МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Университет ИТМО

ОТЧЕТ

По дисциплине:

«Программирование систем управления»

Вариант №6

ВЫПОЛНИЛ:

студент группы E4260,

Карпаев Е.В.

ПРОВЕРИЛ:

Томашевич С. И.

Санкт-Петербург

2023 г.

ОГЛАВЛЕНИЕ

ЗАДАНИЕ

1. Провести моделирование передаточной функции в среде MATLAB с использованием Simulink

$$\frac{y(t)}{u(t)} = \frac{S^3 + S^2 + S}{S^3 + 2S^2 + 2S + 1}$$

2. Провести моделирование входного сигнала в среде MATLAB с использованием Simulink

$$u(t) = 2 * sin\left(2t + \frac{\pi}{3}\right) - 2$$

3. Полученные сигналы необходимо привести к дискретному виду и промоделировать на частотах: 5 Гц, 30 Гц, 100 Гц.

4. Реализовать передаточную функцию на языке C++ для непрерывного случая и трех дискретных.

5. Реализовать входной сигнал на языке C++ для непрерывного случая и трех дискретных с разными частотами.

6. Провести сравнения результатов, полученных в среде Simulink и на языке C++.

МОДЕЛИРОВАНИЕ ПЕРЕДАТЧНОЙ ФУНКЦИИ В СРЕДЕ SIMULINK

Из передаточной функции $\frac{y(t)}{u(t)} = \frac{S^3 + S^2 + S}{S^3 + 2S^2 + 2S + 1}$, используя функционал MATLAB, получим матрицы A, B, C и D, которые будут описывать состояния нашей системы.

$$A = \begin{bmatrix} -2 & -2 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} -0.5 & -0.5 & -0.5 \end{bmatrix}$$

$$D = [1]$$

Указав данные матрицы в блоке «State-Space» в среде Simulink, был получен следующий график (Рисунок 1). В качестве сигнала подается константное значение «10».

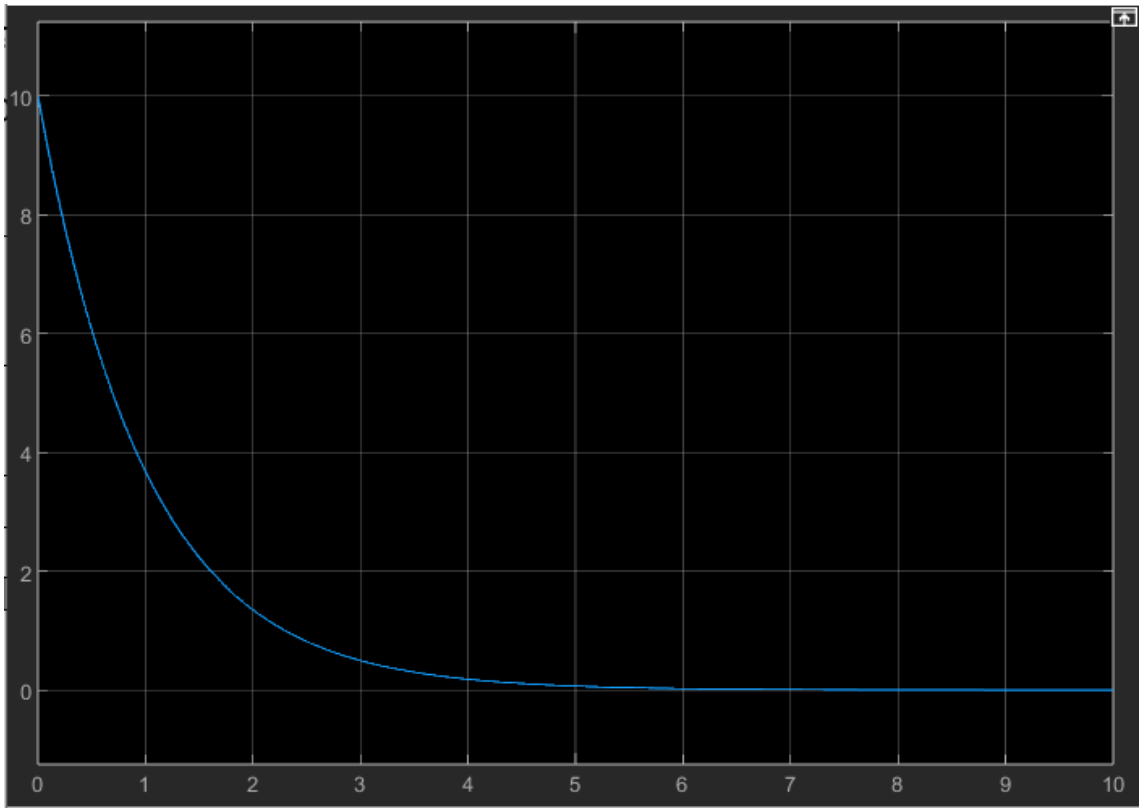

Рисунок 1. График блока State-space

Построим данную систему, используя блоки интегратора, усиления и суммирования (Рисунок 2).

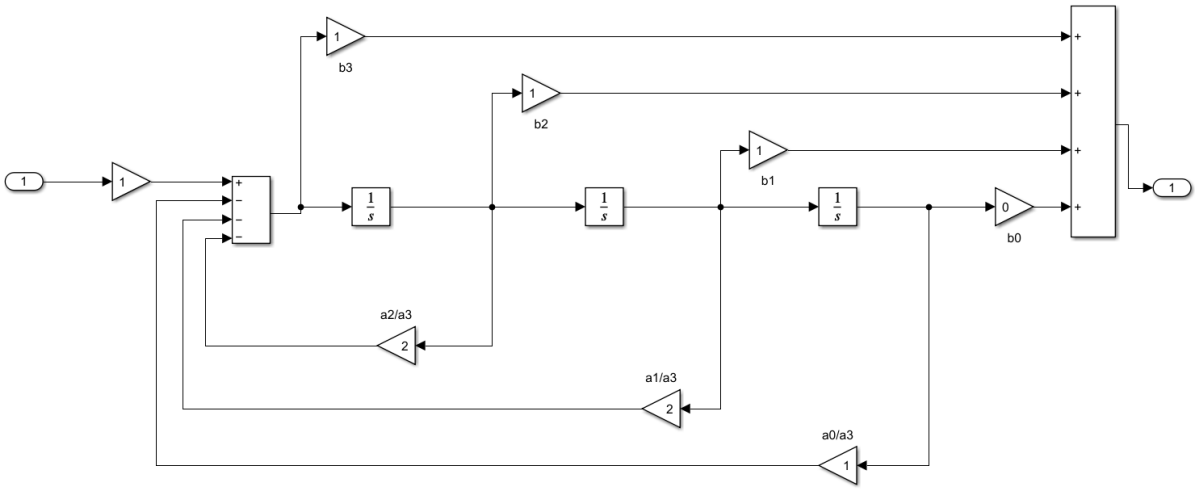

Рисунок 2. Структурная схема непрерывной передаточной функции

Приведем систему к дискретному виду. Для этого, используя функции MATLAB получим матрицы следующего вида.

- Для частоты 5 Гц

$$Ad1 = \begin{bmatrix} 0.6387 & -0.3427 & -0.1626 \\ 0.1626 & 0.9639 & -0.0175 \\ 0.0175 & 0.1975 & 0.9988 \end{bmatrix}$$

$$Bb1 = \begin{bmatrix} 0.3252 \\ 0.0349 \\ 0.0024 \end{bmatrix}$$

- Для частоты 30 Гц

$$Ad2 = \begin{bmatrix} 0.9344 & -0.0650 & -0.0322 \\ 0.0322 & 0.9989 & -0.0005 \\ 0.0005 & 0.0333 & 1.0000 \end{bmatrix}$$

$$Bb2 = \begin{bmatrix} 0.0645 \\ 0.0011 \\ 0.0000 \end{bmatrix}$$

- Для частоты 100 Гц

$$Ad3 = \begin{bmatrix} 0.9801 & -0.0199 & -0.0099 \\ 0.0099 & 0.9999 & -0.0000 \\ 0.0000 & 0.0100 & 1.0000 \end{bmatrix}$$

$$Bb3 = \begin{bmatrix} 0.0198 \\ 0.0001 \\ 0.0000 \end{bmatrix}$$

Построим структурную схему системы, в которой заменим интеграторы на элемент памяти (Рисунок 3).

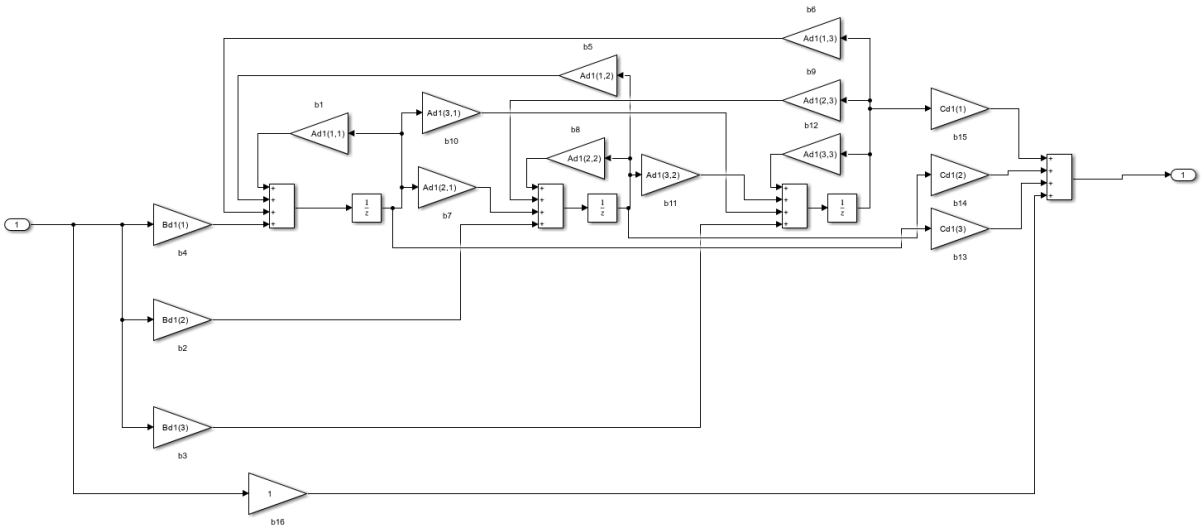

Рисунок 3. Структурная схема дискретной передаточной функции

Выведем графики дискретной системы с разными частотами. Время моделирование 1.5 секунды (Рисунок 4).

