# ЛАБОРАТОРНАЯ РАБОТА 6

**Численное решение задачи Коши**

Вариант 20

Студент: Маркаров М.Г.

Преподаватель: ст.преп.Крупин Г.В.

**Задача 6.1.** Найти приближенное решение задачи Коши для обыкновенного дифференциального уравнения (ОДУ) 1 порядка

  (6.1)



методом Эйлера c постоянным шагом и с помощью встроенных средств *Python***.**

ПОРЯДОК РЕШЕНИЯ ЗАДАЧИ:

1. Найти аналитическое решение задачи.

2. Найти приближенное решение задачи Коши с шагом , N=10 по явному методу Эйлера. Найти величину погрешности по формуле ; здесьи - значения точного и приближенного решений в узлах сетки ,*i=0..N.*

3. Используя встроенную функцию Python scipy.integrate.solve\_ivp, найти решение задачи с тем же шагом.

Найти величину погрешности и сравнить ее со значением п. 2.

4. На одном чертеже построить три графика решения.

Решение:

1. Найдем аналитическое решение задачи:

При r(t)=-1/(2t+1) и t0=0, T=2,y0=1.414



Имеем: y(t)= 1.414 \* exp(-0.5 \* ln(2\*t+1))

Решим всю задачу разом, используя пакеты numpy,matplotlib,scipy

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

def analytical\_solution(t):

return 1.414 \* np.exp(-0.5 \* np.log(2\*t+1))

def euler\_method(f, y0, t,h):

y = np.zeros(len(t))

y[0] = y0

for i in range(1, len(t)):

y[i] = y[i-1] + h \* f(t[i-1], y[i-1])

return y

t0 = 0

T = 2

N = 10

y0 = 1.414

h = (T - t0) / N

t\_values = np.linspace(t0, T, N)

y\_euler = euler\_method(lambda t, y: y \* (-1 / (2 \* t + 1)), y0, t\_values,h)

solution = solve\_ivp(lambda t, y: y \* (-1 / (2 \* t + 1)), t\_span=(t0, T), y0=[y0], max\_step=h)

y\_exact = analytical\_solution(t\_values)

y\_solve\_ivp\_trimmed = solution.y[0][:N] # подгон размерности solve ivp под другие векторы решений

print("Аналитическое решение:", y\_exact)

print("Численное решение методом Эйлера:", y\_euler)

print("Численное решение с использованием solve\_ivp:", y\_solve\_ivp\_trimmed)

error\_euler = np.max(np.abs(y\_exact - y\_euler))

error\_solve\_ivp =np.max( np.abs(y\_exact - y\_solve\_ivp\_trimmed))

print("Погрешность метода Эйлера:", error\_euler)

print("Погрешность solve\_ivp:", error\_solve\_ivp)

plt.plot(t\_values, y\_exact, label='Аналитическое решение', color='blue')

plt.plot(t\_values, y\_euler, label='Метод Эйлера', color='red', marker='o')

plt.plot(solution.t, solution.y[0], label='solve\_ivp', color='green', marker='s')

plt.xlabel('t')

plt.ylabel('y(t)')

plt.legend()

plt.grid(True)

plt.show()

Результат работы программы:

Аналитическое решение: [1.414 1.17651912 1.02883612 0.92568029 0.8484 0.78771962

0.73843742 0.6973805 0.6624891 0.63236002]

Численное решение методом Эйлера: [1.414 1.1312 0.97457231 0.8713823 0.79669239 0.73933054 0.69344105 0.655617 0.62372212 0.59633919]

