Modelos de Optimización con Manim

Este proyecto tiene el objetivo de visualizar conceptos y algoritmos utilizando la biblioteca Manim. Cuenta con una aplicación visual donde podrá realizar las siguientes operaciones:

- Solución geométrica para problemas de programación lineal
- Método Simplex
- Planos cortantes
- Ramificación y acotación
- Ideas geométricas de demostración de teoremas
- Métodos numéricos para la optimización no lineal
- Búsqueda en línea
- Penalización

Requerimientos

Debe contar con una versión de Python entre 3.7 y 3.9. En el archivo de requerimientos podrá necontrar una lista de las dependencias necesarias para la ejecución del programa. Además cuenta con un `Makefile` que permite instalar todo lo requerido cuando ejecute en la terminal el comando `make install`. Si desea hacerlo de forma manual estos son los pasos que debe seguir:

```
    sudo apt update
    sudo apt install libcairo2-dev libpango1.0-dev ffmpeg
    sudo apt install python3-pip
    sudo apt-get install coinor-cbc
    sudo apt-get install -y git pkg-config coinor-libcbc-dev coinor-libosi-
dev coinor-libcoinutils-dev coinor-libcgl-dev
    export COIN_INSTALL_DIR=/usr/
    pip3 install --pre cylp
    python3 -m pip install -r requirements.txt
```

Modo de uso

La aplicación visual fue generada con `streamlit` por lo que para ejecutarla deberá usar el comando:

```
streamlit run ui/main.py
```

Alternativamente puede usar la regla implementada en el `Makefile`:

```
make run
```

o simplemente

```
make
```

La regla por defecto es `run` por lo que no hay diferencia entre usar uno u otro.

Detalles de implementación

La aplicación visual se realizó, como ya fue mencionado, a partir del uso de `streamlit`. Toda la lógica de implementación se encuentra en la carpeta ui (User Interface). La idea fundamental es que por cada acción posible se implementa el código de la página que le corresponde; y existe una página principal donde están listadas todas las opciones posibles. En este caso las opciones se encuentran en un `sidebar` cuyo código puede encontrar aquí

Cada página genera un formulario que deberá ser completado por el usuario que contendrá la información necesaria del problema para ejecutar el algoritmo de solución correspondiente. Esta información produce un archivo `.json` que será utilizado por la lógica del programa para obtener los datos de entrada. La creación del archivo y su ubicación están especificadas en el archivo de la acción correspondiente.

Para el graficado de los modelos se empleó la biblioteca Manim (vea este link para más información), tema que se tocará en secciones siguientes.

A continuación se explicará con más detalle el funcionamiento de cada una de las posibles acciones:

Solución Geométrica para un problema de optimización lineal:

La solución geométrica asociada a un problema de optimización lineal se realizó siguiendo los siguientes pasos:

- Se usan todas las restricciones del problema y se grafican en el plano como rectas.
- Se hallan las intercecciones entre estas rectas, las cuales constituyen puntos.
- Se elige entre todos los puntos de intercepción solo aquellos que cumplan todas las restricciones del problema, los demás se descartan.
- Luego se usa la función objetivo para determinar cuáles son las evaluaciones de estos puntos en la misma y ordenarlos en cuanto a su valor para encontrar el mínimo y el máximo.
- Los resultados de la evaluación se almacenan en una estructura diccionario donde el identificador es un entero entre 0 y la cantidad de puntos de intersección encontrados, después de haber sido ordenados.
- Se crean dos listas **m**, **n** que contienen los valores de **x**, **y** respectivamente de los puntos ya ordenados.
- Para encontrar el mínimo se escoge, de la estructura diccionario anteriormente descrita, el identificador que pertenece al menor valor almacenado en este y se utiliza como posición para indexar en las listas **m**, **n** y de esta forma se obtiene el mínimo. Análogamente se obtiene el máximo, buscando el identificador de mayor valor en el diccionario.