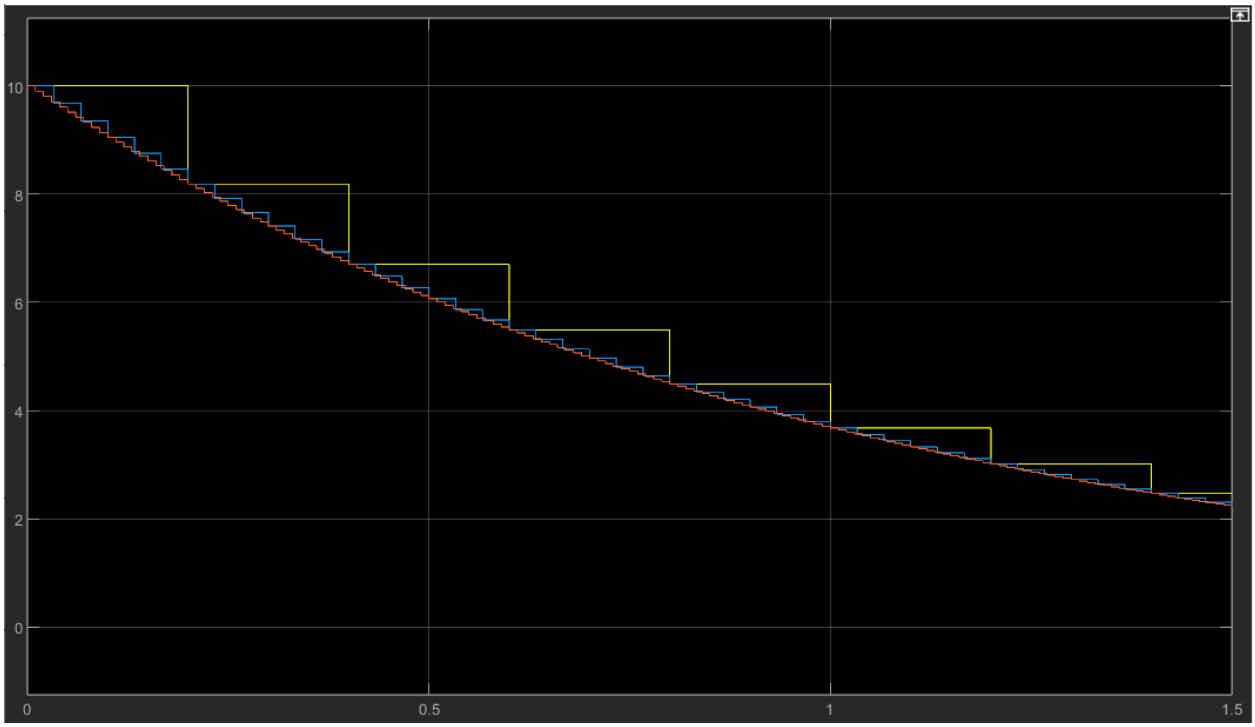

Рисунок 4. Графики дискретной передаточной функции (1.5с)

Аналогично промоделируем 10 секунд и выведем все графики (Рисунок 5).

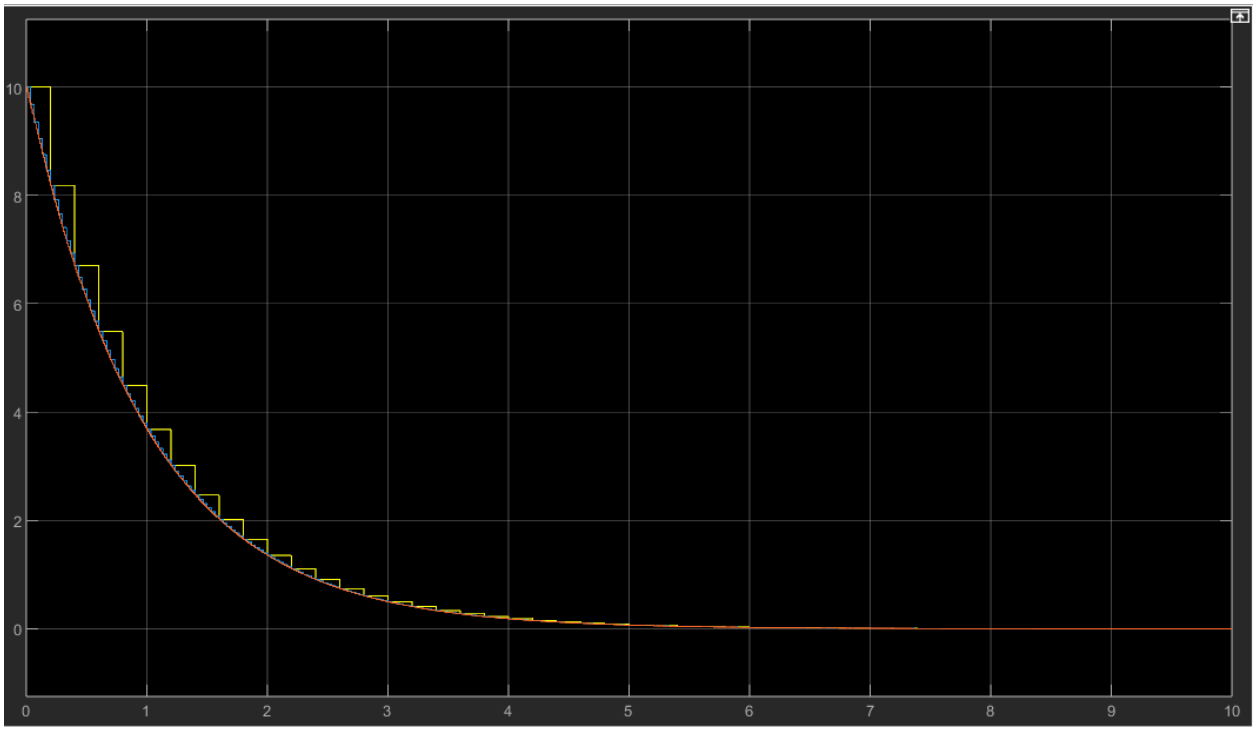

Рисунок 5. Графики дискретной передаточной функции (10с)

МОДЕЛИРОВАНИЕ ВХОДЯЩЕГО СИГНАЛА В MATLAB

Построим входящий синусоидальный сигнал, имеющий вид:

$$u(t) = 2\sin\left(2t + \frac{\pi}{3}\right) - 2$$

При помощи функции diff в MATLAB вычислим первую и вторую производные данной функции:

$$\dot{u}(t) = 4\cos\left(2t + \frac{\pi}{3}\right)$$

$$\ddot{u}(t) = -8sin\left(2t + \frac{\pi}{3}\right)$$

Таким образом, получим коэффициент усиления «-4». Матрицы A, B, C и D выглядят следующим образом:

$$A = \begin{bmatrix} 0 & 1 \\ -4 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

D = [0]

Построим структурную схему (Рисунок 6), использую блоки интегрирования и усиления. Блок суммирования для смещения функции на «-2» и блок «Sine Wave», в котором зададим данный сигнал.

Начальные значения блоков «dot(x2)->x2» - ***2***, «dot(x1)->x1» - ***2*sin(pi/3)***

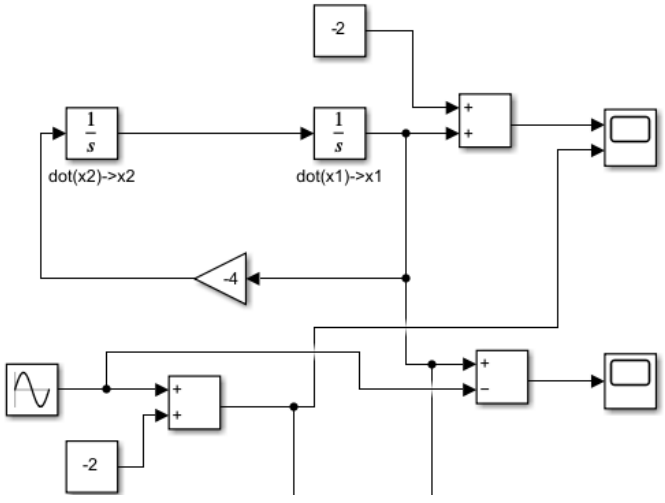

Рисунок 6. Структурная схема непрерывного синуса

При моделировании будет получен следующий сигнал (Рисунок 7).

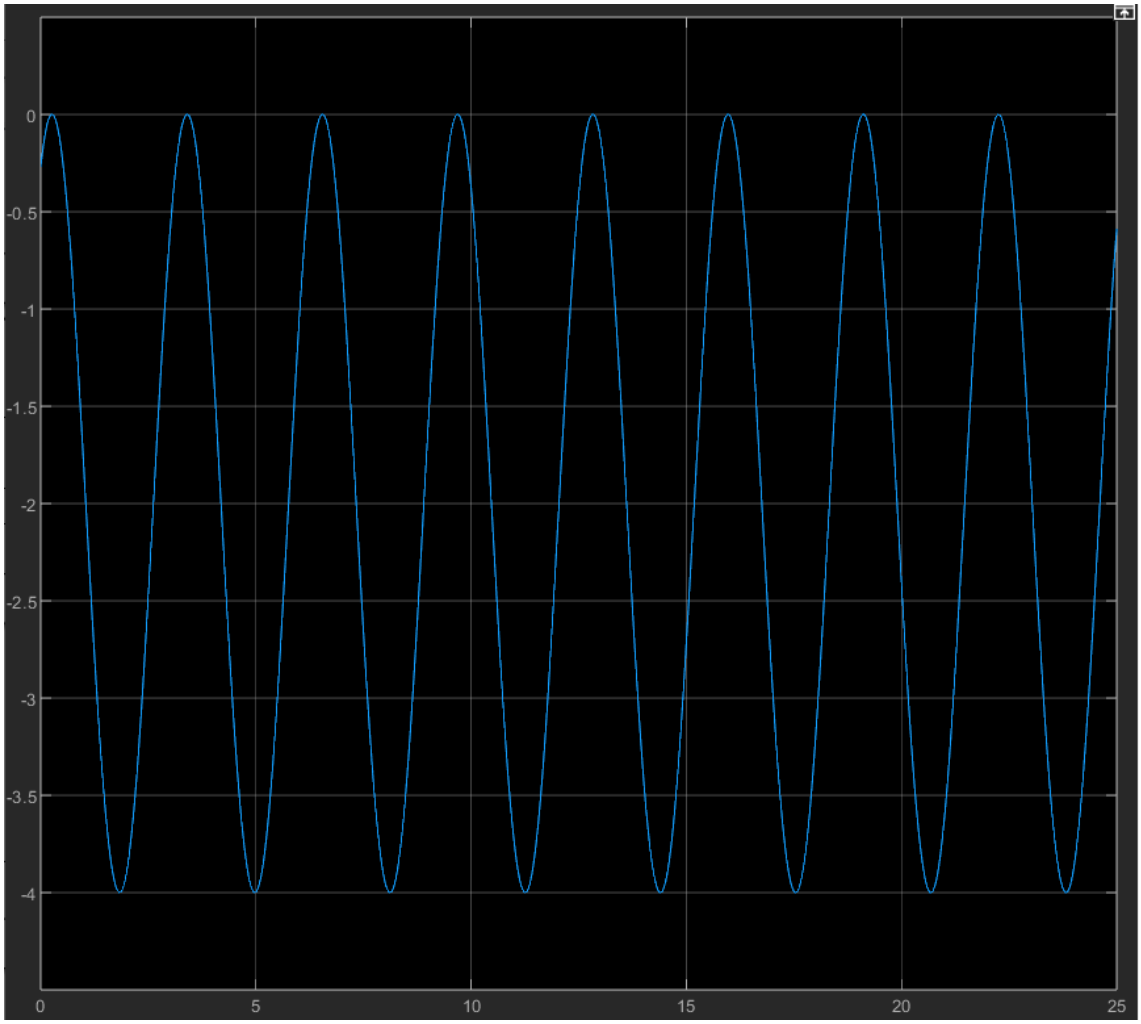

Рисунок 7. График непрерывного синуса

Приведем сигнал к дискретному виду. Для этого используем функцию «c2d» с указанием частоты дискретизации. Получим следующие матрицы:

- Для частоты 5 Гц

$$A = \begin{bmatrix} 0.9211 & 0.1947 \\ -0.7788 & 0.9211 \end{bmatrix}$$

- Для частоты 30 Гц

$$A = \begin{bmatrix} 0.9978 & 0.03331 \\ -0.1332 & 0.9978 \end{bmatrix}$$

- Для частоты 100 Гц

$$A = \begin{bmatrix} 0.9998 & 0.009999 \\ -0.04 & 0.9998 \end{bmatrix}$$

Построим 3 дискретных блока «State-Space» (Рисунок 8), в которых укажем матрицы для 5 Гц, 30 Гц и 100 Гц. Выведем полученные графики вместе, промоделировав 2 секунды (Рисунок 9).

