Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
 высшего образования   
«Национальный исследовательский университет «МЭИ»

Кафедра «Релейная защита и автоматизация энергосистем»

Лабораторная работа №2

«Дифференциальная защита шин»

Выполнил: Максимов Р.С.

Группа: Э-13м-19

Проверил: Холодов А. С.

Москва 2020

**Задание на лабораторную работу №2**

Общие сведения:

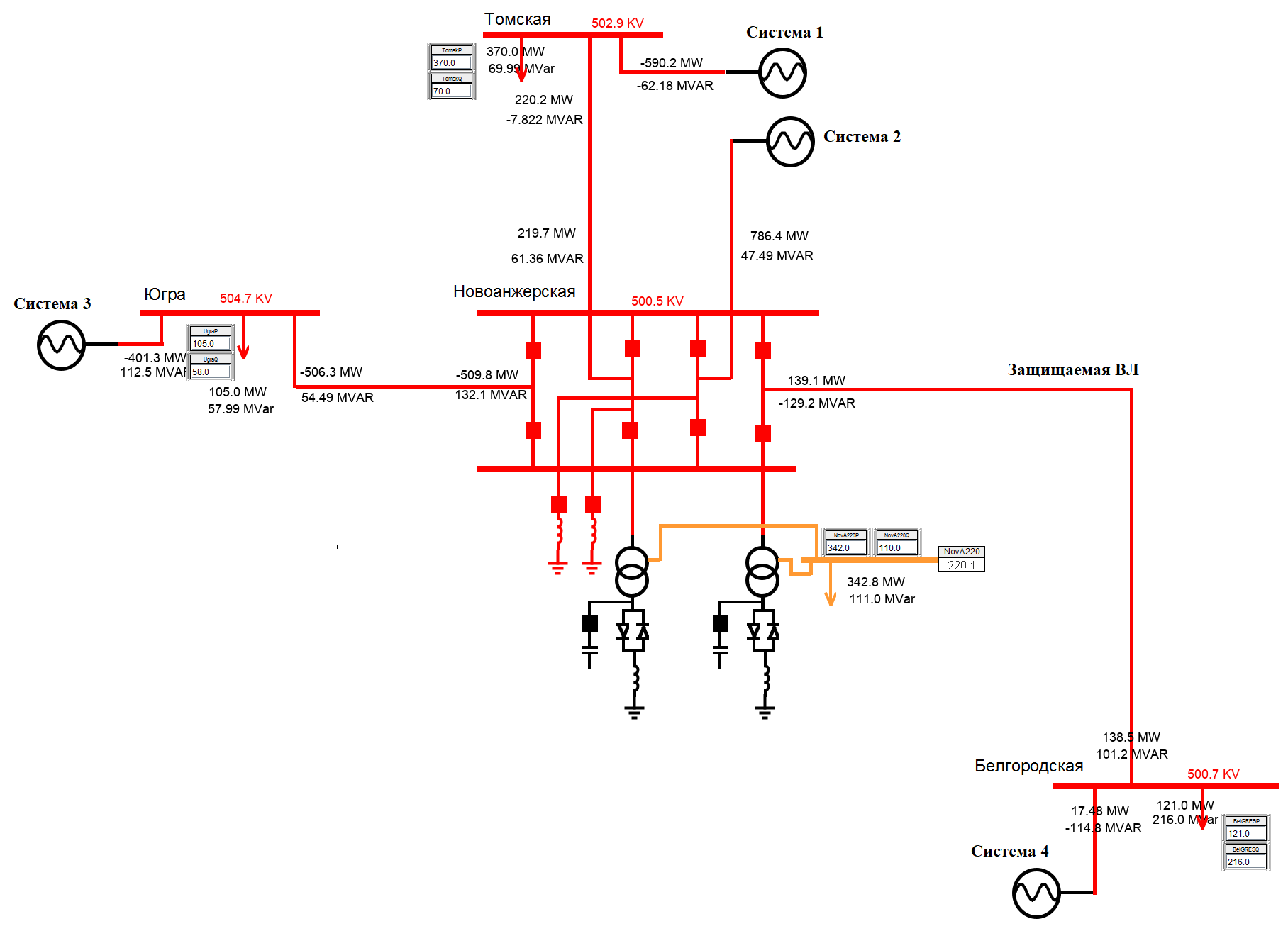
Тип выполняемой защиты: **ДЗШ**

Тип блокировки: **2 гармоника**

Количество присоединений (ДЗШ): **5 присоединений**

Логика производителя: **ЭКРА**

Схема сети:



Предоставляется набор осциллограмм сomtrade с короткими замыканиями в различных точках. Необходимо разработать трехфазную дифференциальную защиту элемента.

Шина:

Uвн = 500 кВ, первичные токи

**Теоретическая часть.**

Дифференциальная токовая защита шин предназначена для быстрого отключения электрических цепей, включенных на сборные шины, при КЗ на сборных шинах или на любом другом оборудовании, входящем в зону действия защиты.

Зона ее действия ограничивается трансформаторами тока, к которым подключены реле защиты. В основу выполнения защиты положен принцип сравнения значений и фаз токов электрических цепей при КЗ и других режимах работы.

Для выполнения защиты дифференциальное реле РТ подключают к трансформаторам тока присоединений, как показано на рис. 1. При таком включении ток в реле всегда будет равен геометрической сумме вторичных токов присоединений. При КЗ на шинах (рис. 1, а) вторичные токи присоединений будут иметь одно направление и через реле будет проходить сумма этих токов:

.

При внешнем КЗ (рис. 1,б) ток в обмотке реле:

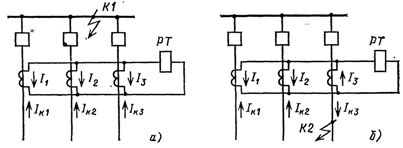


Рис. 1. Токи в реле дифференциальной токовой защиты шин при КЗ на шинах (а) и внешнем КЗ (б).

В качестве защиты шин 500 кВ ООО НПП «Экра» предлагает терминал ШЭ 2710 562. Для этого терминала были рассчитаны уставки, а также представлен программный код алгоритма ДЗШ.

Защита выполнена пофазной и действует при всех видах КЗ на шинах

1. Выравнивание токов присоединений

В рамках данной работы выравнивание не осуществлялось, так как были даны значения первичных токов. Дальнейший расчет уставок осуществлен для первичных величин.

1. Выбор тока начала торможения

Регулируется в диапазоне от 1,00 до 2,00 к базисному току. Расчет уставок производим в первичных значениях токов и принимаем А, как номинальный ток наиболее нагруженного присоединения, что можно наблюдать на рис. 2.

