Minicurso: Simulações com VPython
VIII FISICANDO

Prof. João Teles de Carvalho Neto
Gabriel Antonio Caritá (monitor)
João José Ambrozetto (monitor)

Sumário

1 Objetivos

Objetivos gerais

- Mostrar o potencial que o **Visual Python (VPython)** possui para o ensino de Ciências.
- A duração do minicurso não permite exlorar todas as ferramentas do VPython. Pretendemos, portanto, apresentar as ferramentas e princípios básicos, indicando várias fontes de consulta para quem quiser se aprofundar mais.
- Esperamos que o VPython possa ser uma porta de entrada para o universo de ferramentas de simulações, as quais são tão importantes para às atividades científicas e para os processos de ensino aprendizagem que envolvem fenômenos de mais difícil visualização.

Objetivos específicos

- Apresentar a biblioteca de simulações **Visual Python (VPython)** e suas principais utilidades.
- Apresentar a plataforma **GlowScript** que permite rodar as simulações em VPython *online*.
- Explorar os elementos geométricos básicos do VPython em modo estático.
- Produzir dinâmicas e simulações simples com o VPython, visando principalmente o ensino e a aprendizagem de Física (mas pode ser aplicado a muitas outras áreas).
- Explorar os *widgets* que permitem a interação do usuário.
- Mostrar as diferentes formas de compartilhar as simulações.

2 A linguagem de programação Python

Principais vantagens

- Linguagem de programação interpretada, orientada à objeto e de código aberto, que possibilita uma sintaxe mais amigável e facilidade na depuração dos códigos.
- Extensa documentação: inúmeros livros, tutoriais, cursos on-line e projetos comentados.
- Gigantesco conjunto de bibliotecas aplicadas aos mais diversos escopos: e.g.: **numpy** para calculo numérico, **sympy** para matemática simbólica, **astropy** para astronomia, **selenium** para motores web, **vpython** para simulações, etc.
- Fortíssima comunidade engajada em abarcar novas aplicações ainda inexploradas, aprimorar e compartilhar seus códigos.

Desvantagens

- Lentidão em rodar códigos próprios que contenham muitas execuções cíclicas (*loops*). Para isso, existe a possibilidade de escrever códigos em C e transformá-los em Python ou transformar os códigos em Python para a linguagem C (e.g. Cython).
- Por ter um desenvolvimento muito dinâmico, pode deixar o usuário um pouco perdido com relação às diferentes versões em andamento (e.g. as versões 2.7 ou 3.5 do Python apresentam pequenas incompatibilidades que necessitam ser levadas em conta).

Sugestões de documentação e cursos online

- https://python.org
- https://python.org.br/introducao/
- https://www.tutorialspoint.com/python/
- https://www.youtube.com/user/11Wills11/playlists

3 Apresentação da biblioteca Visual Python

Características gerais

- "Programação 3D para simples mortais". "O VPython facilita a criação de animações e *displays* 3D navegáveis, mesmo para aqueles com pouca experiência em programação. Por ser baseado em Python, também tem muito a oferecer para programadores e pesquisadores experientes." `https://vpython.org/`
- Possui uma série de elementos geométricos prontos: `https://www.glowscript.org/docs/VPythonDocs/primitives.html`
- Possui várias ferramentas de interação com o usuário: `https://www.glowscript.org/docs/VPythonDocs/controls.html`
- Permite a construção de gráficos dinâmicos de vários tipos: `https://www.glowscript.org/docs/VPythonDocs/graph.html`
- Permite a construção de arranjos 3D estáticos.
- Animações com movimentos pré-estabelecidos.
- Simulações via discretização de equações diferenciais.

Exemplos de uso do VPython

- Livro de Física Básica *Matéria e Interações*: `https://www.glowscript.org/#/user/GlowScriptDemos/folder/matterandinteractions/program/MatterAndInteractions`

- *Physics Simulations in Python*, Daniel Schroeder: `http://physics.weber.edu/schroeder/scicomp/PythonManual.pdf`

Sugestão de documentação e vídeos online

- Documentação online: `https://www.glowscript.org/docs/VPythonDocs/index.html`
- Vídeos instrucionais: `https://www.glowscript.org/docs/VPythonDocs/videos.html`
- Tutorial em pdf: `https://www.glowscript.org/docs/VPythonDocs/VPython_Intro.pdf`
- *Physics Simulations in Python*, Daniel Schroeder: `http://physics.weber.edu/schroeder/scicomp/PythonManual.pdf`

4 Utilização da Plataforma Glowscript

Para que serve?