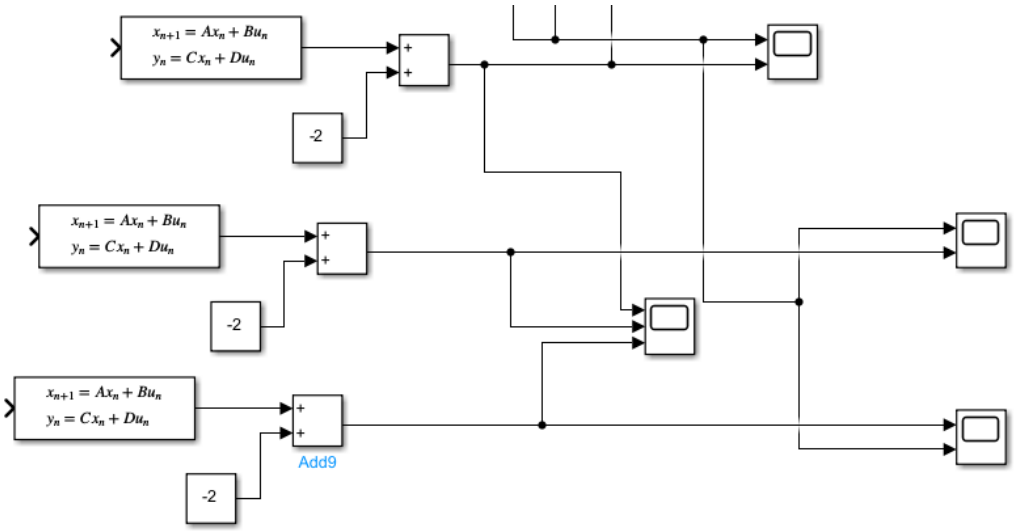

Рисунок 8. Схема подключения дискретных блоков State-Space

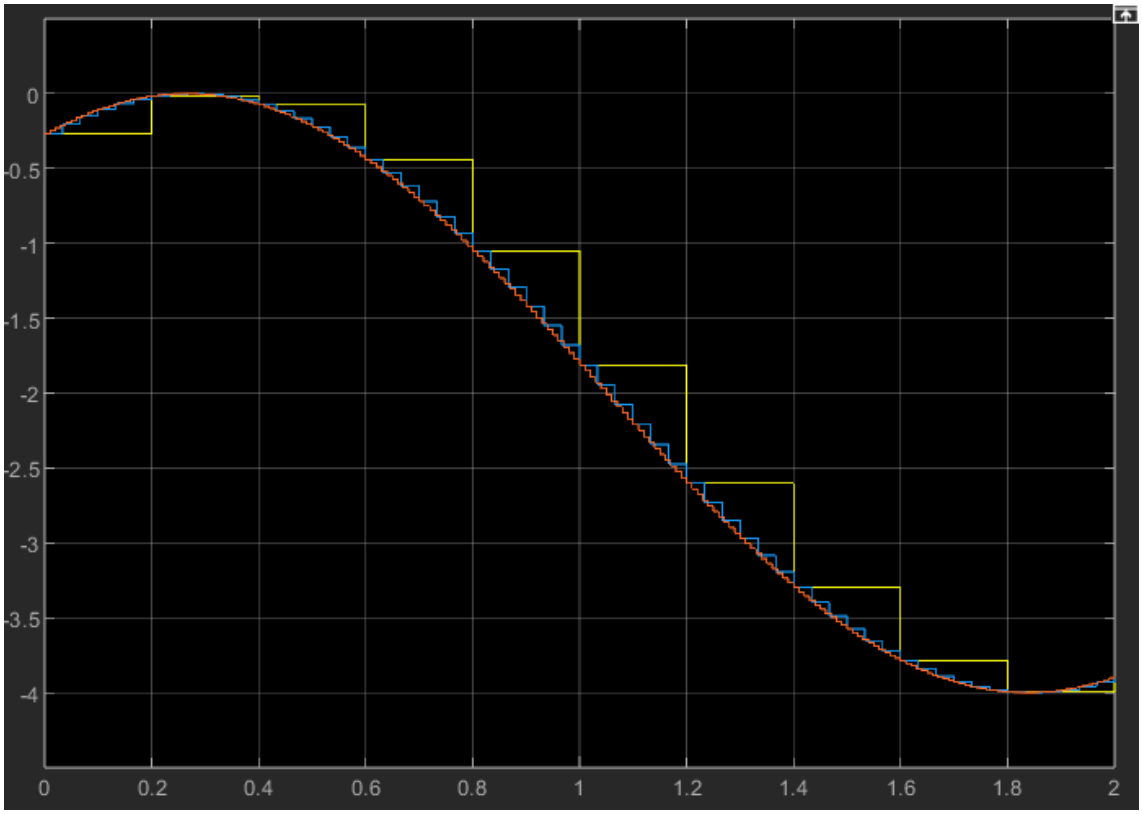

Рисунок 9. Графики дискретных синусов (2с)

Проведем моделирование для 25 секунд (Рисунок 10).

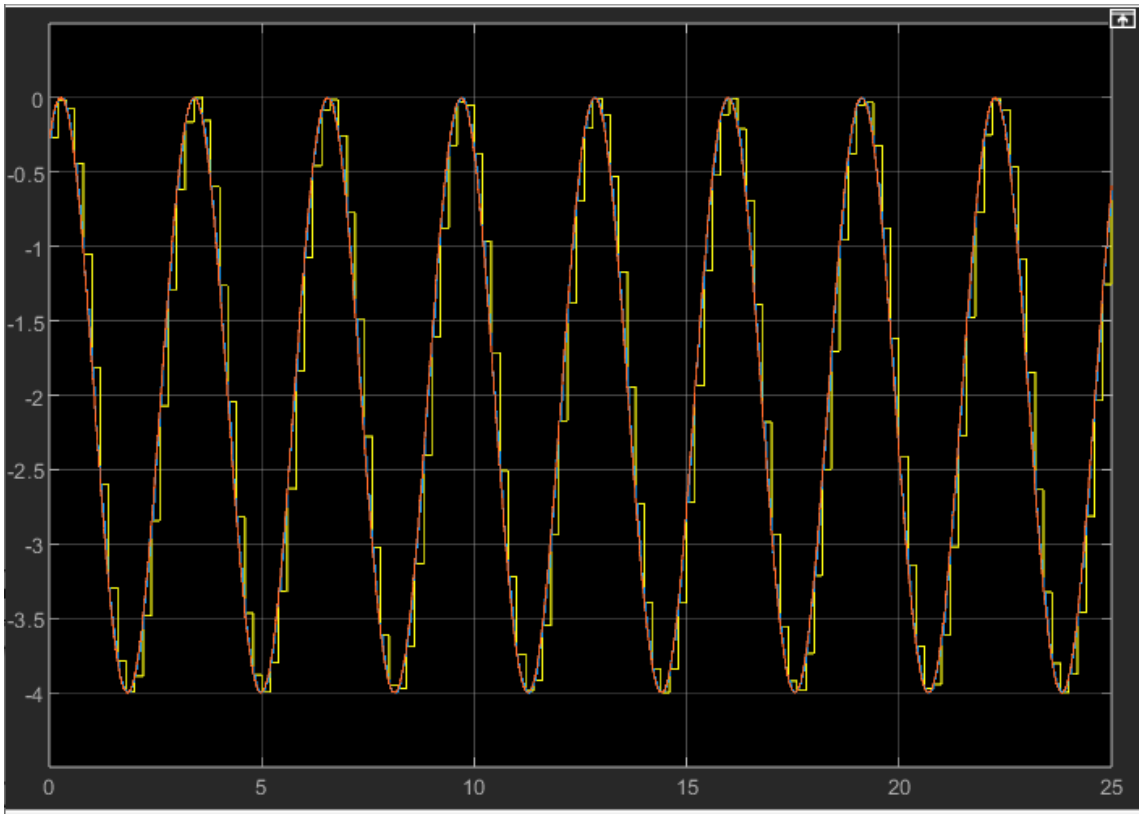

Рисунок 10. Графики дискретных синусов (25с)

РЕАЛИЗАЦИЯ ПЕРЕДАТОЧНОЙ ФУНКЦИИ НА ЯЗЫКЕ C++

Используя матрицы состояний, полученные в ходе построения системы в MATLAB, был реализован класс *StateSpace* для непрерывной системы (см. Приложение А) и класс *Discrete* для дискретной системы (см. Приложение Б). Проведем моделирование непрерывной системы на частоте 5 Гц (Рисунок 11).

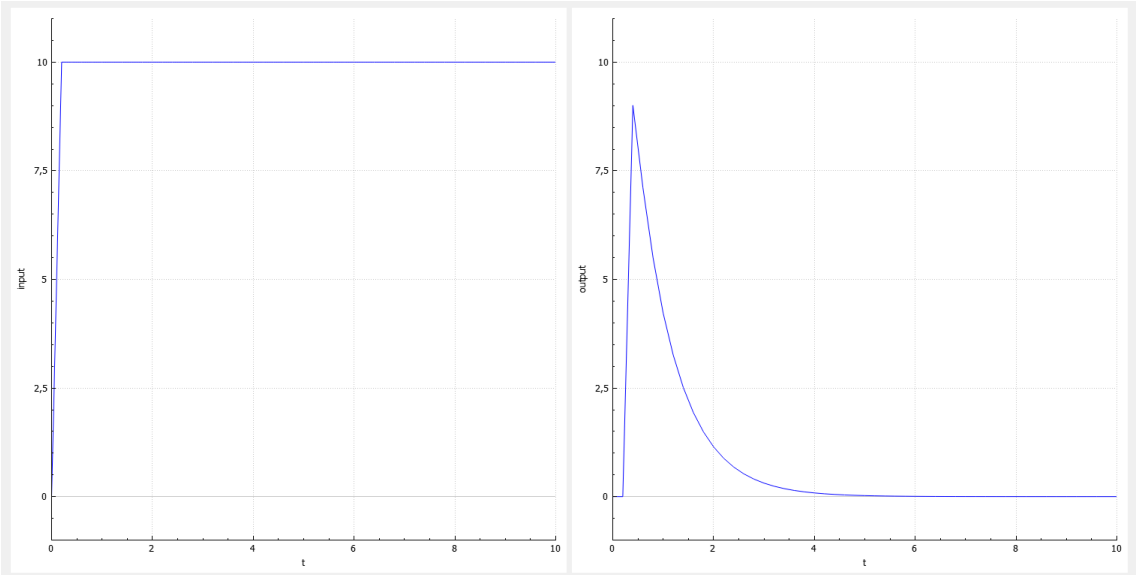

Рисунок 11. Непрерывная система с частотой 5 Гц

Промоделируем непрерывную систему на частоте 30 Гц (Рисунок 12).