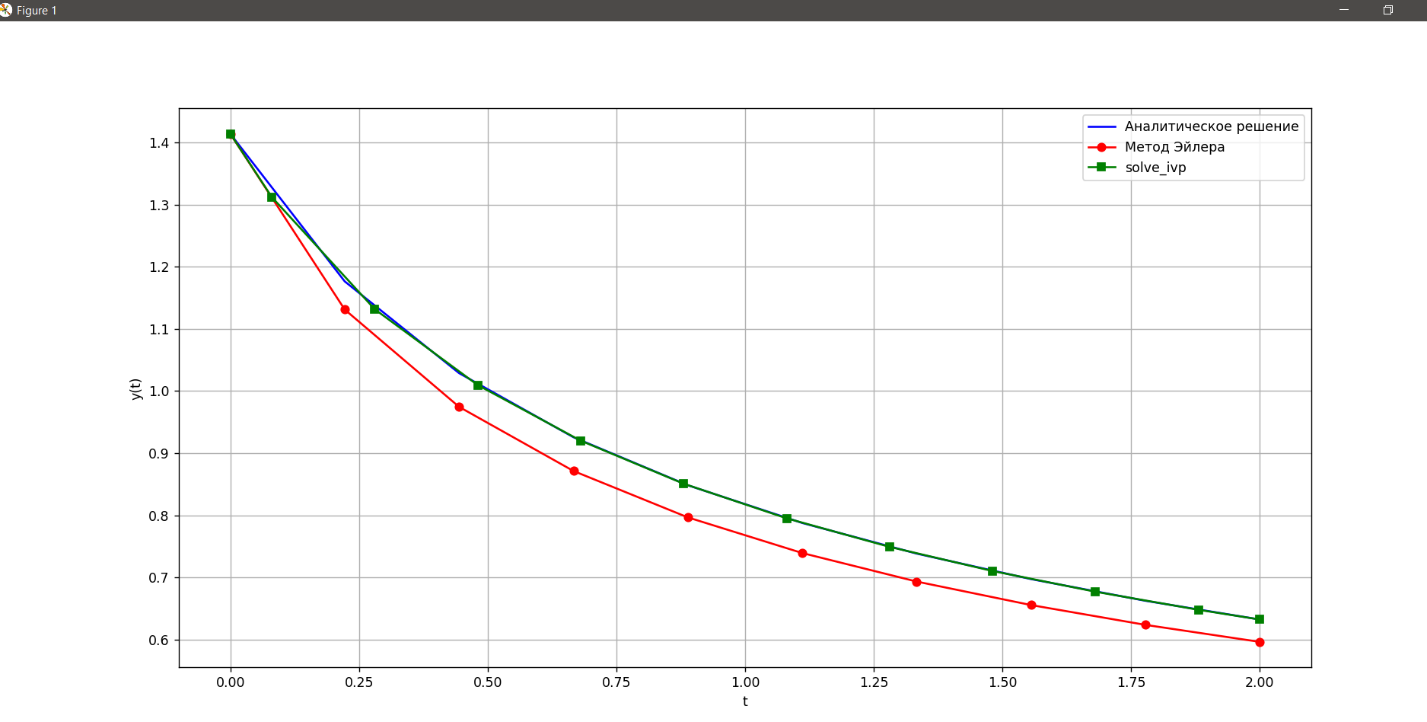
Численное решение с использованием solve\_ivp: [1.414 1.31218285 1.13166775 1.00968869 0.92019993 0.85094161 0.79528499 0.74929124 0.71045325 0.67708926]

Погрешность метода Эйлера: 0.05429799173854577

Погрешность solve\_ivp( с выбранным нами шагом\*): 0.1356637353843868

\*т.к solve ivp дает очень точное решение если не ограничивать его на шаг

Сравнение численных решений и аналитического.



**Задача 6.2.** Найти приближенное решение задачи Коши с заданной точностью .

1.Составить программу решения задачи Коши (6.1) реализующую метод, указанный в индивидуальном варианте с заданной точностью . Для нахождения погрешности использовать правило Рунге.

2.Найти решение задачи Коши методом Эйлера с точностью. Для нахождения погрешности использовать правило Рунге.

3.Сравнить величины шагов, которые потребовались для достижения точности в п.1 и п.2. Объяснить полученные результаты с помощью теоретического материала.

Реализация на языке Python:

Сначала решим задачу для явного метода Эйлера с подбором шага по Рунге:

import numpy as np

def f(t, y):

return y \* (-1 / (2 \* t + 1))

def euler\_step(t, y, h):

return y + h \* f(t, y)

def runge\_rule(y1, y2, p):

return abs(y1 - y2) / (2\*\*p - 1)

t0 = 0

y0 = 1.414

T = 2

epsilon = 1e-6

h = 0.1

while True:

y1 = euler\_step(t0, y0, h)

y2 = euler\_step(t0, y0, h/2)

y2 = euler\_step(t0 + h/2, y2, h/2)

error = runge\_rule(y1, y2, 1)

if error < epsilon:

num\_steps = int((T - t0) / h)

break

h /= 2

t\_values = [t0]

y\_values = [y0]

for i in range(num\_steps):

y\_next = euler\_step(t\_values[-1], y\_values[-1], h)

t\_values.append(t\_values[-1] + h)

y\_values.append(y\_next)

print(f"Количество шагов явного метода Эйлера: {num\_steps}")

