Simulação Computacional de Curvas

Daniel Ruhman e Marcelo Terreiro Prado

O projeto a seguir foi realizado para a disciplina de Matemática Multivariada ministrada pelo Professor Fabio Orfali no curso de Engenharia do INSPER. Ele tem como objetivo o desenvolvimento de uma simulação computacional baseada em métodos numéricos (mais especificamente a aproximação de uma curva por uma linha poligonal composta por *n* segmentos) para calcular o tempo aproximado que um objeto, sujeito apenas à força da gravidade, leva para percorrer uma trajetória dada por curvas parametrizadas e pontos iniciais e finais ou domínio.

Dedução geométrica da parametrização da cicloide

Considere uma circunferência C de raio *r* com um ponto P, fixo. Ao rolar a circunferência sobre uma reta (eixo x), o ponto P percorre uma curva chamada de ciclóide. Queremos descobrir quanto a curva "caminhou" em cada eixo. A seguir, apresentamos a dedução geométrica para sua parametrização, partindo dos seguintes pressupostos:

- $\theta = 0$ no início, gira θ radianos;
- o ponto P coincide com a origem do sistema de coordenadas no início do movimento;

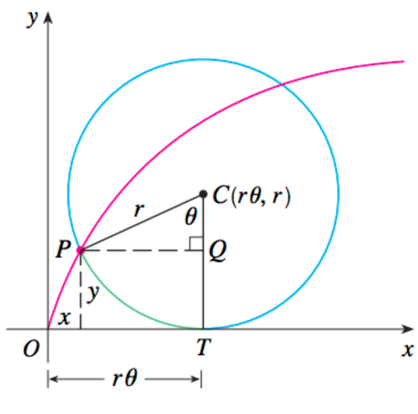


Figura 1: Explicação da dedução geométrica

Podemos então escrever:

$$x = \mathrm{OT} - \mathrm{PQ} \text{ (eq. 1)}$$

$$y = \mathrm{TC} - \mathrm{QC} \text{ (eq. 2)}$$

Com os deslocamentos de x e y em mãos, precisamos descobrir os segmentos de reta para determinar as parametrizações em relação ao parâmetro θ. Comecemos por |OT| . Como C está em contato com a reta, deduzimos que:

$$|OT| = comprimento\ do\ arco\ \widehat{PT} = r \cdot \theta$$

Podemos deduzir pela figura as medidas dos outros 3 segmentos restantes:

$$\mathrm{TC} = r$$

$$sen\theta = \frac{\mathrm{PQ}}{r} \rightarrow \mathrm{PQ} = r \cdot sen\theta$$

$$cos\theta = \frac{\mathrm{QC}}{r} \rightarrow \mathrm{QC} = r \cdot cos\theta$$

Agora resta substituir as medidas dos segmentos encontrados nas equações 1 e 2 e obtemos a parametrização final:

$$x = r(\theta - \text{sen}\ \theta) \qquad y = r(1 - \cos\theta) \qquad \theta \in \mathbb{R}$$

Modelo Computacional

O modelo desenvolvido pelo grupo encontra-se explicado abaixo. Inicialmente, declaramos as variáveis que irão armazenar as parametrizações, além de definir alguns parâmetros e algumas burocracias de código. Também definimos o domínio (ex: de 0 a 2π) das parametrizações.

```
def retornaParametrizacao (t):
    # Parametrização em X
    xParam = np.cos(t)
    # Parametrização em Y
    yParam = np.sin(t)
    return (xParam, yParam)


dominioMin = np.pi
dominioMax = (3*np.pi) / 2
```

```
v0 = 0
tTotal = 0
distTotal = 0
precisao = 0.01
delta = np.arange(dominioMin,dominioMax, precisao)
gravidade = 9.81

xMax, yMax = retornaParametrizacao(dominioMin)
listaX = []
listaY = []
```

Figura 2: Função responsável por calcular a parametrização da curva

Depois, declaramos uma função cujo objetivo é achar os comprimentos dos segmentos de reta que usaremos para aproximar a curva. Esses segmentos são, como mostra a imagem, a hipotenusa entre dois pontos da curva. A diferença entre esses pontos é a precisão do cálculo. Quanto menor a diferença entre eles, melhor a aproximação. A função da figura 4 recebe os pontos para os quais calcular o segmento de reta, obtidos através da função da figura 3. Além disso, ela retorna o ângulo de inclinação das retas que irá nos ajudar a calcular o tempo.

