo de la onda P =', max\_p)

print('Minimo de la onda P =',0)

print('Amplitud de la onda P =',max\_p)

# el tramo 3 es el que contiene al mínimo de la QRS

min\_qrs = tramos[3].subs(x,sym.solvers.solve(sym.diff(tramos[3],x),x)[0])

print('\nMinimo de la onda QRS =', min\_qrs)

# el tramo 4 es el que contiene al pico de la QRS

max\_qrs = tramos[4].subs(x,sym.solvers.solve(sym.diff(tramos[4],x),x)[0])

print('Maximo de la onda QRS =', max\_qrs)

print('Amplitud de la onda QRS =',max\_qrs-min\_qrs)

# el tramo 9 es el que contiene al mínimo de la T

min\_t = tramos[9].subs(x,sym.solvers.solve(sym.diff(tramos[9],x),x)[0])

print('\nMinimo de la onda T =', min\_t)

# el tramo 7 es el que contiene al pico de la T

max\_t = tramos[7].subs(x,sym.solvers.solve(sym.diff(tramos[7],x),x)[0])

print('Maximo de la onda T =', max\_t)

print('Amplitud de la onda T =',max\_t-min\_t)

**Onda P:**

* Máximo de la onda **P** = 6.83571208146327
* Mínimo de la onda **P** = 0
* Amplitud de la onda **P** = 6.83571208146327

**Onda QRS:**

* Mínimo de la onda **QRS** = -5.40581945965823
* Máximo de la onda **QRS** = 61.3819526968236
* Amplitud de la onda **QRS** = 66.7877721564819

**Onda T:**

* Mínimo de la onda **T** = -1.57245088856143
* Máximo de la onda **T** = 25.3752634365201
* Amplitud de la onda **T** = 26.9477143250815

### 1.1.3 - ANÁLISIS DEL AJUSTE LOGRADO

En principio se planteó el cálculo de la norma 2 para analizar el ajuste aplicando el método calcularCalidadAjuste1(), la cual dió un valor de 45.28777. Este valor resulta ser poco representativo, ya que por la manera en que se calcula la norma 2, el aumentar la cantidad de puntos que intervienen en el cálculo producirá que el resultado será cada vez más grande.

def calcularCalidadAjuste1(ecd\_x,ecd\_y,puntos,tramos):

x = sym.symbols('x')

t = 0

sum = 0

for i in range(len(ecd\_x)):

if t<len(puntos) and ecd\_x[i] == puntos[t]:

t+=1

if isinstance(tramos[t],int):

sum+=ecd\_y[i]\*\*2

else:

sum+= (ecd\_y[i] - tramos[t].subs(x,ecd\_x[i]))\*\*2

return sum\*\*(1/2)

Por este motivo, se optó por el cálculo de la norma infinito con el método calcularCalidadAjuste2():

def calcularCalidadAjuste2(ecd\_x,ecd\_y,puntos,tramos):

x = sym.symbols('x')

t = 0

val = []

for i in range(len(ecd\_x)):

if t<len(puntos) and ecd\_x[i] == puntos[t]:

t+=1

if isinstance(tramos[t],int):

val.append(abs(ecd\_y[i]))

else:

val.append(abs(ecd\_y[i] - tramos[t].subs(x,ecd\_x[i])))

return max(val)

Como resultado, se obtuvo un valor de 6.34139. Esto se puede interpretar como que la diferencia en valor absoluto entre la aproximación y la señal dada como máximo será de ese valor. Este valor puede parecer muy alto por la presencia del ruido, pero gráficamente se observa que el la aproximación ajusta bastante bien:

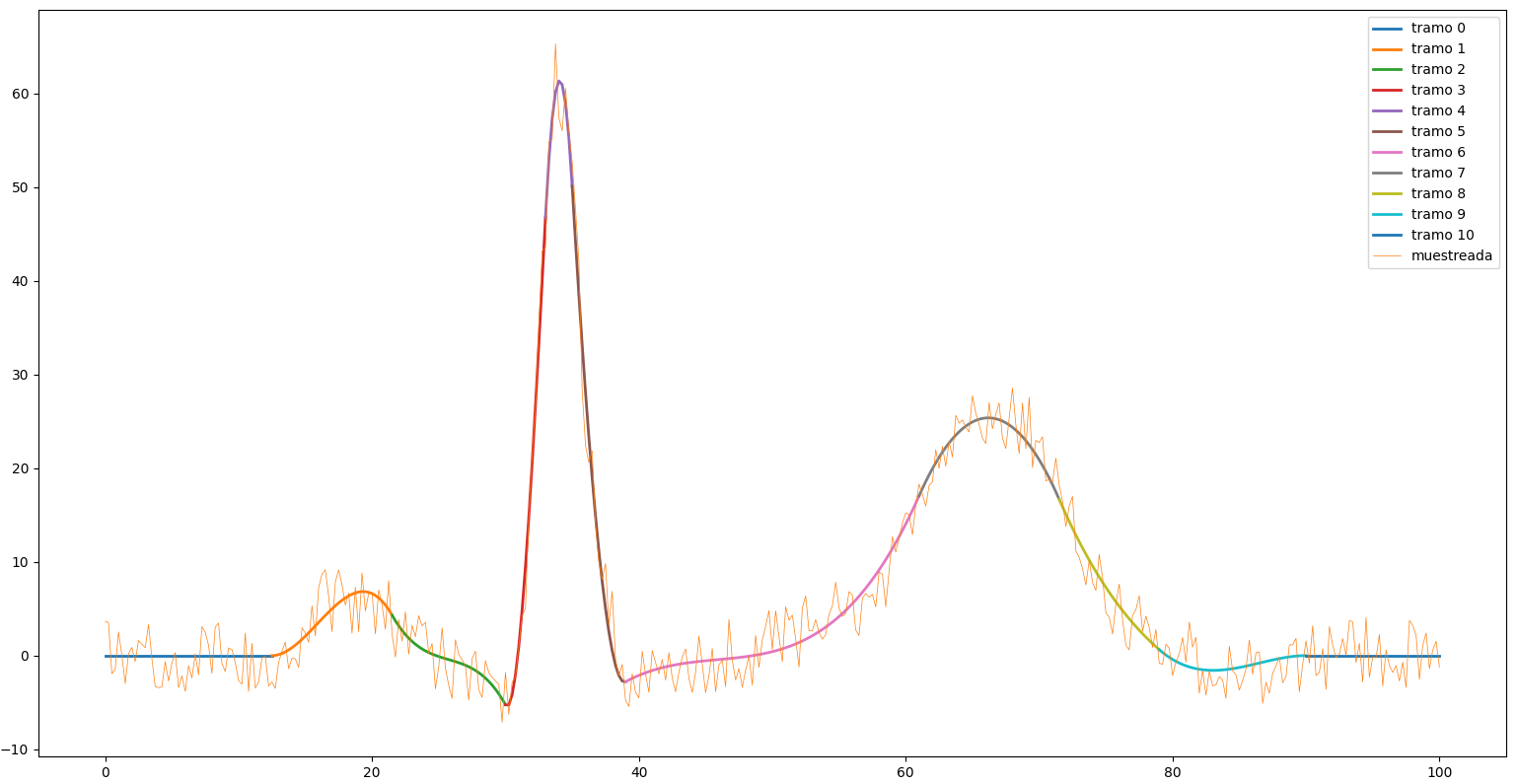


Imagen 4 - La señal y su aproximación

## 1.2 - POLINOMIO DE NEWTON

### 1.2.1 - MODIFICACIÓN DE LA TABLA DE DIFERENCIAS DIVIDIDAS

Para lograr que el polinomio de newton interpole los valores de las derivadas primera, segunda y tercera un nodo además de la ordenada, se desarrolló el siguiente ejemplo:

* Suponga que el polinomio debe contener los siguientes puntos:



* Y suponga que se deben cumplir las siguientes condiciones:

Entonces, para lograr condicionar el polinomio de newton se debe repetir el punto correspondiente a la abscisa 3 dos veces y a partir de esto generar la tabla de diferencias divididas. En la siguiente imagen se muestra una porción de la tabla, con rojo indicando donde se debe establecer el valor de la primera derivada y con verde el de la segunda.

