**Практическая работа 11**

**Тема. Трехмерные графики**

Библиотека Matplotlib главным образом предназначена для создания двумерных графиков, но она поддерживает и функциональные возможности для построения трехмерных графиков, вполне достаточные для множества целей. Простейшим способом создания трехмерного графика является импорт объекта *Axes3D* из модуля *mpl\_toolkits.mplot3d* и определение для аргумента *projection* внутреннего графика значения '3d':

*import matplotlib.pyplot as plt*

*from mpl\_toolkits.mplot3d import Axes3D*

*fig = plt.figure()*

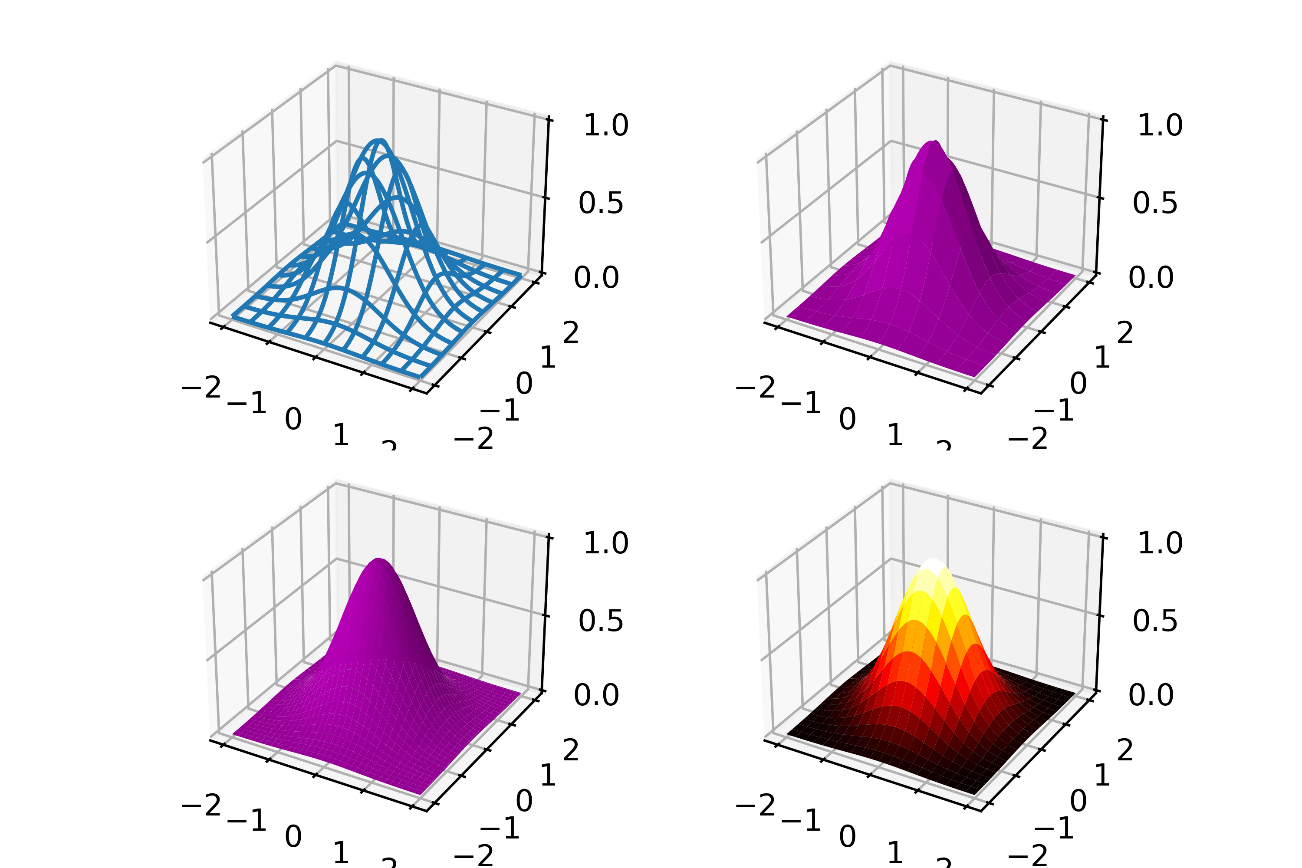
*ax = fig.add\_subplot(projection='3d')*

После этого соответствующий объект *Axes* может отображать данные в трех измерениях как линейный график, точечную диаграмму, каркасную диаграмму или объемную поверхностную диаграмму.

