Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №2 по курсу "Моделирование"

Тема Программно-алгоритмическая реализация метода Рунге-Кутта 4-го порядка точности при решении задачи Коши для системы ОДУ

Студент Перестеронин П.Г.

Группа ИУ7-63Б

Оценка ____________

Преподаватель Градов В.М.

Москва — 2021 г.

Оглавление

Тема работы

Программно-алгоритмическая реализация метода Рунге-Кутта 4-го порядка точности при решении системы ОДУ в задаче Коши.

Цель работы

Получение навыков разработки алгоритмов решения задачи Коши при реализации моделей, построенных на системе ОДУ, с использованием метода Рунге-Кутта 4-го порядка точности.

1 Теоретические сведения

Опишем колебательный контур с помощью системы уравнений:

$$\begin{cases} L_k \frac{dI}{dt} + (R_k + R_p(I)) \cdot I - U_C = 0 \\ \frac{dU_c}{dt} = -\frac{I}{C_k} \end{cases}$$

Значение $R_p(I)$ можно вычислить по формуле:

$R_p = \frac{l_e}{2\pi \cdot \int_0^R \sigma(T(r)) r dr} = \frac{l_e}{2\pi R^2 \cdot \int_0^1 \sigma(T(z)) dz}$

т. к. $z = r/R$.

Значение $T(z)$ вычисляется по формуле:

$T(z) = T_0 + (T_w - T_0) \cdot Z^m$

Заданы начальные параметры:

R = 0.35 см (Радиус трубки)
l_e = 12 см (Расстояние между электродами лампы)
L_k = 187e-6 Гн (Индуктивность)
C_k = 268e-6 Ф (Емкость конденсатора)
R_k = 0.25 Ом (Сопротивление)
U_{c0} = 1400 В (Напряжение на конденсаторе в начальный момент времени)
I_0 = 0..3 А (Сила тока в цепи в начальный момент времени t = 0)
T_w = 2000 K

1.1 Метод Рунге-Кутта четвертого порядка точности

Имеем систему уравнений вида:

$$\begin{cases} u'(x) = f(x, u(x)) \\ u(\xi) = \eta \end{cases}$$

Тогда:

$y_{n+1} = y_n + \frac{k_1+2k_2+2k_3+k_4}{6}$
$k_1 = h_n f(x_n, y_n)$
$k_2 = h_n f(x_n + \frac{h_n}{2}, y_n + \frac{k_1}{2})$
$k_3 = h_n f(x_n + \frac{h_n}{2}, y_n + \frac{k_2}{2})$
$k_4 = h_n f(x_n + h_n, y_n + k_3)$

Рассмотрим обобщение формулы на случай двух переменных. Пусть дана система:

$$\begin{cases} u'(x) = f(x, u, v) \\ v'(x) = \varphi(x, u, v) \\ v(\xi) = v_0 \\ u(\xi) = u_0 \end{cases}$$

Тогда:

$y_{n+1} = y_n + \frac{k_1+2k_2+2k_3+k_4}{6}$
$z_{n+1} = z_n + \frac{q_1+2q_2+2q_3+q_4}{6}$
$k_1 = h_n f(x_n, y_n, z_n)$
$k_2 = h_n f(x_n + \frac{h_n}{2}, y_n + \frac{k_1}{2}, z_n + \frac{q_1}{2})$
$k_3 = h_n f(x_n + \frac{h_n}{2}, y_n + \frac{k_2}{2}, z_n + \frac{q_2}{2})$
$k_4 = h_n f(x_n + h_n, y_n + k_3, z_n + q_3)$
$q_1 = h_n \varphi(x_n, y_n, z_n)$
$q_2 = h_n \varphi(x_n + \frac{h_n}{2}, y_n + \frac{k_1}{2}, z_n + \frac{q_1}{2})$

$$q_3 = h_n\varphi(x_n + \tfrac{h_n}{2}, y_n + \tfrac{k_2}{2}, z_n + \tfrac{q_2}{2})$$
$$q_4 = h_n\varphi(x_n + h_n, y_n + k_3, z_n + q_3)$$

2 Реализация

2.1 Код программы

Ниже представлены исходные коды программы на языке `Elixir`.

Листинг 2.1: Основной модуль приложения

```
defmodule RungeKutta.Application do
  use Application
  import RungeKutta.Runners

  def start(_, _) do
    run()
    {:ok, self()}
  end

  def run() do
    [from, to, step] =
      IO.gets("Input boundaries and step using space as separator: ")
      |> String.trim()
      |> String.split()
      |> Enum.map(fn str_float ->
        {num, _} = Float.parse(str_float)
        num
      end)

    run_simple_graphs(from, to, step)
    #run_const_resistance(from, to, step, 0)
    #run_const_resistance(from, to, step, 200)
  end
end
```