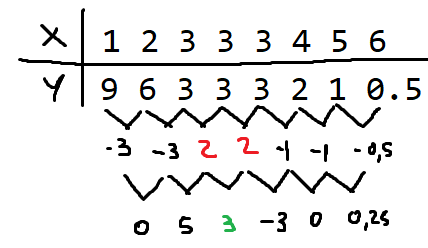


Imagen 5 - Porción de la tabla de diferencias divididas

La primer fila de la tabla de diferencias divididas (que comienza con -3), cada elemento se calcula de la forma . Recordando la definición de derivada tenemos que:

Al estar repitiendo el punto (3,3) se logra que el denominador sea igual a cero y, al establecer un valor en esta posición, se logra condicionar la primera derivada en el punto repetido.

Para la segunda derivada, como se necesita que el denominador de las diferencias divididas sea cero en el cálculo de la segunda fila, el punto debe aparecer 3 veces en la tabla.

El mismo pensamiento se puede extender para la tercera derivada en el punto, necesitando que aparezca el punto 4 veces en la tabla.

En este momento, se destaca el inconveniente de que no es posible para este método condicionar la n-ésima derivada si no se condicionan todas las derivadas anteriores.

Se codificó el ejemplo, y se descubrió que el acondicionamiento de la primera derivada en el punto dado funcionaba, pero con las siguientes derivadas se obtenían otros valores. Esto se debe a que es necesario dividir la imagen de la n-ésima derivada por el factorial de n antes de colocarla en la tabla de diferencias divididas. Entonces, la tabla debería resultar de la siguiente manera:

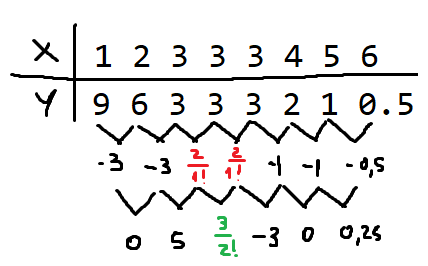


Imagen 6 - Porción de la tabla de diferencias divididas corregida

Se cree que la justificación de la división por el factorial es porque el polinomio de newton que se obtiene al aplicar el método es muy similar al polinomio de taylor, entonces se debe dividir por el factorial para compensar los factores que bajan multiplicando del exponente al derivar el polinomio.

'''

Los argumentos son:

\*El vector de coordenadas x de los puntos por donde tiene que pasar el polinomio [x0,x1,x2,..,xn]

\*El vector de coordenadas y de los puntos por donde tiene que pasar el polinomio [y0,y1,y2,...yn]

\*Un vector con dos vectores:

-El primero contiene las coordenadas x de la primera derivada

-El segundo los valores de las derivadas en las x dadas

[[x1,x3,x8],[13,4,1]] significa que f'(x1)=13, f'(x3)=4 y f'(x8)=1

\*Un vector con dos vectores:

-El primero contiene las coordenadas x de la segunda derivada

-El segundo los valores de las derivadas en las x dadas

\*Un vector con dos vectores:

-El primero contiene las coordenadas x de la tercera derivada

-El segundo los valores de las derivadas en las x dadas

\*Si no se quiere condicionar alguna derivada, directamente hay que pasarle los vectores vacios [[],[]]

'''

def metodo\_newton\_final(x,y,deriv1,deriv2,deriv3):

x,y,deriv1,deriv2,deriv3 = todoAFloat(x,y,deriv1,deriv2,deriv3)

for d in [deriv1, deriv2, deriv3]:

for i in range(0,len(d[0])):

pos = x.index(d[0][i])

x.insert(pos,d[0][i])

y.insert(pos,y[pos])

ans = [y]

for v in range(0,len(x)-1):

ar = ans[v]

aux = []

i = 0

j = v+1

while(j < len(x)):

if x[j]!=x[i]:

aux.append((ar[i+1]-ar[i])/(x[j]-x[i]))

else: #hay que setear los valores dados para las derivadas

if v==0:

aux.append(deriv1[1][deriv1[0].index(x[i])])

if v==1:

aux.append(deriv2[1][deriv2[0].index(x[i])]/2)

if v==2:

aux.append(deriv3[1][deriv3[0].index(x[i])]/6)

j+=1

i+=1

ans.append(aux)

imprimirPolinomio(ans,x)

El método todoAFloat() transforma todos los valores que contiene cada vector al tipo de dato float. Esto se tuvo que realizar ya que la comparación x[j]!=x[i] estaba teniendo problemas cuando se comparaban diferentes tipos de datos.