"O GlowScript é um ambiente poderoso e fácil de usar para criar animações em 3D e publicá-las na web. Em glowscript.org você pode escrever e executar programas GlowScript diretamente no seu navegador, armazená-los na nuvem gratuitamente e compartilhá-los facilmente com outras pessoas."

Como usar?

- Acesse o site do GlowScript: glowscript.org
- Clique em *Sign In* no canto superior direito.
- Use sua conta do Google para fazer *log in*. Caso contrário, crie uma conta Google.
- Caso o *log in* tenha funcionado, deverá aparecer a informação *Signed in as "**seu login**" (Sign out)* no canto superior direito da tela.
- Clique no *"**seu login**"* para acessar a sua área de arquivos.
- Clique em **Add Folder** para criar uma pasta, de forma a organizar melhor seus arquivos. Ao nomear a pasta, desmarque a opção *Public* caso queira que os arquivos contidos nela sejam mantidos privados.
- Clique em *Create New Program* para iniciar a escrita do seu programa em VPython.

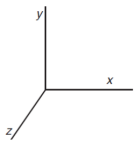
5 Geometrias e operações básicas do VPython

Criando um cubo e alterando a visualização da cena gerada

- Após criar um novo programa, digite: `box()`. Em seguida clique em *Run this program* para executar o programa ou clique Ctrl + 1.
- Para girar o ângulo de visão da cena, mantenha o botão direito do mouse apertado e arraste-o.
- Para alterar o *zoom* da cena, use o botão de rolagem do mouse.
- Para deslocar lateralmente a visão da cena, arraste o mouse com o Shift e o botão esquerdo apertados.
- Atente-se que todas essas operações não deslocam nem giram o objeto, mas apenas o seu ângulo de visão.

Alterando propriedades dos objetos

- A visualização da cena é descrita pelo sistema de coordenadas abaixo, em que o eixo z aponta para fora da tela:



- Posições e deslocamentos são dados pela função `vec`. Exemplo: `v1 = vec(x, y, z)`
- Criemos um cubo, uma esfera e uma seta ligando um ao outro:

```
1 pos_cubo = vec(-2,-2,-2)
2 pos_bola = vec(2,2,2)
3 cubo = box(pos=pos_cubo, size=vec(1,2,3),
4         color=color.green)
5 bola = sphere(pos=pos_bola, radius=0.7,
6         color=color.cyan)
7 seta = arrow(pos=pos_cubo, axis=pos_bola-pos_cubo)
```

- Se quisermos alterar qualquer propriedade do objeto criado basta usar: `variavel_objeto.propriedade = valor`.
- Exemplo 1 - alterar a cor da seta: `seta.color = color.yellow`
- Exemplo 2 - alterar a posição do cubo: `cubo.pos = vector(-2.5,-1,-3)`
- Se quiséssemos sempre vincular a seta ao cubo e à bola, deveríamos ter criado a seta como: `seta = arrow(pos=cubo.pos, axis=bola.pos-cubo.pos)`

Algumas operações com vetores

- Criemos dois vetores:

```
vX = vec(1,0,0)
vY = vec(0,1,0)
setaX = arrow(axis = vX, color = color.blue)
setaY = arrow(axis = vY, color = color.red)
```

- Calculemos o produto vetorial entre eles:

```
vZ = cross(vX, vY)
setaZ = arrow(axis = vZ, color = color.green)
```

Façamos a decomposição do vetor v1 nas componentes paralela e perpendicular ao vetor v2:

```
v1 = vec(-1.0, 3.1, 1.5)
v2 = vec(3.0, 3.0, 2.0)
v1pa = v1.proj(v2) #projeção de v1 na direção de v2
v1pe = v1 - v1pa   #componente de v1 perpendicular a v2
arrow(axis = v1, color = color.green, shaftwidth = 0.3)
arrow(axis = v2, color = color.blue, shaftwidth = 0.3)
arrow(axis = v1pa, color = color.red, shaftwidth = 0.3)
arrow(axis = v1pe, color = color.cyan, shaftwidth = 0.3)

a1 = vertex(pos = vec(0,0,0))
a2 = vertex(pos = v1pa)
a3 = vertex(pos = v1)
a4 = vertex(pos = v1pe)
quad(vs = [a1,a2,a3,a4])
```

