Dokumentation Python Simulation: Schiefer Wurf

February 2024

Contents

1 Einleitung

In der folgenden Dokumentation wird ein Wurf mit einem Objekt, wie beispielsweise einem Ball, näherungsweise beschrieben. Dabei werden alle relevanten physikalischen Einflüsse auf den Wurf analysiert und entsprechend abgeleitet. Anschließend wird der Wurf mithilfe einer Simulation in der Programmiersprache Python umgesetzt. Hierbei kommen die Bibliotheken Matplotlib und NumPy zum Einsatz.

Diese Simulation zielt darauf ab, die Flugbahn eines Balls unter Berücksichtigung von Anfangsgeschwindigkeit und Abwurfwinkel zu approximieren. Dabei werden Fragen beantwortet wie die Flugweite, maximale Höhe, resultierende Geschwindigkeit sowie kinetische und potenzielle Energie des Objekts.

Innerhalb des Programmcodes können Parameter wie der Abwurfwinkel, die Anfangsgeschwindigkeit, die Masse des Objekts und die Abwurfhöhe variabel eingestellt werden. Das grafische Interface ermöglicht zudem das Pausieren der Simulation.

2 Physikalische Analyse

Beim Abwurf eines Objekts, wie in diesem Fall eines Balls, sei der Abwurfwinkel in Grad und die anfängliche Abwurfgeschwindigkeit bekannt. Die Abwurfgeschwindigkeit kann in zwei Komponenten aufgeteilt werden: eine gleichförmige Bewegung in horizontaler Richtung und eine vertikale Bewegung. Dabei wirkt die Erdanziehungskraft F_g auf den Ball, die ihn in entgegengesetzter Richtung zur anfänglich vorhandenen vertikalen Komponente beschleunigt. Die Geschwindigkeit v_{res} ergibt sich aus der Vektoraddition von v_x und v_y. Diese Vektoren bilden zusammen ein rechtwinkliges Dreieck. Da der Abwurfwinkel α bekannt ist, können wir mithilfe der trigonometrischen Beziehungen aus der resultierenden Geschwindigkeit v_{res} und dem Abwurfwinkel die jeweiligen Geschwindigkeitskomponenten berechnen.

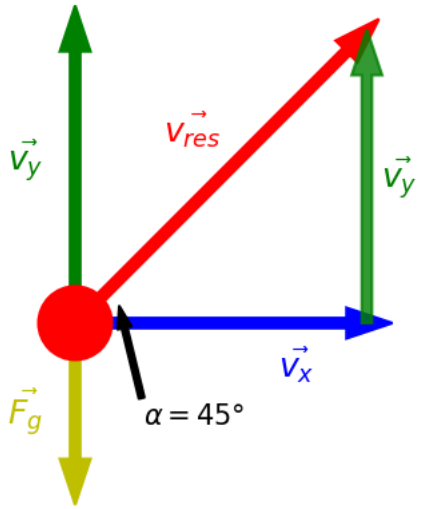

Figure 1: Skizze: Schiefer Wurf

$$v_{y0} = v_{res} \cdot sin(\alpha)$$

$$v_{x0} = v_{res} \cdot cos(\alpha)$$

Für die Strecke, die der Ball in horizontaler Richtung zurücklegt, gilt die Formel für gleichförmige Bewegungen. Dabei trägt nur die Geschwindigkeitskomponente v_x zur Bewegung bei.

$$s_x = v_{x0} \cdot t$$

Für die vertikalen Geschwindigkeitskomponenten gilt, dass die Anfangsgeschwindigkeit v_y pro Zeiteinheit in gegensätzlicher Richtung beschleunigt wird. Somit ergibt sich folgende Gleichung:

$$v_y = v_{y0} - g \cdot t$$

Für die Strecke, die eine gleichmäßig beschleunigte Bewegung zurücklegt, gilt allgemein folgende Beziehung:

$$h = \frac{1}{2} \cdot a \cdot t^2$$

Für die Höhe des Balls wirkt die vertikale Anfangsgeschwindigkeit und die gleichmäßig beschleunigte Bewegung in entgegengesetzte Richtungen ein. Zusätzlich hat die Start-Höhe, von der der Ball geworfen wird, ebenfalls eine Auswirkung.