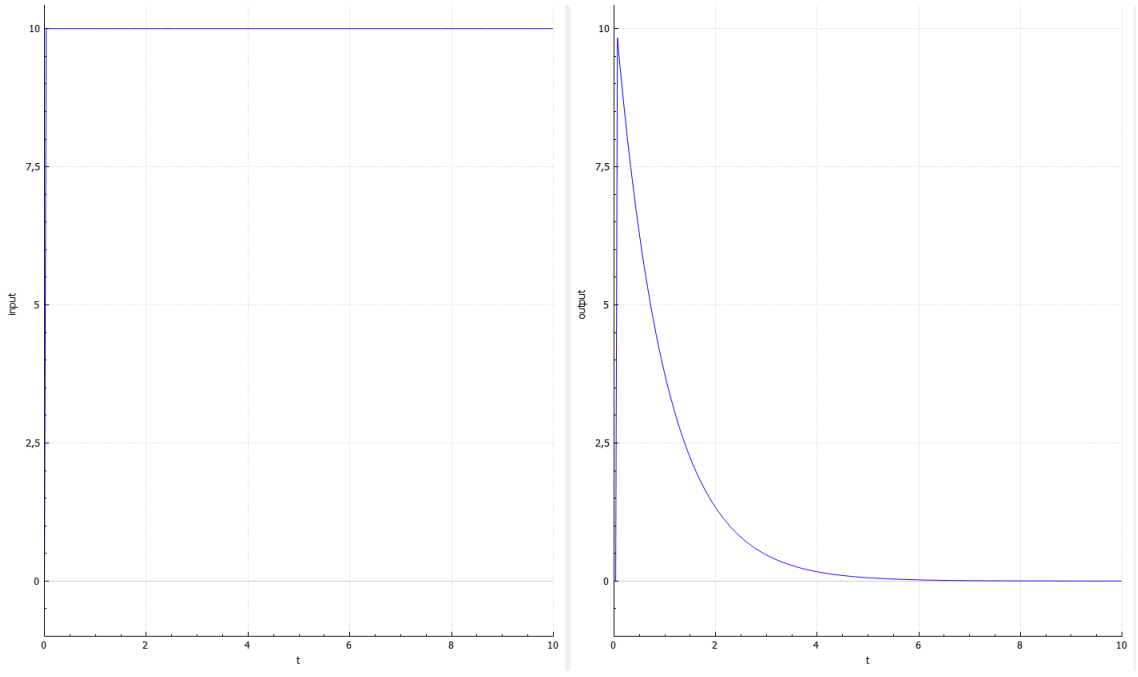

Рисунок 12. Непрерывная система с частотой 30 Гц

Промоделируем непрерывную систему на частоте 100 Гц (Рисунок 13).

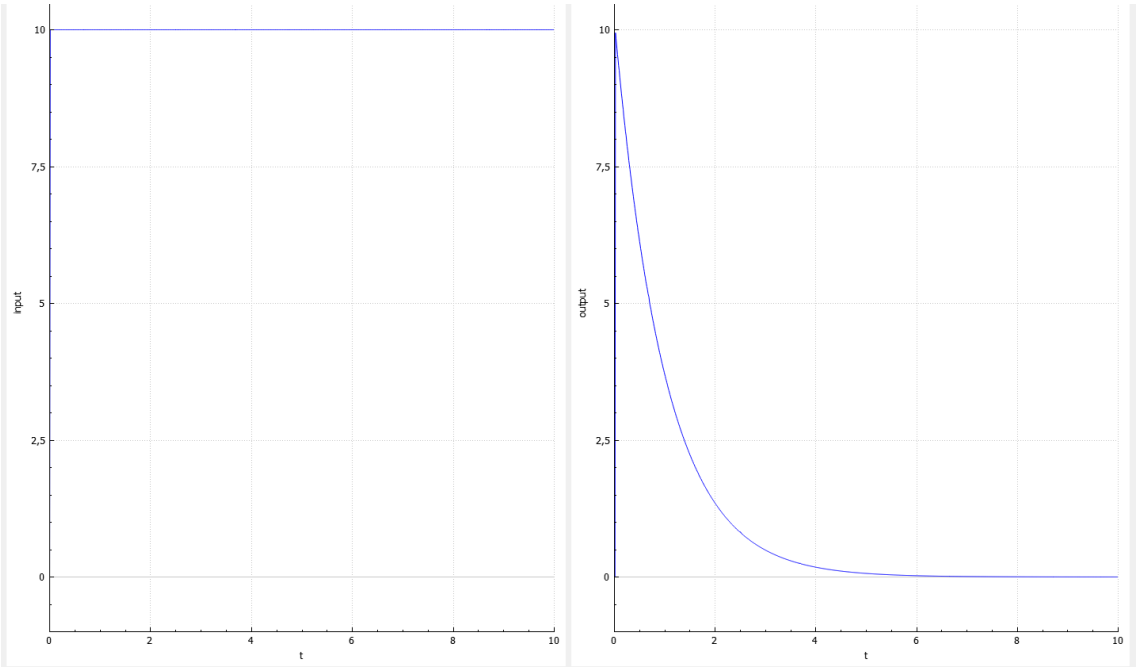

Рисунок 13. Непрерывная система с частотой 100 Гц

Аналогично промоделируем теперь дискретную систему с использованием тех же частот. Начнем с 5 Гц (Рисунок 14).

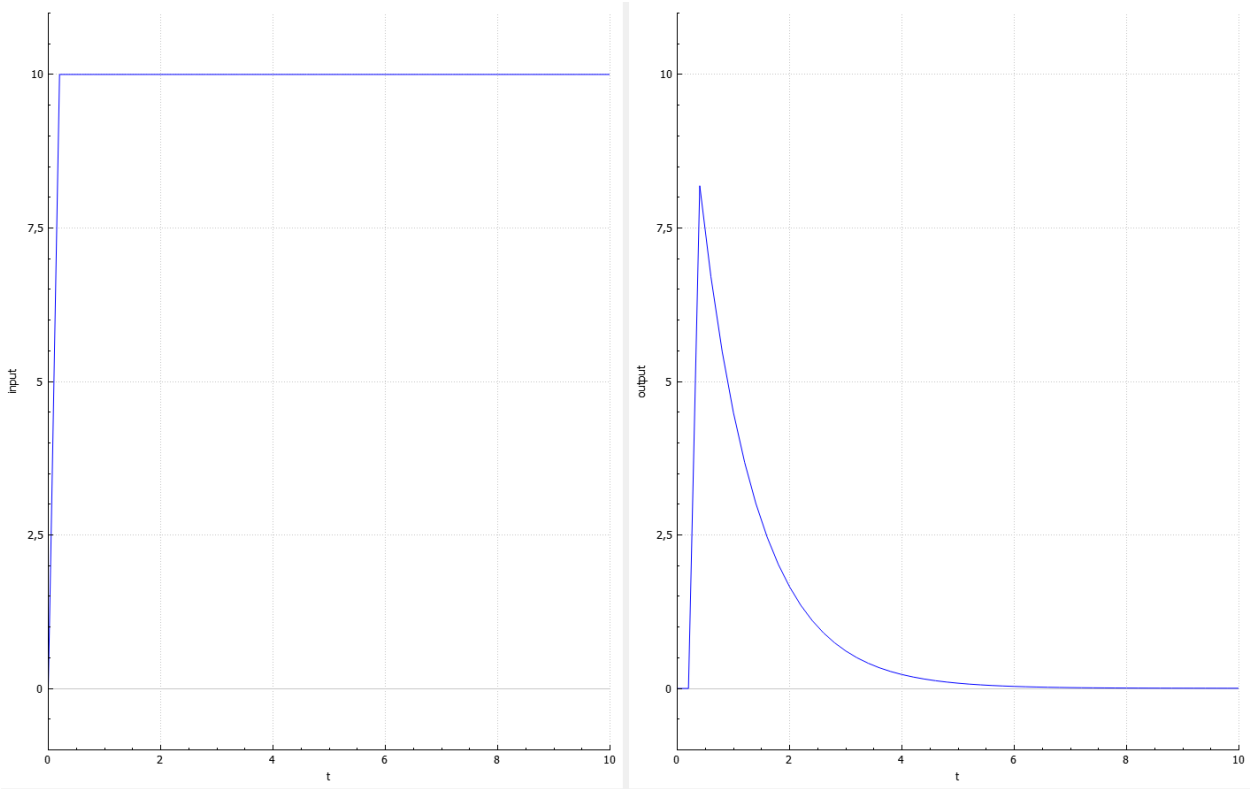

Рисунок 14. Дискретная система с частотой 5 Гц

Промоделируем дискретную систему на частоте 30 Гц (Рисунок 15).

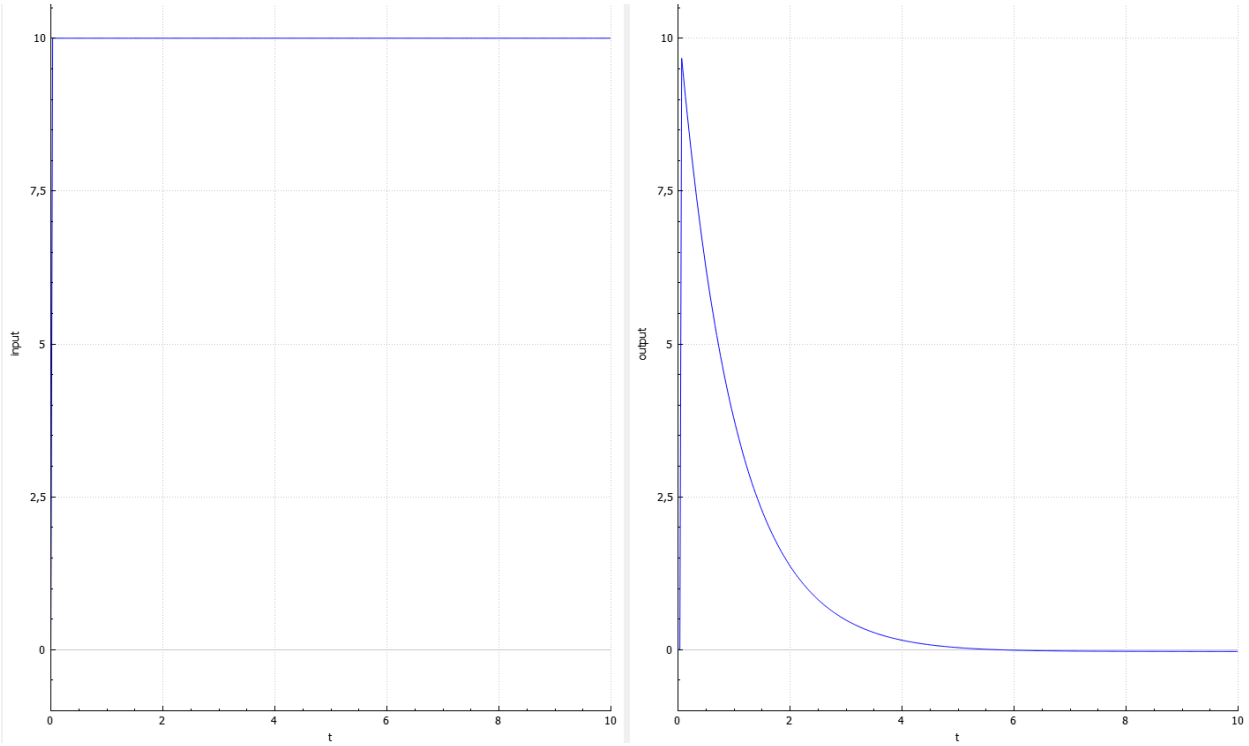

Рисунок 15. Дискретная система с частотой 30 Гц

Промоделируем дискретную систему на частоте 100 Гц (Рисунок 16).

