ENGR 285 — Homework 3

Vincent Edwards

March 24, 2025

Euler Method

Simple Euler Method

$$\frac{dy}{dx} = f(x, y)$$

$$y(x_{i+1}) \approx y(x_i) + h \cdot f(x_i, y_i)$$

Vector Euler Method

$$\frac{d\vec{u}}{dt} = \vec{f}(t, \vec{u}) \quad \text{where} \quad \vec{u}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

$$\vec{u}_{i+1} \approx \vec{u}_i + h \cdot \vec{f}(t_i, \vec{u}_i)$$

Python Program

euler_method.py

```python
import numpy as np

def calculate(x_0, x_f, y_0, h, diff):
    """
    Calculate the x & y values using the simple/vector Euler method on a 1st order ODE

    - x_0: starting x value
    - x_f: final x value
    - y_0: starting y value
    - h: step size (this will be the rounded to be an even multiple of x_f - x_0)
    - diff: function that takes x and y as arguments and returns dy/dx

    Note: If using the vector Euler Method, treat y like the vector u
    """
    delta_x = x_f - x_0
    N = np.round(delta_x / h).astype(int) + 1

    x, h = np.linspace(x_0, x_f, N, retstep=True)
    y = np.array([y_0 for _ in range(N)], dtype=float)

    for i in range(N - 1):
        y[i + 1] = y[i] + h * diff(x[i], y[i])

    return x, y
```

Problem 1

Write a program that utilizes a while-loop to return the square root of an arbitrary positive number to the nearest whole number; i.e. find the largest integer who's square is less than the given number, and compare to the square of the next integer. Do not use any non-elementary math operations (i.e. the only math operations you can use are addition, subtraction, multiplication, and division). Test your code with several numbers of various sizes to check that it is working properly. Write out or copy/paste the code.

```python
def int_sqrt(x):
    """
    Return the square root of an arbitrary positive number to the nearest whole number
    """
    n = 1
    while n * n < x:
        n += 1
    previous = n - 1
    previous_squared = previous * previous
    if (n * n) - x <= x - previous_squared:
        return n
    else:
        return n - 1

test_input = [2, 3, 6, 7, 12, 13, 20, 21, 30, 31, 42]
print("Input | Output")
print("----- | ------")
for x in test_input:
    print(f"{x:5} | {int_sqrt(x):6}")
```

Output:

```
Input | Output
----- | ------
    2 |      1
    3 |      2
    6 |      2
    7 |      3
   12 |      3
   13 |      4
   20 |      4
   21 |      5
   30 |      5
   31 |      6
   42 |      6
```

Problem 2

One of the simplest ways to numerically calculate definite integrals is with Monte Carlo integration. Write a program that will approximately integrate the function $f(x) = e^{-\frac{1}{2}x^2}$ from $0 < x < 1$ by choosing 100 points (x, y) in the square $0 < x < 1$ and $0 < y < 1$ with random coordinates, counting

how many of them lie under the curve, and dividing by the total number of points. Write out or copy/paste the code. Run your program 6 times and record the outcomes.

```
import numpy as np
rng = np.random.default_rng(seed=285)

def f(x):
    return np.exp(-x**2 / 2)

N = 100
count = 0
trials = 6
for i in range(trials):
    x_values = rng.random(N)
    y_values = rng.random(N)
    below = (y_values < f(x_values)).sum()
    integral = below / N
    print(f"Trial {i + 1}: {integral}")
```

Output:

```
Trial 1: 0.79
Trial 2: 0.89
Trial 3: 0.79
Trial 4: 0.84
Trial 5: 0.86
Trial 6: 0.83
```

Problem 3

For each of the following initial value problems, solve using the Euler Method and either sketch or copy/paste a graph of the result over the given interval. Use a step size small enough that continuing to lower the step size further yields no noticeable change in the appearance of the graph.