All diese Faktoren ergeben folgende Gleichung:

$$h = -\frac{1}{2} \cdot g \cdot t^2 + v_{y0} \cdot t + h_{start}$$

Hierbei handelt es sich um eine quadratische Gleichung. Die Nullstelle dieser Gleichung gibt den Zeitpunkt tt in Sekunden an, zu dem der Ball auf den Boden aufkommt.

Für die kinetische Energie des Balls gilt:

$$E_{kin} = \frac{1}{2} \cdot m \cdot v_{res}^2$$

Analog dazu gilt für die potenzielle Energie:

$$E_{pot} = \frac{1}{2} \cdot m \cdot h$$

3 Python Code

Für die Simulation des Wurfs wurde eine Klasse namens "Wurf" entwickelt, welche alle relevanten Größen berechnet und verwaltet. Diese Klasse bietet eine strukturierte und effiziente Möglichkeit, die physikalischen Aspekte des Wurfs zu modellieren und zu untersuchen.

3.1 Klassenvariablen

Für die Klassenvariablen wurden folgende Konstanten gewählt: Die Erdbeschleunigung g mit $9,81\frac{m}{s^2}$, das Intervall, das angibt, wie viele Millisekunden zwischen zwei Bildern der Animation liegen sollen, und die Anzahl an Berechnungspunkten nn. Dies wurde so festgelegt, damit der Benutzer diese Konstanten nicht eigenständig beeinflussen kann.

```
1 class wurf():
2     g = 9.81
3     inervall = 6
4     n = 300
```

Listing 1: Klassenvariablen

3.2 Konstruktor

Im Konstruktor werden die Variablen für den Abwurfwinkel, die Abwurfgeschwindigkeit, die Masse des Objektes und die Abwurfhöhe festgelegt. Darüber hinaus werden die Geschwindigkeitskomponenten in x- und y-Richtung berechnet.

```
1 def __init__(self, a0_neu, v0_neu, m_neu, h_start_neu):
2     self.a0 = np.radians(a0_neu)
3     self.v0 = v0_neu
4     self.m = m_neu
5     self.h_start = h_start_neu
6     self.vx0 = np.sin(self.a0) * self.v0
7     self.vy0 = np.cos(self.a0) * self.v0
```

Listing 2: Konstruktor

3.2.1 Berechnung wann der Ball auf den Boden aufkommt

Im Konstruktor wird zusätzlich berechnet, zu welchem Zeitpunkt das Objekt den Boden erreicht. Dies erfolgt durch die Anwendung der im ersten Kapitel hergeleiteten Gleichung für die Höhe, die auf Null gesetzt wird, um die Nullstellen dieses Polynoms zu bestimmen.

In Python wird dies mithilfe der Numpy-Funktion poly1d umgesetzt, gefolgt von der Verwendung von np.roots zur Berechnung der Nullstellen des Polynoms.

```
1 coeff = [(-1)*(1/2)*self.g, self.vy0, self.h_start]
2 h1 = np.poly1d(coeff)
3 self.tstop = np.roots(h1)
```

Listing 3: Stop zeit

3.2.2 Berechnung wann der Ball am höchsten ankommt

Im Konstruktor wird ebenfalls festgelegt, wann das Objekt den höchsten Punkt erreicht. Hierzu wird die zuvor hergeleitete Gleichung für die Höhe einmal abgeleitet und anschließend die Nullstellen berechnet.

Für diese Aufgabe wird erneut Numpy verwendet. Zur Ableitung eines Polynoms in Numpy wird die Funktion np.polyder(Funktion, m=1) verwendet, wobei m die Ordnung der Ableitung angibt.

```
1  h1_ab1 = np.polyder(coeff, m=1)
2  self.t_hmax = np.roots(h1_ab1)
```

Listing 4: Maximale Höhe

3.3 Achschenbeschränkung des Wurfs

Um die Achsenbeschränkungen für das Koordinatensystem des Wurfs zu berechnen, benötigt man sowohl die maximale Strecke als auch die maximale Höhe, die der Ball erreicht.

