Documentação extra simulador

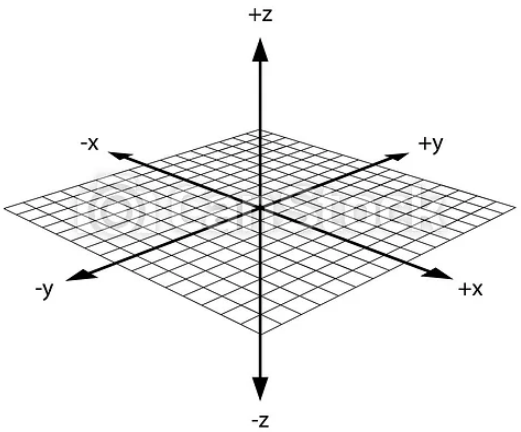
ATENÇÃO: Esse documento tenta explicar de maneira geral mas suficiente o funcionamento de classes e métodos chave para a compreensão da simulação, porém podem existir detalhes faltando.

Por isso uma documentação **completa** foi feita nos próprios arquivos da pasta core/physics, caso esteja procurando alguma informação que falte aqui leia a documentação do método/classe correspondente. No mais boa leitura ;)

Noções iniciais

Sistema de coordenadas

O sistema principal (inercial) adotado é o do matplotlib, que é representado na figura abaixo



O sistema local de coordenadas do foguete segue a mesma orientação, porém seu eixo z é sempre paralelo a direção em que o foguete está apontando (orientação). Isso é possível porque o sistema de coordenadas local rotaciona junto com o corpo.

Pontos de aplicação de forças

Quando uma força é aplicada no centro de massa/gravidade (CG) de um corpo livre ele translada, já quando a força é aplicada em um ponto diferente do CG ele translada e rotaciona ao mesmo tempo.

Classes (maior abstração para menor)

Simulation (core/physics/simulation.py)

```python
class Simulation:
    def __init__(self, rigid_body:RigidBody):
        self.__DELTA_TIME = 1
        self.__rigid_body = rigid_body
        self.__forces = []
```

É responsável pela coordenação da simulação física, essa classe salva os estados do rigid_body e muda ele iterativamente até o tempo determinado.

addForce(force: Force)

Adiciona uma força a lista de forças que serão usadas na simulação.

__applyForces(current_state: DeltaTimeSimulation)

Itera sobre toda a lista de forças atualizando seus valores com base no estado atual (current_state).

simulate(time: int)

Roda a simulação física até o tempo determinado pelo parâmetro time com intervalos de __DELTA_TIME, ou seja, se time = 5 e __DELTA_TIME = 1 (valor padrão) os cálculos serão feitos para os tempos 0,1,2,3,4 e 5 segundos.

A cada segundo simulado o método salva o "estado atual" do rigid_body no valor do dicionário delta_time_simulations (sendo a chave o segundo correspondente), e aplica as forças especificadas no corpo. Esse processo iterativo se repete até o segundo especificado por "time".

Entende-se o "estado atual" como sendo os valores de velocidade, posição, aceleração etc… para aquele segundo, foi criada uma classe DeltaTimeSimulation para armazenar esses dados.

Depois as forças são aplicadas no corpo e consequentemente mudam o seu estado atual (que será salvo na próxima iteração).

RigidBody (core/physics/body/rigid_body.py)

```python
class RigidBody:
    def __init__(self, delimitation_points:list, mass:float, volume:float, moment_of_inertia:float, cp:Vector):
        # variaveis que são definidas fora do escopo do classe
        self.__delimitation_points = delimitation_points # lista de vetores que limitam o corpo (apenas 2, topo e base)
        self.__volume = volume
        self.__mass = mass # também é variável de estado
        self.__moment_of_inertia = moment_of_inertia # também é variável de estado
        self.__cp = cp # também é variável de estado, mas tem tratamento diferente (estruturas)
        self.__cg = Vector(0,0,0) # também é variável de estado, mas tem tratamento diferente (estruturas)

        # variaveis de estado
        self.__velocity = Vector(0,0,0)
        self.__total_acceleration = Vector(0,0,0)
        self.__total_displacement = Vector(0,0,0)
        self.__angular_velocity = Vector(0,0,0)
        self.__total_angular_displacement = Vector(0,0,0)

        # sistema de coordenadas local
        self.__coordinate_system = BodyCoordinateSystem()
        self.__validate()
```