**Каркасные и объемные поверхностные диаграммы**

Простейшим типом объемной поверхностной диаграммы является каркасная диаграмма («проволочная модель»), изображающая линии в трехмерной перспективе, объединяя предоставленный двумерный массив точек *Z* с сеткой значений данных, переданных в двумерных массивах *X* и *Y* (как для методов imshow и *contour*). По умолчанию линии в трехмерном представлении изображаются для каждой точки в массиве, но если точек слишком много, то можно определить значения аргументов *rstride* и *cstride*, чтобы задать шаг изображаемых точек в строках и столбцах массива соответственно. Метод *ax.plot\_surface* работает так же, но создает объемную поверхностную диаграмму из закрашенных элементов (патчей). Для патчей можно установить единый цвет в аргументе *color* или определить стиль с помощью специализированной цветовой схемы в аргументе *cmap*. Для метода *ax.plot\_surface* аргументам *rstride* и *cstride* по умолчанию присваивается значение 10. Практическое применение обоих методов демонстрируется в примере 1.

**Пример 1.** Код в листинге 1 демонстрирует использование разнообразных параметров при создании объемных поверхностных диаграмм. Результаты показаны на рис. 1.



**Рис. 1.** Четыре различные трехмерные поверхностные диаграммы одной функции

**Листинг 2.** Четыре трехмерные поверхностные диаграммы простой двумерной гауссовой функции интерактивном графике направление взгляда можно изменить, если щелкнуть и, не отпуская кнопку мыши, перетаскивать любую точку в области графика. Для фиксации конкретного направления взгляда в статическом изображении графика необходимо явно передать требуемый угол возвышения азимутальный угол (в градусах, в указанном порядке) в метод *ax.view\_init*, как показано в примере 2.

*import numpy as np*

*import matplotlib.pyplot as plt*

*from mpl\_toolkits.mplot3d import Axes3D*

*import matplotlib.cm as cm*

*L, n = 2, 400*

*x = np.linspace(-L, L, n)*

*y = x.copy()*

*X, Y = np.meshgrid(x, y)*

*Z = np.exp(-(X\*\*2 + Y\*\*2))*

*fig, ax = plt.subplots(nrows=2, ncols=2, subplot\_kw={'projection': '3d'})*

*ax[0,0].plot\_wireframe(X, Y, Z, rstride=40, cstride=40)*

*ax[0,1].plot\_surface(X, Y, Z, rstride=40, cstride=40, color='m')*

*ax[1,0].plot\_surface(X, Y, Z, rstride=12, cstride=12, color='m')*

*ax[1,1].plot\_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)*

*for axes in ax.flatten():*

*axes.set\_xticks([-2, -1, 0, 1, 2])*

*axes.set\_yticks([-2, -1, 0, 1, 2])*

*axes.set\_zticks([0, 0.5, 1])*

*fig.tight\_layout()*

*plt.savefig('Simple surface plots.png',dpi=500)*

*plt.show()*

В интерактивном графике направление взгляда можно изменить, если

щелкнуть и, не отпуская кнопку мыши, перетаскивать любую точку в области

графика. Для фиксации конкретного направления взгляда в статическом изображении графика необходимо явно передать требуемый угол возвышения

и азимутальный угол (в градусах, в указанном порядке) в метод *ax.view\_init*,

как показано в примере 2.