Die maximale Strecke wird mithilfe der Formel für gleichförmige Bewegung bestimmt, indem man für die Zeit tt die berechnete Zeit einsetzt, zu der der Ball den Boden erreicht.

Die maximale Höhe wird erreicht, wenn die vertikale Geschwindigkeit, die in Richtung Erde wirkt, genau so groß ist wie die anfängliche vertikale Geschwindigkeitskomponente des Balls $v_{y0} = v_{a0}$. Dies ergibt sich aus dem Gesetz für gleichmäßig beschleunigte Bewegungen:

$$t = \frac{v_{y0}}{g}$$

Dieser Zusammenhang, eingesetzt in die Berechnungsgleichung für die Höhe, ergibt:

$$\mathrm{h} = v_{y0} \cdot (\frac{v_{y0}}{g} - \frac{1}{2} \cdot g \cdot (\frac{v_{y0}}{g})^2) + h_{start}$$

```
1 def achsen_ber(self):
2         sxmax = self.vx0 * self.tstop[0]
3         hmax = self.vy0 * (self.vy0/self.g) - (1/2) * self.g *
4           (self.vy0/self.g)**2 + self.h_start
5         return sxmax, hmax
```

Listing 5: Achschenbeschränkung Wurf

3.4 Berechnung der Wurf-Parabel

Um die Flugbahn des Objekts zu berechnen, werden die Gleichungen für die Höhe h und die Strecke sx aus dem ersten Kapitel verwendet.

```
1 def wurf_berechnung(self, t):
2         h = self.vy0 * (t) - (1/2) * self.g * (t)**2 + self.h_start
3         sx = self.vx0 * (t)
4         return h, sx
```

Listing 6: Berechnung Wurf

3.5 Berechnung des Geschwindigkeitsvektors in y-Richtung

Um den Vektor in vertikaler Richtung zu berechnen, wird zunächst der Betrag des Vektors ermittelt, wobei die entsprechende Formel bereits im vorherigen Kapitel hergeleitet wurde. Anschließend wird der Startpunkt des Vektors berechnet, der genau der Position entspricht, an der sich der Ball zum Zeitpunkt tt befindet. Dazu wird der Endpunkt des Vektors benötigt, der ebenfalls dem Betrag des Vektors entspricht, da der vertikale Vektor keine x-Koordinate besitzt.

```
1  def vek_ber_y(self, t):
2      vy_vektor = self.vy0 - self.g*t
3      v_start_y = self.vy0 * (t) - (1/2) * self.g * (t)**2 + self.h_start
4      v_start_x = self.vx0 * t
5      vy_ende_y = vy_vektor
6      vy_ende_x = 0
7
8      return v_start_y, v_start_x, vy_ende_y, vy_ende_x
```

Listing 7: Berechnung y-Richtung

3.6 Berechnung des Geschwindigkeitsvektors in x-Richtung

Analog dazu wird mit dem Vektor in horizontaler Richtung verfahren:

```
1 def vek_ber_x(self, t):
2     v_start_y = self.vy0 * (t) - (1/2) * self.g * (t)**2 + self.h_start
3     v_start_x = self.vx0 * t
4     v_ende_y = 0
5     v_ende_x = self.vx0
6
7     return v_start_y, v_start_x, v_ende_y, v_ende_x
```

