# Сравнение обнаружителей

Условием минимизации ущерба является соотношение [1]:

|  |  |
| --- | --- |
|  | (1) |

где *P* − вероятность правильного обнаружения повреждения (обнаружение повреждения при его наличии );

*F* − вероятность ложного обнаружения повреждения (обнаружение сигнала при его отсутствии );

 − безразмерный весовой множитель;

− стоимость ошибки отключения потребителя при отсутствии повреждения;

− стоимость неотключения повреждения;

− вероятности отсутствия и наличия повреждения соответственно.

Все три величины *F*, *P*,  (уровень порога срабатывания однозначно связан с ) выводятся друг из друга (см. ниже), поэтому, если известна одна из величин, можно рассчитать две другие по условию минимизации ущерба. Стоит заметить, что в выражении не фигурируют вероятности пропуска цели  и правильного не обнаружения , поэтому статистическая вероятность правильного обнаружения рассчитывается по выражению [2]:

|  |  |
| --- | --- |
|  | (2) |

где − число обнаружений сигнала при его наличии в шумах;

− общее число опытов (с присутствием сигнала в шумах).

Зададимся вероятностью менее одного ложного срабатывания в год. Тогда вероятность ложного срабатывания в окне данных

|  |  |
| --- | --- |
|  | (16) |

где − длительность окна данных в секундах.

Сравним два обнаружителя по условию минимизации ущерба:

- обнаружитель, основанный на расчете энергии принимаемого сигнала;

- обнаружитель, основанный на расчете коэффициента эксцесса;

## Сравнение обнаружителей

Для сравнения обнаружителей в условиях работы в шумах построим графики обнаружения. По оси ординат будет отложено отношение сигнал/шум:

|  |  |
| --- | --- |
|  | (11) |

где *T* − период наблюдаемого сигнала.

Известно, что математическое ожидание квадрата случайной величины , где  − это математическое ожидание. Для центрального гаусовского шума . Поэтому для известного сигнала с энергией  можно записать

|  |  |
| --- | --- |
|  | (12) |

## Обнаружитель на основе расчета энергии сигнала

Уровень порога может быть найден, если задан уровень ложных срабатываний в окне данных *F*:

|  |  |
| --- | --- |
|  | (7) |

где  − пороговый уровень энергии;

 − среднеквадратическое отклонение корреляционной функции , где − импульсная характеристика; − шум. Расчет  можно произвести на основе имитационного моделирования;

Чтобы обнаружитель правильно работал в условиях малого шума [1]

|  |  |
| --- | --- |
|  | (5) |

где  − энергия минимального расчетного сигнала;

Обнаружитель на основе расчета энергии годится для обнаружения сигнала в условиях «больших» шумов.

Второй способ расчета порога: порог (соответствующий условию минимизации ущерба) в условиях известного сигнала и белого шума определяется соотношением [1]:

|  |  |
| --- | --- |
|  | (3) |

где  − спектральная плотность мощности белого шума;

 − энергия известного сигнала, который необходимо найти в условиях шума.

Известно [2], что спектральная плотность мощности она связана с дисперсия шума (если математическое ожидание шума равно нулю)  . Для белого шума , поэтому . Для дискретного белого шума по теореме Котельникова можно записать . Таким образом, для дискретного белого шума , где − период дискретизации.

Зная порог срабатывания и расчетную (ожидаемую) дисперсию шума можно найти вероятности ложно и правильно срабатывания устройства обнаружения:

|  |  |
| --- | --- |
|  | (6) |

где  − функция ошибок;

 − уровень замеряемой корреляции.

## Обнаружитель на основе расчёта коэффициента эксцесса

Согласно центральной предельной теореме [2] шум в измерительных цепях электроэнергетических систем является центральным гаусовским. Поэтому в условиях отсутствия сигнала коэффициент эксцесса шума равен нулю:

|  |  |
| --- | --- |
|  | (8) |

где *e* − коэффициент эксцесса;

− математическое ожидание.

Несмещенная оценка коэффициента эксцесса по статистическим данным:

|  |  |
| --- | --- |
|  | (8) |

где  − выборочная (смещенная) оценка четвертого центрального момента;

 − выборочная (смещенная) оценка дисперсии.