Листинг 2.2: Модуль с интерполяционными функциями

```
defmodule RungeKutta.InterpolatedFuncs do
  import RungeKutta.Integral, only: [trapezoid: 4]
  @step 0.05

  @r 0.35
  @lp 12
  @tw 2000

  @t0_I [
    {0.5, 6730},
    {1, 6790},
    {5, 7150},
```

```
    {10, 7270},
    {50, 8010},
    {200, 9185},
    {400, 10010},
    {800, 11140},
    {1200, 12010}
  ]
  @m_I [
    {0.5, 0.5},
    {1, 0.55},
    {5, 1.7},
    {10, 3},
    {50, 11},
    {200, 32},
    {400, 40},
    {800, 41},
    {1200, 39}
  ]
  @sigma_T [
    {4000, 0.031},
    {5000, 0.27},
    {6000, 2.05},
    {7000, 6.06},
    {8000, 12.0},
    {9000, 19.9},
    {10000, 29.6},
    {11000, 41.1},
    {12000, 54.1},
    {13000, 67.7},
    {14000, 81.5}
  ]

  def m(i) do
    linear_interpolation(@m_I, i)
  end

  def t0(i) do
    linear_interpolation(@t0_I, i)
  end

  def sigma(z, t0_val, m_val) do
    t = t0_val + (@tw - t0_val) * :math.pow(z, m_val)
    linear_interpolation(@sigma_T, t)
  end

  def rp(i) do
    t0_val = t0(i)
    m_val = m(i)
```

```
    integral = trapezoid(0, 1, @step, fn z -> sigma(z, t0_val, m_val) * z end)
    @lp / (2 * :math.pi() * @r * @r * integral)
  end

  def linear_interpolation(table, arg) do
    {{x1, y1}, {x2, y2}} = find_closest_pair(table, arg)
    y1 + (y2 - y1) / (x2 - x1) * (arg - x1)
  end

  def find_closest_pair(table, arg) do
    table
    |> Stream.zip(Stream.drop(table, 1))
    |> Enum.reduce_while(nil, fn {{x1, _}, _} = pair, acc ->
      cond do
        x1 >= arg ->
          case acc do
            nil -> {:halt, pair}
            _ -> {:halt, acc}
          end
        true -> {:cont, pair}
      end
    end)
  end
end
```

Листинг 2.3: Модуль с функциями прогонки с разными параметрами

```
defmodule RungeKutta.Runners do
  import RungeKutta.Helper
  import RungeKutta.Solver
  alias RungeKutta.Plot
  import RungeKutta.MainFuncs, only: [f_const: 4]

  def run_simple_graphs(from, to, step) do
    xs = float_range_generator(from, to, step)
    {ys, zs} = Enum.unzip(generate_iu(from, to, step, &RungeKutta.MainFuncs.f/3))
    rpns = generate_rp(from, to, step)
    t0s = generate_t0(from, to, step)
    #xs
    #|> Enum.zip(rpns)
    #|> Enum.zip(t0s)
    #|> Enum.zip(ys)
    #|> Enum.zip(zs)
    #|> Enum.map(fn {{{{x, rp}, t0}, y}, z} ->
      #IO.puts "x = #{x}, Rp = #{rp}, T0 = #{t0}, I = #{y}, U = #{z};"
    #end)
    plots = Plot.init_plot_collection()
    plots = Plot.plot(plots, xs, ys, "I")
    #plots = Plot.plot(plots, xs, zs, "U")
```

```
    #plots = Plot.plot(plots, xs, rpns, "Rp")
    rp_mul_i = rpns |> Stream.zip(ys) |> Enum.map(fn {rp, i} -> rp * i end)
    #plots = Plot.plot(plots, xs, rp_mul_i, "Rp * I")
    # yn * zn
    #plots = Plot.plot(plots, xs, t0s, "T0")
    Plot.show_all(plots)
  end

  def run_const_resistance(from, to, step, c \\ 0) do
    xs = float_range_generator(from, to, step)
    {ys, zs} = Enum.unzip(generate_iu(from, to, step, fn x, y, z -> f_const(x, y, z, c)
        end))
    plots = Plot.init_plot_collection()
    plots = Plot.plot(plots, xs, ys, "I (R = #{c})")
    plots = Plot.plot(plots, xs, zs, "U (R = #{c})")
    Plot.show_all(plots)
  end
end
```

Листинг 2.4: Модуль с методами численного интегрирования

```
defmodule RungeKutta.Integral do
  import RungeKutta.Helper, only: [float_range_map: 5]

  def trapezoid(from, to, step, func) do
    float_range_map(from, to + step / 2, step, [], fn val, _acc -> func.(val) end)
    |> Stream.chunk_every(2, 1, :discard)
    |> Enum.reduce(0, fn [a, b], acc -> acc + step * (a + b) / 2 end)
  end

  def simpson(from, to, step, func) do
    n = Float.round((to - from) / step)

    float_range_map(from, to + step / 4, step / 2, [], fn val, _acc -> func.(val) end)
    |> Enum.reverse()
    |> (fn vals -> (to - from) / 6 / n * count_all_sums(vals) end).()
  end

  defp count_all_sums(vals) do
    all = Enum.reduce(vals, &Kernel.+/2)
    odds = Stream.drop(vals, 1) |> Stream.take_every(2) |> Enum.reduce(&Kernel.+/2)
    vals = Stream.drop(vals, 2) |> Enum.reverse()
    evens = Stream.drop(vals, 2) |> Stream.take_every(2) |> Enum.reduce(&Kernel.+/2)
    all + odds + 3 * evens
  end
end
```