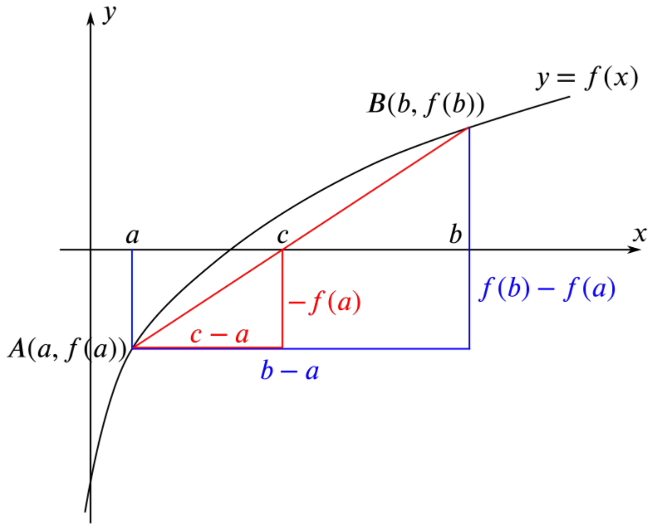


Figura 3: Hipotenusa como aproximação da curva

```
def findHipotenusa (x0, x1, y0, y1):
    deltaX = x1 - x0
    deltaY = y1 - y0
    # Em radiano
    angulo = np.abs(np.arctan(deltaY/deltaX))
    return np.sqrt((deltaX**2) + (deltaY**2)), angulo
```

Figura 4: Função que encontra a aproximação de um trecho da curva

Então, declaramos uma função que irá nos retornar o tempo de percurso de cada segmento de reta. Ela recebe como parâmetros o ângulo de inclinação, a distância (hipotenusa) e a velocidade inicial. E ela retorna o tempo de percurso e a velocidade final (que será utilizada como inicial para a próxima iteração, e assim por diante). Para calcular o tempo, utilizamos a equação:

$$s = s_0 + v_0 . t + \frac{1}{2} . a . t^2$$

Equação 1

Como já possuimos os parâmetros *ΔS* (comprimento do segmento/hipotenusa), v_0 (inicialmente 0, depois igual à velocidade final no segmento anterior) e *a* (aceleração da gravidade), só precisamos isolar o *t*. A Figura 5 representa essa equação em um código de uma função.

```python
# Formula usada
# dist = v0*t + (a*(t**2))/2
def retornaTempo (teta, dist, v0):
    a = gravidade * np.sin(teta)
    delta = v0**2 + 2*a*dist

    if (delta < 0):
        print('err0')
        return (0,0)
    else:
        t1 = (-v0 + np.sqrt(delta))/a
        t2 = (-v0 - np.sqrt(delta))/a
        v = v0 + a*t1
        return (t1,v)
```

Figura 5: Função que encontra o tempo que leva para a bolinha percorrer determinado trecho e sua velocidade final

Por fim, contruímos o loop que junta todas essa funções (figura 6). Ele roda para cada intervalo de precisão definido, e tem o seguinte comportamento:

- Primeiro, descobre as coordenadas dos dois pontos do segmento aproximado por um segmento de reta, com base na precisão pré definida
- Depois, descobre o comprimento desse segmento de reta (hipotenusa) e o seu ângulo de inclinação, usando seu ponto inicial e final.
- Então, calcula o tempo necessário para percorrer tal segmento e o adiciona ao tempo total para percorrer a curva.
- Isso é repetido para cada intervalo de precisão definido, até percorrer a curva inteira.

```python
consegueSubir = True

for t in delta:

    x0, y0 = retornaParametrizacao(t)

    listaX.append(x0)
    listaY.append(y0)
    x1, y1 = retornaParametrizacao(t + precisao)

    if (y1 > yMax):
        consegueSubir = False

    hip, angulo = findHipotenusa(x0,x1,y0,y1)
    distTotal += hip
    t1, v = retornaTempo(angulo, hip, v0)
    v0 = v
    tTotal += t1

if consegueSubir:
    print("Tempo total (s)")
    print(tTotal)
    print('Distância total (m)')
    print(distTotal)

else:
    print('A bolinha não consegue subir essa curva. O Y máximo é {0}'.format(yMax))

plt.plot(listaX,listaY)
plt.title('Figura 1: Quarto de circunferncia')
plt.show()
```

Figura 6: Loop responsável por juntar todas as peças

Por fim, imprime os valores (nesse caso, **um quarto de circunferência**):