При расчете в окне данных можно удобно пользоваться выражением, полученным на основе связи начального и центрального момента четвертого порядка (статистические начальные моменты всегда не смещенные):

|  |  |
| --- | --- |
|  | (9) |

Чтобы выбрать порог *e*0 посчитаем коэффициент эксцесса для множества реализаций шума и найдем по выборке {*e*} среднеквадратическое отклонение. Для возможности сравнения обнаружителей *e*0 подберем так, чтобы вероятность ложного срабатывания *F* была такой же, как и у сравниваемого обнаружителя:

|  |  |
| --- | --- |
|  | (10) |

где  − среднеквадратическое отклонение выборки {*e*}.

Таким образом, наиболее оптимальным обнаружителем будет тот, который соответствует меньшему ущербу по условию , т.е. тот обнаружитель, который имеет большую вероятность правильного обнаружения *P* при одинаковом *F* (однозначно связан с *F*).

## Имитационное моделирование

Для линии 220 кВ Семенов-Мантурово Нижегородской энергосистемы смоделируем однофазное короткое замыкание через переходное сопротивление 100 Ом в наиболее удаленной точке от точки установки устройства регистрации. Таким образом, энергия полученного импульса близка к минимальной. График импульса представлен на рисунке 1.

Рисунок 1 – Импульс напряжения (импульсная характеристика) при повреждении на линии 220 кВ Нижегородской энергосистемы Семенов-Мантурово

Дальнейшие расчёты приведены к первичному напряжению. Продолжительность импульса 15 мкс, энергия – 2362 Дж.

Проводились имитационные испытания однофазных коротких замыканий на разной удаленности с разными переходными сопротивлениями: форма импульса во всех случаях совпадает с представленным импульсом, отличается лишь амплитуда импульса.

Зададимся расчётным уровнем помех с 1,5% от номинального напряжения – 3,3 кВ. Считая закон распределения плотности вероятности шума нормальным, по правилу «трех сигм» найдем среднеквадратическое отклонение расчетного уровня шума (В) и дисперсию (В2):

|  |  |
| --- | --- |
|  | (13) |

Сделаем выборку 1000 реализаций шума и найдем:

- значение функции корреляции  (для дискретного сигнала ) для разных реализаций шума и рассчитаем дисперсию ;

- коэффициенты эксцесса для сигнала шума и среднеквадратическое отклонение коэффициента эксцесса .

Рассчитаем порог *z*0. Длительность ожидаемого импульса равна 15 мкс (зададимся окном данных в 2 раза больше времени ожидаемого импульса – 30 мкс)

|  |  |
| --- | --- |
|  | (16) |

Значения порогов, соответствующие минимальному ущербу и заданной *F*

|  |  |
| --- | --- |
|  | (16) |

Условие Дж выполняется.

Построим кривые обнаружения в условиях изменяющегося шума. Необходимая дисперсия шума для заданного соотношения сигнал шум

|  |  |
| --- | --- |
|  | (18) |

Для каждого уровня шума *D* проведено 100 000 опытов (помеси сигнал+шум) вычислены аварийные сигналы фильтров: корреляционная функция и коэффициент эксцесса

|  |  |
| --- | --- |
|  | (16) |

где − аварийный сигнал (помесь сигнал+шум); − шум. Данные выражения сравнены с порогами *z*0 и *e*0 соответственно и найдены вероятности правильного обнаружения сигнала обоими обнаружителями по выражению .

В ходе имитационного моделирования выяснено, что с увеличением окна данных уменьшается  , уменьшается необходимый порог *e0* и увеличивается вероятность правильного срабатывания *Pe*. Параметры корреляционного обнаружителя от размера окна не зависят (т.к. корреляционный интеграл вычисляется на интервале продолжительности сигнала).

## Результаты

Исходные данные:

Энергия расчетного минимального импульса – 2362 Дж,

продолжительность ожидаемых импульсов – 15 мкс,

импульсная характеристика – рисунок 1,

расчетное значение шума – 1,5% от номинального (3,3 кВ),

ширина окна наблюдения – 30 мкс.

