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» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Отчёт

по лабораторной работе № 5
по дисциплине «Методы оптимизации»

Тема: «Исследование стохастической фильтрации сигналов как задачи двухкритериальной оптимизации с использованием методов прямого пассивного поиска»

Вариант 10

Выполнил: Митрофанов Д.А.,
студент группы ИУ8-33

Проверил: Коннова Н. С.,
доцент каф. ИУ8

г. Москва,
2019 г.

1.1. Цель работы

Изучить основные принципы многокритериальной оптимизации в комбинации с методами случайного и прямого пассивного поиска применительно к задаче фильтрации дискретного сигнала методом взвешенного скользящего среднего.

1.2. Постановка задачи

На интервале $[x_{min}, x_{max}]$ *задан сигнал* $f_k = f(x_k)$, *где дискретная последовательность отсчетов* $x_k = x_{min} + k\frac{(x_{max} - x_{min})}{K}$, $k = \overline{0, K}$, *где* K *- количество отсчетов. На сигнал наложен дискретный равномерный шум* $\sigma = (\sigma_0, \dots, \sigma_K)$ *с нулевым средним и амплитудой, равномерно распределённой на интервале* $[-a, a]$: $\widetilde{f_k} = f_k + \sigma_k$, $\sigma_k = rnd(-a, a)$. *В зависимости от варианта работы необходимо осуществить фильтрацию сигнала* $\widetilde{f_k}$ *одним из методов взвешенного скользящего среднего.*

