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 Е4260,

Карпаев Е.В.

ПРОВЕРИЛ:

Томашевич С. И.

Санкт-Петербург

2023 г.

ОГЛАВЛЕНИЕ

[ЗАДАНИЕ 3](#_Toc152113378)

[МОДЕЛИРОВАНИЕ ПЕРЕДАТЧНОЙ ФУНКЦИИ В СРЕДЕ SIMULINK 4](#_Toc152113379)

[МОДЕЛИРОВАНИЕ ВХОДЯЩЕГО СИГНАЛА В MATLAB 8](#_Toc152113380)

[РЕАЛИЗАЦИЯ ПЕРЕДАТОЧНОЙ ФУНКЦИИ НА ЯЗЫКЕ C++ 12](#_Toc152113381)

[РЕАЛИЗАЦИЯ ВХОДНОГО СИГНАЛА НА ЯЗЫКЕ C++ 16](#_Toc152113382)

[СРАВНЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ 19](#_Toc152113383)

[ВЫВОДЫ 23](#_Toc152113384)

[ПРИЛОЖЕНИЕ А 24](#_Toc152113385)

[ПРИЛОЖЕНИЕ Б 27](#_Toc152113386)

[ПРИЛОЖЕНИЕ В 30](#_Toc152113387)

[ПРИЛОЖЕНИЕ Г 33](#_Toc152113388)

ЗАДАНИЕ

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

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

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.

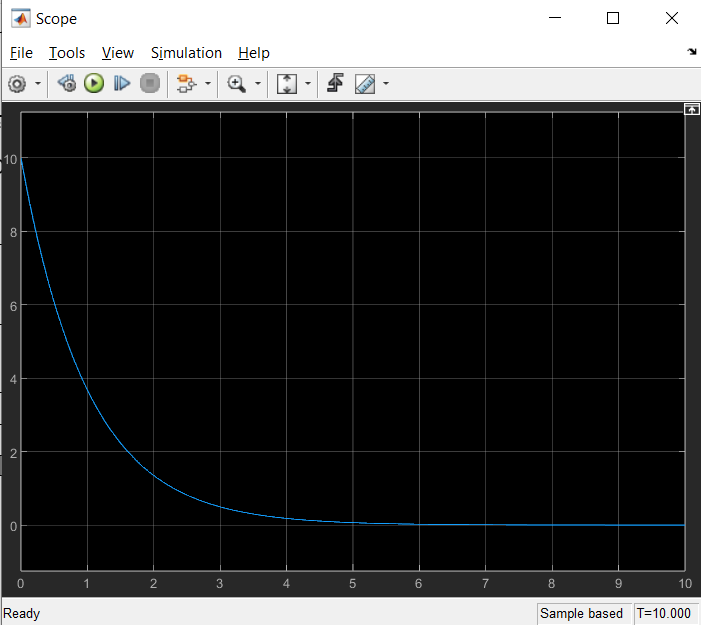
6. Провести сравнения результатов, полученных в среде Simulink и на языке C++.

МОДЕЛИРОВАНИЕ ПЕРЕДАТЧНОЙ ФУНКЦИИ В СРЕДЕ SIMULINK

Из передаточной функции , используя функционал MATLAB, получим матрицы A, B, C и D, которые будут описывать состояния нашей системы.

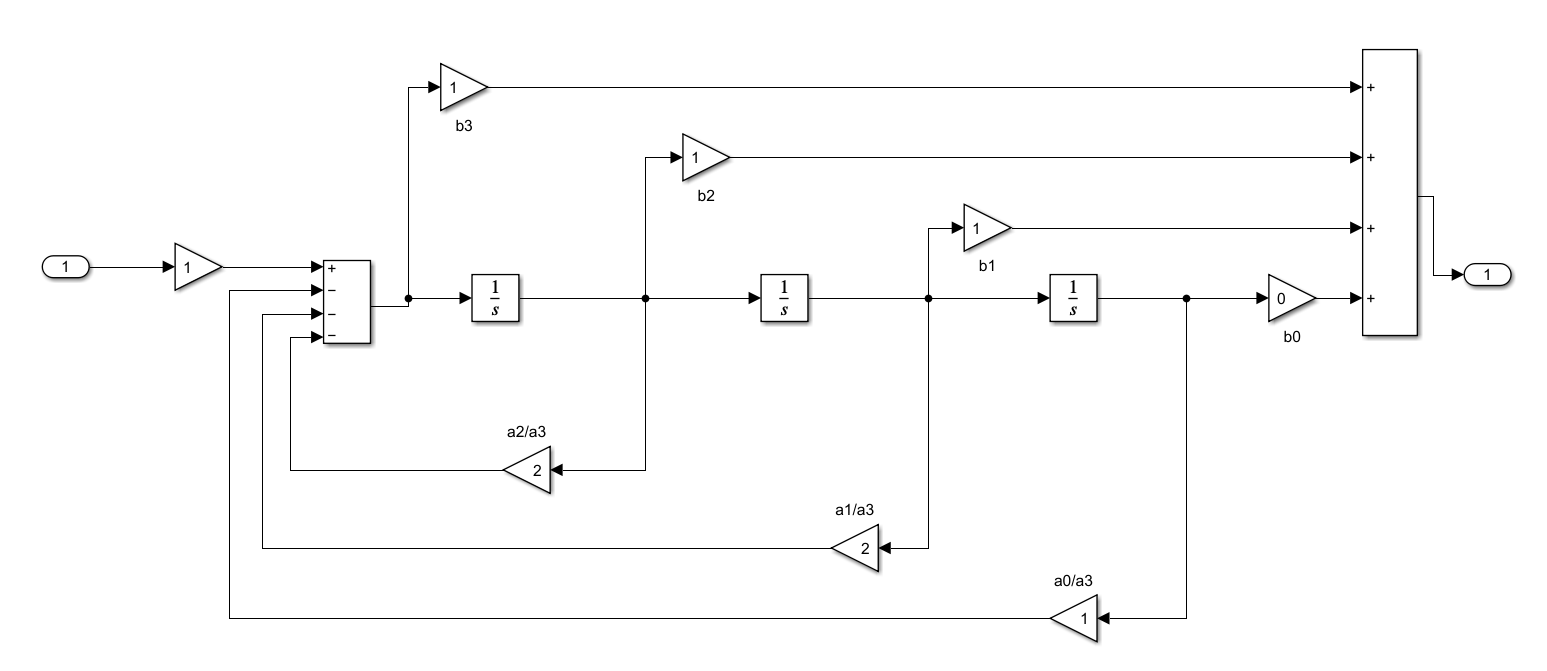
D = [1]

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).

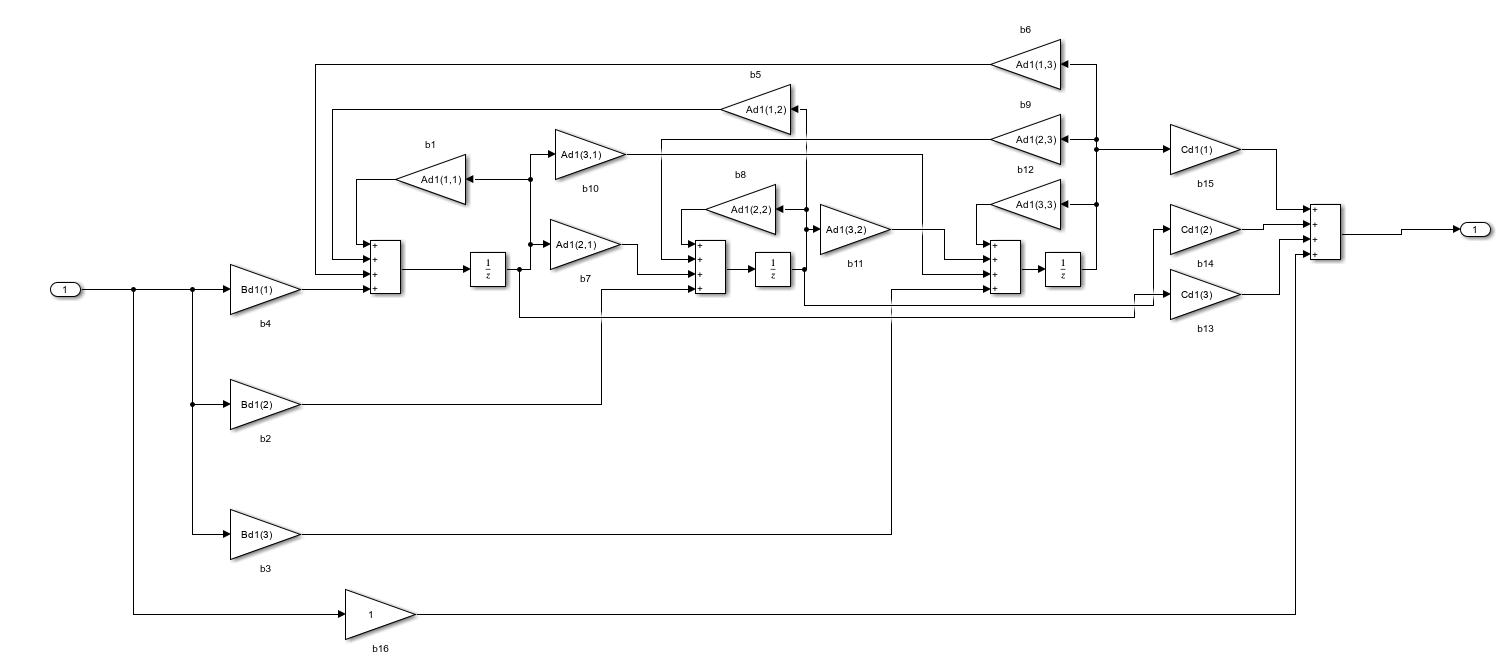


*Рисунок 2. Структурная схема непрерывной передаточной функции*

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

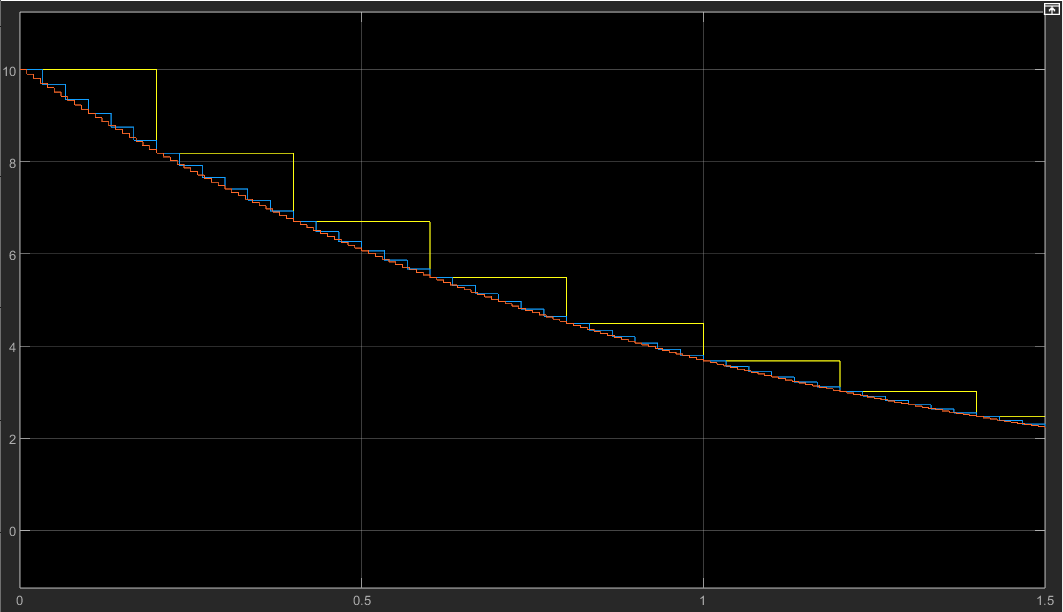
* Для частоты 5 Гц
* Для частоты 30 Гц
* Для частоты 100 Гц