Результаты имитационного моделирования получены с помощью программы detector (приложение А), вывод программы представлен на листинге 1.

|  |
| --- |
| Листинг 1 |
| time ./detector 1e-7 301 1100 signal.txt  usage: ./detector sample\_time data\_window\_in\_samples expect\_noise\_deviation /path/to/signal\_filename  Signal parameters...  Samples: 151  Duration: 1.5e-05  Mean: 37.4802  Max: 33932.9  Min: -17299.9  Deviation: 1.57444e+08  Excess: 1.04852  Energy: 2361.68  Expect-noise deviation, Volt: 1100  Window size, samples: 301  Window size, seconds: 3e-05  Deviation of expected-noise excess, o.e.: 0.0791836  Deviation of expected\_noise-to-signal cross corelation, Volt^2\*sec: 286.437  Count of false work in 1 year: 1  Probability of false work in one sampling window, o.e.: 9.54465e-13  Excess treshold e0, o.e.: 0.55753  Cross-correlation treshold z0, Volt^2\*sec: 2016.8  Calculate table...  q P\_e P\_z  ----------------------  0 0 0  0.0001 0.0384 0.50712  0.0002 0.03577 0.51291  0.0003 0.0365 0.51087  0.0004 0.03712 0.5148  0.0005 0.03639 0.51641  0.0006 0.03778 0.51874  0.0007 0.03766 0.51895  0.0008 0.03812 0.51995  0.0009 0.03797 0.52426  0.001 0.03848 0.51896  0.002 0.03712 0.5319  0.003 0.0377 0.53711  0.004 0.03693 0.54683  0.005 0.03781 0.55003  0.006 0.03738 0.55402  0.007 0.03716 0.55935  0.008 0.03782 0.56278  0.009 0.03726 0.56739  0.01 0.03766 0.57123  0.02 0.03698 0.59983  0.03 0.03817 0.62197  0.04 0.03846 0.63839  0.05 0.03916 0.65819  0.06 0.03913 0.67209  0.07 0.03961 0.68421  0.08 0.04094 0.69546  0.09 0.04103 0.704  0.1 0.04196 0.71295  0.2 0.05349 0.78841  0.3 0.07355 0.83402  0.4 0.10162 0.87236  0.5 0.13646 0.89718  0.6 0.18312 0.91755  0.7 0.23863 0.93266  0.8 0.30181 0.94591  0.9 0.37057 0.95558  1 0.44606 0.96192  2 0.94452 0.9941  3 0.99894 0.99915  4 0.99999 0.99983  5 1 0.99999  6 1 1  7 1 1  8 1 1  9 1 1  10 1 1  20 1 1  30 1 1  40 1 1  50 1 1  60 1 1 |

Характеристики обнаружителей:

оптимальный пороговый уровень обнаружителя (по коэффициенту эксцесса) – 0,558 о.е.,

оптимальный пороговый уровень обнаружителя (по энергии) – 2020 Дж.

Кривые обнаружения представлены на рисунке 2.

Рисунок 2 – Кривые обнаружения для обнаружителя на основе расчета коэффициента эксцесса (P\_e) и на основе расчета корреляции (P\_z)

Литература

1. Ширман Я.Д. Теоретические основы радиолокации // Учебное пособие для вузов. М, изд-во «Советское радио», 1970, стр. 560 т. 25000 экз.

2. Венцель Е.С. Теория вероятностей // М, изд-во «Наука», 1969, 576 с.

ПРИЛОЖЕНИЕ А

Программа detector написана на с++ и состоит из 4 модулей:

- main – код отвечающий за вывод параметров на консоль;

- get\_args – модуль получения входных параметров переданных в командной строке;

- core – расчетное ядро;

- noise\_generator – модуль расчета шума;

- detection\_table – модуль расчета таблицы вероятности правильного обнаружения.

Зависимости:

- boost::Random - используется генерации случайных чисел с нормальным законом распределения.