Листинг 2.5: Модуль с генераций искомых значений

```elixir
defmodule RungeKutta.Solver do
  @u0 1400
  @i0 0.5
  import RungeKutta.Helper, only: [float_range_map: 5]
  import RungeKutta.InterpolatedFuncs, only: [t0: 1, rp: 1]
  import RungeKutta.MainFuncs, only: [phi: 3]

  def generate_t0(from, to, step) do
    generate_iu(from, to, step, &RungeKutta.MainFuncs.f/3)
    |> Stream.map(fn {yn, _zn} -> yn end)
    |> Enum.map(fn i -> t0(i) end)
  end

  def generate_iu(from, to, step, f) do
    float_range_map(from, to, step, [{@i0, @u0}], fn xn, [{yn, zn} | _] ->
      k1 = step * f.(xn, yn, zn)
      p1 = step * phi(xn, yn, zn)
      k2 = step * f.(xn + step / 2, yn + k1 / 2, zn + p1 / 2)
      p2 = step * phi(xn + step / 2, yn + k1 / 2, zn + p1 / 2)
      k3 = step * f.(xn + step / 2, yn + k2 / 2, zn + p2 / 2)
      p3 = step * phi(xn + step / 2, yn + k2 / 2, zn + p2 / 2)
      k4 = step * f.(xn + step, yn + k3, zn + p3)
      p4 = step * phi(xn + step, yn + k3, zn + p3)

      {yn + (k1 + 2 * k2 + 2 * k3 + k4) / 6, zn + (p1 + 2 * p2 + 2 * p3 + p4) / 6}
    end)
    |> Enum.reverse()
  end

  def generate_rp(from, to, step) do
    generate_iu(from, to, step, &RungeKutta.MainFuncs.f/3)
    |> Stream.map(fn {yn, _zn} -> yn end)
    |> Enum.map(fn i -> rp(i) end)
  end
end
```

Листинг 2.6: Модуль с функциями тока и напряжения

```elixir
defmodule RungeKutta.MainFuncs do
  @ck 268.0e-6
  @rk 0.25 / 10.0
  @lk 187.0e-6
  import RungeKutta.InterpolatedFuncs, only: [rp: 1]

  def f(_x, u, v) do
    rpp = rp(u)
    (v - (@rk + rpp) * u) / @lk
  end
```

```
  def f_const(_x, u, v, c) do
    (v - c * u) / @lk
  end

  def phi(_x, u, _v) do
    -u / @ck
  end
end
```

2.2 Результаты работы программы

На рисунках 2.1 – 2.5 представлены графики зависимости от времени импульса t: $I(t)$, $U(t)$, $R_p(t)$, $I(t) \cdot R_p(t)$, $T_0(t)$ при исходных данных. Интервал: $[0, 0.0008]$, шаг $h = 10^{-6}$.