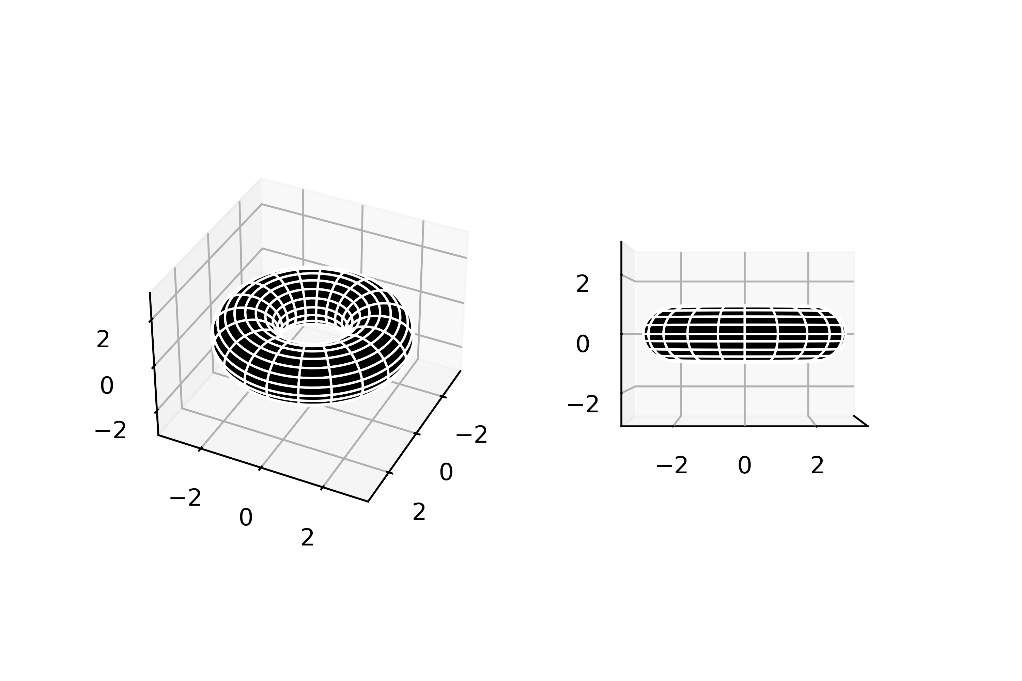
**Пример 2.** Параметрическое определение тора с главным радиусом *c* и радиусом образующей окружности a записывается θ в следующем φ виде:

*x = (c + a cos θ) cos φ*

*y = (c + a cos θ) sin φ*

*z = a sin*

Значения θ и φ должны находиться в интервале от 0 до 2π. Код в листинге 2 выводит два визуальных представления тора, изображаемого как объемная поверхностная диаграмма (см. рис. 2).