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



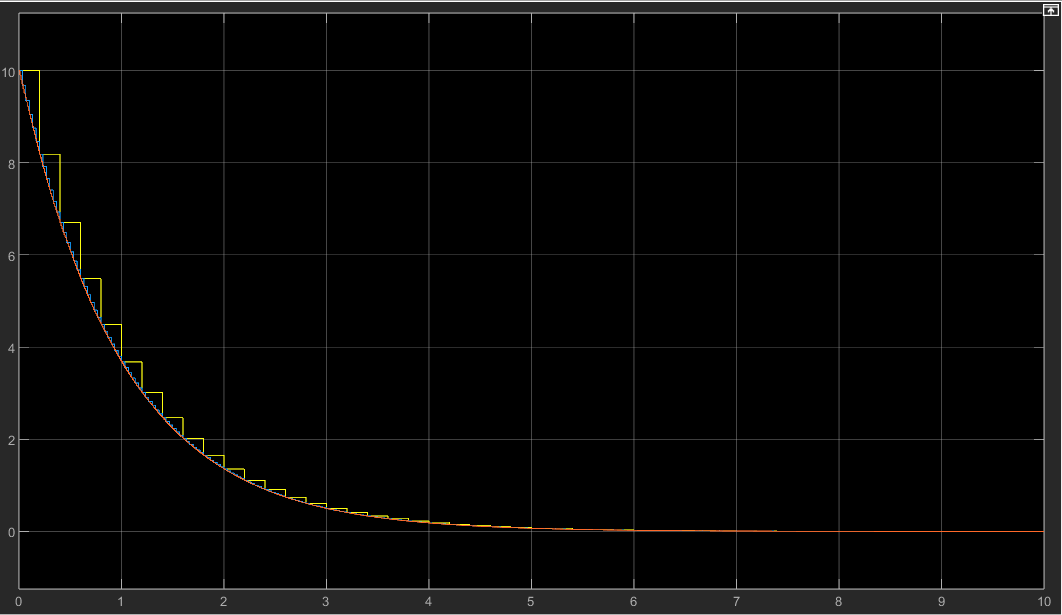
*Рисунок 3. Структурная схема дискретной передаточной функции*

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:

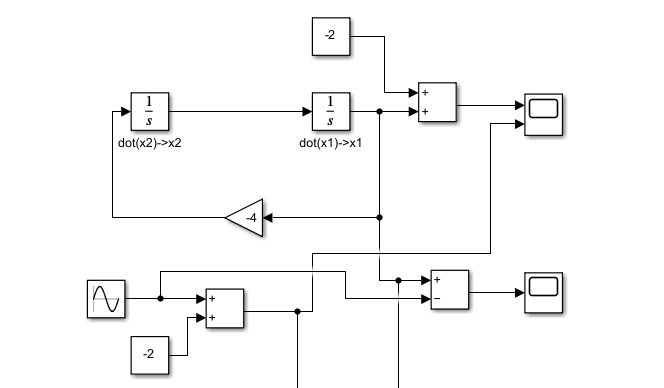
При помощи функции diff в MATLAB вычислим первую и вторую производные данной функции:

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

D = [0]

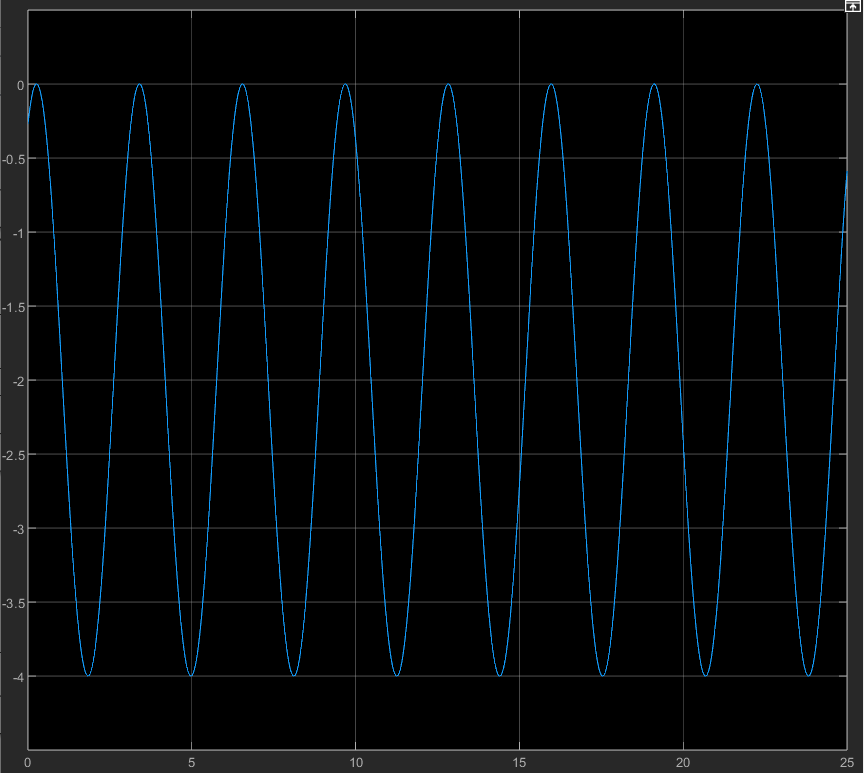
Построим структурную схему (Рисунок 6), использую блоки интегрирования и усиления. Блок суммирования для смещения функции на «-2» и блок «Sine Wave», в котором зададим данный сигнал.

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).

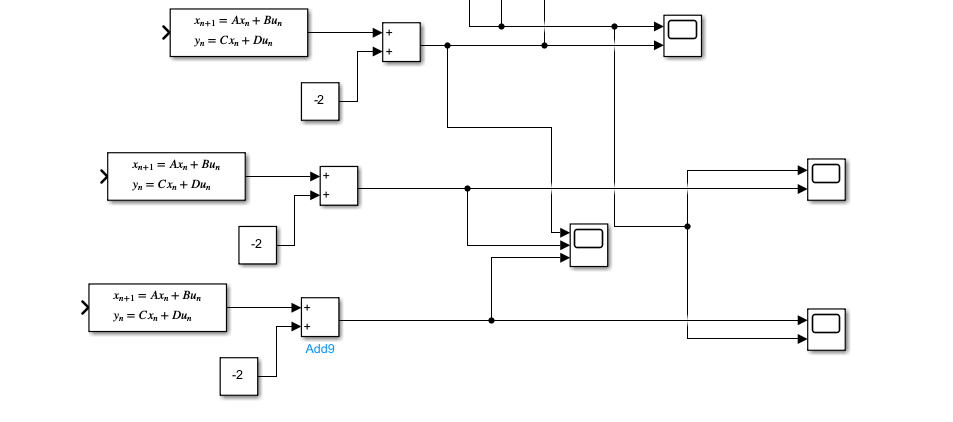


*Рисунок 7. График непрерывного синуса*

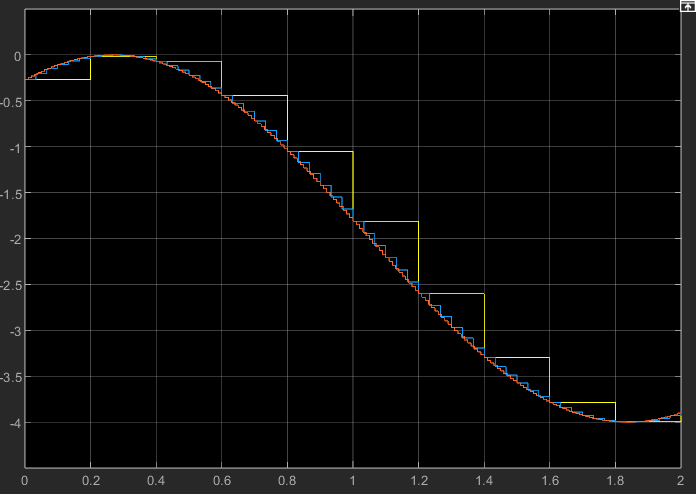
Приведем сигнал к дискретному виду. Для этого используем функцию «c2d» с указанием частоты дискретизации. Получим следующие матрицы:

* Для частоты 5 Гц
* Для частоты 30 Гц
* Для частоты 100 Гц

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).

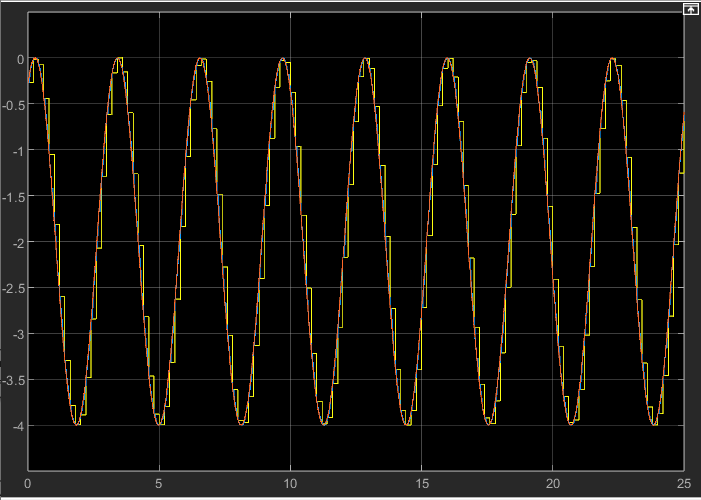


*Рисунок 8. Схема подключения дискретных блоков State-Space*

**

*Рисунок 9. Графики дискретных синусов (2с)*

Проведем моделирование для 25 секунд (Рисунок 10).