Результат работы программы:

Количество шагов явного метода Эйлера: 2560

Теперь реализуем таким же образом неявный метод Эйлера и найдем его число шагов:

~~patch.1.001~~

~~Оптимизировано:~~

~~Добавлено func\_cache как значение f(t0,y0), чтобы не считать его каждый раз.~~

~~В этой версии кода значение функции f(t, y) вычисляется только один раз для каждого временного шага, а затем используется в обеих функциях шага. Это позволяет уменьшить количество вычислений функции и, таким образом, уменьшить общую трудоёмкость решения.~~

~~Добавлены векторы Y и Y\_half. массивы Y и Y\_half помогают оптимизировать код, так как они используются для хранения значений решений дифференциального уравнения на различных временных шагах.~~

*Patch.1002*

*Прогноз-коррекцию я заменил на метод Ньютона ибо так оказалось попроще:*

***Мы можем просто использовать один массив y\_values и другой массив y\_old,запоминая y\_old=y\_values для следующего шага где h\*=h/2 , т.к при новом шаге h\*=h/2 значения y совпадут с y\_old и их не надо считать заново и т.д до достижения err>=eps.***

***Также изменилось число шагов для достижения eps.***

import numpy as np

def implicit\_euler\_step(t, y, h, f, y\_prev=None):

if y\_prev is None:

y\_prev = y

# Метод Ньютона для решения неявного уравнения

def newton\_method(f, f\_prime, initial\_guess, max\_iterations=100, tolerance=1e-6):

x = initial\_guess

for i in range(max\_iterations):

fx = f(x)

if abs(fx) < tolerance:

return x

fpx = f\_prime(x)

if fpx == 0:

return None

x -= fx / fpx

return None

# Уравнение для метода Ньютона

equation = lambda y\_next: y\_next - y - h \* f(t + h, y\_next)

# Производная уравнения

derivative = lambda y\_next: 1 - h \* f(t + h, y\_next)

# Запускаем метод Ньютона

return newton\_method(equation, derivative, y\_prev)

def f(t, y):

return y \* (-1 / (2 \* t + 1))

def solve\_with\_accuracy(f, y0, t0, T, epsilon):

h = 0.1

***y\_old = None***

while True:

t\_values = np.arange(t0, T + h, h)

y\_values = np.zeros(len(t\_values))

y\_values[0] = y0

for i in range(1, len(t\_values)):

y\_values[i] = implicit\_euler\_step(t\_values[i-1], y\_values[i-1], h, f)

if y\_old is not None:

if np.all(np.abs(y\_values[::2] - y\_old) < epsilon):

return t\_values, y\_values

***y\_old = y\_values***

***h /= 2***

t0 = 0

T = 2

y0 = 1.414

epsilon = 1e-6

t\_values, y\_values = solve\_with\_accuracy(f, y0, t0, T, epsilon)

print("Количество шагов, требуемых для достижения точности:", len(t\_values))

Количество шагов неявного метода Эйлера: 10485761

Итого имеем:

Количество шагов явного метода Эйлера: 2560

Количество шагов неявного метода Эйлера: 10485761

Для достижения одной и той же точности epsilon 1e^-6

Причины различия:

Количество шагов, необходимых для достижения одной и той же точности с использованием явного и неявного методов Эйлера, может значительно различаться из-за различий в степени устойчивости и сходимости методов.

Явный метод Эйлера обычно менее устойчив, чем неявный метод Эйлера. Это означает, что при уменьшении шага явный метод может требовать значительно большее количество итераций для достижения заданной точности.

Неявный метод Эйлера, хотя и более сложен в вычислительном плане из-за необходимости решать неявные уравнения, может быть более устойчивым и требовать меньшее количество итераций для достижения точности.

*В нашем случае количество шагов для неявного метода Эйлера значительно больше, вероятно, потому что неявный метод Эйлера включает в себя итерационный процесс для нахождения коррекции на каждом шаге, что приводит к большему числу вычислительных операций. Однако неявный метод может оказаться более устойчивым и требовать меньше шагов для достижения заданной точности.*

*Таким образом, различие в количестве шагов между явным и неявным методами Эйлера обычно обусловлено их разными характеристиками устойчивости и сходимости.*

**Задача 6.3.** Задача Коши для ОДУ 1 порядка следующего вида

 ,  (6.2)

 .

