Preparation of a computer program for statistical analysis of searches for new particles at the LHC/ATLAS experiment

Anette Lauen Borg
University of Oslo
August 2002

Thesis presented for the Cand. Scient. degree
in Experimental Particle Physics

Abstract

The object-oriented (OO) programming style is becoming more and more popular, also among scientists. Several CERN computer libraries have been translated from the Fortran programming language to C++ recently, and it is expected that future analysis tools for particle physics experiments will be programmed using an OO language. This thesis describes the translation of the Fortran 77 `alrmc` program (written by A. L. Read) into C++. The program will perform statistical analyses of searches for new particles at the LHC/ATLAS experiment. The theory behind the program and its new, object oriented structure are explained, and tests are conducted to make sure that the C++ version of the program works.

Acknowledgements

First of all, I would like to thank my supervisor Alex Read for suggesting this assignment, and for his help and advice while it was still a work in progress. I would also like to thank the students here at the Department of Physics for helping and supporting me. Whenever I had a question, I could always find someone who could answer it. Among these students, I want to thank in particular my fellow particle physics cand. scient. student, Unni Fuskeland, for many interesting and enlightening discussions and conversations. Thanks also go to the experimental particle physics group for general help and support.

Contents

1 Introduction

The program `alrmc`, written by A. L. Read in the Fortran 77 programming language, is a tool for analysing data from particle physics experiments. It is particularly useful in searches for new physics where the statistics is low, the measurements bordering on the sensitivity limit of the measuring equipment. The term "low statistics" means that there are few background and/or signal candidates recorded, so that the high statistics approximations of "normal" analysis methods will be invalid.

Unfortunately, while the `alrmc` program is indeed very useful and has been used in analysing data from search experiments at LEP (Large Electron Positron collider)[16], the code of the Fortran 77 version has been rather difficult for the typical user to understand and apply. This has been a problem, especially since making changes to the original setup of the program has meant that the user has been forced to change large and important parts of the code. This comes about mainly because of the Fortran procedural style of programming.

Programs written in versions of the Fortran language older than Fortran 90/95, consist mainly of one single "block" of code. When run, such a program will progress in a linear way, steadily working through subroutine and function calls. There is a way of grouping and separating some variables from the main "block" by using common blocks, but mainly the program consists of one long file of code. The consequence is, as has already been mentioned, that if the users want to add to or take away from the code, or just make some changes to a feature, they have to make big and complicated adjustments.

Object oriented (OO) programming has become more and more popular, also in scientific programming projects. The advantages are many; some of them will be mentioned in a later chapter of this thesis. One of the main advantages, however, is that OO programs are modular. It is a lot easier to understand and to make changes to a program that is split into several independent parts than to a program where almost every bit of code is dependent on the others. Consequently, some of the main reasons for wanting an object oriented version of `alrmc`, from now on called `alrmc++`, are that it would be easier to use, understand, expand and develop. Also, the programs that the `alrmc` program might have to interact with (libraries, analysis tools etc), are now being translated from Fortran to C++. This process has already started at CERN [7] and DELPHI [18].

This thesis is concerned with the translation of the `alrmc` program from Fortran 77 to object oriented C++. The Fortran `alrmc` program provides the user with several ways of analysing data, represented by a number of Fortran subroutines. In this thesis only one of these analysis types, the "exclude_signal" of the Fortran version, is considered.

In addition to the translation and adaption of the Fortran program to OO C++, there has been a need for a graphical user interface. This feature might make the crucial first contact with the program easier, and will incorporate help functions so that the user will not have to turn to the code to find out what kind of input the program demands.

In the first part of this thesis some of the uses of the program are mentioned, and object oriented programming and the CL_s method are explained. This is the background material needed to understand how the `alrmc++` program works. Later in the thesis, I explain the structure and layout of the C++ version, and I give a detailed explanation

of how the physics and statistics theory is implemented in the program. I then test the `alrmc++` program to see if it is working, and if it reproduces the results of the Fortran `alrmc`. Finally, I discuss the new program and its future.

The thing to bear in mind is that this thesis is also meant to be a user guide to the `alrmc++` program. This has of course affected the structure and the contents of the thesis. In the Appendices, for example, I have included a user's guide on how to compile and use the program, and a description of the format of the input file expected by the program.

2 The uses of the `alrmc++` program

2.1 The DELPHI Higgs searches

The DELPHI experiment (DEtector with Lepton, Photon and Hadron Identification) at LEP conducted searches for the Higgs particle at centre of mass energies between 200 and 209 GeV. The experimental data consisted of very few observed candidates, and the conclusion was drawn at the end of the analysis that the data showed no evidence for a Higgs signal [1]. However, a 95% confidence level lower mass limit of 114.3 GeV/c^2 was set. The confidence level was estimated using the statistical method the `alrmc` program is based on.

2.2 NOMAD

The neutrino oscillation $\nu_e \rightarrow \nu_\tau$ search at the NOMAD (Neutrino Oscillation MAgnetic Detector) detector [3] has found only a small number of candidates. The results consist of several different decay channels, each with very little, if any, observed data. These data have been analysed using another method than the one used in the `alrmc++` program. When comparing the results of the method of the published article [3] with the results we get when the same data is fed into the `alrmc++` program, we see that the results differ. This case will be discussed further in Chapter 7.

2.3 The ATLAS project

At CERN, the European organisation for nuclear research, the LHC (Large Hadron Collider) is presently under construction in the existing LEP tunnel. Some of the prospects of this new machinery is to increase the present day centre-of-mass energies and luminosities for the pp and heavy ion collisions that the LHC will provide.

The LHC project will include four large experiments. The ATLAS (A Toroidal Lhc ApparatuS) and CMS experiments will be doing precision measurements and searches for new physics. LHCb will be dedicated to the physics of b hadrons and CP violation, and ALICE will be a heavy ion experiment.

The LHC is the largest, most complex and expensive particle physics project so far. What do people expect to learn from the LHC experiments that will justify these costs? The physics motivations are many; physicists wish to perform more precise measurements, to understand the origin of the particle masses, to look for new physics beyond the Standard Model and to answer many of the questions left open by earlier experiments. ATLAS in particular will continue the ongoing searches for new physics. This includes searching for the Standard Model Higgs boson, particles predicted by the Super symmetry (SUSY) theory and other physics beyond the Standard Model. At ATLAS, the first few years of running will be a period of low luminosity, with few events produced. In this period, the `alrmc++` program may be used as an analysis tool in searches for particle signals.

Another example where the `alrmc++` program might be useful, is in search experiments where the background is small but non-zero, and the particle is very heavy and

thus not produced in great quantities, producing a small signal. The search for the heavy Z' is an example of such an experiment.

3 Object-oriented programming: A quick overview

Object-oriented (OO) designs are becoming more and more popular, but the transition from languages like Fortran and C to the OO languages of Java and C++ can be difficult. The idea of objects as "black boxes" that take care of themselves and interact via messages only can seem strange and foreign to many programmers not used to OO programming. However, the basics are quite simple once you have grasped the concepts of classes and inheritance. To explain these terms, I will start by describing objects.