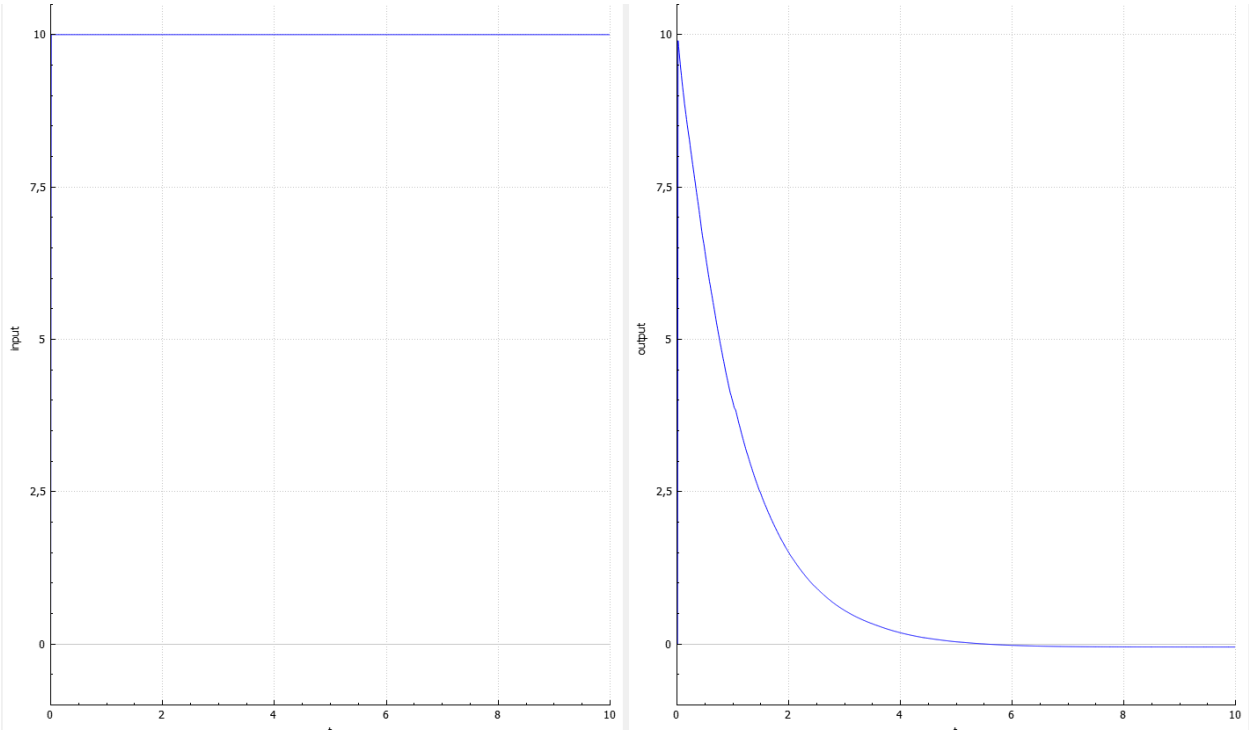

Рисунок 16. Дискретная система с частотой 100 Гц

Как видно из графиков, непрерывная система совпадает и дискретной, если при моделировании используется одинаковая для систем частота. Так же заметим, что чем выше частота, тем раньше начинает происходить моделирование выходного сигнала.

РЕАЛИЗАЦИЯ ВХОДНОГО СИГНАЛА НА ЯЗЫКЕ C++

Промоделируем непрерывный синус, код которого реализован в классе *SinAnalog* (см. Приложение В), на частотах 5 Гц, 30 Гц и 100 Гц (Рисунок 17-19).

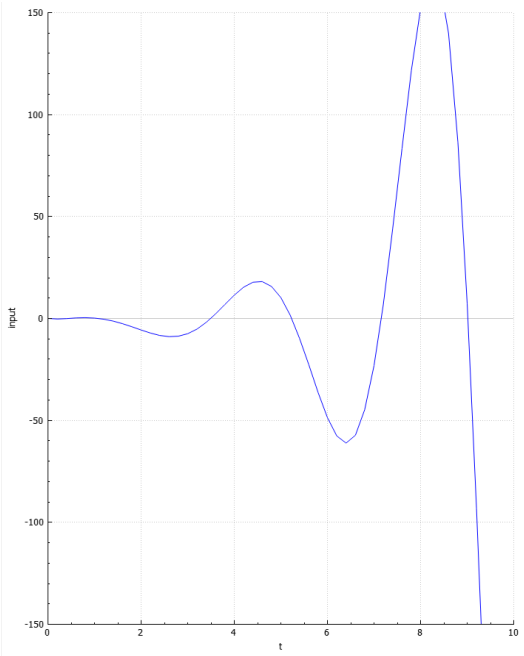

Рисунок 17. Непрерывный входной сигнал с частотой 5 Гц

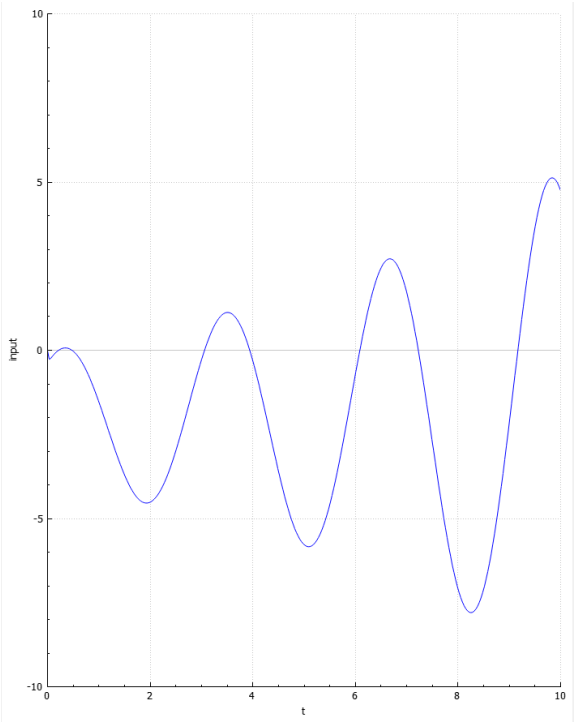

Рисунок 18. Непрерывный входной сигнал с частотой 30 Гц

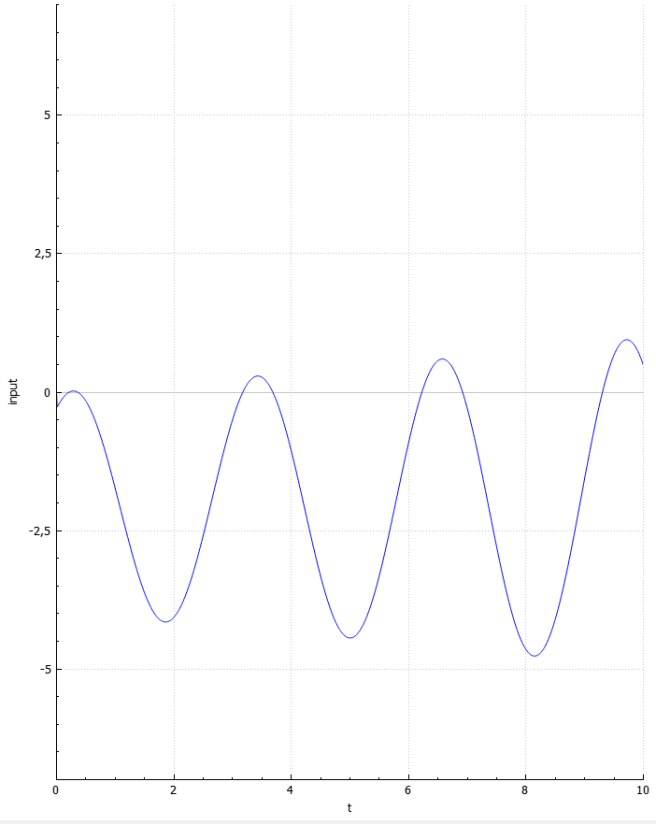

Рисунок 19. Непрерывный входной сигнал с частотой 100 Гц

Теперь проведем моделирование дискретного входного сигнала, код которого реализован в классе *SinDiscrete* (см. Приложение Г), с частотами 5 Гц, 30 Гц и 100 Гц (Рисунок 20-22).

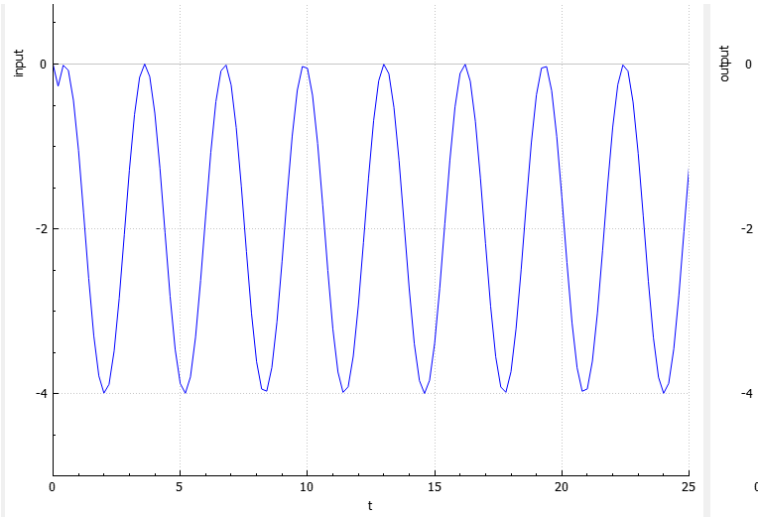

Рисунок 20. Дискретный входной сигнал с частотой 5 Гц

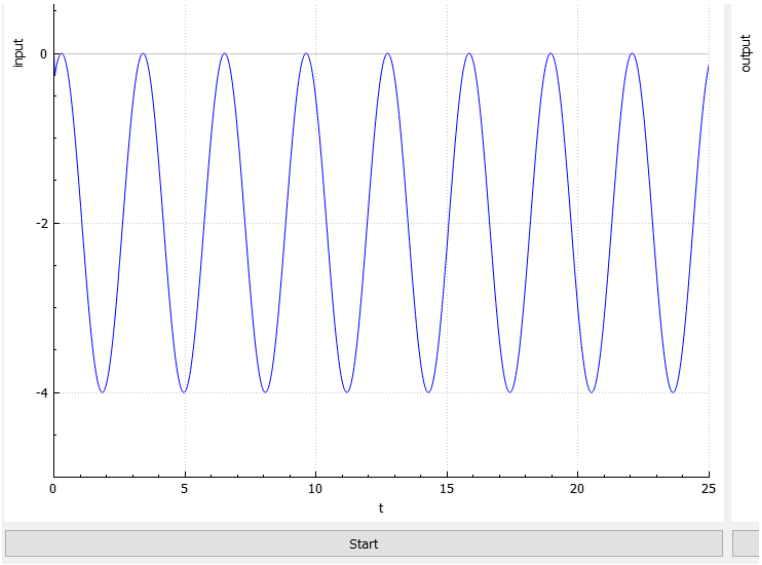

Рисунок 21. Дискретный входной сигнал с частотой 30 Гц

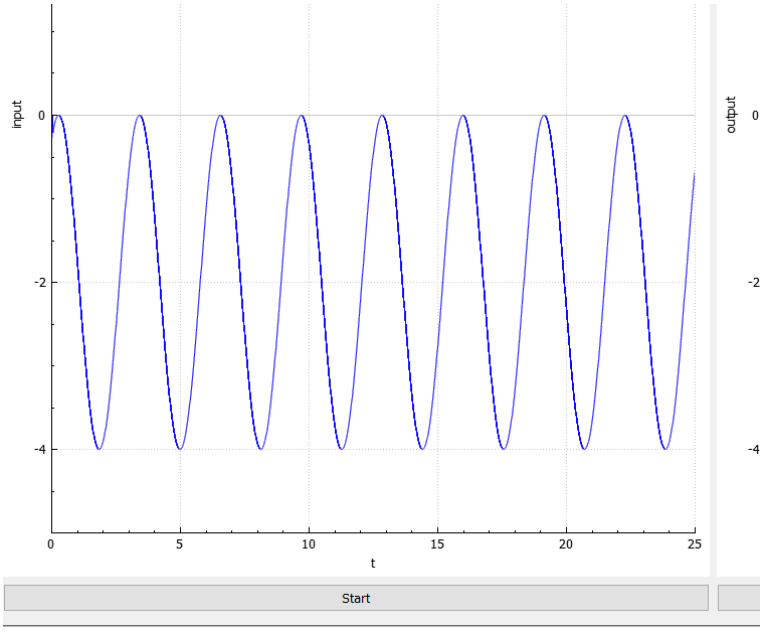

Рисунок 21. Дискретный входной сигнал с частотой 100 Гц

СРАВНЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Чтобы убедиться, что все графики были получены верно, сравним попарно их реализацию в MATLAB и на языке C++ (Рисунок 23 – 30).