Luego para graficar en Manim, se siguen los pasos anteriores hasta obtener todos los datos necesarios tales como las rectas, las intersecciones de estas rectas que son válidas para el problema, así como el punto máximo y mínimo. Hacemos aparecer las líneas, luego los puntos intersección, luego el polígono asociado a estos puntos y por último los puntos de máximo y mínimo en caso de existir.

Detalles del código:

Todo el proceso para hallar las intersecciones así como el mínimo y el máximo están en el archivo **geometric.py** en un método llamado **geometric_aproach**, el cual recibe 4 parámetros, las restricciones en forma de listas de strings, la ecuación objetivo en forma de string y dos listas de valores **x**, **y** con los cuales se formarán las rectas.

Luego el proceso para graficar con Manim se encuentra en el archivo **Geo_Manim.py** donde primero se realiza el llamado a **geometric_aproach** para obtener todos los datos necesarios y luego se procede a graficar en orden las líneas, puntos de intersección, el polígono asociado a estos puntos y por último los puntos de máximo y mínimo.

Por último, los detalles como la función y las restricciones se almacenan en un archivo **geometric_aproach.json** el cual se utiliza como configuración del problema a resolver.

Luego el proceso para graficar con Manim se encuentra en el archivo **Geo_Manim.py** donde primero se realiza el llamado a **geometric_aproach** para obtener todos los datos necesarios y luego se procede a graficar en orden las líneas, puntos de intersección, el polígono asociado a estos puntos y por último los puntos de máximo y mínimo.

Por último, los detalles como la función y las restricciones se almacenan en un archivo **geometric_aproach.json** el cual se utiliza como configuración del problema a resolver.

Se utilizó como nombre de variables *"x"* y *"y"*.

Método Simplex:

El paquete `simplex` consta de dos módulos principales: manim_model y simplex_method además contiene un módulo utils con funciones complementarias.

simplex_method

El punto de entrada de este módulo es `call_simplex` que se encarga de decodificar el `json` con los datos del problema y llama al método que se encarga de resolverlo, o sea, de ejecutar el algoritmo Simplex. En este método también se produce un mensaje con los resultados obtenidos del algoritmo y lo almacena en un archivo temporal `.temp`; dicho archivo será utilizado por la lógica de la aplicación visual que mostrará su contenido junto con el archivo multimedia que se muestran como resultado del cómputo. Una vez que ha cumplido su cometido, el archivo temporal es eliminado.

Para llevar a cabo el método Simplex se empleó la biblioteca `scipy.optimize` que presenta una función `linprog` (vea este link para más información) a la cual se le puede pasar como parámetro el método que se desea utilizar, en este caso `"simplex"`. Esta función también recibe como parámetro opcional una función `callback` que se ejecuta en cada iteración del método. En el caso de este proyecto, la función `callback` se encarga de almacenar los resultados parciales del algoritmo en cada iteración en una lista de resultados. Puede observarlo en el método `simplex_algorithm` del módulo en cuestión.

Se retorna entonces la lista de resultados y la respuesta final.

manim_model

Para graficar con Manim se implementaron dos clases, una para graficar modelos en dos dimensiones (plano XY) y en tres dimensiones. Aunque la cantidad de variables puede ser mayor que 3, en general el análisis del problema se realizará en la proyección sobre el plano XY. Solo en caso de que el modelo tenga exactamente 3 variables, se mostrará también el modelo en tres dimensiones.

Para trabajar en la proyección de la superficie, lo primero que se realiza es una transformación de los datos si la cantidad de variables es mayor que 2. Puede ver como se grafican las funciones de las restricciones del problema. Luego se toma la lista de resultados del algoritmo simplex para obtener los puntos que se computaron en cada iteración, de modo que se mostrará en la figura estos puntos de modo secuencial. Ellos se pintarán de amarillo. Si el algoritmo Simplex fue exitoso en encontrar el punto para el cual se alcanza el óptimo de la función, entonces se añade este punto con color rojo para indicarlo.