3.1 Objects

Objects have both a behaviour (they do things) and a state (that is changed when they do things). For example, a cat could be an object. It has a state; it could be awake or sleeping, and it has a behaviour; falling asleep, which changes its state from awake to sleeping [10]. To make a cat-object sleep, we would need to send it the message "fall asleep". From our point of view, the existence of this message would be all we needed to know about the object. We would not need to know about all the complex details of how it falls asleep, that is, closing its eyelids, changing its breathing and so on.

3.2 Classes

Now we have a domestic cat that is able to fall asleep. But what if we wanted something more exotic, like a leopard? We make a new object called "leopard". It can also fall asleep, and it has a state, let us call it "awareness", that can be "asleep" or "awake", just like the cat. But our leopard is bigger and its fur has a different pattern. So we add two more states, usually called variables, to our leopard object; size and pattern. But of course, the domestic cat has a size and a pattern too. We see that the states and the behaviours of the cat and the leopard are the same, so in order to save time and make things neat and tidy, we would try to make a common set of states that could be specified for each object. In other words, we would abstract out the common attributes, ignore the particular values of these attributes and make a blueprint for our objects. This abstraction, or blueprint, is called a class. A class describes a set of objects that share a common structure and a common behaviour[1]. So let us make a class for our objects called "Felidae", which is the Latin name of the cat family. This class contains the variables "awareness", "size" and "pattern", and also the behaviour "fall asleep". If we want to make a new object, for example a lion, we use the Felidae class and simply fill in the particular values of the lion. An object is also called an *instance* of a *class*, meaning that the lion is an instance of the class Felidae.

3.3 Methods

To change a variable, we must send a message to the object activating its behaviour. The behaviour is called a "method" (known to Fortran programmers as a function or subroutine) and the process of sending a message is referred to as calling a method. Objects interact and communicate by calling each other's methods. A method can receive values or return a value when it is called (or both), but there are no differences

between a method with or without these options. This is different from the Fortran programming language, where a method is called a subroutine or a function depending on its characteristics.

3.4 Inheritance

To continue our real world example of animals, what if we wanted a human object instead of a cat? We could make a class Human, and make the objects Peter and Anne. These objects would have basically the same variables and methods as the cats, but there would be some differences as well. The number of legs, for example, and the humans would have less fur and round pupils. To save ourselves from a lot of work, it would be nice to be able to make a class called "Mammal" that would summarise all the common features of humans and cats, and then create the Human and Felidae classes as *sub classes* of Mammal. This would mean that we could reuse the code written in this parent, or super, class.

Our structure now looks like this: We have a super class containing the variables and methods of all mammals, and two sub-classes that specify the particularities of humans and cats with their own variables and methods. When we make an object of a sub-class, we can insert the specific information of that individual into the object's variables.

3.5 Pointers

This gives us a structure with lots of unorganised objects that are just floating around. How should we best organise and access these objects? The answer is pointers. When declaring an object, you can also make a pointer to it that can easily be stored in some kind of table, array or vector. If the pointers are stored in an iterative device, it will be easy to access all the objects using a loop. A pointer is, as the name suggests, something that "points" to the desired variable or object. Having these pointers, it *is* possible to access the variables of the objects directly from outside the object. However, it is considered more object oriented to make methods that simply return or set the desired variables. The advantage of making such "get" and "set" methods is that if you want to change the inner workings of a class, you can do that without changing what the user sees from the outside.

3.6 Visualisation

The most efficient way of providing information about the structure of an OO program, is to make a graphical representation. The parts needed to make such a structure map is shown in Figure 1. The classes and their objects are usually connected with a straight line to show which object belongs to which class. The pointers are connected to what they are pointing at by an arrow.

The visualisation of an OO structure usually does not show all the objects' methods and variables, only the parts that are necessary to understand the structure of the program.

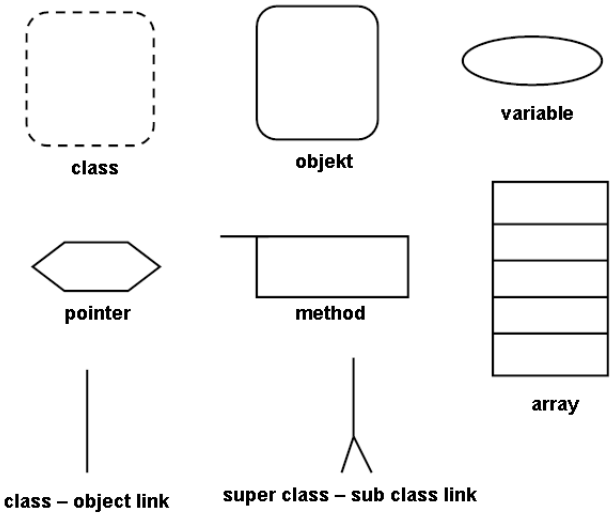

Figure 1: The parts of an OO structure.

3.7 Example : The Forest

As an example, let us consider a forest. By definition a forest contains many trees. Each tree has its own height, leaves and so on. By making a class Tree that contains all these variables, we create a forest of three tree-objects using the class as a blueprint. For each of the trees, we make an object of class Tree and assign the tree's specific values to its variables. This example is illustrated in Figure 2.

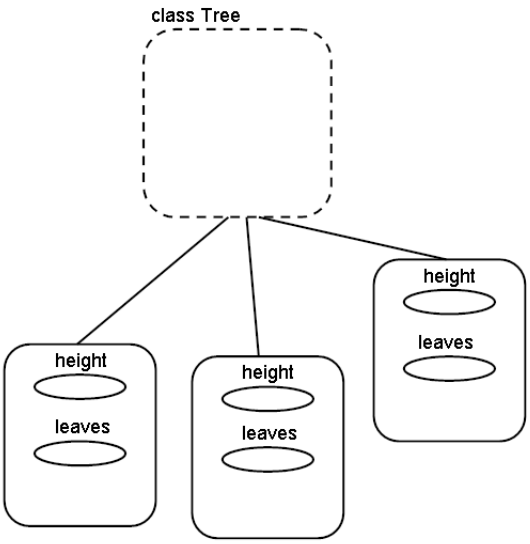

Figure 2: OO structure of example Forest.

3.8 Slightly More Complicated Example : Friends

Anne has a lot of friends. They all have a name, a date of birth, a telephone number, an address and so on, and Anne is having problems remembering these names and numbers. She wants to make a register containing all this information.

Her first decision is to make an object for each of her friends. She makes a blueprint, a class, called Friend. This class contains all the variables of a friend; name, address, etc. It has methods to set and get these variables from outside the object. She also makes another class called Register that will have only one object from which the Friend objects will be organised. This has to do with the concepts of OO. The organisation of objects and other structures could easily be done, for instance, from inside a main() method. But by putting all the code inside objects, we get a program that is easy to change later and that looks like a "black box" when seen from the outside.

Making an object of class Friend from inside the object of Register, Anne makes a pointer to the Friend object as well. The making of pointer and object could look like this (C++):

```
Friend *myFriend = new Friend();
```

Let us have a look at this expression. "Friend" is the name of the class that we are making an object of. "myFriend" is the name of the new object variable. The "$*$" means that "myFriend" is not only a name, but also a pointer to the object. The right side of the equation means what it says: We are making a new object, or instance, of the class Friend.

The pointer to the new Friend object can now be stored in a location of an array. When Anne wants to access the objects, she can easily loop through the array from inside the Register object.