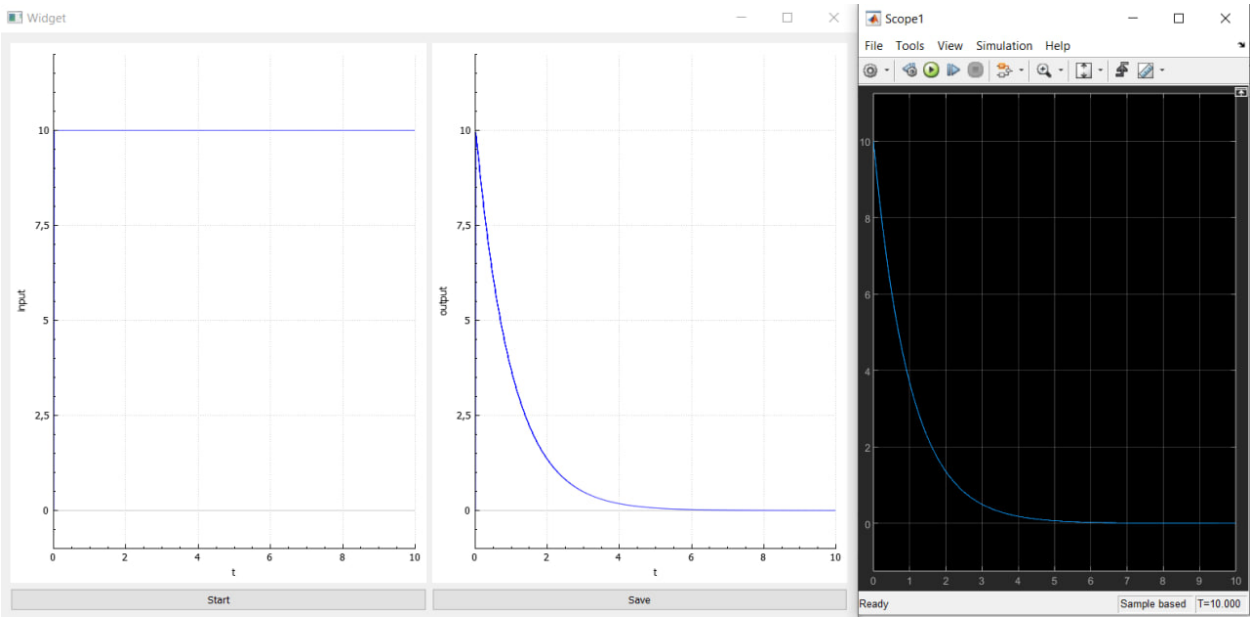

Рисунок 23. Сравнение непрерывной системы

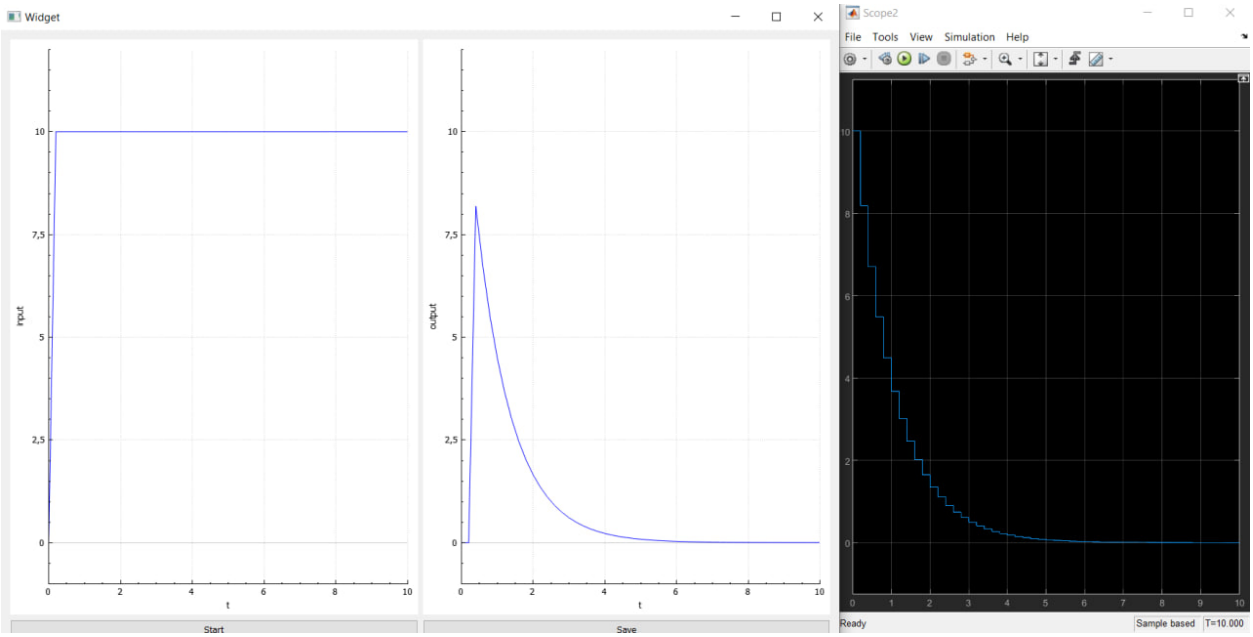

Рисунок 24. Сравнение дискретной системы (5 Гц)

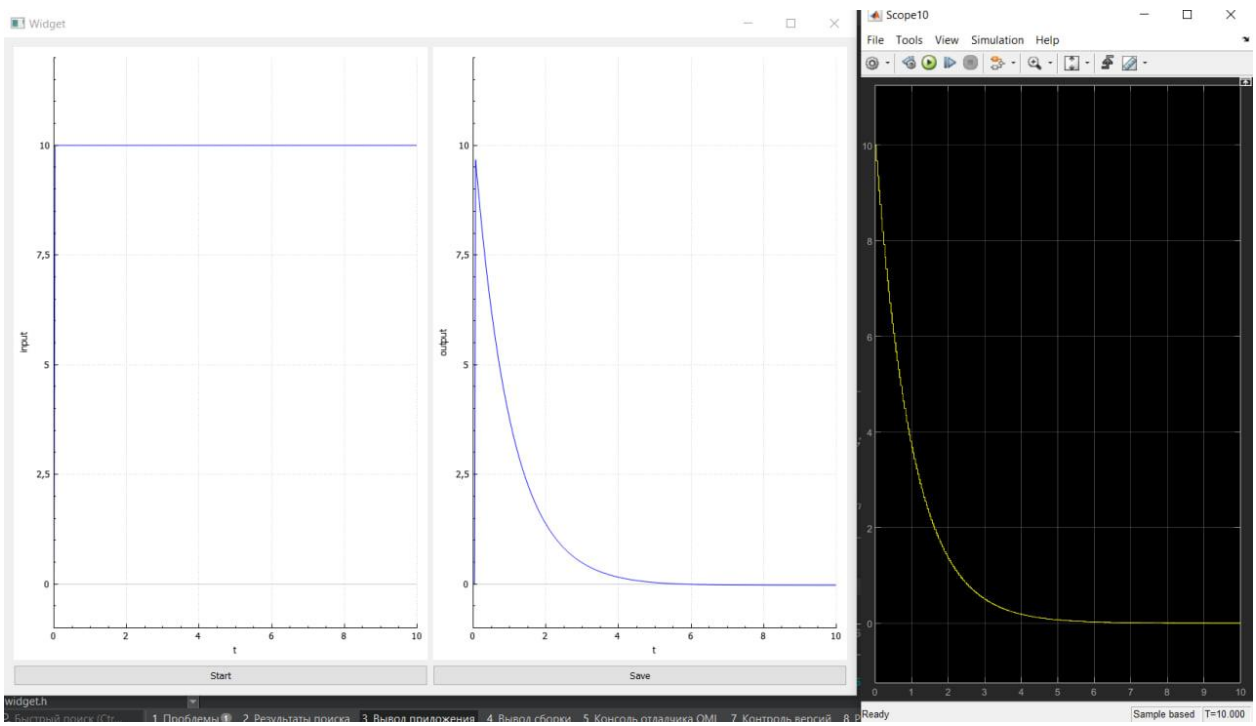

Рисунок 25. Сравнение дискретной системы (30 Гц)

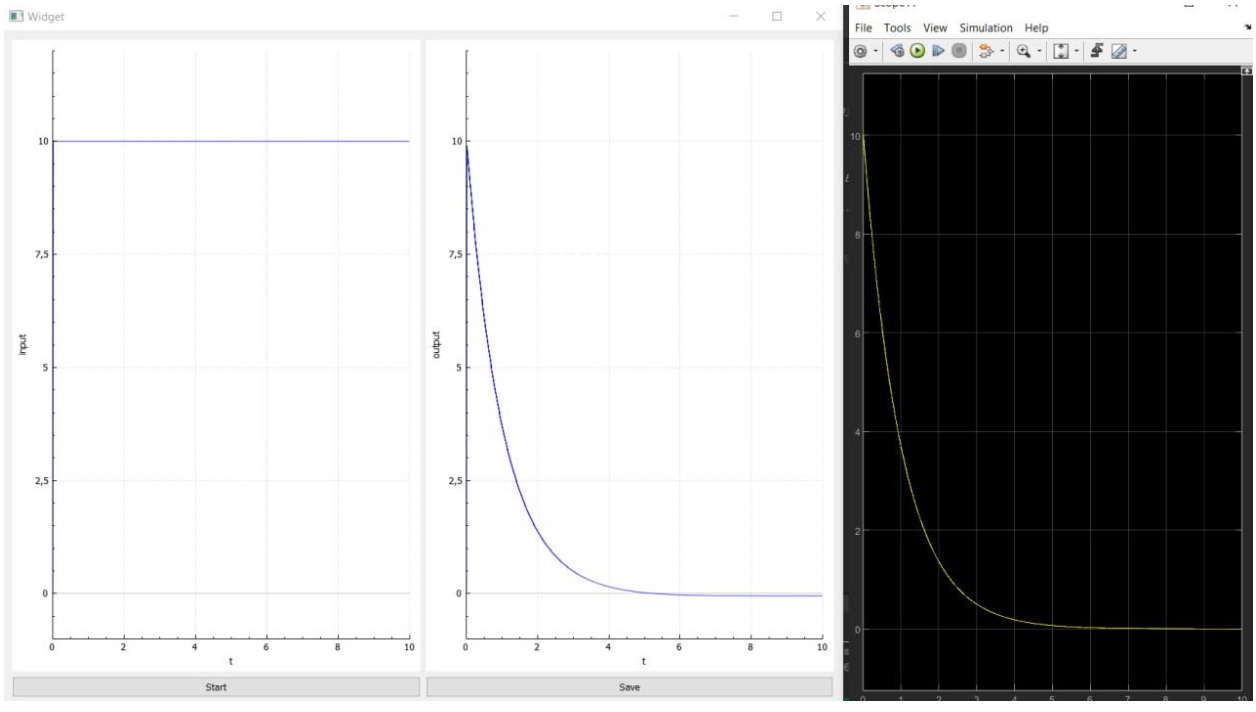

Рисунок 26. Сравнение дискретной системы (100 Гц)

При сравнении непрерывного синуса, реализованного на языке C++, *SAMPLINGTIMEMSEC* равен 1, что равно 1кГц.