Listing 8: Berechnung x-Richtung

3.7 Berechnung des Resultierenden Geschwindigkeitsvektors

Für den resultierenden Geschwindigkeitsvektor wird analog wie zuvor verfahren. Zusätzlich wird der Betrag der resultierenden Geschwindigkeit berechnet und als Rückgabewert der Funktion übergeben.

```
1  def vek_ber_res(self, t):
2      v_start_y = self.vy0 * (t) - (1/2) * self.g * (t)**2 + self.h_start
3      v_start_x = self.vx0 * t
4      vres_ende_x = self.vx0
5      vres_ende_y = self.vy0 - self.g*t
6      v_res = vres_ende_x + vres_ende_y
7
8      return v_start_y, v_start_x, vres_ende_x, vres_ende_y, v_res
```

Listing 9: Berechnung Resultierenden Geschwindigkeit

3.8 Berechnung der Kenetischen Energie

Um die kinetische Energie des Balls zu berechnen, wird die allgemeine Formel für kinetische Energie angewendet. Dabei werden für jeden neuen Zeitpunkt tt auch die Punkte aus der Vergangenheit berechnet, um einen kontinuierlich erweiterten Graphen zu erzeugen.

```
1 def Ekin_ber(self, t):
2     x = np.linspace(0, t, int(25*t))
3     V_res = np.sqrt((self.vx0)**2 + (self.vy0 - self.g*x)**2)
4     Ekin = (1/2) * self.m * (V_res)**2
5     Ekin_textfeld = (1/2) * self.m * (np.sqrt((self.vx0)**2 + (self.vy0 - self.g
6
7     return Ekin, x, Ekin_textfeld
```

Listing 10: Berechnung Kenetischen Energie

3.9 Bestimmung des Startwertes und Achsenbeschränkungen der Kenetischen Energie

Um die kinetische Energie zum Startzeitpunkt zu berechnen, wird der Zeitpunkt t=0t=0 in die Gleichung für die kinetische Energie eingesetzt. Somit ist die maximale kinetische Energie auch zu Beginn vorhanden.

Die minimale kinetische Energie wird erreicht, wenn der Ball den Boden erreicht hat, da er ab diesem Zeitpunkt nicht mehr in Bewegung ist.

```
1  def Ekin_ber_achsen(self):
2      Ekin_start = (1/2) * self.m * (self.vx0 + self.vy0)**2
3      Ekin_min = (1/2) * self.m * (self.vx0 + (self.vy0 -
4     self.g*self.tstop[0]))**2
5      Ekin_max = Ekin_start
6
7      return Ekin_start, Ekin_min, Ekin_max
```

Listing 11: Achsenbeschränkung Kenetischen Energie

3.10 Berechnung der Potenziellen Energie

Analog dazu wird für die Potenziale Energie verfahren.

```
1 def Epot_ber(self, t):
2     x = np.linspace(0, t, int(25*t))
3     h = self.vy0 * (x) - (1/2) * self.g * (x)**2 + self.h_start
4     Epot = (1/2) * self.m * h**2
5     Epot_textfeld = (1/2) * self.m * (self.vy0 * (t) - (1/2) * self.g * (t)**2 +
6
7     return x, Epot, Epot_textfeld
```

Listing 12: Berechnung Potenzielle Energie

3.11 Bestimmung des Startwertes und Achsenbeschränkungen der Potenziellen Energie

Analog zur Kentischen Energie:

```
1  def Epot_ber_achsen(self):
2      h_max = self.vy0 * (self.t_hmax[0]) - (1/2) * self.g *
3     (self.t_hmax[0])**2 + self.h_start
4      Epot_max = (1/2) * self.m * (h_max)**2
5      Epot_start = (1/2) * self.m * self.h_start
6
7      return h_max, Epot_max, Epot_start
```