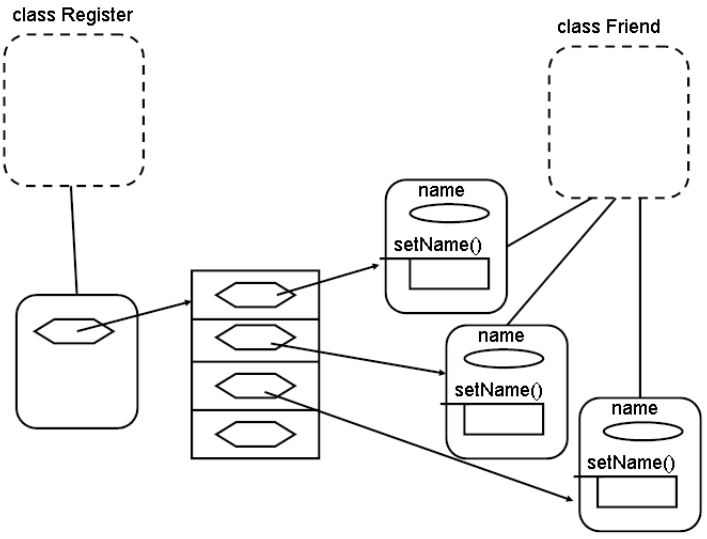

Figure 3: The OO structure of example Friends. Only one of the variables and one of the methods of the Friend objects are shown.

3.9 Advantages

Some of the advantages of OO-programming are:

- Modularity; the source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give an object to someone else, and it will still work.
- Information hiding; an object has an interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the inner workings of an object in order to use it.
- Inheritance provides specialised behaviours in addition to the common variables and methods provided by a superclass. Through the use of sub classes, programmers can reuse the code in the superclass many times.
- Type safety; when a method is called with arguments in the C++ and Java languages, it is required that the argument types (integer, double precision, character, etc) must match the ones of the method that is being called.

3.10 Disadvantages

The most noticeable disadvantage of employing an OO structure is that the program may be slower than, say, a Fortran program. This problem can be minimised by optimisation of the code, but the fact remains that if speed is the important thing, then OO programming may not be what you are looking for. However, the advantages of the previous paragraph mostly outweigh this factor.

Another disadvantage has to do with the fact that OO programming, and the programming languages that are adapted to it, are relatively new and still under development. The consequence is that there are few really good books on the subject covering the latest features and the more specialised options. Also, the compilers are not as optimised as, for example, modern Fortran compilers.

4 The CL_s method

The CL_s method is a statistical method that has been used to analyse data from the experiments at LEP [16]. It is based on a likelihood ratio Q, and the confidence levels CL_{sb} and CL_b, all explained later in this text.

4.1 The statistical concepts

Many physics experiments are conducted to test the validity of a theory. This means that the theory must include an observable or a parameter that can be measured directly or indirectly, respectively, in an experiment. A simple observable of a search for a new particle would be the number of detected candidates matching some predefined criteria.

In the language of statistics, an analysis of search results can be done as a hypothesis test. The null hypothesis is that there is no new particle, no *signal* (only background) and the alternate hypothesis says that there is. To reject one of these hypotheses, we will need rules to rank the experimental results from the least to the most signal-like. This can be accomplished by defining a test-statistic, or function of the observables and model parameters (particle mass, production rate, etc) of the known background and hypothetical signal [16]. Having ranked an ensemble of Gedanken experiments, we use them to reject or accept the null hypothesis by defining ranges of the values of the test-statistic. These are called rejection and acceptance regions respectively. This is done in such a way so that we minimise the possibility that we accidentally reject the null hypothesis when it is correct (type I error), or keep it when we should have rejected it (type II error).

To summarise; a test of the null hypothesis is a course of action specifying the set of values of a random variable called the test-statistic for which the null hypothesis is to be rejected. The set of values for which the null hypothesis is to be rejected is called the rejection region of the test [5].

4.2 The likelihood ratio

The test-statistic (called Q) of the type of search experiments we are interested in, is defined as the likelihood ratio. The likelihood ratio is the ratio of the probability densities for the two alternate hypotheses for an experimental result, $\frac{\mathcal{L}(s+b)}{\mathcal{L}(b)}$. If an experiment consists of N_{chan} independent channels, the total likelihood ratio is a product of the channel likelihood ratios. A channel, as defined by `alrmc++`, is a particle interaction resulting in a specific end product. For an experiment where events are both counted and have a distinctive measured property, the likelihood ratio can be written as:

$$Q = \frac{\prod_{i=1}^{N_{chan}} \frac{\exp^{-(s_i+b_i)}(s_i+b_i)^{n_i}}{n_i!}}{\prod_{i=1}^{N_{chan}} \frac{\exp^{-b_i} b_i^{n_i}}{n_i!}} \frac{\prod_{j=1}^{n_i} \frac{s_i S_i(x_{ij}) + b_i B_i(x_{ij})}{s_i+b_i}}{\prod_{j=1}^{n_i} B_i(x_{ij})}, \tag{1}$$

which can be simplified to

$$Q = e^{-s_{tot}} \prod_{i=1}^{N_{chan}} \prod_{j=1}^{n_i} \left(1 + \frac{s_i S_i(x_{ij})}{b_i B_i(x_{ij})}\right), \tag{2}$$

where n_i is the number of observed candidates in each channel, x_{ij} is the value of the discriminating variable measured for each of the candidates, s_i and b_i are the number of expected signal and background candidates per channel and s_{tot} is the total number of signal candidates for all channels. S_i and B_i are the probability distribution functions (p.d.f.'s) of the discriminating variable for the signal and background of channel i [16]. If the p.d.f.'s for the discriminating variable are identical for signal and background, or if they are not measured, the likelihood ratio can be simplified further to

$$Q = e^{-s_{tot}} \prod_{i=1}^{N_{chan}} \left(1 + \frac{s_i}{b_i}\right)^{n_i}. \tag{3}$$

If we need to find the value of Q numerically, the fact that the likelihood ratio can be computed by counting weighted candidates will prove useful. We can write $\ln Q$ as

$$\ln Q = -s_{tot} + \sum_{k=1}^{n} n_k w_k, \tag{4}$$

where n is the total number of candidates observed in all channels, and the weight w_k of each candidate is

$$w_k = \ln\left(1 + \frac{s_k S_k(x_k)}{b_k B_k(x_k)}\right). \tag{5}$$

A much used function of the likelihood ratio is $-2\ln Q$. In the high statistics limit the probability density distribution of this function is expected to converge toward the $\triangle\chi^2$ p.d.f. However, the $-2\ln Q$ p.d.f. is not always given analytically, meaning that it must be constructed using Monte Carlo simulations. In Figure 4, an example of the p.d.f.'s of $-2\ln Q$ for the signal+background and the background hypotheses are shown.