6 Animações com movimentos pré-estabelecidos

Exemplo: sistema massa-mola amortecido

```
mesa = box(pos=vec(0,0,-0.15), size=vec(3,2,0.3), color=color.cyan)
apoio = box(pos=vec(1.35,0,0.25), size=vec(0.3,2,0.5), color=color.cyan)

bloco = box(pos=vec(0,0,0.25), size=vec(0.5,0.5,0.5), color=color.red)
mola = helix(pos=apoio.pos, axis=bloco.pos-apoio.pos,
                          radius=0.2, coils = 10, color=color.orange)

T = 1.0         #Período de oscilação em segundos
tc = 10.0       #tempo característico de decaímento em segundos
N = 30          #número de amostragens por período
xm = 0.8        #amplitude inicial da oscilação

w = 2*pi/T      #frequência de oscilação [rad/s]
dt = T/N        #tamanho do passo temporal da animação
t = 0.0
while True:
    sleep(dt)
    x = xm*exp(-t/tc)*cos(w*t)
    t = t + dt
    bloco.pos = vec(x,0,0.25)
    mola.axis = bloco.pos-apoio.pos
```

Inclusão de um gráfico para amostrar o movimento

Adicione ao início do programa:

```
s = 'Gráfico do deslocamento do sistema massa-mola.'
grafico = graph(title=s, xtitle='tempo [s]', ytitle='Amplitude [u.a.]',
                          fast=True, width=800)
curva = gcurve(color=color.blue, width=4, markers=False,
                          marker_color=color.orange, label='curve')
```

E ao final do laço `while`:

```
curva.plot(t, x)
```

O código completo do sistema massa-mola fica:

```
s = 'Gráfico do deslocamento do sistema massa-mola.'
grafico = graph(title=s, xtitle='tempo [s]', ytitle='Amplitude [u.a.]',
                          fast=True, width=800)
curva = gcurve(color=color.blue, width=4, markers=False,
                          marker_color=color.orange, label='curve')

mesa = box(pos=vec(0,0,-0.15), size=vec(3,2,0.3), color=color.cyan)
apoio = box(pos=vec(1.35,0,0.25), size=vec(0.3,2,0.5), color=color.cyan)

bloco = box(pos=vec(0,0,0.25), size=vec(0.5,0.5,0.5), color=color.red)
mola = helix(pos=apoio.pos, axis=bloco.pos-apoio.pos,
                          radius=0.2, coils = 10, color=color.orange)

T = 1.0         #Período de oscilação em segundos
tc = 10.0       #tempo característico de decaímento em segundos
N = 30          #número de amostragens por período
xm = 0.8        #amplitude inicial da oscilação

w = 2*pi/T      #frequência de oscilação [rad/s]
dt = T/N        #tamanho do passo temporal da animação
t = 0.0
while True:
    sleep(dt)
```

```
24      x = xm*exp(-t/tc)*cos(w*t)
25      t = t + dt
26      bloco.pos = vec(x,0,0.25)
27      mola.axis = bloco.pos-apoio.pos
28      curva.plot(t, x)
```

7 Simulações via discretização de equações diferenciais

Sistema massa-mola a partir da 2ª lei de Newton

Podemos incluir um texto descritivo, inclusive usando LaTeX:

```
MathJax.Hub.Queue(["Typeset",MathJax.Hub]) #Comando para Latex
scene.caption = '''Simulação do sistema massa-mola a partir da equação diferencial
do movimento: \\(m\\frac{dv}{dt}=-kx-bv\\)
Espera-se observar o comportamento previsto pelas soluções analíticas:
    \\( \\omega_{0}=\\sqrt{k/m}\\), \\( \\gamma=\\frac{b}{2m}\\)
    1) Regime subamortecido:    \\(\\omega_{0}>\\gamma\\)
    2) Regime crítico:          \\(\\omega_{0}=\\gamma\\)
    3) Regime superamortecido: \\(\\omega_{0}<\\gamma\\)'''
```