Part a

$y' = (x - y)^2$ subject to $y(0) = 0.5$ for $0 < x < 2$

```
import euler_method
import matplotlib.pyplot as plt

x, y = euler_method.calculate(0, 2, 0.5, 0.01, lambda x, y: (x - y)**2)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set(xlabel="$x$", ylabel="$y$")
fig.savefig("media/03_a.svg")
```

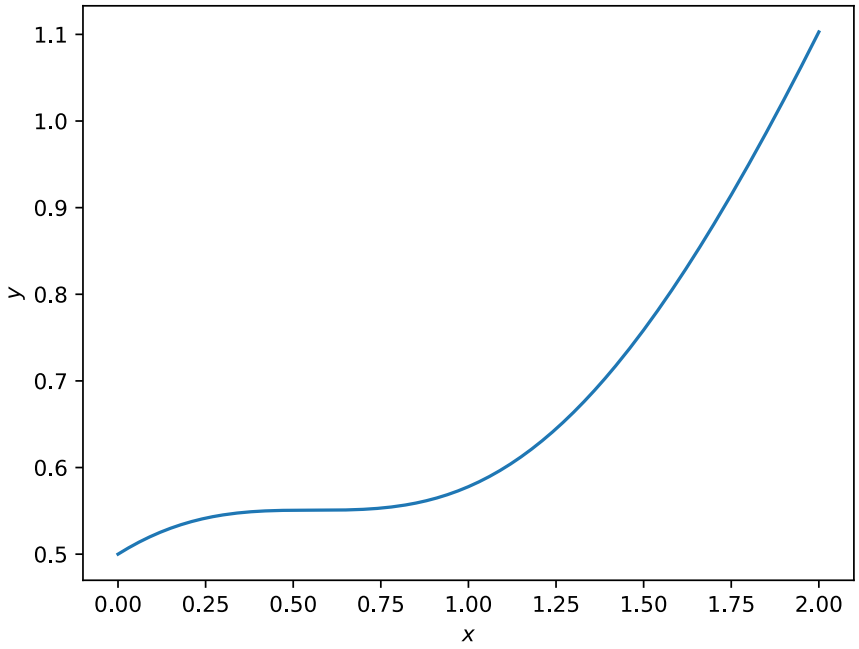

Figure 1: Graph of the solution to $y' = (x - y)^2$ subject to $y(0) = 0.5$

Part b

$y' = \sin(x + y^2)$ subject to $y(-\pi) = 0.1$ for $-\pi < x < 2\pi$

```
import euler_method
import numpy as np
import matplotlib.pyplot as plt

x, y = euler_method.calculate(-np.pi, 2*np.pi, 0.1, 0.01, lambda x, y: np.sin(x +
y**2))
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set(xlabel="$x$", ylabel="$y$")
fig.savefig("media/03_b.svg")
```