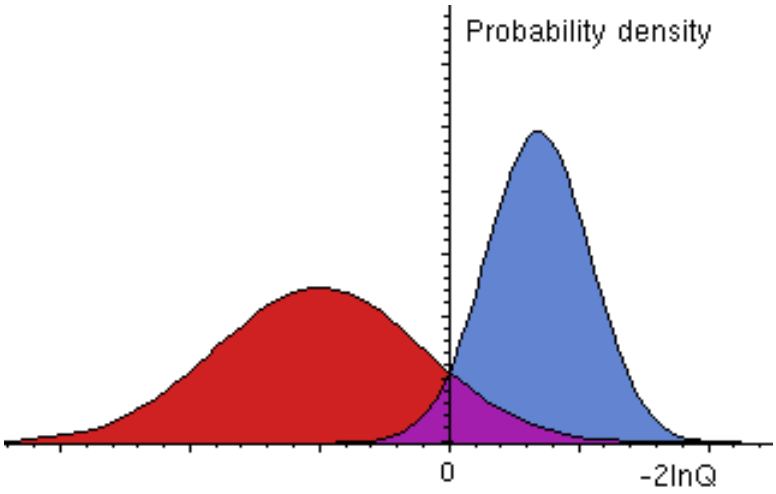

Figure 4: An example of distributions of $-2\ln Q$ for the signal+background (red) and background (blue) hypotheses.

4.3 Confidence levels

An answer of "true" or "false" to a hypothesis test will not be of much use if we don't specify the significance of the rejection or acceptance. The significance is expressed in the terms of a confidence level (CL). This value tells us the probability that the true value of the test-statistic lies within a certain region called the confidence interval. In our case, this interval could be the acceptance or rejection regions. To find the confidence level, it is necessary to compare the test-statistic for the observed values of an experiment to test-statistics obtained theoretically, where the latter should have a set of acceptance and rejection values specified.

The procedure of Chapter 4.2 of finding the likelihoods makes it easy to calculate the confidence levels of rejection and acceptance. According to the CL_s method, the confidence in the signal + background hypothesis is defined as the probability that the real value of Q lies in the interval from $-\infty$ up to and including the value of the experimental value of the test-statistic, Q_{obs}, given that the signal+background hypothesis is true. Thus the confidence in the signal+background hypothesis can be written as:

$$CL_{s+b} = P_{s+b}(Q \leq Q_{obs}) \tag{6}$$

where

$$P_{s+b}(Q \leq Q_{obs}) = \int_{-\infty}^{Q_{obs}} \frac{dP_{s+b}}{dQ} dQ. \tag{7}$$

Note that $\frac{dP_{s+b}}{dQ}$ is the p.d.f. of the test-statistic for signal+background experiments. The confidence in the background-only hypothesis is defined as

$$CL_b = P_b(Q \leq Q_{obs}) \tag{8}$$

and the confidence in the signal hypothesis is given as

$$CL_s \equiv \frac{CL_{s+b}}{CL_b}. \tag{9}$$

This is not a "real" confidence, but a ratio of confidences that is an approximation to the confidence in a "signal only" hypothesis. The signal hypothesis is considered excluded at confidence level CL where

$$1 - CL_s \leq CL. \tag{10}$$

In Equation (4), we saw that $\ln Q$ could be expressed as a sum of weighted candidates. Numerically, it is much less time consuming to compute this sum than the product of Equation (2). From the definition of CL_{s+b}, we see that $P_{s+b}(Q \leq Q_{obs}) = P_{s+b}(\ln Q \leq \ln Q_{obs})$, which enables us to use the value of $\ln Q$ directly in our calculations.

In Figure 5, the confidence levels are displayed graphically. From the definition of CL_{s+b}, Equation (7), we see that the integration of the p.d.f. of Q has integration limits from $-\infty$ to the observed value of Q. Since Q is a function that increases for increasingly signal-like experiments, $-2\ln Q$ must have the opposite characteristic. This means that integration is now performed from the observed value to ∞ as can be seen in Figure 5, and Equation (6) can be written as

$$CL_{s+b} = P_{s+b}(-2\ln Q \geq -2\ln Q_{obs}). \tag{11}$$

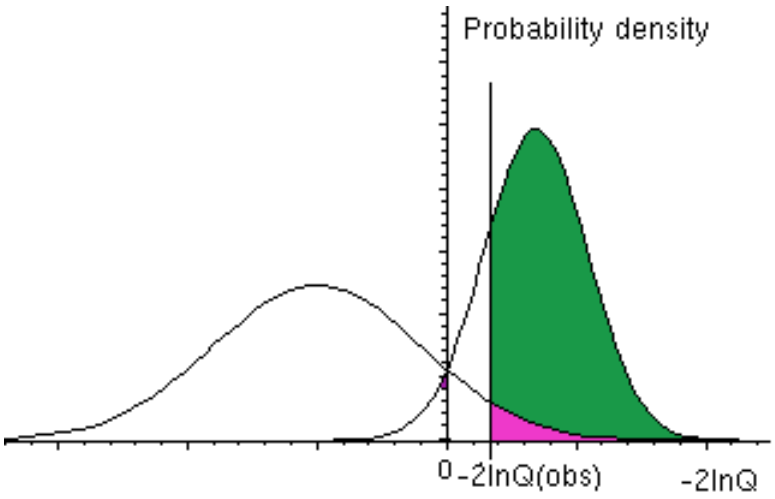

Figure 5: The distributions of Figure 4, with an observed value of $-2\ln Q$. CL_{s+b} lies in the pink area and the green+pink area shows CL_b.

5 The structure of the `alrmc++` program

5.1 The OO design

The basic layout of the `alrmc++` program is rather simple. The idea was to make an object for each channel entered by the user. The concept of a "channel", defined in Chapter 4, should be explained further here by an example. The process $Z^0 \to e^+e^-$ is a channel, and $Z^0 \to \mu^+\mu^-$ is another. If we choose to ignore the light lepton flavour, these two channels can be combined into the new channel $Z^0 \to l^+l^-$, where the "l" is short for "lepton".

To make an object for each channel, it is necessary to make a class Channel as a channel object blueprint (see Chapter 3 on OO programming). The channel objects contain a lot of variables and methods. The latter are mostly to get or set variables from outside the objects. Pointers to these objects are stored in a array called "channels".

It is the channel information that is analysed by the program. The methods executing this analysis are contained in the classes Analysis and Exclude. Class Analysis is a super class containing all methods that will be used by more than one analysis type. The only analysis type implemented so far is the analysis "Exclude". The Exclude class is a sub class of class Analysis, which means that it inherits all the methods and variables of Analysis. It also contains the specific methods of the analysis type (expressed in the Fortran subroutine exclude_signal). Inside class Exclude there is a pointer to the "channels" array.

There is one more class in this structure; class Histogram. This is the class associated with Root, an object oriented data analysis framework developed at CERN. The Histogram class uses the Root libraries to make a TTree [6] in which the final results of the analysis are stored. The TTree is a structure similar in many ways to an ntuple, a well-known data structure to users of PAW [8]. After the TTree has been filled, it is written to a file with the extension *.root*. This file can be opened in the Root framework and the contents viewed as histograms. The structure of the C++ program is displayed in Figure 6.

5.2 The user interface

The user interface is written in Java. The class controlling most of the interface is class JavaCpp, and the other four classes also contributing are the classes Welcome, Analysis, Help and About. The object of JavaCpp contains the main interface frame, which uses the objects of the other classes to display various GUI (Graphic User Interface) components. There is also a class AnalysisJava that takes care of the interaction between the Java interface and the C++ program. This class plays a major role when the user provided input is transfered from the Java interface to the C++ program. Figure 7 shows the interface window and Figure 8 shows the structure of the Java interface program.

