

 Open in Colab
[\(https://colab.research.google.com/github/Sergei-Volkov/ComputationalMathsMIPT/blob/master/2Semester/HW1/HW1.ipynb\)](https://colab.research.google.com/github/Sergei-Volkov/ComputationalMathsMIPT/blob/master/2Semester/HW1/HW1.ipynb)

Задача 13.4

```
In [ ]: 1 from IPython.display import Image
```

```
In [ ]: 1 Image(filename='13.4.jpg')
```

Out[2]: Экогенетическая модель

Рассмотрим пример системы уравнений, которая описывает изменения численности популяций двух видов и эволюцию некого генетического признака α . Система ОДУ имеет вид:

$$\begin{aligned}\dot{x} &= x(1 - 0.5x - \frac{2}{7\alpha^2}y), \\ \dot{y} &= y(2\alpha - 3.5\alpha^2x - 0.5y), \\ \dot{\alpha} &= \varepsilon(2 - 7\alpha x).\end{aligned}$$

Параметры задачи таковы: $\varepsilon \leq 0.01$, $0 < x(0) < 1$, $y(0) = 1.7$, $\alpha(0) = 1$, конечное время интегрирования $T_k = 3000$. Наличие малого параметра в третьем уравнении системы показывает, что генетический признак меняется медленнее, чем численность популяций. Решение системы — релаксационные колебания.

Для численного решения используются ФДН-методы:

$$k = 2: \frac{3}{2}y_{n+1} - 2y_n + \frac{1}{2}y_{n-1} = hf_{n+1},$$

$$k = 3: \frac{11}{6}y_{n+1} - 3y_n + \frac{3}{2}y_{n-1} - \frac{1}{3}y_{n-2} = hf_{n+1},$$

$$k = 4: \frac{25}{12}y_{n+1} - 4y_n + 3y_{n-1} - \frac{4}{3}y_{n-2} + \frac{1}{4}y_{n-3} = hf_{n+1},$$

причем значения в недостающих точках доопределяются с помощью метода Рунге-Кутты (таблица 2).

Сравнить полученные численные результаты с результатами вычислений по однократно диагональным неявным методам Рунге-Кутты с двумя стадиями (второго порядка аппроксимации, асимптотически устойчивому, таблица 1 и третьего порядка аппроксимации, Таблица 2)

Таблица 1.

$\frac{2+\sqrt{2}}{2}$	$\frac{2+\sqrt{2}}{2}$	0
$\frac{2-\sqrt{2}}{2}$	$-\sqrt{2}$	$\frac{2+\sqrt{2}}{2}$
	1/2	1/2

Таблица 2.

$\frac{9+\sqrt{3}}{6}$	$\frac{9+\sqrt{3}}{6}$	0
$\frac{3-\sqrt{3}}{6}$	$\frac{3-2\sqrt{3}}{6}$	$\frac{3+\sqrt{3}}{6}$
	1/2	1/2

Построить функции устойчивости всех используемых численных методов

```
In [3]: 1 import numpy as np
2 from scipy.optimize import fsolve
3 import matplotlib.pyplot as plt
```

Правая часть системы

Заметим, что она не зависит явным образом от t , т.е. $f = f(u)$.

```
In [4]: 1 def f(u, epsilon=0.01):
2     x, y, alpha = u
3     return np.array([
4         x*(1 - x/2 -(2/7*alpha**2)*y),
5         y*(2*alpha - (7/2*alpha**2)*x - y/2),
6         epsilon*(2 - 7*alpha*x)
7     ])
```

Реализация методов

Однократно диагонально неявные методы Рунге-Кутты

В общем случае решается система уравнений вида

$$\dot{u} = f(t, u(t)), \quad f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$$

s-стадийным неяным методом Рунге-Кутты (PK) с определяющими коэффициентами a_{ij} , b_i , c_i называется метод вида

$$\begin{aligned} u^{n+1} &= u^n + \tau \sum_{i=1}^s b_i k_i \\ k_i &= f\left(t_n + c_i \tau, u^n + \tau \sum_{j=1}^{i-1} a_{ij} k_j\right) \end{aligned}$$

В данной задаче используются однократно диагонально неявные методы PK, т.е. такие, что для них выполнено $a_{ij} = 0$, $j > i$ и $\forall i \mapsto a_{ii} = a$.

В этих условиях для вычисления k_i требуется решить нелинейное уравнение следующего вида:

$$F(k_i) = f\left(t_n + c_i \tau, u^n + \tau \sum_{j=1}^{i-1} a_{ij} k_j + \tau a k_i\right) - k_i = 0$$

Так как система ДУ из условия задачи является автономной, нелинейное уравнение преобразуется к виду:

$$F(k_i) = f\left(u^n + \tau \sum_{j=1}^{i-1} a_{ij} k_j + \tau a k_i\right) - k_i = 0$$

Будем решать его с помощью функции `fsolve` пакета `scipy.optimize`.

```
In [5]: 1 class SDIRK_AutonomousSystem:
2     """
3         Однократно диагонально неявные методы Рунге-Кутты для автономных систем дифференциальных уравнений.
4
5             Параметры:
6                 a_matrix - матрица размера sxs с a_ij = 0, i>j и a_ii=a для всех i
7                 b_vector, c_vector - векторы размера sx1
8
9     """
10    def __init__(self, a_matrix, b_vector):
11        self._A = a_matrix
12        self._b = b_vector
13        self._s = len(b_vector)
14
15    def __nonlinear_equation(self, k_i, func, k, i, tau, u_n, params):
16        return func(
17            u_n
18            + tau*(self._A[i - 1, :i] @ np.vstack((k[1:i], k_i))),
19            **params
20        ) - k_i
21
22
23    def __call__(self, func, t_0, t_max, tau, u_0, **params):
24        """
25            Обсчет системы ДУ с правой частью func(u) на отрезке [t_0, t_max] с шагом tau (НУ u(t_0) = u_0).
26
27            Параметры:
28                func - функция правой части дифференциального уравнения f(t, u)
29                t_0 - левая граница отрезка, на котором решается задача Коши
30                t_max - правая граница отрезка, на котором решается задача Коши
31                tau - величина шага на итерации
32                u_0 - значение u(t_0)
33
34            Возвращает значения u на сетке.
35        """
36        dim = len(u_0) # размерность системы
37        N = int((t_max - t_0) / tau) # число шагов
38        u = np.zeros((N+1, dim)) # вектор u(t) в узлах сетки
39        u[0] = u_0 # начальное приближение
40        for n in range(N):
41            k = np.zeros((self._s + 1, dim))
42            for i in range(1, self._s + 1):
43                k[i] = fsolve(self.__nonlinear_equation, k[i-1], args=(func, k, i, tau, u[n], params), maxfev=5000)
44            u[n + 1] = u[n] + tau*(self._b @ k[1:])
45        return np.array([t_0 + n * tau for n in range(N+1)]), u # сетка и решение
```

```
In [6]: 1 RKtable_2_order = [
2     np.array([
3         # A
4         [(2 + np.sqrt(2))/2, 0],
5         [-np.sqrt(2), (2 + np.sqrt(2))/2]
6     ]),
7     np.array([1/2]*2), # b
8 ]
9 RKtable_3_order = [
10    np.array([
11        # A
12        [(3 + np.sqrt(3))/6, 0],
13        [(3 - 2*np.sqrt(3))/6, (3 + np.sqrt(3))/6]
14    ]),
15    np.array([1/2]*2), # b
]
```

```
In [7]: 1 RK_order2, RK_order3 = SDIRK_AutonomousSystem(*RKtable_2_order), SDIRK_AutonomousSystem(*RKtable_3_order)
```

Проверка методов

```
In [121]: 1 import pandas as pd
2 from numpy.linalg import norm
3 from math import inf
4 from scipy.optimize import curve_fit
```