Listing 13: Achsenbeschränkung der Potenziellen Energie

3.12 Erstellung der Animation

Die Animationen werden alle durch eine Funktion gestartet und geupdatet.

```
def animate_wurf(self, t):

    h, sx = self.wurf_berechnung(t)
    ball.set_data(sx, h)

    v_start_y, v_start_x, vy_ende_y, vy_ende_x = self.vek_ber_y(t)
    vecy.set_offsets([v_start_x, v_start_y])
    vecy.set_UVC(vy_ende_x, vy_ende_y)

    v_start_y, v_start_x, v_ende_y, v_ende_x = self.vek_ber_x(t)
    vecx.set_offsets([v_start_x, v_start_y])
    vecx.set_UVC(v_ende_x, v_ende_y)

    v_start_y, v_start_x, vres_ende_x, vres_ende_y, v_res = self.vek_ber_res(t)
    vecres.set_offsets([v_start_x, v_start_y])
    vecres.set_UVC(vres_ende_x, vres_ende_y)

    Ekin, x_kin, Ekin_textfeld = self.Ekin_ber(t)
    line_ekin.set_data(x_kin, Ekin)

    x_pot, Epot, Epot_textfeld = self.Epot_ber(t)
    line_epot.set_data(x_pot, Epot)

    h_txt.set_text("h=" + str(np.round(h, decimals=2)) + "m")
    sx_txt.set_text("Strecke=" + str(np.round(sx, decimals=2))
    + "m")
    Epot_txt.set_text("Epot=" + str(np.round(Epot_textfeld,
    decimals=2)) + "Joul")
    Ekin_txt.set_text("Ekin=" + str(np.round(Ekin_textfeld,
    decimals=2)) + 'Joul')
    vres_txt.set_text("vres=" + str(np.round(v_res, decimals=2))
    + "m/s")

    return ball, vecy, vecx, vecres, line_ekin,
    line_epot, h_txt, sx_txt, Epot_txt, Ekin_txt, vres_txt,
```

Listing 14: Erstellung der Animation

3.13 Plot erstellen

Es wird vorerst ein Figure erstellt, der 10 Zoll Bereit und 9 Zoll Lang ist. Zusätzlich wird dieses Figure in 18 Zeilen und 20 Spalten eingeteilt.

```
1 fig = plt.figure(figsize=(10,9))
2 gs = fig.add_gridspec(18,20)
```

Listing 15: Plot erstellen

3.13.1 Wurf-Plot

Der Wurf Plot geht von Zeile 6 bis 13 und von Spalte 0 bis 10. Für das Objekt wird durch ax.plot() ein Runder Roter Marker in der Größe 15 verwendet.

```
1 sxmax, hmax = wurf_instanz1.achsen_ber()
2
3 ax = fig.add_subplot(gs[6:13, 0:10])
4 ax.axis([0,sxmax+0.5,0,hmax+0.5]) # Achsen Skalierung
5 ax.set_title("Wurf")
6 ball, = ax.plot([], [], 'ro', ms=15)
```

Listing 16: Wurf-Plot

Geschwindigkeitsvektoren

Die Vektoren werden jeweils mit der Funktion ax.quiver() erstellt.

```
# Vektor in y-Richtung erstellen

vecy = ax.quiver([0],[wurf_instanz1.h_start],[0],
[wurf_instanz1.h_start+wurf_instanz1.vy0], angles='xy',
scale_units='xy', scale=4, color='b', label="Vy in m/s")

# Vektor in x-Richtung erstellen

vecx = ax.quiver([0], [wurf_instanz1.h_start], [0],
[wurf_instanz1.vx0], angles='xy', scale_units='xy', scale=4,
color='g', label="Vx in m/s")

# Resultierender Vektor erstellen

vecres = ax.quiver([0], [wurf_instanz1.h_start], [wurf_instanz1.vx0],
[wurf_instanz1.vy0], angles='xy', scale_units='xy', scale=4,
color='r', label="Vres in m/s")
ax.legend(loc=0)
```

Listing 17: Geschwindigkeitsvektoren

3.13.2 Kenetische Energie Plot

Der Plot der Kenetischen Energie geht von Zeile 6 bis 8 und von Spalte 12 bis 20.

```
1 Ekin_start, Ekin_min, Ekin_max = wurf_instanz1.Ekin_ber_achsen()
2
3 ax = fig.add_subplot(gs[6:8, 12:20])
4 ax.set_title("Kenetische Energie")
5 ax.axis([0, wurf_instanz1.tstop[0]+0.5, Ekin_min-1, Ekin_max+0.5])
6
7 line_ekin, = ax.plot(0, Ekin_start, 'b', lw=2,
8 label="Kenetische Energie in Joul") # Plot der Kenetischen Energie
9 ax.grid(True)
```

