МИНОБРНАУКИ РОССИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

ОТЧЕТ

по практической работе

по дисциплине «Программирование систем управления»

Студент гр. R4140	____________________	Федоров И.А.
Преподаватель	____________________	Томашевич С.И.

Санкт-Петербург

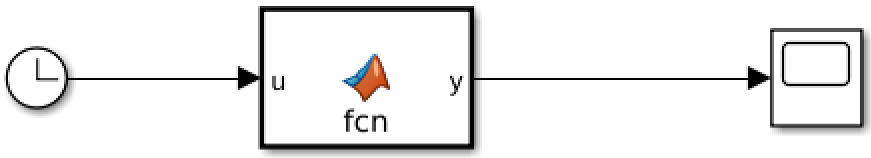
2022

<u>Вариант 16</u>:

$$u(t) = e^{-0.5t} + \cos(-0.5t)$$

$$\frac{y(t)}{u(t)} = \frac{s^3 + 4s^2 + 4s + 2}{s^3 + 4s^2 + 4s + 1}$$

1) Простейшая модель генератора в среде Simulink может быть реализована с помощью блока MATLAB Function:




```
function y = fcn(u)

y = exp(-0.5*u) + cos(-0.5 * u);
```

Рисунок 1 – Простейшая модель генератора

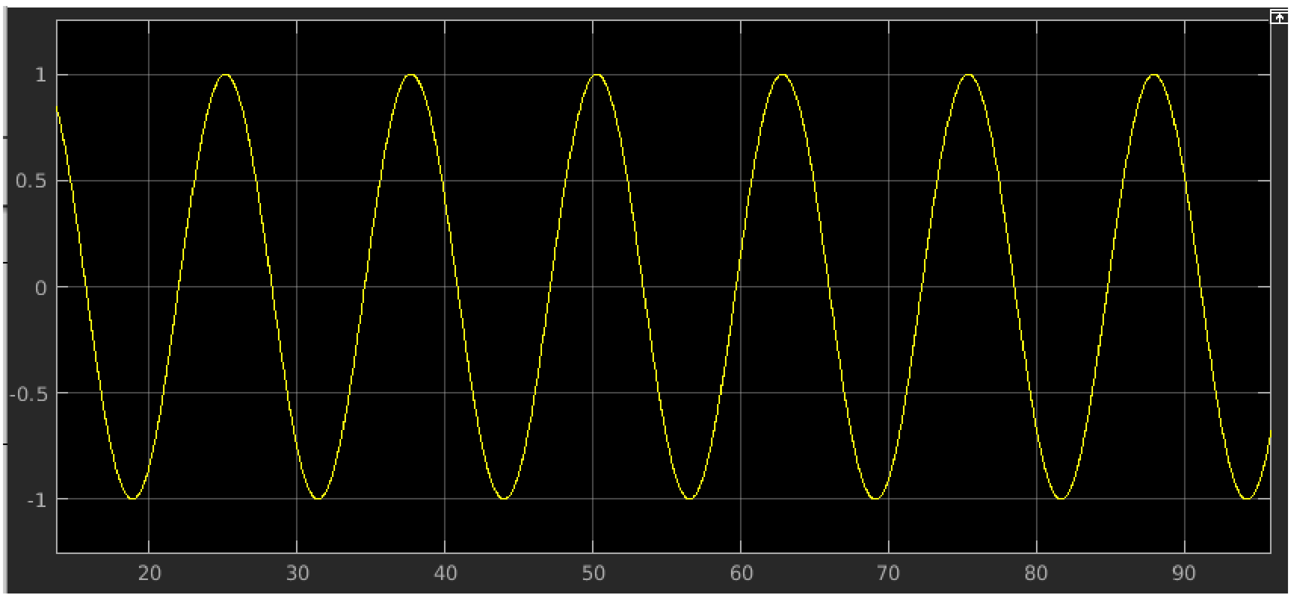


Рисунок 2 – График для генератора

2) Составим структурную схему для генератора. Разделим генератор на две части:

$$u_1 = \cos(-0.5t)$$
$$\dot{u}_1 = -(-0.5)\sin(-0.5t)$$
$$\ddot{u}_1 = -(-0.5)^2\cos(-0.5t)$$
$$\ddot{u}_1 = -w^2 u_1$$

$$u_2 = e^{-0.5t}$$
$$\dot{u}_2 = -0.5e^{-0.5t} = -\frac{1}{2}u_2$$

Получим следующую структурную схему:

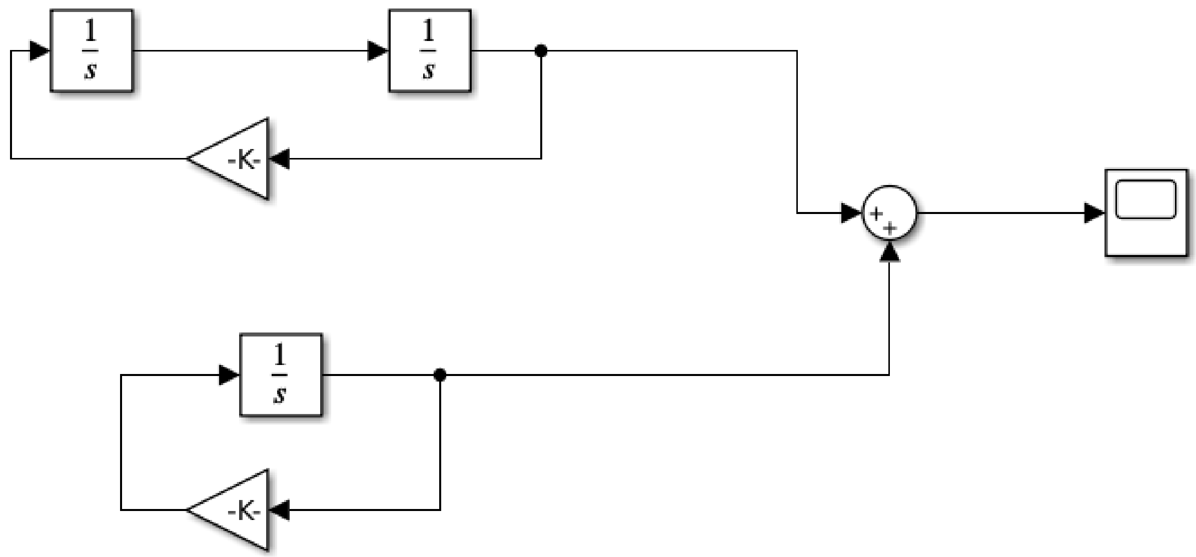


Рисунок 3 – Структурная схема

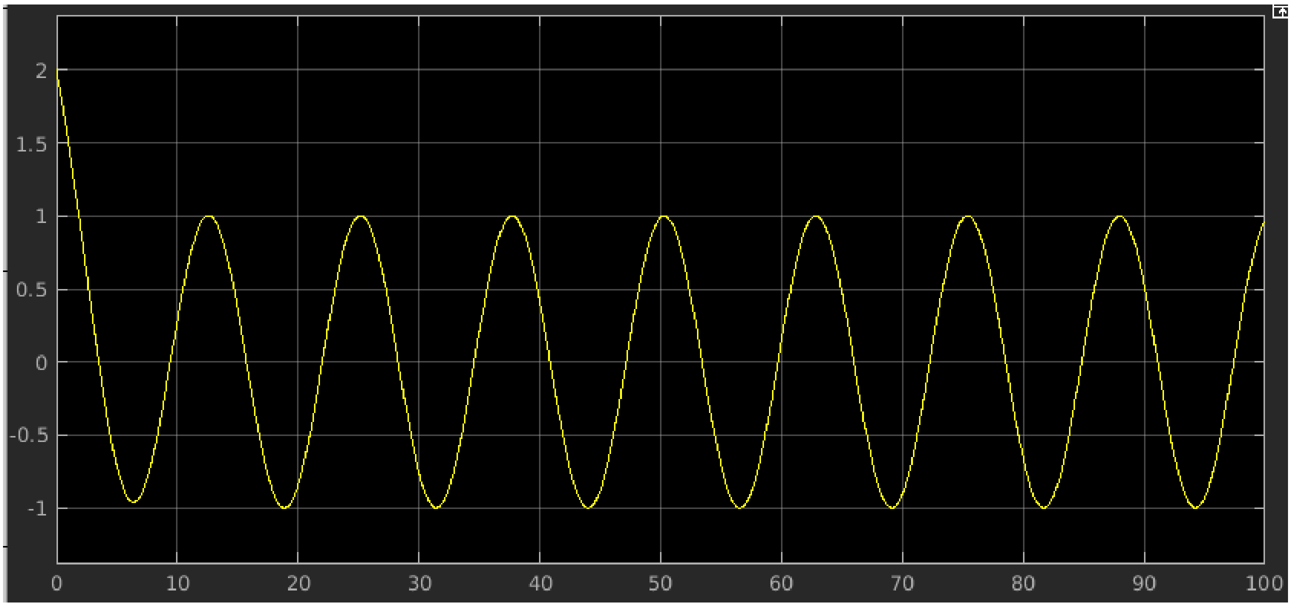


Рисунок 4 – График для структурной схемы

Можно заметить, что графики совпали, схема составлена верно.

3) Приведем в форме Вход-Состояние-Выход:

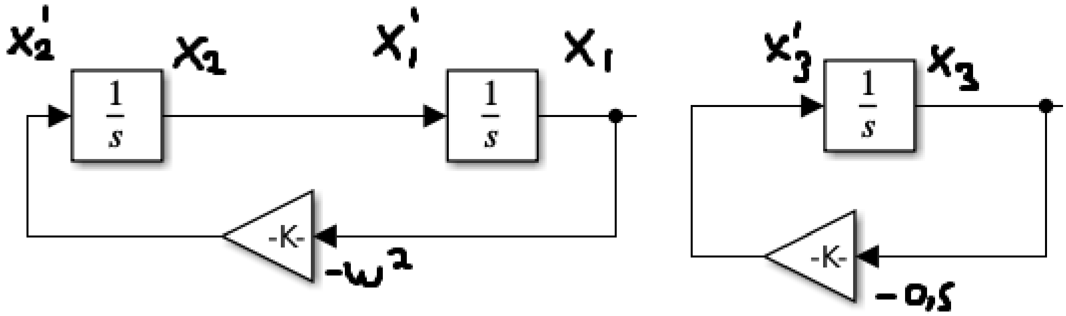


Рисунок 5

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -w^2 & 0 & 0 \\ 0 & 0 & -0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} u(t) = \begin{bmatrix} 0 & 1 & 0 \\ -(-0.5^2) & 0 & 0 \\ 0 & 0 & -0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

С помощью блока State-Space можно реализовать в Simulink:

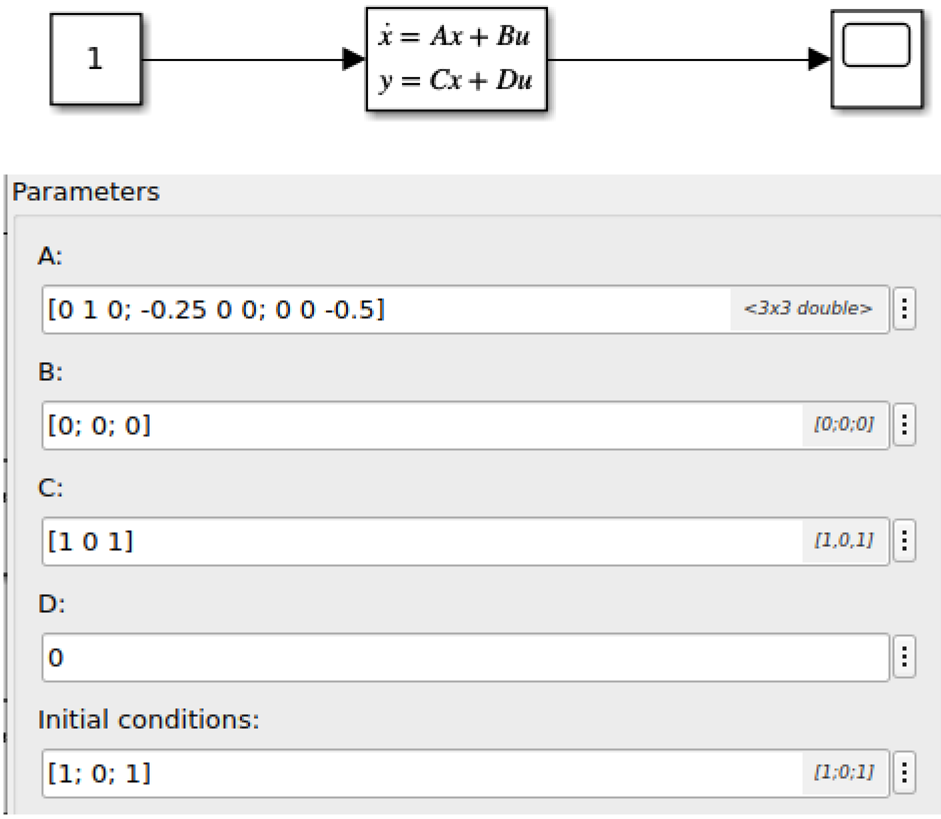


Рисунок 6

4) Дискретизируем генератор с частотой дискретизации 5, 30, 100 Гц.