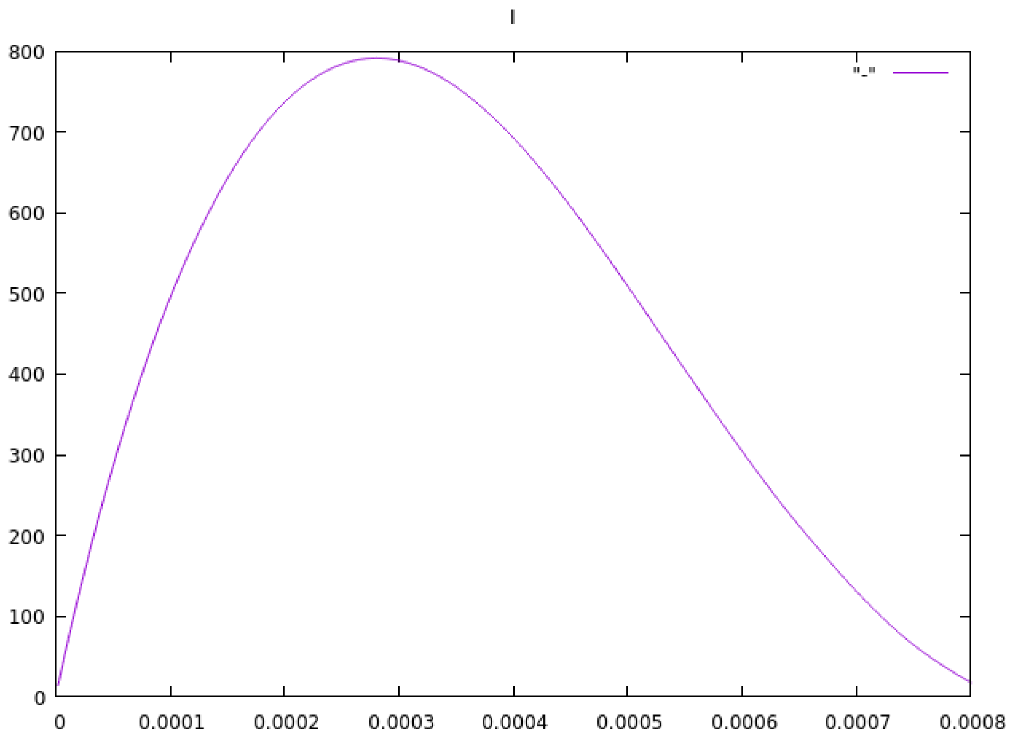

Рис. 2.1: График $I(t)$

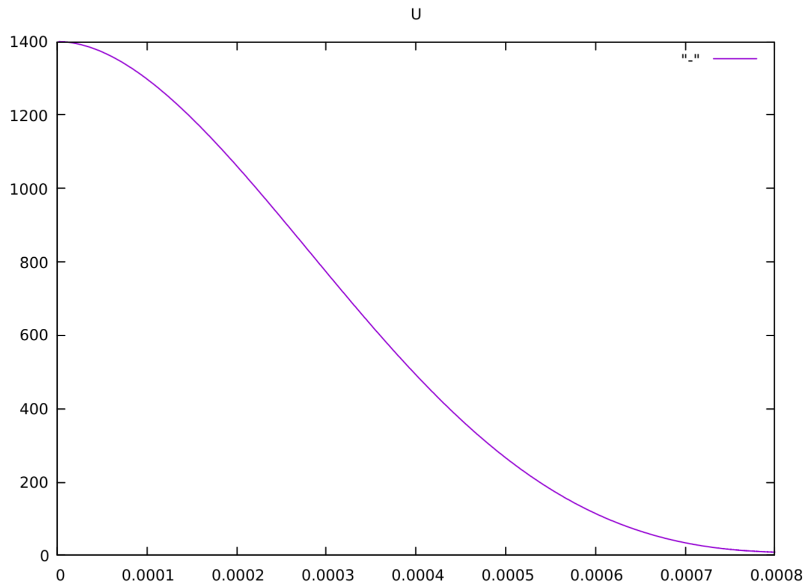

Рис. 2.2: График $U(t)$

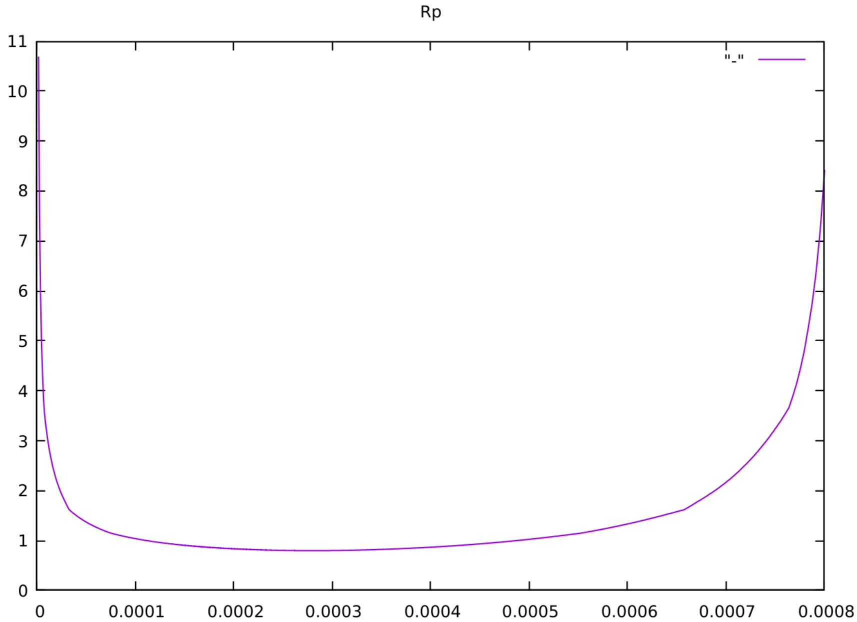

Рис. 2.3: График $R_p(t)$

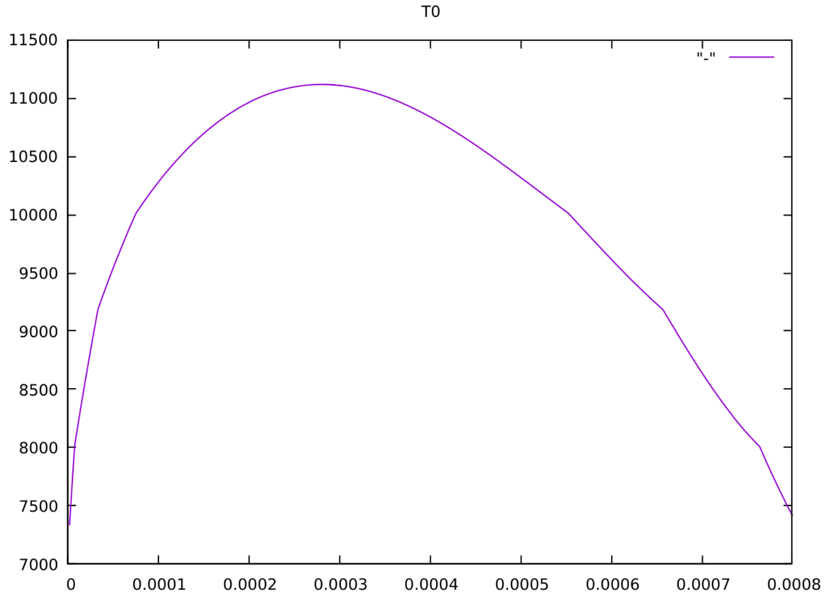

Рис. 2.4: График $T0(t)$

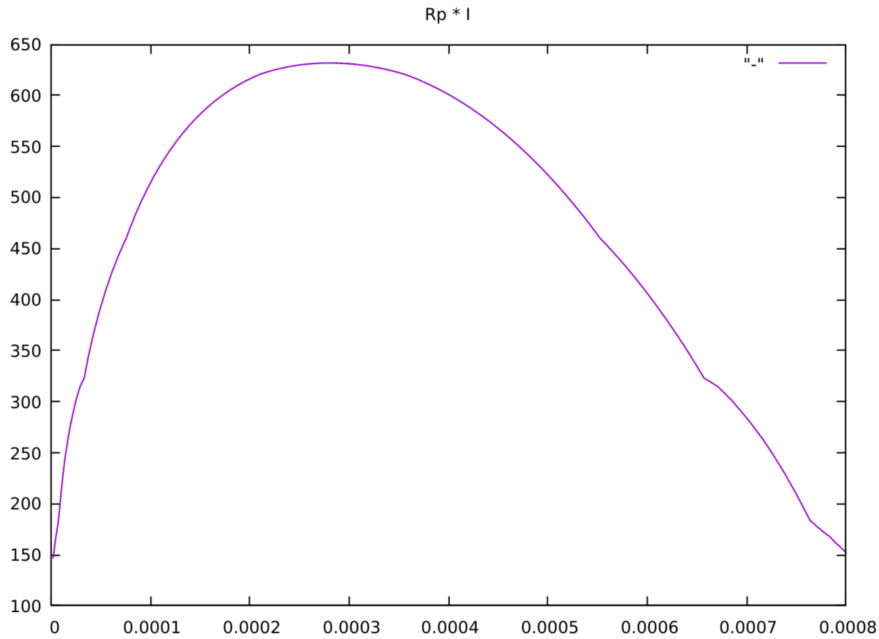