**Рис. 2.** Два визуальных представления одного и того же тора: а) ,  б) , 

**Листинг 2.** Трехмерная поверхностная диаграмма тора

import numpy as np

*import matplotlib.pyplot as plt*

*from mpl\_toolkits.mplot3d import Axes3D*

*n = 100*

*theta = np.linspace(0, 2.\*np.pi, n)*

*phi = np.linspace(0, 2.\*np.pi, n)*

*theta, phi = np.meshgrid(theta, phi)*

*c, a = 2, 1*

*x = (c + a\*np.cos(theta)) \* np.cos(phi)*

*y = (c + a\*np.cos(theta)) \* np.sin(phi)*

*z = a \* np.sin(theta)*

*fig = plt.figure()*

*ax1 = fig.add\_subplot(121, projection='3d')*

*ax1.set\_zlim(-3,3)*

*ax1.plot\_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')*

*ax1.view\_init(36, 26)*

*ax2 = fig.add\_subplot(122, projection='3d')*

*ax2.set\_zlim(-3,3)*

*ax2.plot\_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')*

*ax2.view\_init(0, 0)*

*ax2.set\_xticks([])*

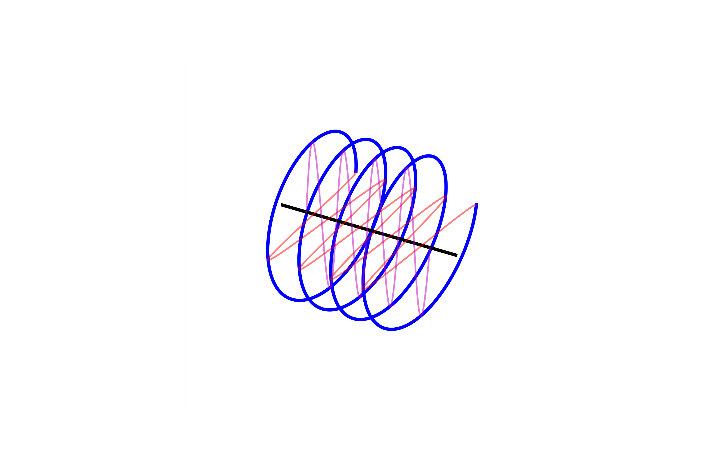
*plt.savefig('A torus.png',dpi=500)*

*plt.show()*

**Линейные графики и точечные диаграммы**

Линейные графики и точечные диаграммы в трех измерениях создаются тем же способом, что и для двух измерений: основные вызываемые методы *ax.plot(x, y, z)* и ax.scatter(x, y, z) соответственно, где *x*, *y*, *z* – одномерные массивы равной длины. Но для таких графиков возможно добавление только ограниченной аннотации, если не пользоваться методами с расширенным набором функциональных возможностей.

**Пример 3.** В листинге 3 показан простой пример трехмерного графика спирали (винтовой линии), которая может представлять, например, свет с круговой поляризацией (см. рис. 3).



**Рис. 3.** Изображение света с круговой поляризацией в виде спирали на трехмерном графике

**Листинг 3.** Изображение спирали на трехмерном графике

*import numpy as np*

*import matplotlib.pyplot as plt*

*from mpl\_toolkits.mplot3d import Axes3D*

*n = 1000*

*fig = plt.figure()*

*ax = fig.add\_subplot(111, projection='3d')*

*# Plot a helix along the x-axis*

*theta\_max = 8 \* np.pi*

*theta = np.linspace(0, theta\_max, n)*

*x = theta*

*z = np.sin(theta)*

*y = np.cos(theta)*

*ax.plot(x, y, z, 'b', lw=2)*

*# An line through the centre of the helix*

*ax.plot((-theta\_max\*0.2, theta\_max \* 1.2), (0,0), (0,0), color='k', lw=2)*

*# sin/cos components of the helix (e.g. electric and magnetic field*

*# components of a circularly-polarized electromagnetic wave*

*ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)*

*ax.plot(x, [0]\*n, z, color='m', lw=1, alpha=0.5)*

*# Remove axis planes, ticks and labels*

*ax.set\_axis\_off()*

*plt.savefig('Depicting a helix.png',dpi=500)*

*plt.show()*

**Анимация**

В этом разделе предлагается краткое введение в практическое использование класса *FuncAnimation* для создания анимированных графиков и диаграмм в программе (скрипте) на языке Python или в виртуальной блокнотной среде Jupyter Notebook. Библиотека Matplotlib предоставляет функциональные возможности анимации в модуле *animation*, который необходимо явно импортировать перед использованием:

*import matplotlib.animation as animation*

**Анимация данных на графике**

**Простая анимированная линия**

Класс *FuncAnimation* создает эффект анимации, многократно вызывая предоставленную функцию *func*, которая обновляет объекты, отображаемые в объекте рисунка *Matplotlib Figure fig*. Дополнительные аргументы описаны в табл. 1.

