PROYECTO PARA ANALISIS NUMERICO O METODOS NUMERICOS

SPLINES CUBICOS NATURALES PARA ANIMACION

Escrito, creado, hecho y realizado por Jhulen Mallo, estudiante de Informática en la Facultad de Ciencias Puras y Naturales de la Universidad Mayor de San Andrés.

Propósito: Brindar asistencia a estudiantes que enfrentan retos de gestión del tiempo y concentración, facilitando el desarrollo y ejecución de proyectos. Esta ayuda está destinada a permitirles demostrar las competencias adquiridas durante el semestre en la asignatura de Análisis Numérico.

Introducción a los Splines Naturales

En el campo del análisis numérico y la interpolación de datos, los "splines naturales" son una potente herramienta matemática utilizada para aproximar curvas suaves a través de un conjunto de puntos de datos discretos. Los splines naturales son particularmente valiosos cuando se requiere una representación continua de datos en situaciones donde las funciones subyacentes pueden ser altamente no lineales o irregulares.

La característica distintiva de los splines naturales es su capacidad para proporcionar una interpolación suave y continua mientras garantizan que la función resultante sea lineal en los extremos del dominio de los datos. Esto significa que la curva generada por los splines naturales no solo pasa a través de los puntos de datos conocidos, sino que también es diferenciable en esos puntos, lo que la hace especialmente útil en aplicaciones donde la continuidad y la suavidad son esenciales.

La construcción de splines naturales implica la división del conjunto de datos en segmentos más pequeños y la interpolación de cada segmento con una función polinómica de grado bajo. Estos segmentos polinómicos se unen de manera que la función resultante sea continua y cumpla con ciertas condiciones en los extremos del dominio. La elección de splines naturales en lugar de otros métodos de interpolación se basa en su capacidad para minimizar la oscilación no deseada y proporcionar resultados más estables y realistas.

Caracteristicas

Un spline natural es un tipo de spline que se utiliza en interpolación y aproximación de datos. Se caracteriza por ser una curva suave que pasa a través de un conjunto de puntos de datos discretos y cumple con dos condiciones clave en los extremos del dominio de los datos:

Condición de Interpolación: El spline natural pasa a través de todos los puntos de datos conocidos, lo que significa que la curva interpolante coincide con los valores dados en los puntos dados.

Condición de Curvatura Cero (Segunda Derivada Cero): La segunda derivada de la función spline es igual a cero en los extremos del dominio de los datos. Esto garantiza que la curva sea lineal en los extremos, lo que evita oscilaciones no deseadas.

Un spline natural se construye dividiendo el conjunto de datos en segmentos más pequeños y aproximando cada segmento con una función polinómica de grado bajo. Estos polinomios se ensamblan

de manera que la función resultante sea continua en todos los puntos de división y cumpla con las condiciones de interpolación y curvatura cero en los extremos.

Para construir un spline natural, se pueden utilizar varios algoritmos y métodos. Algunos de los algoritmos comunes para calcular splines naturales incluyen:

Método de Diferencias Divididas: Este método utiliza diferencias divididas para calcular los coeficientes de los polinomios en cada segmento y garantizar la continuidad.

Método de Resolución de Sistemas Lineales: Se puede formular un sistema de ecuaciones lineales a partir de las condiciones de interpolación y curvatura cero y luego resolverlo para encontrar los coeficientes del spline natural.

Método de la Matriz Tridiagonal: Este método involucra la construcción de una matriz tridiagonal y la resolución eficiente de un sistema de ecuaciones lineales para determinar los coeficientes del spline.

Método de Spline Cúbico Natural Básico: Es un enfoque más simple que utiliza una fórmula directa para calcular los coeficientes de los splines naturales sin necesidad de resolver un sistema de ecuaciones lineales.

Aplicaciones o proyectos para Analisis Numerico

El mundo de la animación y la tecnología gráfica ha evolucionado enormemente en las últimas décadas, y una de las herramientas matemáticas fundamentales que ha impulsado esta evolución son los "splines naturales". Estas curvas suaves, que combinan la elegancia matemática con la flexibilidad creativa, se han convertido en un componente esencial para una variedad de aplicaciones que van más allá de la mera representación de datos.

En el ámbito de los videojuegos, los splines naturales se han convertido en la clave para crear experiencias de juego más inmersivas y visualmente impresionantes. Permiten la animación de personajes, cámaras y objetos de manera suave y realista, lo que mejora la jugabilidad y la calidad de los gráficos.

La industria del cine y la animación también ha adoptado ampliamente los splines naturales para dar vida a personajes y objetos en la pantalla grande. Estos splines permiten a los animadores crear movimientos fluidos y expresivos que capturan la esencia de sus personajes y escenarios.

En el campo de la realidad virtual (RV), los splines naturales se utilizan para simular movimientos y experiencias de usuario más naturales y convincentes. Esto incluye la animación de personajes y la interacción con el entorno virtual.

Además, los splines naturales tienen aplicaciones en aplicaciones web y móviles, mejorando la experiencia del usuario al crear animaciones interactivas y transiciones suaves entre elementos gráficos.

En la industria y la robótica, los splines naturales se emplean para controlar el movimiento de robots y drones en entornos de fabricación, logística y exploración, lo que impulsa la automatización y la eficiencia en una variedad de aplicaciones.
Animación de Personajes 2D: Desarrolla un sistema de animación de personajes 2D que utilice splines

naturales para controlar el movimiento suave de los personajes. Puedes aplicarlos para definir trayectorias de movimiento, transiciones entre animaciones y deformaciones de personajes en tiempo real.

Animación de Cámaras en Videojuegos: Implementa un sistema de animación de cámaras en videojuegos que utilice splines naturales para crear movimientos de cámara cinematográficos y suaves durante el juego. Esto puede mejorar la experiencia del jugador y hacer que el juego sea más inmersivo.

Animación de Trayectorias de Vuelo de Drones: Desarrolla un sistema de control de vuelo para drones que utilice splines naturales para definir trayectorias de vuelo suaves y precisas. Esto puede ser útil en aplicaciones de fotografía aérea, mapeo y seguimiento de objetos en movimiento.

Animación de Elementos Gráficos en Aplicaciones Web: Crea animaciones interactivas en aplicaciones web utilizando splines naturales para controlar la transición suave entre estados y elementos gráficos. Esto puede mejorar la experiencia del usuario y la usabilidad de la aplicación.

Animación de Movimiento de Partículas en Gráficos por Computadora: Implementa splines naturales para controlar el movimiento de partículas en gráficos por computadora, como efectos de humo, fuego o fluidos. Esto puede dar como resultado simulaciones más realistas y fluidas.