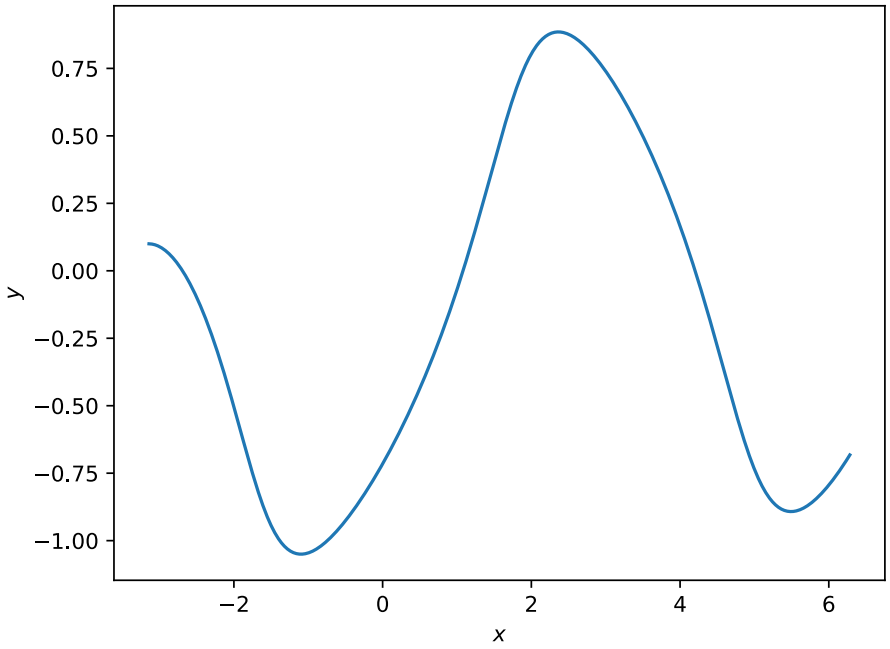

Figure 2: Graph of the solution to $y' = \sin(x + y^2)$ subject to $y(-\pi) = 0.1$

Problem 4

For each of the following initial value problems, first solve for $y(x)$ by separating the equation. Then use the Euler Method to determine an estimate for the maximum step size such that $y(4)$ is accurate to less than 1%. (i.e. run an Euler Method program repeatedly, decreasing step size each time, until the calculated value of $y(4)$ is less than 1% different from the exact answer)

problem_04.py

```
import euler_method
import numpy as np

def relative_error(x, x_true):
    return np.abs(x - x_true) / x_true

def maximize_h_within_error(x_0, x_f, y_0, diff, y_true, error_threshold, h_min=1e-9):
    N = 1
    while (h := (x_f - x_0) / N) > h_min:
        x, y = euler_method.calculate(x_0, x_f, y_0, h, diff)
        y_estimate = y[-1]
        error = relative_error(y_estimate, y_true)
        if error < error_threshold:
            return h, y_estimate, error
        N += 1
    return None, None, None
```

Part a

$2y' = \tan y$ subject to $y(0) = 0.05$

$$2y' = \tan(y)$$
$$\frac{dy}{dx} = \frac{1}{2}\tan(y)$$
$$\frac{dy}{\tan(y)} = \frac{1}{2}dx$$
$$\int \cot(y)dy = \int \frac{1}{2}dx$$
$$\ln(\sin(y)) = \frac{1}{2}x + C$$
$$\sin(y) = Ce^{\frac{1}{2}x}$$
$$y = \sin^{-1}\left(Ce^{\frac{1}{2}x}\right)$$

$\underline{y(0) = 0.05}$

$$0.05 = \sin^{-1}\left(Ce^{\frac{1}{2}(0)}\right)$$
$$C = \sin(0.05)$$
$$y = \sin^{-1}\left(\sin(0.05)e^{\frac{1}{2}x}\right)$$
$$y(4) = \sin^{-1}\left(\sin(0.05)e^{\frac{1}{2}(4)}\right)$$
$$y(4) = \sin^{-1}(\sin(0.05)e^2) \approx 0.3783$$

```
import problem_04
import numpy as np

def f(x, y):
    return np.tan(y) / 2

x_0 = 0
x_f = 4
y_0 = 0.05

y_true = np.asin(np.sin(0.05) * np.exp(2))
threshold = 0.01

h, y_estimate, relative_error = problem_04.maximize_h_within_error(x_0, x_f, y_0, f,
y_true, threshold)

if h is None:
    print("Estimates do not get close enough to the true value as h decreases")
else:
    print(f"With h = {h:.3}, y(4) is estimated to be {y_estimate:.4}
({relative_error:.4%} from the true value)")
```

Output:

```
With h = 0.0186, y(4) is estimated to be 0.3745 (0.9998% from the true value)
```