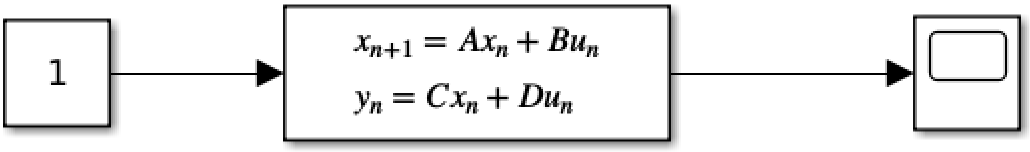
$$x[k+1] = A_d x[k] + B_d u[k], \quad y[k] = C_d x[k] + D_d u[k],$$

$$A_d = e^{A\Delta t}, \quad B_d = A^{-1}(A_d - I_n)B, \quad C_d = C, \quad D_d = D.$$

Получить новые матрицы можно с помощью следующего скрипта:

```
A1 = expm(A*dt1);
A2 = expm(A*dt2);
A3 = expm(A*dt3);

B1 = inv(A) * (A1 - eye(size(A))) * B;
B2 = inv(A) * (A2 - eye(size(A))) * B;
B3 = inv(A) * (A3 - eye(size(A))) * B;
```

Для реализации используется блок Discrete State-Space:



Получим следующие графики для непрерывного и дискретного вида (графики в сравнении с непрерывном приведены в **приложении 1**):

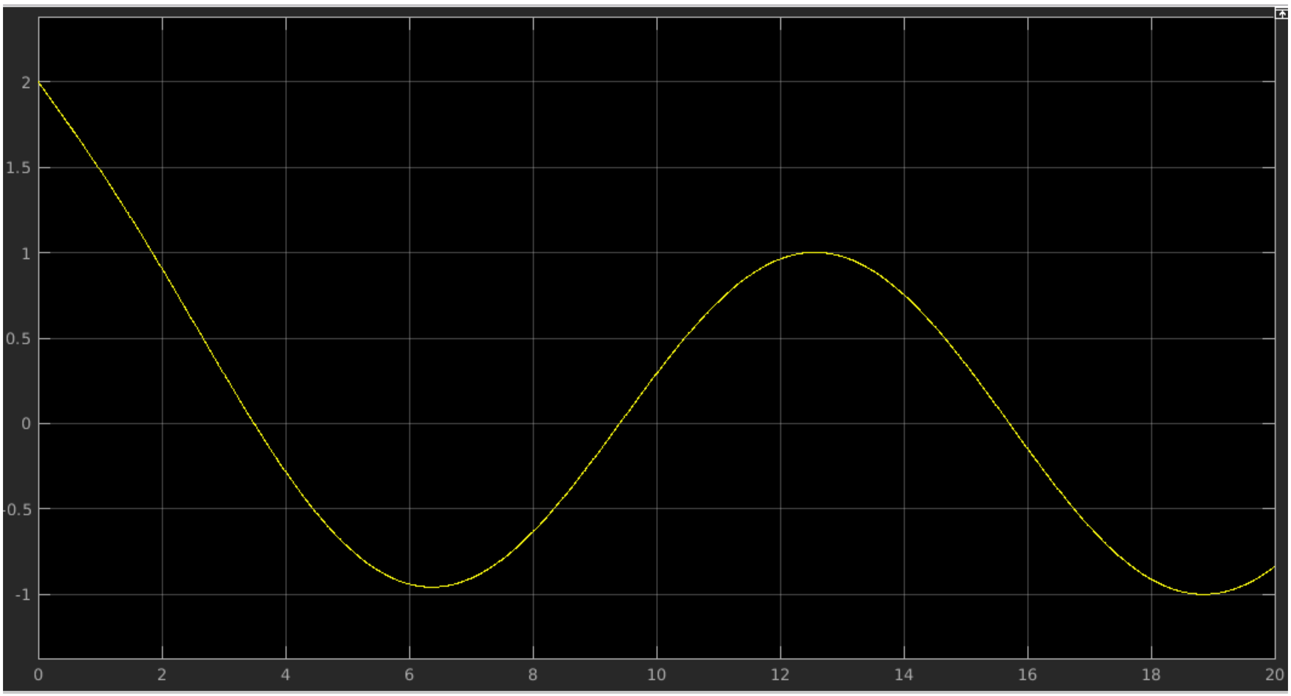


Рисунок 7 – Для непрерывной системы

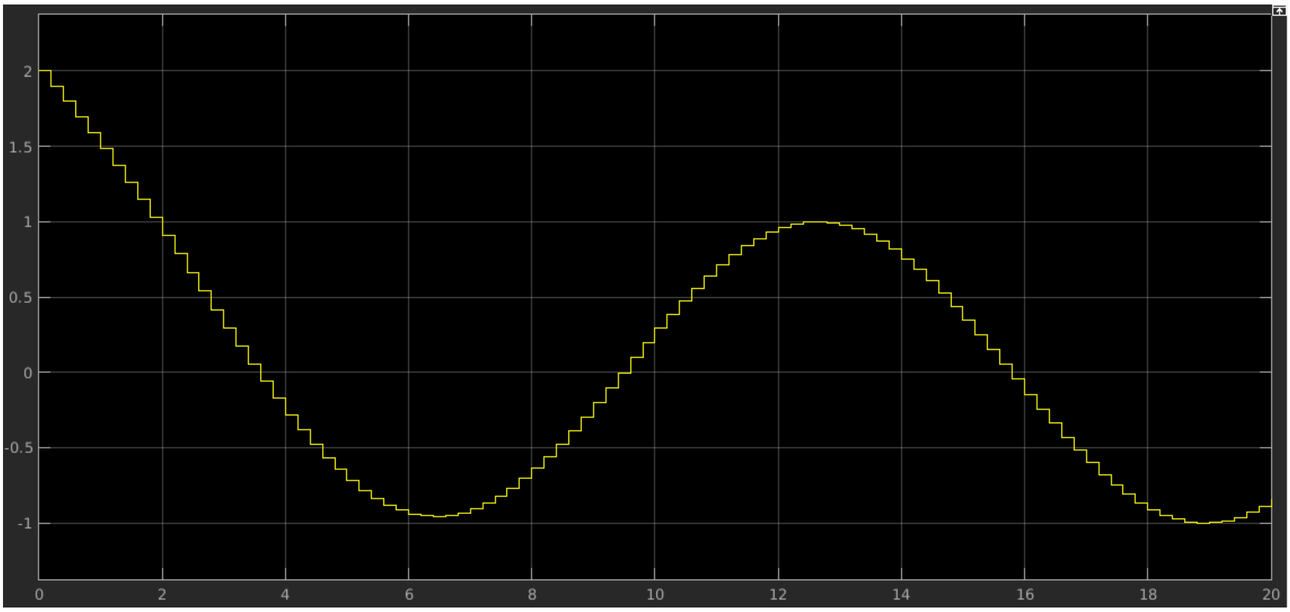


Рисунок 8 – Дискретный вид для частоты 5 Гц

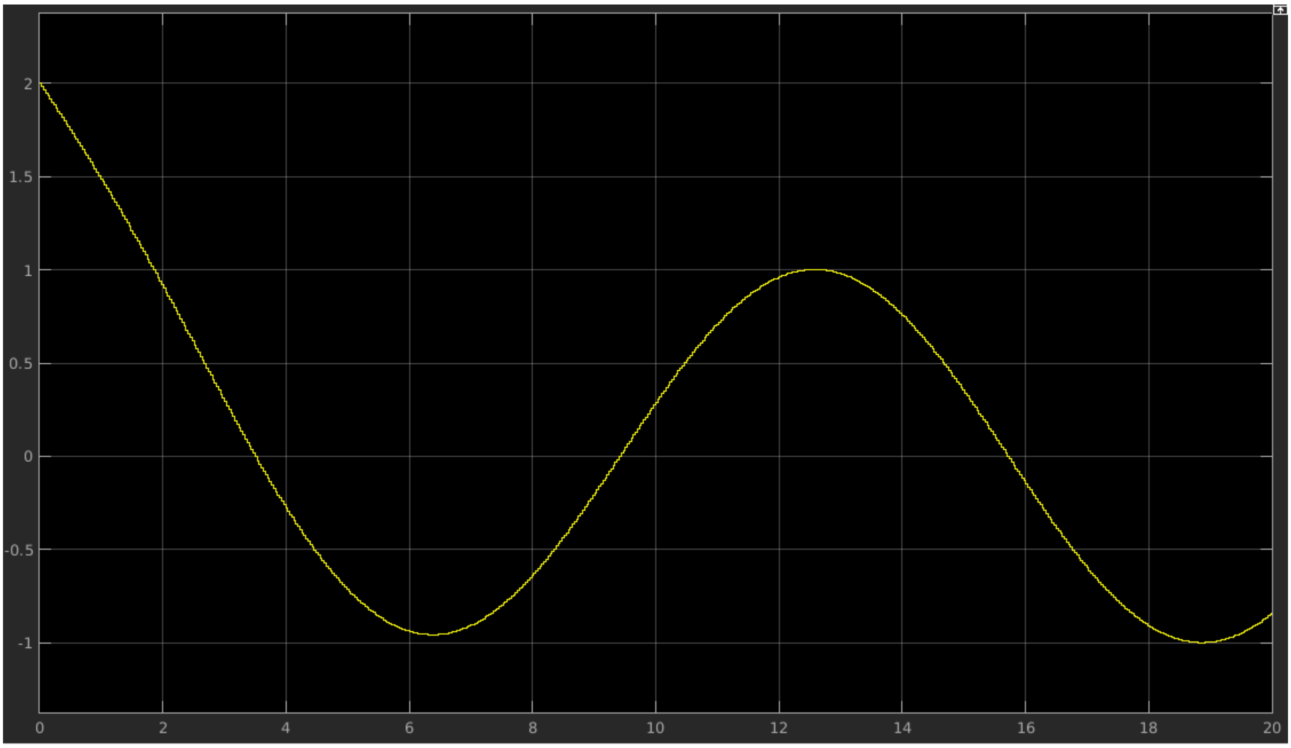


Рисунок 9 – Дискретный вид для частоты 30 Гц

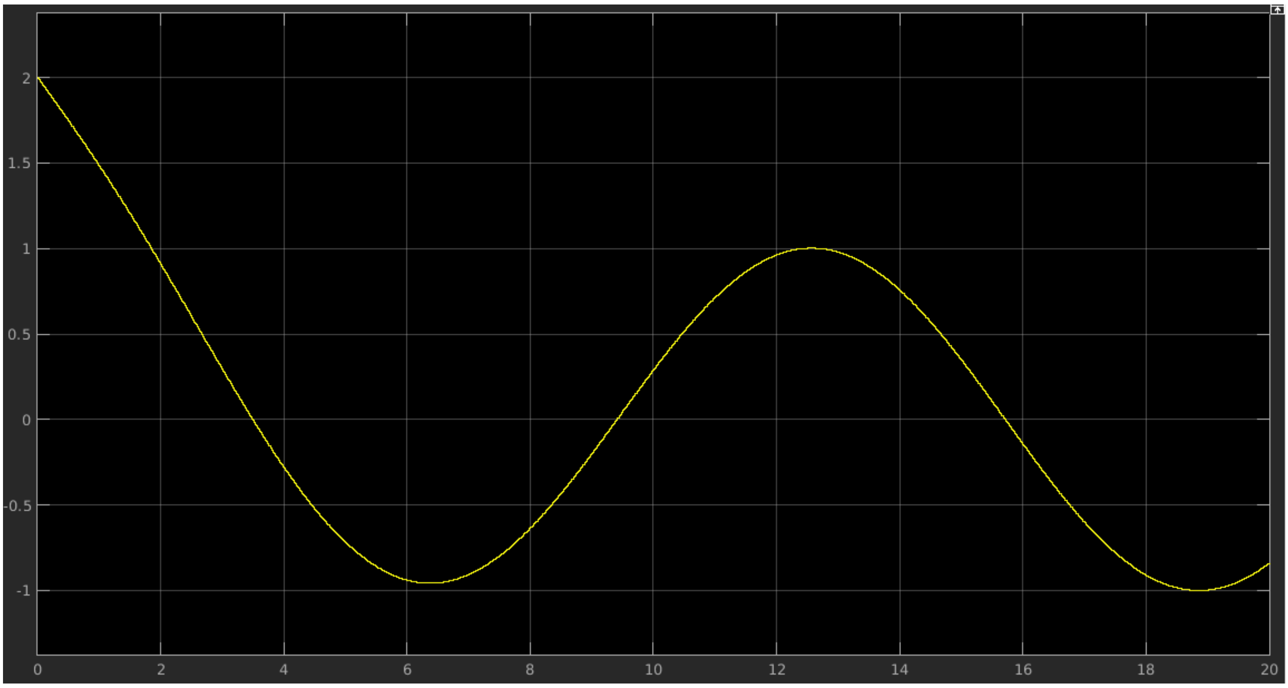


Рисунок 10 – Дискретный вид для частоты 30 Гц

Несложно заметить, что с увеличением частоты уменьшается шаг дискретизации, что делает график более гладким.

5) Имеем следующую передаточную функцию:

$$\frac{y(t)}{u(t)} = \frac{s^3+4s^2+4s+2}{s^3+4s^2+4s+1}$$

Получим вид Вход-Состояние-Выход:

$$\frac{y(s)}{x_1(s)} \cdot \frac{x_1(s)}{u(s)} = \frac{s^3+4s^2+4s+2}{1} \cdot \frac{1}{s^3+4s^2+4s+1}$$

$$y(s) = x_1[s^3+4s^2+4s+2]$$

$$y = \dddot{x}_1 + 4\ddot{x}_1 + 4\dot{x}_1 + 48x_1;\ x_2 = \dot{x}_1,\ x_3 = \dot{x}_2 = \ddot{x}_1$$

$$y = \dot{x}_3 + 4x_3 + 4x_2 + 2x_1;$$

$$u = \dot{x}_3 + 4x_3 + 4x_2 + 1x_1;$$

$$\dot{x}_3 = u - 4x_3 - 4x_2 - x_1;$$

$$y = u - 4x_3 - 4x_2 - x_1 + 4x_3 + 4x_2 + 2x_1 = u + x_1$$

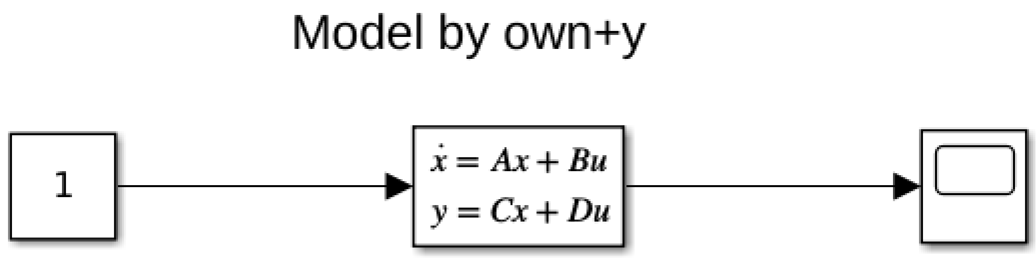
$$\dot{x}_3 = u - 4x_3 - 4x_2 - x_1;$$

$$y = u + x_1$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & -4 & -4 \end{bmatrix}_{A_{3x3}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}_{B_{3x1}} u$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}_{C_{1x3}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + [1]_D u$$