1.3. Условие варианта

Метод фильтрации	Метрика близости
Среднее геометрическое	Евклидова

2. Графики сигналов

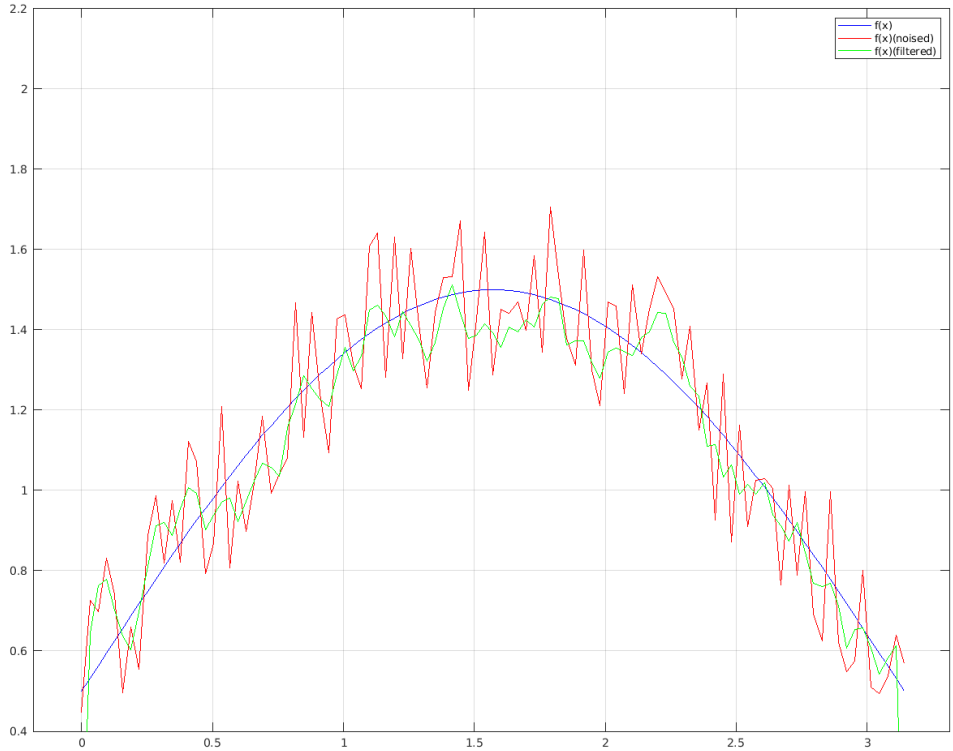

Рисунок 1 - Графики исходного сигнала, шума, очищенного сигнала для r = 3

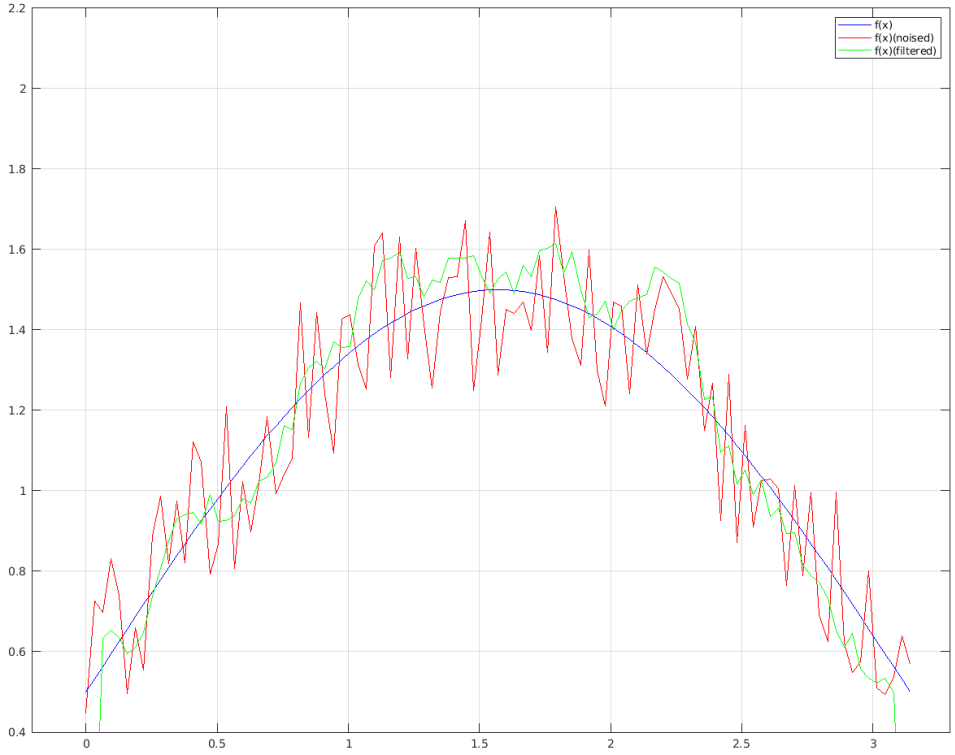

Рисунок 2 - Графики исходного сигнала, шума, очищенного сигнала для r = 5

3. Расчёт с помощью программы

For r = 3

h	dis	alpha	w	d
0	1.0133	[0.357770,0.284460,0.357770]	1.00478	0.131143
0.1	1.01328	[0.357560,0.284880,0.357560]	1.00476	0.131144
0.2	1.0132	[0.355787,0.288426,0.355787]	1.00468	0.131169
0.3	1.01321	[0.356388,0.287225,0.356388]	1.00469	0.131157
0.4	1.01321	[0.356371,0.287257,0.356371]	1.00468	0.131157
0.5	1.0132	[0.356124,0.287752,0.356124]	1.00468	0.131162
0.6	1.01321	[0.356224,0.287553,0.356224]	1.00468	0.13116
0.7	1.0132	[0.356052,0.287897,0.356052]	1.00468	0.131163
0.8	1.01322	[0.355016,0.289967,0.355016]	1.0047	0.131189
0.9	1.0132	[0.356109,0.287782,0.356109]	1.00468	0.131162
1	1.0132	[0.355912,0.288176,0.355912]	1.00468	0.131166