Part b

$6yy' = x\sin(y^2)$ subject to $y(0) = 1.5$

$$
\begin{aligned}
6yy' &= x\sin(y^2) \\
\frac{dy}{dx} &= \frac{x\sin(y^2)}{6y} \\
\frac{y}{\sin(y^2)}dy &= \frac{x}{6}dx \\
\int \frac{y}{\sin(y^2)}dy &= \int \frac{x}{6}dx \\
\int y\csc(y^2)dy &= \int \frac{x}{6}dx \\
-\frac{1}{2}\sinh^{-1}(\cot(y^2)) &= \frac{1}{12}x^2 + C \\
\sinh^{-1}(\cot(y^2)) &= -\frac{1}{6}x^2 + C \\
\cot(y^2) &= \sinh\left(-\frac{1}{6}x^2 + C\right) \\
y^2 &= \cot^{-1}\left[\sinh\left(C - \frac{1}{6}x^2\right)\right] \\
y &= \pm\sqrt{\cot^{-1}\left[\sinh\left(C - \frac{1}{6}x^2\right)\right]} \\
y &= \sqrt{\cot^{-1}\left[\sinh\left(C - \frac{1}{6}x^2\right)\right]}
\end{aligned}
$$

$\underline{y(0) = 1.5}$

$$
\begin{aligned}
1.5 &= \pm\sqrt{\cot^{-1}\left[\sinh\left(C - \frac{1}{6}(0)^2\right)\right]} \\
2.25 &= \cot^{-1}[\sinh(C)] \\
\cot(2.25) &= \sinh(C) \\
C &= \sinh^{-1}(\cot(2.25)) \\
y &= \sqrt{\cot^{-1}\left[\sinh\left(\sinh^{-1}(\cot(2.25)) - \frac{1}{6}x^2\right)\right]} \\
y(4) &= \sqrt{\cot^{-1}\left[\sinh\left(\sinh^{-1}(\cot(2.25)) - \frac{1}{6}(4)^2\right)\right]} \\
y(4) &= \sqrt{\cot^{-1}\left[\sinh\left(\sinh^{-1}(\cot(2.25)) - \frac{8}{3}\right)\right]} \approx 1.7536
\end{aligned}
$$

```
import problem_04
import numpy as np

def f(x, y):
    return x * np.sin(y**2) / (6 * y)

x_0 = 0
x_f = 4
y_0 = 1.5

C = np.asinh(1 / np.tan(2.25))
y_true = np.sqrt(np.atan2(1, np.sinh(C - 8/3)))
threshold = 0.01

h, y_estimate, relative_error = problem_04.maximize_h_within_error(x_0, x_f, y_0, f,
```

```
y_true, threshold)

if h is None:
    print("Estimates do not get close enough to the true value as h decreases")
else:
    print(f"With h = {h:.3}, y(4) is estimated to be {y_estimate:.4}
({relative_error:.4%} from the true value)")
```

Output:

```
With h = 0.8, y(4) is estimated to be 1.766 (0.7166% from the true value)
```

Problem 5

Error in the Euler Method is difficult to conceptualize outside of physical contexts; in this problem we will look at the degree of the violation of energy conservation present in Euler solutions of a simple spring-mass system. Consider the system

$$\frac{dx}{dt} = v \quad \text{and} \quad \frac{dv}{dt} = -\frac{k}{m}x$$

We'll choose $m = 1$ and $k = \pi^2$ so that the period should be $T = 2$.

a) Plot the trajectory of the mass over time $0 < t < 4$ with the initial conditions $x(0) = 1$ and $v(0) = 0$ for several different step sizes between 0.001 and 0.1. Describe how you can tell visually from the trajectories whether there is significant error in the calculation, and describe how the magnitude of that error appears to change when the step size increases.