**Таблица 1.** Аргументы для FuncAnimation

|  |  |
| --- | --- |
| Аргумент | Описание |
| fig | Объект Matplotlib Figure, который необходимо анимировать |
| func | Функция, вызываемая для создания каждого кадра анимации посредством манипуляции объектами в рисунке Figure |
| frames | Источник объекта, передаваемого в функцию func для каждого кадра. Если задано значение  None (по умолчанию), то передается постоянно увеличивающийся целочисленный индекс.  Также можно передавать итерируемые объекты или функцию генератора |
| init\_func | Функция, вызываемая для создания пустого кадра анимации. Если определен аргумент  blit=True, то этот аргумент является обязательным |
| fargs | Любые дополнительные аргументы, передаваемые в функцию func |
| interval | Интервалы времени для паузы между кадрами в мс (по умолчанию 200) |
| repeat | Флаг, логическое значение, определяющее, должна ли анимация зацикливаться (повторяться) или нет (по умолчанию True) |
| blit | Флаг, логическое значение, определяющее, должна ли использоваться операция комбинирования битовых карт изображений для оптимизации анимации (по умолчанию  False). Подробнее см. в тексте |

Объект рисунка *Figure* и содержащиеся в нем объекты *Axes* должны быть созданы до вызова *FuncAnimation*, и любые ссылки на изображаемые объекты необходимо сохранять, чтобы с этими данными могла работать функция анимации. Например, данные *(x, y)*, изображаемые в объекте *Line2D*, могут быть (пере)настроены с помощью собственного метода *set\_data*. Это показано в коде листинга 1, где анимируется затухающая синусоидальная кривая.

**Пример 1.** В коде листинга 1 выполняется анимация затухающей синусоидальной кривой, которая, например, может представлять затухающий звук при ударе по камертону с постоянной (фиксированной) частотой:



**Листинг 1.** Анимация затухающей синусоидальной кривой

*%matplotlib notebook*

*import numpy as np*

*import matplotlib.pyplot as plt*

*import matplotlib.animation as animation*

*# Time step for the animation (s), max time to animate for (s).*

*dt, tmax = 0.01, 5*

*# Signal frequency (s-1), decay constant (s-1).*

*f, alpha = 2.5, 1*

*# These lists will hold the data to plot.*

*t, M = [], []*

*# Draw an empty plot, but preset the plot x- and y-limits.*

*fig, ax = plt.subplots()*

*line, = ax.plot([], [])*

*ax.set\_xlim(0, tmax)*

*ax.set\_ylim(-1, 1)*

*ax.set\_xlabel('t /s')*

*ax.set\_ylabel('M (arb. units)')*

*def animate(i):*

*"""Draw the frame i of the animation."""*

*global t, M*

*# Append this time point and its data and set the plotted line data.*

*\_t = i\*dt*

*t.append(\_t)*

*M.append(np.sin(2\*np.pi\*f\*\_t) \* np.exp(-alpha\*\_t))*

*line.set\_data(t, M)*

*# Interval between frames in ms, total number of frames to use.*

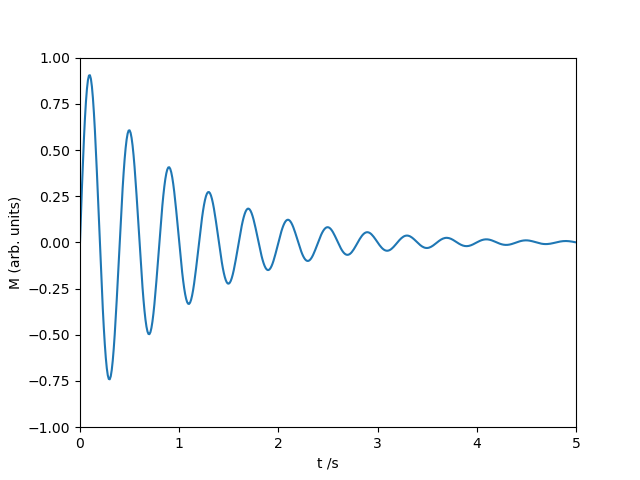
*interval, nframes = 1000 \* dt, int(tmax / dt)*