h*	J	w	d
1	1.00468	1.00468	0.131166

For r = 5

h	dis	alpha	w	d
0	0.987329	[0.446506,0.188220,0.067999,0.188220,0.446506]	0.976281	0.147292
0.1	0.844798	[0.339944,0.180705,0.169304,0.180705,0.339944]	0.830074	0.157035
0.2	0.830169	[0.287935,0.190264,0.169955,0.190264,0.287935]	0.814355	0.161269
0.3	0.823885	[0.318347,0.208487,0.183176,0.208487,0.318347]	0.807476	0.16361
0.4	0.823653	[0.282032,0.211892,0.168827,0.211892,0.282032]	0.807348	0.163073
0.5	0.819076	[0.245754,0.217072,0.205810,0.217072,0.245754]	0.80124	0.169999
0.6	0.826268	[0.340235,0.209042,0.208608,0.209042,0.340235]	0.808363	0.171077
0.7	0.816307	[0.295687,0.229053,0.225670,0.229053,0.295687]	0.796144	0.180309
0.8	0.82412	[0.255916,0.264625,0.221552,0.264625,0.255916]	0.802261	0.18855
0.9	0.816805	[0.268924,0.248395,0.212977,0.248395,0.268924]	0.796586	0.180614
1	0.816456	[0.302047,0.228405,0.194066,0.228405,0.302047]	0.798443	0.170556

h*	J	w	d
0.7	0.611393	0.796144	0.180309

Код программы приведён в Приложении.

4. Выводы

В процессе выполнения лабораторной работы я выполнил исследование стохастической фильтрации сигналов как задачи двухкритериальной оптимизации с использованием методов прямого пассивного поиска. В итоге, применяя случайный и прямой пассивный поиск, я получил сигнал, который, визуально, является отфильтрованной версией исходного сигнала с наложенным дискретным шумом.

Приложение

Файл main.cpp

```
#include <iostream>
#include "Programm.h"

int main()
	{
		worker work;
		work.pass();
		return 0;
	}
```