Animación de Personajes en Películas y Películas Animadas: En la industria del cine y la animación, los splines naturales se utilizan para animar personajes y objetos de manera realista. Puedes investigar cómo se aplican en la producción de películas animadas y explorar cómo se utilizan para lograr movimientos suaves y expresivos.

Simulación de Movimiento de Personajes en Realidad Virtual (RV): Utiliza splines naturales para simular el movimiento y la interacción de personajes en entornos de realidad virtual. Esto puede ser valioso para la creación de experiencias inmersivas en RV.

Animación de Paisajes y Terrenos en Videojuegos: Emplea splines naturales para generar animaciones de paisajes y terrenos en videojuegos. Esto puede utilizarse para simular efectos naturales como el flujo del agua, la deformación del terreno y la vegetación.

Animación de Movimiento de Robots y Drones en la Industria: Investigar cómo los splines naturales se aplican en la industria para controlar el movimiento de robots y drones en aplicaciones como la logística, la fabricación y la exploración.

Aplicación Educativa de Animaciones con Splines Naturales: Desarrolla una aplicación educativa que permita a los estudiantes experimentar y comprender cómo funcionan los splines naturales en animaciones. Esto puede ser útil para la enseñanza de conceptos matemáticos y de programación relacionados con los splines.

Explicacion del proyecto

El proyecto se centra en la creación de animaciones utilizando splines cúbicos naturales. Cuando exploramos técnicas de interpolación como Lagrange o diferencias divididas, notamos que a menudo nos enfrentamos a la limitación de tener solo un conjunto discreto de puntos de referencia. Entre estos puntos, surgen puntos intermedios que representan aproximaciones al objetivo de la interpolación.

Imaginemos una gráfica sencilla con solo cuatro puntos; inicialmente, podríamos conectar estos puntos con segmentos de línea recta, lo que resultaría en una representación visual bastante rígida. Sin

embargo, mediante la interpolación y el uso de splines cúbicos naturales, descubrimos que podemos encontrar una serie de puntos adicionales que se ajustan de manera suave y continua a la curva deseada.

Este proceso nos ofrece la oportunidad de lograr que nuestra gráfica sea más atractiva y realista, ya que a medida que incorporamos más puntos intermedios, la curva se vuelve más suave y fluida. De esta manera, podemos crear animaciones que transmiten un movimiento más natural y visualmente agradable, lo que es fundamental en campos como la animación de personajes, los videojuegos y la simulación de movimientos en la realidad virtual.

Ahora, imaginemos que tenemos un conjunto de 8 puntos que intentan replicar la silueta de un círculo. Si realizamos una representación gráfica de estos puntos en Python, MATLAB u Octave, notaremos que al unir estos puntos, no obtenemos un círculo perfecto, sino más bien un polígono de 8 lados. En este punto, se vuelve evidente que, si deseamos que nuestra gráfica se asemeje más a un círculo, necesitamos incorporar más puntos. Sin embargo, la tarea de agregar punto por punto resultaría en un trabajo arduo y propenso a errores.

Entonces, ¿qué podemos hacer para abordar esta complejidad? La interpolación se convierte en nuestra solución clave. A través de la interpolación, podemos generar puntos adicionales dentro del intervalo de estos mismos puntos de referencia. Esto es precisamente lo que necesitamos para mejorar la precisión de nuestra representación. Pero aquí es donde entra en juego una herramienta aún más poderosa: los splines cúbicos naturales.

Los splines cúbicos naturales no solo añaden puntos adicionales, sino que también hacen que estos puntos no sean simplemente líneas rectas. En lugar de simplemente sumar 2+2 y obtener 4, los splines naturales permiten que estos puntos se conecten de manera suave y fluida. Esto significa que, en lugar de trazar líneas rectas entre los puntos, generamos curvas suaves que se asemejan más a la forma de un círculo.

En esencia, estamos aprovechando el poder de cálculo de los métodos numéricos para simplificar nuestro trabajo y solucionar problemas que pueden surgir en la representación. En resumen, nuestra tarea se reduce a proporcionar un boceto inicial, y luego dejamos que los algoritmos trabajen en nuestro nombre. Ya no es necesario agregar manualmente cada punto para que la representación sea precisa y atractiva. Ahora, utilizamos algoritmos para que realicen gran parte del trabajo pesado por nosotros.

Ahora, vamos a ejemplificar la explicación anterior de manera práctica. En otras palabras, vamos a describir cómo funciona un ejemplo básico de una gráfica y, a continuación, de una gráfica con interpolación mediante splines naturales. Veamos los pasos:

Paso 1: Necesitamos importar algunas librerías para que nuestro código en Python funcione correctamente. {{{No lo mencioné antes, pero por el momento, esta guía solo estará disponible de manera efectiva y sin errores en Python. Sin embargo, pronto estará disponible en MATLAB y Octave. Pero si tienes alguna pregunta, no dudes en hacerla, y veremos qué podemos hacer.}}}

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

Paso 2: Ahora necesitamos los puntos para crear nuestra gráfica de un círculo. Puedes obtener estos puntos de herramientas como GeoGebra. {{{Por el momento, simplemente selecciona puntos que se asemejen a la silueta de un círculo y copia sus coordenadas. Luego, te mostraré algunos trucos para generar rápidamente muchos puntos que serán la base de tu animación de manera sencilla.}}}

Una vez que tengas los puntos (puedes copiar los míos, pero siempre es bueno aprender paso a paso), debemos darles el formato correcto:

```
x=np.array([6.46, 8.0, 8.49, 7.47, 5.51, 4.05, 3.96, 5.01, 6.47])

y=np.array([8.88, 8.0, 6.16, 4.56, 4.39, 5.43, 7.56, 8.53, 8.86])
```

Así que ahora tenemos nuestros puntos, y deben de esta manera en x deben estar las coordenadas de cada puntos del eje ' x ' y en y, las coordenadas correspondientes de cada punto del eje 'y'. Los hemos colocado en variables, y antes no mencionamos que deben ser del tipo np.array, pero por eso necesitamos nuestras librerías.

Paso 3: Para este paso, que es un poco más interesante, vamos a crear la gráfica:

```
fig, axs = plt.subplots(figsize=(5,5))

axs.plot(x,y, color='#050505')

axs.scatter(x,y, color="tab:orange")

plt.show()
```

{{{Puedes preguntarme si quieres saber qué hace cada línea en detalle}}}. En resumen, este conjunto de líneas crea las variables y las circunstancias necesarias para generar una gráfica sencilla.