С помощью блока State-Space в Simulink реализуем данную схему:



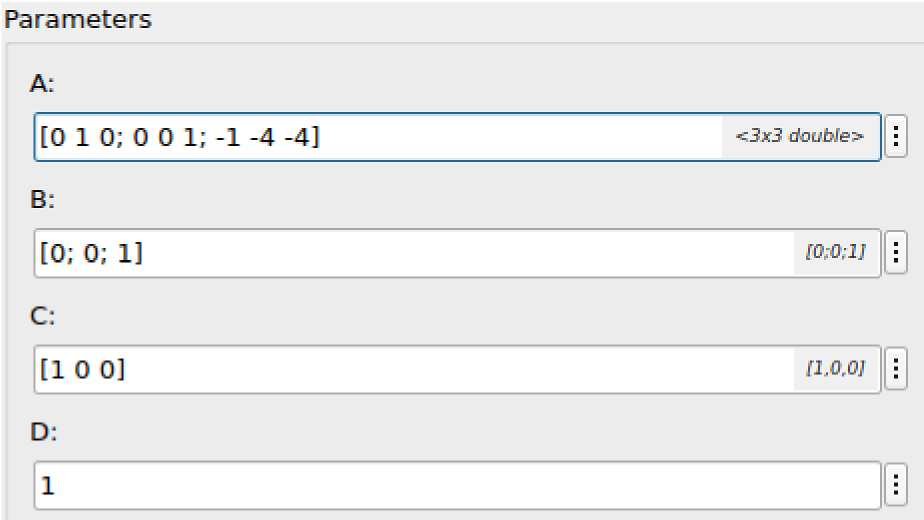


Рисунок 11 – Модель в виде ВСВ

Полученный вид - управляемая каноническая форма. Для сравнения, получим ее с помощью скрипта на matlab, а также получим наблюдаемую каноническую форму:

```
num = [1 4 4 2];
den = [1 4 4 1];

Gp = tf(num, den);

% observable form
GpssObs = canon(Gp,'companion');
GpssObsA = GpssObs.A;
GpssObsB = GpssObs.B;
GpssObsC = GpssObs.C;
GpssObsD = GpssObs.D;

% controll form
GpssConA = GpssObsA.';
GpssConB = GpssObsC.';
GpssConC = GpssObsB.';
GpssConD = GpssObsD;

% dicrete controll form
GpssConA1 = expm(GpssConA*dt1);
GpssConA2 = expm(GpssConA*dt2);
GpssConA3 = expm(GpssConA*dt3);
```

Рисунок 12

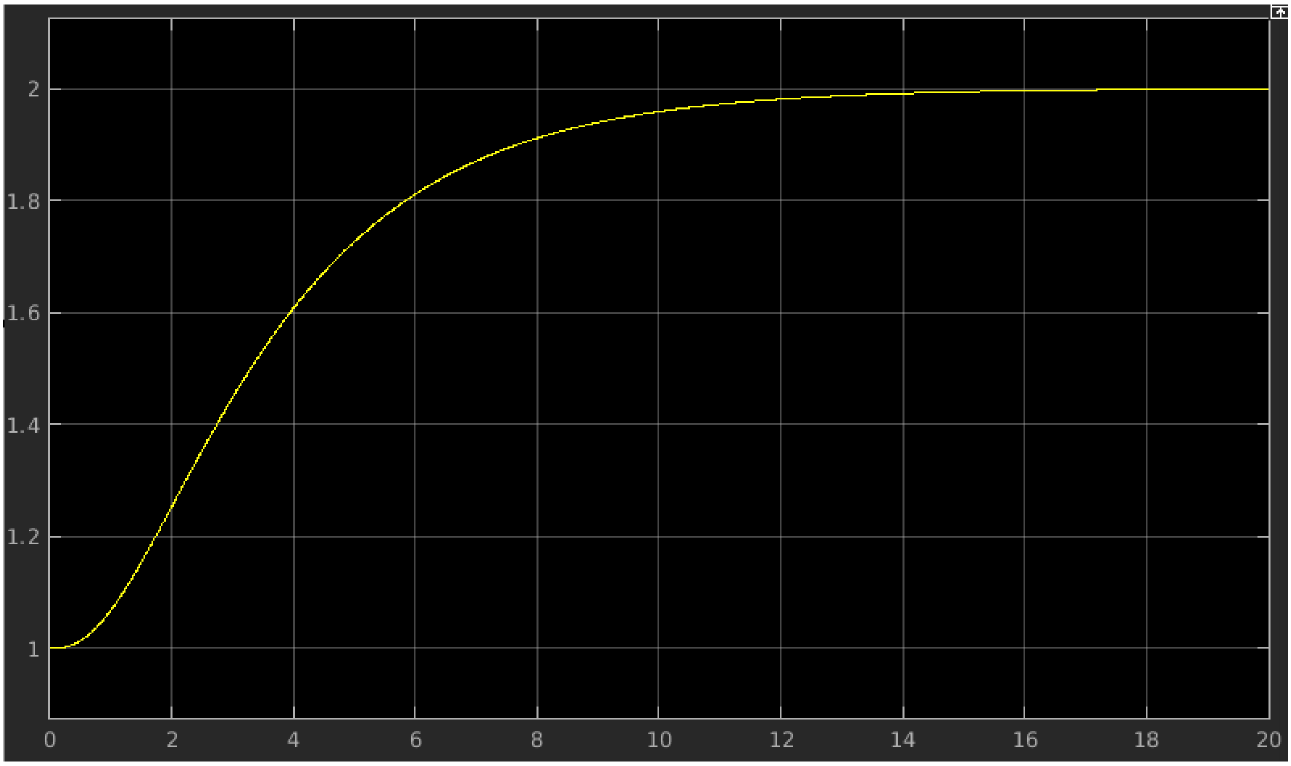


Рисунок 13 – ВСВ в управляемой канонической форме

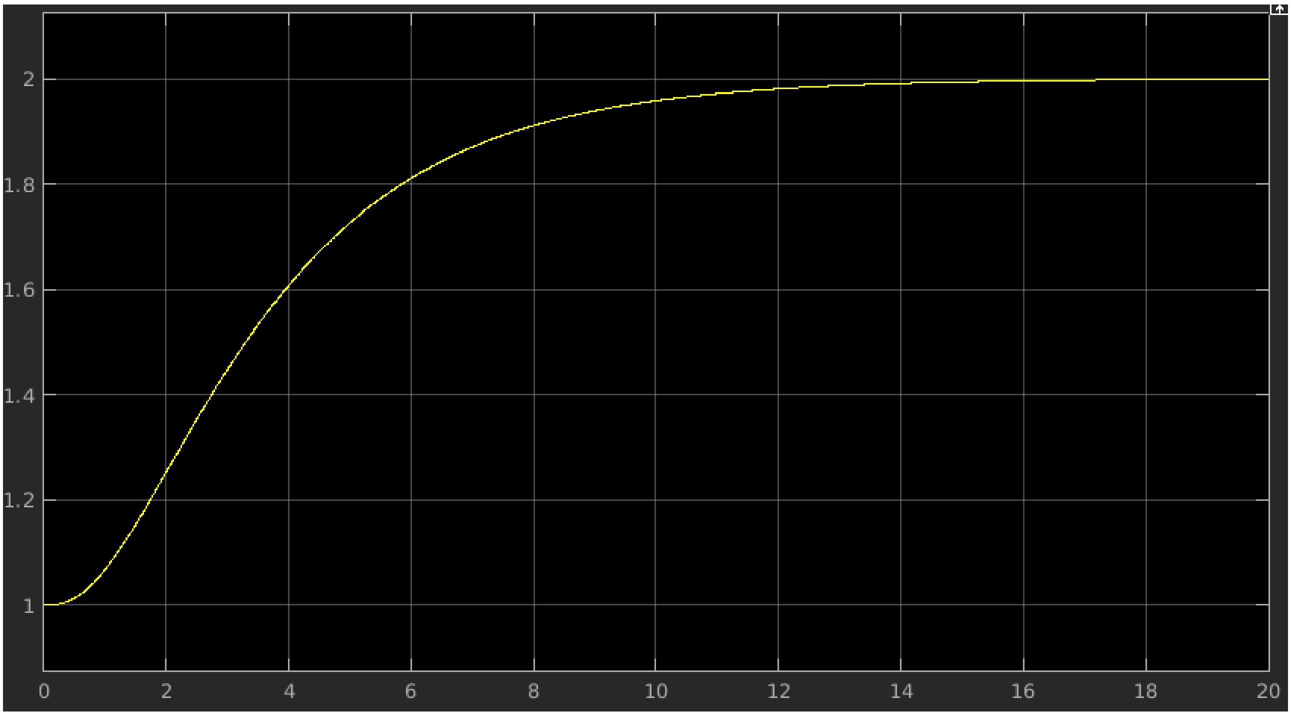


Рисунок 14 – ВСВ в наблюдаемой канонической форме

Видим, что графики совпадают. Также скрипт дает аналогичные матрицы для управляемой канонической формы, полученные вручную.

6) На основе модели ВСВ составим структурную схему:

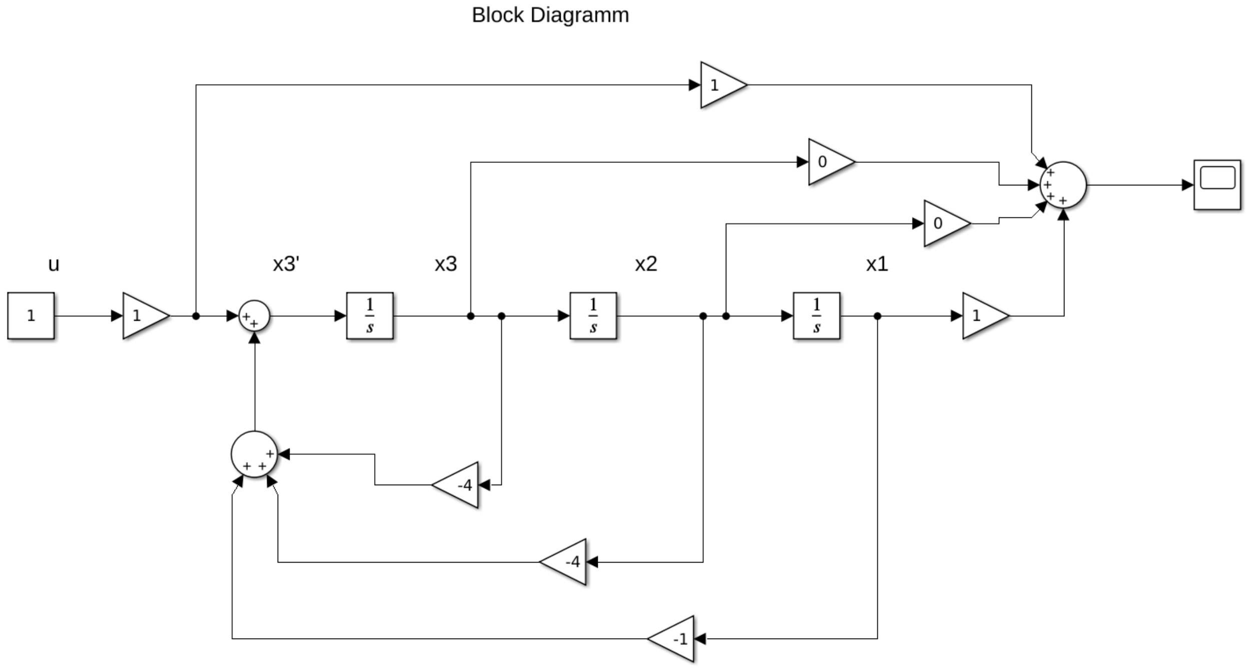


Рисунок 15 – Структурная схема объекта управления

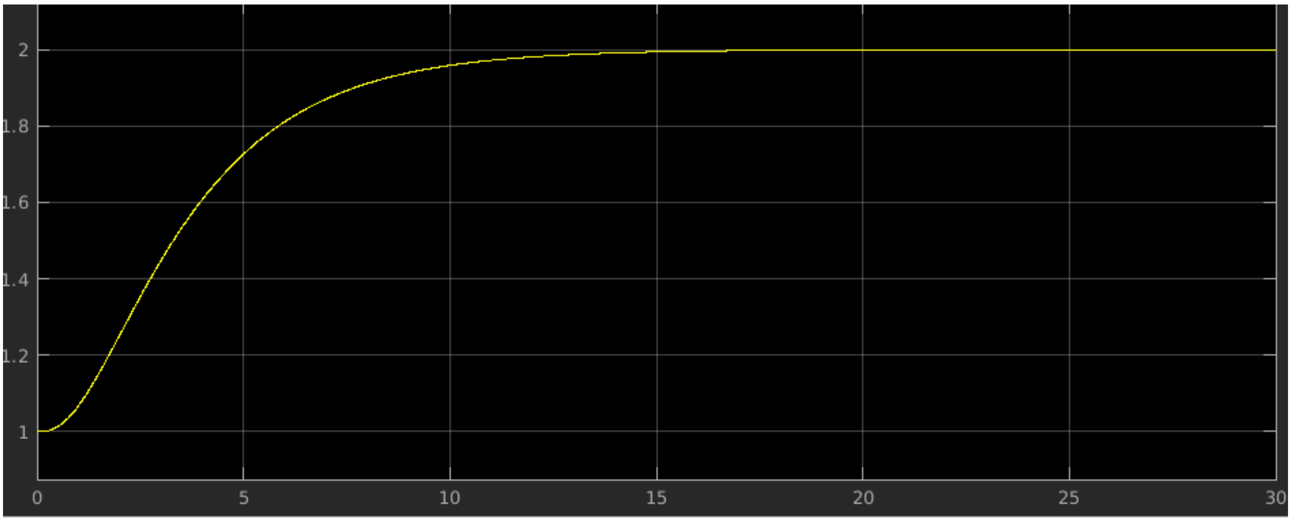


Рисунок 16 – График для структурной схемы

7) Дискретизация проводится по такому же принципу, как и для генератора. Получим следующие графики для частот 5, 30, 100 Гц.