Файл Programm.h

```
#include <iostream>
#include <cmath>
#include <utility>
#include <stdexcept>
#include <vector>
#include <random>
#include <algorithm>

#include "Save.h"
/*
	Структура для описания условий задачи
*/
struct Data
	{
		std::pair<double,double> X_interval = {0.0, 3.141592653589793238463}; // Интервал
		size_t K_ = 100; // Количество отсчетов.
		double amplitude_of_noise = 0.5; // Амплитуда равномерного шума
		double P = 0.95; // Вероятность попадания в окрестность экстремума
		double eps = 0.01; // Интервал неопределенности
		double J = 90000000.0;
		size_t L_ = 10u; // Интервал неопределенности
		std::vector <size_t> radius_ {3,5}; // Размер скользящего окна
		std::vector <size_t> M_;
		std::vector <double> alpha;
		std::vector <double> xk;
		std::vector <double> F_xk_;
		std::vector <double> F_xk_noised;
		std::vector <double> F_xk_filtered;
		std::vector <double> lambda; // Веса свертки
		std::pair<double,double> omega_delta{90000000.0,90000000.0};

		void init()
			{
				std::random_device rd;
				std::mt19937 gen(rd());
				std::uniform_real_distribution <double> dis (-
amplitude_of_noise/2.0,amplitude_of_noise/2.0);

				for (size_t index = 0u;index <= L_;++index)
					{
						lambda.push_back(0.1 * index);
					}
				// Вычисляем xk
				for (size_t index = 0u;index <= K_;++index)
					{
						xk.push_back(calculate_x_k(index));
					}
				// Вычисляем f(xk) и f(xk) с шумами
				for (size_t index = 0u;index <= K_;++index)
```

```
                                        {
                                                F_xk_.push_back(signal(xk[index]));

        F_xk_noised.push_back(signal_plus_noise(xk[index],dis(gen)));
                                        }
                                // Вычисляем M
                                for (size_t index = 0u;index < radius_.size(); ++index)
                                        {
                                                M_.push_back((radius_[index]-1u)/2u);
                                        }
                                std::cout<<"";
                        }

                const double calculate_x_k(size_t k)
                        {
                                if (k >= 0u && k <= 100u)
                                        {
                                                return X_interval.first + k * (X_interval.second -
X_interval.first)/K_;
                                        }
                                else
                                        {
                                                throw std::runtime_error("Неверное значение для k!");
                                        }
                        }
                /**
                        Функция вычисления сигнала для xk
                */
                const double signal(double Xk_)
                        {
                                return std::sin(Xk_) + 0.5;
                        }
                /**
                        Функция вычисления сигнала с шумом для xk
                */
                const double signal_plus_noise(double Xk_,double noise)
                        {
                                return std::sin(Xk_) + 0.5 + noise;
                        }
                /**
                        Функция получения числа испытаний N
                */
                const double get_N()
                        {
                                return (log(1.0-P)/log(1.0 - ( eps/( X_interval.second -
X_interval.first ) )));
                        }
                /**
                        Функция вычисления среднего геометрического
                */
                double geom_average(std::vector<double> alpha,size_t M,size_t k)
                        {
                                double temp = 1.0;
                                if(k < M || k > K_ - M)       // k-M должно быть больше или равно 0
                                        {
                                                return 0.0;
                                        }
                                size_t degree = 0u;

                                for (size_t index = k - M ; index <= k + M ;++index)
                                        {
                                                degree = index + M - k ;
                                                if (degree >= alpha.size() )
                                                        {
                                                                degree  = alpha[ alpha.size()-(degree-
(alpha.size()-1u))-1u ];
                                                        }
```

```
                                        temp *= pow(F_xk_noised[index],alpha[degree]);
                                }
                        return temp;
                }
        double sum(std::vector<double> alpha,size_t a,size_t b)
                {
                        double temp = 0.0;
                        for (size_t index = a;index <= b;++index)
                                {
                                        if (index + 1u > alpha.size())
                                                {
                                                        temp+=alpha[index-alpha.size()];
                                                        continue;
                                                }
                                        temp+=alpha[index];
                                }
                        return temp;
                }
        void init_Fx_filtered(std::vector <double> input_alpha,size_t M)
                {
                        F_xk_filtered.clear();
                        for (size_t index = 0u;index <= K_;++index)
                                {

        F_xk_filtered.push_back(geom_average(input_alpha,M,index));
                                }
                }
        };

class worker
        {
                double N_;
                V10 v10;
                std::vector <Save> save;
                Data data;
        public:
                worker()
                        {
                                data.init();
                                N_ = data.get_N();
                        }
                void pass()
                        {
                                Save temp_save;
                                std::pair<double,double> temp_omega_delta;
                                double temp_J = 0.0;
                                std::random_device rd;
                                std::mt19937 gen(rd());
                                std::uniform_real_distribution <double> dis (0.0, 1.0);
                                std::vector<double> temp_alpha;

                                double temp;

                                temp_save.graphic.x = data.xk;
                                temp_save.graphic.Function = data.F_xk_;
                                temp_save.graphic.Function_noised = data.F_xk_noised;

                                for (size_t jump = 0u; jump < data.radius_.size(); ++jump)
                                        {
                                                double r = data.radius_[jump];
                                                double M = data.M_[jump];