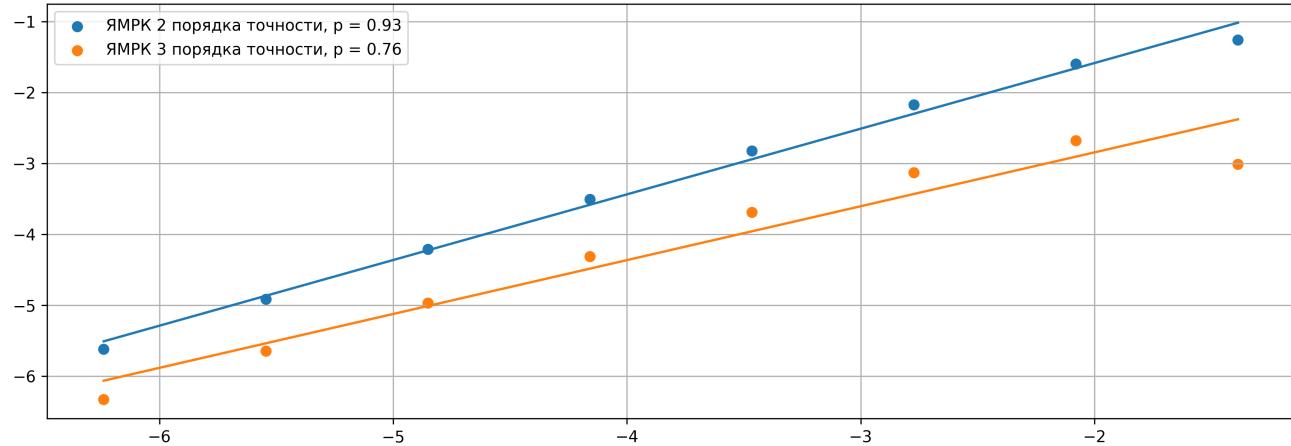
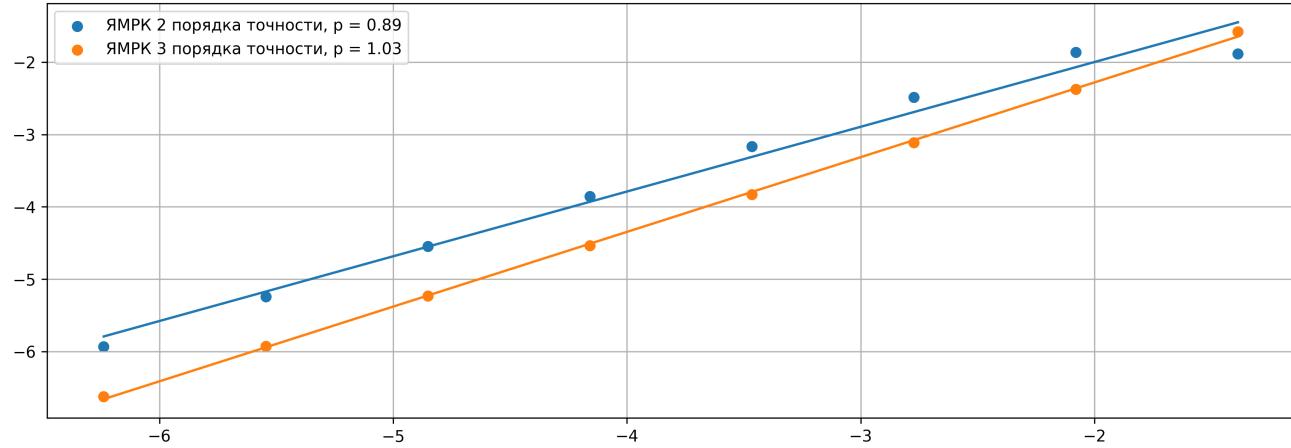
```
In [153]: 1 def test_system(u):
2     x, y = u
3     return np.array([x, -10*y])
```

```
In [154]: 1 def linear_func(x, k, b):
2     return k * x + b
```

```
In [155]: 1 C_norm_x = pd.DataFrame(
2     columns=['ЯМРК {} порядка точности'.format(j + 2) for j in range(2)],
3     index = [2**(-i) for i in range(2, 10)])
4 )
5 C_norm_y = pd.DataFrame(
6     columns=['ЯМРК {} порядка точности'.format(j + 2) for j in range(2)],
7     index = [2**(-i) for i in range(2, 10)])
8 )
```

```
In [181]: 1 for tau in [2**(-i) for i in range(2, 10)]:
2     for j, method in enumerate([RK_order2, RK_order3]):
3         t, u = method(test_system, 0, 1, tau, [1, 1])
4         C_norm_x['ЯМРК {} порядка точности'.format(j + 2)][tau] = np.log(norm(u[:, 0] - np.exp(t), inf))
5         C_norm_y['ЯМРК {} порядка точности'.format(j + 2)][tau] = np.log(norm(u[:, 1] - np.exp(-10*t), inf))
```

```
In [182]: 1 fig = plt.figure(figsize=(6.4 * 2.2, 4.8 * 2.2), dpi=300)
2 for i, C_norm in enumerate([C_norm_x, C_norm_y]):
3     plt.subplot(2, 1, i+1)
4     for column in C_norm.columns:
5         params, _ = curve_fit(
6             linear_func,
7             np.log(C_norm.index),
8             C_norm[column])
9     plt.scatter(np.log(C_norm.index), C_norm[column], label=column + ', p = {}'.format(round(params[0], 2)))
10    plt.plot(np.log(C_norm.index), linear_func(np.log(C_norm.index), *params))
11    plt.grid()
12    plt.legend(loc='upper left')
```



Функции устойчивости

Для методов РК функцию устойчивости $R(z)$, $z = \tau\lambda$ можно найти следующим образом:

$$R(z) = \frac{\det(\mathbf{E} - z\mathbf{A} + z\mathbf{e}\mathbf{b}^T)}{\det(\mathbf{E} - z\mathbf{A})}$$

Область устойчивости метода задается выражением:

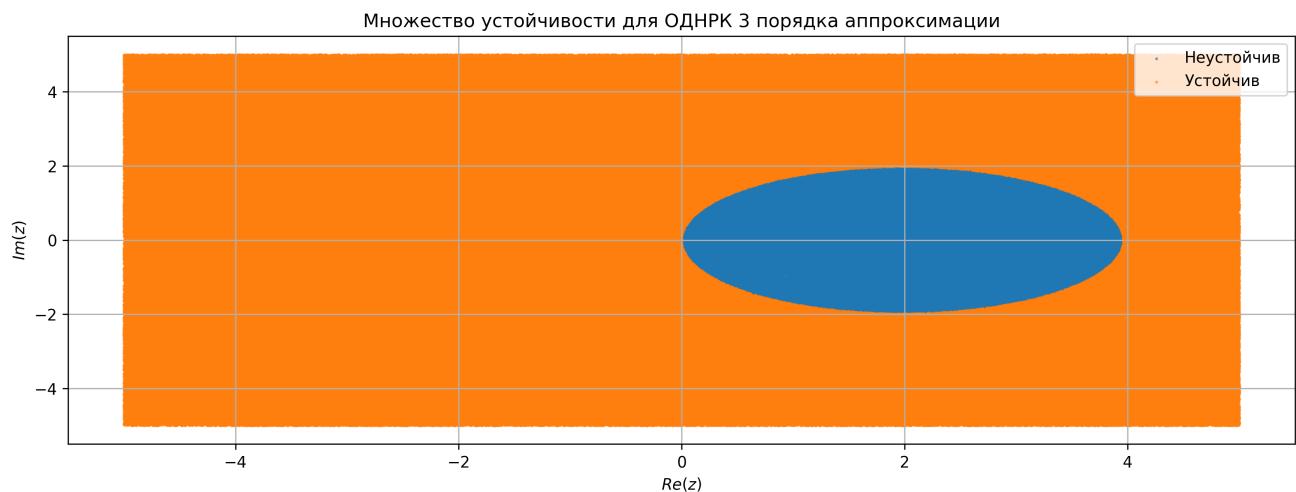
$$|R(z)| \leq 1$$

```
In [8]: 1 from numpy.linalg import det
```

```
In [9]: 1 def R(z: complex, A, b):
2     s = len(b)
3     return det(np.eye(s) - z*A + z*np.ones((s, 1)) @ b.reshape(1, 2)) / det(np.eye(s) - z*A)
```

```
In [11]: 1 x_range, y_range = np.random.uniform(-5, 5, size=1000000), np.random.uniform(-5, 5, size=1000000)
2 z_range = x_range + 1j*y_range
3 stab_2_order = np.array([abs(R(z, *table_2_order)) <= 1 for z in z_range])
4 stab_3_order = np.array([abs(R(z, *table_3_order)) <= 1 for z in z_range])
5
6 fig = plt.figure(figsize=(6.4 * 2.2, 4.8 * 2.2), dpi=300)
7
8
9 def stab_func(value):
10     return 'Устойчив' if value else 'Неустойчив'
11
12 for k, order in enumerate([stab_2_order, stab_3_order]):
13     plt.subplot(2, 1, k+1)
14     for g in np.unique(order):
15         i = np.where(order == g)
16         plt.scatter(x_range[i], y_range[i], s=1, label=stab_func(g), alpha=.5, cmap='Reds')
17     plt.title(f'Множество устойчивости для ОДНРК {k+2} порядка аппроксимации')
18     plt.ylabel('$Im(z)$')
19     if k == 1:
20         plt.xlabel('$Re(z)$')
21     plt.legend(loc='upper right'); plt.grid()
```

/usr/local/lib/python3.7/dist-packages/google/colab/_event_manager.py:28: UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
 func(*args, **kwargs)
/usr/local/lib/python3.7/dist-packages/IPython/core/pylabtools.py:125: UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
fig.canvas.print_figure(bytes_io, **kw)



ФДН-методы

В задаче используются следующие методы (k - порядок аппроксимации):

$$\begin{aligned} k = 2 : \frac{3}{2}u_{n+1} - 2u_n + \frac{1}{2}u_{n-1} &= \tau f_{n+1} \\ k = 3 : \frac{11}{6}u_{n+1} - 3u_n + \frac{3}{2}u_{n-1} - \frac{1}{3}u_{n-2} &= \tau f_{n+1} \\ k = 4 : \frac{25}{12}u_{n+1} - 4u_n + 3u_{n-1} - \frac{4}{3}u_{n-2} + \frac{1}{4}u_{n-3} &= \tau f_{n+1} \end{aligned}$$

или в более удобном для реализации виде (учитывая, что решаемая система - автономная):

$$\left(\vec{K}, u_{n-k+1}^{n+1} \right) = \tau f_{n+1} = \tau f(u^{n+1}),$$

где \vec{K} - вектор коэффициентов ФДН-метода, а $u_{n-k+1}^{n+1} = (u^{n-k+1}, u^n, \dots, u^{n+1})^T$, k - порядок аппроксимации формулы.