> Note que si usted posee el archivo `json` con los datos de entrada, no necesita utilizar la aplicación visual para obtener el archivo multimedia de resultado, puede directamente invocar a `manim` desde la terminal.

Planos cortantes

Dado un problema lineal de optimización con restricciones, el método `construct` de la escena `Canvas` (que se encuentra aquí) se encarga de computar los cortes, basado en la estrategia de cortes de Gomory para problemas con solución entera, del problema especificado en el archivo `json` (que se encuentra aquí)

Un ejemplo de entrada:

```
{
  "vars": ["x", "y"],
  "func": "2*x-3*y",
  "constraints": ["x + 3*y <= 5", "2*x + y <= 6", "-x <= 0", "-y <= 0"],
  "x_range": [0, 8],
  "y_range": [0, 8],
  "A": [[ 1,  3],
  [  2,  1],
  [  -1,  0],
  [  0,  -1]
  ],
  "b": [5, 6, 0, 0],
  "c": [2, -3]
}
```

Se deben especificar las variables a usar, así como las restricciones con signo <=, la función a optimizar debe estar escrita en términos que permitan minimizarla, es decir, se debe utilizar el opuesto de los coeficientes si se desea hallar el máximo. Además, deben proveerse la matriz *A*, y vectores *b* , *c*, de manera tal que el problema sea expresado como $$\min{ca} ~~\text{tal que}~~ Ax \leq b$$. Note que este archivo será generado por la aplicación visual, así que no debería contener errores.

Primeramente, se cargan de entrada en el método **load_cp_model** del módulo **input_parser**, el cual recibe como parámetro la ubicación del archivo que contiene los datos, en este caso, por defecto se hará:

```
load_cp_model('./src/cutting_planes/model_cp.json')
```

Con esto, se graficarán las restricciones del problema, y luego utilizando una versión modificada del método `bnSolve` de la libería `coinor.cuppy`, cuya versión modificada se adjunta con el código de nuestro proyecto aquí.

Para graficar las restricciones se hace un preprocesamiento, donde se detectan aquellas restricciones de la forma $$x \leq c$$, con *c* constante, pues de estas restricciones no es posible generar una función lambda que pasarle a manim para graficar, y como alternativa se usa la función *get_vertical_line*.

Para computar los cortes además existen dos métodos, especificando como parámetro a la función `bnSolve`. Uno de ellos, el más eficiente, es `gomoryMixedIntegerCut`, el cual en ocasiones genera cortes con coeficientes extremadamente grandes. Esto no es necesariamente malo, pero manim a la hora de representar esto tiene bugs visuales debido a la manera en que grafica, de modo que pueden aparecer varios cortes en lugar de uno. Esto ocurre con el ejemplo adicional provisto en el archivo `module_cp_2.json`. Como alternativa, se puede usar el método `liftAndProject`. Este, por otra parte, es considerablemente más ineficiente y genera múltiples cortes innecesarios, en cambio, las representaciones visuales son más exactas.

Una vez obtenidos los cortes, estos se grafican, luego se agregan los ejes de coordenadas para rápida referencia. Finalmente, agregamos el punto óptimo calculado y sus coordenadas a la escena.

Ramificación y acotación:

Este método se encuentra dentro del archivo `bab.py` y se utiliza para encontrar soluciones enteras en un problema de minimización agregando restricciones a medida que se va resolviendo la problemática original.

```
    def branch_and_bound_int(
        vars, func, constraints, initial_point, verbose=False
    ):
```

Parámetros de entrada:

- vars (list[string]): variables que posee la función a minimizar
- func (string): función que se desea minimizar
- constraints (list[string]): restricciones que posee la función a minimizar
- initial_point (list[float]): punto inicial para comenzar el algoritmo de minimización
- verbose (bool) *opcional*: imprimir los pasos que va realizando el método a medida que va minimizando