As declarações geométricas dos objetos continuam como antes:

```
s = 'Gráfico do deslocamento do sistema massa-mola.'
grafico = graph(title=s, xtitle='tempo [s]', ytitle='Amplitude [u.a.]',
                        fast=True, width=800)
curva = gcurve(color=color.blue, width=4, markers=False,
                        marker_color=color.orange, label='curve')

mesa = box(pos=vec(0,0,-0.15), size=vec(3,2,0.3), color=color.cyan)
apoio = box(pos=vec(1.35,0,0.25), size=vec(0.3,2,0.5), color=color.cyan)

bloco = box(pos=vec(0,0,0.25), size=vec(0.5,0.5,0.5), color=color.red)
mola = helix(pos=apoio.pos, axis=bloco.pos-apoio.pos,
                        radius=0.2, coils = 10, color=color.orange)
```

As variáveis dinâmicas serão calculadas numericamente usando a 2ª lei de Newton:

- $m\dfrac{\Delta v}{\Delta t} = F$
- $m\left(v_n - v_{n-1}\right) = F\Delta t$
- $\boxed{v_n = v_{n-1} + \dfrac{F\Delta t}{m}}$
- $\dfrac{\Delta x}{\Delta t} = v$
- $x_n - x_{n-1} = v_n\Delta t$
- $\boxed{x_n = x_{n-1} + v_n\Delta t}$
- É necessário fornecer as condições iniciais: x_0 e v_0

A implementação da dinâmica pode ser escrita assim:

```
bloco.massa = 1.0    #massa do bloco em [kg]
mola.k = 30.0        #constante elática da mola em [N/m]
bloco.b = 1.0        #coeficiente de arrasto [N.s/m]

x0 = 0.8         #posição inicial do bloco [m]
v0 = 0.0         #velocidade inicial do bloco [m/s]

print('w0 = '+str(sqrt(mola.k/bloco.massa))+ ' rad/s')
print('gama = '+str(bloco.b/(2*bloco.massa))+ ' rad/s')

dt = 0.01            #passo temporal [s]
t = 0.0
x = x0
v = v0
while True:
    sleep(dt)
    bloco.pos = vec(x,0,0.25)
    v += -(mola.k*x + bloco.b*v)*dt/bloco.massa
    x += v*dt
    t = t + dt
    mola.axis = bloco.pos-apoio.pos
    curva.plot(t, x)
```

- Um dos interesses principais na simulação de fenômenos a partir das equações diferenciais está na possibilidade de testar diversos modelos de interação, muitos dos quais não possuem solução analítica.

- Por exemplo, no sistema massa-mola, poderíamos utilizar uma força de atrito que fosse função de outras potências da velocidade ao invés de uma dependência puramente linear.

- De forma geral, poderíamos substituir a expressão para v no código anterior por:

```
v += -(mola.k*x + bloco.b*abs(atrito(v))*v/abs(v))*dt/bloco.massa
```

- Em que `atrito(v)` é uma função qualquer que depende da velocidade v e pode ser declarada anteriormente ao laço `while`. Como exemplo, para uma dependência quadrática em v, teríamos:

```
def atrito(v):
        return v**2
```

- A função `abs(x)` retorna o módulo de `x`. A forma como ela é usada aqui garante que a força de atrito seja sempre oposta a direção da velocidade, independente da paridade da função `atrito(v)`.

Exemplo de controles interativos: *botões*

Vamos incluir três botões: **Reiniciar**, **Pausar** e **Continuar**

```
rodando = 1                         #flag do estado de execução

def Pausar(b):              #função ligada a Pausar
    global rodando
    rodando = 0

def Continuar(b):           #função ligada a Continuar
    global rodando
    rodando = 1

def Reiniciar(b):           #função ligada a Reiniciar
    global rodando
    rodando = 2

button(text="Pausar", pos=scene.title_anchor, bind=Pausar)
button(text="Continuar", pos=scene.title_anchor, bind=Continuar)
button(text="Reiniciar", pos=scene.title_anchor, bind=Reiniciar)
```

Vamos colocar um condicional no laço de execução:

```
while True:
    sleep(dt)
    if rodando > 0:
        if rodando == 2:
            t = 0.0
            x = x0
            v = v0
            curva.delete()
            rodando = 1
        bloco.pos = vec(x,0,0.25)
        v += -(mola.k*x + bloco.b*v)*dt/bloco.massa
        x += v*dt
        t = t + dt
        mola.axis = bloco.pos-apoio.pos
        curva.plot(t, x)
```