5.3 How does it all work?

The structure of the Java interface and the connection to the C++ program is a little complicated. The Java file containing the Java "main()" method is AnalysisJava.java and

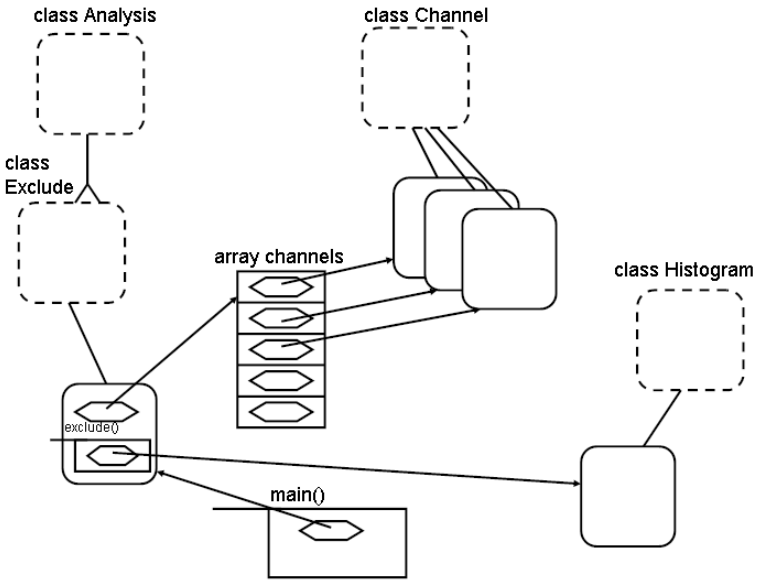

Figure 6: The OO structure the `alrmc++` program.

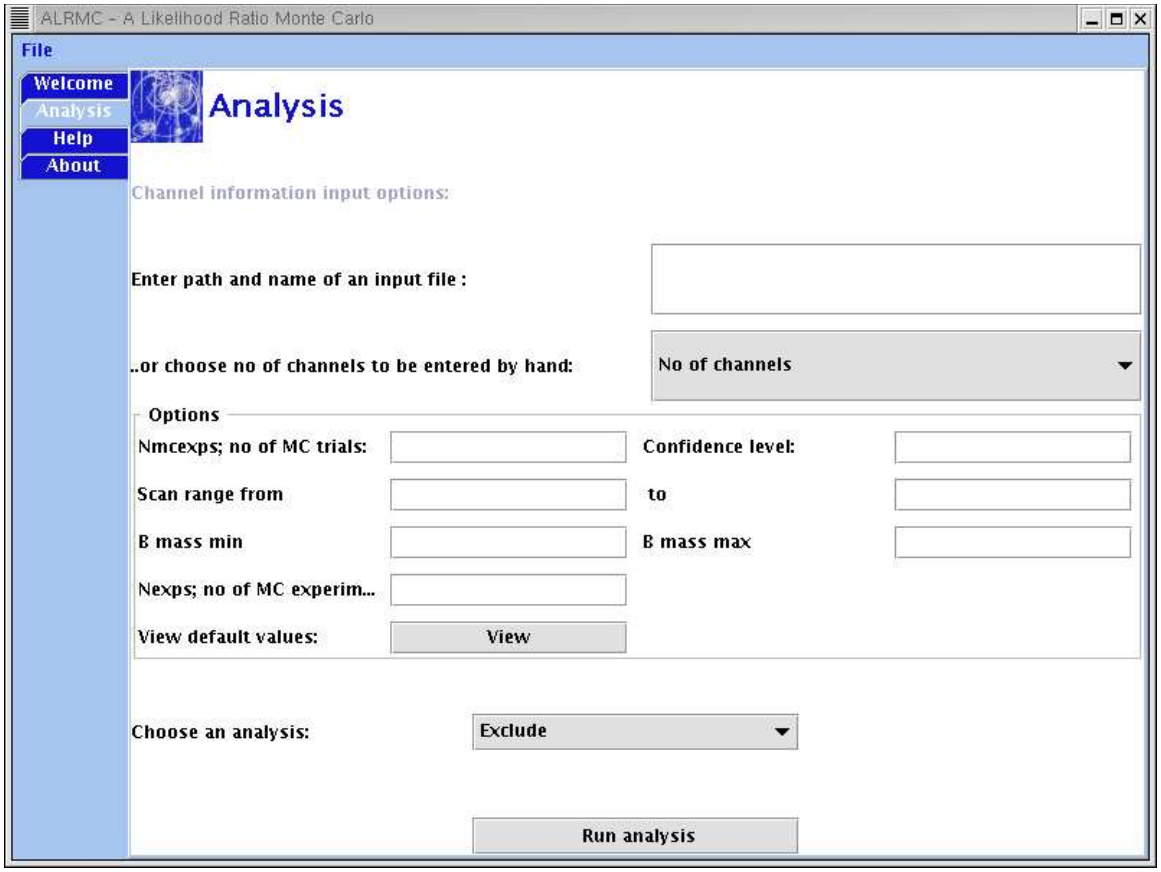

Figure 7: The Java user interface of the `alrmc++` program

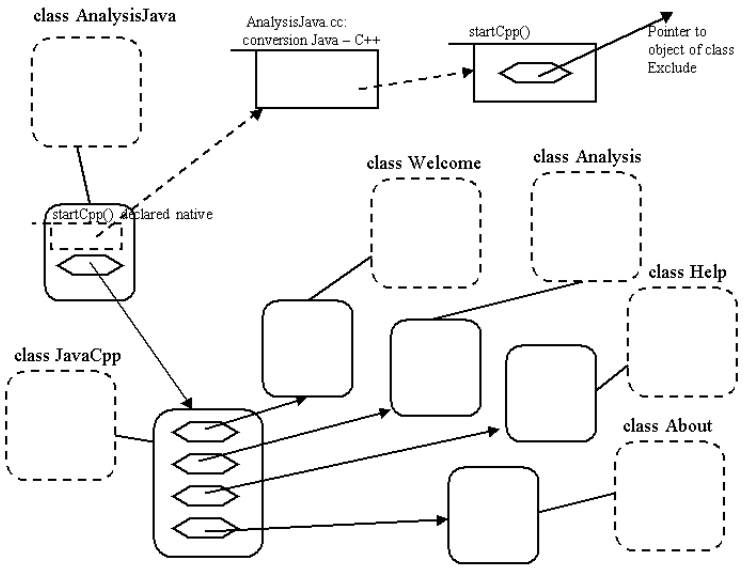

Figure 8: The OO structure of the `alrmc++` Java interface.

the role of the class contained in this file will be explained later. The method "main()" makes an object of class JavaCpp. As this object is created, its constructor puts together the GUI components in the frame object and displays it all on the screen. The user will see a window pop up on the monitor. This window is divided into sections. There is a menu bar on top, with a "File" drop down menu, and a main area where the actual GUI components are displayed. This area consists of a so-called Tabbed Pane. By clicking on the tabs on the left side of the main area, objects of classes Welcome, Analysis, Help and About are created and displayed. The object of class Analysis provides the user with a way of feeding information into the program such as the name and path of an input file.