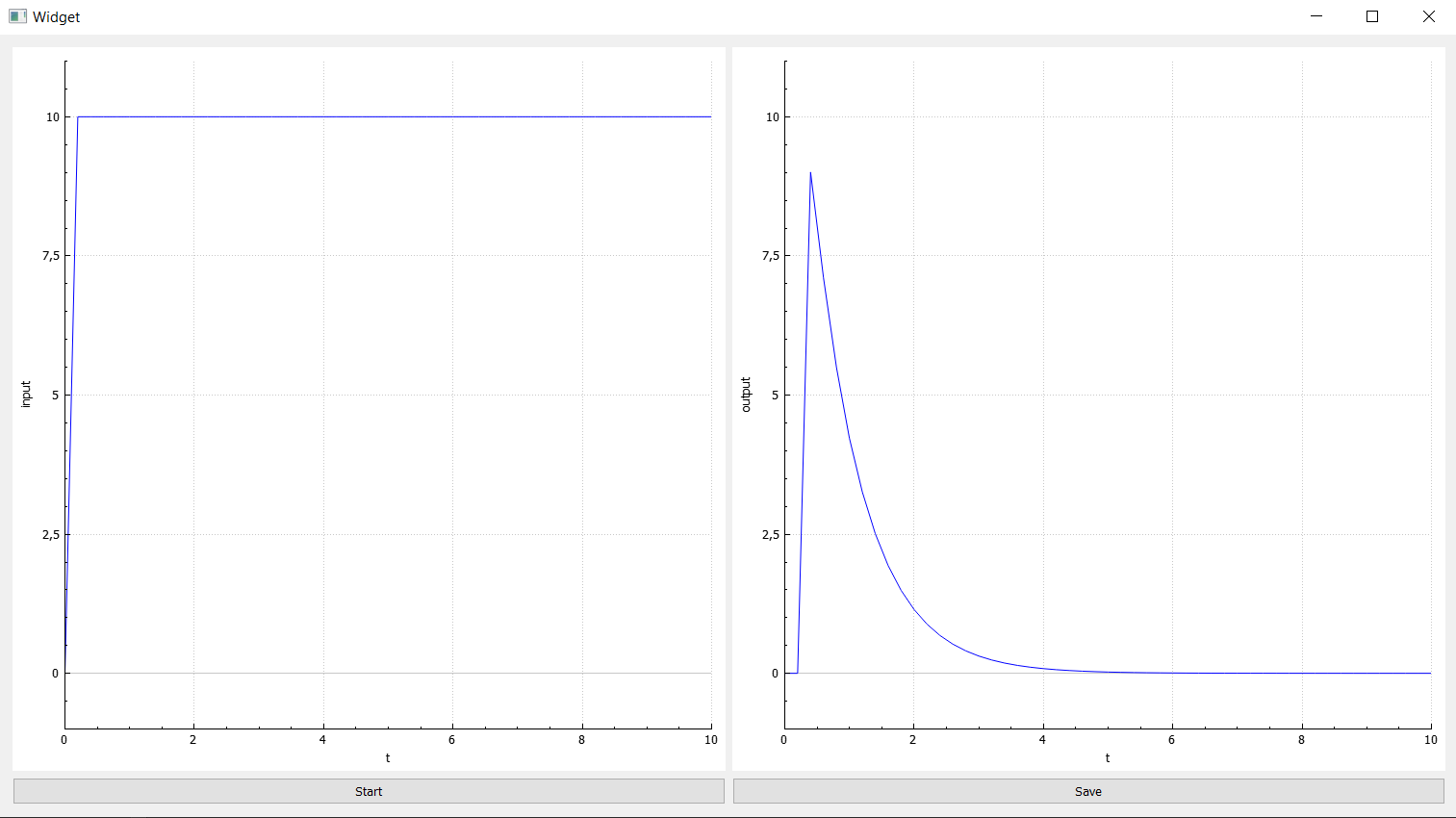


*Рисунок 10. Графики дискретных синусов (25с)*

РЕАЛИЗАЦИЯ ПЕРЕДАТОЧНОЙ ФУНКЦИИ НА ЯЗЫКЕ C++

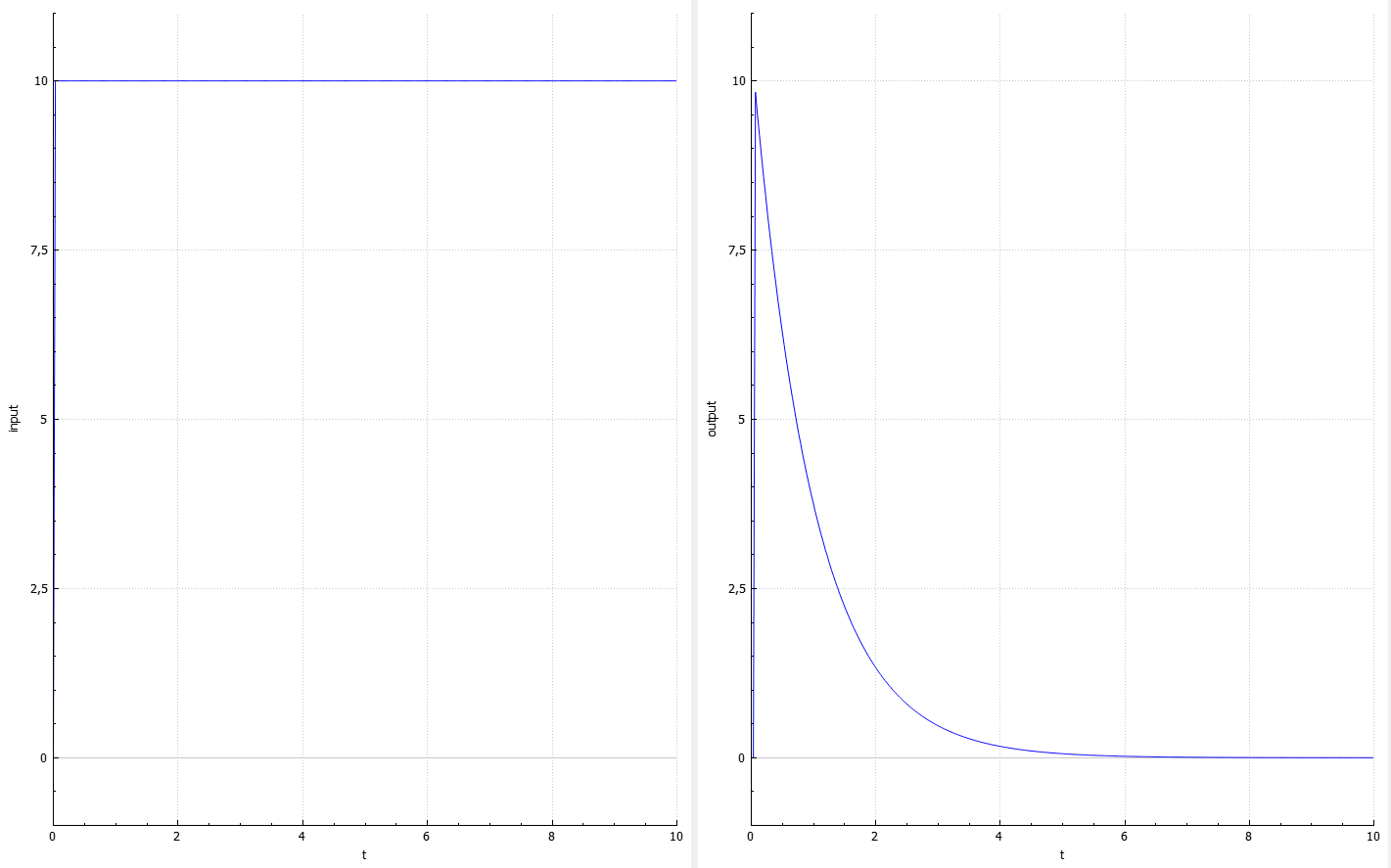
Используя матрицы состояний, полученные в ходе построения системы в MATLAB, был реализован класс *StateSpace* для непрерывной системы(см. Приложение А) и класс *Discrete* для дискретной системы (см. Приложение Б).

Проведем моделирование непрерывной системы на частоте 5 Гц (Рисунок 11).



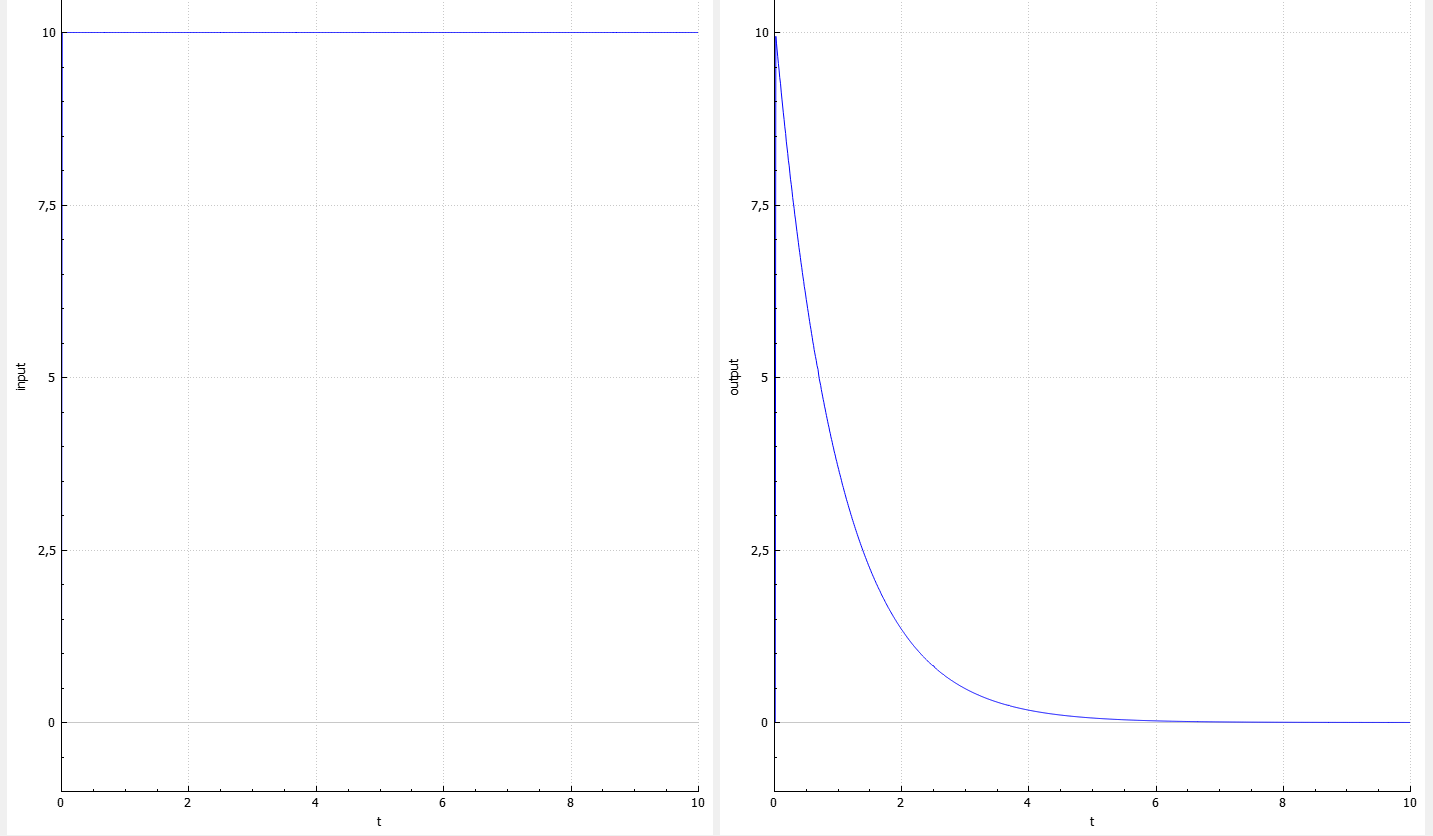
*Рисунок 11. Непрерывная система с частотой 5 Гц*