|  |
| --- |
| main.cpp |
| #include <fstream>  #include <iostream>  #include <noise\_generator.h>  #include <get\_args.h>  #include <core.h>  #include "detection\_table.h"  using namespace std;  void init(int argc, char \* argv[]);  void calculate\_tresholds(const double & noise\_sigma);  variables info;  int main(int argc, char \* argv[])  {  init(argc, argv);  cout << "Signal parameters... " << endl;  info.signal().print();  cout << endl;  cout << "Expect-noise deviation, Volt: " << info.expected\_noise\_sigma() << endl;  cout << "Window size, samples: " << info.window\_size() << endl;  double window\_duration = info.sample\_time()\*(info.window\_size()-1);  cout << "Window size, seconds: " << window\_duration << endl;  cout << endl;  calculate\_tresholds (info.expected\_noise\_sigma());  cout << "Excess treshold e0, o.e.: " << info.e0() << endl;  cout << "Cross-correlation treshold z0, Volt^2\*sec: " << info.z0() << endl;  //cout << "Maxinum-noise deviation for energy detector, Volt: " << sqrt(info.z0() / window\_duration) << endl;  cout << endl;  cout << "Calculate table..." << endl;  vector<double> q;  double start=1e-4;  q.push\_back(0);  q.push\_back(start);  for(int i=0; i<50; i++){  q.push\_back(q[i+1] + start\*pow(10,i/9));  }  detection\_table table( q, info.signal());  table.calculate(info);  table.print();  }  void init(int argc, char \* argv[]){  if(argc < 4){  cout << "usage: " << argv[0] << " sample\_time data\_window\_in\_samples expect\_noise\_deviation /path/to/signal\_filename" << endl;  exit(1);  }  info.set\_sample\_time (atof (argv[1]));  info.set\_window\_size (atoi (argv[2]));  info.set\_expected\_noise\_sigma (atoi (argv[3]));  samples \_signal = get\_signal (argv[4]);  \_signal.set\_sample\_time (info.sample\_time());  info.set\_signal (\_signal);  }  void calculate\_tresholds(const double & noise\_sigma){  const double mean = 0;  noise\_generator noiser(mean,noise\_sigma);  samples noise(info.sample\_time());  noise.set\_window\_size (info.window\_size());  samples e;  samples z;  for(int i=0; i<CYCLES; i++){  noiser.generate\_to (noise);  e.add( noise.excess() );  z.add( cross\_correlation\_0(noise, info.signal()) );  }  double sigma\_e = e.deviation();  double sigma\_z = z.deviation();  double F = 1 \* info.window\_size() \* info.sample\_time() / (24\*3600\*365);  cout << "Deviation of expected-noise excess, o.e.: " << sigma\_e << endl;  cout << "Deviation of expected\_noise-to-signal cross corelation, Volt^2\*sec: " << sigma\_z << endl;  cout << "Count of false work in 1 year, o.e.: " << 1 << endl;  cout << "Probability of false work in one sampling window, o.e.: " << F << endl;  info.set\_e0 ( treshold(F, sigma\_e) );  info.set\_z0 ( treshold(F, sigma\_z) );  } |

|  |
| --- |
| get\_args.h |
| #include <fstream>  #include <iostream>  #include "core.h"  using namespace std;  samples get\_signal(char \* filename){  samples signal;  ifstream in(filename);  if(!in){  cout << "File " << filename << " not found" << endl;  return signal;  }  double s;  while(in >> s){  signal.add(s);  }  in.close();  return signal;  } |

|  |
| --- |
| core.h |
| #ifndef \_CORE  #define \_CORE  #define CYCLES 1e5  #include <math.h>  #include <vector>  #include <boost/math/special\_functions/erf.hpp>  using namespace std;  double cross\_correlation\_0(const samples & X, const samples & Y);  double treshold(double F, double sigma);  class samples{  private:  double \_deviation() const;  double \_central\_moment4() const;  double sample\_time;  public:  vector<double> sample;  double mean() const;  double deviation() const;  double raw\_moment(const int level) const;  double excess() const;  double energy() const;  samples(const double & sample\_time);  samples(const double & sample\_time, int window\_size);  samples();  void set\_sample\_time(const double & v);  double get\_sample\_time() const;  void print() const;  void set(const vector<double> & data\_samples);  void add(double discret\_sample);  void set\_window\_size(int n);  samples& operator=(const samples&);  samples operator+(const samples&) const;  unsigned int window\_size() const;  double duration() const;  };  class variables{  private:  double \_sample\_time;  double \_window\_size;  double \_e0, \_z0, \_noise\_sigma;  samples \_signal;  public:  variables() {\_sample\_time = 0; \_window\_size = 0; \_e0 = 0; \_z0 = 0; \_noise\_sigma=0; }  void set\_sample\_time(const double & v) {\_sample\_time = v;}  void set\_window\_size(const double & v) {\_window\_size = v;}  void set\_signal(samples & s) {\_signal = s;}  double sample\_time() const {return \_sample\_time;}  double window\_size() const {return \_window\_size;}  const samples & signal()const {return \_signal;}  void set\_e0(const double & v) {\_e0 = v;}  void set\_z0(const double & v) {\_z0 = v;}  double e0() const {return \_e0;}  double z0() const {return \_z0;}  void set\_expected\_noise\_sigma(const double & v) { \_noise\_sigma = v;}  double expected\_noise\_sigma() {return \_noise\_sigma;}  };  #endif |