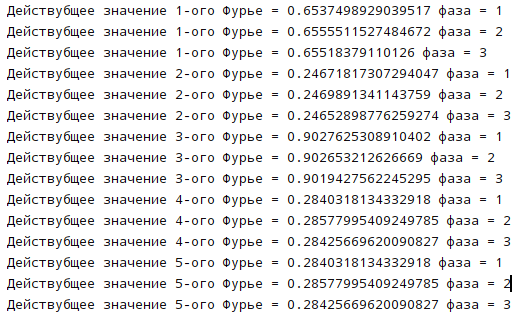


Рис. 2. Действующие значения токов отходящих присоединений.

1. Расчет начального тока срабатывания

При расчёте **начального тока срабатывания** ПО дифференциальной защиты учитываются следующие условия:

* 1. Отстройка от максимального тока в защите при разрыве ее вторичных цепей в рабочем режиме:

 А, где

 =1,2 - коэффициент отстройки;

=902 А – первичный ток нагрузки линии w, как наиболее нагруженного присоединения;

* 1. Отстройка от расчетного первичного тока небаланса в режиме, соответствующем началу торможения:

А, где

=1,5 - коэффициент отстройки, учитывающий погрешности ДЗШ, ошибки расчета и необходимый запас;

 - составляющая первичного тока небаланса, обусловленная погрешностью ТТ в режиме, соответствующем началу торможения:

, где

 = 1,0 – коэффициент однотипности ТТ;

= 1,3 – коэффициент, учитывающий переходный режим,

принимается равным 1,3 при  = 1,0 Iном;

= 0,1 – полная относительная погрешность основных ТТ;

 = 0,0 – полная относительная погрешность промежуточных ТТ

принимается равной 0, так как они не установлены;

 = 0,02 – полная относительная погрешность выравнивания;

= 902 – ток начала торможения.

В качестве начального тока срабатывания выбираем наибольшее из рассчитанных значений, то есть равное 1082,4 А.

1. Расчет коэффициента торможения

Коэффициент торможения дифференциальной защиты выбираем из условий:

* 1. Отстройки от тока небаланса в переходном режиме при внешнем КЗ:

, где

 = 1,5 – коэффициент отстройки;

= 1082,4 А– принятое значение начального тока срабатывания защиты;

= 902 А – принятое значение тока начала торможения;

 - значение максимального расчетного тока небаланса при расчетном внешнем КЗ, протекающего через защиту, от которого защита должна быть отстроена выбором ;



= 1,0 – коэффициент однотипности ТТ;

2,0 – коэффициент, учитывающий переходный режим внешнего КЗ;

= 0,1 – полная относительная погрешность основных ТТ;

= 0,02 – полная относительная погрешность выравнивания;

= 0,0 – полная относительная погрешность промежуточных ТТ;

принимается равной 0, так как ПТТ не установлены;

– максимальное значение тока внешнего металлического КЗ:

А; (Было получено из comtrade файла)

А.

 - значение расчетного тормозного тока в защите при расчетном внешнем КЗ;

,

Тогда:



Таким образом, коэффициента торможения:

.

* 1. Отстройка от тока небаланса в режиме качаний:

В рамках данной работы не было исходных данных для режима качания.

Коэффициент торможения принимается не меньше 0,6, поэтому 

1. Блокировка по 2 гармонике

Также осуществляется блокировка защиты по отношению тока второй гармонике к первой. В случае превышения данного отношения 0,15, защита блокируется.

**Практическая часть.**

В классе Main запускаем работу программы, создаем объект класса InputData, на вход которого подаем имя файла comtrade (cfg+data) и количество значений аналоговых сигналов.

**public class** Main {  
 **public static void** main(String[] args) **throws** FileNotFoundException {  
  
 InputData inD1 = **new** InputData(**"KzB"**, 5);  
 inD1.start();  
  
 }  
}

Класс inputData необходим для считывания данных с предоставляемых файлов и работы с ними. В данном случае, осуществляется считывание с файлов формата .cfg и .dat. В методе start() реализован алгоритм работы с файлами, а именно преобразование значений файла в мгновенные значения токов. В данном классе создаются объекты классов SampleValues, RMSValues, Fourie, Breakers в соответствии с заданным количеством аналоговых сигналов, а также объекты классов Logic, Vector, OutPutData. Также было создано 2 объекта класса Charts для построения графиков в отдельных окнах: аналговые и дискретные значения.

**import** java.io.\*;  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
  
**public class** InputData {  
  
 **public** File **comtrCfg**, **comtrDat**;  
 **private** BufferedReader **br**;  
 **private** String **line**;  
 **private** String[] **lineData**;  
 **private double**[] **k1**;  
 **private double**[] **k2**;  
 **private** String **comtradeName**;  
 **private int numbers**;  
 **private** String **nameFile**;  
 **private** ArrayList<SampleValues> **sv** = **new** ArrayList<SampleValues>();  
 **private** ArrayList<RMSValues> **rms** = **new** ArrayList<RMSValues>();  
 **private** ArrayList<Fourie> **filter** = **new** ArrayList<Fourie>();  
 **private** OutputData **od** = **new** OutputData();  
 **private** ArrayList<Breaker> **breakers** = **new** ArrayList<Breaker>();  
 **private** Vector **vectors** = **new** Vector();  
 **private boolean t** = **false**;  
 **private** Logic **logic** = **new** Logic();  
  
  
 **public** InputData(String nameFile, **int** numbers) {  
 **this**.**nameFile** = nameFile;  
 **this**.**numbers** = numbers;  
 }  
  