Retorna:

```
    {
        "min": min,
        "min_values": min_values,
```

```
        "tree": tree
    }
```

donde:

- "min" (float): mínimo valor alcanzado
- "min_values" (list[float]): x en que se alcanza el mínimo valor
- "tree" (root): el recorrido que se realizó para alcanzar el mínimo valor

`root` posee la siguiente estructura:

```
    {
        "father": father,
        "added_constraints": added_constraints,
        "childrens": childrens,
        "lv": lv,
        "step": step,
        "initial_point": initial_point,
        "constraints": constraints,
        "best_point": best_point,
        "evaluation": evaluation,
    }
```

donde:

- "father" (root): el nodo padre desde el que se agregó la nueva restricción
- "added_constraints" (sp.Relational): restricción añadida con respecto al nodo padre
- "childrens" (list[root]): nodos hijos que surgen al agregar nuevas restricciones al nodo actual
- "lv" (int): nivel en el árbol del nodo(la raíz posee lv 0)
- "step" (int): indica cuantos nodos fueron procesados antes del actual
- "initial_point" (list[float]): punto inicial utilizado para la optimización del nodo actual
- "constraints" (list[constraint]): lista de restricciones utilizadas en el nodo actual
- "best_point" (list[float]): x en que se alcanza el mínimo valor en el nodo actual
- "evaluation" (float): mínimo valor alcanzado en el nodo actual

`constraint` posee la estructura:

```
    {
        "type": type,
        "fun": fun,
    }
```

donde:

- "type" ("eq" o "ineq"): indica si la restricción es una inecuación o una ecuación
- "fun" (lambda list[float]: float): la evaluación de la restricción en un punto dado

La idea central detrás del algoritmo es ir agregando restricciones mientras las componentes del vector no sean enteras. Por cada una de las variables no enteras se crean dos nodos hijos, asumamos que la variable no entera es *x* y su valor es *f*, entonces los dos hijos nuevos creados poseen las restricciones `x <= parte_entera_por_debajo(f)` y `x >= parte_entera_por_debajo(f) + 1` respectivamente.

Graficar con manim

La implementación se encuentra dentro del archivo `manim_bab.py` y se utiliza para generar una animación del proceso de optimización utilizando ramificación y acotación de una problemática planteada dentro de un archivo que debe ser creado con el nombre `input.json`. Esta debe tener la siguiente estructura:

```
{
    "vars": vars,
    "func": func,
    "constraints": constraints,
    "initial_point": initial_point,
    "u_range": u_range,
    "v_range": v_range,
    "stroke_width": stroke_width
}
```

donde:

- "vars" (list[string]): las diferentes variables que se encuentran en la función a minimizar
- "func" (string): la función que se desea minimizar
- "constraints" (list[string]): las diferentes restricciones que se desean agregar
- "initial_point" (list[float]): punto inicial que se tomará para la minimización de la función
- "u_range" ([float, float]): el intervalo que se generará en el gráfico en el eje x
- "v_range" ([float, float]): el intervalo que se generará en el gráfico en el eje y
- "stroke_width" (float): grosor de las líneas en el gráfico a generar

Ejemplo:

```
{
    "vars": ["x", "y"],
    "func": "- (7 * x * y / 2.71828 ** ( x ** 2 + y ** 2))",
    "constraints": ["x >= -2", "y >= -2", "x <= 2", "y <= 2"],
    "initial_point": [1, 1],
    "u_range": [-5, 5],
    "v_range": [-5, 5],
    "stroke_width": 0.5
}
```

La idea central en este método es representar la función a minimizar como centro de la animación e ilustrar los distintos nodos de la solución alcanzados mediante un recorrido del árbol de soluciones.