Рис. 2.5: График $I(t) \cdot R_p(t)$

На рисунке 2.6 представлен график $I(t)$, при $R_k + R_p = 0$. Интервал: [0, 0.0008], шаг $h = 10^{-6}$.

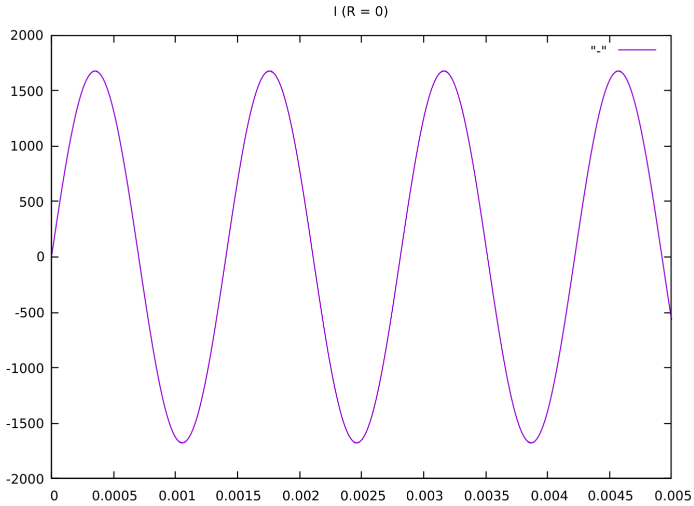

Рис. 2.6: График зависимости $I(t)$ при $R = 0$

На рисунке 2.7 представлен график $I(t)$, при $R_k + R_p = 200$. Интервал: [0, 0.00002], шаг $h = 10^{-7}$.