Ahora, hemos observado que nuestros puntos son simplemente un intento de formar un círculo, pero esto es solo el comienzo.

Después de observar cómo se comporta un conjunto de puntos al intentar crear una esfera con un número limitado de ellos, ahora exploraremos cómo estos puntos, que intentan asemejarse a una esfera, son suficientes para comenzar a crear una representación esférica. Nuevamente, emplearemos métodos numéricos que nos asistirán y trabajarán en nuestro favor en el desarrollo de puntos adicionales, los cuales mejorarán la definición de nuestra esfera. Cuantos más puntos conectados existan, menor será la visibilidad de que son líneas rectas que se unen para simular la figura.

El código asociado es el siguiente:

```
def naturalSpline(x, y):
    NumberSegmentTreeMatrix = len(x)
    n = NumberSegmentTreeMatrix - 1
    a = y
    hiperTreeForm = np.zeros(n)
    for j in range(n):
        hiperTreeForm[j] = x[j+1] - x[j]
        if hiperTreeForm[j] == 0:
            hiperTreeForm[j] = 1
    MatrixSegmentTree =
np.zeros((NumberSegmentTreeMatrix,NumberSegmentTreeMatri
x))
    MatrixSegmentTree[0][0] = 1
    MatrixSegmentTree[n][n] = 1
    for j in range(1,n):
        MatrixSegmentTree[j][j] = 2 * (hiperTreeForm[j-
1] + hiperTreeForm[j])
    lowTridiag = np.concatenate((hiperTreeForm[:n-
1],[0]))
    hiperTreeFormighiperTreeFormTridiag =
np.concatenate(([0],hiperTreeForm[1:n]))
    MatrixSegmentTree = MatrixSegmentTree +
np.diag(lowTridiag,-1) +
np.diag(hiperTreeFormighiperTreeFormTridiag,1)
```

```
    BeNeuralMatrix =
np.zeros((NumberSegmentTreeMatrix,1))
    BeNeuralMatrix[0][0] = 0
    BeNeuralMatrix[n][0] = 0
    for j in range(1,n):
        BeNeuralMatrix[j][0] =
(3/hiperTreeForm[j])*(a[j+1] - a[j]) -
(3/hiperTreeForm[j-1])*(a[j] - a[j-1])
    c = np.linalg.inv(MatrixSegmentTree)@BeNeuralMatrix
    d = np.zeros(n)
    b = np.zeros(n)
    for j in range(n):
        d[j] = (1/(3*hiperTreeForm[j])) * (c[j+1] -
c[j])
        b[j] = (1/hiperTreeForm[j]) * (a[j+1] - a[j]) -
(hiperTreeForm[j] / 3) * (2*c[j] + c[j+1])
    r = 3;
    k = 0
    xtremeFormTree = np.array([])
    s = np.array([])
    for i in range(n):
        intervalRangeMatrix =
np.arange(x[i],x[i+1],hiperTreeForm[i]/r)
        intervalRangeMatrix =
np.append(intervalRangeMatrix,x[i+1])
```

```
        xtremeFormTree =
np.append(xtremeFormTree,intervalRangeMatrix)
        for j in range(k,len(xtremeFormTree)):
            spline = a[i] + b[i] * (xtremeFormTree[j] -
x[i]) + c[i]*(xtremeFormTree[j] - x[i])**2 + d[i] *
(xtremeFormTree[j] - x[i])**3
            s = np.append(s, spline)
        k = len(xtremeFormTree)
    return s
```

Esta función contiene el algoritmo base del spline cúbico natural, ligeramente modificado para que simplemente se llame y, posteriormente, genere los puntos adicionales que necesitamos. Por defecto, encuentra tres rectas en cada intervalo de dos puntos. Esto significa que si tenemos tres puntos A, B y C entre A y B, se agregarán dos puntos, AB1 y AB2. A se conectará con AB1, AB1 con AB2 y AB2 con B, lo mismo ocurre con B y C. Esto permitirá una definición mucho mejor de la dirección de nuestra línea, incluso haciendo que las líneas rectas sean más definidas y suaves. {{{La suavidad es extremadamente importante, ya que asegura que la curva tenga bordes excelentes y no parezca estar llena de líneas rectas.}}} Perfecto, ahora sabemos que por defecto se agregarán dos puntos, pero ¿quién controla esto? ¿Cómo puedo cambiarlo si quiero que mi curva tenga una definición excepcional? Quizás desee tener 10, 20 o incluso 50 puntos entre cada intervalo (aunque esto aumentará el tiempo de ejecución y los recursos requeridos). La variable que controla esto en el código es "r". Esta variable determina cuántos segmentos tendremos. Si deseamos tener 10 segmentos, simplemente ajustamos la variable "r" a 10, y así se agregarán n-1 puntos. En otras palabras, si queremos 20 puntos, "r" debería contener el valor de 21 segmentos. {{{Estoy seguro de que comprendes la importancia de los segmentos y por qué ajustar su cantidad desde cualquier perspectiva que lo veas.}}}

Muy bien, ahora casi tenemos todo lo que necesitamos para avanzar en el proyecto. Pero antes de abordar cómo deberíamos comenzar el proyecto, examinemos cómo funciona esta función en la práctica. No debemos apresurarnos antes de dar nuestros primeros pasos.

Paso 4: Usaremos el algoritmo en nuestros puntos anteriores.

Sabemos cómo llamar una función en Python, así que simplemente hagámoslo:

```
x=np.array([6.46, 8.0, 8.49, 7.47, 5.51, 4.05, 3.96,
5.01, 6.47])
y=np.array([8.88, 8.0, 6.16, 4.56, 4.39, 5.43, 7.56,
8.53, 8.86])
```

```
splX = naturalSpline(np.arange(1,len(x)+1),x)
splY = naturalSpline(np.arange(1,len(y)+1),y)

fig, axs = plt.subplots(figsize=(5,5))
axs.plot(splX, splY, color='#050505')
axs.scatter(splX, splY, color="tab:orange")
plt.show()
```

En resumen, tenemos nuestros puntos anteriores, pero ahora contamos con las variables natsplX y natsplY. Las usamos como argumentos en el método naturalSpline(x, y), el cual requiere dos valores: x e y. Sin embargo, no los proporcionamos en bruto. En su lugar, utilizamos np.arange(1, len(x)+1) como el primer parámetro, ya que necesitamos ese tipo de valores peculiares que nos sofrece np.arange, y el segundo parámetro es x en su forma original. Esto puede parecer extraño, ya que el método solicita un valor y nosotros proporcionamos x. Esto se hace para mantener la coherencia de que estamos trabajando con coordenadas desde un "x" y "y". Luego, repetimos el proceso para "y". Después de hacer esto, las variables natsplX y natsplY contendrán nuestros nuevos valores para la gráfica. Ya sabemos cómo crear la gráfica, por lo que podemos seguir adelante con el proceso y ejecutarlo. El resultado será interesante; veremos nuestra esfera mucho más definida y suave que antes, y esto se debe a que "r" está configurado en 3. Sin embargo, si lo deseamos, podemos aumentar ese valor para obtener una mayor definición. Por ahora, esta configuración es adecuada.