 **public void** start() **throws** FileNotFoundException {  
 **int** period = 20; *// millisec* **double** step = 0.001;  
 **for** (**int** i = 0; i < **numbers**; i++) {  
 **sv**.add(**new** SampleValues());  
 **rms**.add(**new** RMSValues());  
 **filter**.add(**new** Fourie(i));  
 **filter**.get(i).set();  
 **filter**.get(i).setVector(**vectors**);  
 **filter**.get(i).setPeriod(period);  
 **breakers**.add(**new** Breaker());  
 }  
 **try** {  
 *//объект класса Charts для построения графика токов в отдельном окне* Charts chartss = **new** Charts(**"Токи"**);  
 chartss.setTimeStep(step);  
 *//объект класса Charts для построения графика дискретных значений в отдельном окне* Charts chartsDiscrete = **new** Charts(**"Дискретные сигналы"**);  
 chartsDiscrete.setTimeStep(step);  
 *//генерируем серии для токов* String fun;  
 **for** (**int** i = 0; i < 2 \* **numbers** + 2; i++) {  
 **if** (i < 5) {  
 fun = **"М. зн."** + (i + 1); *//"Мгн. значение " +i+"-ого фидера"* } **else if** (i < 10) {  
 fun = **"Д. зн. "** + (i - 4); *//"Действ. значение " +i+"-ого фидера"* } **else if** (i < 11) {  
 fun = **"Диф. ток"**;  
 } **else** {  
 fun = **"Торм. ток"**;  
 }  
 chartss.createAnalogChart(fun, i);  
 chartss.addSeries(**"Фаза А"**, i, 0);  
 chartss.addSeries(**"Фаза B"**, i, 1);  
 chartss.addSeries(**"Фаза С"**, i, 2);  
 }  
  
  
 chartsDiscrete.createAnalogChart(**"trip"**, 0);  
 chartsDiscrete.addSeries(**"Trip"**, 0, 0);  
 chartsDiscrete.createAnalogChart(**"Str"**, 1);  
 chartsDiscrete.addSeries(**"Str"**, 1, 0);  
 chartsDiscrete.createAnalogChart(**"trip"**, 2);  
 chartsDiscrete.addSeries(**"Blk"**, 2, 0);  
  
 **logic**.setVectors(**vectors**);  
 **logic**.setBlkSecondHarmonic(0.15);  
 **logic**.setOd(**od**);  
 **logic**.setBeginingDiffCurrent(1.082);  
 **logic**.setCoefDrag(0.6);  
 **logic**.setBeginingDragCurrent(0.9);  
  
 **od**.setBreakers(**breakers**);  
 **od**.setSetTime(0.04);  
 **od**.setTimeStep(step);  
  
 *//путь к файлам comtrade* **comtradeName** = **nameFile**;  
 String path = **"D:\\education\\Algoritms\\Лабораторная работа №2\\ОпытыComtrade\\DPB\\5 sections\\"**;  
 String cfgname = path + **comtradeName** + **".cfg"**;  
 String datName = path + **comtradeName** + **".dat"**;  
 **comtrCfg** = **new** File(cfgname);  
 **comtrDat** = **new** File(datName);  
  
 *//открываем cfg файл для получения коэф a и b для расчета y = ax+b* **br** = **new** BufferedReader(**new** FileReader(**comtrCfg**));  
 **int** lineNumber = 0, count = 0, numberData = 100;  
 **try** {  
 **while** ((**line** = **br**.readLine()) != **null**) {  
*// System.out.println(line);* lineNumber++;  
 **if** (lineNumber == 2) {  
 *//получаем количество аналоговых сигналов во 2 строке cfg файла "4,3A,1D"* numberData = Integer.*parseInt*(**line**.split(**","**)[1].replaceAll(**"A"**, **""**));  
 *//создаем double " массивы " с размерностью равной количеству* **k1** = **new double**[numberData];  
 **k2** = **new double**[numberData];  
 }  
 *//коэф находятся на 3,4,5 строке это 5 и 6 элемент строки при парсинге* **if** (lineNumber > 2 && lineNumber < numberData + 3) {  
 **k1**[count] = Double.*parseDouble*(**line**.split(**","**)[5]);  
 **k2**[count] = Double.*parseDouble*(**line**.split(**","**)[6]);  
 count++;  
 }  
 }  
 count = 0;  
 **br** = **new** BufferedReader(**new** FileReader(**comtrDat**));  
 **while** ((**line** = **br**.readLine()) != **null**) {  
 count++;  
 **if** ((count > 100 && count < 1200)) {  
 **lineData** = **line**.split(**","**);  
 **int** b = 0;  
 **int** i = 0;  
 **while** (i < **numbers**) { *//проходимся по всем фидерам* **if** (!**t**) {  
 **sv**.get(i).setPhA(Double.*parseDouble*(**lineData**[b + 2]) \* **k1**[b] + **k2**[b]);  
 **sv**.get(i).setPhB(Double.*parseDouble*(**lineData**[b + 3]) \* **k1**[b + 1] + **k2**[b + 1]);  
 **sv**.get(i).setPhC(Double.*parseDouble*(**lineData**[b + 4]) \* **k1**[b + 2] + **k2**[b + 2]);  
 } **else** {  
 **sv**.get(i).setPhA(0.);  
 **sv**.get(i).setPhB(0.);  
 **sv**.get(i).setPhC(0.);  
 }  
 *//вывод мгновенных значений* chartss.addAnalogData(i, 0, **sv**.get(i).getPhA());  
 chartss.addAnalogData(i, 1, **sv**.get(i).getPhB());  
 chartss.addAnalogData(i, 2, **sv**.get(i).getPhC());  
 b = b + 3; *//чтобы прыгать через фазы для следующего фидера  
 //объект SV помещаем в объект filter,чтобы получать значения* **filter**.get(i).setSv(**sv**.get(i));  
 *//объект rms помещаем в объект filter,чтобы устанавливать значения* **filter**.get(i).setRms(**rms**.get(i));  
 *//расчет ортогональных составляющих* **filter**.get(i).calculate();  
 *//вывод средних значений* chartss.addAnalogData(i + 5, 0, **rms**.get(i).getPhA());  
 chartss.addAnalogData(i + 5, 1, **rms**.get(i).getPhB());  
 chartss.addAnalogData(i + 5, 2, **rms**.get(i).getPhC());  
*// System.out.println("ФАЗА А = " + rms.get(i).getPhA());  
// System.out.println("ФАЗА В = " + rms.get(i).getPhB());  
// System.out.println("ФАЗА С = " + rms.get(i).getPhC());* i++;  
 }  
  
 *//отсылка векторов в логику, если токи есть* **if** (!**t**) **logic**.setVectors();  
 *//отключение (прекращение цикла(для имитации отключения), так как произошло срабатывание, выключатели отключены)* **breakers**.forEach(e -> **t** = **t** | (!e.isState()));  
 *//диф ток пофазно* chartss.addAnalogData(2 \* **numbers**, 0, **logic**.getDiffCurrent()[0]);  
 chartss.addAnalogData(2 \* **numbers**, 1, **logic**.getDiffCurrent()[1]);  
 chartss.addAnalogData(2 \* **numbers**, 2, **logic**.getDiffCurrent()[2]);  
 *//ток торможения пофазно* chartss.addAnalogData(2 \* **numbers** + 1, 0, **logic**.getCurrentDrag()[0]);  
 chartss.addAnalogData(2 \* **numbers** + 1, 1, **logic**.getCurrentDrag()[1]);  
 chartss.addAnalogData(2 \* **numbers** + 1, 2, **logic**.getCurrentDrag()[2]);  
 *//вывод дискретных значений  
 //сигнал блокировки (преобразование boolean -> int)* chartsDiscrete.addAnalogData(0, 0, boolToInt(**od**.getTripper()));  
 chartsDiscrete.addAnalogData(1, 0, boolToInt(**od**.getStr()));  
 chartsDiscrete.addAnalogData(2, 0, boolToInt(**od**.isBlk()));  
 }  
 }  
 } **catch** (ArrayIndexOutOfBoundsException e) {  
 e.printStackTrace();  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
  