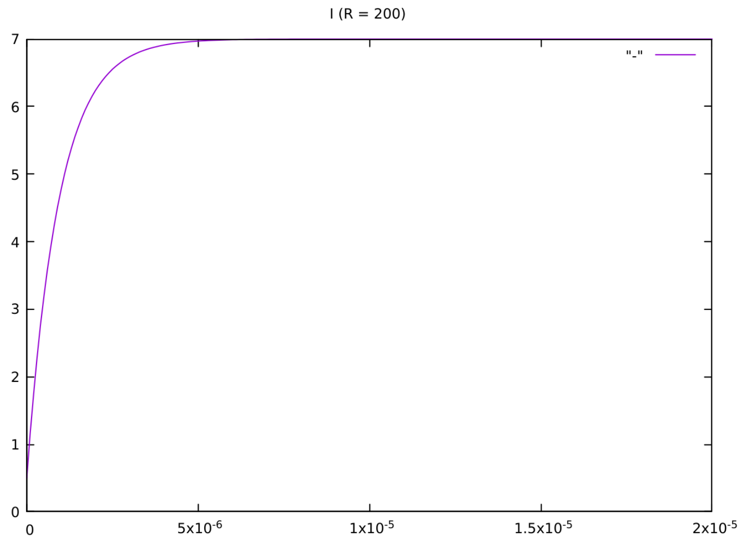

Рис. 2.7: График зависимости $I(t)$ при $R = 200$

На рисунках 2.8 - 2.14 представлены результаты исследования влияния параметров контура C_k, L_k и R_k на длительность импульса t.

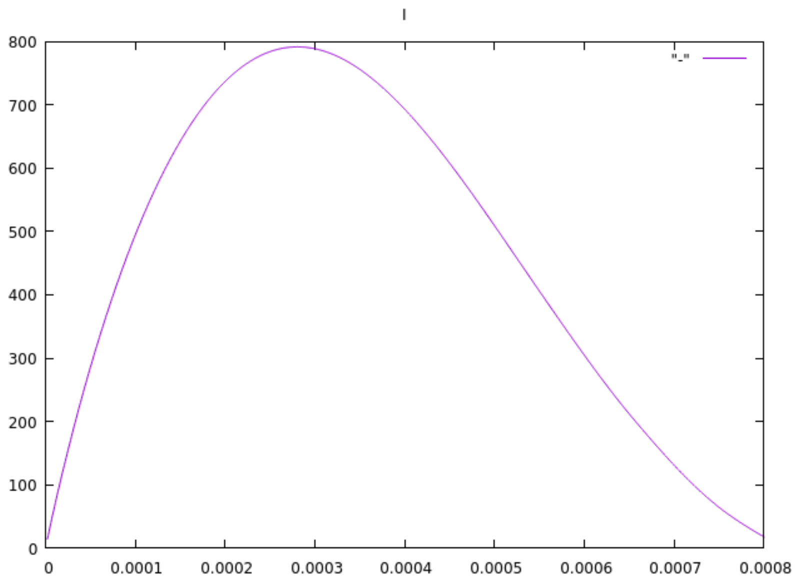

Рис. 2.8: График зависимости $I(t)$ при начальных параметрах

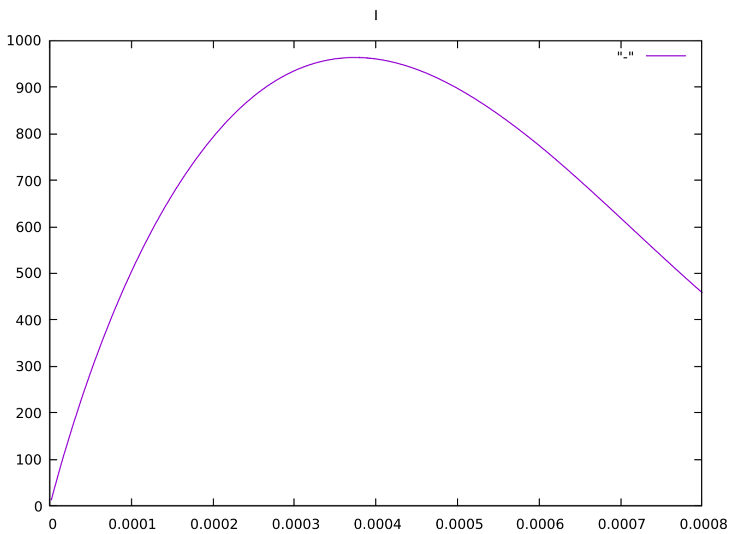

Рис. 2.9: График зависимости $I(t)$ при увеличении начального значения C_k в 2 раза.

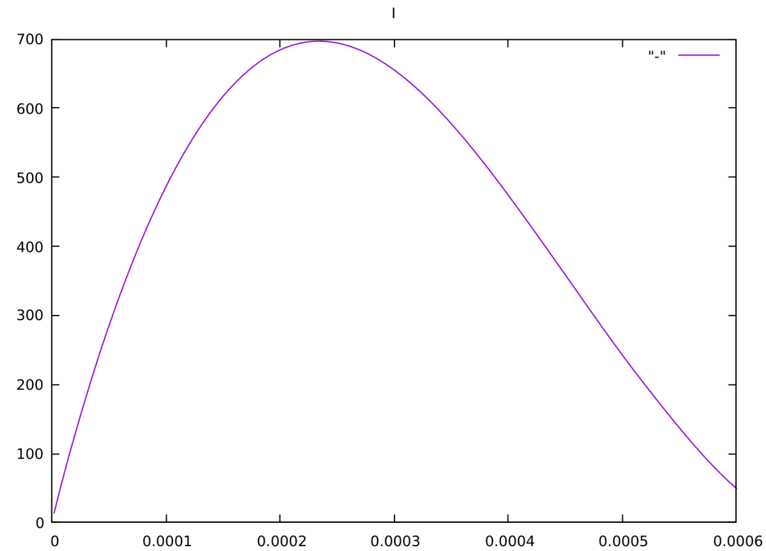

Рис. 2.10: График зависимости $I(t)$ при уменьшении начального значения C_k в 1.5 раза.