описывает изменение биомассы  любого промыслового вида рыбы в океане. Здесь  - плотность насыщения,  - удельная скорость роста биомассы при ,  - постоянная, характеризующая интенсивность промысла.

A) Промоделировать процесс изменения биомассы в зависимости от интенсивности промысла.

B) Определить, при какой интенсивности количество выловленной за время  рыбы  является наибольшим. Определить диапазон хищнического лова

(т.е. значения интенсивности промысла, при которых вид полностью исчезает).

ПОРЯДОК РЕШЕНИЯ ЗАДАЧИ:

1. Промасштабировать задачу (1): вводя новые переменные , , получить задачу

 (6.3)

,

где , .

2. Составить программу решения задачи Коши с заданной точностью ℇ .

3. (A) Решить задачу Коши (2) на отрезке по времени  при минимальном и максимальном значениях параметра  из указанного в задании диапазона. Приближенно определить по графику момент времени, при котором численность популяции становится вдвое больше (меньше) начальной, а также момент времени, начиная с которого численность стабилизируется.

4. (B) Задать множество значений параметра , изменяя его на заданном отрезке с шагом 0.1. Для каждого значения параметра найти приближенное решение задачи Коши на отрезке по времени  .

5. Для каждого полученного решения вычислить интеграл  и определить оптимальное значение параметра , соответствующее максимальному значению интеграла.

6. Построить графики найденных решений при разных значениях параметра . Определить визуально, при каких значениях параметра происходит исчезновение популяции.

7. Составить отчет по задаче.

Решение задачи:

Совместим пункты 1,2,3 .

Решать будем через пакет SciPy:

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

def f(t, y, p, beta):

gamma = p \* beta

return y \* (1 - y) - p \* y

beta = 0.9

y0 = 0.3

gamma\_min = 0.2

gamma\_max = 1.2

t\_span = (0, 50)

eps = 1e-4

doubling\_times = {}

stabilization\_times = {}

for gamma in [gamma\_min, gamma\_max]:

p = gamma / beta

sol = solve\_ivp(lambda t, y: f(t, y, p, beta), t\_span, [y0], method='RK45', atol=eps, rtol=eps)

doubling\_time = None

stabilization\_time = None

for i in range(len(sol.t)):

if doubling\_time is None and sol.y[0][i] >= 2 \* y0:

doubling\_time = sol.t[i]

if stabilization\_time is None and doubling\_time is not None and abs(sol.y[0][i] - sol.y[0][i - 1]) < 0.1:

stabilization\_time = sol.t[i]

break

doubling\_times[gamma] = doubling\_time

stabilization\_times[gamma] = stabilization\_time

plt.plot(sol.t, sol.y[0], label=f'gamma = {gamma}')

plt.xlabel('Время')

plt.ylabel('Численность популяции')

plt.title('Динамика численности популяции')

plt.legend()

plt.grid(True)

plt.show()

print("Момент времени удвоения численности популяции:")

for gamma, time in doubling\_times.items():

print(f"При gamma = {gamma}: {time}")

print("Момент времени стабилизации численности популяции:")

for gamma, time in stabilization\_times.items():

print(f"При gamma = {gamma}: {time}")

В результате программа выдает время удвоения и стабилизации популяций(если она удваивается) и также рисует графики изменения популяции:

Момент времени удвоения численности популяции:

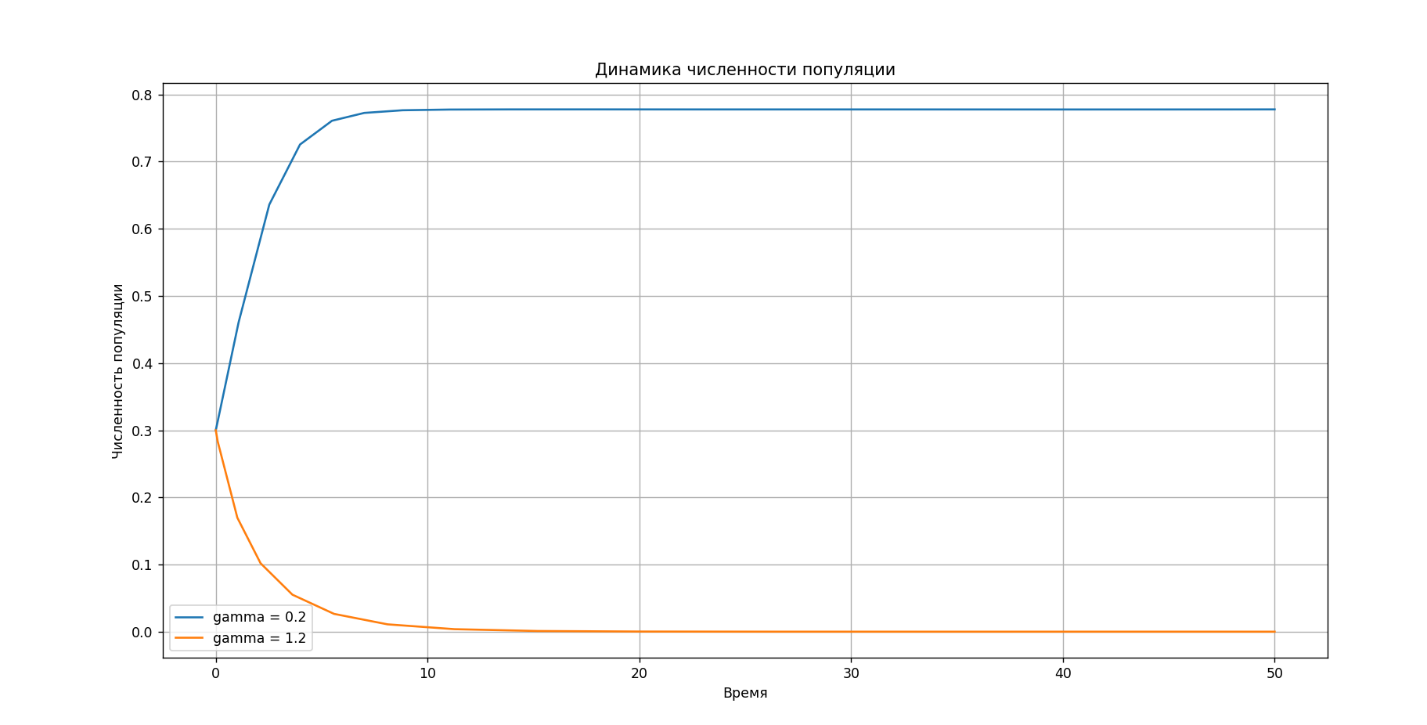
При gamma = 0.2: 2.5283797037114435

При gamma = 1.2: None # популяция не растет а убывает

Момент времени стабилизации численности популяции:

При gamma = 0.2: 13.886441950909504

При gamma = 1.2: 20.245722578446764



Совместим оставшиеся пункты в один:

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

def f(t, y, p, beta):

gamma = p \* beta

return y \* (1 - y) - p \* y

def compute\_integral\_simpson(sol, gamma, beta):

T = beta \* sol.t[-1]

y = sol.y[0]

h = sol.t[1] - sol.t[0]

integral = gamma \* (y[0] + y[-1]) / 2

for i in range(1, len(y) - 1):

if i % 2 == 0:

integral += gamma \* y[i]

else:

integral += 2 \* gamma \* y[i]

integral \*= h / 3

return integral

beta = 0.9

y0 = 0.3

gamma\_values = np.arange(0.2, 1.3, 0.1)

t\_span = (0, 50)

integrals = {}

solutions={}

for gamma in gamma\_values:

p = gamma / beta

sol = solve\_ivp(lambda t, y: f(t, y, p, beta), t\_span, [y0], method='RK45')

solutions[gamma] = sol

integrals[gamma] = compute\_integral\_simpson(sol, gamma, beta)

optimal\_gamma = max(integrals, key=integrals.get)

plt.figure(figsize=(10, 6))

for gamma, sol in solutions.items():

plt.plot(sol.t, sol.y[0], label=f'gamma = {gamma}')

plt.xlabel('Время')

plt.ylabel('Численность популяции')

plt.title('Динамика численности популяции при разных значениях параметра gamma')

plt.legend(loc='upper right')