 }  
  
 *// функция для преобразования boolean -> int* **public int** boolToInt(**boolean** b) {  
 **return** Boolean.*compare*(b, **false**);  
 }  
  
  
}

Класс Fourie преобразует мгновенные значения сигналов в действующее, а также реализует разложение сигналов на ортогональные составляющие по 1 и 2 гармонике, последнее нужно для сигнала блокировки.

**import** java.util.ArrayList;  
  
**public class** Fourie {  
 **private double**[] **bufferPhA** = **new double**[80];  
 **private double**[] **bufferPhB** = **new double**[80];  
 **private double**[] **bufferPhC** = **new double**[80];  
 **private double**[] **A0** = **new double**[3]; *//3 - 3 phases* **public double**[] **Ak1** = **new double**[3];  
 **public double**[] **Bk1** = **new double**[3];  
 **public double**[] **Ak2** = **new double**[3];  
 **public double**[] **Bk2** = **new double**[3];  
 **private** ArrayList<**double**[]> **buff** = **new** ArrayList<**double**[]>();  
 **private int count** = 0;  
 **private** SampleValues **sv**;  
 **private** RMSValues **rms**;  
 **private** Vector **vector**;  
 **private int number**;  
 **private int period** = 20; *//количество точек за период* **public void** set() {  
 **buff**.add(**bufferPhA**);  
 **buff**.add(**bufferPhB**);  
 **buff**.add(**bufferPhC**);  
 }  
  
 Fourie(**int** number) {  
 **this**.**number** = number;  
 }  
  
  
 **public void** calculate() {  
 **for** (**int** i = 0; i < 3; i++) { *//3 - 3 phases* **double**[] actual\_buf = **buff**.get(i);  
 **double** sumPh = **sv**.get(i + 1) - actual\_buf[**count**];  
 *//Алгоритм фурье: постоянная составляющая + 1 гармоника  
 //расчет постоянной составляющей, возникает при КЗ. В норм. режиме равна нулю -> интеграл синусоиды = 0* **A0**[i] = **A0**[i] + sumPh / **period**;  
 *//расчет cos и sin составляющей для первой гармоники* **Ak1**[i] = **Ak1**[i] + 2 \* (Math.*cos*(**count** \* 2 \* Math.***PI*** / **period**) \* sumPh) / **period**;  
 **Bk1**[i] = **Bk1**[i] + 2 \* (Math.*sin*(**count** \* 2 \* Math.***PI*** / **period**) \* sumPh) / **period**;  
 *//расчет ортогональных составляющих 2 гармоники для блокировки* **Ak2**[i] = **Ak2**[i] + 2 \* (Math.*cos*(**count** \* 2 \* 2 \* Math.***PI*** / **period**) \* sumPh) / **period**;  
 **Bk2**[i] = **Bk2**[i] + 2 \* (Math.*sin*(**count** \* 2 \* 2 \* Math.***PI*** / **period**) \* sumPh) / **period**;  
 *//расчет действующего значения для 1 гармоники по cos и sin составляющей* **double** Ck1 = Math.*sqrt*((Math.*pow*(**Ak1**[i], 2) + Math.*pow*(**Bk1**[i], 2)) / 2);  
 **double** x = Math.*sqrt*(Math.*pow*(Ck1, 2) + Math.*pow*(**A0**[i], 2));  
*// System.out.println("Действубщее значение "+(number+1)+"-ого Фурье = "+x +" фаза = "+(i+1));  
 //суммарная составляющая* actual\_buf[**count**] = **sv**.get(i + 1);  
 **buff**.set(i, actual\_buf);  
 *//устанавливаем действующие значения для построения графика* **rms**.set(i + 1, x);  
 }  
 **count**++;  
 **if** (**count** == **period**) {  
 **count** = 0;  
 }  
  
 **vector**.getVectorsFirstAndSecondHarmonic(**Ak1**, **Bk1**, **Ak2**, **Bk2**, **number**);  
  
 }  
  
  
 **public void** setSv(SampleValues sv) {  
 **this**.**sv** = sv;  
 }  
  
 **public void** setRms(RMSValues rms) {  
 **this**.**rms** = rms;  
 }  
  
 **public** Vector getVector() {  
 **return vector**;  
 }  
  
 **public void** setVector(Vector vector) {  
 **this**.**vector** = vector;  
 }  
  
 **public void** setPeriod(**int** period) {  
 **this**.**period** = period;  
 }  
  
}

Объект класса Vector принимает значения ортогональных составляющих от всех объектов класса Fourie (их 5), что необходимо для объекта класса Logic, и расфасовывает их пофазно для удобства следующим образом: [AAAAA,BBBBB,CCCCC]

**import** java.util.ArrayList;  
  
**public class** Vector {  
 **private double**[] **cosFirst** = **new double**[15]; *// 5 \* 3 ???? 5 - отходящих линий, 3 - фазы* **private double**[] **sinFirst** = **new double**[15]; *// 5 \* 3 ???? 5 - отходящих линий, 3 - фазы* **private double**[] **cosSecond** = **new double**[15]; *// 5 \* 3 ???? 5 - отходящих линий, 3 - фазы* **private double**[] **sinSecond** = **new double**[15]; *// 5 \* 3 ???? 5 - отходящих линий, 3 - фазы* **public void** getVectorsFirstAndSecondHarmonic(**double**[] x\_harmonic1, **double**[] y\_harmonic1, **double**[] x\_harmonic2,  
 **double**[] y\_harmonic2, **int** numberFourie) {  
  
 **for** (**int** i = 0; i < 15; i = i + 5) { *// 5 \* 3 ???? 5 - отходящих линий, 3 - фазы* **cosFirst**[numberFourie + i] = x\_harmonic1[i / 5]; *//[AAAAA,BBBBB,CCCCC] - кос. составляющие по 1 гармонике* **sinFirst**[numberFourie + i] = y\_harmonic1[i / 5];  
 **cosSecond**[numberFourie + i] = x\_harmonic2[i / 5];  
 **sinSecond**[numberFourie + i] = y\_harmonic2[i / 5];  
 }  
  