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



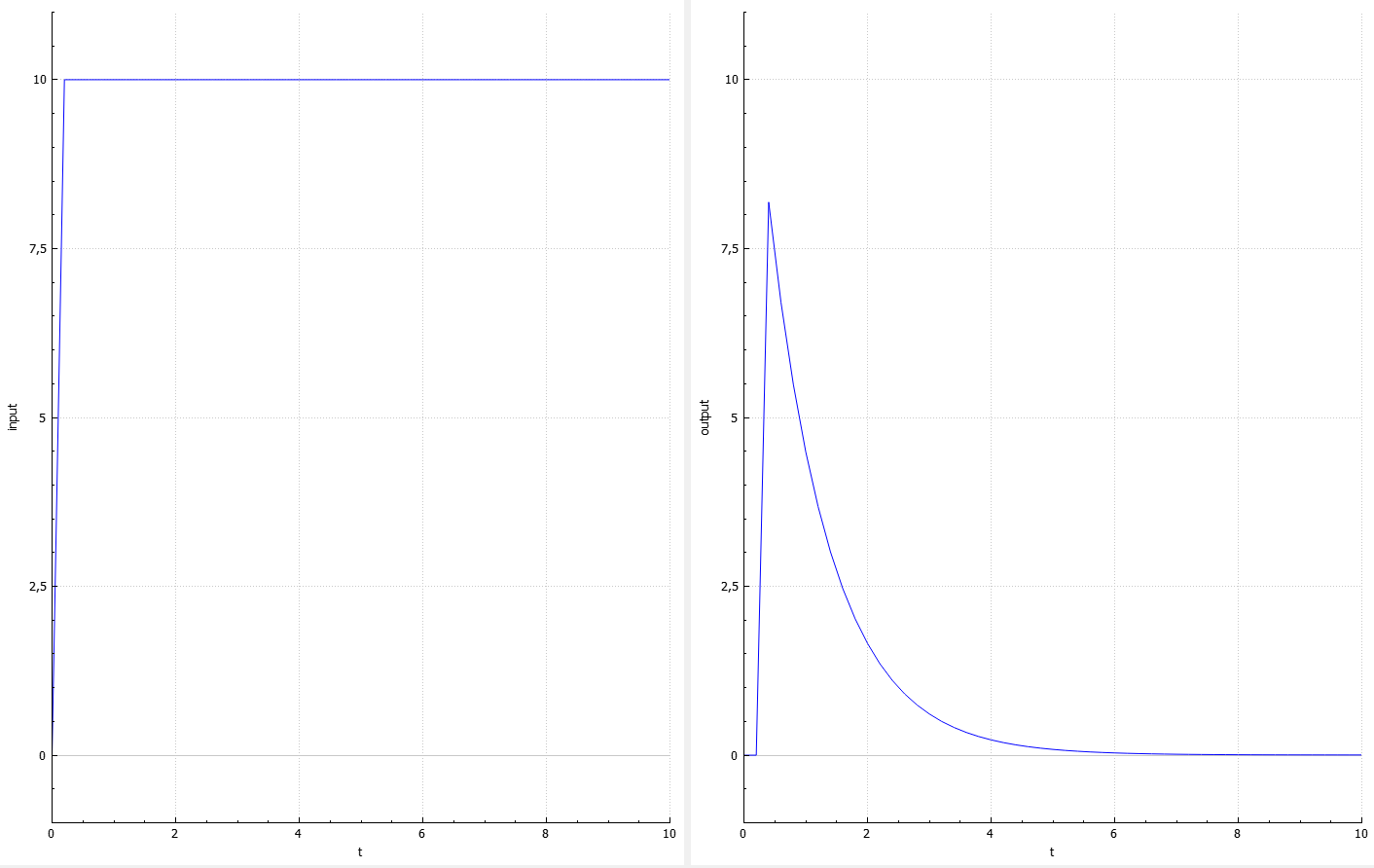
*Рисунок 12. Непрерывная система с частотой 30 Гц*

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



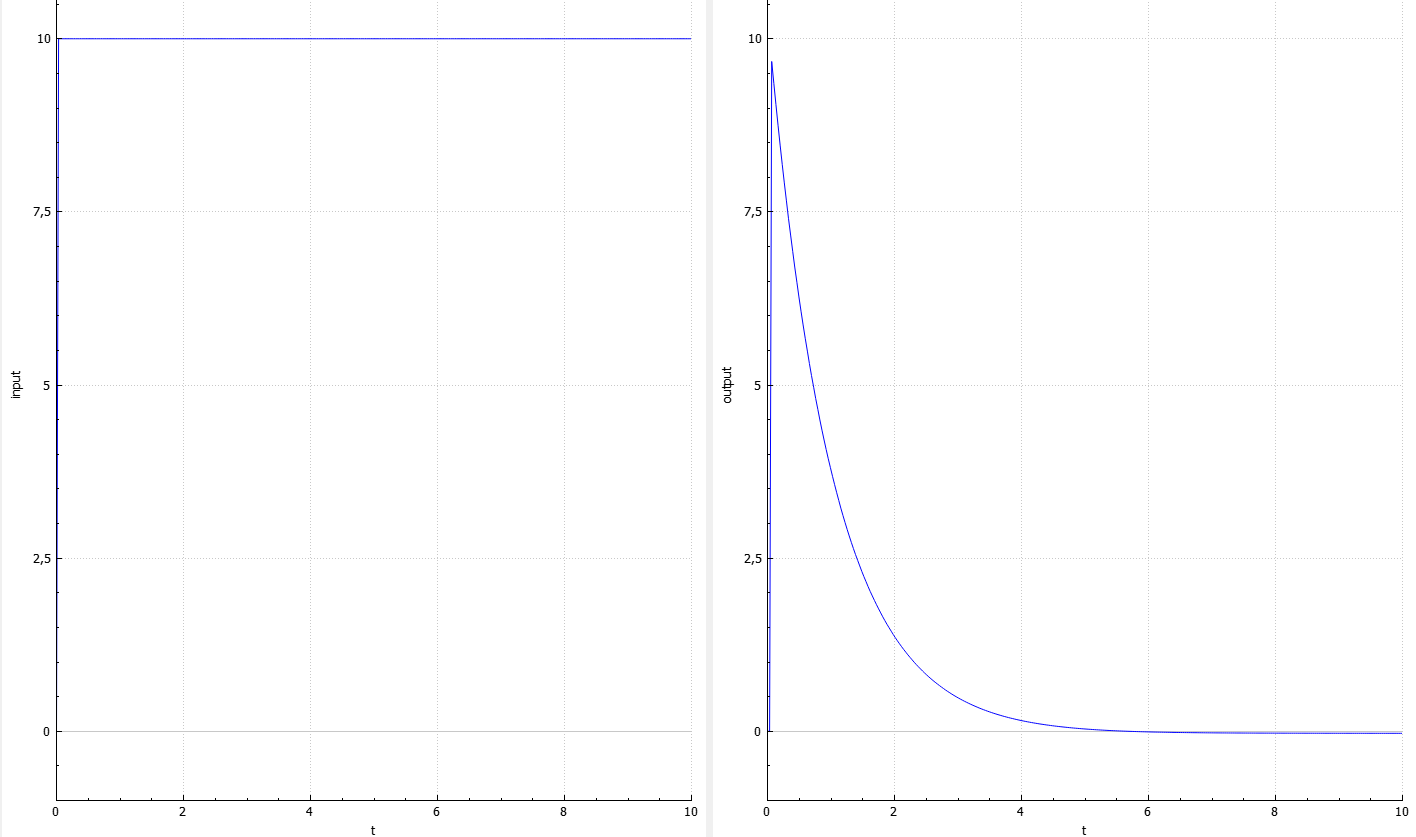
*Рисунок 13. Непрерывная система с частотой 100 Гц*