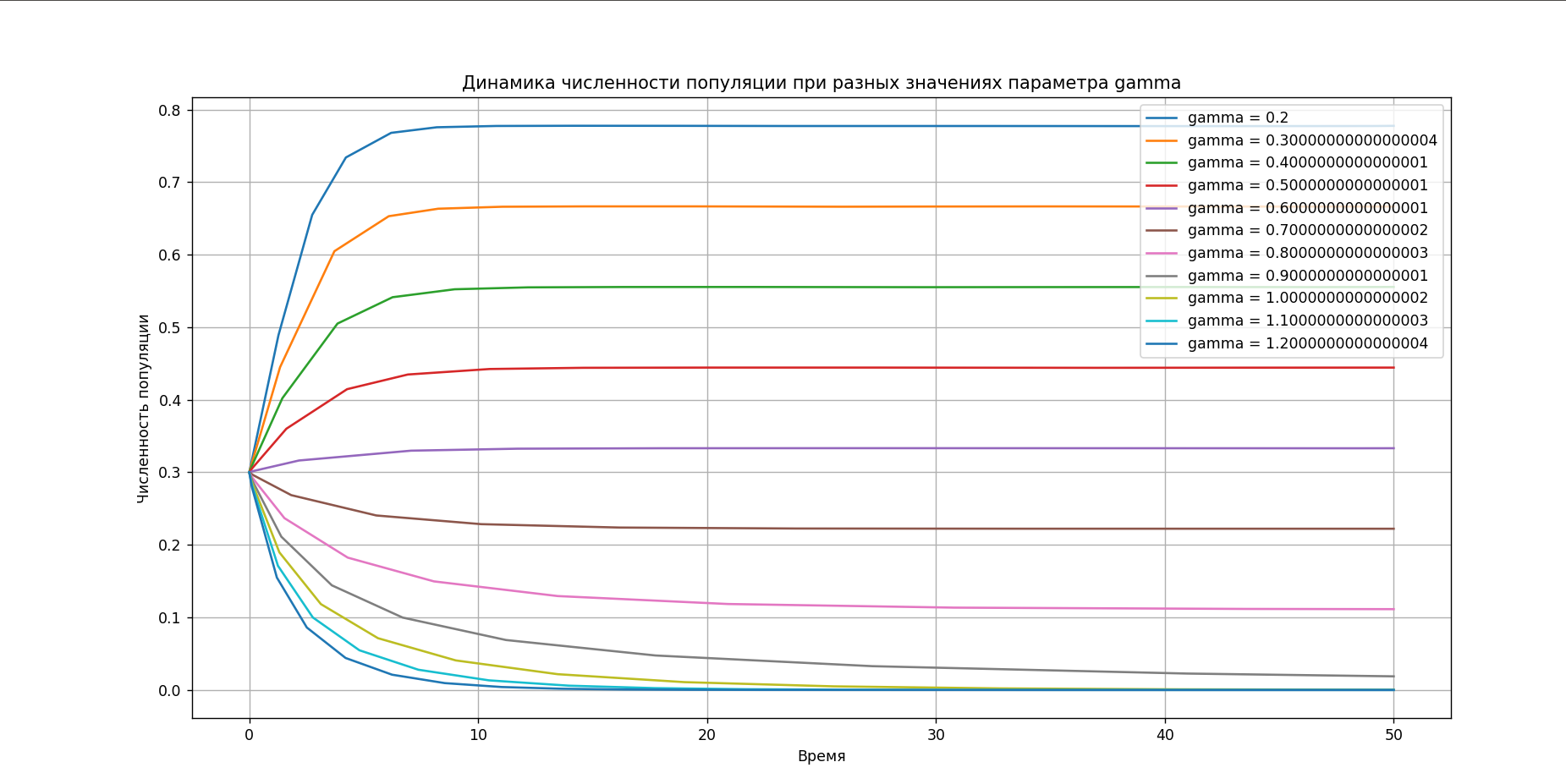
plt.grid(True)

plt.show()

print("Значение параметра gamma, соответствующее максимальному интегралу:", optimal\_gamma)

В результате получим наилучшее gamma для интеграла и графики:

Значение параметра gamma, соответствующее максимальному интегралу: 0.4



Проанализируем графики:

Популяция сходится к нулю(вымирает) при gamma от 1 до 1.2

Вывод по задаче: Была составлена и изучена модель поведения численности популяции вида, основанная на задаче Коши. Была выяснена зависимость популяции от интенсивности промысла( gamma) и при каких параметра gamma вид полностью вымрет.