 }  
  
 **public double**[] getCosFirst() {  
 **return cosFirst**;  
 }  
  
 **public double**[] getSinFirst() {  
 **return sinFirst**;  
 }  
  
 **public double**[] getCosSecond() {  
 **return cosSecond**;  
 }  
  
 **public double**[] getSinSecond() {  
 **return sinSecond**;  
 }  
}

Объект класса Logic принимает значения всех ортогональных составляющих пофазно с помощью метода **setVectors()**, который запускается в объекте InputData, после того как все значения будут получены от всех отходящих линий от шины, он необходим для расчет дифф. тока по 1 и 2 гармонике пофазно. С помощью метода **summ()** рассчитывается дифф. ток по 1 гармонике, на ее вход отсылаются ортогональные составляющиие, а с помощью **getCurrentDrag**() находится тормозной ток. После этого запускается метод **protect(),** который сначала сравнивает значения дифф. тока и тормозного тока и формирует сигнал пуска защиты, а потом проверяет на отсутствие блокировки по 2 гармонике, после чего формируется сигнал срабатывания защиты. Формирование выходных дискретных сигналов производится пофазно, то есть для наличия сигнала блокировки достаточно наличия блокировки в одной фазе. Сигнал пуска формируется при наличии выполнения условия превышения дифф тока над тормозным, сигнал срабатывания уже формируется при отсутствии блокировки по 2 гармонике.

**import** java.util.ArrayList;  
  
**public class** Logic {  
 **private double**[] **diffCurrent** = **new double**[3];  
 **private double**[] **blkdiff** = **new double**[3];  
 **private double blkSecondHarmonic**;  
 **private** Vector **vectors**;  
 **private** OutputData **od**;  
 **private double beginingDiffCurrent**; *//Id0* **private double beginingDragCurrent**; *//It0* **private double coefDrag**; *//kt* **private double**[] **currentDrag** = **new double**[3];  
  
 **public void** setVectors() {  
 **for** (**int** i = 0; i < 3; i++) {  
 *//диф ток по 1 гармонике* **diffCurrent**[i] = getsumm(i \* 5, **vectors**.getCosFirst(), **vectors**.getSinFirst());  
 *//диф ток по 2 гармонике, для блокировки* **blkdiff**[i] = getCurrentDrag(i \* 5, **vectors**.getCosSecond(), **vectors**.getSinSecond());  
 }  
 *//запуск логики* protect();  
  
 }  
  
 **private void** protect() {  
 **boolean** str = **false**;  
 **boolean** blk = **false**;  
 **boolean** trip = **false**;  
 **for** (**int** i = 0; i < 3; i++) {  
 *//It* **currentDrag**[i] = getCurrentDrag(i \* 5, **vectors**.getCosFirst(), **vectors**.getSinFirst());  
 **if** (**diffCurrent**[i] > **coefDrag** \* (**currentDrag**[i] - **beginingDragCurrent**) + **beginingDiffCurrent**) {  
 str = str | **true**;  
 *//проверка на блокировку по 2 гармонике* **if** (blocking(**blkdiff**[i] / **diffCurrent**[i])) {  
 blk = blk | **false**;  
 trip = trip | **true**;  
 } **else** {  
 blk = blk | **true**;  
 }  
 }  
 }  
 **od**.setBlk(blk);  
 **od**.setTripper(trip);  
 **od**.setStr(str);  
 *//прием дискретных сигналов от логики* **od**.takeDiscreteSignals();  
  
 }  
  
 **private boolean** blocking(**double** relation) {  
 **return** (relation < **blkSecondHarmonic**);  
 }  
  
 **private double** getsumm(**int** phasa, **double**[] Icos, **double**[] Isin) {  
 **double** summX = 0.;  
 **double** summY = 0.;  
 **double** resultCurrent = 0;  
 **for** (**int** i = phasa; i < (5 + phasa); i++) {  
 summX = summX + Icos[i];  
 summY = summY + Isin[i];  
 }  
 resultCurrent = Math.*sqrt*((Math.*pow*(summX, 2) + Math.*pow*(summY, 2)) / 2);  
 **return** resultCurrent;  
 }  
  
 **private double** getCurrentDrag(**int** phasa, **double**[] Icos, **double**[] Isin) {  
 **double** summ = 0.;  
 **double** currentDrag = 0;  
 **for** (**int** i = phasa; i < (5 + phasa); i++) {  
 summ = summ + Math.*abs*(Icos[i]) + Math.*abs*(Isin[i]);  
 }  
 currentDrag = summ / 2;  
 **return** currentDrag;  
 }  
  
  
 **public double**[] getDiffCurrent() {  
 **return diffCurrent**;  
 }  
  
 **public void** setVectors(Vector vectors) {  
 **this**.**vectors** = vectors;  
 }  
  
 **public void** setBlkSecondHarmonic(**double** blkSecondHarmonic) {  
 **this**.**blkSecondHarmonic** = blkSecondHarmonic;  
 }  
  
 **public void** setOd(OutputData od) {  
 **this**.**od** = od;  
 }  
  
 **public void** setBeginingDiffCurrent(**double** beginingDiffCurrent) {  
 **this**.**beginingDiffCurrent** = beginingDiffCurrent;  
 }  
  
 **public void** setBeginingDragCurrent(**double** beginingDragCurrent) {  
 **this**.**beginingDragCurrent** = beginingDragCurrent;  
 }  
  
 **public void** setCoefDrag(**double** coefDrag) {  
 **this**.**coefDrag** = coefDrag;  
 }  
  
 **public double**[] getCurrentDrag() {  
 **return currentDrag**;  
 }  
  
  
}

Объект класса OutputData принимает на вход значения дискретных сигналов. Также здесь реализована выдержка времени. При появлении сигнала срабатывания для каждого выключателя устанавливается false, что сигнализирует о его отключении.