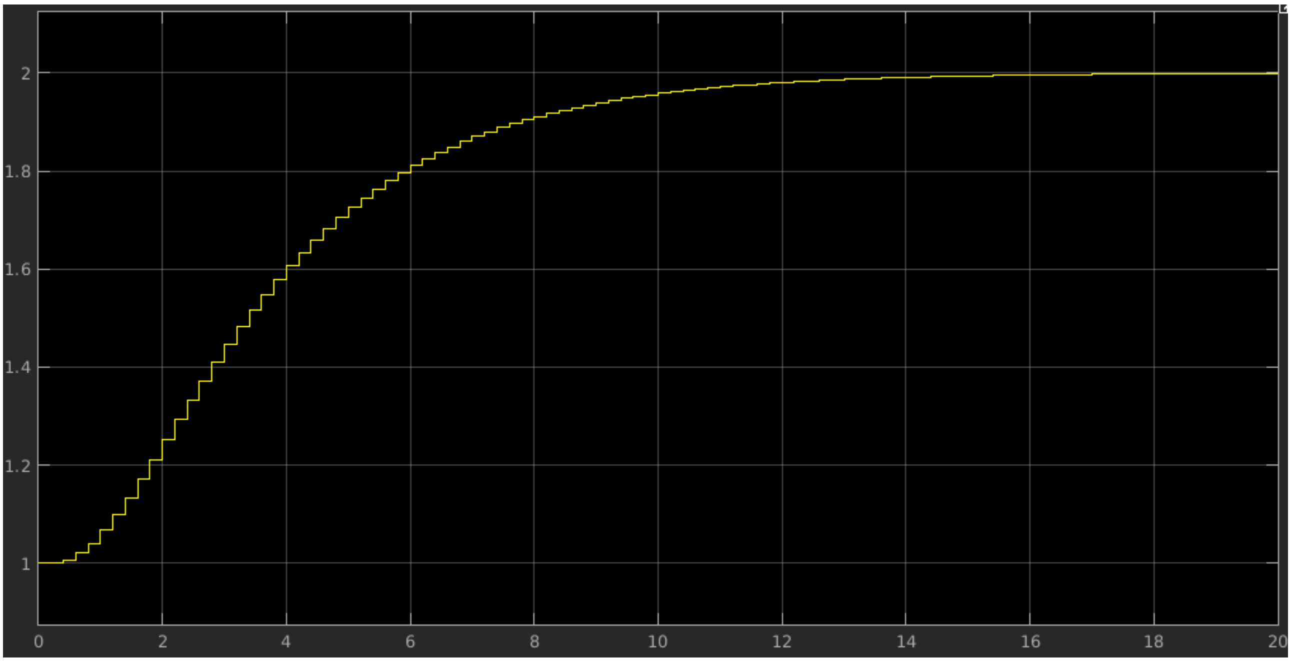


Рисунок 17 – Дискретная модель для частоты 5 Гц

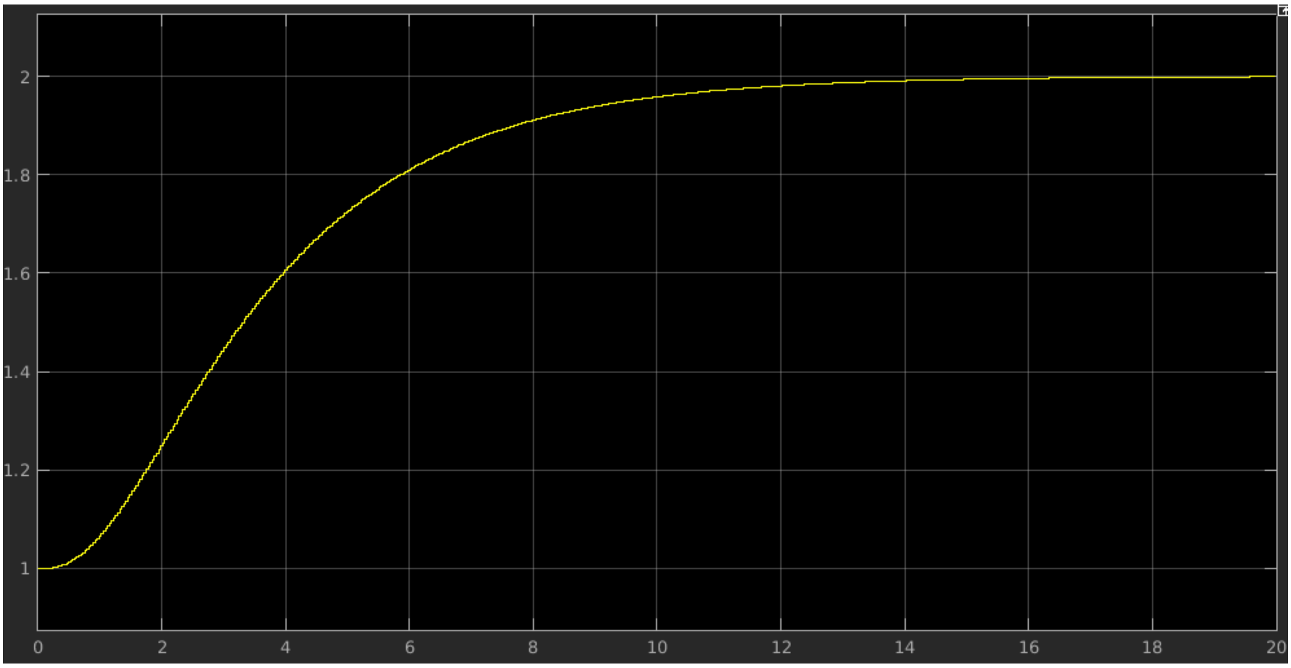


Рисунок 18 – Дискретная модель для частоты 30 Гц

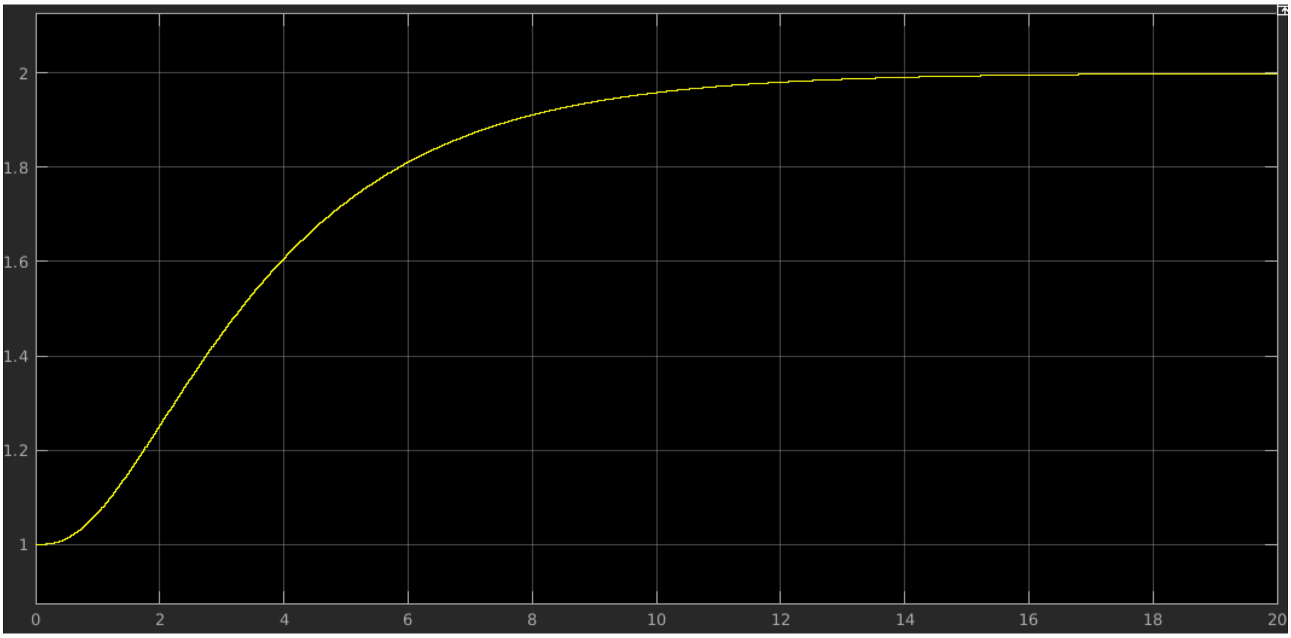


Рисунок 19 – Дискретная модель для частоты 100 Гц

Аналогично, с увеличением частоты уменьшается шаг, и график "сглаживается".

8) Программно реализуем класс интегратора. В качестве метода аппроксимации используется метод трапеции:

lintegrator.h

```
#include <QObject>

class LIntegrator : public QObject
{
    Q_OBJECT
public:
    LIntegrator(double state, QObject *parent = nullptr);
    double update(double inputVal, double dt);
    double getState();

private:
    double state_;
    double prevState_;
};
```

lintegrator.cpp

```
#include "lintegrator.h"

LIntegrator::LIntegrator(double state, QObject *parent)
    : QObject(parent), state_(state), prevState_(0.0)
{}

double LIntegrator::update(double inputVal, double dt)
{
    this->state_ = state_ + (prevState_ + inputVal) * dt / 2.0;
// y[k-1] + (u[k] + u[k+1]) * dt/2
// this->state_ = state_ + prevState_ * dt;
    prevState_ = inputVal;
    return state_;
}

double LIntegrator::getState()
{
    return state_;
}
```

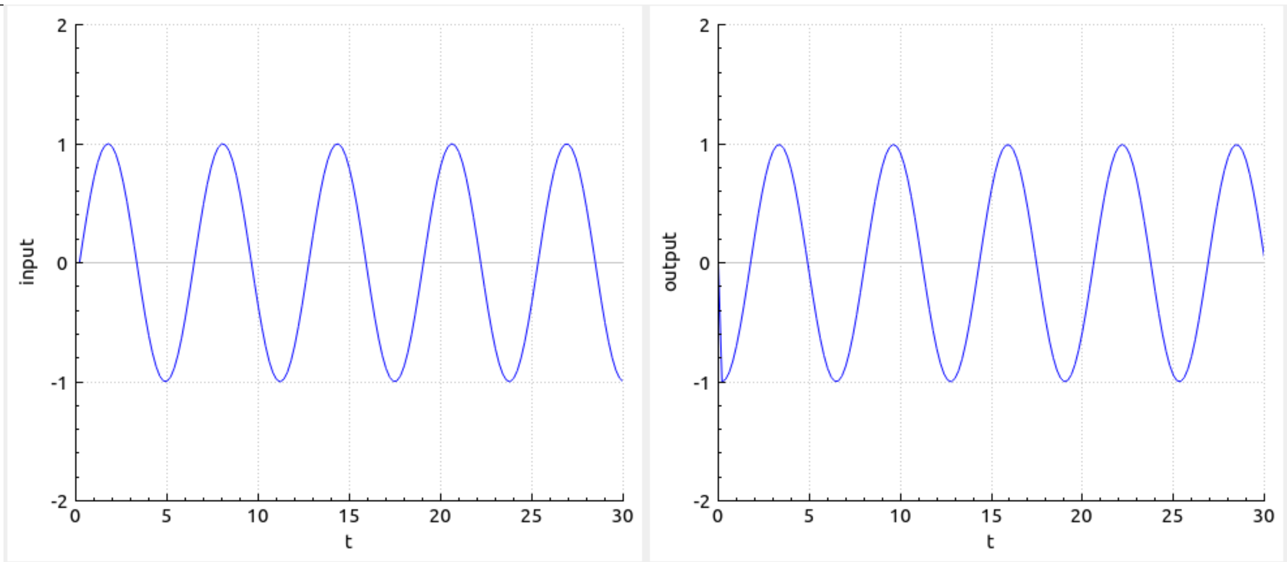


Рисунок 19 – Слева синусоида, справа - результат интегратора

9) Программно реализуем структурную схему генератора. Получим следующий класс:

```
class LIntegrator;
class Gain;

class GeneratorException{};

class Generator : public QObject
{
    Q_OBJECT
public:
    Generator(std::vector<double>& states, std::vector<double>& gains, QObject
*parent = nullptr);
    double update(double inputVal, double dt);

private:
    void InitGenerator(double state1, double state2, double state3, double
gain1, double gain2);
    LIntegrator* integrator_1;
    LIntegrator* integrator_2;
    LIntegrator* integrator_3;

    Gain* gain_1;
    Gain* gain_2;
};
```

```
#include "generator.h"
#include "lintegrator.h"
#include "blocks/gain/gain.h"

Generator::Generator(std::vector<double> &states, std::vector<double> &gains,
QObject *parent)
    : QObject(parent),
      integrator_1(nullptr), integrator_2(nullptr), integrator_3(nullptr),
      gain_1(nullptr), gain_2(nullptr){
    if (states.size() < 3 || gains.size() < 2) throw GeneratorException();
    InitGenerator(states[0], states[1], states[2], gains[0], gains[1]);
}

double Generator::update(double inputVal, double dt)
{
    double currState2 = this->integrator_2->getState();

    double resState2 = integrator_2->update(
                integrator_1->update(gain_1->update(currState2), dt),
                dt);
    double currState3 = integrator_3->getState();
    double resState3 = integrator_3->update(gain_2->update(currState3), dt);
    return resState2 + resState3;
}

void Generator::InitGenerator(double state1, double state2, double state3,
double gain1, double gain2)
{
    integrator_1 = new LIntegrator(state1);
    integrator_2 = new LIntegrator(state2);
    integrator_3 = new LIntegrator(state3);
    gain_1 = new Gain(gain1);
    gain_2 = new Gain(gain2);
}
```

Для правильной реализации, необходимо запоминать текущее состояние интеграторов, передавая данные значения в качестве входа на умножитель (gain).