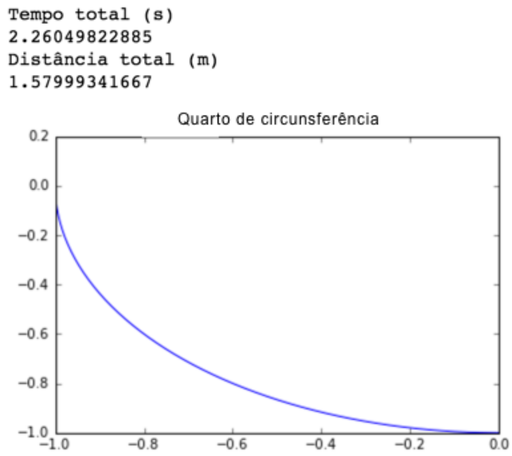


Figura 7: Curva de um quarto de circunferência

Validação

Abaixo está a validação da nossa simulação computacional. Nela, utilizamos uma reta parametrizada. Olhando seu domínio, fica claro que a distância percorrida faz sentido. Utilizando pitágoras, pode-se perceber que a distância vale raiz quadradada de 200, o que bate com nosso resultado.

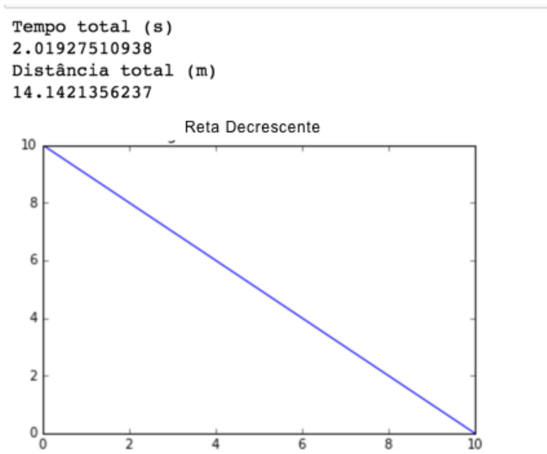


Figura 8: Validação de uma reta decrescente

Figura 9: Validação de uma reta crescente

Vale ressaltar que nossa implementação também considera o caso da bolinha não ter energia suficiente para subir a curva (Figura 9).

Para o cálculo do tempo, utilizamos uma reta vertical definida com x constante. Em seguida, calculamos o tempo que levaria para a bolinha percorrer o trajeto inteiro e validamos utilizando a física. Esse tempo precisa ser igual ao tempo necessário para ela cair em queda-livre.

Tivemos bastante dificuldade **na primeira versão do relatório** para validar o tempo que a bolinha leva para percorrer determinada curva. Abaixo estão descritas as tentativas realizadas.

1. Calcular o tempo que a bolinha leva para percorrer uma reta. Tivemos problemas para calcular o tempo teórico que levaria (tanto pela literatura matemática quanto pela física), e por isso acabamos trocando de tentativa.

2. Calcular o tempo que a bolinha leva para percorrer uma reta vertical, com x constante. Da mesma forma que na tentativa acima, tivemos problemas. Dessa vez foi com o código. Ele reclamava de nossa equação utilizada na função `retornaTempo`, provavelmente devido a maneira com que dividimos a curva e distribuímos as forças. Sabendo esse tempo, utilizaríamos a fórmula da cinemática de posição em função do tempo para checarmos por valores iguais. Acreditamos ser por conta da utilização de `cosseno` e não `seno` na função `retornaTempo`. Entretanto, não conseguimos fazer a função funcionar com o `seno`.

3. Descobrir o tempo real de uma ciclóide e comparar com o tempo calculado pela simulação. Conversamos nosso colega (Eduardo Ferrari) e testamos com a ciclóide construída por ele. Tivemos muita dificuldade para medir o tempo, que da menos de 1 segundo na rampa construída. Além disso, tivemos dificuldade em descobrir os parâmetros exatos da ciclóide construída para equacionar uma semelhante.

Porém, para a reentrega conseguimos resolver os problemas. Utilizamos o seno na decomposição de forças e, percebemos que para calcular o angulo da inclinação de cada segmento, seria necessário utilizar o módulo de DeltaY/DeltaX. Feito isso, o modelo foi corrigido e agora se encontra de acordo com a realidade. Segue abaixo a validação física-teórica para o caso de uma reta decrescente.

Primeiro, criamos os parâmetros de nossa reta. Seu comprimento é $\sqrt{200}$ (aproximadamente 14,14) e sua inclinação é 45º. Consideramos $v_0 = 0$ e $\Delta s = 14,14$. Em seguida, calculamos o valor da aceleração, utilizando a seguinte lógica:

$$a = g.seno(45º) = 9.81.sen(45º) = 6,93m/s^2$$

Por fim, aplicamos esses valores na *Equação 1*:

$$14,14 = \frac{6,93.t^2}{2}$$

$$t = \sqrt{\frac{28,28}{6,93}} \simeq 2,02s$$

Comparando com o tempo simulado da figura 8, pode-se perceber que os dois são muito próximos. Portanto, nosso modelo se comporta de maneira adequada com o esperado.