Таким образом для нахождения значения u^{n+1} требуется решить следующую нелинейную систему:

$$F(u^{n+1}) = \left(\vec{K}, u_{n-k+1}^{n+1} \right) - \tau f(u^{n+1}) = 0,$$

Видно, что для корректного решения задачи с помощью ФДН-метода порядка аппроксимации k требуется дополнительно задать начальные значения функции в $k - 1$ точках u^1, \dots, u^{k-1} . Значения в этих точках будут вычислены с помощью однократно диагонально неявного метода РК 3-го порядка аппроксимации (см. выше).

```
In [14]: 1 class BDF_AutonomousSystem:
2     """
3         ФДН-методы для автономных систем дифференциальных уравнений.
4
5             Параметры:
6                 K - вектор коэффициентов метода размерности k+1
7                     (начиная с точки с наименьшим индексом, k - порядок аппроксимации метода)
8     """
9     def __init__(self, K):
10         self._K = K
11         self.__k = len(K) - 1
12
13
14     def __nonlinear_equation(self, u_next, func, u, i, tau, params):
15         return self._K @ np.vstack((u[i-self.__k: i], u_next)) - tau*func(u_next, **params)
16
17
18     def __call__(self, func, t_0, t_max, tau, u_0, **params):
19         """
20             Обсчет системы ДУ с правой частью func(u) на отрезке [t_0, t_max] с шагом tau (НУ u(t_0) = u_0).
21
22             Параметры:
23                 func - функция правой части дифференциального уравнения f(t, u)
24                 t_0 - левая граница отрезка, на котором решается задача Коши
25                 t_max - правая граница отрезка, на котором решается задача Коши
26                 tau - величина шага на итерации
27                 u_0 - значение u(t_0)
28
29             Возвращает значения u на сетке.
30         """
31         dim = len(u_0) # размерность системы
32         N = int((t_max - t_0) / tau) # число шагов
33         u = np.zeros((N+1, dim)) # вектор u(t) в узлах сетки
34         u[0] = u_0 # начальное приближение
35         RK_order3 = SDIRK_AutonomousSystem(*[ # метод РК для нахождения значений в первых k-1 точках
36             np.array([
37                 [(3 + sqrt(3))/6, 0],
38                 [(3 - 2*sqrt(3))/6, (3 + sqrt(3))/6]
39             ]),
40             np.array([1/2]*2), # b
41             np.array([(3 + sqrt(3))/6, (3 - sqrt(3))/6]) # c
42         ])
43         for i in range(1, self.__k):
44             u[i] = RK_order3(func, t_0 + (i-1)*tau, t_0 + i*tau, tau, u[i-1])[1][-1]
45         for i in range(self.__k, N+1):
46             u[i] = fsolve(self.__nonlinear_equation, u[i-1], args=(func, u, i, tau, params), maxfev=5000)
47         return [t_0 + n * tau for n in range(N+1)], u # сетка и решение
```

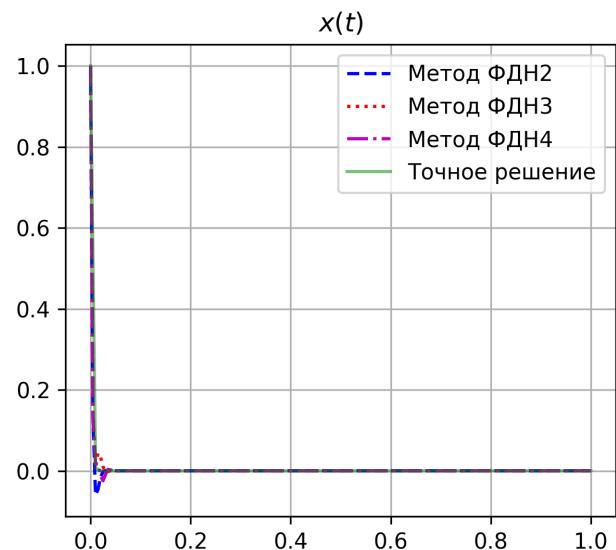
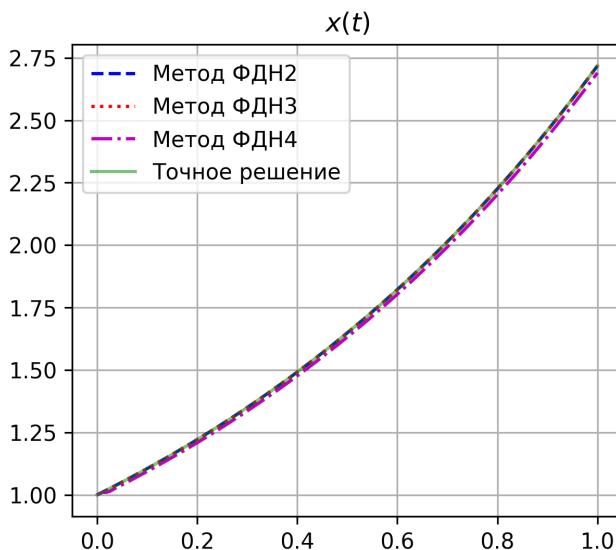
```
In [12]: 1 BDF_koefs_2order = np.array([1/2, -2, 3/2])
2 BDF_koefs_3order = np.array([-1/3, 3/2, -3, 11/6])
3 BDF_koefs_4order = np.array([1/4, -4/3, 3, -4, 25/12])
```

```
In [15]: 1 BDF_order2, BDF_order3, BDF_order4 = BDF_AutonomousSystem(BDF_koefs_2order), BDF_AutonomousSystem(BDF_koefs_3order), B
```

Проверка методов

```
In [ ]: 1 t2, u2 = BDF_order2(test_system, 0, 1, 0.005, [1, 1])
2 t3, u3 = BDF_order3(test_system, 0, 1, 0.005, [1, 1])
3 t4, u4 = BDF_order4(test_system, 0, 1, 0.005, [1, 1])
```

```
In [ ]: 1 plt.figure(figsize=(10,4), dpi=300)
2 tt = np.linspace(0, 1, 100)
3
4 plt.subplot(1, 2, 1)
5 plt.plot(t2, u2[:, 0], 'b--', label='Метод ФДН2')
6 plt.plot(t3, u3[:, 0], 'r:', label='Метод ФДН3')
7 plt.plot(t4, u4[:, 0], 'm-.', label='Метод ФДН4')
8 plt.plot(tt, np.exp(tt), 'g', label='Точное решение', alpha=.5)
9 plt.title('$x(t)$'); plt.legend(); plt.grid()
10
11 plt.subplot(1, 2, 2)
12 plt.plot(t2, u2[:, 1], 'b--', label='Метод ФДН2')
13 plt.plot(t3, u3[:, 1], 'r:', label='Метод ФДН3')
14 plt.plot(t4, u4[:, 1], 'm-.', label='Метод ФДН4')
15 plt.plot(tt, np.exp(-500*tt), 'g', label='Точное решение', alpha=.5)
16 plt.title('$x(t)$'); plt.legend(); plt.grid()
```



Видно, что с повышением порядка аппроксимации методы ФДН все больше теряют в устойчивости.