Ideas geométricas de las soluciones de los teoremas

En este caso solamente se desarrolló la idea detrás del algoritmo Simplex acerca de los puntos extremales del conjunto de puntos factibles, para ello se muestra un ejemplo sobre un modelo específico que evidencia el funcionamiento de este procedimiento. Por tanto, en este caso solamente se necesitó el código en simplexEx. Este es una réplica del código presente en la implementación del Método Simplex, explicado anteriormente, solo que se conocen los datos del modelo de antemano, y se añade con flechas la señalización del movimiento de los puntos que contine la lista de resultados, sobre el conjunto de puntos factibles.

Métodos numéricos para la optimización no lineal

Estos métodos se encuentran dentro de los archivos `gradient.py`, `gradient_conj` y `newton`,los cuales implementan el Método de gradiente, Método de gradiente conjugado y el Método de Newton respectivamente. Estos se utilizan para encontrar soluciones mínimas en una función sin restricciones, tienen la ventaja que no necesitan que la función sea lineal.

```
    def gradient(vars, func, initial_point, cycles=100, verbose=False):
    def gradient_conj(vars, func, initial_point, cycles=100,
verbose=False):
    def def newton(vars, func, initial_point, cycles=100):
```

tal que:

- vars (list[string]): variables que posee la función a minimizar
- func (string): función que se desea minimizar
- initial_point (list[float]): punto inicial para comenzar el algorítmo de minimización
- cycles (int): cantidad de iteraciones máximas que se desean realizar para obtener una aproximación del mínimo valor.
- verbose (bool) *opcional*: imprimir los pasos que va realizando el método a medida que va minimizando

y retorna:

```
    {
        "min": min,
        "min_values": min_values,
        "points": points
    }
```

donde:

- "min" (list[float]): x en que se alcanza el mínimo valor
- "min_values" (float): mínimo valor alcanzado
- "points" (list[point]): el recorrido que se realizó para alcanzar el mínimo valor

`point` posee la estructura:

- En el caso del Método de Newton

```
list(float)
```

- En caso del Método del gradiente

```
{
    "point": point,
    "value": value,
    "iteration": iteration,
    "gradient": gradient,
    "step_arrived": step_arrived,
}
```

- En caso del Método del gradiente conjugado

```
{
    "point": point,
    "value": value,
    "iteration": iteration,
    "gradient": gradient,
    "s": s,
    "step_arrived": step_arrived,
}
```

En estos dos últimos se tiene que.

- "point" (list[float]): valores de las componentes del punto mínimo actual
- "value" (float): mínimo valor alcanzado en el punto acutal
- "iteration" (int): indica cuantos puntos fueron procesados antes del actual
- "gradient" (list[float]): Evaluación del gradiente en el punto actual
- "s" (list[float]): En caso del Método del gradiente este valor no está presente pues siempre es el opuesto del gradiente, pero en el Método del gradiente conjugado, este valor se recalcula en cada iteración con respecto a resultados de este anteriores y por esto se almacena.
- "step_arrived" (float): tamaño de paso utilizado en la iteración anterior para llegar al punto actual.

Las ideas detrás de estos algoritmos consisten en que, dado un punto inicial,se debe desplazar en una dirección que asegure que la función disminuya. Para ello, se hace uso de las derivadas de la función en cada componente y se desplaza en sentido contrario al gradiente. Para calcular cuánto debe ser el desplazamiento en una dirección se optimiza el tamaño del paso utilizando un algoritmo de optimización de una sola variable.

Graficar con manim

La implementación se encuentra dentro del archivo `manim_numerical_optimization.py`y se utiliza para generar una animación del proceso de optimización utilizando distintos métodos de optimización

numérica para problemas no lineales, la problemática debe estar planteada dentro de un archivo que debe ser creado con el nombre `input.json`. Esta debe tener la siguiente estructura:

```
{
    "vars": vars,
    "func": func,
    "initial_point": initial_point,
    "u_range": u_range,
    "v_range": v_range,
    "stroke_width": stroke_width
    "cycles": cycles
}
```

donde:

- "vars" (list[string]): las diferentes variables que se encuentran en la función a minimizar
- "func" (string): la función que se desea minimizar
- "initial_point" (list[float]): punto inicial que se tomará para la minimización de la función
- "u_range" ([float, float]): el intervalo que se generará en el gráfico en el eje x
- "v_range" ([float, float]): el intervalo que se generará en el gráfico en el eje y
- "stroke_width" (float): grosor de las líneas en el gráfico a generar
- "cycles" (int): cantidad de iteraciones máximas que se desean realizar para obtener una aproximación del mínimo valor.