Получим следующий график для реализованного генератора:

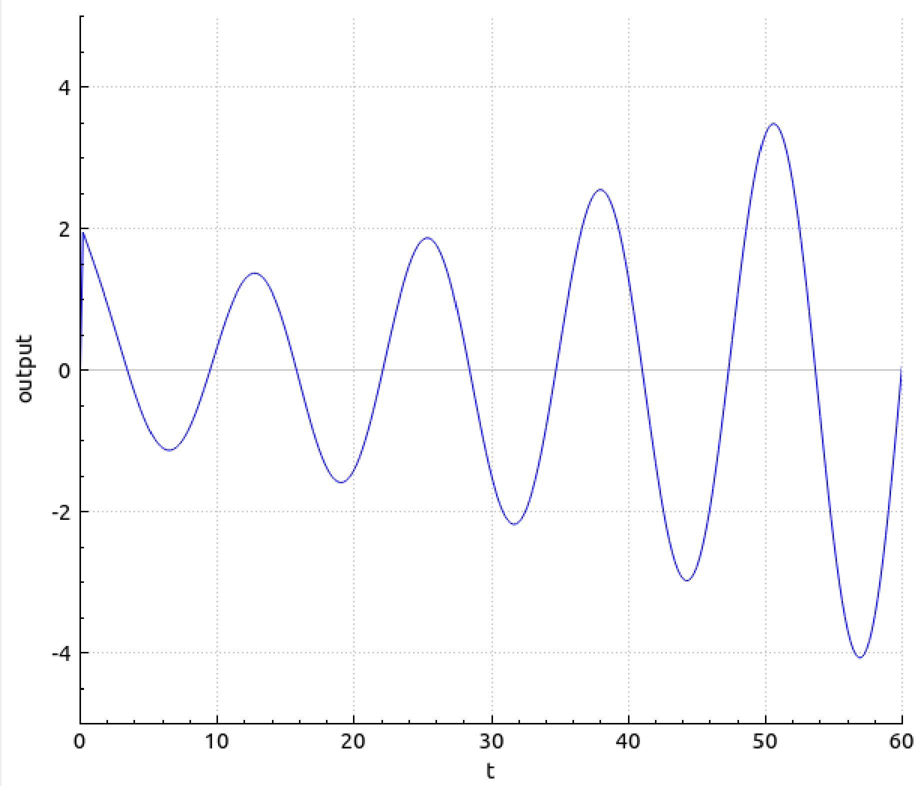


Рисунок 20 – График для генератора

Можно заметить, что со временем идет расхождение. Метод трапеций, примененный в интеграторе, неидеальный и имеет ошибку. Кроме того, интегратор, реализованный через структурную схему, накапливает ошибку. Ошибка от интегратора попадает на блок gain, после чего, идет на вход другого интегратора и т.д.

Для сравнения, ниже приведен график для метода прямоугольников. Можно увидеть, что данный метод обладает большей ошибкой.

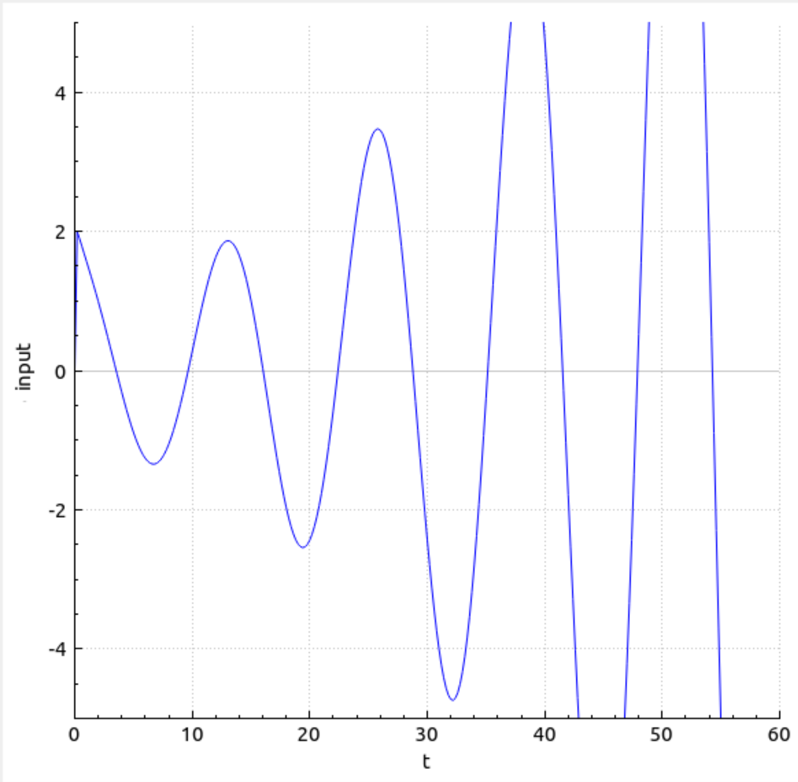


Рисунок 21 – Для метода прямоугольников

Расхождение структурной схемы можно также получить в системе Simulink, установив в интеграторах метод Эйлера:

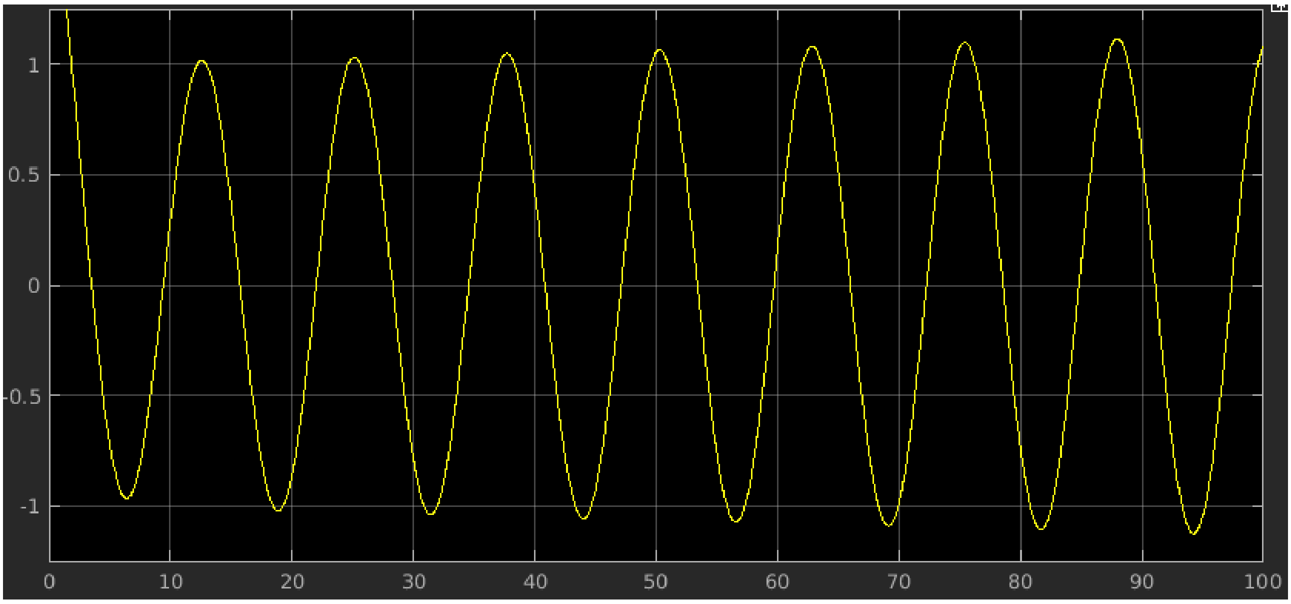


Рисунок 22 – График при использовании метода Эйлера

```
>> eig(A)

ans =

   0.0000 + 0.5000i
   0.0000 - 0.5000i
  -0.5000 + 0.0000i
```

Дискретный генератор реализован с помощью класса, описанного ниже (вместе с дискретным объектом управления). По рисунку ниже видно, что дискретный генератор обладает устойчивостью, в отличие от непрерывного.

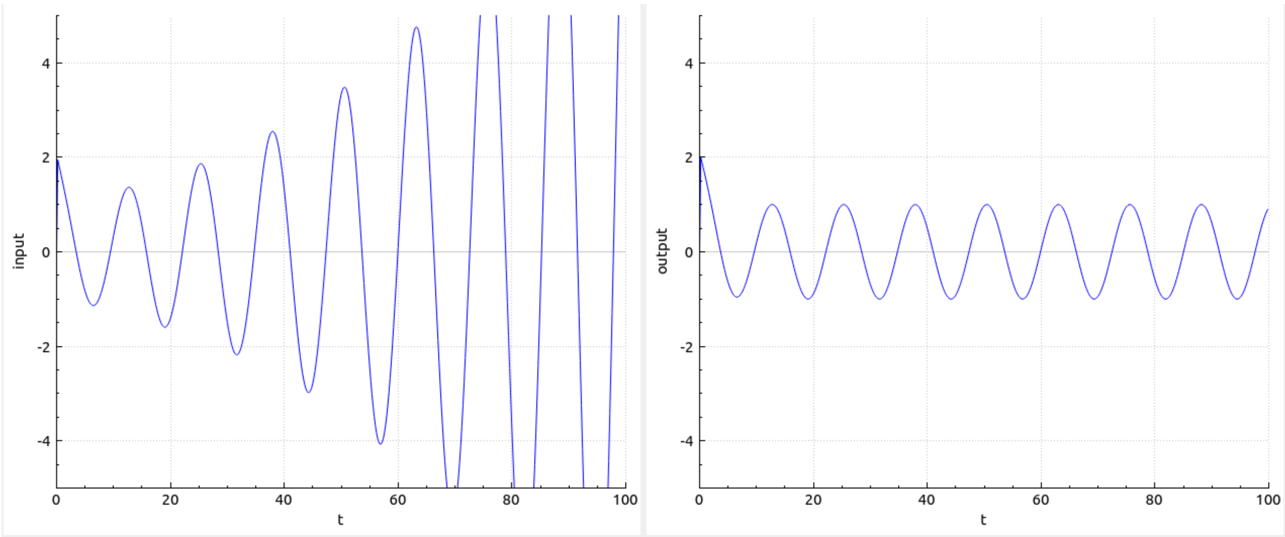


Рисунок 23 – Сравнение непрерывного (метод трапеций) и дискретного генераторов

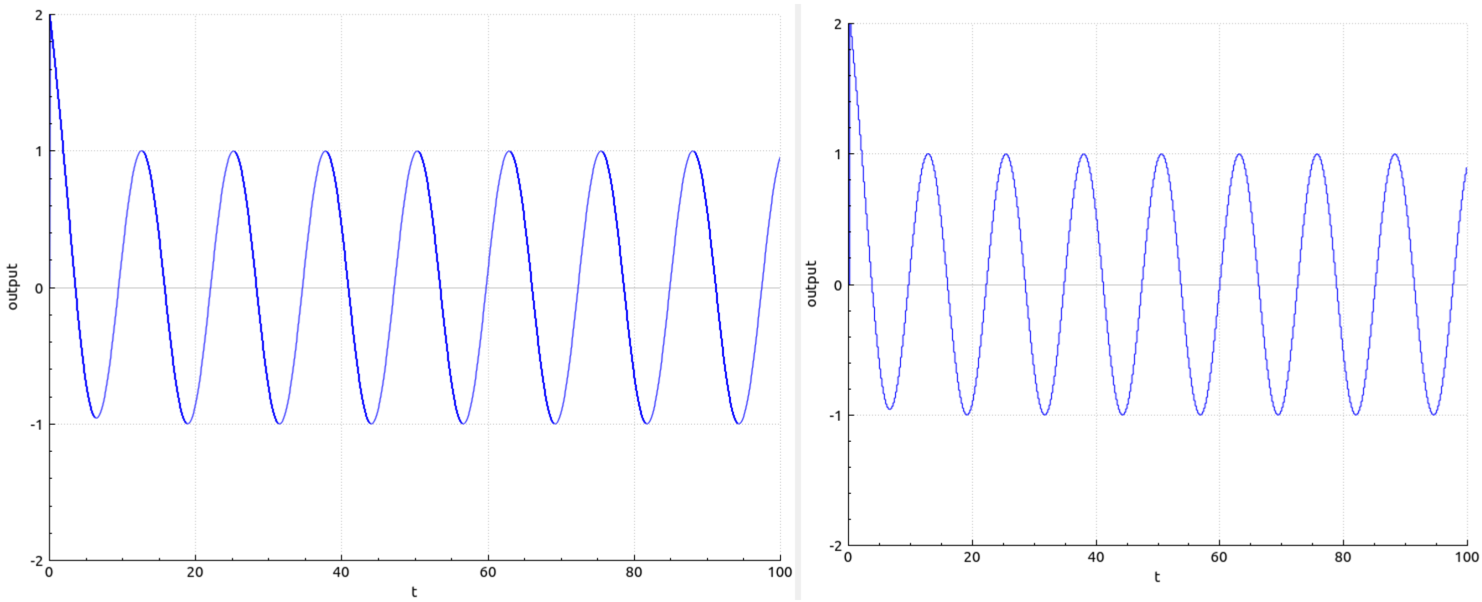
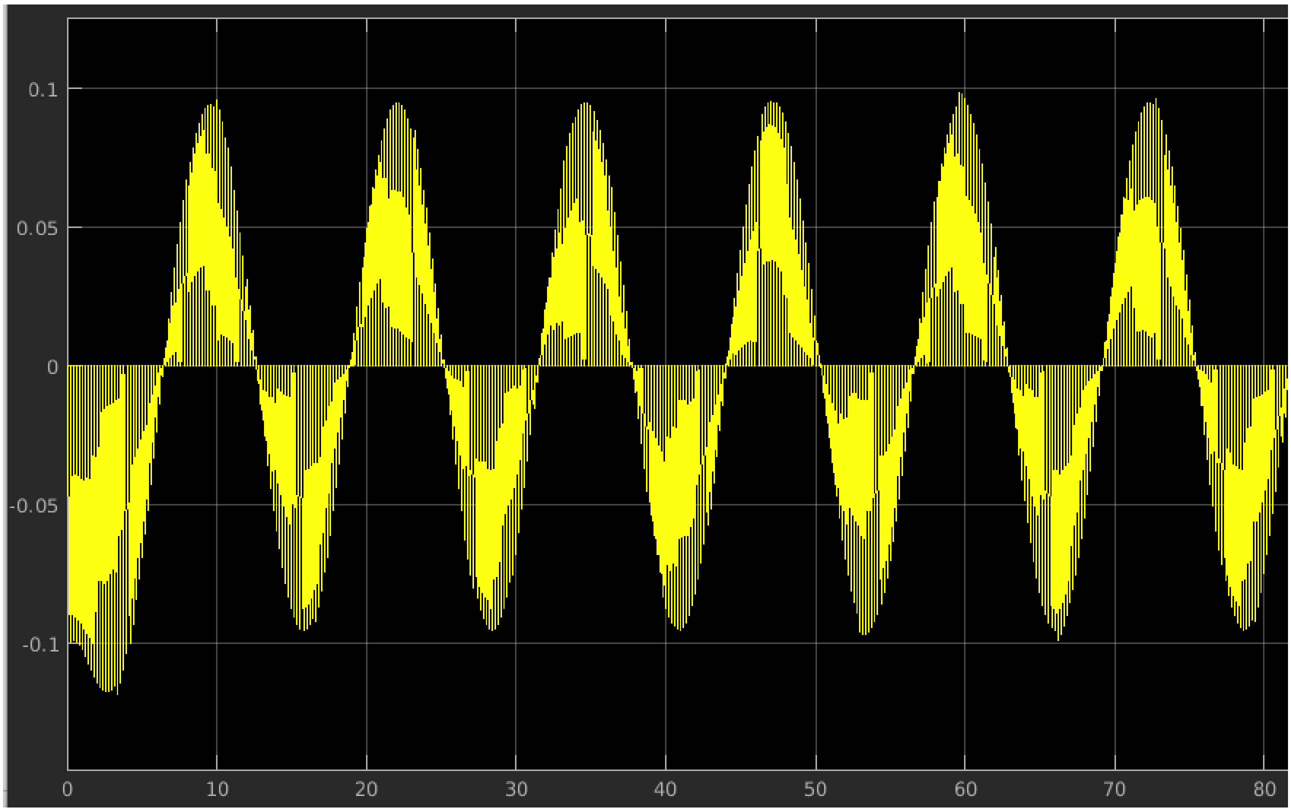
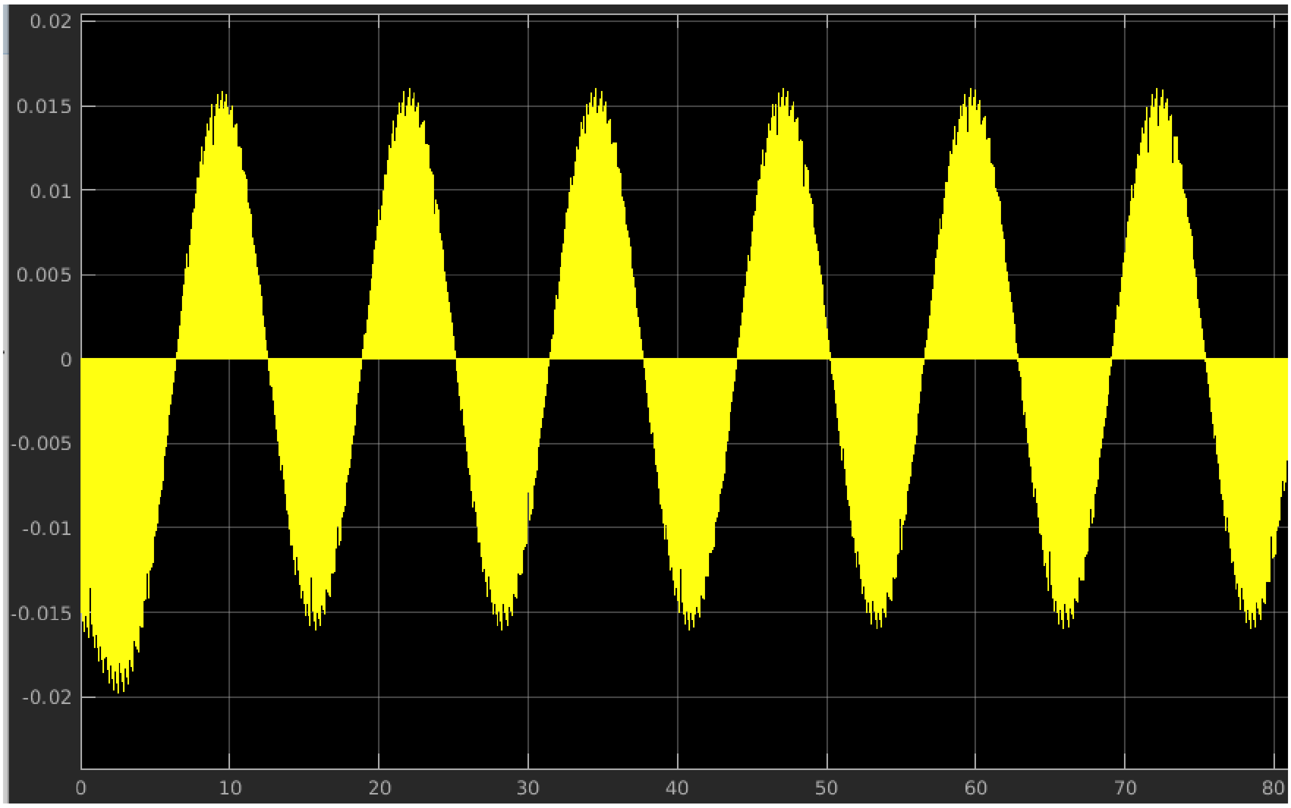