Функции устойчивости

Для нахождения функции устойчивости ФДН-метода нужно непосредственно подставить модельное уравнение Далквиста $\dot{u} = \lambda u$ в метод, учитывая, что

$$u_{n+m} = [R(z)]^m u_n.$$

(формула для $R(z = \tau\lambda)$ метода РК выводится так же)

Поскольку в этой задаче зависимость $R(z)$ не является однозначной, рассмотрим обратную ей, уже функцию, $z(R)$:

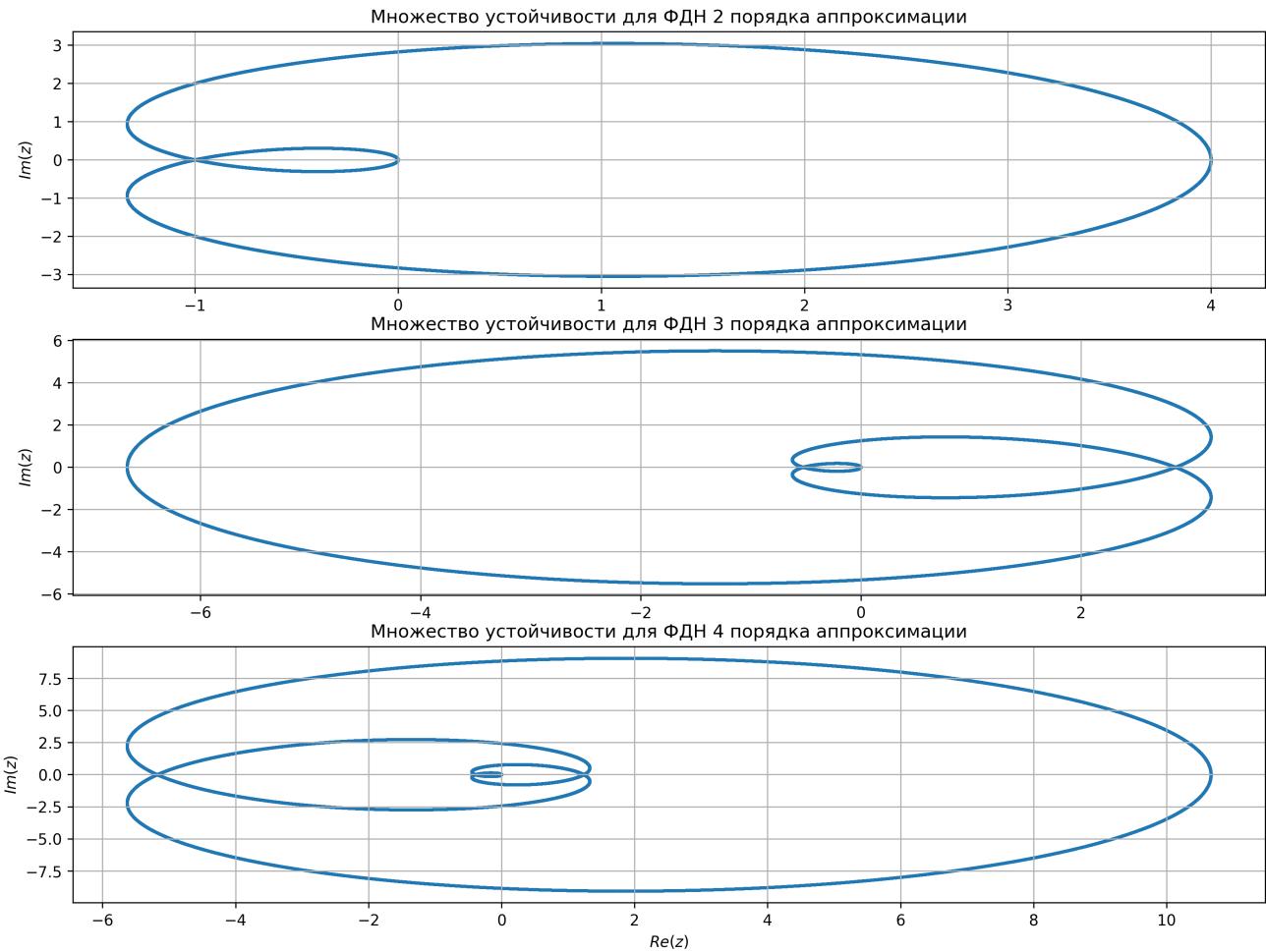
$$\begin{aligned} k &= 2 : \frac{3}{2} - \frac{2}{R} + \frac{1}{2R^2} = z \\ k &= 3 : \frac{11}{6} - \frac{3}{R} + \frac{3}{2R^2} - \frac{1}{3R^3} = z \\ k &= 4 : \frac{25}{12} - \frac{4}{R} + \frac{3}{R^2} - \frac{4}{3R^3} + \frac{1}{4R^4} = z \end{aligned}$$

Для устойчивости нужно, чтобы было $|R(z)| \leq 1$. Рассмотрим, какие точки z переходят в границу единичного круга $|R(z)| = 1$, т.е. $R = e^{i\theta}$.

Построим $z(\theta)$ для $\theta \in [0, 2\pi]$:

```
In [28]: 1 def z(R, koefs):
2     return koefs @ np.array([R**(-i) for i in range(len(koefs))])
```

```
In [35]: 1 R = np.exp(1j*np.random.uniform(0, 2*np.pi, size=100000))
2
3 fig = plt.figure(figsize=(6.4 * 2.2, 4.8 * 2.2), dpi=300)
4 for k, koefs in enumerate([BDF_koefs_2order, BDF_koefs_3order, BDF_koefs_4order]):
5     z_range = z(R, koefs)
6     plt.subplot(3, 1, k+1)
7     plt.scatter(z_range.real, z_range.imag, s=1, alpha=.5)
8     plt.title(f'Множество устойчивости для ФДН {k+2} порядка аппроксимации')
9     plt.ylabel('$Im(z)$')
10    if k == 2:
11        plt.xlabel('$Re(z)$')
12    plt.grid()
```



Рассмотрим для начала множество устойчивости для ФДН 2 порядка. Внутри большой области, не включая петлю, метод является нестабильным. Снаружи этого множества и внутри петли метод устойчив.

Аналогично рекурсивным образом можно понять, какие области являются областями неустойчивости методов более высоких порядков (большое множество - неуст., петля внутри - уст., петля внутри петли - неуст., и т.д.).

Это можно понять и исходя из принципа сохранения области: если при обходе множества $|R(z)| = 1$ область устойчивости оставалась слева, то и при обходе множества z эта область будет слева.

Решение задачи

ФДН-методы

Промежуток времени, на котором решается задача - от $T_0 = 0$ до $T_k = 3000$. Шаг τ будем брать из следующего множества: $\{0.5, 0.1, 0.05, 0.01, 0.001\}$. Начальные условия: $y(0) = 1.7, \alpha(0) = 1, x(0)$ берутся из множества $\{0.1, 0.3, \dots, 0.9\}$. Малый параметр ε будем брать из множества $\{0.01, 0.005, 0.001, 0.0005, 0.0001\}$.

```
In [ ]: 1 from itertools import product
2 from tqdm import tqdm
```

```
In [ ]: 1 def plot_phase_diagrams_and_solutions(x0_arr, epsilon_arr, tau_arr, method):
2     x_dim, y_dim = len(x0_arr), len(epsilon_arr)
3     fig1, axes1 = plt.subplots(x_dim, y_dim)
4     fig2, axes2 = plt.subplots(x_dim, y_dim)
5     fig3, axes3 = plt.subplots(x_dim, y_dim)
6     for fig in [fig1, fig2, fig3]:
7         fig.set_size_inches(6.4 * 2.2, 4.8 * 2.2)
8         fig.set_dpi(300)
9         fig.tight_layout(pad=2)
10    fig1.suptitle(r'Фазовые траектории в зависимости от  $x_0$ ,  $\varepsilon$  и  $\tau$ ', y=1.05)
11    fig2.suptitle(r'Решение  $x(t)$  в зависимости от  $x_0$ ,  $\varepsilon$  и  $\tau$ ', y=1.05)
12    fig3.suptitle(r'Решение  $y(t)$  в зависимости от  $x_0$ ,  $\varepsilon$  и  $\tau$ ', y=1.05)
13    for i, (x0, epsilon) in tqdm(enumerate(product(
14        x0_arr,
15        epsilon_arr
16    )):
17        for tau in tau_arr:
18            t, u = method(f, 0, 3000, tau, np.array([x0, 1.7, 1]), epsilon=epsilon)
19            axes1[i // y_dim, i % y_dim].scatter(u[:, 0], u[:, 1], s=3, alpha=.5, label=r'$\tau = {}'.format(tau))
20            axes2[i // y_dim, i % y_dim].scatter(t, u[:, 0], s=3, alpha=.5, label=r'$\tau = {}'.format(tau))
21            axes3[i // y_dim, i % y_dim].scatter(t, u[:, 1], s=3, alpha=.5, label=r'$\tau = {}'.format(tau))
22        for axes in [axes1, axes2, axes3]:
23            axes[i // y_dim, i % y_dim].legend(loc='upper right', prop={'size': 7})
24            axes[i // y_dim, i % y_dim].set_title(r'$x_0$ = {}, $\varepsilon$ = {}'.format(round(x0, 1), epsilon))
25            axes[i // y_dim, i % y_dim].grid()
26            axes[i // y_dim, i % y_dim].ticklabel_format(useOffset=False)
27        for axes in [axes1, axes2, axes3]:
28            for i, ax in enumerate(axes):
29                if i == len(axes) - 1:
30                    ax[0].set(xlabel=r'$x(t)$', ylabel=r'$y(t)$')
31                    ax[1].set(xlabel=r'$x(t)$')
32                else:
33                    ax[0].set(ylabel=r'$y(t)$')
```

```
In [ ]: 1 for method in [BDF_order2, BDF_order3, BDF_order4]:
2     plot_phase_diagrams_and_solutions(
3         x0_arr=np.arange(0.1, 1.0, 0.2),
4         epsilon_arr=[1e-3, 5e-3, 1e-3, 5e-4, 1e-4],
5         tau_arr=[1, 0.1, 0.01],
6         method=method
7     )
```