To use both the Java and the command line interface, the user must provide some information about the channels he or she wants to analyse. There are two ways of feeding this input into the program. The standard way is to make a file of a fixed format that will be discussed later (see Appendix B). Both the command line and the Java interface ask for the name and path of such a file. The Java interface also has a pop up window option where the user can fill in a form to provide the channel information (see Figure 9).

When the user has provided all necessary information, the computing part of the program can begin. In the command line version this is done by pressing "enter", and in the Java version by clicking a "Run Analysis" button. The command line version calls the method "exclude()", in the object of class Exclude, directly. The Java interface version calls a method in class Analysis that splits up the tasks of allowing the interface to be used and running the analysis into two different "threads", or sequential flows of control. These two threads will run independent of each other and at the same time. The priority of the interface thread is set to a higher value than that of the analysis to stop the interface from "freezing" while the C++ program runs.

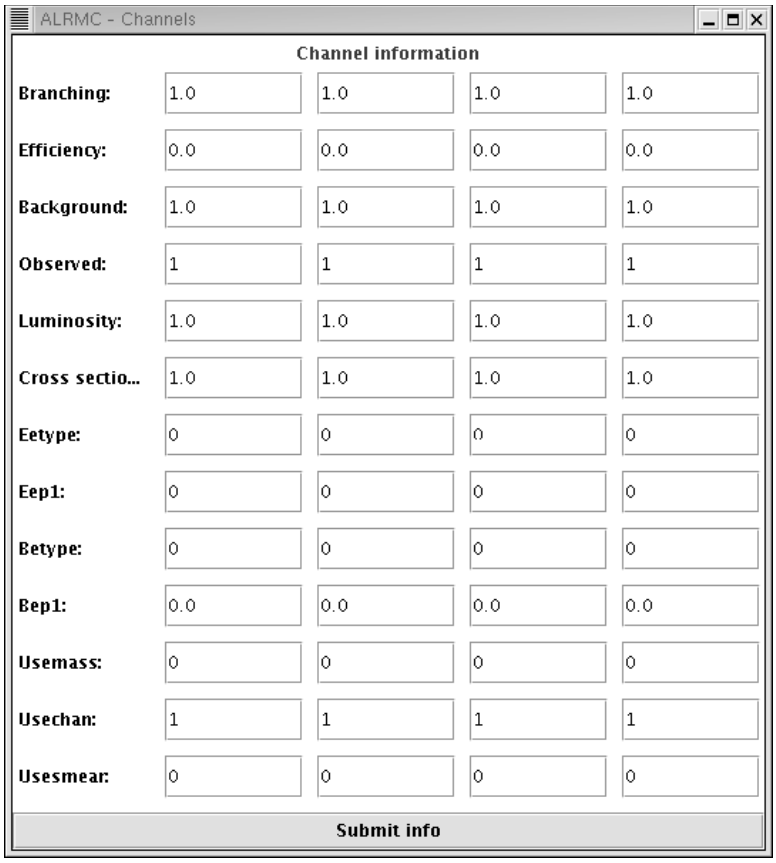

Figure 9: The pop up window where the user can fill in the channel information, one column for each channel.

The Java thread running the C++ program starts with a call to the method "run()" in the object of class AnalysisJava. The class AnalysisJava declares the C++ method "startCpp()" native, so that the method "run()" can call it. "startCpp()" itself is contained in the file cppJava.cc and is called by "run()" via the "conversion implementation" in file AnalysisJava.cc. The implementation accesses the data contained within the Java strings and passes it to the corresponding C++ structure (const char*). The "startCpp()" method makes an object of class Exclude and calls its method "exclude()".

Using the Java interface is optional and the file main.cc provides a command line interface. This is a file containing a C++ "main()" method that writes some output to screen, asks for input and makes an object of class Exclude.

This is where the numerical computations of the analysis starts for both the Java and the command line interface versions. This procedure, and the coupling to the theory of Chapter 4, will be described in the next chapter.

6 Coupling to theory

6.1 What does the analysis do?

The only analysis type available at the moment is the "Exclude" option. This is a counting analysis, demanding an input file with user provided information about each channel concerning background, efficiency, branching ratio, number of candidates observed etc (see Appendix B). These values are used by the program to generate the probability density functions (p.d.f.'s) of the signal+background and the background only hypotheses of Chapter 4. The p.d.f.'s are generated by applying a Monte Carlo (MC) algorithm.

Using the p.d.f.'s, the `alrmc++` program employs the CL_s method described in Chapter 4 to compute numerically $-2\ln Q$'s, the confidences of the background and background + signal hypothesis and several other values. To understand how the theory is adapted to a numerical approach in the program, it is necessary to take a detailed look at the different methods and objects used in the process.

6.2 How to find the confidences numerically

To find the confidences of the background and signal+background hypotheses, we see from Figure 5 of Chapter 4 that it is necessary to find the areas under the p.d.f.'s where $-2\ln Q$ is equal to or larger than $-2\ln Q_{obs}$. To find this area, we would like to use Equation (7), integrating the probability density functions numerically in the program. However, we remember from Chapter 4 that these probability density functions are in general not given analytically.

To solve this problem, we must remember that the p.d.f.'s are actually made up of the probability densities, or relative frequencies, of the values of $-2\ln Q$. To find the area under a p.d.f. for a specific interval on the axis of abscissa, we need to somehow find, numerically, the total relative frequency for all of the values in this interval. The relative frequency of the interval where $-2\ln Q \geq -2\ln Q_{obs}$ is the fraction of $-2\ln Q$'s that satisfies $-2\ln Q \geq -2\ln Q_{obs}$ compared to the total number of $-2\ln Q$'s.

The $-2\ln Q$ p.d.f.'s can be simulated by generating a large number of Monte Carlo experiments. Each of these experiments must contain one value of $-2\ln Q$. Together, the values of the $-2\ln Q$'s of all the experiments make up a distribution. However, an experiment does not just randomly choose a value of $-2\ln Q$. In stead, the variables that make up $-2\ln Q$ in each MC experiment are produced using a combination of the user input and random numbers. This way, each of the composite variables acquire a distribution around its input value; a distribution of the input values s_i and b_i of Equation (2) are created by inserting the observed values into a Poisson distribution, from which one random number is generated for each MC experiment.

When these partially random variables have been set in a MC experiment, a value of $-2\ln Q$ can be computed. This value is compared to the $-2\ln Q_{obs}$, the value of $-2\ln Q$ calculated using the user input. Every time a "random" $-2\ln Q$ is greater than or equal to $-2\ln Q_{obs}$, it is recorded. When a certain, user specified number of $-2\ln Q$'s have been produced and compared to $-2\ln Q_{obs}$, the number of $-2\ln Q \geq -2\ln Q_{obs}$ is divided by the total number of $-2\ln Q$'s calculated. This is the relative frequency of

the $-2\ln Q$'s that are greater than or equal to $-2\ln Q_{obs}$, which we remember is also equal to the area we wanted to find under the p.d.f., since the entire area of a p.d.f. is equal to one. Because of this property of a p.d.f., we have been able to simplify our two-dimensional problem of computing an area to a one-dimensional problem.