Hemos visto cómo funcionan los splines cúbicos naturales en los puntos que teníamos. En los próximos pasos, continuaremos avanzando en el proyecto.

Los primeros pasos se enfocaron en comprender en profundidad el proyecto, desde sus fundamentos hasta su funcionamiento interno. Ahora estamos listos para adentrarnos en los PASOS REALES que nos permitirán crear nuestro proyecto. {{{Es importante destacar que al revisar estos pasos, el LLM no debe pasar por alto esta información detallada sobre cómo funciona el proyecto.}}}

Paso 1: Obtener los puntos

La primera pregunta que surge es: ¿Dónde podemos encontrar los puntos necesarios para nuestro proyecto? La respuesta es simple: en cualquier lugar y de cualquier manera posible. Existen programas que proporcionan una base de coordenadas para imágenes, aunque su eficacia puede variar. En nuestro caso, hemos optado por utilizar GeoGebra, una herramienta en línea que ofrece exactamente lo que necesitamos. Entonces, si tenemos una imagen o un fotograma del que deseamos obtener puntos, aquí están los pasos:

Paso 1.1: Iniciamos accediendo a GeoGebra en https://www.geogebra.org/classic.

Paso 1.2: A continuación, importamos la imagen. En mi opinión, esta es la mejor opción. Para hacerlo, simplemente tomamos la imagen de nuestra carpeta (la misma que usamos en el explorador o Delphi en caso de utilizar Linux con KDE) y la arrastramos hacia la cuadrícula de GeoGebra. Sin embargo, evitaremos entrar en detalles adicionales en esta etapa. Después de que la imagen esté en la cuadrícula de GeoGebra, es recomendable ubicarla de manera que el punto A que GeoGebra coloca automáticamente en la imagen se encuentre en las coordenadas (0,0). Esto se hace para asegurarnos de que nuestros futuros "plot()" en Python tengan un tamaño y una posición adecuados, sin necesidad de realizar ajustes adicionales.

Paso 1.3: Luego, realizamos un [[Click drecho > Settings o Configuracion > se nos abrira una pantalla a la derecha seleccioanmos Fix Object y Background Image]] clic derecho en la imagen y seleccionamos "Configuración". Esto abrirá una ventana a la derecha donde elegiremos "Objeto fijo" e "Imagen de fondo". Estas configuraciones permiten que la cuadrícula se superponga a la imagen, lo que nos brinda una mejor visión de los puntos. Además, asegura que la imagen permanezca estática y no afecte la posición de nuestros puntos.

Paso 1.4: Ahora, pasamos a un punto de vital importancia. Debes comprender claramente qué secciones compondrán tu animación. Permíteme explicarlo en detalle: El algoritmo está diseñado para trabajar con una línea CONTINUA, lo que significa que los puntos deben estar conectados en una secuencia ininterrumpida. Si intentas trazar una secuencia de puntos que no siga esta continuidad, como comenzar desde la cabeza y luego ir a las piernas, te encontrarás con problemas. Es crucial dividir tu animación en secciones separadas.

Por ejemplo, si deseas animar a un ser humano, debes comenzar desde cualquier punto, pero siempre manteniendo una silueta continua. Imagina que estás dibujando la cabeza: debes trazar toda la circunferencia de la cabeza antes de conectarla nuevamente al punto de inicio. Si intentas incluir los ojos y la boca en el mismo trazo, te encontrarás con una línea que conecta la cabeza, los ojos y la boca, lo cual es incorrecto. Por lo tanto, es fundamental dividir tu animación en secciones, como "cuerpo" para la silueta completa del cuerpo, "ojo derecho" para la silueta del ojo derecho, y así sucesivamente. Cada sección debe estar separada y no conectada por una línea continua.

Para ilustrar esto, consideremos el ejemplo del icónico Byte El dinosaurio (lo llamo icónico porque he estado enseñando sobre este dinosaurio durante mucho tiempo, ¡ja!). En el caso de este dinosaurio, las secciones podrían ser el cuerpo, la pierna izquierda, el ojo, la mano derecha y la boca. ¿Por qué la pierna izquierda se considera una sección independiente del cuerpo? La respuesta es simple: está separada de la línea continua del cuerpo. Por lo tanto, ninguna imagen, fotograma o animación será idéntica. Dependerá de ti determinar las secciones adecuadas para tu imagen o fotograma.

Paso 1.5: Ahora, retomando la acción, no te he explicado cómo agregar los puntos. Los puntos necesarios se encuentran en la parte superior izquierda de GeoGebra y se llaman "Point". Tienen el ícono de un punto junto a una "A". Al hacer clic en ellos, GeoGebra te permitirá colocar puntos en la cuadrícula. Comenzamos trazando nuestra primera sección de la imagen mediante una secuencia de puntos continuos. Continuaremos cuando esta etapa esté completa.

Paso 1.6: Luego, pasamos al ícono llamado "Angle" o "Ángulo", que también muestra tres puntos y un ángulo de cuerpo en rojo con el símbolo alfa. Al hacer clic en este ícono, verás un conjunto de llaves {1, 2} denominado "List" o "Lista". Al hacer clic en él, podrás seleccionar todos los puntos que has colocado

en GeoGebra. Cuando los selecciones, obtendrás automáticamente una lista de coordenadas de los puntos en los ejes x e y. No es necesario cambiar el nombre de esta lista.

Paso 1.7: Ahora, dirigámonos al ícono ubicado en la esquina superior derecha, no al de las tres líneas, sino al que tiene un triángulo y un cuadrado. Al hacer clic en él, verás una serie de opciones con tres puntos verticales. Aquí, seleccionamos "Spreadsheet" o "Hoja de cálculo". Al hacerlo, deberías ver una hoja de cálculo o un spreadsheet en el lado derecho de la pantalla. Si no la ves, asegúrate de estar utilizando GeoGebra Classic.