0it [00:00, ?it/s]/usr/local/lib/python3.7/dist-packages/scipy/optimize/minpack.py:162: RuntimeWarning: The number of calls to function has reached maxfev = 5000.
 warnings.warn(msg, RuntimeWarning)
 /usr/local/lib/python3.7/dist-packages/scipy/optimize/minpack.py:162: RuntimeWarning: The iteration is not making good progress, as measured by the improvement from the last ten iterations.
 warnings.warn(msg, RuntimeWarning)

1it [01:29, 89.26s/it]/usr/local/lib/python3.7/dist-packages/scipy/optimize/minpack.py:162: RuntimeWarning: The iteration is not making good progress, as measured by the improvement from the last five Jacobian evaluations.
 warnings.warn(msg, RuntimeWarning)

2it [02:37, 83.03s/it]

3it [04:08, 85.48s/it]

4it [05:19, 80.97s/it]

5it [06:35, 79.56s/it]

6it [07:46, 76.81s/it]

7it [08:50, 73.20s/it]

8it [10:00, 72.25s/it]

9it [11:12, 71.93s/it]

10it [12:29, 73.52s/it]

11it [13:40, 72.77s/it]

12it [14:46, 70.70s/it]

13it [15:58, 71.10s/it]

14it [17:10, 71.43s/it]

15it [18:26, 72.93s/it]

16it [19:55, 77.69s/it]

17it [21:00, 73.99s/it]

18it [22:29, 78.27s/it]

19it [23:38, 75.67s/it]

20it [24:54, 75.52s/it]

21it [26:23, 79.72s/it]

22it [27:30, 75.77s/it]

23it [28:58, 79.54s/it]

24it [30:09, 76.90s/it]

25it [31:24, 75.38s/it]

0it [00:00, ?it/s]

1it [01:29, 89.07s/it]

2it [02:32, 81.36s/it]

3it [04:01, 83.59s/it]

4it [05:08, 78.75s/it]

5it [06:20, 76.77s/it]

6it [07:53, 81.68s/it]

7it [08:59, 76.93s/it]

8it [10:29, 80.90s/it]

9it [11:38, 77.12s/it]

10it [12:50, 75.62s/it]

11it [14:23, 80.91s/it]

12it [15:27, 75.68s/it]

13it [16:59, 80.77s/it]

14it [18:06, 76.47s/it]

15it [19:15, 74.32s/it]

16it [20:43, 78.40s/it]

17it [21:45, 73.57s/it]

18it [23:13, 77.84s/it]

19it [24:21, 74.81s/it]

20it [25:31, 73.44s/it]

21it [26:40, 71.98s/it]

22it [27:44, 69.83s/it]

23it [28:54, 69.81s/it]

24it [30:02, 69.18s/it]

25it [31:13, 74.94s/it]

0it [00:00, ?it/s]

1it [01:33, 93.84s/it]

2it [02:38, 85.12s/it]

3it [04:10, 87.15s/it]

4it [05:17, 81.11s/it]

5it [06:28, 78.15s/it]

6it [07:36, 74.95s/it]

7it [08:38, 71.18s/it]

8it [09:47, 70.47s/it]

9it [10:54, 69.46s/it]

10it [12:03, 69.36s/it]

11it [13:11, 69.04s/it]

12it [14:15, 67.26s/it]

13it [15:23, 67.56s/it]

14it [16:30, 67.35s/it]

15it [17:40, 68.18s/it]

16it [18:48, 68.18s/it]

17it [19:50, 66.40s/it]

18it [20:58, 66.88s/it]

19it [22:05, 66.90s/it]

20it [23:15, 67.69s/it]

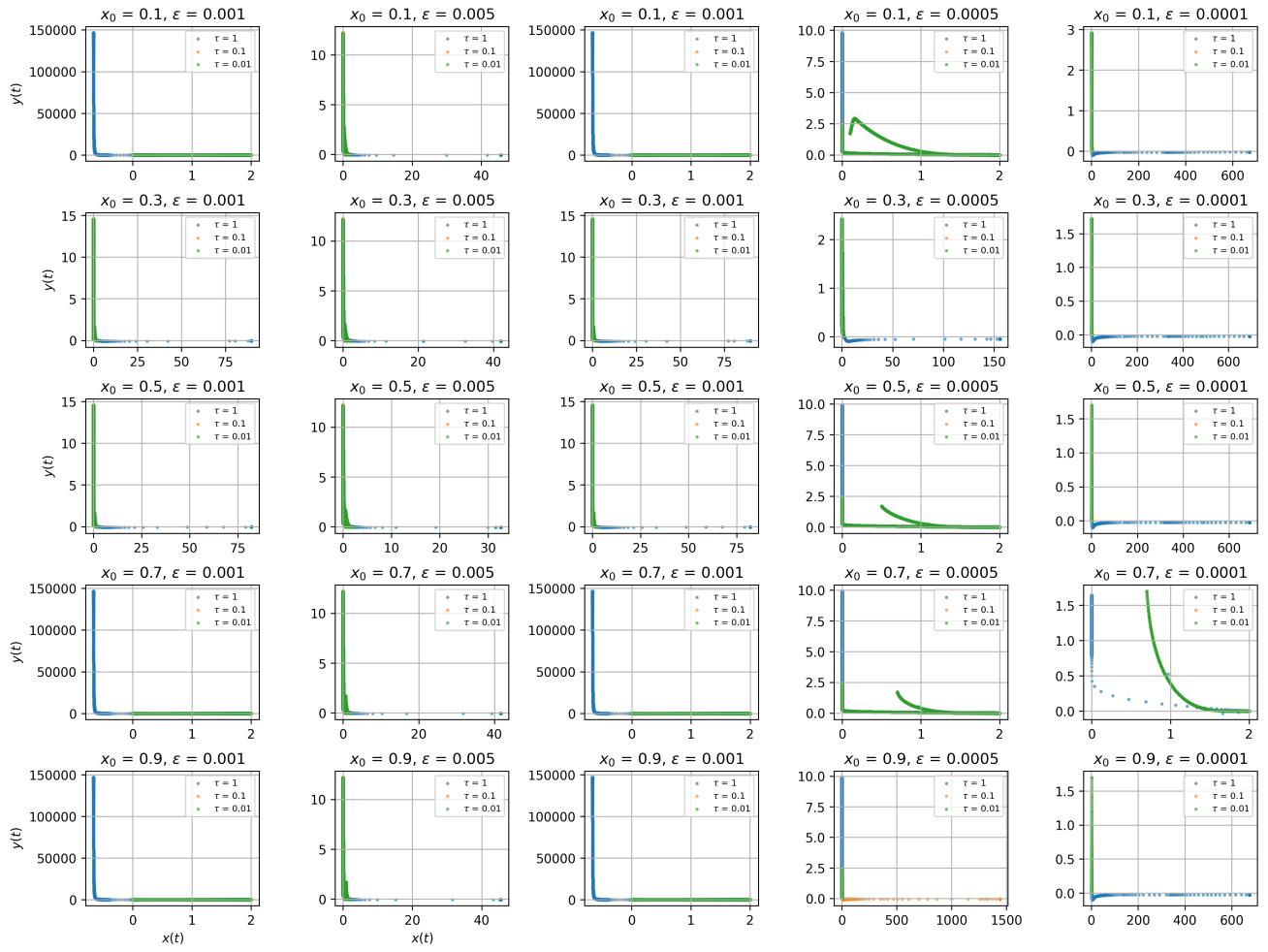
21it [24:49, 75.72s/it]