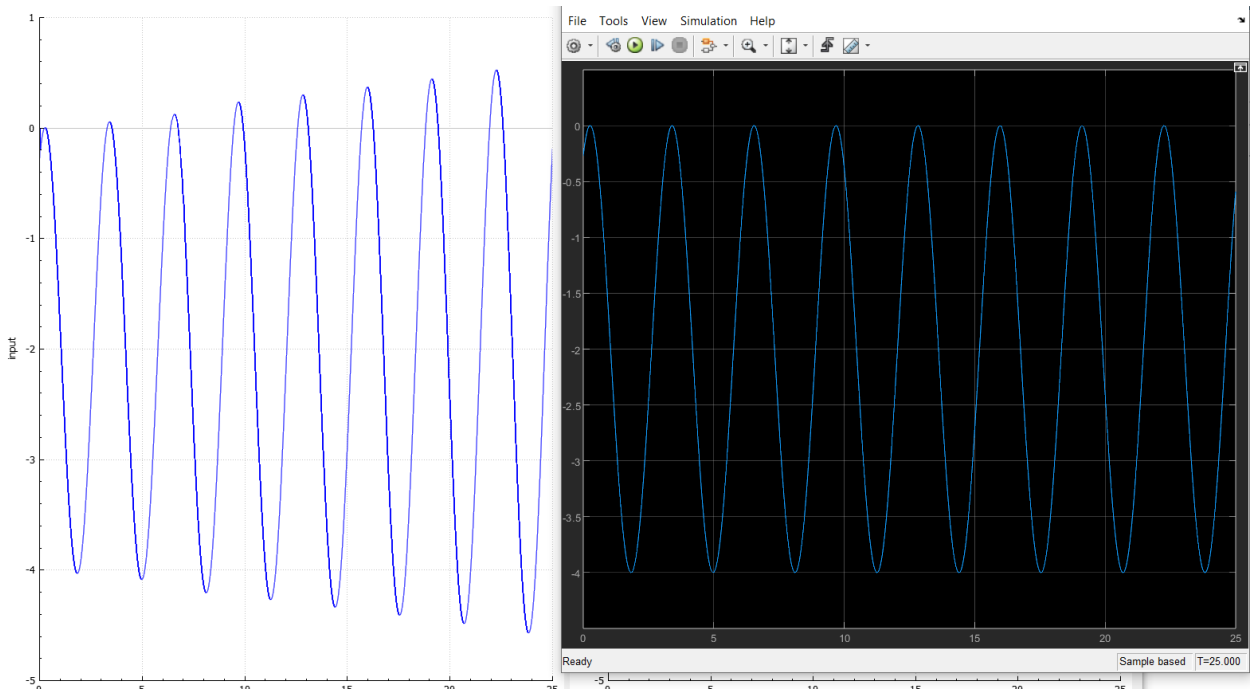

Рисунок 27. Сравнение непрерывного синуса

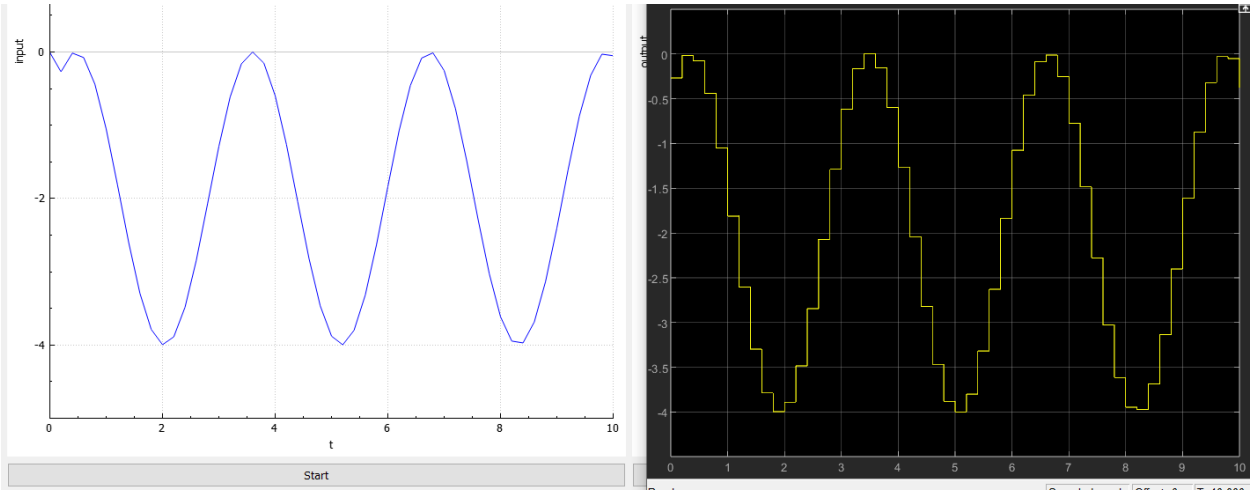

Рисунок 28. Сравнение дискретного синуса (5 Гц)

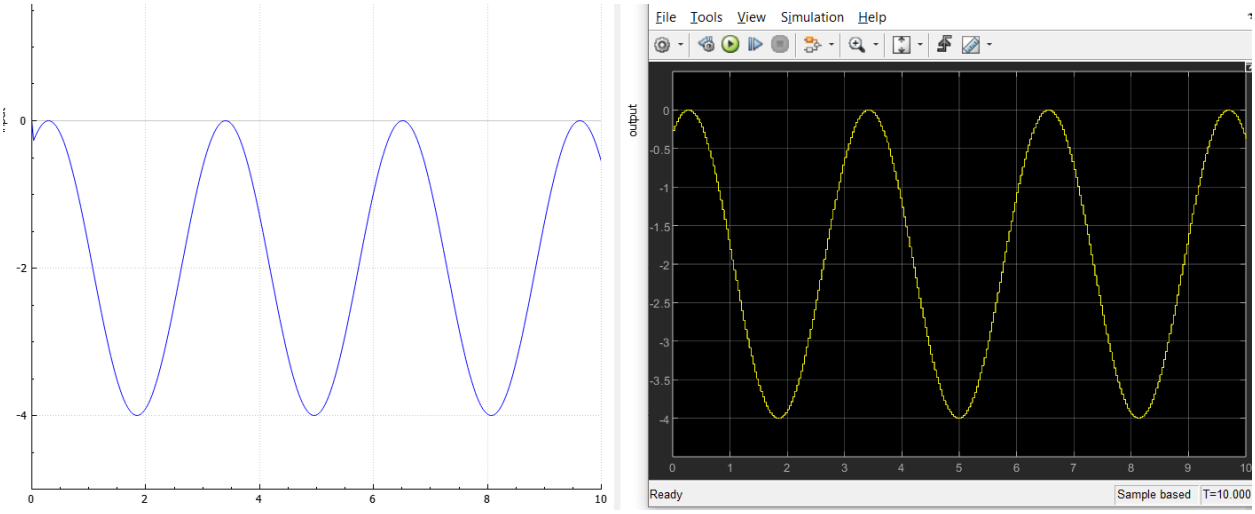

Рисунок 29. Сравнение дискретного синуса (30 Гц)

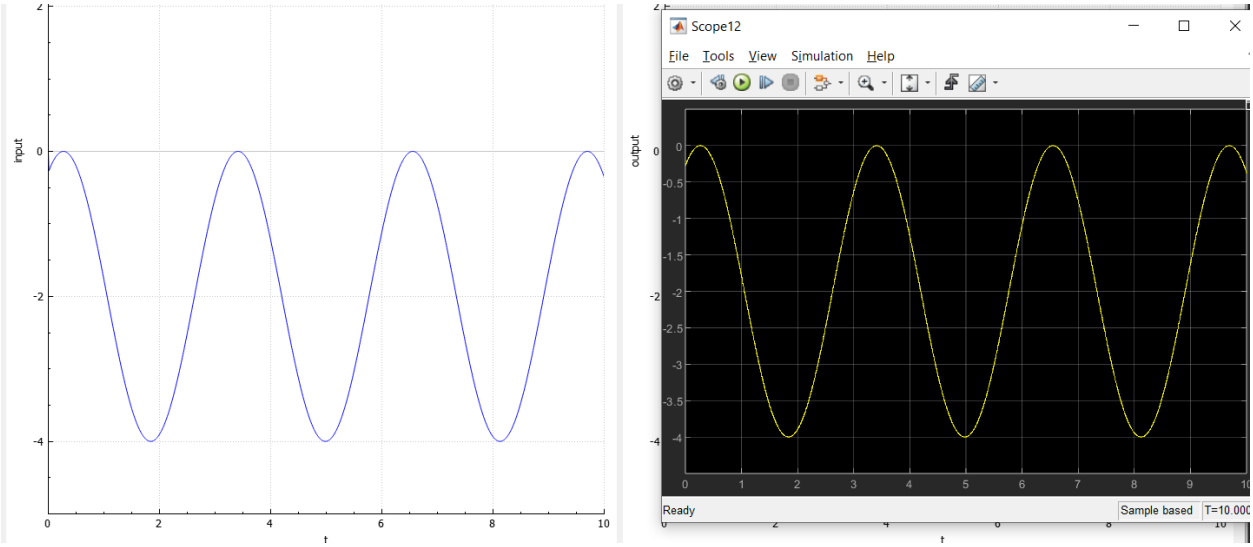

Рисунок 30. Сравнение дискретного синуса (100 Гц)

ВЫВОДЫ

Сравнение графиков непрерывного сигнала, реализованного двумя разными способами, показывает, что величина амплитуды сильно зависит от частоты: чем меньше частота, тем выше амплитуда. Таким образом, чтобы сигнал, реализованный на C++, был похож на тот, что строит MATLAB, необходимо при его построении использовать очень высокую частоту (порядка 1кГц).