**import** java.util.ArrayList;  
  
**public class** OutputData {  
  
 **private boolean trip** = **false**;  
 **private boolean str** = **false**;  
 **private boolean blk** = **false**;  
 **private double timeStep**;  
 **private double setTime**;  
 **private double time**;  
 **private** ArrayList<Breaker> **breakers** = **new** ArrayList<Breaker>();  
  
 **public void** takeDiscreteSignals() {  
  
 **if** (**trip** & !**blk**) { *//если срабатывание, то проходим выдержку времени (20-40 миллисекунд, чтобы удостовериться)* **time** = **time** + **timeStep**;  
 **if** (**time** > **setTime**) { *//проверка на выдержку времени* **breakers**.forEach(e -> e.setState(**false**)); *// приваиваем значение false для состояния выключателей* **time** = 0; *//сброс счетчика* }  
 }  
 }  
  
  
 **public boolean** getTripper() {  
 **return trip**;  
 }  
  
 **public void** setTripper(**boolean** trip) {  
 **this**.**trip** = trip;  
 }  
  
 **public boolean** getStr() {  
 **return str**;  
 }  
  
 **public void** setStr(**boolean** str) {  
 **this**.**str** = str;  
 }  
  
 **public void** setBlk(**boolean** blk) {  
 **this**.**blk** = blk;  
 }  
  
  
 **public void** setTimeStep(**double** timeStep) {  
 **this**.**timeStep** = timeStep;  
 }  
  
 **public void** setSetTime(**double** setTime) {  
 **this**.**setTime** = setTime;  
 }  
  
 **public void** setBreakers(ArrayList<Breaker> breakers) {  
 **this**.**breakers** = breakers;  
 }  
  
 **public boolean** isBlk() {  
 **return blk**;  
 }  
}

Объект класса Breaker принимает на вход сигнал срабатывания или не срабатывания.

**public class** Breaker {  
  
 **private boolean state** = **true**; *// true - включен* **public void** setState(**boolean** state) {  
 **this**.**state** = state;  
 }  
  
 **public boolean** isState() {  
 **return state**;  
 }  
  
}

**Результаты.**

1. Файл - KzB

На рис. 5 представлена работа алгоритма ДЗШ для 5 присоединений. На рис. 3 представлены осциллограммы аналоговых значений, а на рис. 4 дискретных значений. По ним видно, что при КЗ сначала сформировался сигнал блокировки по 2 гармонике и сигнал пуска, после этого сигнал блокировки исчез, и уже начал формироваться сигнал срабатывания защиты, но по осциллограммам аналоговых значений видно, что токи исчезли только через 40 мс после появления сигнала срабатывания, что связано с выдержкой времени – 2 периода.