Ejemplo:

```
{
    "vars": ["x", "y"],
    "func": "- (7 * x * y / 2.71828 ** ( x ** 2 + y ** 2))",
    "initial_point": [1, 1],
    "u_range": [-5, 5],
    "v_range": [-5, 5],
    "stroke_width": 0.5,
    "cycles": 500
}
```

La idea central en esta función es representar la función a minimizar como centro de la animación e ilustrar los distintos puntos de la solución alcanzados mediante un recorrido de la lista de soluciones de los distintos métodos.

Búsqueda en la línea

Dado un problema de optimización (no necesariamente lineal) con restricciones, la función `construct` de la escena `ThreeDCanvas` (que se encuentra aquí) se encarga de computar el punto óptimo para el problema especificado en el archivo line_search

Ejemplo de input:

```
{
  "vars": ["x", "y"],
  "func": "x**2 *y",
  "constraints": ["x**2 + y**2 <= 3"],
  "initial_point": [1, 1],
  "x_range": [-2,2],
  "y_range": [-2,2],
  "camera_phi": 45,
  "camera_theta": -45

}
```

Se deben especificar las variables a usar, así como las restricciones con signo <=. La función a optimizar debe estar escrita en términos que permitan minimizarla, es decir, se debe utilizar el opuesto de los coeficientes si se desea hallar el máximo. También se debe especificar un punto inicial, así como el rango de *x* y *y* que se desea graficar. Opcionalmente, debido a que muchas veces las funciones obstruirán los puntos que se desean ver, o directamente obstruirán la cámara en su totalidad, se pueden especificar los ángulos de rotación para la cámara centrada en la escena.

Primeramente se procesarán los datos del problema especificados en el archivo de input, el cual por defecto será model_ls. Se creará un objeto de tipo `Surface` para graficar la función principal con los valores de `x_range` y `y_range` dados. Además, se colocará la cámara en el ángulo especificado, y se hará rotar a la misma con respecto a la escena, debido a que en ocasiones la complejidad de las funciones con las que se trabaja hará muy tedioso el proceso de ajustar la cámara a un ángulo específico para observar los puntos óptimos computados.

Para hallar el valor óptimo mediante la estrategia de búsqueda en la línea, utilizamos como dirección de descenso siempre el opuesto al gradiente de la función evaluada en el punto inicial, llamémosle a este valor D. Luego utilizando la función `scipy.line_search` (vea aquí para más información), calculamos el valor de α óptimo por el cual debemos multiplicar el opuesto del gradiente, y posteriormente nos movemos en esa dirección `p*alpha` unidades, de manera que si el punto del q partimos es p_0, avanzaríamos hasta $p_1=p_0+\alpha D$. Si el punto que obtenemos siguiendo esta idea cumple con todas las restricciones del problema, si no se detecta un descenso infinito, y si la diferencia entre el k-ésimo punto y el k+1-ésimo punto computado es mayor que cierta tolerancia, actualizamos el valor del gradiente que utilizaremos para la siguiente iteración, y seguimos el ciclo en la línea 78, en caso contrario, graficamos los puntos resultantes luego de obtener sus coordenadas con respecto al eje de coordenadas de tres dimensiones que utilizamos como punto de referencia para la escena.

Penalización:

Método de Penalización:

Este método reemplaza la función objetivo con restricciones por una función sin restricciones, la cual es formada por:

- La función objetivo.
- Un término adicional por cada restricción, que es positivo si el punto actual *x* viola las restricciones o *0* de otro modo.