ПРИЛОЖЕНИЕ А

statespace.h

```
#ifndef STATESPACE_H
#define STATESPACE_H

#include "blocks/integrator/integrator.h"
#include <vector>
#include <cstdint>

class StateSpace
{
public:
    StateSpace(std::vector<float> &m_initial_conditions,
               std::vector<std::vector<float>> &matrix_A,
               std::vector<float> &matrix_B,
               std::vector<float> &matrix_C,
               std::vector<float> &matrix_D);

    ~StateSpace();

    float getOutput();
    float update(float input, float dt);

private:
    Integrator *m_integrator_X1;
    Integrator *m_integrator_X2;
    Integrator *m_integrator_X3;

    float m_previous_input = 0.0;

    std::vector<std::vector<float>> m_matrix_A;
    std::vector<float> m_matrix_B;
    std::vector<float> m_matrix_C;
    std::vector<float> m_matrix_D;

    std::vector<float> m_initial_conditions;
};

#endif // STATESPACE_H
```

```
#include "statespace.h"

StateSpace::StateSpace(std::vector<float> &m_initial_conditions,
                       std::vector<std::vector<float>> &matrix_A,
                       std::vector<float> &matrix_B,
                       std::vector<float> &matrix_C,
                       std::vector<float> &matrix_D)

{
    m_integrator_X1 = new Integrator(m_initial_conditions[0]);
    m_integrator_X2 = new Integrator(m_initial_conditions[1]);
    m_integrator_X3 = new Integrator(m_initial_conditions[2]);

    m_matrix_A.resize(3);

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        m_matrix_A[i].resize(3);
    }

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        for(uint64_t j = 0; j != matrix_A[i].size(); ++j){
            m_matrix_A[i][j] = matrix_A[i][j];
        }
    }

    for(uint64_t i = 0; i != matrix_B.size(); ++i){
        m_matrix_B.push_back(matrix_B[i]);
    }

    for(uint64_t i = 0; i != matrix_C.size(); ++i){
        m_matrix_C.push_back(matrix_C[i]);
    }

    for(uint64_t i = 0; i != matrix_D.size(); ++i){
        m_matrix_D.push_back(matrix_D[i]);
    }

}

StateSpace::~StateSpace()
{
    delete m_integrator_X1;
    delete m_integrator_X2;
    delete m_integrator_X3;
}

float StateSpace::getOutput()
{
    return m_matrix_C[0] * m_integrator_X1->state() +
           m_matrix_C[1] * m_integrator_X2->state() +
           m_matrix_C[2] * m_integrator_X3->state() +
```

```
            m_matrix_D[0] * m_previous_input;
}

float StateSpace::update(float input, float dt)
{
    float tmp_x1 = m_integrator_X1->state();
    float tmp_x2 = m_integrator_X2->state();
    float tmp_x3 = m_integrator_X3->state();
    float output = getOutput();

    m_integrator_X1->update((m_matrix_A[0][0]    *    tmp_x1)    +
(m_matrix_A[0][1]  *  tmp_x2)  +  (m_matrix_A[0][2]  *  tmp_x3)  +
(m_matrix_B[0] * input), dt);
    m_integrator_X2->update((m_matrix_A[1][0]    *    tmp_x1)    +
(m_matrix_A[1][1]  *  tmp_x2)  +  (m_matrix_A[1][2]  *  tmp_x3)  +
(m_matrix_B[1] * input), dt);
    m_integrator_X3->update((m_matrix_A[2][0]    *    tmp_x1)    +
(m_matrix_A[2][1]  *  tmp_x2)  +  (m_matrix_A[1][2]  *  tmp_x3)  +
(m_matrix_B[2] * input), dt);

    m_previous_input = input;

    return output;
}
```

ПРИЛОЖЕНИЕ Б

discrete.h

```
#ifndef DISCRETE_H
#define DISCRETE_H

#include <vector>
#include <cstdint>

class Discrete
{
public:
    Discrete(std::vector<float> &initial_conditions,
             std::vector<std::vector<float>> &matrix_A,
             std::vector<float> &matrix_B,
             std::vector<float> &matrix_C,
             std::vector<float> &matrix_D);

    float getOutput();

    float update(float input);

private:
    std::vector<std::vector<float>> m_matrix_A;
    std::vector<float> m_matrix_B;
    std::vector<float> m_matrix_C;
    std::vector<float> m_matrix_D;

    std::vector<float> m_initial_conditions;

    float m_previous_input = 0.0;
};

#endif // DISCRETE_H
```

```
#include "discrete.h"

Discrete::Discrete(std::vector<float> &initial_conditions,
                   std::vector<std::vector<float>> &matrix_A,
                   std::vector<float> &matrix_B,
                   std::vector<float> &matrix_C,
                   std::vector<float> &matrix_D)
{
    m_matrix_A.resize(3);

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        m_matrix_A[i].resize(3);
    }

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        for(uint64_t j = 0; j != matrix_A[i].size(); ++j){
            m_matrix_A[i][j] = matrix_A[i][j];
        }
    }

    for(uint64_t i = 0; i != matrix_B.size(); ++i){
        m_matrix_B.push_back(matrix_B[i]);
    }

    for(uint64_t i = 0; i != matrix_C.size(); ++i){
        m_matrix_C.push_back(matrix_C[i]);
    }

    for(uint64_t i = 0; i != matrix_D.size(); ++i){
        m_matrix_D.push_back(matrix_D[i]);
    }

    for(uint64_t i = 0; i != initial_conditions.size(); ++i){
        m_initial_conditions.push_back(initial_conditions[i]);
    }
}

float Discrete::getOutput()
{
    return m_initial_conditions[0] * m_matrix_C[0] +
           m_initial_conditions[1] * m_matrix_C[1] +
           m_initial_conditions[2] * m_matrix_C[2] +
           m_previous_input * m_matrix_D[0];
}

float Discrete::update(float input)
{
    float output = getOutput();

    float tmp_x1 = m_initial_conditions[0];
    float tmp_x2 = m_initial_conditions[1];
```

```
    float tmp_x3 = m_initial_conditions[2];

    m_initial_conditions[0] = input * m_matrix_B[0] + tmp_x1 *
m_matrix_A[0][0]  +  tmp_x2  *  m_matrix_A[0][1]  +  tmp_x3  *
m_matrix_A[0][2];
    m_initial_conditions[1] = input * m_matrix_B[1] + tmp_x1 *
m_matrix_A[1][0]  +  tmp_x2  *  m_matrix_A[1][1]  +  tmp_x3  *
m_matrix_A[1][2];
    m_initial_conditions[2] = input * m_matrix_B[2] + tmp_x1 *
m_matrix_A[2][0]  +  tmp_x2  *  m_matrix_A[2][1]  +  tmp_x3  *
m_matrix_A[2][2];

    m_previous_input = input;

    return output;
}
```

ПРИЛОЖЕНИЕ В

sinanalog.h

```
#ifndef SINANALOG_H
#define SINANALOG_H
#include <vector>
#include <cstdint>
#include "blocks/integrator/integrator.h"

class SinAnalog
{
public:
    SinAnalog(std::vector<float> &initial_conditions,
              std::vector<std::vector<float>> &matrix_A,
              std::vector<float> &matrix_B,
              std::vector<float> &matrix_C,
              std::vector<float> &matrix_D);

    ~SinAnalog();

    float getOutput();
    float update(float input, float dt);

private:
    Integrator *m_integrator_X1;
    Integrator *m_integrator_X2;

    float m_previous_input = 0.0;

    std::vector<std::vector<float>> m_matrix_A;
    std::vector<float> m_matrix_B;
    std::vector<float> m_matrix_C;
    std::vector<float> m_matrix_D;

    std::vector<float> m_initial_conditions;
};

#endif // SINANALOG_H
```

```
#include "sinanalog.h"

SinAnalog::SinAnalog(std::vector<float> &initial_conditions,
                     std::vector<std::vector<float>> &matrix_A,
                     std::vector<float> &matrix_B,
                     std::vector<float> &matrix_C,
                     std::vector<float> &matrix_D)

{
    m_integrator_X1 = new Integrator(initial_conditions[0]);
    m_integrator_X2 = new Integrator(initial_conditions[1]);

    m_matrix_A.resize(2);

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        m_matrix_A[i].resize(2);
    }

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        for(uint64_t j = 0; j != matrix_A[i].size(); ++j){
            m_matrix_A[i][j] = matrix_A[i][j];
        }
    }

    for(uint64_t i = 0; i != matrix_B.size(); ++i){
        m_matrix_B.push_back(matrix_B[i]);
    }

    for(uint64_t i = 0; i != matrix_C.size(); ++i){
        m_matrix_C.push_back(matrix_C[i]);
    }

    for(uint64_t i = 0; i != matrix_D.size(); ++i){
        m_matrix_D.push_back(matrix_D[i]);
    }

}

SinAnalog::~SinAnalog()
{
    delete m_integrator_X1;
    delete m_integrator_X2;
}

float SinAnalog::getOutput()
{
    return m_matrix_C[0] * m_integrator_X1->state() +
           m_matrix_C[1] * m_integrator_X2->state() +
           m_matrix_D[0] * m_previous_input - 2.0;
}
```

```
float SinAnalog::update(float input, float dt)
{
    float tmp_x1 = m_integrator_X1->state();
    float tmp_x2 = m_integrator_X2->state();
    float output = getOutput();

    m_integrator_X1->update((m_matrix_A[0][0]    *    tmp_x1)    +
(m_matrix_A[0][1] * tmp_x2)  + (m_matrix_B[0] * input), dt);
    m_integrator_X2->update((m_matrix_A[1][0]    *    tmp_x1)    +
(m_matrix_A[1][1] * tmp_x2)  + (m_matrix_B[1] * input), dt);

    m_previous_input = input;

    return output;
}
```

ПРИЛОЖЕНИЕ Г

sindiscrete.h

```
#ifndef SINDISCRETE_H
#define SINDISCRETE_H
#include <vector>
#include <cstdint>

class SinDiscrete
{
public:
    SinDiscrete(std::vector<float> &initial_conditions,
                std::vector<std::vector<float>> &matrix_A,
                std::vector<float> &matrix_B,
                std::vector<float> &matrix_C,
                std::vector<float> &matrix_D);

    float getOutput();

    float update(float input);

private:
    std::vector<std::vector<float>> m_matrix_A;
    std::vector<float> m_matrix_B;
    std::vector<float> m_matrix_C;
    std::vector<float> m_matrix_D;

    std::vector<float> m_initial_conditions;

    float m_previous_input = 0.0;
};

#endif // SINDISCRETE_H
```

```
#include "sindiscrete.h"

SinDiscrete::SinDiscrete(std::vector<float> &initial_conditions,
                         std::vector<std::vector<float>>
&matrix_A,
                         std::vector<float> &matrix_B,
                         std::vector<float> &matrix_C,
                         std::vector<float> &matrix_D)
{
    m_matrix_A.resize(2);

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        m_matrix_A[i].resize(2);
    }

    for(uint64_t i = 0; i != matrix_A.size(); ++i){
        for(uint64_t j = 0; j != matrix_A[i].size(); ++j){
            m_matrix_A[i][j] = matrix_A[i][j];
        }
    }

    for(uint64_t i = 0; i != matrix_B.size(); ++i){
        m_matrix_B.push_back(matrix_B[i]);
    }

    for(uint64_t i = 0; i != matrix_C.size(); ++i){
        m_matrix_C.push_back(matrix_C[i]);
    }

    for(uint64_t i = 0; i != matrix_D.size(); ++i){
        m_matrix_D.push_back(matrix_D[i]);
    }

    for(uint64_t i = 0; i != initial_conditions.size(); ++i){
        m_initial_conditions.push_back(initial_conditions[i]);
    }
}

float SinDiscrete::getOutput()
{
    return (m_initial_conditions[0] * m_matrix_C[0] +
            m_initial_conditions[1] * m_matrix_C[1] +
            m_previous_input * m_matrix_D[0]) - 2.0;
}

float SinDiscrete::update(float input)
{
    float output = getOutput();

    float tmp_x1 = m_initial_conditions[0];
    float tmp_x2 = m_initial_conditions[1];
```

```
    m_initial_conditions[0] = input * m_matrix_B[0] + tmp_x1 *
m_matrix_A[0][0] + tmp_x2 * m_matrix_A[0][1];
    m_initial_conditions[1] = input * m_matrix_B[1] + tmp_x1 *
m_matrix_A[1][0] + tmp_x2 * m_matrix_A[1][1];

    m_previous_input = input;

    return output;
}
```