Listing 18: Plot Kenetischen Energie

3.13.3 Potenzielle Energie Plot

Der Plot für die Potenzielle Energie geht von der Zeile 11 bis 13 und von der Spalte 12:20.

```
1 h_max, Epot_max, Epot_start = wurf_instanz1.Epot_ber_achsen()
2 ax = fig.add_subplot(gs[11:13, 12:20])
3 ax.set_title("Potenzielle Energie")
4 ax.axis([0, wurf_instanz1.tstop[0]+0.03, 0, Epot_max+0.2])
5
6 line_epot, = ax.plot(0, Epot_start,'r',lw=2,
7 label="Potenzielle Energie in Joul")
8 ax.grid(True)
```

Listing 19: Plot Potenzielle Energie

3.13.4 Plot für die Textfelder

Textfelder für die Energien

Die Polts für die Textfelder geht von der Zeile 16 bis 17 und der Spalte 12 bis 20. Die textfelder werden mit der Funktion ax.text() erstellt, der Rahmen wird in box_s definiert.

```
1  ax = fig.add_subplot(gs[16:17, 12:20])
2  box_s = {'facecolor': 'none',
3           'edgecolor': 'black',
4           'boxstyle': 'square'
5       }
6  ax.set_xticks([])
7  ax.set_yticks([])
8  ax.set_frame_on(False)
9
10 Ekin_txt = ax.text(0.05, 0.3, '', bbox=box_s)
11 Epot_txt = ax.text(0.5, 0.3, '', bbox=box_s)
```

Listing 20: Textfelder Energie

Textfelder für den Wurf

So Analago auch für die Textfelder für den Wurf, allerdings geht diser Plot von Zeile 16 bis 17 und Spalte $0:10$.

```
1 ax = fig.add_subplot(gs[16:17, 0:10])
2 ax.set_xticks([])
3 ax.set_yticks([])
4 ax.set_frame_on(False)
5
6 vres_txt = ax.text(0.05, 0.3, '', bbox=box_s)
7 h_txt = ax.text(0.41, 0.3, '', bbox=box_s)
8 sx_txt = ax.text(0.66, 0.3, '', bbox=box_s)
```

Listing 21: Textfelder Wurf

3.13.5 Animation ausführen

Die Animation ruf die Animate Funktion innerhalb der Klasse Wurf auf.

```
1 ani = FuncAnimation(fig, wurf_instanz1.animate_wurf, frames=t,
2 interval=wurf_instanz1.inervall, blit=True)
3 plt.show()
```

Listing 22: Animation ausführen

3.14 Hinzufügen des Pause/Start Button's

Die Pause und Start Buttons werden initalisiert, indem man die innerhalb von Matplotlib verfügbaren Widgets verwendet.

```
1  ax = fig.add_subplot(gs[0:3, 0:20])
2  ax.set_xticks([])
3  ax.set_yticks([])
4  ax.set_frame_on(False)
5
6  ax_box_pause_button = fig.add_axes([0.1, 0.85, 0.10, 0.07])
7  ax_pause_button = Button(ax_box_pause_button, "Pause")
8  ax_pause_button.on_clicked(clicked_pause_button)
9
10 ax_box_resum_button = fig.add_axes([0.6, 0.85, 0.10, 0.07])
11 ax_resum_button = Button(ax_box_resum_button, "Resum")
12 ax_resum_button.on_clicked(clicked_resum_button)
```

Listing 23: Pause/Start

Die Funktionen für die Button's werden außerhalb der Klasse Wurf definiert.

```
1  def clicked_pause_button(event):
2        ani.pause()
3
4  def clicked_resum_button(event):
5        ani.resume()
```

Listing 24: Button Funktionen

List of Figures

Codeverzeichnis