b) The energy of the spring-mass system is given by

$$E(t) = \frac{1}{2}m[v(t)]^2 + \frac{1}{2}k[x(t)]^2$$

and theoretically should not change in time; i.e. $E(t) = E(0)$. Add code to your program from part (a) to additionally calculate $E(t)$ and plot it as a function of time $0 < t < 4$ for three different step sizes between 0.001 and 0.1. Comment on any trends that you observe.

```
import euler_method
import numpy as np
import matplotlib.pyplot as plt

def u_prime(t, u, m, k):
    x, v = u
    return np.array([
        v,
        -k/m * x
    ])

def energy(x, v, m, k):
    K = 1/2 * m * v**2
    U_s = 1/2 * k * x**2
    return K + U_s
```

```
t_0 = 0
t_f = 4
u_0 = np.array([1, 0])
m = 1
k = np.pi**2

for h in [0.001, 0.01, 0.1]:
    diff = lambda t, u: u_prime(t, u, m, k)
    t, u = euler_method.calculate(t_0, t_f, u_0, h, diff)
    x = u[:, 0]
    v = u[:, 1]
    E = energy(x, v, m, k)

    fig, axes = plt.subplots(1, 2, figsize=[12.8, 4.8])

    position_ax = axes[0]
    position_ax.plot(t, x, "o")
    position_ax.set(xlabel="$t$", ylabel="$x$")

    energy_ax = axes[1]
    energy_ax.plot(t, E, "o")
    energy_ax.set(xlabel="$t$", ylabel="$E$")

    fig.tight_layout()
    fig.savefig(f"media/05_h_{h}.svg")
```

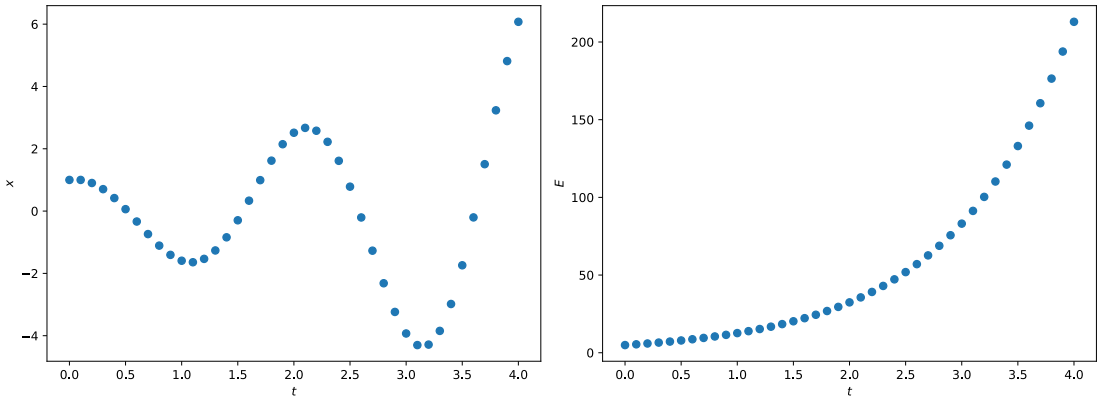

Figure 3: Position vs Time & Energy vs Time ($h = 0.1$)

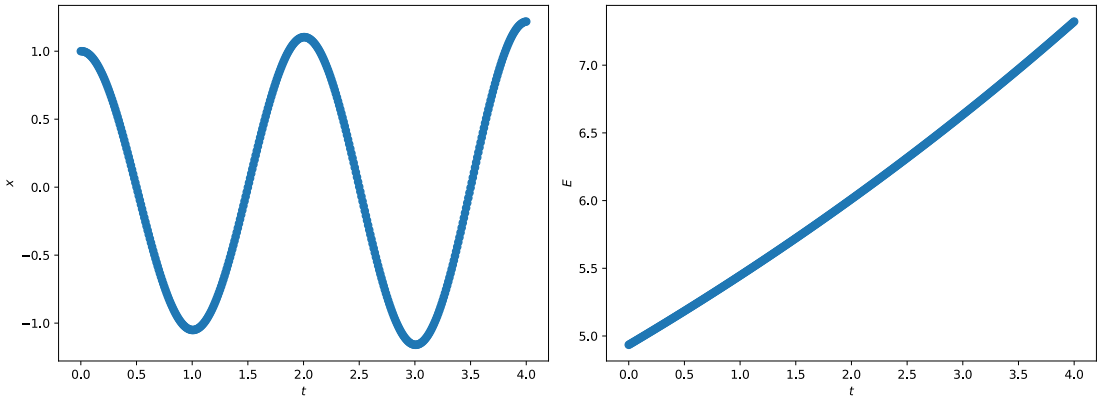

Figure 4: Position vs Time & Energy vs Time ($h = 0.01$)