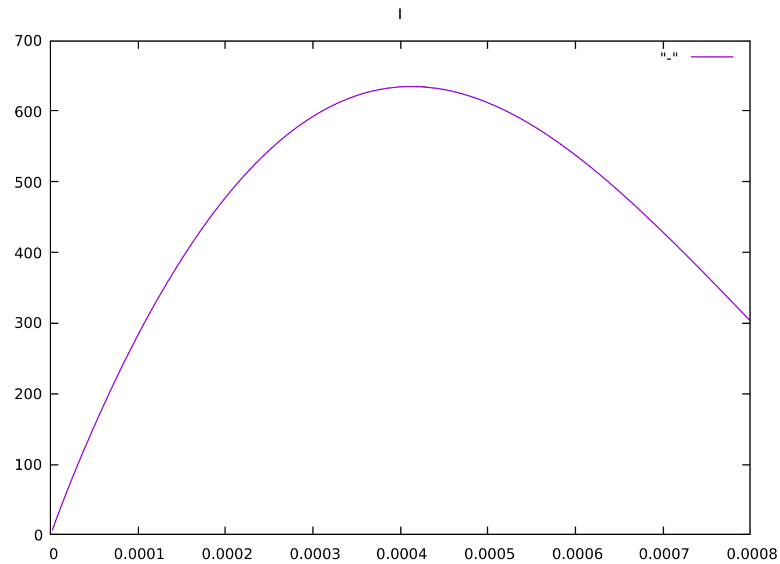

Рис. 2.11: График зависимости $I(t)$ при увеличении начального значения L_k в 2 раза

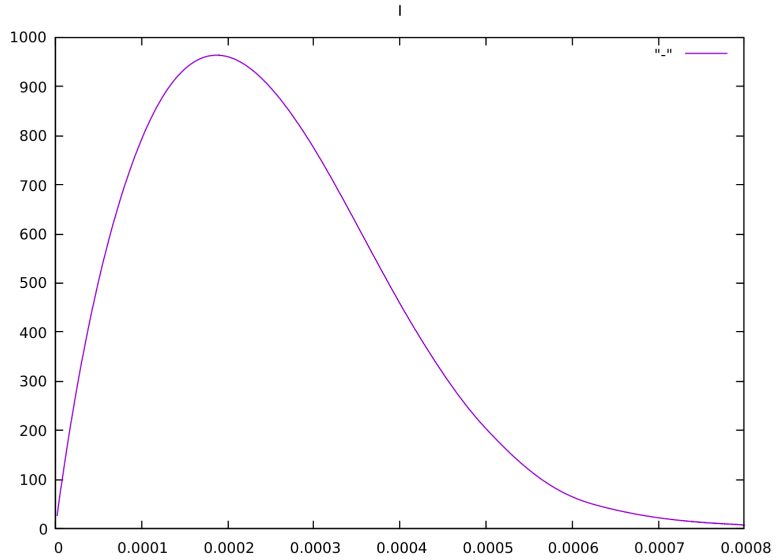

Рис. 2.12: График зависимости $I(t)$ при уменьшении начального значения L_k в 2 раза

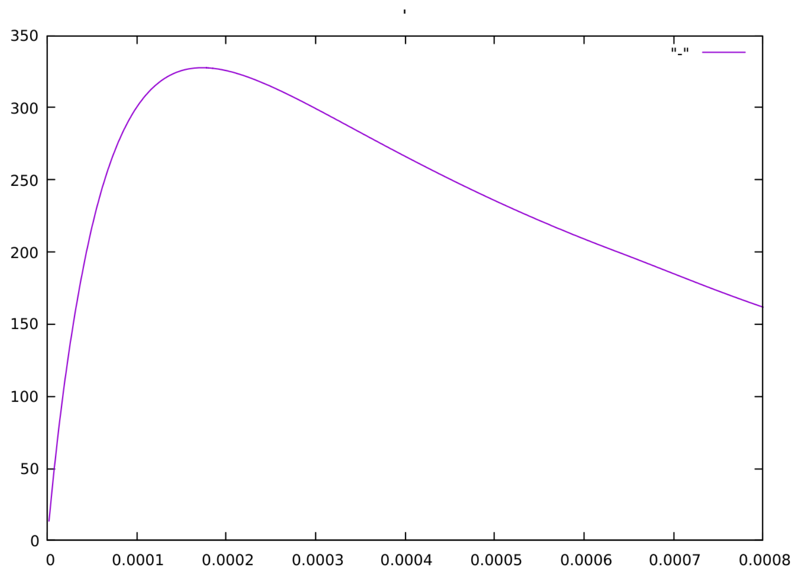

Рис. 2.13: График зависимости $I(t)$ при увеличении начального значения R_k в 10 раз