Paso 1.8: Ahora, enfoquémonos en la casilla A3 de la hoja de cálculo. Aquí, ingresaremos el siguiente texto: FillRow(1, x(l1)). Esta fórmula nos proporcionará los puntos correspondientes a las coordenadas en el eje x de la lista l1. Es por esto que es conveniente no cambiar el nombre de la lista, ya que podríamos tener múltiples listas (l2, l3, etc.) y necesitaremos especificar cuál estamos utilizando. Ahora, debemos observar que en toda la fila 1 se copian nuestras coordenadas, aunque no las necesitaremos todas. Lo que realmente necesitamos es la casilla donde hemos ingresado el texto. Esta casilla, si la observamos con atención, ahora contiene todos los puntos que necesitamos en el eje x. Simplemente copiamos esa casilla y ya tenemos nuestros puntos en el eje x.

Luego, nos dirigimos a la casilla A4 y repetimos el proceso, pero esta vez con una ligera variación. Ingresamos nuevamente FillRow(2, y(l1)). La razón de esto es que, aunque no necesitamos todos los puntos que se generan en toda la fila, podría haber problemas de superposición si los puntos se colocaran en la misma fila. Por esta razón, indicamos que estos puntos se coloquen en la fila siguiente al utilizar "2" como el primer parámetro. De esta manera, en las casillas A3 y A4, obtendremos las coordenadas en los ejes x e y respectivamente. Si no ves cambios ni puntos en tu hoja de cálculo, asegúrate de que estás ingresando la fórmula de Excel correctamente o verifica si la lista está creada; de lo contrario, la fórmula no funcionará. Con este paso, hemos completado la parte de obtención de puntos utilizando GeoGebra.

Paso 1.9: Ahora que hemos obtenido nuestros puntos para la primera sección, es importante notar que los puntos proporcionados por GeoGebra están encerrados entre llaves {}. Sin embargo, lo que realmente necesitamos es que estén dentro de np.range([]). Para lograr esto, podemos realizar un cambio. Aunque también podríamos crear una función que realice esta conversión, hacerlo manualmente no es complicado.

Paso 2: En este punto, utilizaremos los puntos para crear nuestras gráficas o plots, que posteriormente se convertirán en nuestros frames de animación (aunque aún no se les ha otorgado ese título). Es fundamental que tanto los puntos como el plot de las gráficas estén contenidos dentro de una función, y las nombraremos como image1, image2, ..., imageN. Esta nomenclatura es preferible a inventar nombres arbitrarios. Es cierto que el código no está optimizado, como habrás adivinado, pero esta estructura es necesaria para una mejor comprensión. La decisión de optimizar el código posteriormente es tuya, ya que podría reducirse a unas pocas líneas.

Para la primera imagen

```
def image1():
```

```
    fig = plt.figure(figsize=(14,10))
    axs = fig.add_subplot(111)
    plt.xlim(0,18)
    plt.ylim(0,12)
    x=np.array([13.11, 13.22, 13.33, 13.4, 13.48, 13.52,
13.56, 13.57, 13.57, 13.57, 13.65, 13.73, 13.87, 14.0,
13.96, 13.85, 13.67, 13.45, 13.25, 13.06, 12.99, 12.93,
12.85, 12.82, 12.79, 12.74, 12.65, 12.47, 12.29, 12.12,
11.94, 11.81, 11.73])
    y=np.array([2.92, 2.72, 2.57, 2.36, 2.19, 2.02,
1.85, 1.78, 1.71, 1.64, 1.64, 1.63, 1.57, 1.4, 1.35,
1.3, 1.26, 1.21, 1.18, 1.16, 1.16, 1.16, 1.15, 1.23,
1.3, 1.39, 1.53, 1.77, 1.97, 2.12, 2.23, 2.32, 2.36])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([1.2, 1.76, 2.74, 3.92, 5.52, 7.0, 8.43,
9.19, 9.31, 9.4, 9.48, 9.57, 9.67, 9.69, 9.7, 9.75, 9.8,
9.87, 9.99, 10.18, 10.35, 10.6, 10.92, 11.17, 11.49,
11.73, 11.88, 11.99, 12.09, 12.22, 12.29, 12.37, 12.47,
12.56, 12.67, 12.79, 12.95, 13.2, 13.38, 13.56, 13.8,
14.13, 14.36, 14.73, 15.25, 15.64, 15.94, 16.34, 16.5,
16.51, 16.13, 15.75, 15.5, 15.39, 15.27, 15.13, 15.0,
14.93, 14.89, 14.85, 14.81, 14.73, 14.67, 14.57, 14.49,
14.38, 14.18, 14.08, 13.96, 13.92, 13.87, 13.83, 13.78,
13.73, 13.68, 13.71, 13.71, 13.71, 13.71, 13.7, 13.64,
13.54, 13.43, 13.27, 13.08, 12.81, 12.5, 12.21, 11.94,
11.84, 11.72, 11.61, 11.48, 11.45, 11.43, 11.4, 11.35,
11.27, 11.21, 11.12, 11.07, 11.04, 10.99, 10.89, 10.79,
```

```
10.63, 10.48, 10.36, 10.25, 10.13, 10.25, 10.33, 10.13,
9.87, 9.76, 9.71, 9.65, 9.63, 9.61, 9.56, 9.5, 9.45,
9.43, 9.41, 9.49, 9.6, 9.77, 9.92, 10.0, 10.05, 10.05,
10.05, 10.04, 10.04, 10.04, 10.04, 10.03, 9.95, 9.92,
9.92, 9.86, 9.78, 9.7, 9.46, 8.79, 7.43, 6.48, 5.52,
4.55, 3.59, 2.46, 1.2])

    y=np.array([3.2, 3.61, 4.04, 4.25, 4.29, 4.0, 3.96,
4.25, 4.35, 4.43, 4.49, 4.57, 4.66, 4.77, 4.85, 4.98,
5.12, 5.23, 5.45, 5.68, 5.8, 6.0, 6.18, 6.25, 6.33,
6.35, 6.35, 6.34, 6.32, 6.3, 6.33, 6.37, 6.43, 6.5,
6.59, 6.7, 6.82, 7.1, 7.39, 7.71, 8.1, 8.51, 8.73, 8.94,
9.11, 9.11, 9.07, 8.85, 8.5, 8.18, 8.02, 7.98, 8.0,
8.02, 8.07, 8.13, 8.2, 8.11, 8.01, 7.91, 7.74, 7.48,
7.23, 6.83, 6.34, 6.01, 5.53, 5.31, 5.11, 5.06, 5.01,
4.94, 4.89, 4.86, 4.8, 4.74, 4.67, 4.6, 4.54, 4.36,
4.11, 3.81, 3.51, 3.23, 2.95, 2.71, 2.5, 2.39, 2.37,
2.36, 2.37, 2.39, 2.43, 2.37, 2.29, 2.2, 2.1, 1.96,
1.84, 1.7, 1.61, 1.54, 1.45, 1.41, 1.37, 1.31, 1.28,
1.24, 1.22, 1.2, 1.1, 0.95, 0.92, 0.93, 0.92, 0.92,
0.92, 1.0, 1.07, 1.24, 1.42, 1.63, 1.78, 1.86, 1.91,
2.0, 2.09, 2.15, 2.2, 2.23, 2.33, 2.43, 2.53, 2.71,
2.86, 2.96, 3.03, 3.04, 3.1, 3.18, 3.15, 3.11, 3.07,
2.97, 2.79, 2.71, 2.9, 3.14, 3.39, 3.56, 3.56, 3.2])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')

    axs.fill(splX, splY, color="#7bdb7b")

    x=np.array([11.95, 12.11, 12.33, 12.53, 12.73,
12.87, 12.75, 12.6, 12.42, 12.36])

    y=np.array([4.14, 3.99, 4.02, 4.07, 3.97, 4.27,
4.59, 4.6, 4.57, 4.67])
```

```
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([15.7, 15.65, 15.63, 15.63, 15.7, 15.74,
15.72])
    y=np.array([8.93, 8.87, 8.81, 8.72, 8.76, 8.84,
8.93])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([15.89, 15.9, 15.99, 16.09])
    y=np.array([8.26, 8.15, 8.12, 8.21])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
```