### 1.2.2 - VERIFICACIONES CON EJEMPLOS

#### 1.2.2.1 - EJEMPLO 0

* Puntos por los que tiene que pasar el polinomio: (1,2), (2,4), (3,8), (4,16), (5,32)
* f ’(1) = 3, f ’(3) = 5, f ’(5) = -20
* f’’(1) = -3, f’’(3) = 70
* f’’’(1) = 50, f’’’(3) = 2

Polinomio obtenido:

2.0+3.0(x-1.0)+-0.5(x-1.0)(x-1.0)+8.333333333333334(x-1.0)(x-1.0)(x-1.0)+-8.833333333333334(x-1.0)(x-1.0)(x-1.0)(x-1.0)+4.791666666666667(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)+-2.7083333333333335(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)+3.697916666666667(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)(x-3.0)+-7.28125(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)(x-3.0)(x-3.0)+4.58641975308642(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)(x-3.0)(x-3.0)(x-3.0)+-1.6884705343364197(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)(x-3.0)(x-3.0)(x-3.0)(x-4.0)+0.5579095534336419(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-2.0)(x-3.0)(x-3.0)(x-3.0)(x-3.0)(x-4.0)(x-5.0)

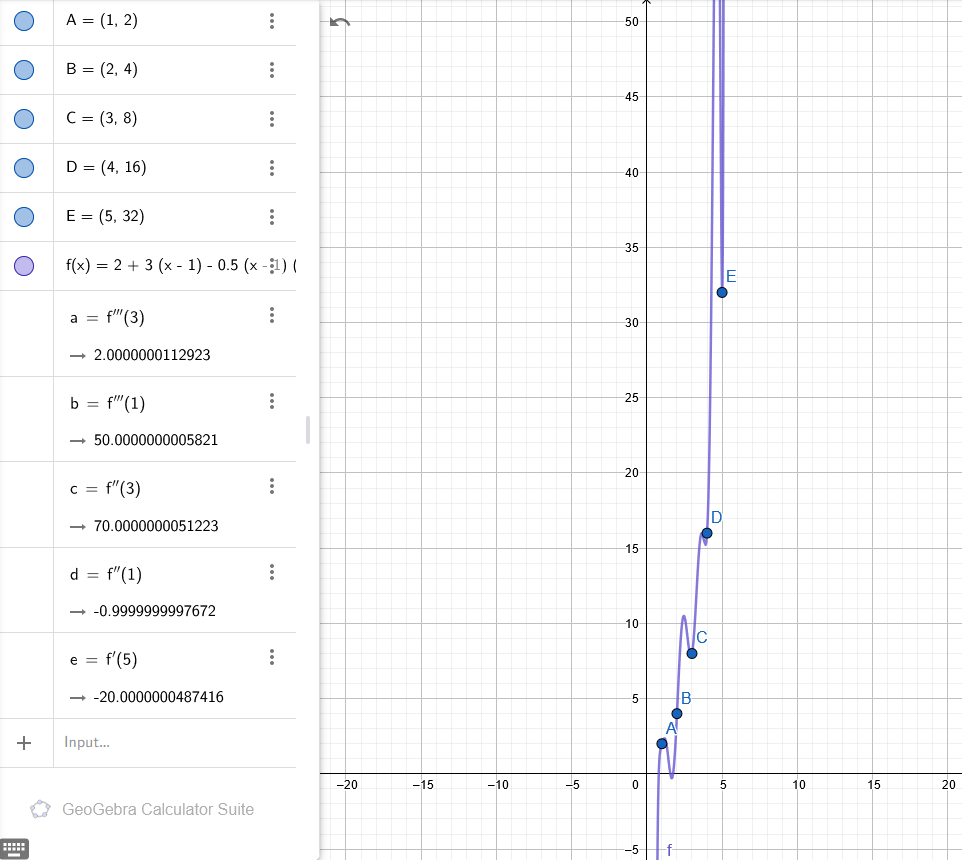
****

Imagen 7 - Ejemplo 0

#### 1.2.2.2 - EJEMPLO 1

* Puntos por los que tiene que pasar el polinomio: (-5,0), (-2,7), (3,-5), (8,-7), (16,-9)
* f ’(-5) = 2, f ’(3) = -2
* f’’(3) = -5
* f’’’(3) = 10