*# Animate once (set repeat=False so the animation doesn't loop).*

*ani = animation.FuncAnimation(fig, animate, frames=nframes, repeat=False,*

*interval=interval)*

*plt.show()*



**Рис.1.** Завершающий кадр анимации затухающей синусоидальной кривой

**Комбинирование битовых карт изображений**

В примере из предыдущего раздела весь объект линии должен был перерисовываться для каждого кадра. При весьма большом объеме данных или при сложном изображении это может замедлять анимацию. В этом случае может помочь операция комбинирования битовых карт изображения – блиттинг (blitting – Bit Blt, сокращение от bit block transfer), методика из области компьютерной графики, позволяющая в цикле анимации перерисовывать только те части изображения, которые изменяются между кадрами. Это исключает необходимость перерисовки всех отображаемых точек данных. Для использования операции блиттинга требуется некоторый дополнительный код: необходимо определить метод, передаваемый в аргументе *init\_func* объекту *FuncAnimation*. Этот передаваемый метод создает пустой кадр, но возвращает последовательность объектов анимации, которые должны быть перерисованы в каждом кадре. Функция *func*, вызываемая для каждого отображаемого кадра, также должна возвращать последовательность измененных объектов анимации.

**Пример 2.** Код в листинге 2 повторяет анимацию из примера 1, но использует методику блиттинга и явно передает аргументы в функцию анимации вместо объявления их глобальными объектами в теле этой функции.

**Листинг 30.** Анимация затухающей синусоидальной кривой с использованием *blit=True*

*%matplotlib notebook*

*import numpy as np*

*import matplotlib.pyplot as plt*

*import matplotlib.animation as animation*

*# Time step for the animation (s), max time to animate for (s).*

*dt, tmax = 0.01, 5*

*# Signal frequency (s-1), decay constant (s-1).*

*f, alpha = 2.5, 1*

*# These lists will hold the data to plot.*

*t, M = [], []*

*# Draw an empty plot, but preset the plot x- and y-limits.*

*fig, ax = plt.subplots()*

*line, = ax.plot([], [])*

*ax.set\_xlim(0, tmax)*

*ax.set\_ylim(-1, 1)*

*ax.set\_xlabel('t /s')*

*ax.set\_ylabel('M (arb. units)')*

*def init():*

*return line,*

*def animate(i, t, M):*

*"""Draw the frame i of the animation."""*

*# Append this time point and its data and set the plotted line data.*

*\_t = i\*dt*

*t.append(\_t)*

*M.append(np.sin(2\*np.pi\*f\*\_t) \* np.exp(-alpha\*\_t))*

*line.set\_data(t, M)*

*return line,*

*# Interval between frames in ms, total number of frames to use.*

*interval, nframes = 1000 \* dt, int(tmax / dt)*

*# Animate once (set repeat=False so the animation doesn't loop).*

*ani = animation.FuncAnimation(fig, animate, frames=nframes, init\_func=init,*

*fargs=(t, M), repeat=False, interval=interval, blit=True)*

*plt.show()*

**Анимация других объектов Matplotlib**

Для анимации других объектов Matplotlib, таких как патчи и надписи аннотации, ссылка на них должна быть сохранена и обработана в каждом кадре. Подобно объектам типа *Line2D*, содержащим метод *set\_data*, эти прочие классы также имеют методы-«установщики» (setter-методы) (например, методы *set\_* *center*, *set\_radius* для патча *Circle*), использование которых демонстрируется в примере 3.

**Пример 3.** Программа в листинге 3 выполняет анимацию прыгающего мяча из начального положения *(0, y0)* с начальной скоростью *(vx0, 0)*. Положение мяча, хронология траектории и надпись-метка высоты изменяются в каждом кадре.