La mayoría de las formas de este método usan un coeficiente positivo, el cual se utiliza para multiplicar la parte de la función de las restricciones. Haciendo que este coeficiente sea más grande se penaliza más fuerte cada vez la violación de las restricciones para llegar más cerca a la región factible para el problema restringido.

Estos son considerados `exterior_penalty_method`, dado que el término de penalización en la función para cada restricción no es *0* cuando *x* es no factible con respecto a esa restricción.

El método cuadrático de penalización es el que cada término de penalización es elevado al cuadrado. Además para las restricciones de igualdad y desigualdad el término de penalización tienen distintas formas:

- igualdad: h(x) => $(h(x))^2$
- desigualdad: g(x) => $max(0, g(x))^2$

$\Phi(x)$

Detalles del código:

Todo el proceso para hallar las intersecciones así como el mínimo y el máximo están en el archivo **penalty_newton.py** en un método llamado **penalty_newton**.

El método se puede describir de la siguiente forma:

Dados el coeficiente de penalización (μ_0), una tolerancia ($\tau_0>0$) y un punto inicial (x_0)

- for k = 0, 1,2, . . . encontrar un minimizador aproximado x_k para la función y termina cuando $||\nabla \Phi(x_k)||<= \tau_k$

    - *if* se satisface el test de convergencia final, parar con la aproximacion x_k.
    - *else* aumentar el coeficiente de penalización y coger el punto hallado como nuevo `start_point`.

  end (for)

En el proceso se va modificando la función de penalización dado que el coeficiente por el que se multiplican los términos de penalización varía.

Al final se devuelve un diccionario con la cantidad de iteraciones realizadas, los puntos intermedios y una lista con los valores de **x**, **y** y el resultado de evaluar la función en estos (**z**).

Luego el proceso para graficar con Manim se encuentra en el archivo **Penalty_Manim.py** donde primero se realiza el llamado a **penalty_newton** para obtener todos los datos necesarios y luego se procede a graficar en orden, el eje de coordenadas, la función y los puntos resultantes.

Por último, los detalles como la función y las restricciones se almacenan en un archivo **penalty_settings.json** el cual se utiliza como configuración del problema a resolver.

Se utilizó como nombre de variables *"x"* y *"y"*.

Ejecutar Manim

Si desea ejecutar manim directamente, sin necesidad de utilizar la aplicación visual; solo necesita generar el archivo `json` correspondiente para llevar a cabo la acción y deberá ejecutar el siguiente comando en la terminal, en la dirección raíz del proyecto:

```
manim -pql main.py <nombre_de_la_clase>
```

En el código de `main.py` podrá encontrar las clases que realizan cada acción, solo deberá sustituir su nombre en el comando anterior. Esto generará un archivo multimedia con el resultado de su cómputo en la dirección `"media/videos/main/480p15/<nombre_de_la_clase>.mp4"`

El flag **-ql** es para la calidad, la **l** significa low; existen otros como **m**, **h**, **k** los cuales corresponden a *medium*, *high* y *4k* respectivamente. Se debe tener cuidado al usar estos por el consumo de CPU. También podemos añadirle **-p** al flag anterior quedando **-pql**, como se muestra en este caso, para que al terminar de ejecutar el programa muestre el resultado en un video directamente. Los demás parámetros deben ser el nombre del *.py* a ejecutar y el nombre de la clase que implementa las clases de Manim. El flag `--format=gif` es usado para que el archivo que retorne sea en formato gif y no un video.

Contáctenos

Link del repositorio en github: github.com/codersUP/manim_optimization_models

Desarrolladores:

- Carmen Irene Cabrera Rodríguez - cicr99
- Enrique Martínez González - kikeXD
- Andy A. Castañeda Guerra - Yumenio
- Richard García de la Osa - Regnod