Polinomio obtenido:

0.0+2.0(x--5.0)+0.11111111111111116(x--5.0)(x--5.0)+-0.08784722222222223(x--5.0)(x--5.0)(x--2.0)+0.021475694444444443(x--5.0)(x--5.0)(x--2.0)(x-3.0)+-0.01205881076388889(x--5.0)(x--5.0)(x--2.0)(x-3.0)(x-3.0)+0.009499978298611112(x--5.0)(x--5.0)(x--2.0)(x-3.0)(x-3.0)(x-3.0)+-0.0014979136104947405(x--5.0)(x--5.0)(x--2.0)(x-3.0)(x-3.0)(x-3.0)(x-3.0)+0.00010391995592411018(x--5.0)(x--5.0)(x--2.0)(x-3.0)(x-3.0)(x-3.0)(x-3.0)(x-8.0)

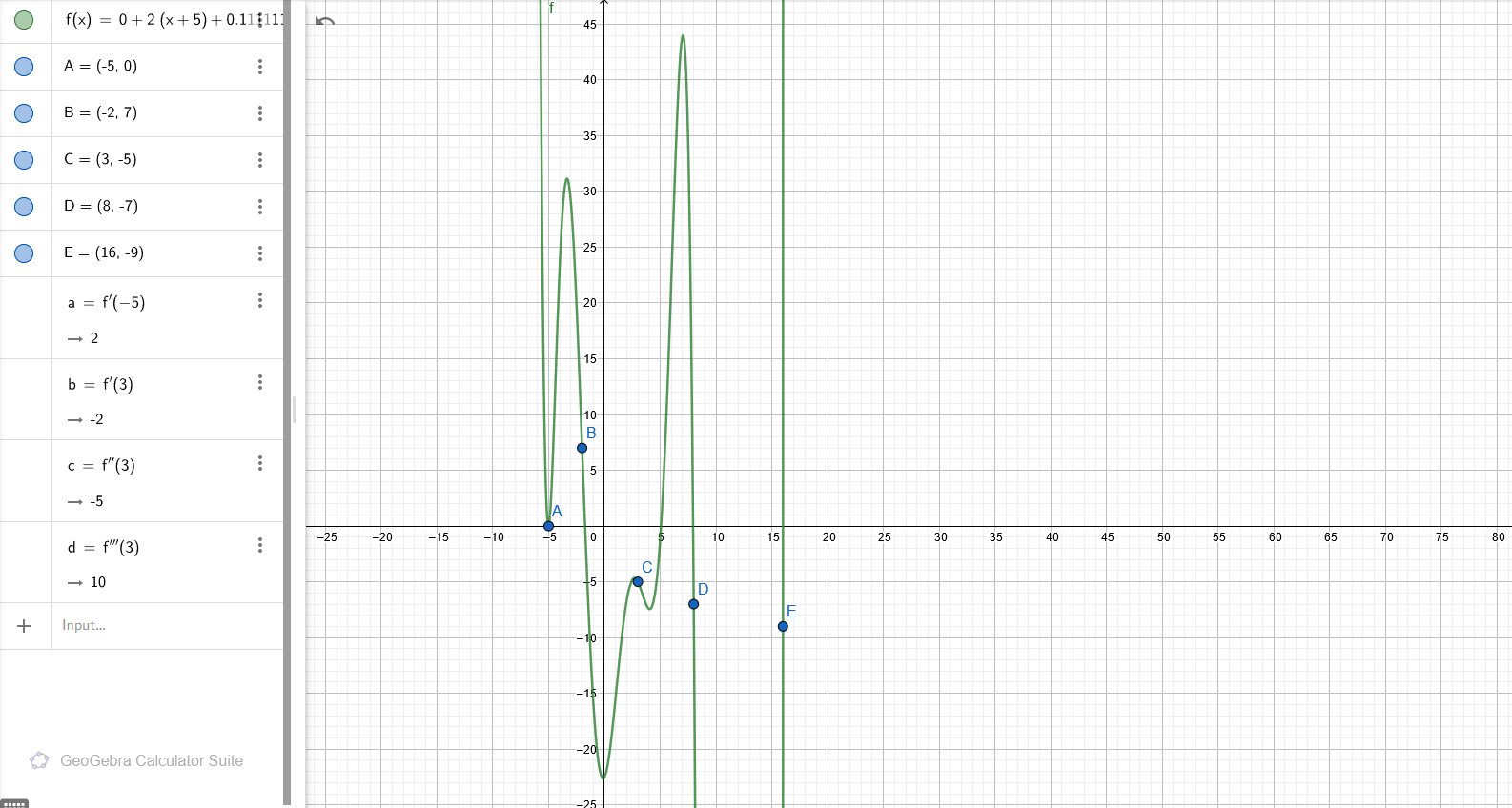
****

Imagen 8 - Ejemplo 1

#### 1.2.2.3 - EJEMPLO 2

* Puntos por los que tiene que pasar el polinomio: (5,15), (10,10), (30,15), (20,5)