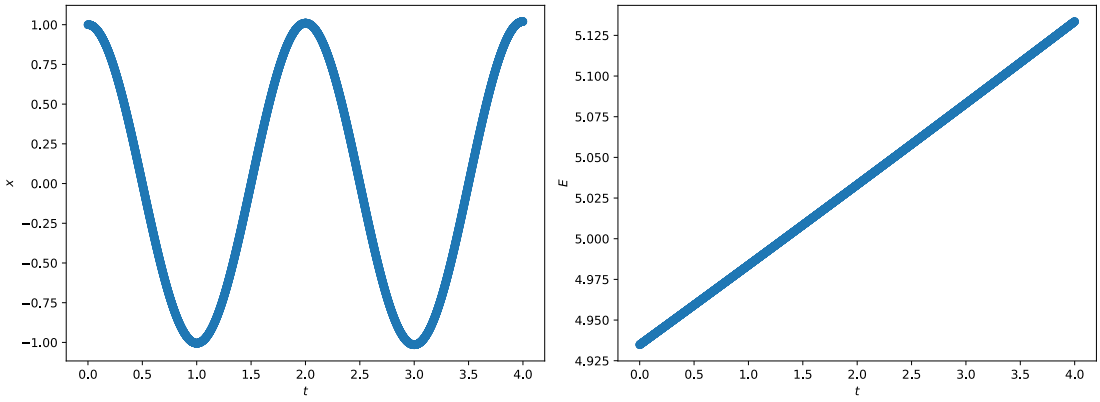

Figure 5: Position vs Time & Energy vs Time ($h = 0.001$)

The total energy of the system, and thus the amplitude, are supposed to remain constant. If the amplitude increases or decreases over time, then the total energy must be increasing or decreasing, which should not happen.

In the position vs time graphs, the amplitude increases over time, indicating that the energy must be increasing too. There must be some error in the calculations using the Euler Method. The effect is more pronounced for larger step sizes (h). Larger step sizes lead to larger changes in amplitude and total energy. In other words, decreasing the step size reduces the error.

Looking at the energy vs time graphs, the energy increases over time at an increasing rate. An increase in energy was expected on account of the increase in amplitude. Since the energy changes continuously, a boost in energy is gained on each iteration of the Euler Method, Thus, each iteration accumulates some error.

Problem 6

Air resistance is notoriously difficult to model; as it turns out the most accurate models of air resistance are complicated functions of the speed. Consider a more general air resistance model, where we take the friction force to be proportional to the speed to an arbitrary power α:

$$\frac{dv}{dt} = -g - kv|v|^{\alpha-1}$$

We would ultimately like to compare the maximum height achieved by vertically thrown objects as we vary α.

Part a

In order to compare these air resistance models, we should choose the proportionality constants so that the models have the same *terminal speed* (i.e. the speed at which a *dropped* object will not change its velocity; $\frac{dv}{dt} = 0$). Express the proportionality constant k in terms of the terminal speed v_∞, the power α, and g.

$\frac{dv}{dt} = 0$ when $v = -v_\infty$

$0 = -g - k(-v_\infty)|-v_\infty|^{\alpha-1}$

$g = kv_\infty(v_\infty)^{\alpha-1}$

$g = k(v_\infty)^\alpha$

$k = \frac{g}{(v_\infty)^\alpha}$

Parts b & c

b) From here we will work in units of time where $g = 1$ and units of distance where $v_\infty = 1$ for convenience. Write a program using the vector Euler Method that will determine the maximum height achieved by a vertically thrown object. Choose a value for v_0 such that $0 < v_0 < 1$ and record the maximum height for 5 different values of α such that $1 < \alpha < 2$.