Classe central do programa, é responsável pela simulação física em si. Essa classe é capaz de translacionar e rotacionar o corpo, sendo que o "corpo" é entendido como o conjunto formado pelo centro de massa, centro de pressão e os pontos que delimitam a base e o topo do foguete.

Todos os dados relativos a características do corpo como massa, volume, centro de pressão e etc devem ser fornecidos à classe. É de EXTREMA importância ao subsistema de estruturas notar que o centro de gravidade é centrado na origem (0,0,0) no instante inicial, então todos os vetores que representam o corpo devem levar isso em consideração.

Não é preciso entender os detalhes mínimos de implementação das funções dessa classe, os títulos dos métodos já descrevem bem a sua funcionalidade.

DeltaTimeSimulation (core/physics/delta_time_simulation.py)

```python
class DeltaTimeSimulation:
    def __init__(self, rigid_body, time:int):
        self.cg = rigid_body.cg()
        self.cp = rigid_body.cp()
        self.velocity = rigid_body.velocity()
        self.acceleration = rigid_body.acceleration()
        self.angular_velocity = rigid_body.angularVelocity()
        self.mass = rigid_body.mass()
        self.looking_direction = rigid_body.getLookingDirection()
        self.time = time
```

É responsável por armazenar os dados do estado atual do rigid_body, para que depois esses possam ser usados na plotagem de gráficos.

Force -> herda Vector (core/physics/forces/force.py)

```python
class Force(Vector, ABC):
    def __init__(self, x, y, z, application_point:ApplicationPoint, cg_offset:float=None):
        super().__init__(x, y, z)
        self.__application_point = application_point
        self.__cg_offset = cg_offset # valores positivos -> acima do cg, valores negativos -> abaixo do cg
        self.__validate()
```

Classe abstrata que fornece todas as funcionalidades necessárias para a criação de forças variáveis (ou constantes) customizadas, sendo necessário somente a implementação de seu método calculate(). Force herda Vector, o que a faz ter todas as funcionalidades de um vetor como será explicado mais para frente.

Além das coordenadas (x,y,z) do vetor é necessário especificar o application_point, que nada mais é que o ponto do foguete onde a força será aplicada. Existem 3 opções de pontos de aplicação:

Centro de massa/gravidade = ApplicationPoint.CG
Centro de pressão = ApplicationPoint.CP
Customizável = ApplicationPoint.CUSTOM

É necessário dar um valor para a variável opcional cg_offset quando ApplicationPoint.CUSTOM é especificado. cg_offset representa a distância do ponto

de aplicação até o centro de massa, tendo valor positivo quando o ponto de aplicação está acima do CG e negativo quando está abaixo.

Exemplos:
Considere que a força seja constante (método calculate() não altera nada).

test_force = TestForce(10,0,0, ApplicationPoint.CG) -> Força de 10 N aplicada no CG
(translação)

test_force = TestForce(10,0,0, ApplicationPoint.CP) -> Força de 10 N aplicada no CP
(rotação e translação)

test_force = TestForce(10,0,0, ApplicationPoint.CUSTOM, cg_offset=1) -> Força de 10 N aplicada 1 m **acima** do CG.
(rotação e translação)

test_force = TestForce(10,0,0, ApplicationPoint.CUSTOM, cg_offset=-1) -> Força de 10 N aplicada 1 m **abaixo** do CG.
(rotação e translação)

calculate(current_state: DeltaTimeSimulation)

Método abstrato que deve ser implementado nas classes que herdam Force, esse método é chamado em Physics.applyForces() iterativamente e sua função é calcular os valores de (x,y,z) e talvez application_point com base no estado atual.