Рисунок 24 – Слева для частоты 30 Гц, справа для частоты 5 Гц

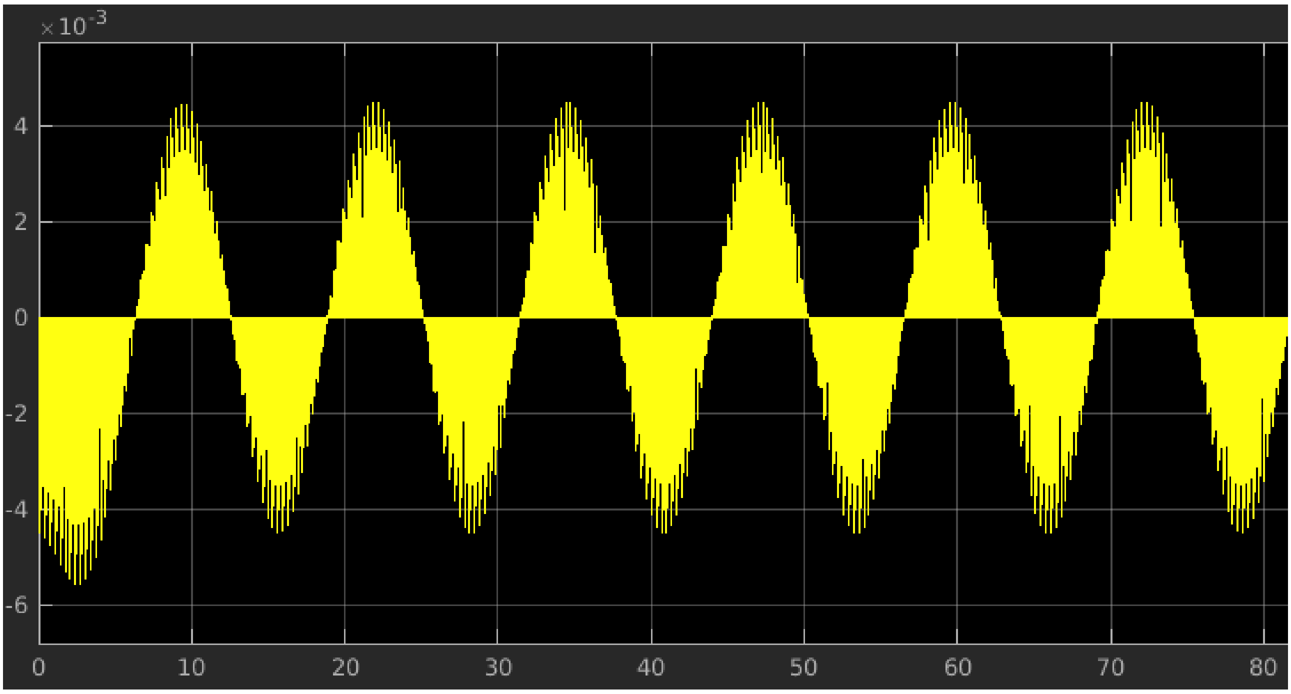
Частота дискретизации позволяет улучшить точность приближения, что можно увидеть, если рассмотреть графики разницы между значением, полученным по формуле (блок Matlab function) и реализованной дискретной формой:



Для частоты 5 Гц

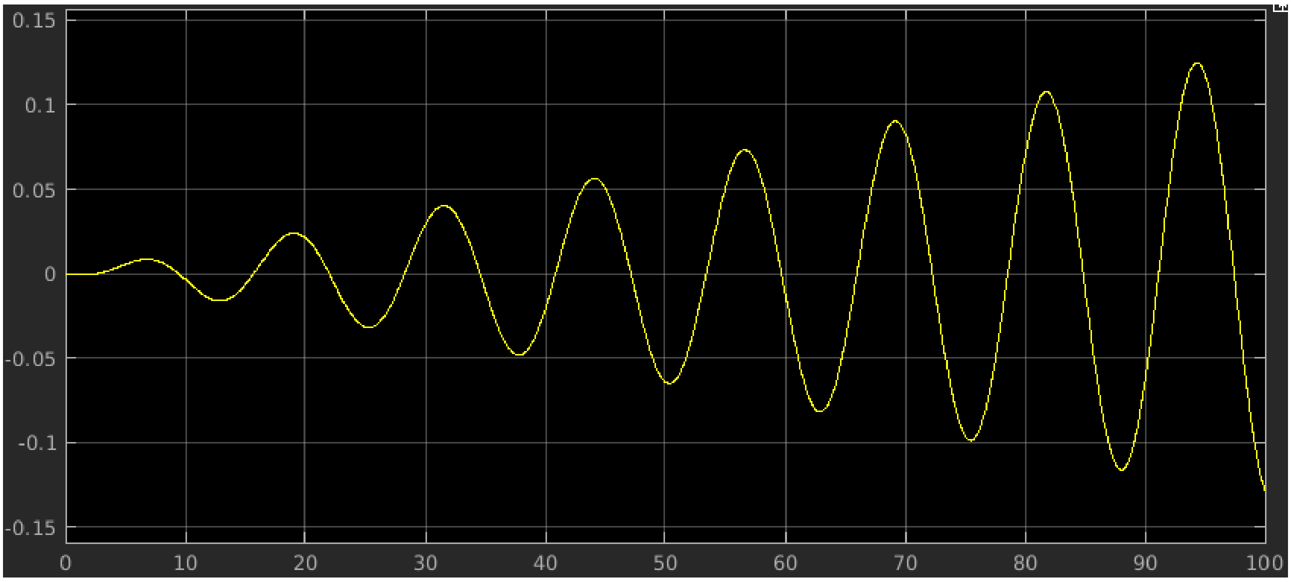


Для частоты 30 Гц



Для частоты 100 Гц

Можно также рассмотреть поведение ошибки, при установке метода Эйлера для интеграторов:



Видно, что непрерывная модель расходится.

10) Программно реализуем объект управления. Получим следующий класс, представленный ниже. Данный класс принимает переменное число интеграторов и умножителей.

```
#include <QVector>
#include <QPair>

class LIntegrator;
class Gain;

class ControlModelError{};

class ControlModel
{
public:
    ControlModel(QVector<double>& states, QVector<QPair<double, double>>& gains,
QPair<double, double>& sigGain);
    double update(double inputVal, double dt);

private:
    QVector<LIntegrator*> integrators_;
    QVector<QPair<Gain*,Gain*>> gains_;
    QPair<Gain*,Gain*> inputSignalGain_;
};
```

```
#include "controlmodel.h"
#include "lintegrator.h"
#include "blocks/gain/gain.h"

/** setting integrators from left to right
    gains set from first = by init state to second = result state
*/
ControlModel::ControlModel(QVector<double>& states, QVector<QPair<double,
double>>& gains,
                           QPair<double, double>& sigGain)
{
    if (states.size() == 0 || states.size() != gains.size())
        throw ControlModelError();

    for (auto state : states){
        integrators_.push_back(new LIntegrator(state));
    }

    for (auto currgain: gains){
        gains_.push_back(QPair<Gain*, Gain*>(
                             new Gain(currgain.first), new
Gain(currgain.second)));
    }

    this->inputSignalGain_.first = new Gain(sigGain.first);
    this->inputSignalGain_.second = new Gain(sigGain.second);
}

double ControlModel::update(double inputVal, double dt)
{
    double currStates[integrators_.size()];

    for (int i = integrators_.size()-1; i >= 0; i--)
    {
        currStates[i] = gains_[i].first->update(integrators_[i]->getState());
```

```
    }

    double currInput = inputSignalGain_.first->update(inputVal);

    /** init for result summ  */
    double currInputIntegrator = currInput;
    for (int i = 0; i < integrators_.size(); i++){
        currInputIntegrator += currStates[i];
    }


    for (int i = 0; i < integrators_.size(); i++){
        currInputIntegrator = integrators_[i]->update(currInputIntegrator, dt);
    }

    /** get result summ */
    currInput = inputSignalGain_.second->update(currInput);
    for (int i = 0; i < integrators_.size(); i++){
        currInput += gains_[i].second->update(integrators_[i]->getState());
    }

    return currInput;
}
```

Получим следующий график:

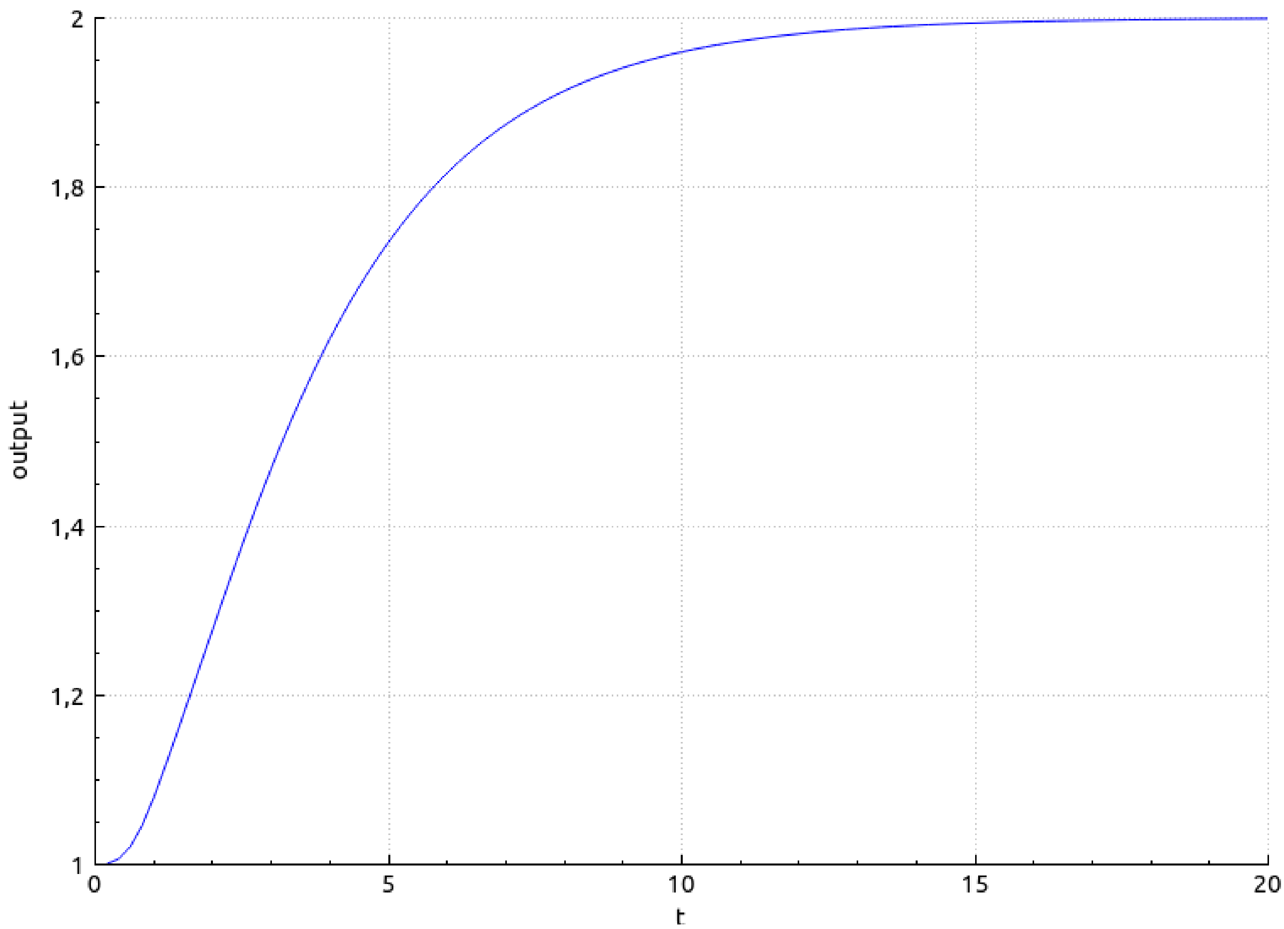


Рисунок 25

11) Программно реализуем дискретные модели объектов управления с помощью реккурентных формул:

$$x[k+1] = A_d x[k] + B_d u[k], \quad y[k] = C_d x[k] + D_d u[k]$$

Был реализован простой класс матрицы:

```
class SimpleMatrix{
public:
    SimpleMatrix(size_t rows, size_t columns,  double initVal=0.0);
    double getElem(size_t row, size_t col);
    void setElem(size_t row, size_t col, double value);
    void getSize(int& row, int& col);
    size_t getRows();
    size_t getColumns();

private:
    QVector<QVector<double>> data_;
    size_t rows_;
    size_t columns_;
};
```

```
class DiscreteControlModelError{};

class DiscreteControlModel
{
public:
    DiscreteControlModel(SimpleMatrix* A, QVector<double> B, QVector<double> C,
double D, QVector<double> x, double y);
    double update(double val, double dt);
    double getState();
    ~DiscreteControlModel();

private:
    SimpleMatrix* A_;
    QVector<double> B_;
    QVector<double> C_;
    double D_;
    QVector<double> xCurr;
    double yCurr;
};
```

```
#include <iostream>

SimpleMatrix::SimpleMatrix(size_t rows, size_t columns, double initVal)
    : rows_(rows), columns_(columns)
{
    QVector<double> currRow = QVector<double>(columns_, initVal);
    for (int i = 0; i < rows_; i++){
        data_.push_back(currRow);
    }
}

double SimpleMatrix::getElem(size_t row, size_t col)
{
    return data_[row][col];
}
```

```cpp
void SimpleMatrix::setElem(size_t row, size_t col, double value)
{
    data_[row][col] = value;
}

void SimpleMatrix::getSize(int &row, int &col)
{
    row = rows_; col = columns_;
}

size_t SimpleMatrix::getRows()
{
    return rows_;
}

size_t SimpleMatrix::getColumns()
{
    return columns_;
}

DiscreteControlModel::DiscreteControlModel(SimpleMatrix* A, QVector<double> B,
QVector<double> C, double D, QVector<double> x, double y)
    : A_(A), B_(B), C_(C), D_(D), xCurr(x), yCurr(y)
{
    if (A_->getColumns() != A_->getRows() ||
            A_->getRows() != xCurr.size() ||
            A_->getRows() != B_.size() ||
            C_.size() != xCurr.size())
        throw DiscreteControlModelError();
}

double DiscreteControlModel::update(double val, double dt)
{
    auto currX = xCurr;
    /** recalc x[k+1] = A*x[k] + B*u(t) */
    for (int cRow = 0; cRow < xCurr.size(); cRow++)
    {
        xCurr[cRow] = B_[cRow] * val;
        for (int cCol = 0; cCol < A_->getColumns(); cCol++)
        {
            xCurr[cRow] += A_->getElem(cRow, cCol) * currX[cCol];
        }
    }

    /** recalc y by x[k] and return */
    auto currY = yCurr;
    yCurr = D_ * val;
    for (int i = 0; i < C_.size(); i++){
        yCurr += C_[i] * currX[i];
    }
    return yCurr;
}

double DiscreteControlModel::getState()
{
    return yCurr;
}

DiscreteControlModel::~DiscreteControlModel()
{
    delete A_;
}
```

Получим следующие графики в сравнении с непрерывным видом объекта управления.

Ступенчатый вид графика можно легко получить следующим способом:

```
#ifdef STEPS_GRAPH
    outputPlot->graph(0)->addData(relativeTime / 1000.0, discreteModel->getState());
#endif
    outputPlot->graph(0)->addData(relativeTime / 1000.0, discreteModel->update(1, dt / 1000.0));
```

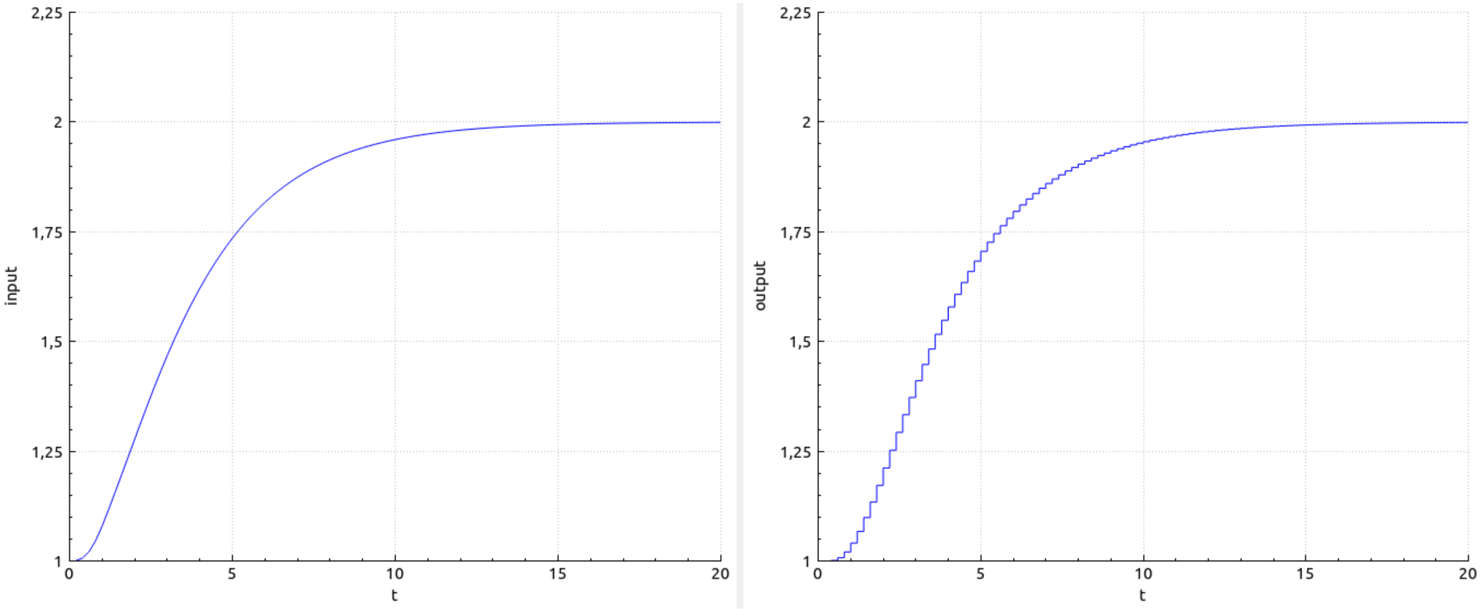


Рисунок 26 - Сравнение с дискретным объектом (5 Гц)

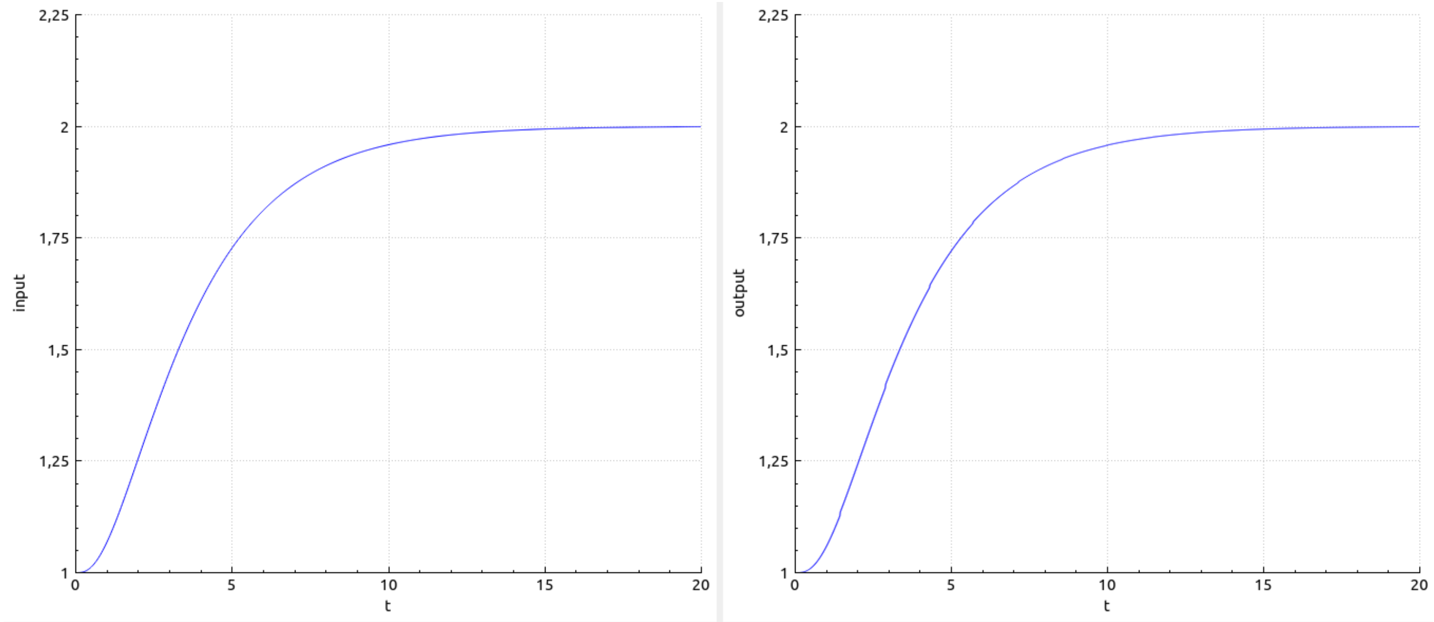


Рисунок 26 - Сравнение с дискретным объектом (30 Гц)

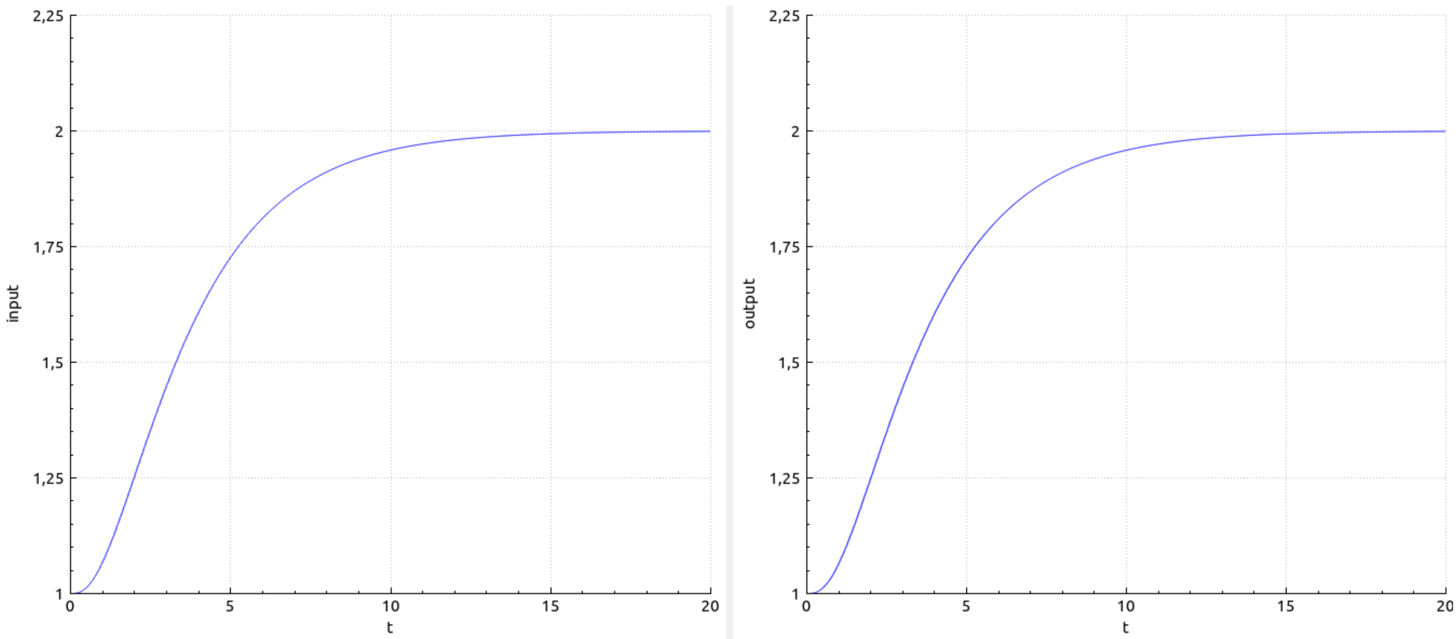


Рисунок 26 - Сравнение с дискретным объектом (100 Гц)

Аналогично можно увидеть, что увеличение частоты дискретизации, которое приводит к уменьшению шага, повышает гладкость графика. Непрерывный вид является устойчивым, это можно проверить, посмотрев на собственные числа матрицы А.