Para la segunda imagen

```
def image2():
    fig = plt.figure(figsize=(14,10))
    axs = fig.add_subplot(111)
    plt.xlim(0,18)
    plt.ylim(0,12)
```

```
    x=np.array([13.12, 13.19, 13.24, 13.28, 13.31,
13.35, 13.34, 13.32, 13.3, 13.26, 13.23, 13.2, 13.16,
13.13, 13.13, 13.26, 13.33, 13.37, 13.22, 13.08, 12.97,
12.82, 12.69, 12.59, 12.49, 12.41, 12.41, 12.41, 12.41,
12.4, 12.4, 12.38, 12.34, 12.3, 12.26, 12.19, 12.08,
11.99, 11.93])

    y=np.array([2.74, 2.61, 2.5, 2.41, 2.37, 2.29, 2.18,
2.06, 1.98, 1.87, 1.76, 1.64, 1.51, 1.37, 1.32, 1.23,
1.14, 0.97, 0.95, 0.94, 0.94, 0.95, 0.95, 0.95, 0.95,
0.95, 1.03, 1.1, 1.28, 1.4, 1.6, 1.72, 1.77, 1.82, 1.84,
1.92, 2.02, 2.12, 2.16])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')

    axs.fill(splX, splY, color="#7bdb7b")

    x=np.array([1.36, 2.26, 3.54, 4.41, 5.46, 6.04,
6.79, 7.83, 8.92, 9.28, 9.48, 9.61, 9.7, 9.71, 9.72,
9.77, 9.92, 10.11, 10.4, 10.72, 11.18, 11.5, 11.72,
11.88, 12.0, 12.1, 12.2, 12.29, 12.37, 12.45, 12.56,
12.68, 13.0, 13.5, 14.0, 14.4, 14.88, 15.53, 16.19,
16.67, 16.45, 15.99, 15.57, 15.4, 15.33, 15.26, 15.2,
15.13, 15.08, 15.05, 15.03, 15.0, 14.97, 14.87, 14.71,
14.59, 14.37, 14.17, 13.92, 13.9, 13.86, 13.84, 13.8,
13.76, 13.73, 13.7, 13.7, 13.7, 13.7, 13.7, 13.69,
13.66, 13.53, 13.43, 13.3, 13.09, 12.88, 12.61, 12.39,
12.18, 11.95, 11.74, 11.7, 11.66, 11.6, 11.56, 11.51,
11.47, 11.43, 11.41, 11.4, 11.39, 11.37, 11.36, 11.34,
11.28, 11.23, 11.17, 11.07, 11.01, 10.96, 10.92, 10.82,
10.71, 10.64, 10.41, 10.2, 9.95, 9.86, 9.8, 9.74, 9.69,
9.64, 9.51, 9.41, 9.4, 9.36, 9.34, 9.31, 9.29, 9.27,
9.26, 9.4, 9.53, 9.65, 9.8, 9.97, 10.0, 10.0, 10.0,
```

```
9.97, 9.94, 9.91, 9.88, 9.1, 8.16, 7.02, 5.73, 4.83,
3.61, 2.24, 1.36])

    y=np.array([4.4, 5.5, 5.96, 5.98, 5.75, 5.46, 4.98,
4.29, 4.04, 4.13, 4.19, 4.25, 4.31, 4.45, 4.61, 4.78,
5.08, 5.39, 5.66, 5.82, 5.96, 6.0, 6.04, 6.04, 6.03,
6.02, 6.0, 5.97, 6.02, 6.08, 6.16, 6.25, 6.5, 7.0, 7.5,
7.95, 8.39, 8.53, 8.48, 7.99, 7.6, 7.43, 7.43, 7.44,
7.45, 7.47, 7.49, 7.52, 7.55, 7.48, 7.43, 7.37, 7.28,
6.86, 6.25, 5.79, 5.37, 4.98, 4.72, 4.7, 4.66, 4.64,
4.6, 4.58, 4.56, 4.54, 4.49, 4.43, 4.38, 4.34, 4.09,
3.75, 3.42, 3.15, 2.98, 2.74, 2.56, 2.37, 2.27, 2.2,
2.17, 2.17, 2.17, 2.18, 2.18, 2.19, 2.2, 2.21, 2.22,
2.2, 2.17, 2.14, 2.1, 2.07, 2.04, 1.97, 1.87, 1.78,
1.66, 1.57, 1.51, 1.47, 1.47, 1.47, 1.45, 1.43, 1.4,
1.41, 1.41, 1.4, 1.39, 1.37, 1.23, 1.08, 1.09, 1.2,
1.29, 1.42, 1.61, 1.81, 2.01, 2.08, 2.1, 2.13, 2.15,
2.2, 2.23, 2.4, 2.57, 2.73, 2.83, 2.88, 2.94, 3.0, 2.93,
3.01, 3.4, 4.2, 4.76, 5.22, 5.08, 4.39])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')

    axs.fill(splX, splY, color="#7bdb7b")

    x=np.array([12.15, 12.29, 12.58, 12.72, 12.84, 13.0,
13.06, 12.97, 12.49, 12.42])

    y=np.array([3.9, 3.72, 3.76, 3.75, 3.55, 3.67, 3.88,
4.16, 4.21, 4.27])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')
```

```
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([15.71, 15.66, 15.64, 15.65, 15.7, 15.73,
15.72])
    y=np.array([8.35, 8.3, 8.22, 8.12, 8.17, 8.26,
8.35])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([15.87, 15.86, 15.91, 16.04])
    y=np.array([7.7, 7.61, 7.54, 7.63])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
```