Por exemplo, a força peso depende da gravidade (constante para nosso caso) e da massa do corpo (que varia ao longo do vôo), essa força em específico tem valores de (x,y) nulos pois sempre aponta para baixo. Então o método calculate() de WeightForce deve calcular o valor do peso e igualar o eixo z ao resultado:

```
def calculate(self, current_state: DeltaTimeSimulation):
    weight_magnitude = current_state.mass * Constants.GRAVITY.value
    self.setZ(weight_magnitude)
    self.setX(0)
    self.setY(0)
```

Sendo que:
Constants.GRAVITY.value = -9.8.
current_state.mass representa a massa do corpo naquele instante de tempo.

Os detalhes de como a implementação desse método deve ser feita vão variar para cada caso, mas de modo geral se deve chamar self.setX(), self.setY(), self.setZ() para mudar os valores da força e pegar os dados que variam a partir do current_state.
Sendo importante dizer que que os valores de (x,y,z) dados ao instanciar a classe são os valores iniciais que podem ou não ser mudados ao longo do tempo, mais detalhes serão dados a frente.

Para um exemplo mais geral de como criar uma força veja a classe TranslationTestForce, e para um mais específico veja WeightForce.

Vector

```
class Vector:
    def __init__(self, x, y, z):
        self.__x = x
        self.__y = y
        self.__z = z
```

Classe que implementa toda a funcionalidade de vetores, incluindo a capacidade de realizar contas vetoriais.

magnitudeRelativeTo(root: Vector)

Retorna a magnitude do vetor em relação a outro vetor root, a principal diferença desse método é que o sinal da magnitude retornada é negativo se os dois vetores estiverem em sentidos contrários (diferente da magnitude normal que é sempre positiva).

__mul__(), __add__(), __sub__()

Esses métodos permitem a realização do produto por escalar, adição e subtração. **NÃO** é necessário chamar-los em nenhum momento já que o Python os chama automaticamente quando são usados os operadores correspondentes (*, +, -). Uma limitação de __mul__() é que o número do produto por escalar deve vir **depois** do vetor, se for colocado antes um erro aparecerá.
Exemplos:

vec1= Vector(2,2,1)
vec2 = Vector(3,3,0)

```
print(vec1 + vec2) # (5,5,1)
print(vec1 - vec2) # (-1,-1,1)

print(vec1 * 2) # (4,4,2)
print(2 * vec1) # TypeError: unsupported operand type(s) for *: 'int' and Vector'
```

BodyCoordinateSystem (core/physics/body/body_coordinate_system.py)

```
class BodyCoordinateSystem():
    def __init__(self):
        self.__x_axis = Vector(1,0,0)
        self.__y_axis = Vector(0,1,0)
        self.__z_axis = Vector(0,0,1)
```

Classe responsável por gerenciar o sistema de coordenadas do foguete, sua função é rotacionar o sistema de coordenadas a medida que o foguete rotaciona, a utilidade disso é poder ter o vetor que representa a direção que o foguete está apontando (orientação) de maneira fácil. Mais funcionalidades específicas ao subsistema de estruturas serão adicionadas a essa classe no futuro.

ApplicationPoint

```
class ApplicationPoint(Enum):
    CG = 1
    CP = 2
    CUSTOM = 3
```

O único propósito dessa classe é servir para verificação em condicionais if, o importante é a chave e não o valor. No geral, essa classe só será usada para identificar qual é o tipo de ponto de aplicação.

Constants (utils/constants.py)

```
class Constants(enum.Enum):
    GRAVITY = -9.8
```

Classe que armazena valores de constantes comuns que são usadas em várias partes do programa, podem ser adicionadas novas constantes se necessário.

Paths (utils/paths.py)

Classe que armazena os caminhos relativos de pastas que são usadas em várias partes do programa, podem ser adicionados novos caminhos se necessário.

```
class Paths(enum.Enum):
    MATERIALS = "data/materials"
    PARACHUTES = "data/parachutes"
    PROPELLANTS = "data/propellants"
```