|  |
| --- |
| noise\_generator.h |
| #ifndef \_NOISE\_GENERATOR  #define \_NOISE\_GENERATOR  #include <boost/random/normal\_distribution.hpp>  #include <boost/random/lagged\_fibonacci.hpp>  #include <boost/random/variate\_generator.hpp>  #include <ctime>  #include "core.h"  using namespace std;  /\* generate normal distributed noise \*/  class noise\_generator  {  private:  boost::lagged\_fibonacci19937 rnd\_engine;  boost::normal\_distribution<double> distribution;  public:  noise\_generator (const double mean, const double sigma);  ~noise\_generator();  void generate\_to (samples & noise);  void set\_sigma(const double & sigma);  double sigma() const;  };  #endif |

|  |
| --- |
| detection\_table.h |
| #ifndef \_DETECTION\_TABLE  #define \_DETECTION\_TABLE  #include "core.h"  #include "noise\_generator.h"  using namespace std;  class detection\_table{  private:  const vector<double> q; // signal/noise  const samples signal;  void calc\_table\_row(noise\_generator & noiser, const variables & info);  public:  vector<double> Pe;  vector<double> Pz;  detection\_table(vector<double> & q, samples signal);  void calculate(const variables & info);  void print() const;  };  #endif |

|  |
| --- |
| core.cpp |
| #include <core.h>  #include <iostream>  using namespace std;  samples::  samples(const double & \_sample\_time):  sample\_time (\_sample\_time){}  samples::  samples():  sample\_time(0){}  void  samples::  set\_sample\_time(const double & v){  sample\_time = v;  }  samples::  samples(const double & \_sample\_time, int window\_size){  sample\_time = \_sample\_time;  sample.resize(window\_size);  }  double  samples::  get\_sample\_time() const{  return sample\_time;  }  double  samples::  mean() const{  double sum = 0;  int cnt=sample.size();  for(int i=0; i<cnt; i++){  sum += sample[i];  }  return sum/cnt;  }  double  samples::  raw\_moment(const int level) const{  double sum = 0;  int cnt=sample.size();  for(int i=0; i<cnt; i++){  sum += pow(sample[i],level);  }  return sum/cnt;  }  double  samples::  \_deviation() const{  double sum = 0;  int cnt=sample.size();  for(int i=0; i<cnt; i++){  sum += sample[i]\*sample[i];  }  return sum/cnt - pow(mean(),2);  }  double  samples::  deviation() const{  int cnt=sample.size();  return \_deviation()\*cnt/(cnt-1);  }  unsigned int  samples::  window\_size() const{  return sample.size();  }  double  samples::  \_central\_moment4() const{  double sum = 0;  double m = mean();  int cnt=sample.size();  for(int i=0; i<cnt; i++){  sum += pow (sample[i]-m, 4);  }  return sum/cnt;  }  double  samples::  excess() const{  int n = sample.size();  double a = (pow(n,2)-1)/(n-2)/(n-3);  double b = \_central\_moment4()/pow(\_deviation(),2) + ((double)6)/(n+1) - 3.0;  return a\*b;  }  double  samples::  energy() const{  if( !sample\_time ){  cout << "Can't calculate energy. Sample time is not set.";  return 0;  }  double sum = 0;  int cnt=sample.size();  for(int i=0; i<cnt; i++){  sum += pow (sample[i], 2);  }  return sum\*sample\_time;  }  void  samples::  set(const vector<double> & data\_samples){  sample = data\_samples;  }  void  samples::  add(double s){  sample.push\_back(s);  }  void  samples::  set\_window\_size(int n){  sample.resize(n);  }  samples&  samples::  operator=(const samples & right){  sample\_time = right.sample\_time;  sample = right.sample;  return \*this;  }  samples  samples::  operator+(const samples & second) const{  int f = sample.size();  int s = second.sample.size();  if(f>s){  samples result = \*this;  for(int i=0; i<s; i++)  result.sample[i] += second.sample[i];  return result;  }  else{  samples result = second;  for(int i=0; i<f; i++)  result.sample[i] += sample[i];  return result;  }  }  void  samples::  print() const{  double max = -1e6, min = 1e6;  for(int i=0, cnt=sample.size(); i<cnt; i++){  double v = sample.at(i);  if(v>max) max = v;  if(v<min) min = v;  }  cout << "Samples: " << sample.size() << endl;  cout << "Duration: " << duration() << endl;  cout << "Mean: " << mean() << endl;  cout << "Max: " << max << endl;  cout << "Min: " << min << endl;  cout << "Deviation: " << deviation() << endl;  cout << "Excess: " << excess() << endl;  cout << "Energy: " << energy() << endl;  }  double  samples::  duration() const{  return sample\_time \* (sample.size()-1);  }  double  cross\_correlation\_0(const samples & X, const samples & Y){  double sample\_time = X.get\_sample\_time();  if( sample\_time == 0 || Y.get\_sample\_time() != sample\_time ){  cout << "Can't calculate cross\_correlation(0) . Sample time is not set or X, Y sample times different.";  return 0;  }  double sum = 0;  int n = X.window\_size();  int m = Y.window\_size();  int cnt = (n<m)?n:m;  for(int i=0; i<cnt; i++){  sum += X.sample[i]\*Y.sample[i];  }  return sum\*sample\_time;  }  double  treshold(double F, double sigma){  return sqrt(2) \* sigma \* boost::math::erf\_inv(1-2\*F);  } |