* f ’(5) = -2, f ’(10) = -2
* f’’(5) = -1

Polinomio obtenido:

15.0+-2.0(x-5.0)+-0.5(x-5.0)(x-5.0)+0.13999999999999999(x-5.0)(x-5.0)(x-5.0)+-0.044(x-5.0)(x-5.0)(x-5.0)(x-10.0)+0.001908(x-5.0)(x-5.0)(x-5.0)(x-10.0)(x-10.0)+-0.00014845925925925925(x-5.0)(x-5.0)(x-5.0)(x-10.0)(x-10.0)(x-30.0)

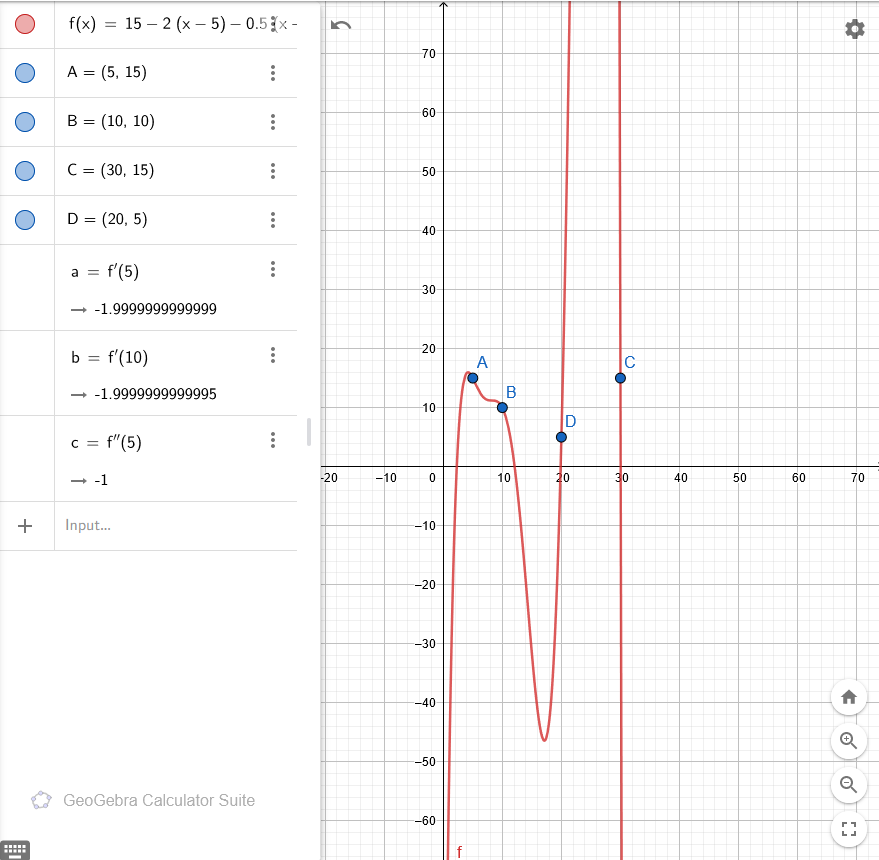
****

Imagen 9 - Ejemplo 2

#### 1.2.2.4 - EJEMPLO 3

* Puntos por los que tiene que pasar el polinomio: (5,5), (-4,2), (1,1), (5,-3)
* f ’(-5) = -4, f ’(5) = -3, f ’(1) = 1
* f’’(1) = 1, f’’(5) = 50
* f’’’(1) = -5