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



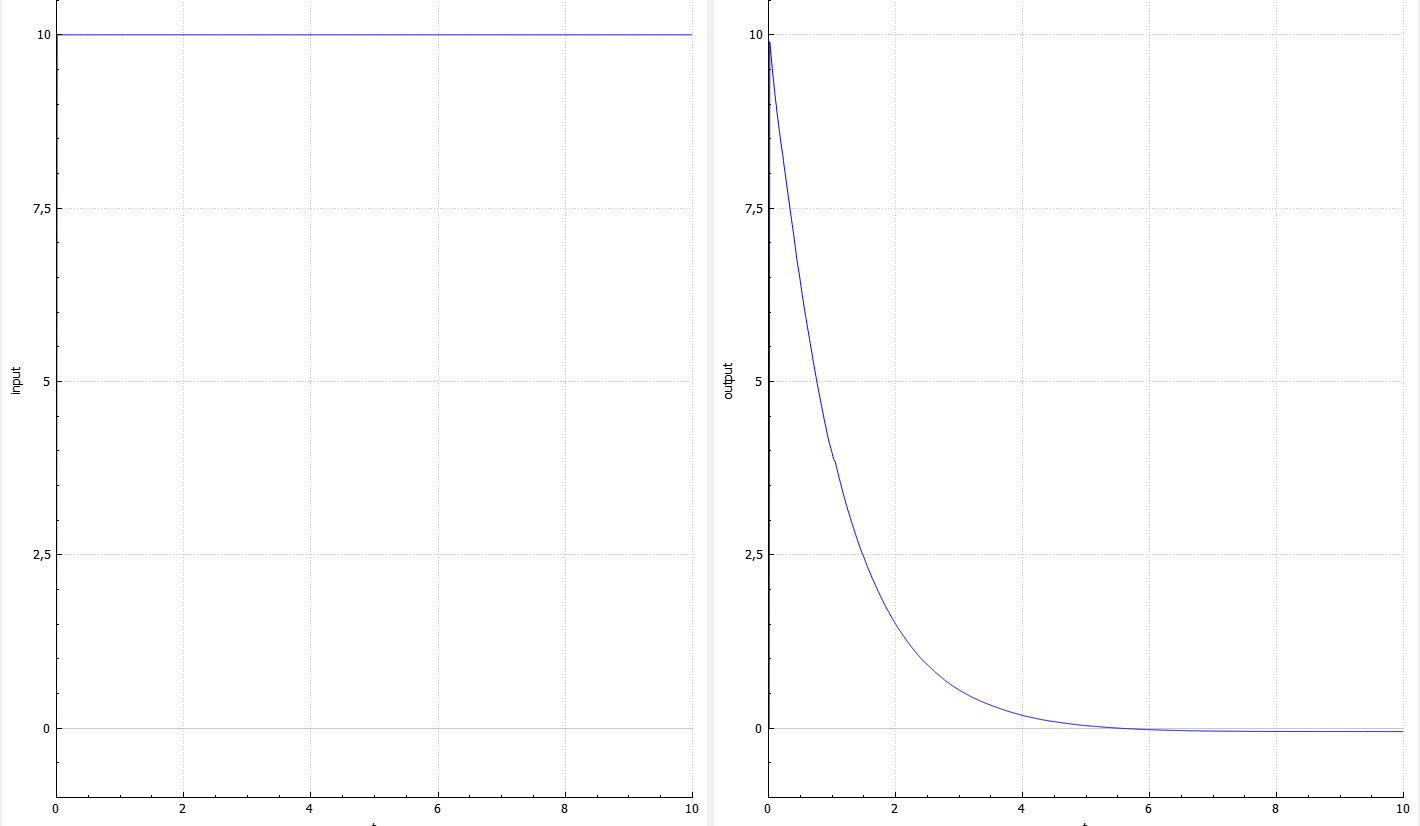
*Рисунок 14. Дискретная система с частотой 5 Гц*

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).

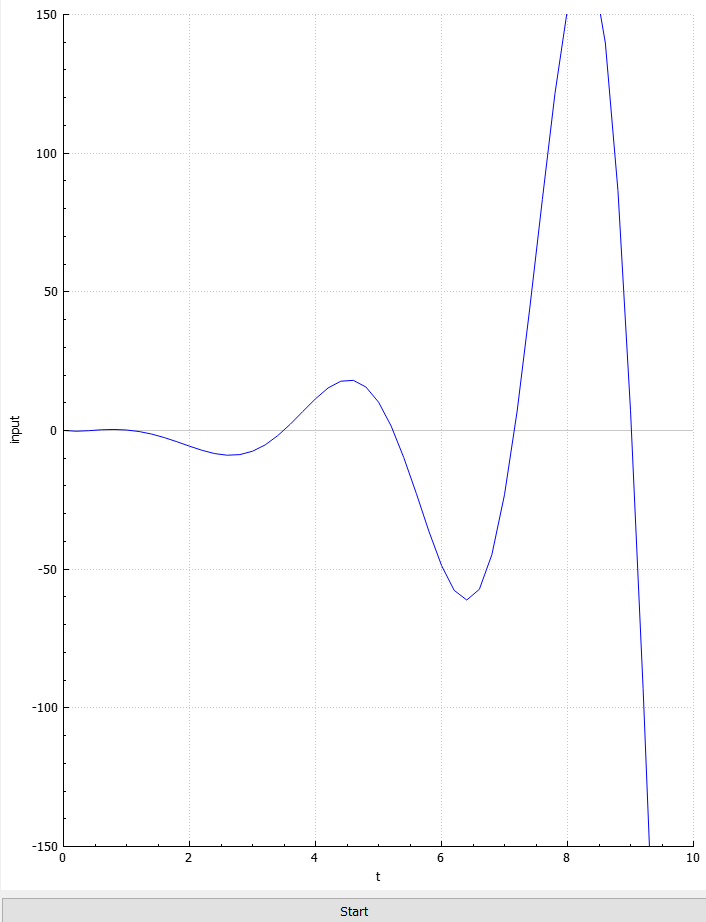


*Рисунок 16. Дискретная система с частотой 100 Гц*

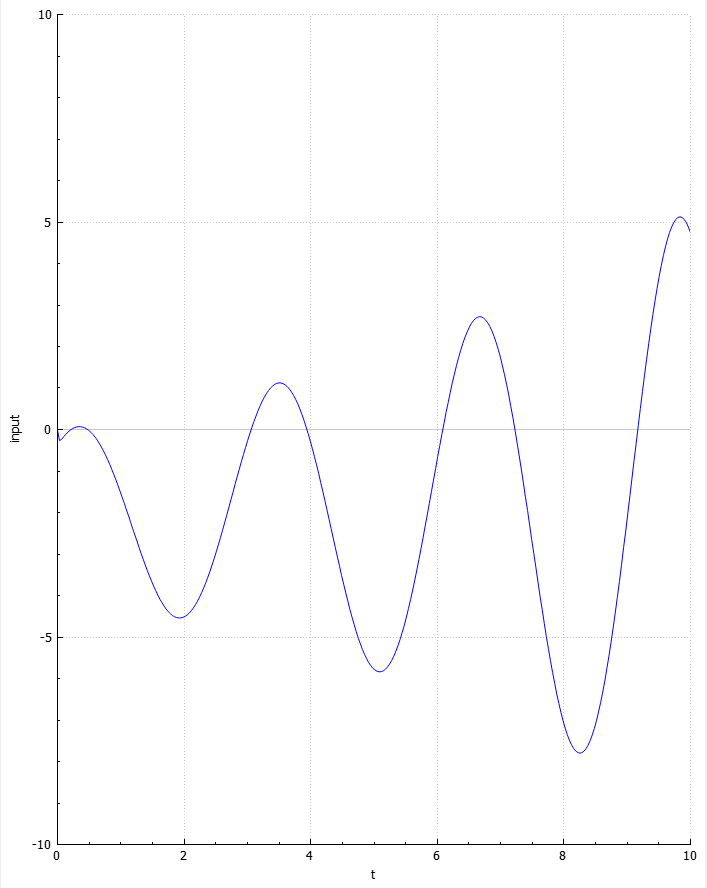
Как видно из графиков, непрерывная система совпадает и дискретной, если при моделировании используется одинаковая для систем частота. Так же заметим, что чем выше частота, тем раньше начинает происходить моделирование выходного сигнала.

РЕАЛИЗАЦИЯ ВХОДНОГО СИГНАЛА НА ЯЗЫКЕ C++

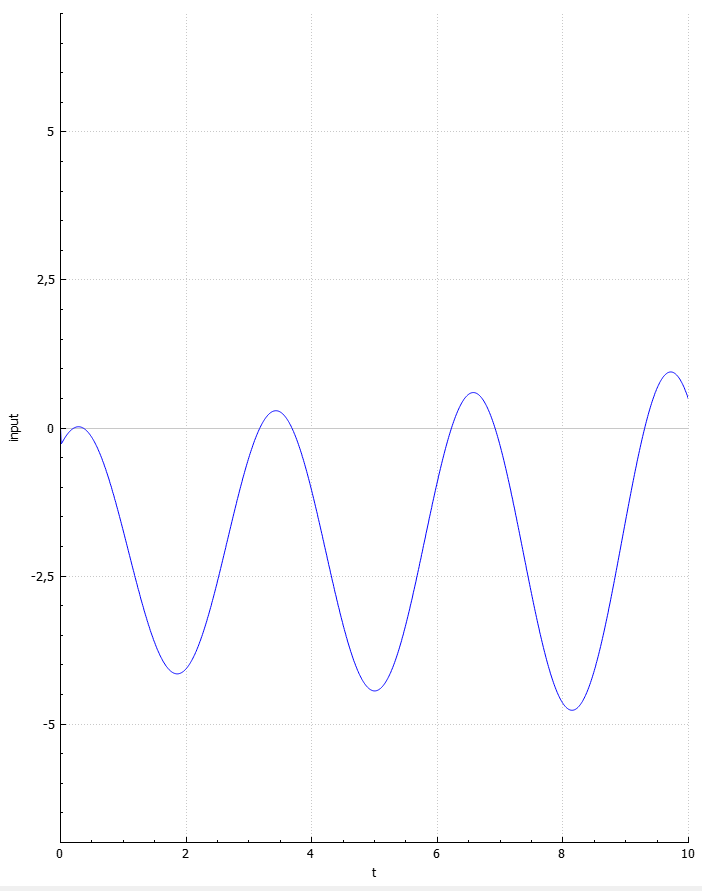
Промоделируем непрерывный синус, код которого реализован в классе *SinAnalog* (см. Приложение В), на частотах 5 Гц, 30 Гц и 100 Гц (Рисунок 17-19).