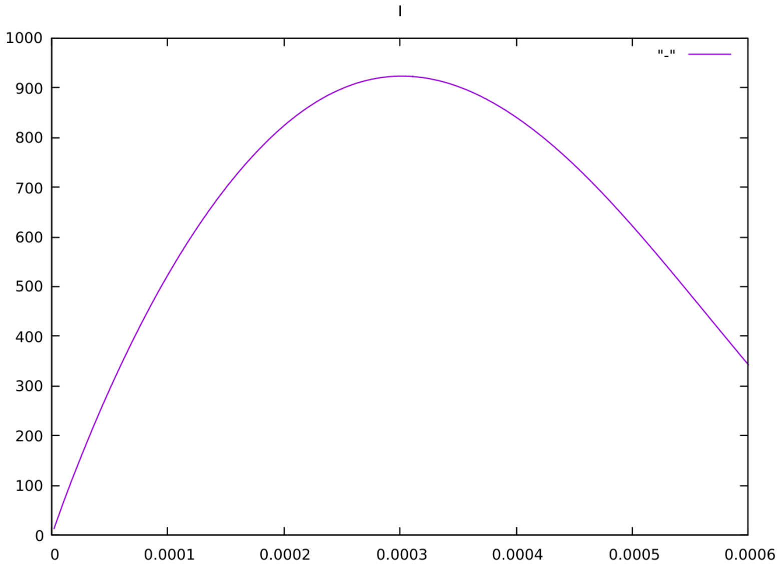

Рис. 2.14: График зависимости $I(t)$ при уменьшении начального значения R_k в 10 раз

Выводы:

- увеличение C_k приводит к увеличению длительности импульса t;
- уменьшение C_k приводит к уменьшению длительности импульса t;
- увеличение L_k приводит к увеличению длительности импульса t;
- уменьшение L_k приводит к уменьшению длительности импульса t;
- увеличение R_k приводит к увеличению длительности импульса t;
- уменьшение R_k приводит к уменьшению длительности импульса t;

3 Ответы на вопросы

1. Какие способы тестирования программы, кроме указанного в п. 2, можете предложить ещё?

Мы можем влиять на цепь главным образом за счет добавления/удаления объектов, обладающих сопротивлением (лампа, резистор): если R_k мало, то возникнут затухающие колебания, если велико R_k — апериодическое затухание.

2. Получите систему разностных уравнений для решения сформулированной задачи неявным методом трапеций. Опишите алгоритм реализации полученных уравнений.

$$U_{n+1} = U_n + \frac{h}{2} + f(x_n, u_n) + f(x_{n+1}, u_{n+1}) + O(h^2) \tag{3.1}$$

$$\begin{cases} \frac{dI}{dT} = \frac{U-(R_k+R_p(I))I}{L_k} \\ \frac{dU}{dt} = -\frac{I}{C_k} \end{cases} \tag{3.2}$$

$$I_{n+1} = I_n + \frac{h}{2}(\frac{U_n - (R_k + R_p(I_n))I_n}{L_k} + \frac{U_{n+1} - (R_k + R_p(I_{n+1}))I_{n+1}}{L_k}) \tag{3.3}$$

$$U_{n+1} = U_n + \frac{h}{2}(-\frac{I_n}{C_k} - \frac{I_{n+1}}{C_k}) = U_n - \frac{h}{2}(\frac{I_n + I_{n+1}}{C_k}) \tag{3.4}$$

Подставляя (**3.4**) в (**3.3**), имеем:

$$I_{n+1} = I_n + \frac{h}{2L_k}(2U_n - (R_k + R_p(I_n) + \frac{h}{2C_k})I_n - (R_k + R_p(I_{n+1}) + \frac{h}{2C_k})I_{n+1}) \tag{3.5}$$

3. Из каких соображений проводится выбор численного метода того или иного порядка точности, учитывая, что чем выше порядок точности метода, тем он более сложени требует, как правило, больших ресурсов вычислительной системы?

Выбор численного метода проводится исходя из требований поставленной задачи, объема вычислений, а также от свойств функций, используемых в вычислениях.

Оценивается погрешность для частного случая вида правой части дифференциального уравнения: $\varphi(x, \nu) = \varphi(x)$

Так, если $\varphi(x, \nu)$ непрерывна и ограничена и ограничены и непрерывны её четвертые производные, то наилучший результат достигаем при использовании метода Рунге-Кутта 4 порядка:

$y_{n+1} = y_n + \frac{k_1+2k_2+2k_3+k_4}{6}$, где

$k_1 = h_n f(x_n, y_n)$
$k_2 = h_n f(x_n + \frac{h_n)}{2}, y_n + \frac{k_1}{2}$
$k_3 = h_n f(x_n + \frac{h_n)}{2}, y_n + \frac{k_2}{2}$
$k_4 = h_n f(x_n + \frac{h_n)}{2}, y_n + k_3$

Однако в случае если $\varphi(x, \nu)$ не имеет таких производных, то четвертый порядок схемы не может быть достигнут и стоит применять более простые схемы.