Polinomio obtenido:

5.0+-4.0(x--5.0)+1.0(x--5.0)(x--5.0)+-0.0888888888888889(x--5.0)(x--5.0)(x--4.0)+0.008518518518518522(x--5.0)(x--5.0)(x--4.0)(x-1.0)+0.0010740740740740734(x--5.0)(x--5.0)(x--4.0)(x-1.0)(x-1.0)+-0.0055131687242798355(x--5.0)(x--5.0)(x--4.0)(x-1.0)(x-1.0)(x-1.0)+0.0012301440329218107(x--5.0)(x--5.0)(x--4.0)(x-1.0)(x-1.0)(x-1.0)(x-1.0)+-0.0002539094650205761(x--5.0)(x--5.0)(x--4.0)(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-5.0)+0.0001576056670096022(x--5.0)(x--5.0)(x--4.0)(x-1.0)(x-1.0)(x-1.0)(x-1.0)(x-5.0)(x-5.0)

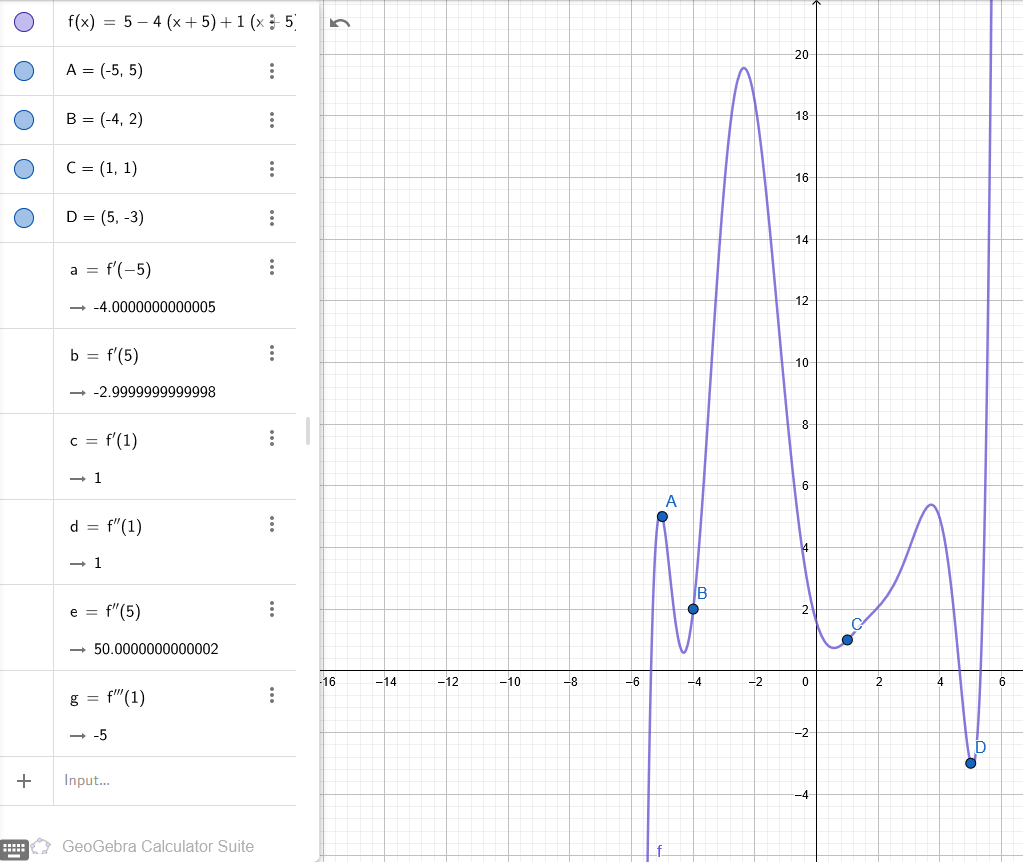
****

Imagen 10 - Ejemplo 3

#### 1.2.2.5 - EJEMPLO 4

* Puntos por los que tiene que pasar el polinomio: (-3,3), (-2,-2), (-1,1), (0,0), (1,-1)
* f ’(-3) = 1, f ’(-2) = 1, f ’(-1) = 1
* f ’’(-3) = 1, f ‘’(-2) = 1, f ‘’(-1) = 1
* f ’’’(-3) = 1, f ’’’(-2) = 1, f ‘’’(-1) = 1