c) Repeat the previous part for a different value of v_0 such that $v_0 > 1$.

```
import euler_method
import numpy as np

def u_prime(t, u, g, v_oo, alpha):
    v = u[1]
    k = g / v_oo**alpha
    return np.array([
        v,
        -g - k * v * np.abs(v)**(alpha - 1)
    ])

v_oo = 1
g = 1
t_0 = 0
h = 0.001
alpha_values = np.linspace(1, 2, 5)
v_0_values = [0.75, 2.50]

for i, v_0 in enumerate(v_0_values):
    t_f = v_0 / g
    u_0 = np.array([0, v_0])

    y_max_values = []
    for alpha in alpha_values:
        diff = lambda t, u: u_prime(t, u, g, v_oo, alpha)
```

```
        t, u = euler_method.calculate(t_0, t_f, u_0, h, diff)
        y = u[:, 0]
        y_max = y.max()
        y_max_values.append(y_max)

    if i != 0: print()
    print(f"v_0 = {v_0}")
    print()

    print("alpha | max height")
    print("----- | ----------")
    for alpha, y_max in zip(alpha_values, y_max_values):
        print(f"{alpha:5.2f} | {y_max:10.4f}")
```

Output:

```
v_0 = 0.75

alpha | max height
----- | ----------
 1.00 |     0.1907
 1.25 |     0.2004
 1.50 |     0.2090
 1.75 |     0.2166
 2.00 |     0.2234

v_0 = 2.5

alpha | max height
----- | ----------
 1.00 |     1.2479
 1.25 |     1.1762
 1.50 |     1.1091
 1.75 |     1.0471
 2.00 |     0.9904
```

Problem 7

In this problem you'll code a simple simulation of a baseball at-bat. Suppose a baseball pitcher will throw a pitch into the strikezone 50% of the time, and the batter will swing at a pitch 50% of the time. When the pitch is in the strikezone (and the batter swings), the batter hits in-field 40% of the time, hits a foul ball 30% of the time, and misses 30% of the time. If the pitch is not in the strikezone (and the batter swings), these probabilities change to 15%, 35%, and 50% respectively. The following is a list of the at-bat rules of baseball:

- Batters start the at-bat with 0 "strikes" and 0 "balls".
- If the batter swings and misses, they add a strike.
- If the pitch is in the strikezone and the batter doesn't swing, they add a strike.
- If the pitch is out of the strikezone and the batter doesn't swing, they add a ball.

- If the batter hits a foul ball, they add a strike unless they are at 2 strikes (if they are, nothing happens and the at-bat continues).
- The at-bat ends with an in-field hit, or when the batter reaches 3 strikes (a strikeout), or when the batter reaches 4 balls (a walk).

a) Write a program using a while-loop that will simulate a complete at-bat according to the probabilities and rules above. Your program should output the total number of pitches, the total number of foul balls, and the result of the at-bat.
b) Convert the while-loop portion of your code into a function, and then build a program that will use the function 100 times and then use the results to calculate the following statistics for at-bats:

- The average number of pitches
- The average number of foul balls
- The overall probability that the at-bat will result in an in-field hit
- The overall probability that the at-bat will result in a strikeout
- The overall probability that the at-bat will result in a walk

```
import numpy as np
rng = np.random.default_rng(seed=285)

# Swing and At-Bat Results
IN_FIELD_HIT = 0
FOUL_BALL = 1
STRIKE = 2
BALL = 3
STRIKEOUT = 4
WALK = 5

def simulate_pitch(p_strikezone):
    """
    Return True if the pitch is thrown into the strikezone, and False otherwise.
    p_strikezone is the probability the pitch makes in into the strikezone.
    """
    return rng.random() < p_strikezone

def simulate_swing_decision(p_swing):
    """
    Return True if the batter swings at a pitch, and False otherwise.
    p_swing is the probability the batter swings.
    """
    return rng.random() < p_swing

def simulate_swing_result(p_results):
    """
    Return an integer representing the result of the swing.
    p_results is a list containing the probabilities of each possible result:
    - in-field hit
    - foul ball
    - strike
    """
    random_num = rng.random()
```

```
    if random_num < p_results[0]:
        return IN_FIELD_HIT
    elif random_num < p_results[0] + p_results[1]:
        return FOUL_BALL
    else:
        return STRIKE

def simulate_single_at_bat(p_strikezone, p_swing, p_strikezone_results,
p_not_strikezone_results):
    """
    Simulate a single at-bat with the given probabilities.
    Return an integer representing the result of the single at bat.
    Possible results:
    - in-field hit
    - foul ball
    - strike
    - ball
    """
    pitch_in_strikezone = simulate_pitch(p_strikezone)
    batter_swings = simulate_swing_decision(p_swing)
    if batter_swings:
        if pitch_in_strikezone:
            swing_result = simulate_swing_result(p_strikezone_results)
        else:
            swing_result = simulate_swing_result(p_not_strikezone_results)
        return swing_result
    elif pitch_in_strikezone:
        return STRIKE
    else:
        return BALL


def simulate_complete_at_bat(p_strikezone=0.5, p_swing=0.5, p_strikezone_results=[0.4,
0.3, 0.3], p_not_strikezone_results=[0.15, 0.35, 0.50]):
    """
    Simulate an complete at-bat with the given probabilities.
    Return the outcome in a dictionary containing:
    - pitches
    - foul_balls
    - result (an integer representing an in-field hit, a strikeout, or a walk)
    - strikes
    - balls
    """
    strike_limit = 3
    ball_limit = 4

    strikes = 0
    balls = 0
    pitches = 0
    foul_balls = 0
    successful_hit = False
```

```
    while (strikes < strike_limit) and (balls < ball_limit) and not successful_hit:
        pitches += 1
        result = simulate_single_at_bat(p_strikezone, p_swing, p_strikezone_results,
p_not_strikezone_results)
        if result == IN_FIELD_HIT:
            successful_hit = True
        elif result == FOUL_BALL:
            foul_balls += 1
            if strikes < strike_limit - 1: strikes += 1
        elif result == STRIKE:
            strikes += 1
        else:
            balls += 1

    if successful_hit:
        at_bat_result = IN_FIELD_HIT
    elif strikes == strike_limit:
        at_bat_result = STRIKEOUT
    else:
        at_bat_result = WALK

    outcome = {
        "pitches": pitches,
        "foul_balls": foul_balls,
        "result": at_bat_result,
        "strikes": strikes,
        "balls": balls,
    }

    return outcome

def calculate_result_probability(outcomes, result):
    """
    Given a list of at-bat outcomes, calculate the probability of a specific result.
    Possible results:
    - in-field hit
    - strikeout
    - walk
    """
    N = len(outcomes)
    count = 0
    for outcome in outcomes:
        if outcome["result"] == result:
            count += 1
    return count / N

trials = 100
at_bat_outcomes = []
for _ in range(trials):
    outcome = simulate_complete_at_bat()
```

```
    at_bat_outcomes.append(outcome)

avg_pitches = np.mean([outcome["pitches"] for outcome in at_bat_outcomes])
avg_foul_balls = np.mean([outcome["foul_balls"] for outcome in at_bat_outcomes])
p_in_field_hit = calculate_result_probability(at_bat_outcomes, IN_FIELD_HIT)
p_strikeout = calculate_result_probability(at_bat_outcomes, STRIKEOUT)
p_walk = calculate_result_probability(at_bat_outcomes, WALK)

print(f"Average number of pitches: {avg_pitches}")
print(f"Average number of foul balls: {avg_foul_balls}")
print(f"Overall probability of in-field hit: {p_in_field_hit:.0%}")
print(f"Overall probability of strikeout: {p_strikeout:.0%}")
print(f"Overall probability of walk: {p_walk:.0%}")
```

Output:

```
Average number of pitches: 3.47
Average number of foul balls: 0.56
Overall probability of in-field hit: 54%
Overall probability of strikeout: 41%
Overall probability of walk: 5%
```