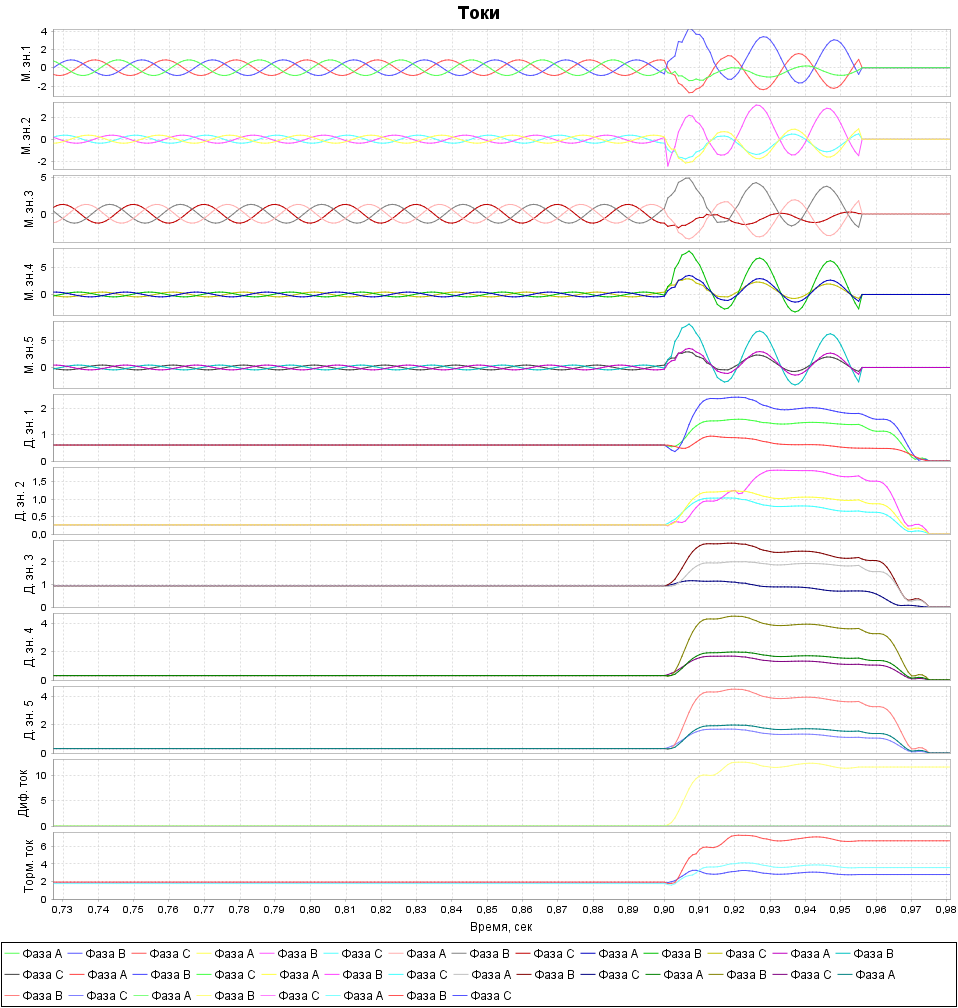


Рис. 3 Осциллограммы аналоговых значений KzB

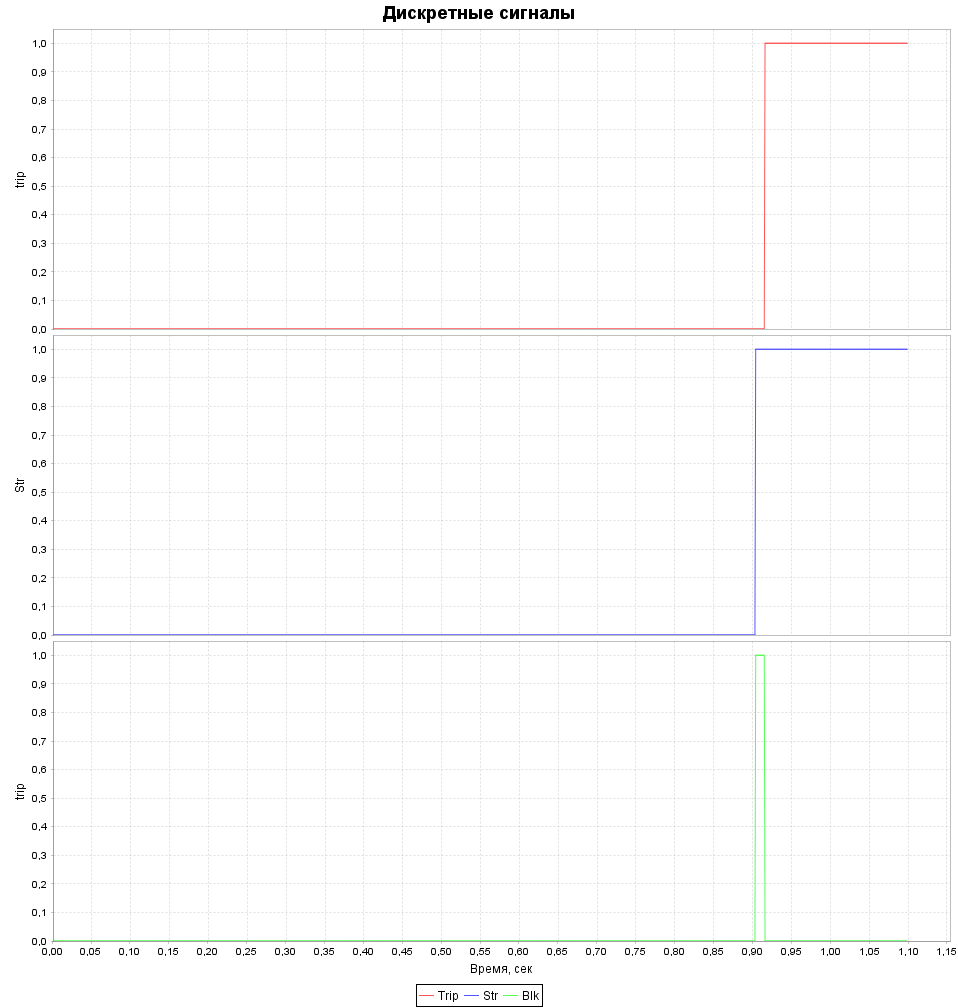


Рис. 4 Осциллограммы дискретных значений KzB

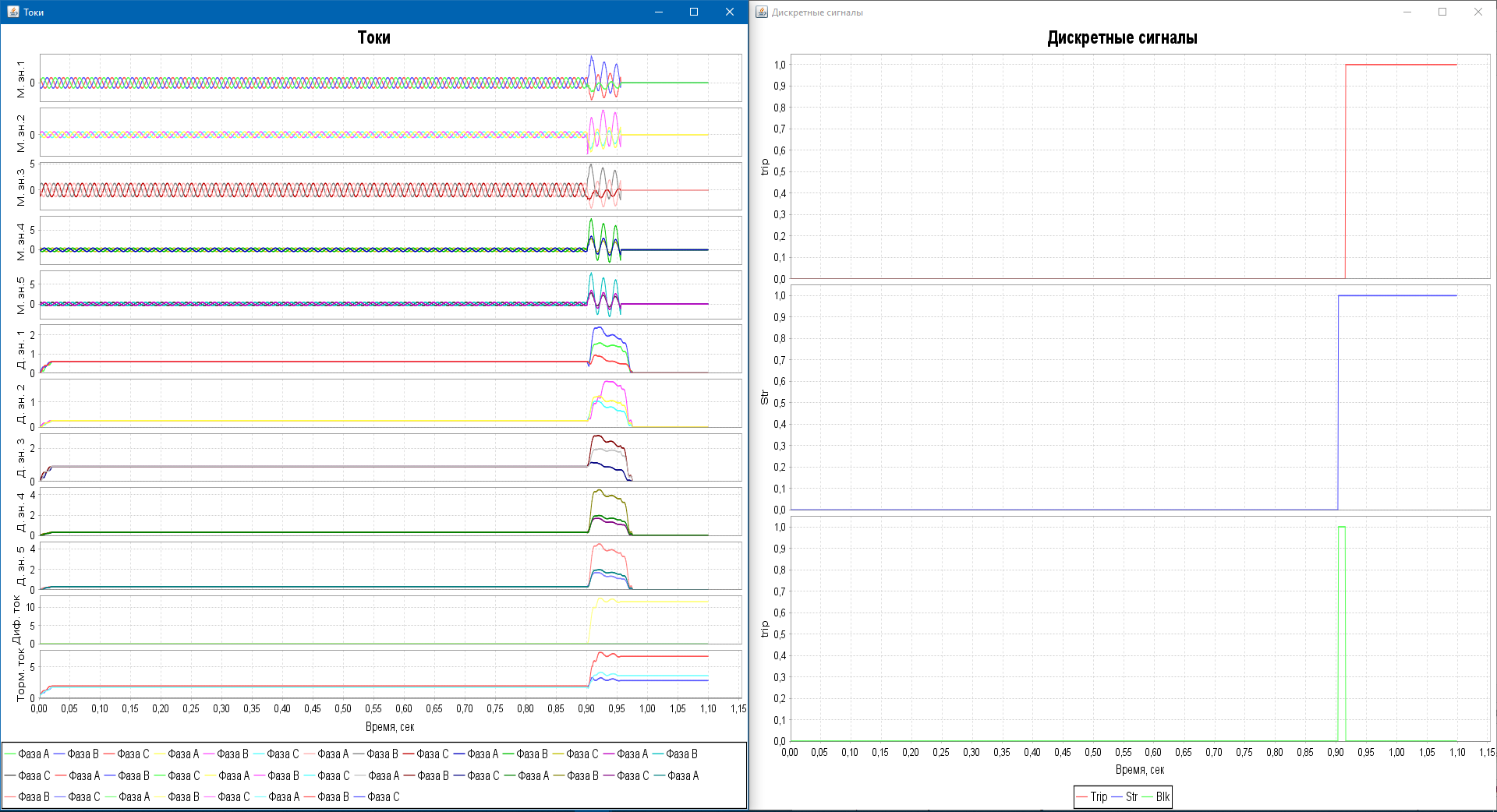


Рис. 5. Осциллограммы аналоговых и дискретных значений для файла KzB

1. Файл - KzAB

На рис. 6 представлены осциллограммы аналоговых значений. По ним видно, что данное КЗ является внешним КЗ, так как маленький дифф.ток на всех фазах и высокий тормозной ток. Соответственно, выходные сигналы у защиты не формировались.