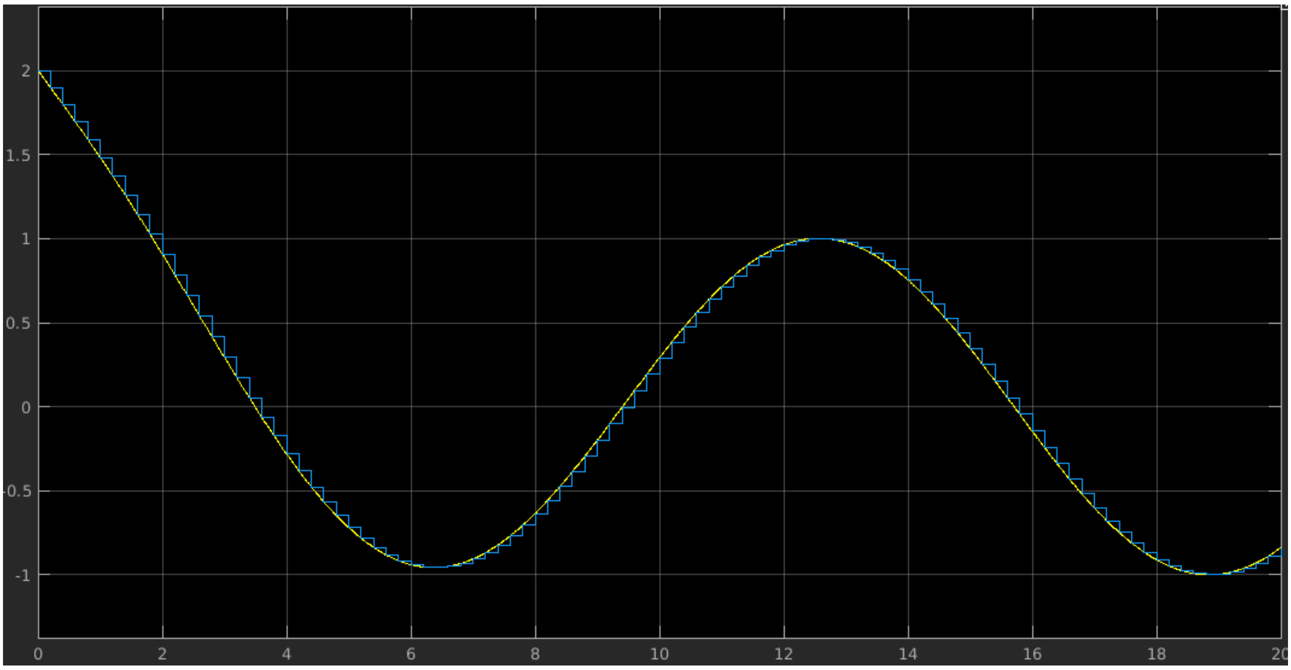
```
% controll form
GpssConA = GpssObsA.';
GpssConB = GpssObsC.';
GpssConC = GpssObsB.';
GpssConD = GpssObsD;
```

```
>> eig(GpssConA)

ans =

   -0.3820
   -1.0000
   -2.6180
```

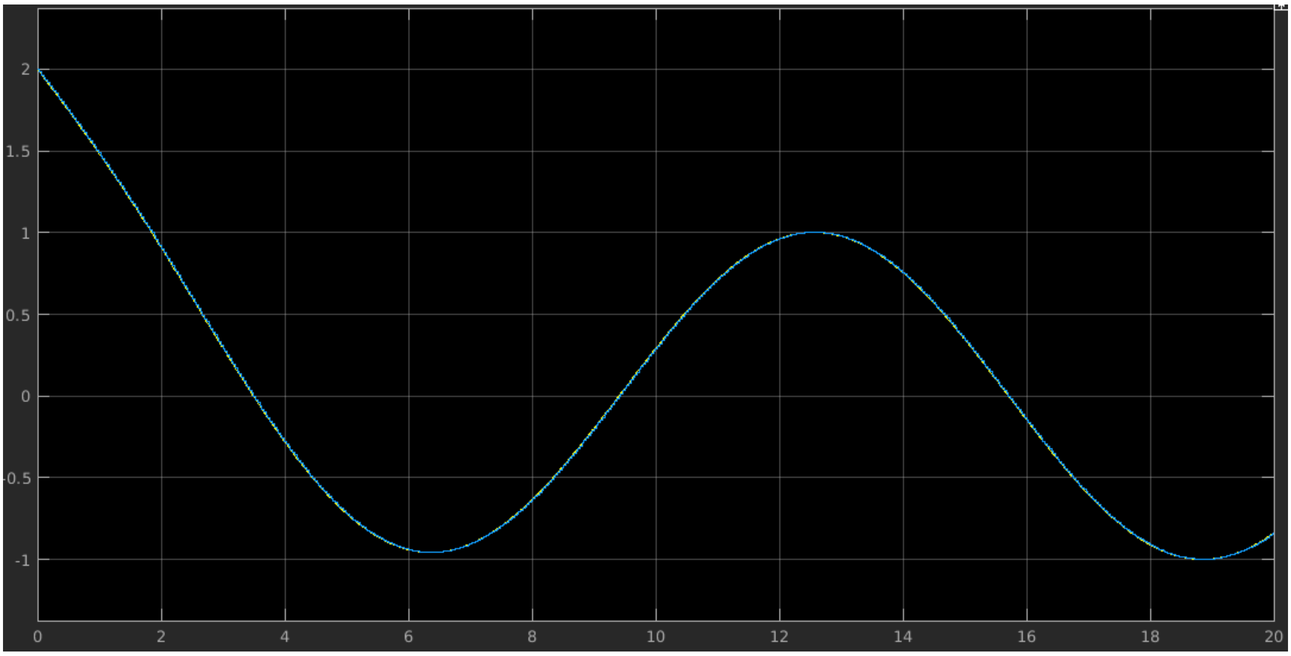
В ходе выполнения работы был реализован генератор в среде Simulink. Была реализована структурная схема для генератора, получена форма Вход-Состояние-Выход. Были рассчитаны матрицы и получены дискретные версии для частот дискретизации 5, 30 и 100 Гц. Для программной реализации в среде Qt Creator был реализован класс интегратора и на его основе реализована непрерывная версия генератора. Было проведено сравнение, в результате которого можно было наблюдать, что с увеличением частоты уменьшается шаг дискретизации, график становится более гладким, а точность апроксимации увеличивается. Т.к. интегратор был реализован с помощью метода трапеций, то при моделировании видно, что непрерывный генератор расходится. Метод трапеций имеет ошибку, которая увеличивается на блоке gain и подается на вход другому интегратору.

В среде Simulink был реализован объект управления. Была получена управляемая каноническая форма, структурная схема и дискретные формы. Аналогично, были реализованы соответствующие классы на языке C++.

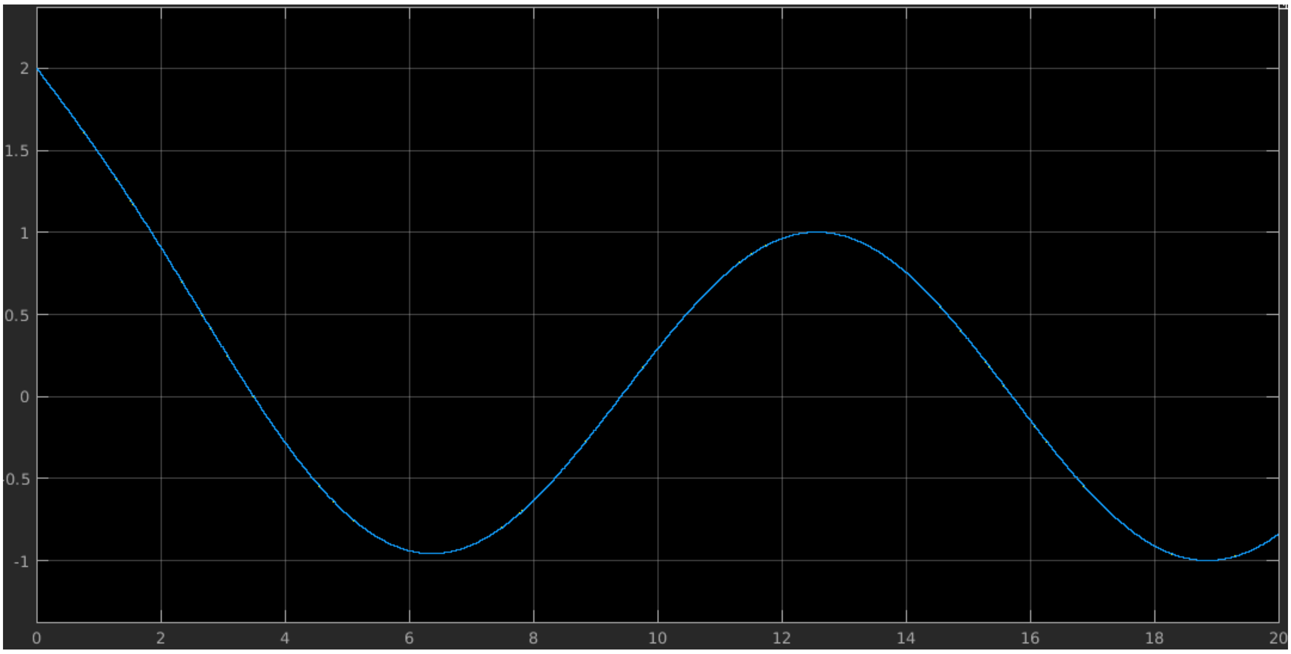
Приложение 1



Генератор, 5 ГЦ и непрерывная



Генератор, 30 ГЦ и непрерывная



Генератор, 100 ГЦ и непрерывная

Приложение 2

Полный код проекта и matlab моделей и скриптов можно посмотреть по ссылке.

https://github.com/IlyaFedorov115/Programming_Control_System_ITMO/tree/Matlab_Part

https://github.com/IlyaFedorov115/Programming_Control_System_ITMO/tree/Qt_part

Код для матриц (как константы).

```
#ifndef CONSTANS_H
#define CONSTANS_H

#include <QVector>
#include "discretecontrolmodel.h"


/** Matrixes for Discrete Generator */

SimpleMatrix* getDGeneratorA1()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.99500417); result->setElem(0, 1, 0.19966683);
result->setElem(0, 2, 0.00000000);
    result->setElem(1, 0, -0.04991671); result->setElem(1, 1, 0.99500417);
result->setElem(1, 2, 0.00000000);
    result->setElem(2, 0, 0.00000000); result->setElem(2, 1, 0.00000000);
result->setElem(2, 2, 0.90483742);
    return result;
}


SimpleMatrix* getDGeneratorA2()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.99986388); result->setElem(0, 1, 0.03299850);
result->setElem(0, 2, 0.00000000);
    result->setElem(1, 0, -0.00824963); result->setElem(1, 1, 0.99986388);
result->setElem(1, 2, 0.00000000);
    result->setElem(2, 0, 0.00000000); result->setElem(2, 1, 0.00000000);
result->setElem(2, 2, 0.98363538);
    return result;
}



SimpleMatrix* getDGeneratorA3()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.99998750); result->setElem(0, 1, 0.00999996);
result->setElem(0, 2, 0.00000000);
    result->setElem(1, 0, -0.00249999); result->setElem(1, 1, 0.99998750);
result->setElem(1, 2, 0.00000000);
    result->setElem(2, 0, 0.00000000); result->setElem(2, 1, 0.00000000);
result->setElem(2, 2, 0.99501248);
    return result;
```

```
}


QVector<double> DGeneratorB {0.0, 0.0, 0.0};
QVector<double> DGeneratorC {1.0, 0.0, 1.0};
double DGeneratorD = 0.0;



/** Matrixes For Discrete Control object */

SimpleMatrix* getAdt1()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.998904057); result->setElem(0, 1, 0.195559240);
result->setElem(0, 2, 0.015385936);
    result->setElem(1, 0, -0.015385936); result->setElem(1, 1, 0.937360312);
result->setElem(1, 2, 0.134015495);
    result->setElem(2, 0, -0.134015495); result->setElem(2, 1, -0.551447918);
result->setElem(2, 2, 0.401298331);
    return result;
}

SimpleMatrix* getAdt2()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.999994204); result->setElem(0, 1, 0.032976769);
result->setElem(0, 2, 0.000521124);
    result->setElem(1, 0, -0.000521124); result->setElem(1, 1, 0.997909707);
result->setElem(1, 2, 0.030892272);
    result->setElem(2, 0, -0.030892272); result->setElem(2, 1, -0.124090211);
result->setElem(2, 2, 0.874340620);
    return result;
}

SimpleMatrix* getAdt3()
{
    SimpleMatrix* result = new SimpleMatrix(3, 3);
    result->setElem(0, 0, 0.999999835); result->setElem(0, 1, 0.009999340);
result->setElem(0, 2, 0.000049338);
    result->setElem(1, 0, -0.000049338); result->setElem(1, 1, 0.999802482);
result->setElem(1, 2, 0.009801986);
    result->setElem(2, 0, -0.009801986); result->setElem(2, 1, -0.039257284);
result->setElem(2, 2, 0.960594536);
    return result;
}


/** For ALL */
QVector<double> GpssConC {1.0, 0.0, 0.0};
double GpssConD = 1.0;

/** dt-1 */


QVector<double> GpssConB1 {0.001095943, 0.015385936, 0.134015495};


/** dt-2 */
QVector<double> GpssConB2 {0.000005796, 0.000521124, 0.030892272};


/** dt-3 */
```

```
QVector<double> GpssConB3 {0.000000165, 0.000049338, 0.009801986};


#endif // CONSTANS_H
```