Exemplo de controles interativos: controles deslizantes

Criar as funções que alteram o valor dos parâmetros m, k e b:

```
def setmassa(m):
    bloco.massa = m.value
    m_text.text = 'Massa = '+'{:1.1f}'.format(m.value)+' kg\n'
    calc_amort()

def setk(k):
    mola.k = k.value
    k_text.text = 'k = '+'{:1.1f}'.format(k.value)+' N/m\n'
    calc_amort()

def setb(b):
    bloco.b = b.value
    b_text.text = 'b = '+'{:1.1f}'.format(b.value)+' N.s/m\n'
    calc_amort()

def calc_amort():
    w0_text.text='w0 = '+'{:1.2f}'.format(sqrt(mola.k/bloco.massa))+' rad/s\n'
    g_text.text='gama = '+'{:1.2f}'.format(bloco.b/(2*bloco.massa))+' rad/s\n\n'
```

Criar os controles deslizantes dos parâmetros m, k e b:

```
scene.append_to_caption('\n\n')
s_massa = slider(min=0.1, max=10.0, value=bloco.massa, length=220, bind=setmassa,
                 right=15)
m_text = wtext(text='Massa = '+'{:1.1f}'.format(s_massa.value)+' kg\n',
                 pos=scene.caption_anchor)

s_k = slider(min=1, max=50, value=mola.k, length=220, bind=setk, right=15)
k_text = wtext(text='k = '+'{:1.1f}'.format(s_k.value)+' N/m\n',
               pos=scene.caption_anchor)
```

```
10
11 s_b = slider(min=0, max=10, value=bloco.b, length=220, bind=setb, right=15)
12 b_text = wtext(text='b = '+'{:1.1f}'.format(s_b.value)+' N.s/m\n\n',
13                 pos=scene.caption_anchor)
14
15 w0_text = wtext(text='w0 = '+'{:1.2f}'.format(sqrt(mola.k/bloco.massa))+
16                  ' rad/s\n', pos=scene.caption_anchor)
17 g_text = wtext(text='gama = '+'{:1.2f}'.format(bloco.b/(2*bloco.massa))+
18                  ' rad/s\n\n', pos=scene.caption_anchor)
```

8 Compartilhamento das simulações em VPython

Através do site do GlowScript

- Entre no arquivo ou pasta do GlowScript que deseja compartilhar.
- Clique em *Share or export this program*.
- Copie o link gerado no primeiro item. É algo como https://www.glowscript.org/#/user/login/caminho, em que *login* é o seu nome de login e *caminho* é o nome do arquivo ou pasta que está sendo compartilhado.
- Quem tiver acesso a esse link poderá executar a simulação no GlowScript sem precisar fazer login, desde que o arquivo esteja definido como *Public*.

Distribuindo o código html para rodar localmente

- Copie o código gerado ao clicar em *Share or export this program*.
- Cole o código em um editor txt e salve com extensão html. Exemplo: codigo.html.
- Abra o arquivo codigo.html com o seu navegador de internet favorito.
- A simulação deverá rodar tranquilamente. Obs.: testei exclusivamente com o Firefox e funcionou.

Embutindo a simulação no seu site pessoal

- Copie o código gerado ao clicar em *Share or export this program*.
- Cole o código na página html do seu site.
- Exemplo 1: crie um site pessoal no Google e cole o código html em uma página do site utilizando a opção *Incorporar*.
- Exemplo 2: crie uma página html no Moodle e cole o código html.

9 Sugestões de outras plataformas de simulação

- **Easy Java Simulations (EJS)** (`https://www.um.es/fem/EjsWiki/`): permite criar simulações independetes que podem rodar sozinhas ou serem incorporadas a *websites*. Possui organização de variáveis, elementos geométricos e widgets que podem ser configurados através de uma interface gráfica. Também permite a criação de documentação sobre a simulação produzida. Permite a solução de equações diferenciais por métodos numéricos sofisticados. Pode-se incluir códigos em Java para o caso de simulações mais elaboradas. É um dos simuladores de física mais completos.
- **GeoGebra**

Obrigado pela participação e boas simulações a tod@s!!!