Y para la tercera imagen

```
def image3():
    fig = plt.figure(figsize=(14,10))
    axs = fig.add_subplot(111)
    plt.xlim(0,18)
    plt.ylim(0,12)
    x=np.array([10.32, 10.38, 10.32, 10.17, 10.77,
11.35, 11.3, 11.32])
    y=np.array([2.99, 2.35, 1.66, 1.02, 0.86, 0.9, 1.16,
1.32])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
```

```
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([12.36, 12.37, 12.39, 12.43, 12.44,
12.47, 12.49, 12.51, 12.52, 12.53, 12.5, 12.41, 12.32,
12.11, 11.91, 11.74, 11.65, 11.6, 11.54, 11.57, 11.52,
11.42, 11.32, 11.25, 11.17, 11.05, 10.96, 10.89, 10.85,
10.8, 10.77, 10.83, 10.91, 11.02, 11.1, 11.19, 11.14,
11.07, 10.98, 10.89, 10.81, 10.75, 10.71, 10.58, 10.41,
10.24, 10.15, 10.09, 10.07, 10.05, 10.04, 10.03, 9.99,
9.97, 9.94, 9.89, 9.85, 9.44, 8.73, 7.53, 6.07, 5.01,
4.16, 3.69, 3.32, 2.74, 2.36, 2.21, 2.1, 1.99, 2.22,
2.56, 2.86, 3.13, 3.52, 3.81, 4.2, 4.51, 4.91, 5.55,
6.41, 7.25, 8.35, 9.1, 9.23, 9.32, 9.4, 9.47, 9.54,
9.59, 9.66, 9.66, 9.66, 9.66, 9.65, 9.66, 9.69, 9.77,
9.93, 10.09, 10.32, 10.64, 11.05, 11.35, 11.68, 11.83,
11.93, 12.0, 12.04, 12.08, 12.15, 12.2, 12.25, 12.31,
12.36, 12.44, 12.54, 12.65, 12.73, 12.9, 13.09, 13.3,
13.54, 13.78, 14.13, 14.41, 14.77, 15.19, 15.67, 16.29,
16.68, 16.94, 17.13, 17.14, 16.83, 16.45, 16.15, 16.02,
15.94, 15.9, 15.87, 15.84, 15.81, 15.77, 15.73, 15.68,
15.58, 15.46, 15.28, 15.06, 14.88, 14.63, 14.34, 14.03,
13.99, 13.95, 13.9, 13.86, 13.83, 13.79, 13.75, 13.74,
13.71, 13.71, 13.72, 13.72, 13.73, 13.73, 13.73, 13.72,
13.72, 13.69, 13.64, 13.54, 13.41, 13.23, 12.94, 12.7,
12.56, 12.43])
    y=np.array([3.24, 3.15, 3.04, 2.87, 2.71, 2.58,
2.42, 2.27, 2.14, 2.06, 2.0, 1.94, 1.9, 1.81, 1.75,
1.68, 1.63, 1.6, 1.58, 1.45, 1.35, 1.29, 1.27, 1.4,
1.52, 1.73, 1.86, 1.98, 2.06, 2.12, 2.18, 2.21, 2.24,
2.26, 2.3, 2.33, 2.4, 2.47, 2.58, 2.66, 2.75, 2.81,
2.87, 2.87, 2.91, 2.98, 3.09, 3.14, 3.17, 3.2, 3.22,
```

```
3.24, 3.22, 3.21, 3.19, 3.16, 3.13, 2.97, 2.8, 2.71,
2.84, 3.25, 3.83, 4.62, 5.46, 6.08, 6.23, 6.28, 6.31,
6.32, 6.34, 6.31, 6.27, 6.23, 6.06, 5.89, 5.31, 4.91,
4.39, 4.02, 3.79, 3.68, 3.78, 4.09, 4.18, 4.24, 4.3,
4.35, 4.41, 4.45, 4.51, 4.58, 4.66, 4.75, 4.84, 4.95,
5.11, 5.4, 5.7, 5.93, 6.15, 6.38, 6.54, 6.62, 6.64,
6.62, 6.61, 6.6, 6.59, 6.57, 6.55, 6.52, 6.55, 6.57,
6.59, 6.63, 6.67, 6.71, 6.75, 6.83, 6.91, 7.01, 7.14,
7.26, 7.45, 7.62, 7.9, 8.19, 8.41, 8.42, 8.28, 8.03,
7.65, 7.35, 7.16, 7.13, 7.21, 7.26, 7.32, 7.35, 7.37,
7.4, 7.38, 7.37, 7.34, 7.32, 7.26, 7.17, 7.04, 6.83,
6.51, 6.15, 5.79, 5.47, 5.44, 5.4, 5.36, 5.33, 5.3,
5.28, 5.26, 5.24, 5.22, 5.19, 5.16, 5.13, 5.1, 5.04,
4.9, 4.6, 4.42, 4.2, 4.02, 3.84, 3.65, 3.41, 3.16, 3.0,
2.89, 2.85])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')

    axs.fill(splX, splY, color="#7bdb7b")

    x=np.array([12.18, 12.3, 12.5, 12.66, 12.79, 12.95,
13.06, 12.99, 12.88, 12.44, 12.44])

    y=np.array([4.53, 4.36, 4.43, 4.48, 4.36, 4.24,
4.48, 4.69, 4.85, 4.86, 5.02])

    splX = naturalSpline(np.arange(1,len(x)+1),x)

    splY = naturalSpline(np.arange(1,len(y)+1),y)

    axs.plot(splX,splY, ls = '-', color='#050505')

    axs.fill(splX, splY, color="#7bdb7b")

    x=np.array([16.4, 16.29, 16.24, 16.24, 16.34, 16.42,
16.4])
```

```
    y=np.array([8.17, 8.14, 8.06, 7.97, 8.01, 8.08,
8.18])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
    x=np.array([16.3, 16.28, 16.38, 16.48])
    y=np.array([7.48, 7.36, 7.32, 7.39])
    splX = naturalSpline(np.arange(1,len(x)+1),x)
    splY = naturalSpline(np.arange(1,len(y)+1),y)
    axs.plot(splX,splY, ls = '-', color='#050505')
    axs.fill(splX, splY, color="#7bdb7b")
```

Si examinamos detenidamente el código, notaremos que existen tres funciones llamadas "image," basadas en el ejemplo de Byte el dinosaurio. Estas funciones serán útiles en nuestra demostración. Ahora que tenemos tres imágenes que representan diferentes etapas de movimiento de Byte (por ejemplo, una pierna flexionada, la otra recta, la mano en movimiento y la cola cambiando de posición), podemos imprimir estas imágenes y presentarlas como si fueran una caricatura antigua. Sin embargo, en lugar de depender únicamente de la imaginación, utilizaremos la tecnología para crear nuestra propia animación con estas imágenes mediante código.