|  |
| --- |
| noise\_generator.cpp |
| #include <noise\_generator.h>  using namespace std;  noise\_generator::  noise\_generator (const double mean, const double sigma)  : distribution(mean,sigma){  rnd\_engine.seed(static\_cast<unsigned>(std::time(0)));  }  noise\_generator::  ~noise\_generator() { }  void  noise\_generator::  generate\_to (samples & noise){  for(int i=0, cnt=noise.sample.size(); i<cnt; i++){  noise.sample[i] = distribution (rnd\_engine);  }  }  void  noise\_generator::  set\_sigma(const double & sigma){  distribution = boost::normal\_distribution<double>(distribution.mean(), sigma);  }  double  noise\_generator::  sigma() const{  return distribution.sigma();  } |

|  |
| --- |
| detection\_table.cpp |
| #include "detection\_table.h"  #include "noise\_generator.h"  using namespace std;  detection\_table::  detection\_table(vector<double> & \_q, samples \_signal):  q(\_q), signal(\_signal)  {}  void  detection\_table::  calculate(const variables & info){  noise\_generator noiser(0,1);  for(int i=0, cnt=q.size(); i<cnt; i++){  {  cout << "\r" << (int)100.0\*i/cnt << "%" << " ";  cout.flush();  }  double noise\_sigma = sqrt( signal.energy() / q.at(i) / signal.duration() );  noiser.set\_sigma (noise\_sigma);  calc\_table\_row(noiser, info);  }  cout << "\r \r";  }  void  detection\_table::  calc\_table\_row( noise\_generator & noiser, const variables & info){  static samples noise(info.sample\_time(), info.window\_size());  int cnt\_e = 0, cnt\_z = 0;  for(int i=0; i<CYCLES; i++){  noiser.generate\_to (noise);  samples signal\_with\_noise = signal + noise;  if( signal\_with\_noise.excess() >= info.e0() ) cnt\_e++;  if( cross\_correlation\_0(signal\_with\_noise, info.signal()) >= info.z0() ) cnt\_z++;  }  Pe.push\_back ( ((double)cnt\_e) / CYCLES );  Pz.push\_back ( ((double)cnt\_z) / CYCLES );  }  void  detection\_table::  print() const{  cout << "q\tP\_e\tP\_z" << endl;  cout << "----------------------" << endl;  for(int i=0, cnt=q.size(); i<cnt; i++){  cout << q.at(i) << "\t" << Pe.at(i) << "\t" << Pz.at(i) << endl;  }  } |