                                                temp_save.r = data.radius_[jump];
                                                for (size_t l = 0u; l < data.lambda.size(); ++l) //
для различных значений весов
                                                        {
                                                                temp_save.h = data.lambda[l];
```

```
                                                        for (size_t index = 0u; index < N_;
++index)              // Число испытаний N
                                                                {

        temp_alpha.reserve(M+1u);
                                                                temp_alpha.resize(M+1u);
                                                                temp_alpha.back() =
dis(gen);
                                                                if (data.M_[jump] >= 2u)
                                                                        {
                                                                                for
(size_t count = 1u; count < data.M_[jump] ; ++count)
                                                                                        {
        std::uniform_real_distribution <double> temp_dis(0.0,1.0 - data.sum(temp_alpha,M, r-M-1));

        temp_alpha.at(count) = 0.5 * temp_dis(gen);
                                                                                        }
                                                                        }
                                                                temp_alpha.front() =
0.5*(1.0 - data.sum(temp_alpha,1u,r-2u));
                                                                // Уже сгенерирован
набор альфа
        data.init_Fx_filtered(temp_alpha,M); // Находим отфильтрованную функцию
                                                                // Находим критерий
зашумленности (по Евклиду)
                                                                double summ = 0.0;
                                                                for (size_t kol = 1u;
kol <= data.K_; ++kol)
                                                                        {
                                                                                summ +=
pow((data.F_xk_filtered[kol] - data.F_xk_filtered[kol - 1u]) , 2);
                                                                        }
                                                                temp_omega_delta.first =
sqrt(summ); summ=0.0;
                                                                // Находим критерий
отличия (по Евклиду)
                                                                for (size_t kol = 0u;
kol <= data.K_; ++kol)
                                                                        {
                                                                                summ +=
pow((data.F_xk_filtered[kol] - data.F_xk_noised[kol]) , 2);
                                                                        }
                                                                temp_omega_delta.second
= sqrt(summ/data.K_);

                                                                temp_J = (
data.lambda[l] * temp_omega_delta.first) + (1 - data.lambda[l])*temp_omega_delta.second;

                                                                if (temp_J < data.J)
                                                                        {
                                                                                data.J =
temp_J;
        data.omega_delta =    temp_omega_delta;

        temp_save.alpha = temp_alpha;

        temp_save.metrics.delta = temp_omega_delta.second;

        temp_save.metrics.omega = temp_omega_delta.first;

        temp_save.metrics.J = temp_J;

        temp_save.graphic.Function_filtered = data.F_xk_filtered;
```

```
	temp_save.distance = sqrt(pow(temp_save.metrics.delta,2) +
pow(temp_save.metrics.omega,2));
                                                                                }
                                                                        }
                                                                data.J = 90000000.0;
                                                                data.omega_delta =
{90000000.0,90000000.0};
                                                                save.push_back(temp_save);
                                                        }

                                                if (jump == 0u)
                                                        {
                                                                v10.resut1.saves = save ;
                                                                save.clear();
                                                        }
                                                else
                                                        {
                                                                v10.resut2.saves = save ;
                                                                save.clear();
                                                        }
                                        }

                                find_best();
                                std::cout<<"\t\t\t\t## For r = 3 ##\n";
                                v10.resut1.print();
                                std::cout<<"\t\t\t\t## For r = 5 ##\n";
                                v10.resut2.print();
                                /*std::cout<<"\n\n For r = 3\n";
                                v10.resut1.best_save.graphic.print();
                                std::cout<<"For r = 5 \n";
                                v10.resut2.best_save.graphic.print();*/
                        }
                void find_best()
                        {
                                Save temp_best;
                                double best_distance = 999999900.0;
                                for (Save save:v10.resut1.saves)
                                        {
                                                if (save.distance < best_distance)
                                                        {
                                                                best_distance = save.distance;
                                                                temp_best = save;
                                                        }
                                        }
                                v10.resut1.best_save = temp_best;
                                best_distance = 999999900.0;
                                for (Save save:v10.resut2.saves)
                                        {
                                                if (save.distance < best_distance)
                                                        {
                                                                best_distance = save.distance;
                                                                temp_best = save;
                                                        }
                                        }
                                v10.resut2.best_save = temp_best;
                        }
        };
```