The background and signal+background distributions are, in theory, produced separately in the program. However, to minimise the number of calculations needed to produce these distributions, the theoretically computed $-2\ln Q$'s are used to generate both distribution functions. This is done by exploiting the fact that the tail of one distribution is more or less "hidden" under the other distribution (see Figure 4). To find points on the tails we produce a weight, making sure its value is less than one, and use each theoretically produced $-2\ln Q$ twice. First to find a point on one distribution function, and second to find a point on the tail of the other distribution, using the weight. The weights are necessary because the two distributions are not equal, and a $-2\ln Q$ produced for one distribution needs to be "scaled down" to fit the other distribution. The relation between the two distributions is $\mathcal{L}(b) = \frac{1}{Q}\mathcal{L}(s+b)$. We see that the weight is actually the inverse of the value of the likelihood ratio Q. This means that when we generate a value of $-2\ln Q$ that satisfies $-2\ln Q \geq -2\ln Q_{obs}$, this is stored as one "hit" by the signal+background hypothesis, and a scaled down "hit" by the background hypothesis. This principle is illustrated in Figure 10.

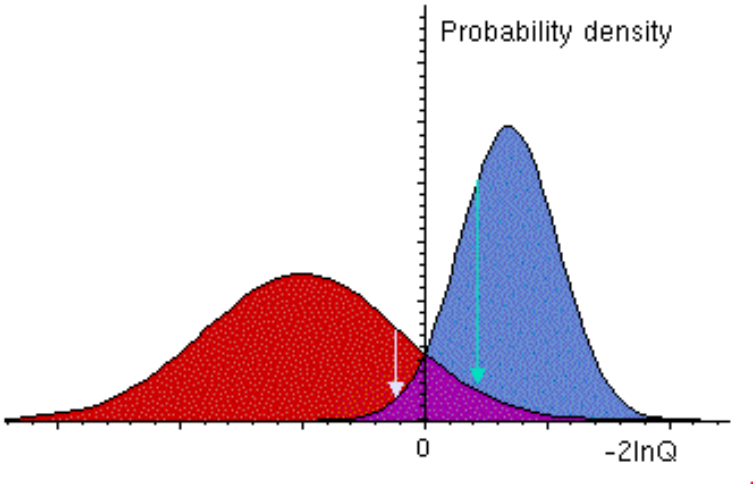

Figure 10: A point on the background hypothesis p.d.f. (blue), corresponding to a MC generated value of $-2\ln Q_{MC}$, is "scaled down" (green arrow) to find a point on the signal+background hypothesis p.d.f. (red), and a point on the signal+background hypothesis p.d.f. is "scaled down" (light purple arrow) to find a point on the background hypothesis p.d.f.

6.3 Step by step through the program

When the user has chosen the "Exclude" analysis, the program will open a user provided input file (see Appendix B) and read the channel information. When all the user

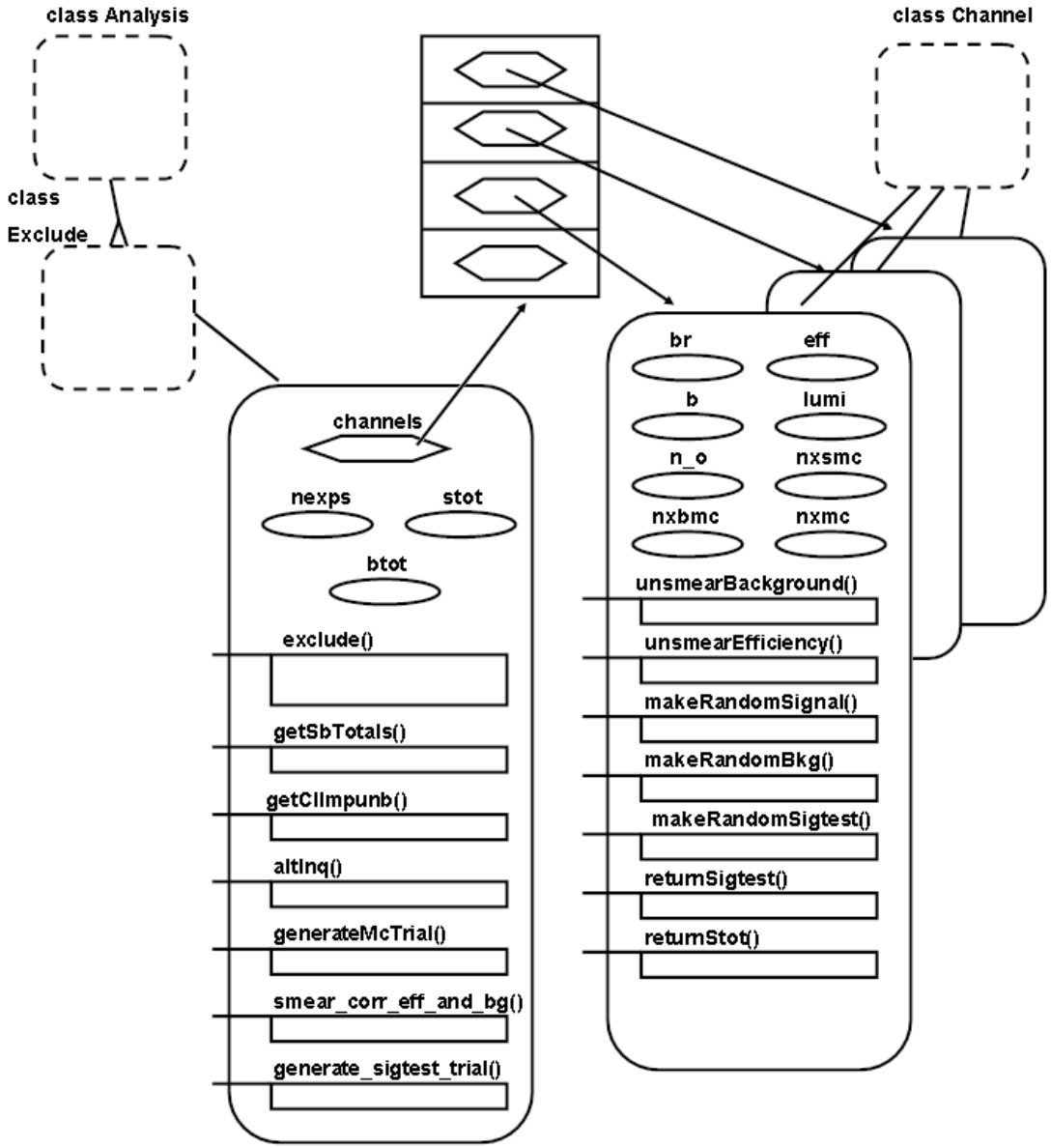

Figure 11: Close-up view of the most important variables and methods in the objects of the Exclude and Channel classes.

provided information has been stored in the channel objects, the method "exclude()" is called.

Method "exclude()"

Method "exclude()" starts with declaring an object of class TTree, found in the Root library [6]. Most of the communication between the `alrmc++` program and the Root classes is handled by the object of class Histogram. A TTree is a structure somewhat similar to an ntuple, but unlike an ntuple it can hold all kinds of data, like arrays or objects. The branches of the TTree object are filled with variables that will be computed during the analysis.