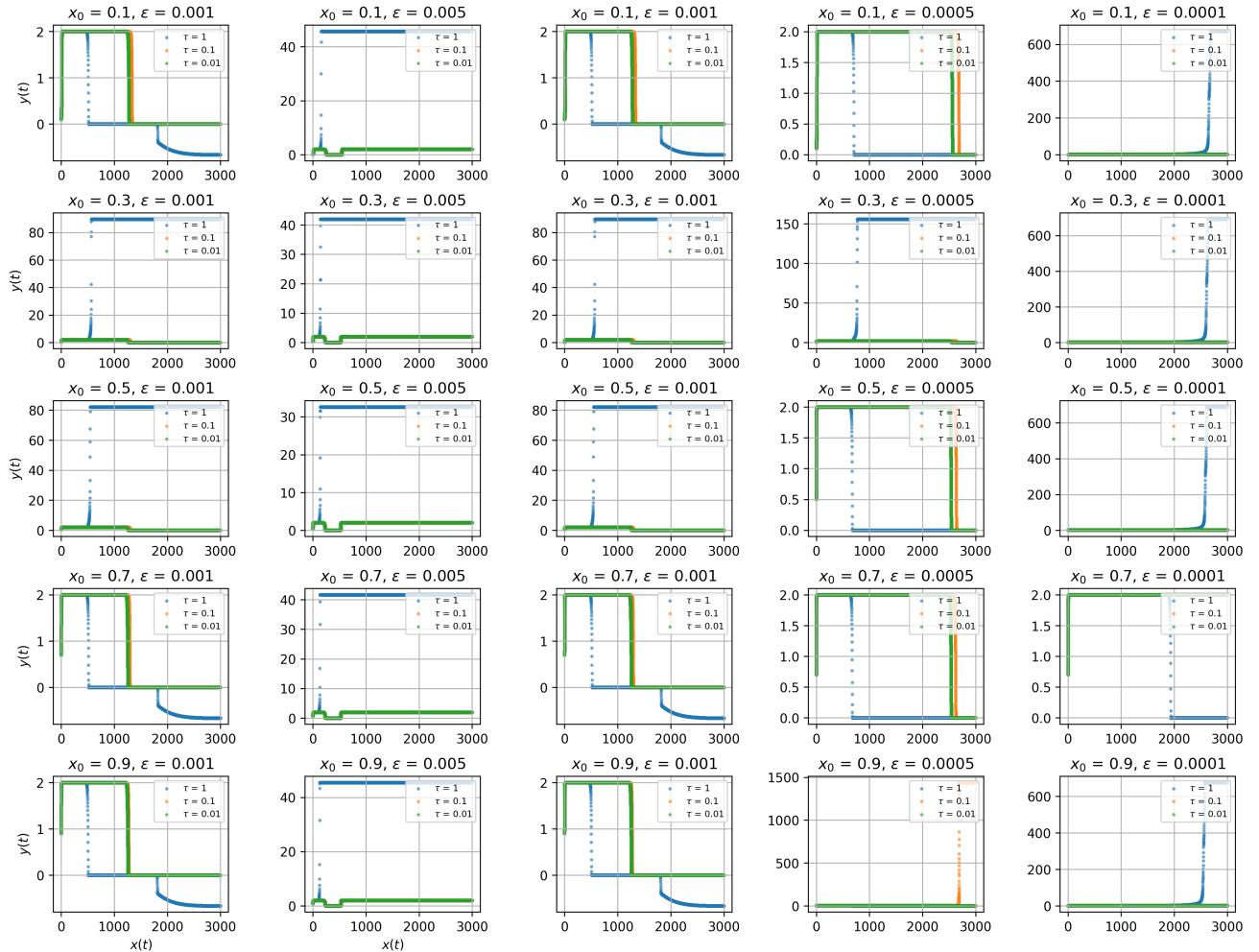
22it [25:52, 71.96s/it]

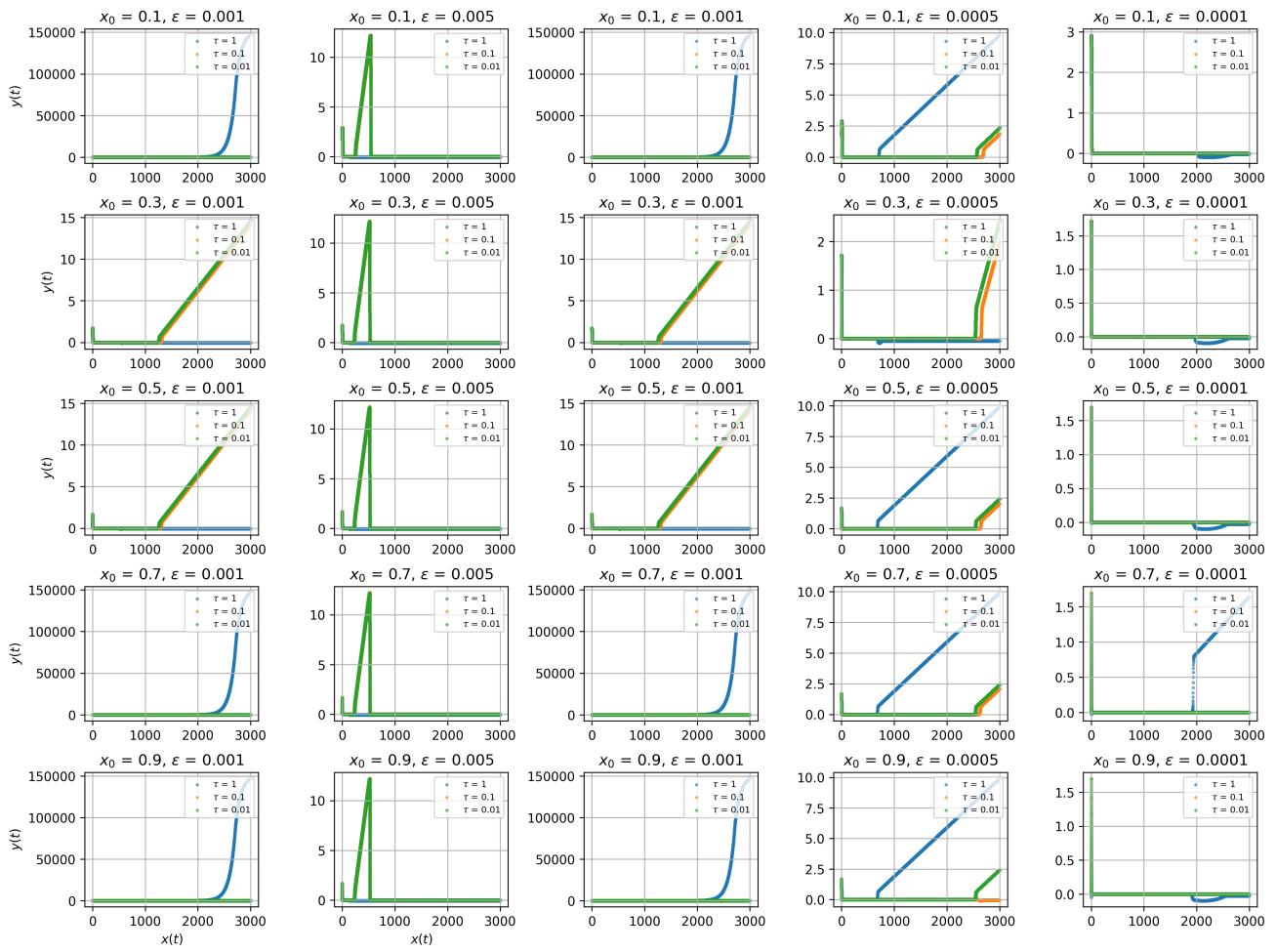
23it [27:25, 78.22s/it]

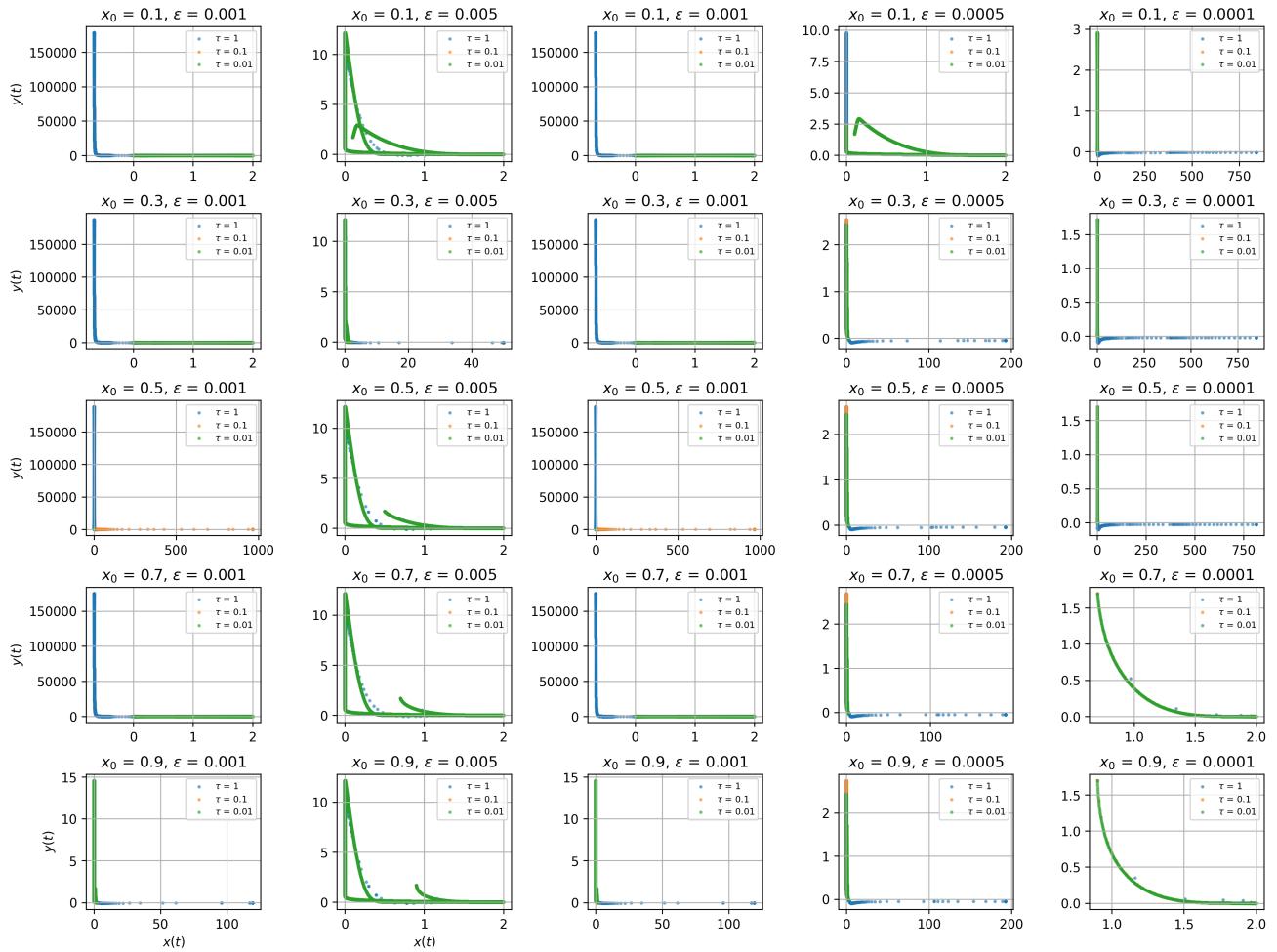
24it [28:32, 74.89s/it]