*Рисунок 17. Непрерывный входной сигнал с частотой 5 Гц*

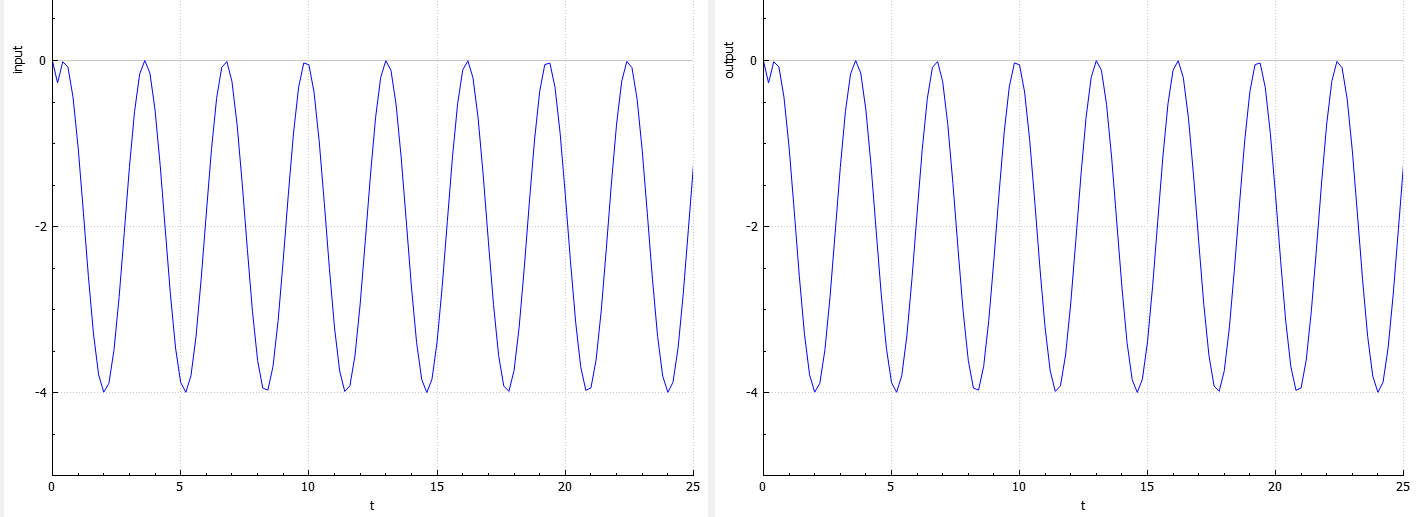
**

*Рисунок 18. Непрерывный входной сигнал с частотой 30 Гц*

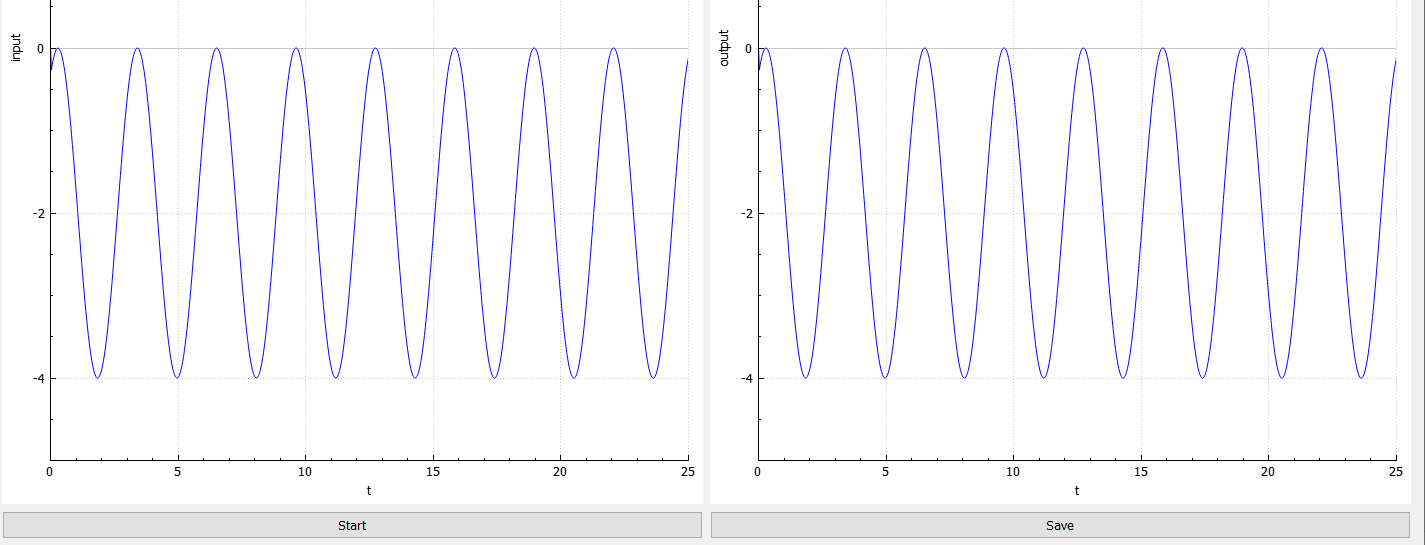
**

*Рисунок 19. Непрерывный входной сигнал с частотой 100 Гц*

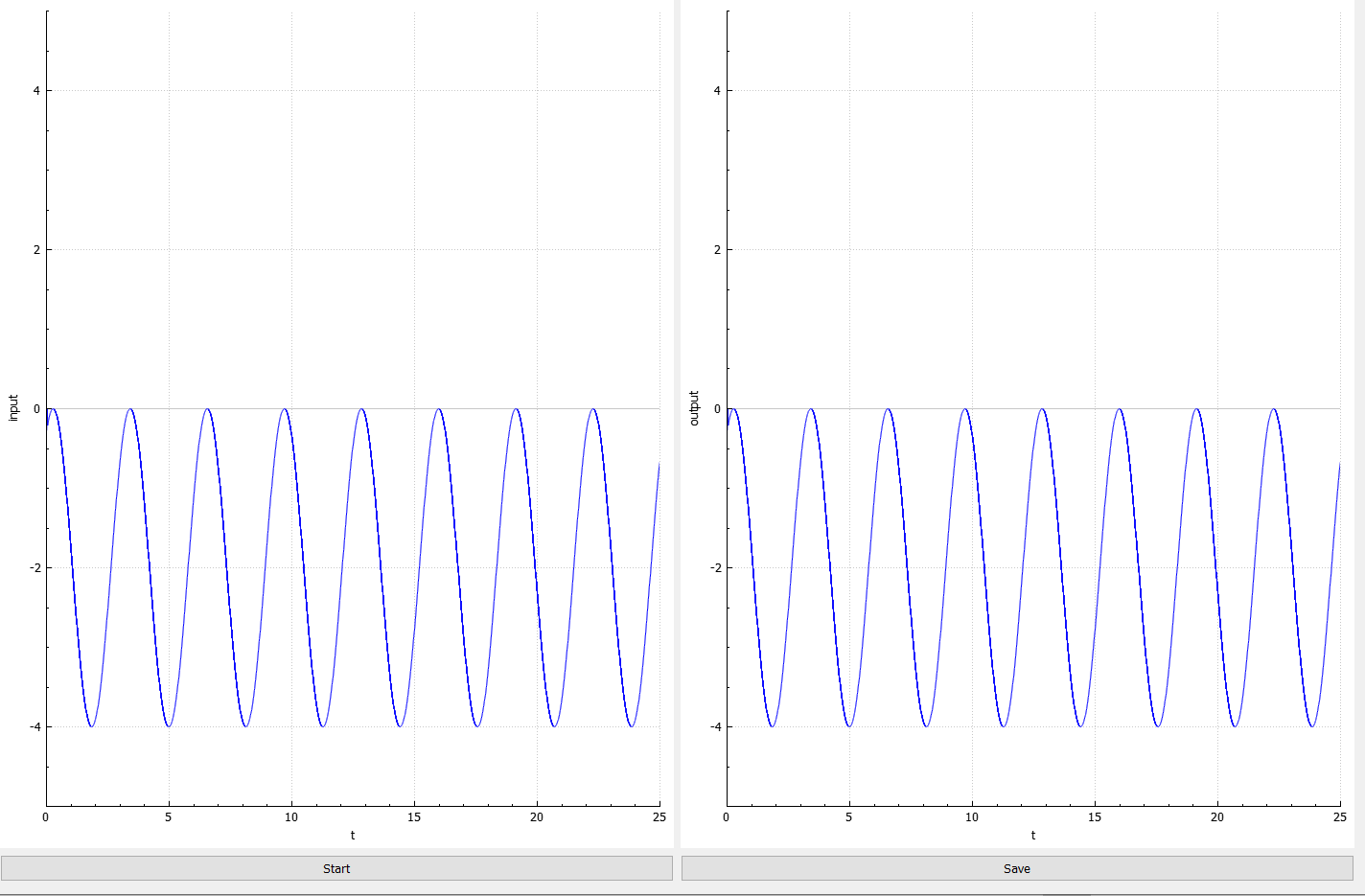
Теперь проведем моделирование дискретного входного сигнала, код которого реализован в классе *SinDiscrete* (см. Приложение Г), с частотами 5 Гц, 30 Гц и 100 Гц (Рисунок 20-22).



*Рисунок 20. Дискретный входной сигнал с частотой 5 Гц*

**

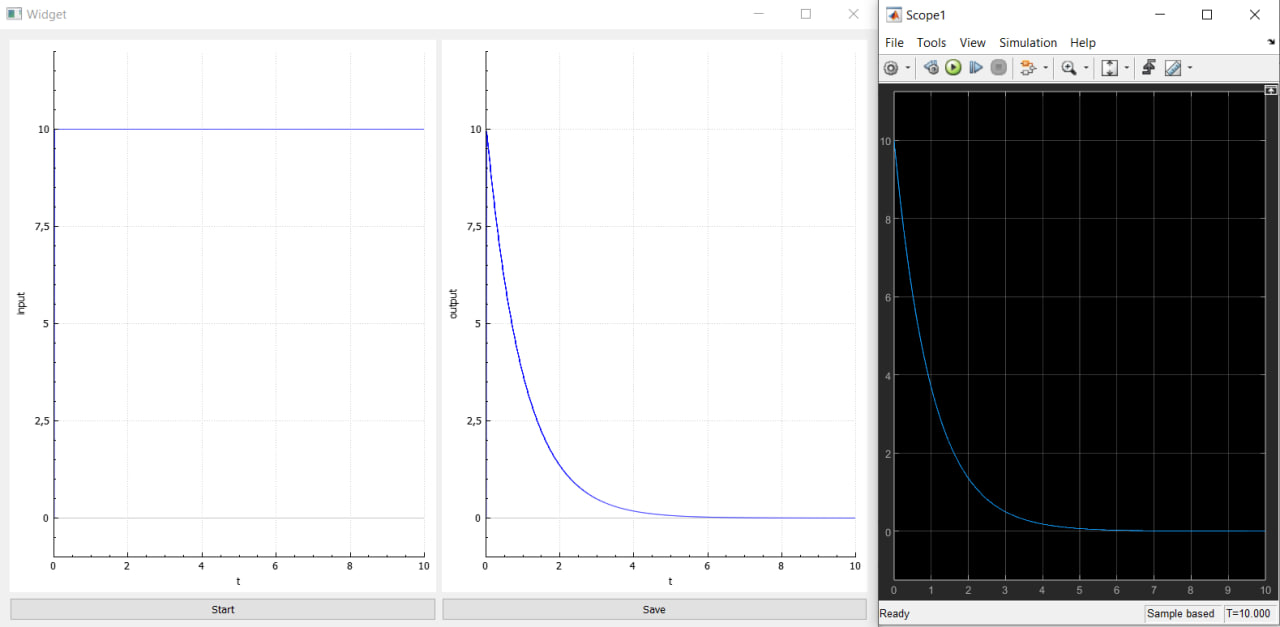
*Рисунок 21. Дискретный входной сигнал с частотой 30 Гц*