Здесь в аргументе *frames* для *FuncAnimation* передается функция генератора *get\_pos*, которая возвращает следующее положение мяча на каждой итерации. Это положение обрабатывается в функции анимации *animate* вместо целочисленного индекса текущего кадра.

**Листинг 3.** Анимация прыгающего мяча

*%matplotlib notebook*

*import matplotlib.pyplot as plt*

*import matplotlib.animation as animation*

*# Acceleration due to gravity, m.s-2.*

*g = 9.81*

*# The maximum x-range of ball's trajectory to plot.*

*XMAX = 5*

*# The coefficient of restitution for bounces (-v\_up/v\_down).*

*cor = 0.65*

*# The time step for the animation.*

*dt = 0.005*

*# Initial position and velocity vectors.*

*x0, y0 = 0, 4*

*vx0, vy0 = 1, 0*

*def get\_pos(t=0):*

*"""A generator yielding the ball's position at time t."""*

*x, y, vx, vy = x0, y0, vx0, vy0*

*while x < XMAX:*

*t += dt*

*x += vx0 \* dt*

*y += vy \* dt*

*vy -= g \* dt*

*if y < 0:*

*# bounce!*

*y = 0*

*vy = -vy \* cor*

*yield x, y*

*def init():*

*"""Initialize the animation figure."""*

*ax.set\_xlim(0, XMAX)*

*ax.set\_ylim(0, y0)*

*ax.set\_xlabel('$x$ /m')*

*ax.set\_ylabel('$y$ /m')*

*line.set\_data(xdata, ydata)*

*ball.set\_center((x0, y0))*

*height\_text.set\_text(f'Height: {y0:.1f} m')*

*return line, ball, height\_text*

*def animate(pos):*

*"""For each frame, advance the animation to the new position, pos."""*

*x, y = pos*

*xdata.append(x)*

*ydata.append(y)*

*line.set\_data(xdata, ydata)*

*ball.set\_center((x, y))*

*height\_text.set\_text(f'Height: {y:.1f} m')*

*return line, ball, height\_text*

*# Set up a new Figure, with equal aspect ratio so the ball appears round.*

*fig, ax = plt.subplots()*

*ax.set\_aspect('equal')*

*# These are the objects we need to keep track of.*

*line, = ax.plot([], [], lw=2)*

*ball = plt.Circle((x0, y0), 0.08)*

*height\_text = ax.text(XMAX\*0.5, y0\*0.8, f'Height: {y0:.1f} m')*

*ax.add\_patch(ball)*

*xdata, ydata = [], []*

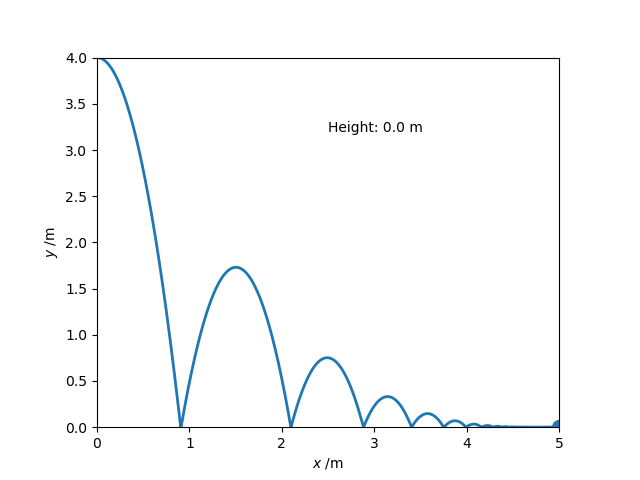
*interval = 1000\*dt*

*ani = animation.FuncAnimation(fig, animate, get\_pos, blit=True,*

*interval=interval, repeat=False, init\_func=init)*

*plt.show()*

Завершающий кадр анимации показан на рис. 3.



**Рис. 3.** Завершающий кадр анимации прыгающего мяча

**Задачи**

**З1.** Реализовать примеры трехмерных изображений и анимации из лекции.