Файл Save.h

```
#pragma once
#include <vector>
#include <iostream>
#include <string>
#include <iomanip>
#include <iterator>
```

```
struct Graph
	{
		std::vector <double> x;		// Вектор параметров
		std::vector <double> Function;		// Вектор значений функции
		std::vector <double> Function_noised; // Вектор значений функции с шумами
		std::vector <double> Function_filtered; // Вектор значений отфильтрованной функции

		void print()
			{
				std::cout<<"x = ["<<x.front;
				for (auto num = std::next(x.begin());num !=x.end();++num)
					{
						std::cout<<", "<<(*num);
					}
				std::cout<<"]\n";
				std::cout<<"y(x) = ["<<Function.front;
				for (auto num = std::next(Function.begin());num !=
Function.end();++num)
					{
						std::cout<<", "<<(*num);
					}
				std::cout<<"]\n";
				std::cout<<"y1(x) = ["<<Function_noised.front;
				for (auto num = std::next(Function_noised.begin());num !=
Function_noised.end();++num)
					{
						std::cout<<", "<<(*num);
					}
				std::cout<<"]\n";
				std::cout<<"y2(x) = ["<<Function_filtered.front;
				for (auto num = std::next(Function_filtered.begin());num !=
Function_filtered.end();++num)
					{
						std::cout<<", "<<(*num);
					}
				std::cout<<"]\n";
			}
	};
struct Mertics
	{
		double omega = 0.0;
		double delta = 0.0;
		double J = 0.0;
	};
struct Save
	{
		double r = 0.0;
		double h = 0.0;
		double distance = 0.0;
		std::vector <double> alpha;
		Graph graphic;
		Mertics metrics;
		void print()
			{
				std::string temp ;
				size_t reserve = (alpha.size() + (alpha.end()-alpha.begin()-1u))*8u
+ 4u;
				temp=std::to_string(alpha.front());
				for (size_t index = 1u ; index < alpha.size() + (alpha.end()-
alpha.begin()-1u) ; ++index)
					{
						if (index >= alpha.size())
							{
								temp =temp + ',' +
std::to_string(alpha[ alpha.size()-(index-(alpha.size()-1u))-1u ]);
								continue;
```

```
                                        }
                                    temp =temp + ',' + std::to_string(alpha[index]);
                                }
                            std::cout << std::left << std::setprecision(6) <<"| "<<
std::setw(10)<<h<<" | "<< std::setw(10)<<distance<<" |["<< std::setw(reserve)<<temp<<"]| "<<
std::setw(10)<<metrics.omega<<" | "<< std::setw(10)<<metrics.delta<<" |\n";
                        }
        };
struct Result
        {
                std::vector <Save> saves;
                Save best_save;

                void print()
                        {
                            size_t reserve = (saves.front().alpha.size() +
(saves.front().alpha.end()-saves.front().alpha.begin()-1u))*8u + 4u;
                            std::cout << std::left << std::setprecision(6) <<"| "<<
std::setw(10)<<"h"<<" | "<< std::setw(10)<<"dis"<<" | "<< std::setw(reserve)<<"alpha"<<" | "<<
std::setw(10)<<"w"<<" | "<< std::setw(10)<<"d"<<" |\n";
                            for (Save save:saves)
                                {
                                    save.print();
                                }
                            std::cout<<"\n";
                            print_best();
                            std::cout<<"\n";
                        }
                void print_best()
                        {
                            std::cout << std::left << std::setprecision(6) <<"| "<<
std::setw(10)<<"h*"<<" | "<< std::setw(10)<<"J"<<" | "<< std::setw(10)<<"w"<<" | "<<
std::setw(10)<<"d"<<" |\n";
                            std::cout << std::left << std::setprecision(6) <<"| "<<
std::setw(10)<<best_save.h<<" | "<< std::setw(10)<<best_save.metrics.J<<" | "<<
std::setw(10)<<best_save.metrics.omega<<" | "<< std::setw(10)<<best_save.metrics.delta<<" |\n";
                        }
        };

struct V10
        {
                Result resut1; // Для r=3
                Result resut2; // Для r=5
        };
```