**

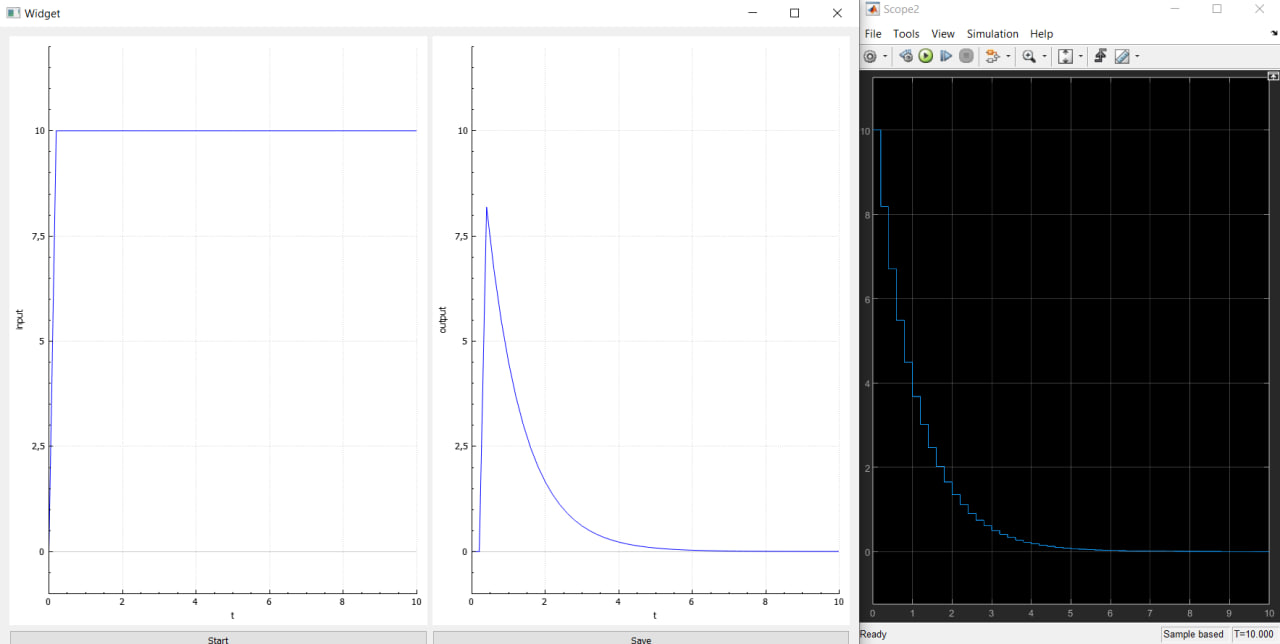
*Рисунок 21. Дискретный входной сигнал с частотой 100 Гц*

СРАВНЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

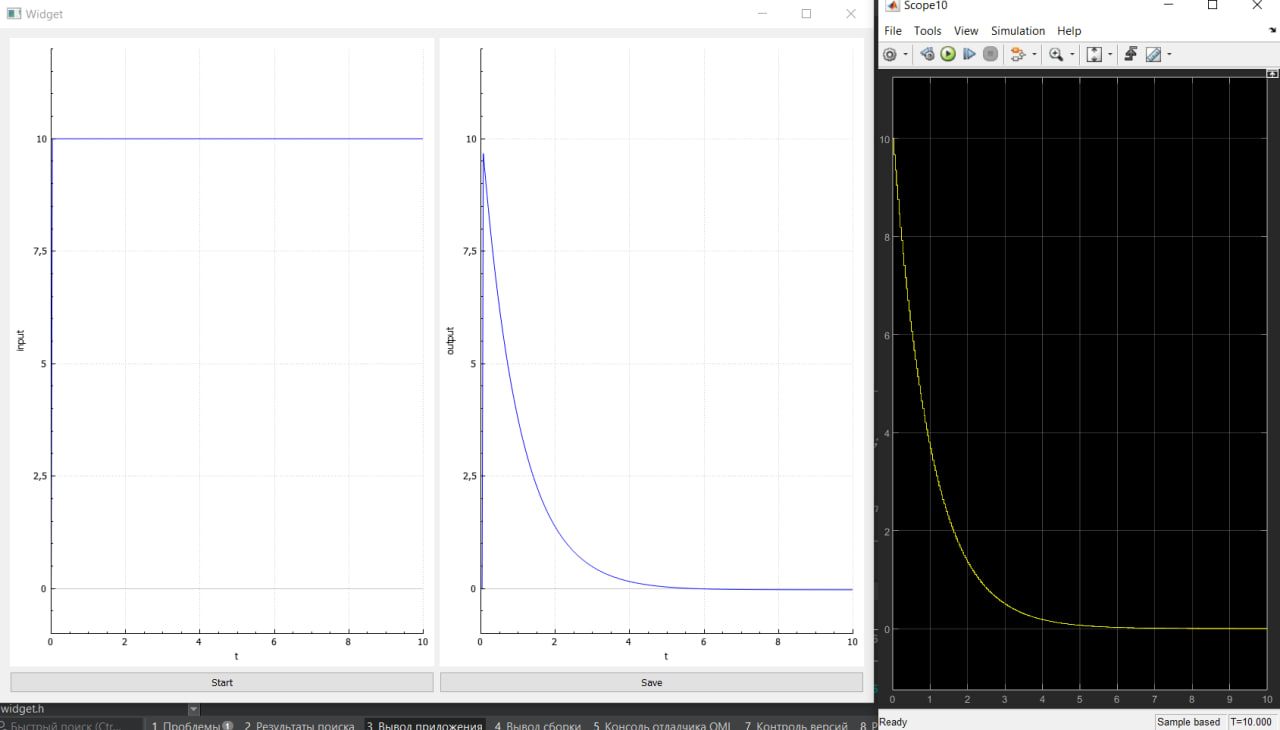
Чтобы убедиться, что все графики были получены верно, сравним попарно их реализацию в MATLAB и на языке C++ (Рисунок 23 – 30).



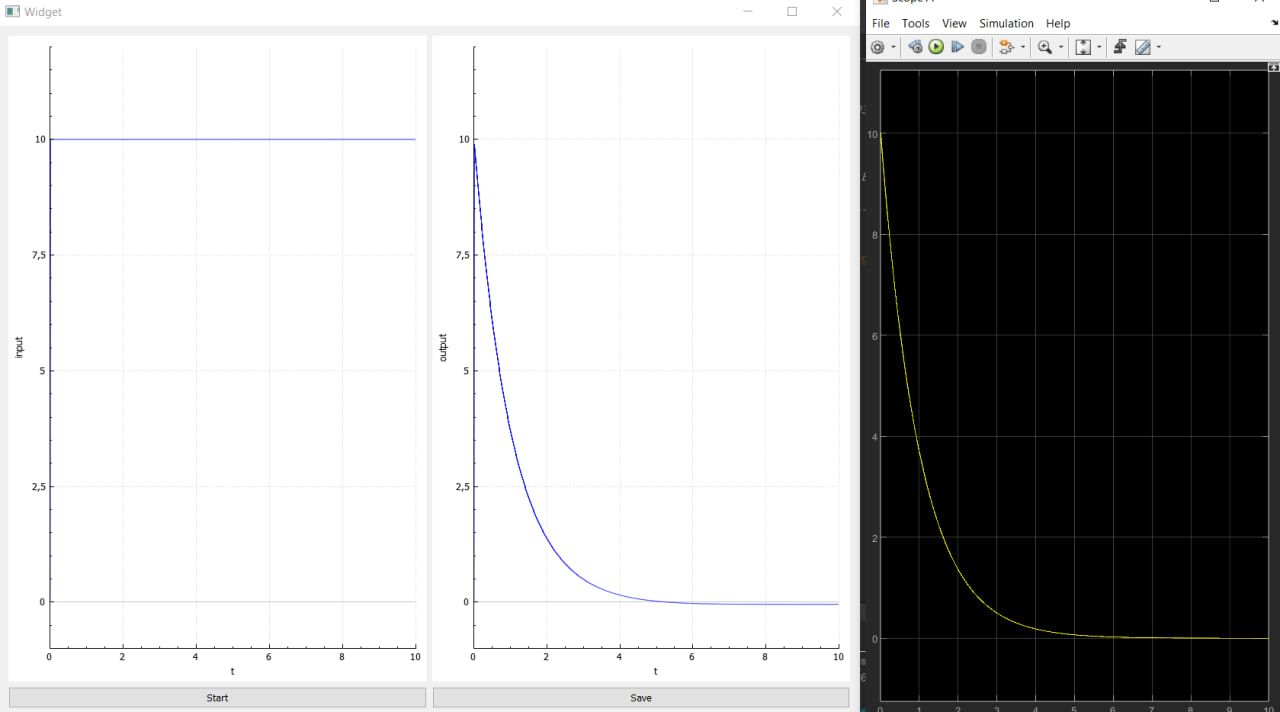
*Рисунок 23. Сравнение непрерывной системы*