Polinomio obtenido:

3.0+1.0(x--3.0)+0.5(x--3.0)(x--3.0)+0.16666666666666666(x--3.0)(x--3.0)(x--3.0)+-6.666666666666667(x--3.0)(x--3.0)(x--3.0)(x--3.0)+25.166666666666668(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)+-61.16666666666667(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)+120.33333333333334(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)+-78.125(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)+49.333333333333336(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)(x--1.0)+-29.125000000000004(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)(x--1.0)(x--1.0)+13.312500000000004(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)(x--1.0)(x--1.0)(x--1.0)+-3.011574074074075(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)(x--1.0)(x--1.0)(x--1.0)(x--1.0)+0.21735146604938282(x--3.0)(x--3.0)(x--3.0)(x--3.0)(x--2.0)(x--2.0)(x--2.0)(x--2.0)(x--1.0)(x--1.0)(x--1.0)(x--1.0)(x-0.0)

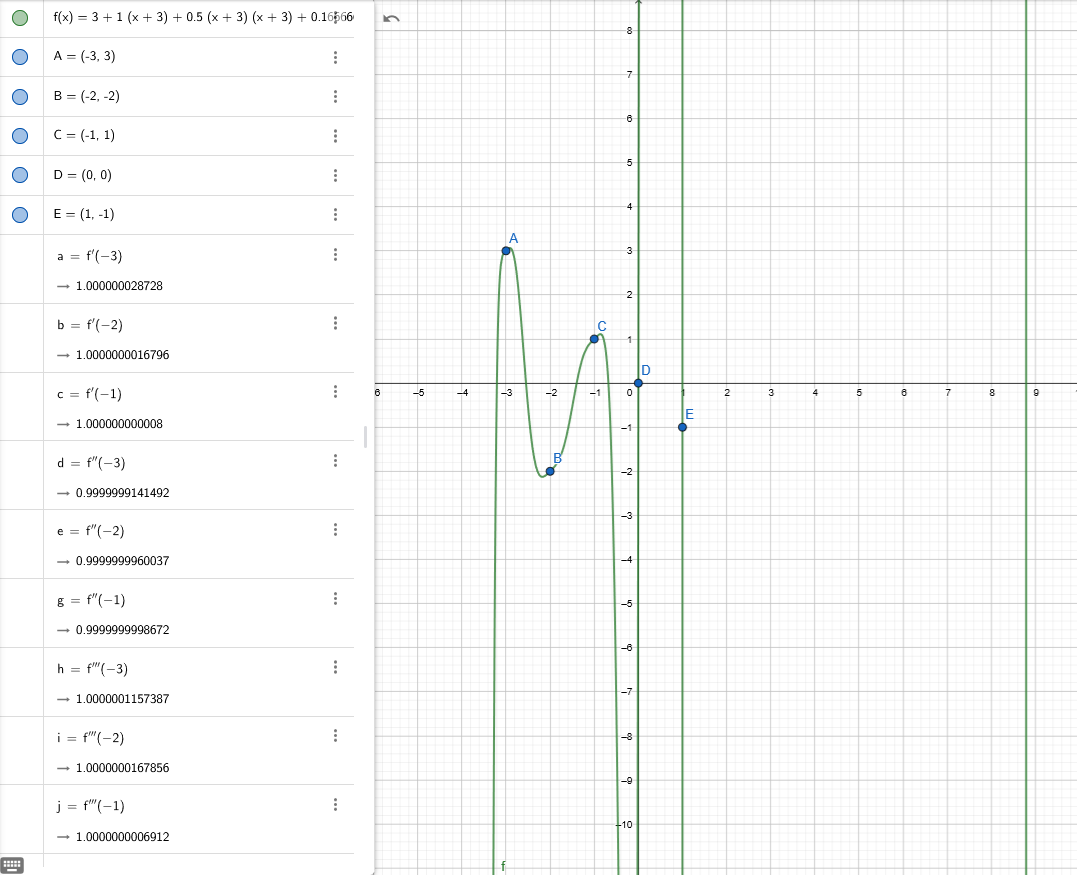
****

Imagen 11 - Ejemplo 4