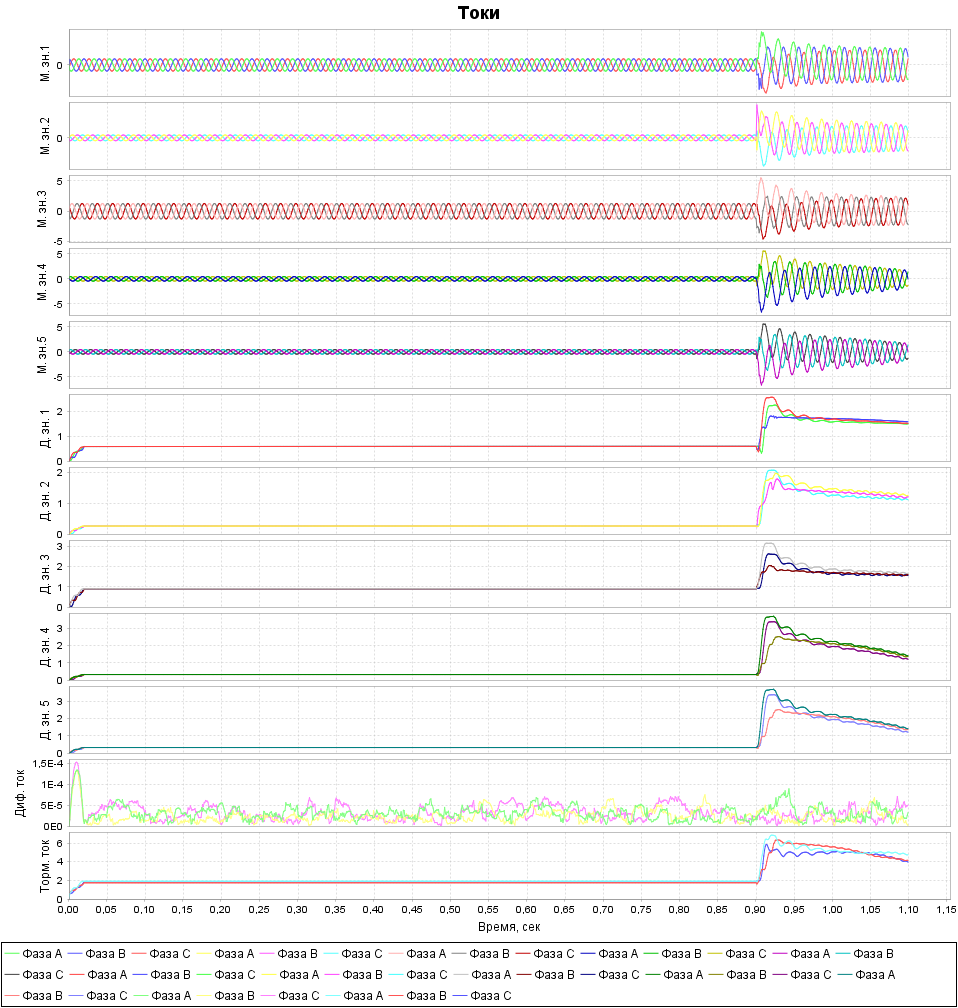


Рис. 6 Осциллограммы аналоговых значений KzAB

1. Файл - Vkl

На рис. 7 представлены осциллограммы аналоговых значений. По ним видно, что нет никакого КЗ, но есть изменение режима работы. Произошло включение первого присоединения. Соответственно, выходные сигналы у защиты не формировались.

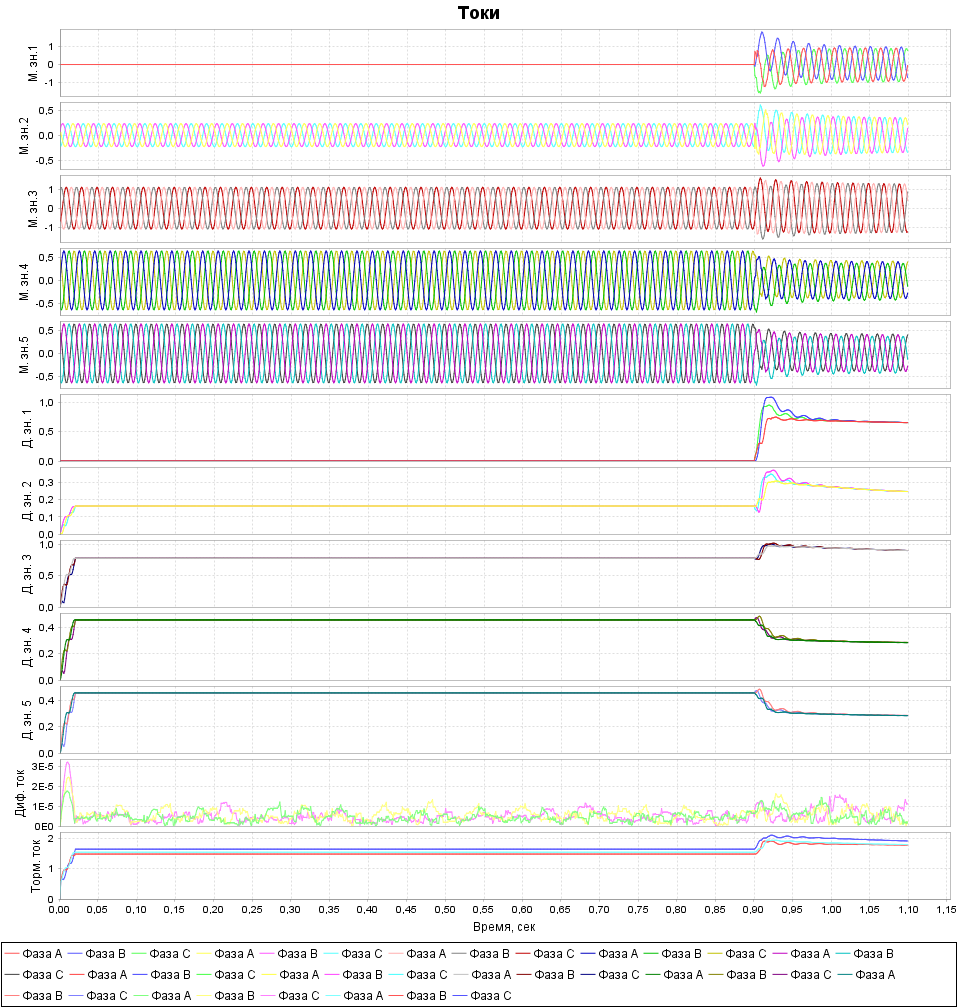


Рис. 7 Осциллограммы аналоговых значений Vkl