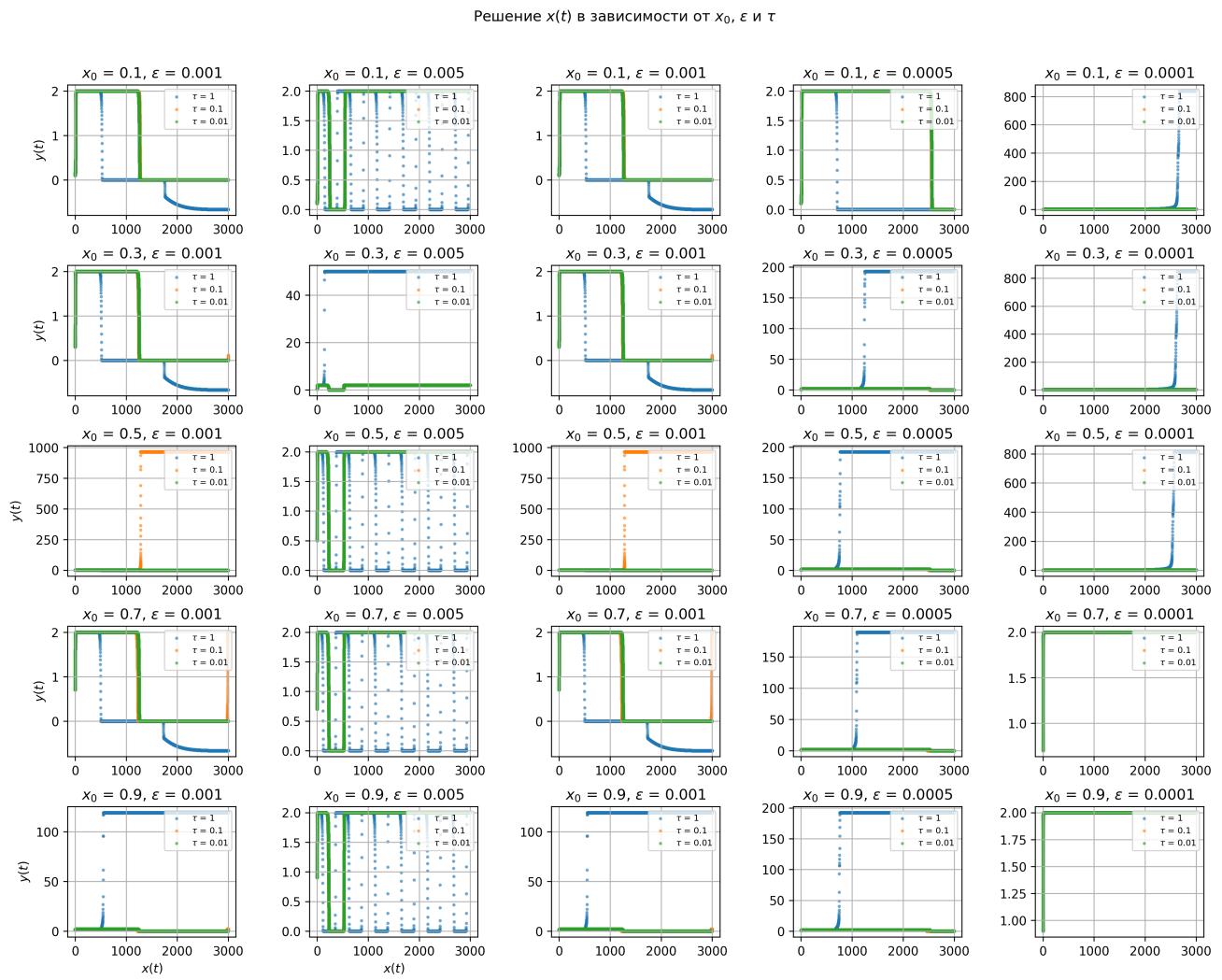
25it [29:43, 71.33s/it]

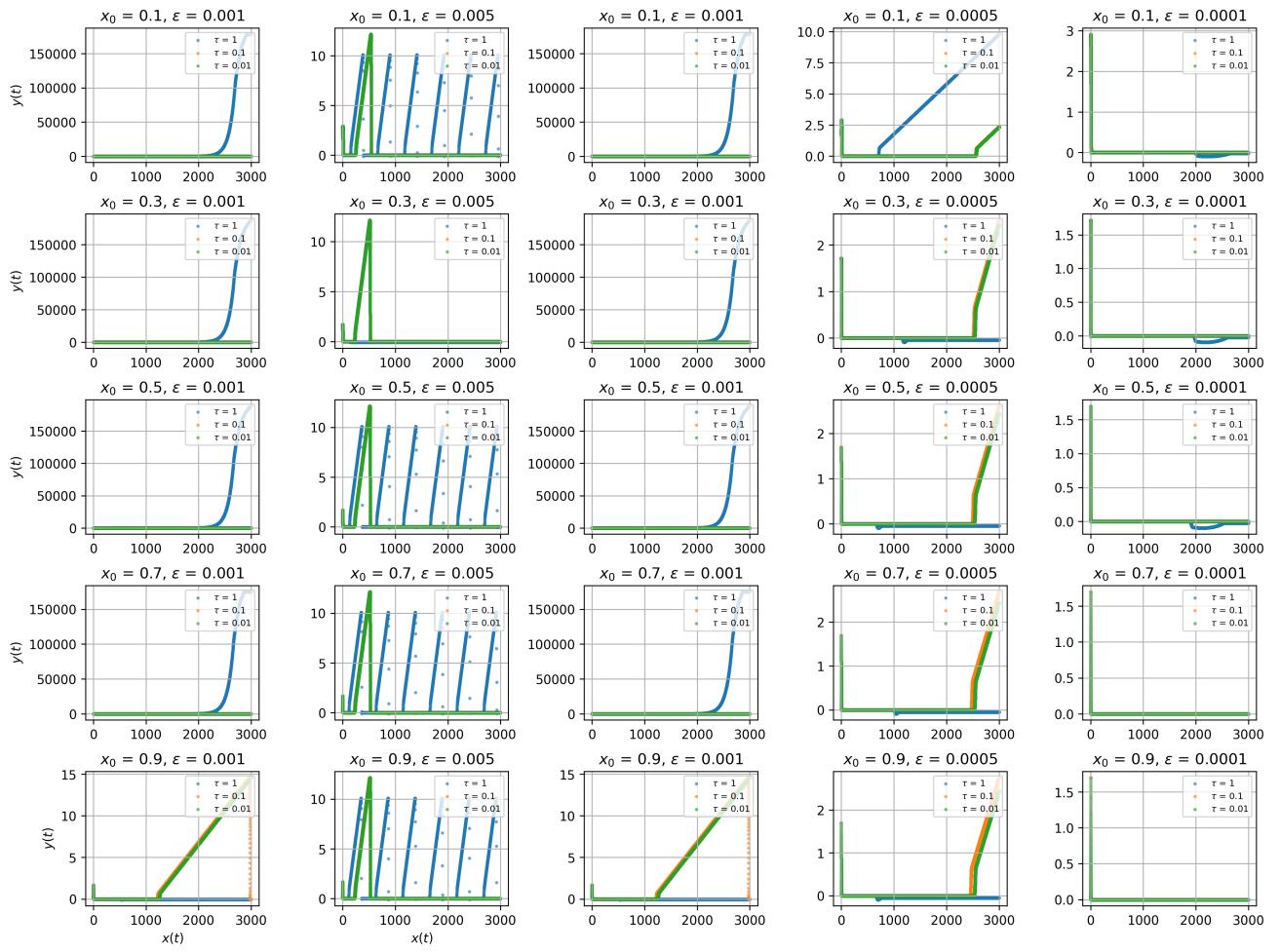
Фазовые траектории в зависимости от x_0 , ε и τ 