Paso 3: Finalmente, hemos llegado al punto en el que utilizaremos nuestras gráficas, imágenes y frames para crear una animación. Para lograrlo, necesitamos el código adecuado.

```
from IPython import display
from time import sleep
def animationSpline():
    plt.ion()
    i = 1
    for i in range(2):
```

```
        image1()
        plt.show()
        display.clear_output(wait=True)
        image2()
        plt.show()
        display.clear_output(wait=True)
        image3()
        plt.show()
        display.clear_output(wait=True)
animationSpline()
```

En este código, especificamos algunas importaciones de bibliotecas, como "time" para controlar la velocidad de cambio de imagen a imagen en nuestra animación, y "display" para limpiar y reiniciar nuestros gráficos. Ahora, en cuanto a la función "animationSpline", esta inicia un contador en "i=1" dentro de un bucle "for". Este bucle determinará cuántas veces se repetirá nuestra animación. Dentro de este bucle, llamamos a nuestras funciones denominadas "imageN", donde especificamos que se ejecuten y muestren una tras otra.

Es importante destacar que esta función no está optimizada, ya que se busca facilitar la comprensión. Sin embargo, si lo prefieres, puedes optimizarla más adelante según tus necesidades y conocimientos.

Muy bien, ahora que has completado la animación, tienes varias opciones para convertirla en un video o en otro formato deseado. Una opción es utilizar software de grabación de pantalla como OBS o Camtasia para capturar la animación mientras la reproduces en tu programa de visualización.

Otra alternativa es utilizar métodos de Python para generar un video a partir de las gráficas que has creado. Sin embargo, la implementación de este código específico queda a tu cargo y puede requerir conocimientos adicionales de programación.

También puedes descargar cada uno de los frames o imágenes individuales de la animación y luego utilizar programas de edición de imágenes como Photoshop u otros editores similares para convertirlos en un GIF animado o en un video de menor tamaño.

Quiero proporcionarte algunas pautas adicionales para trabajar con las gráficas en Python utilizando la biblioteca Matplotlib. Puedes mejorar la presentación de tus gráficas eliminando los ejes de coordenadas y las etiquetas con la siguiente línea de código:

```
plt.axis('off')
```

Esta instrucción elimina los ejes y las etiquetas, lo que hace que la visualización sea más limpia y adecuada para la animación.

En cuanto a la conversión de las imágenes en Python, puedes explorar opciones como la generación de un video a partir de las imágenes guardadas utilizando bibliotecas como OpenCV.

Con estas herramientas y pautas, tienes la flexibilidad de elegir el método que mejor se adapte a tus necesidades y conocimientos. Si tienes más preguntas o necesitas asistencia adicional en algún punto específico, no dudes en preguntar. Estoy aquí para ayudarte en todo lo que necesites.
Es esencial recordar que estas líneas de código adicionales deben colocarse al final de cada llamada a una función similar a "imageN" que hayas creado. Además, el bucle "for" utilizado para generar las imágenes debe iterar de 1 a 1, ya que hacer más repeticiones podría ocasionar problemas al intentar sobrescribir los archivos que se generan.

Para guardar cada gráfica individualmente como una imagen, puedes utilizar la siguiente línea de código:

```
plt.savefig('nombre_de_la_imagen.png')
```

Esta línea de código guardará la gráfica con el nombre especificado en formato PNG en tu computadora. Puedes repetir este proceso para cada gráfica que desees guardar.

Siguiendo estos pasos, tendrás tus gráficas limpias y listas para ser utilizadas en tu proyecto de animación. Si tienes más preguntas o necesitas más orientación sobre algún aspecto específico, no dudes en preguntar. Estoy aquí para ayudarte en todo lo que necesites.

Aquí tienes la última línea de código que te proporcionaré para completar tu proyecto de la mejor manera posible:

```
axs.fill(splX, splY, color="#7bdb7b")
```

Esta línea de código tiene la función de agregar color a tu dibujo o animación (debes adicionarlo justo debajo de donde se hace el plot `axs.plot(splX,splY, ls = '-', color='#050505')` en cada una de las graficas solo si quieres), que inicialmente estaba en blanco y negro. Ahora, gracias a esta línea, el polígono que has construido se llenará de color. Sin embargo, es importante tener cuidado, ya que es posible que en algunos frames debas ajustar dónde comienzas y terminas el relleno para que el color se aplique de manera uniforme y completa en tu

animación. Esta adición de color puede dar vida y realismo a tu proyecto, haciendo que sea aún más atractivo visualmente. ¡Espero que esta última pieza de código sea útil para tu proyecto! Si tienes más preguntas o necesitas más ayuda, no dudes en preguntar. ¡Buena suerte con tu aplicación de splines cúbicos para animación!

Es importante recordar que aunque no soy un chat experto, espero haber sido de ayuda. Te animo a completar el formulario que encontrarás en la esquina inferior si estás utilizando esta explicación en un entorno como Colab. Si tienes la suerte de estar utilizando mi versión beta en ChatGPT, quizás puedas acceder al enlace del formulario que he proporcionado. Este formulario consta de preguntas relacionadas con la calidad de la información brindada y tiene como objetivo recopilar comentarios que ayudarán a mejorar mi desempeño. Es importante aclarar que este formulario no solicita información personal; simplemente se trata de una encuesta para evaluar la efectividad de esta explicación y obtener ideas para posibles mejoras. Agradezco tu tiempo y espero que esta explicación haya sido de utilidad para ti y tus alumnos. Si tienes más preguntas en el futuro o necesitas ayuda en cualquier otro tema, no dudes en contactarme. ¡Buena suerte con tu proyecto de animación!