*Рисунок 24. Сравнение дискретной системы (5 Гц)*

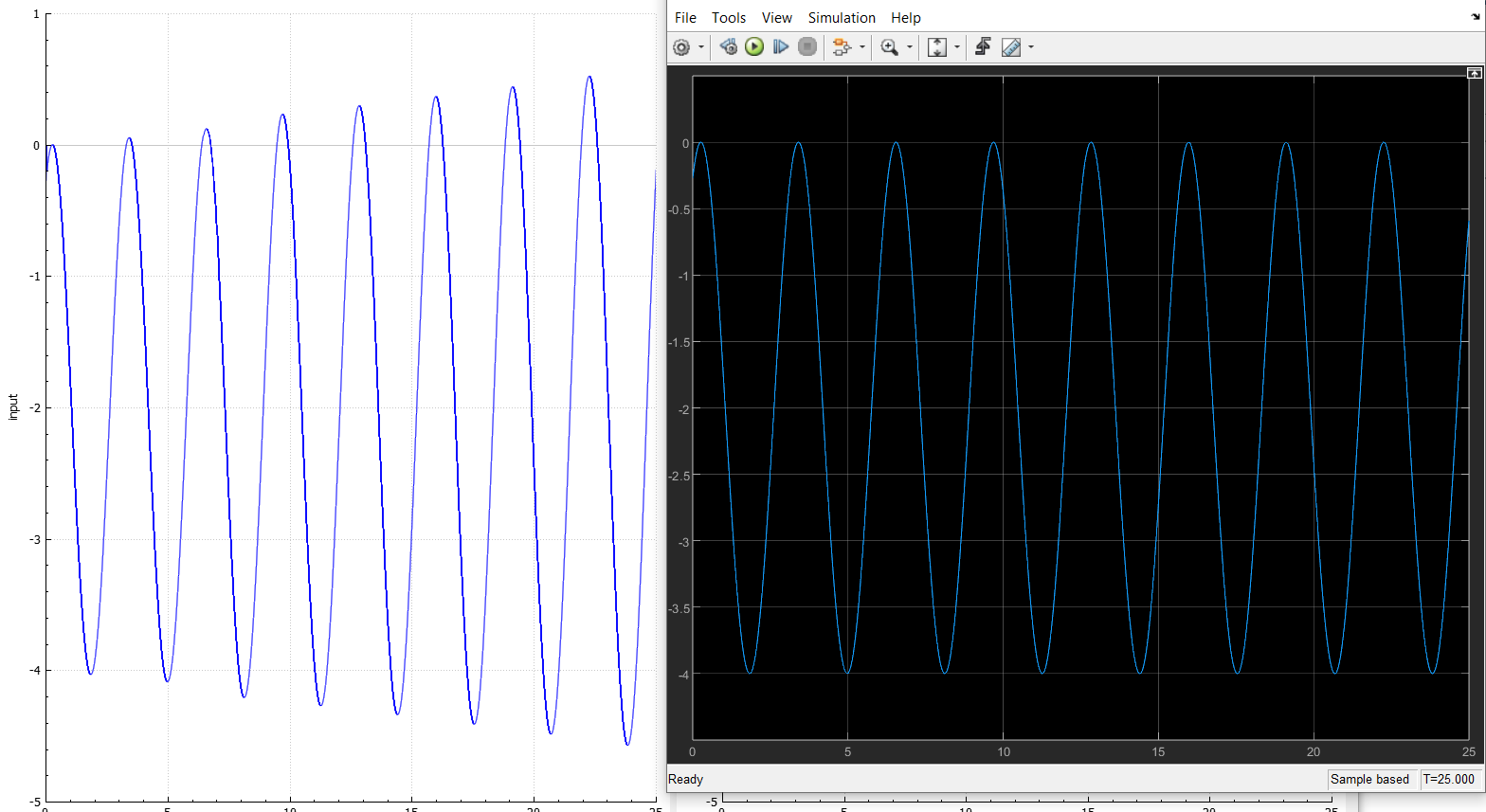


*Рисунок 25. Сравнение дискретной системы (30 Гц)*

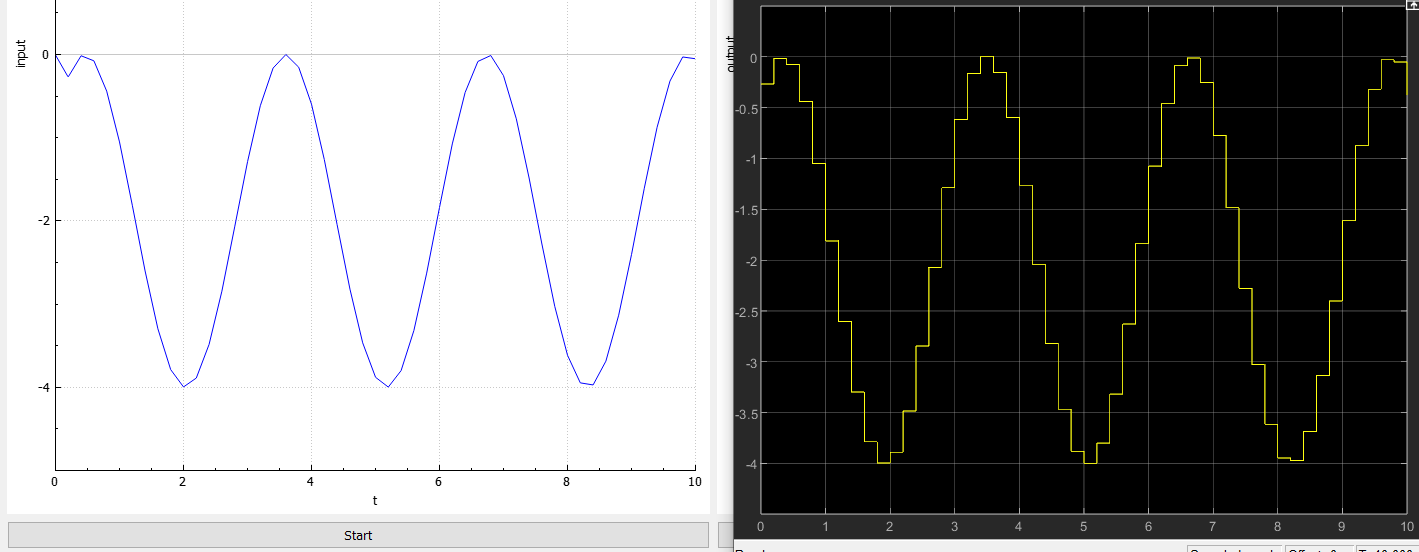


*Рисунок 26. Сравнение дискретной системы (100 Гц)*

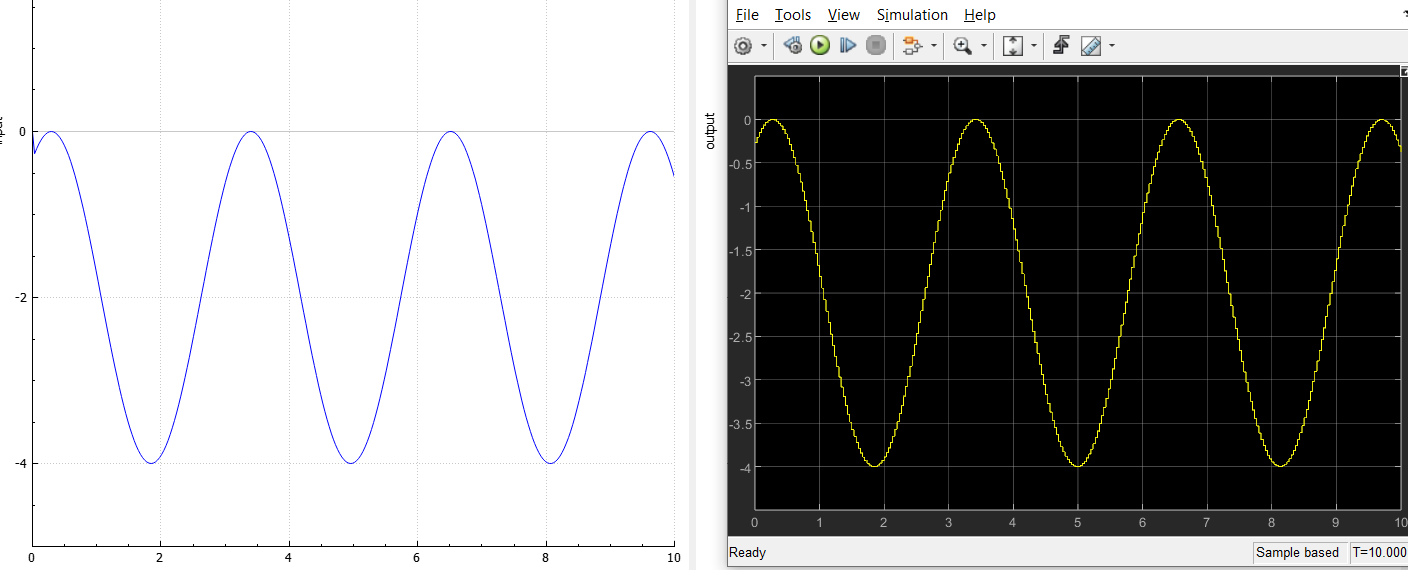
При сравнении непрерывного синуса, реализованного на языке C++, *SAMPLINGTIMEMSEC* равен 1, что равно 1кГц.



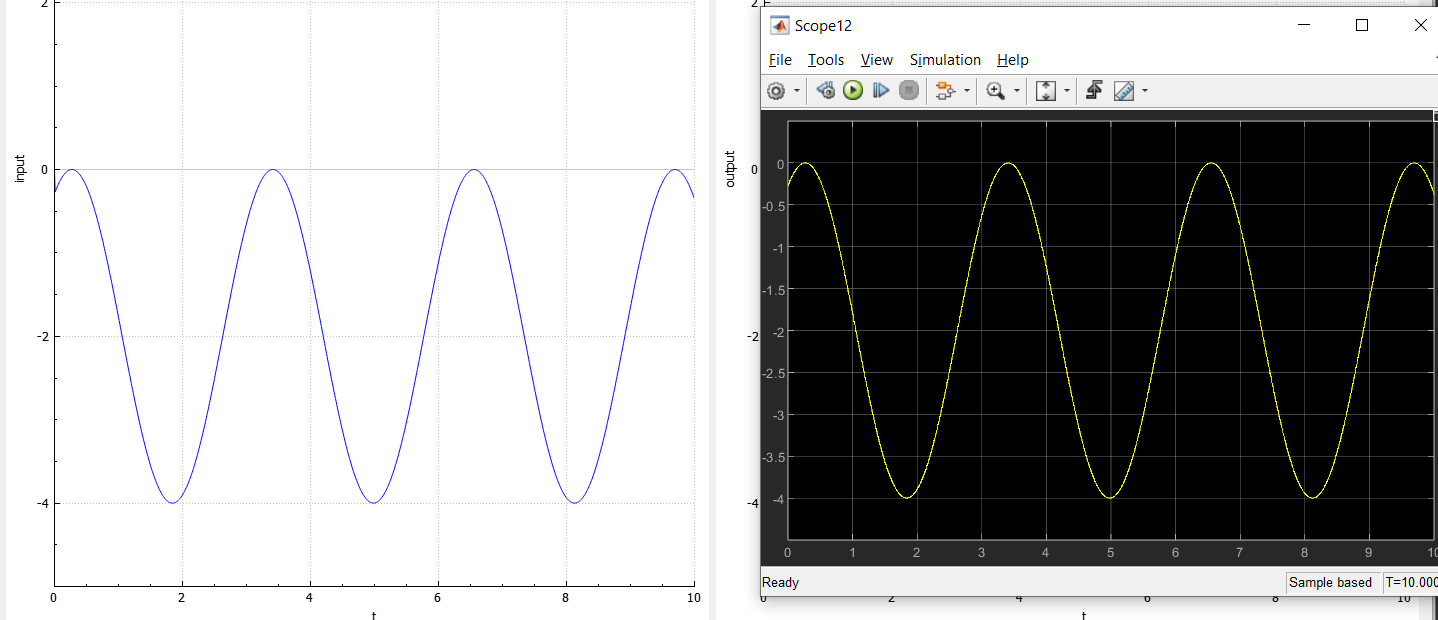
*Рисунок 27. Сравнение непрерывного синуса*



*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

statespace.cpp

#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

discrete.cpp

#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

sinanalog.cpp

#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

sindiscrete.cpp

#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;

}