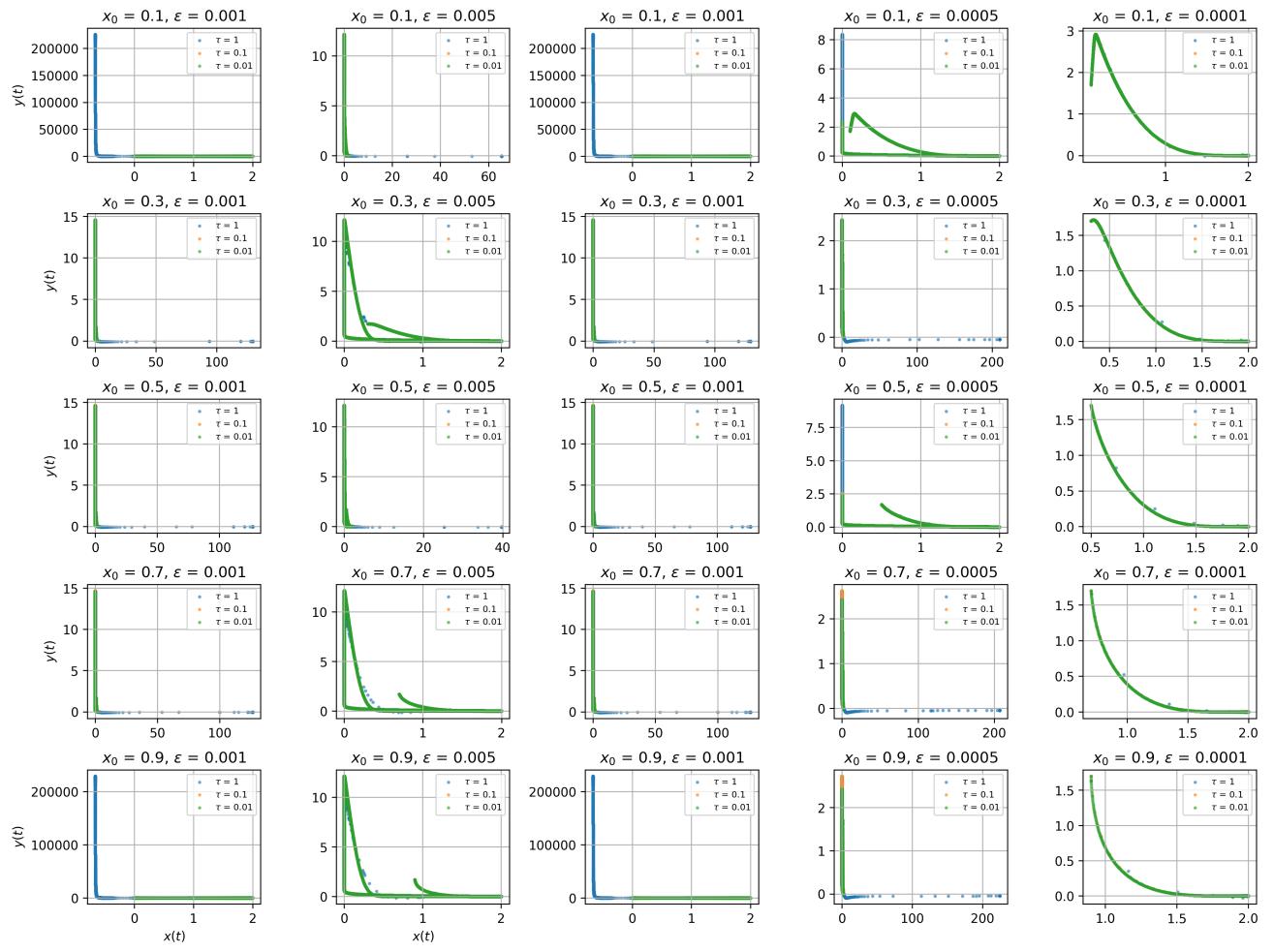
Решение $x(t)$ в зависимости от x_0 , ε и τ 

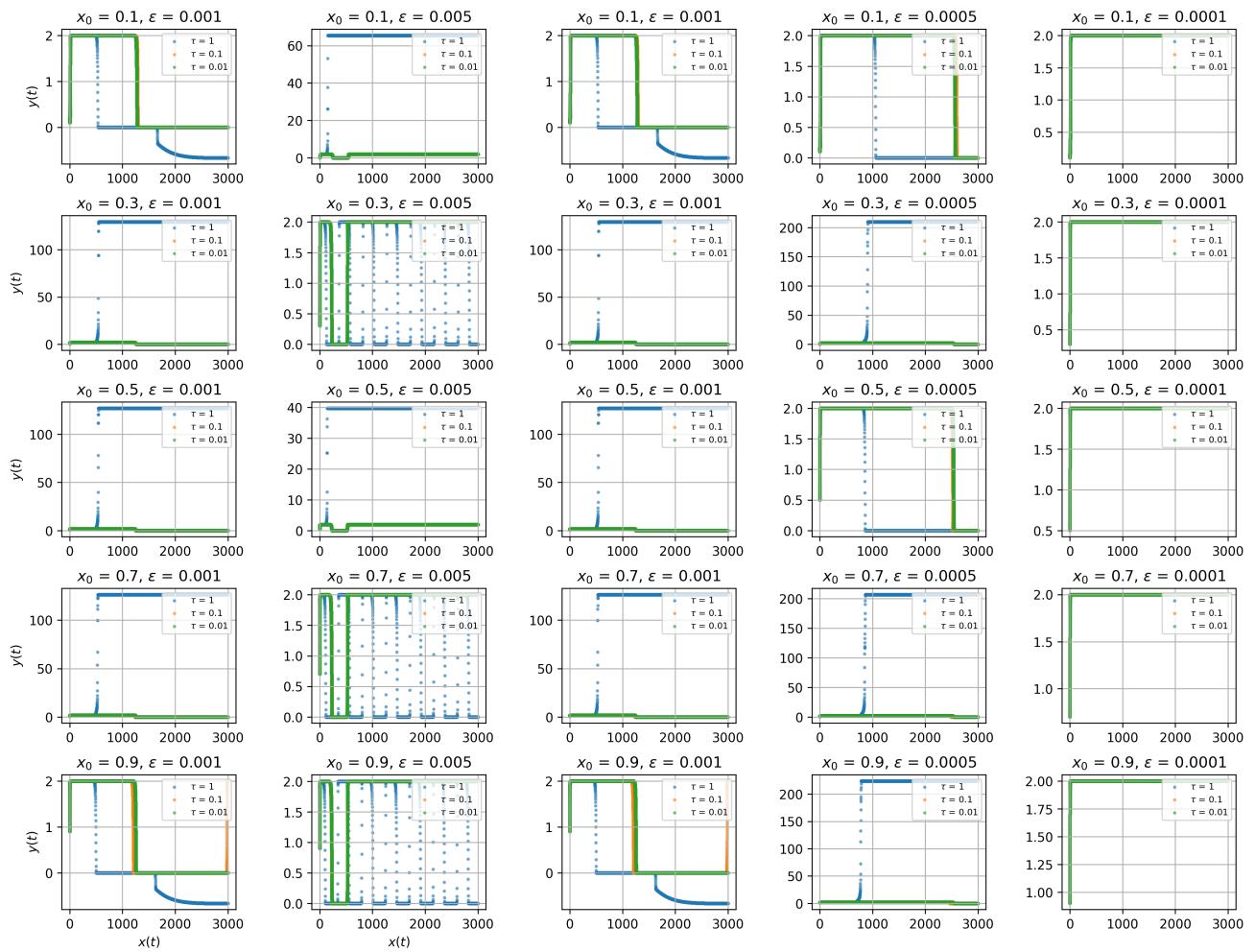
Решение $y(t)$ в зависимости от x_0 , ε и τ 

Фазовые траектории в зависимости от x_0 , ε и τ 



Решение $y(t)$ в зависимости от x_0, ε и τ 

Фазовые траектории в зависимости от x_0 , ε и τ 

Решение $x(t)$ в зависимости от x_0 , ε и τ 

Решение $y(t)$ в зависимости от x_0, ε и τ 