Plots de Curvas Parametrizadas

A partir da função definida na figura 2, `retornaParametrizacao`, foi possível experimentar diversos modelos de curvas para testar nossa simulação. Abaixo estão exemplos de plots que fizemos para algumas curvas parametrizadas. As imagens à esquerda são as funções utilizadas e o domínio. As imagens à direita são o resultado do plot.

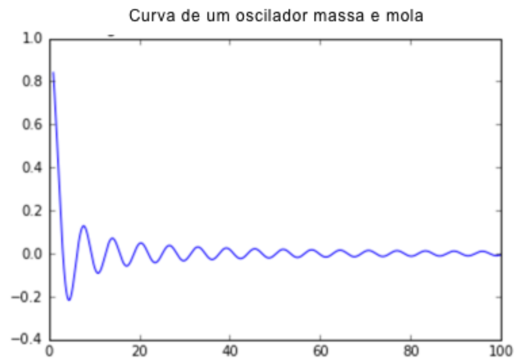
1. Curva de um oscilador massa-mola

```python
def retornaParametrizacao (t):
    # Parametrização em X
    xParam = t
    # Parametrização em Y
    yParam = np.sin(t) / t
    return (xParam, yParam)

dominioMin = 1
dominioMax = 100
```

Parametrização de um oscilador

```
Tempo total (s)
4.54522367649
Distância total (m)
99.2361320764
```



Curva de um oscilador massa-mola

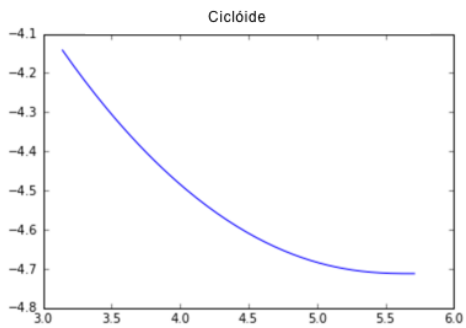
2. Curva de uma ciclóide

```python
def retornaParametrizacao (t):
    # Parametrização em X
    xParam = (t - np.sin(t))
    # Parametrização em Y
    yParam = -(t -np.cos(t))
    return (xParam, yParam)

dominioMin = np.pi
dominioMax = 3*np.pi/2
```

Parametrização de uma cicloíde

```
Tempo total (s)
0.767815164148
Distância total (m)
2.66911308936
```



Curva de uma ciclóide

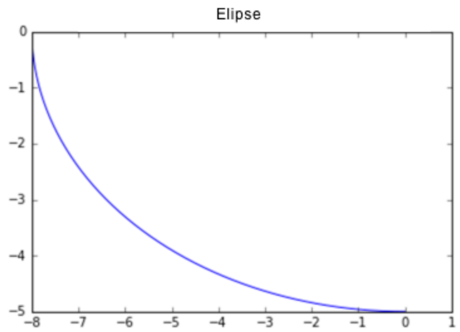
3. Curva de uma elipse

```
def retornaParametrizacao (t):
    # Parametrização em X
    xParam = 8 * np.sin(t)
    # Parametrização em Y
    yParam = 5 * np.cos(t)
    return (xParam, yParam)


dominioMin = np.pi
dominioMax = 3*np.pi/2
```

Parametrização de uma elipse

```
Tempo total (s)
1.48227279029
Distância total (m)
10.3925447961
```



Curva de uma elipse

Conclusões

Um dos objetivos do projeto era identificar se a bolinha caía mais rápido em uma ciclóide do que em outra curva qualquer. Comparando as figuras 7 e o Item 2 acima, percebemos que de fato a ciclóide apresenta um menor tempo de queda, com 0.767 segundos em comparação aos 2.260 segundos da circunferência. Entretanto, precisamos analisar os eixos. Tivemos problemas na hora de plotá-las com ponto final e inicial definidos, já que nosso código foi estruturado com base em domínios. Dessa forma, não podemos comparar seus tempos com exatidão.

O modelo computacional criado é válido. Inicialmente, tivemos algumas dificuldades em validá-lo usando a física. Porém, após uma segunda tentativa com a ajuda do Professor Fábio Orfali, corrigimos o erro inicial e agora o modelo representa melhor a realidade.