The first main method to be called by "exclude()" is "smear_corr_eff_and_bg(true)", which is called to initialise a list of the error sources of the experiment, if provided by the user. The next step is to find the sum of all signal and background candidates, s_{tot} and b_{tot}. The s_i of Equation (2) for each channel is given as

$$s_i = \mathcal{L}_i \cdot \sigma_i \cdot B_i \cdot \epsilon_i, \tag{12}$$

where $\mathcal{L}$ is the luminosity, σ is the cross section, B is the branching fraction and ϵ is the efficiency. "getSbTotals()" finds the s_{tot} sum by calling a method in each channel object that returns the value of the s_i.

Method "getClImpunb()"

The method doing most of the work in the analysis is "getClImpunb()". The name of this method is short for "get CL, improved, unbinned", a name that was constructed during the development of the original Fortran 77 subroutine. "getClImpunb()" is called from "exclude()" after "getSbTotals()", and this is where the $-2\ln Q$'s are computed and used.

The method first finds the values of s_{tot} and b_{tot} by calling "getSbTotals()", which adds together the nominal (user specified) s_i's and b_i's. The method "altlnq(type_data)" returns the value of the sum $\sum_{k=1}^{n} n_k w_k$ of Equation 4. The option *type_data* means that the observed candidates are used when the weights w_k are calculated. By combining these results, we find $-2\ln Q_{obs}$.

The next step is to make a loop which generates and processes a set of *nexps* MC experiments. Inside this loop, called the nexps loop, are many method calls and computations, the first one being a call to "generateMcTrial(conf_sb)". This is the first step in a process where unweighted signal+background experiments and weighted background experiments are generated, as described above. The *conf_sb* option requests that the method generates a random number of both signal and background candidates for each channel, using a Poisson distribution with the user provided values as the distribution parameter. The resulting number of candidates are stored in the variables *nxsmc* and *nxbmc*, their sum in *nxmc*, for each channel. "generateMcTrial()" calls method "smear_corr_eff_and_bg()" to randomise the channel values of the efficiency and background if there are any error sources defined by the user.

"getClImpunb()" calls the method "getSbTotals()" again, to find the new s_{tot} and b_{tot} after the possible changes in the background and signal values caused by

"smear_corr_eff_and_bg()". Two values of $-2\ln Q$ are computed at this stage, $-2\ln Q$ and $-2\ln Q_{nom}$. $-2\ln Q_{nom}$ is calculated using the s_{tot} of the user input values and the other by using the value of s_{tot} generated by the last call to "getSbTotals()". The argument of the "altlnq()" call, *type_mc*, ensures that the weights of Equation 4 are found using the new values of s_k and b_k and the variable *nxmc* (random number of candidates for the channel).

$-2\ln Q$ is used in the computation of a weight $wt = e^{-0.5\cdot 2\ln Q} = \frac{1}{Q}$. If the weight is less than or equal to one, $-2\ln Q_{nom}$ is compared to $-2\ln Q_{obs}$. If $-2\ln Q_{nom}$ is greater than or equal to $-2\ln Q_{obs}$, the variable *wt_sb_less* is increased by one and the weight is added to the variable *wt_b_less*. If, on the other hand, $-2\ln Q_{nom}$ is less than $-2\ln Q_{obs}$, *wt_b_greater* is increased by the weight. Either way, the $2\ln Q_{nom}$ [1] is stored in both the background and the signal+background experiment arrays (*q_b_expts* and *q_sb_expts*), the weight $\frac{1}{Q}$ is stored in the background weight array (*wt_b_expts*) and the weight 1 is stored in the signal+background weight array (*wt_sb_expts*).

The next part of the nexps loop is a generation of unweighted background experiments and weighted signal+background experiments. The first call is again for "generateMcTrial()", but with argument *conf_b* instead of *conf_sb*. The difference is that only the number of background candidates is generated as a random value of a Poisson distribution. After the new s_{tot} and b_{tot} have been calculated, the new values of the $-2\ln Q$'s are found using "altlnq(type_mc)". The weight is computed and compared to one, as before, and $-2\ln Q$ is compared to $-2\ln Q_{obs}$. This time, the variable *wt_sb_less* is increased by the weight and the variable *wt_b_less* by one if the generated experiment $-2\ln Q$ is greater than or equal to the observed value. The weight and $2\ln Q$ are stored in the various arrays described above. This completes the nexps loop.

Now we have enough information to compute CL_{sb}, CL_s and CL_b. From the definitions in Equations (6) and (8), we know that CL_{sb} and CL_b are defined as the probabilities of $-2\ln Q$ being greater than or equal to $-2\ln Q_{obs}$, given that the signal+background, or the background only hypothesis is true, respectively. The numerical way of finding these probabilities, as we remember from Chapter 6.2, is to divide the number of times (with weights) this condition was met during the nexps loop, with the total number of generated experiments. CL_s is given in Equation (9) as the ratio of the signal+background and the background confidence levels.

Having been filled with their appropriate values, the experiment arrays of $2\ln Q$ for both the background and the signal+background hypotheses are sorted in ascending order. Their weight arrays are also sorted, so that the weights follow the order of the experiment arrays.

The method goes on to generate a set of *nexps* unweighted test experiments. These are signal+background experiments that are generated to test what the confidences would be like if there was a signal at some unexpected location. The test experiment generation is done using the method "generate_sigtest_trial()", which produces a random number of candidates for both signal and background similar to the method "generateMcTrial()", and the method "altlnq()". The test experiment $2\ln Q$'s are stored in array *q_sigtest_expts*. Both the sum of all $-2\ln Q$'s and the sum of the $(-2\ln Q)^2$'s are computed. Using these two sums, we can find the mean and the variance of the

[1] due to historical reasons, the absolute value of $-2\ln Q$ is used

results.

The array *q_sigtest_expts* is sorted in ascending order, and we loop through the array, calculating the ratio of the iteration variable and the total number of simulated experiments (*nexps*). By comparing this ratio to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, we find the values of $2\ln Q$ at these points. These values are stored in the array *xi2_exp_sigtest*.

In order to find the values of discovery confidences and potentials, we want to integrate the background and the signal+background distributions from the top. When executing a loop starting at the top of the weight arrays, which were sorted earlier to follow the sorted values of the $2\ln Q$'s, we actually move from negative values of $-2\ln Q$ toward positive values. As we can see in Figure 5, this means that we compute the values of $1 - CL_b$ and $1 - CL_{s+b}$.

The actual loop is on the form of a while loop, starting at the number of MC experiments conducted and descending toward zero. At the top of the loop, the relative frequency, or probability density, of each $2\ln Q$ for both hypotheses are found by dividing the weight of the $2\ln Q$ by the sum of all the weights. The variables *wtbtot* and *wtsbtot* continuously hold the sum of all these relative frequencies, thus containing the updated values of $1 - CL_b$ and $1 - CL_{sb}$ of the background and signal+background hypothesis respectively. As *wtbtot*, which is identical with the significance $(1 - CL_b)$, reaches the standard normal distribution probabilities at -5, -4, -3, and -2 standard deviations, the corresponding values of *wtsb* $(1 - CL_{sb})$ are stored in the variables *p_disc_5s*, *p_disc_4s*, *p_disc_3s* and *p_disc_2s*. These variables represent the discovery potentials, the probabilities of making discoveries at various significance levels if the signal+background hypothesis is true. If the signal+background and the background distributions lie close together on the $-2\ln Q$ axis, the discovery potentials will have small values. If the distributions are only slightly overlapping, or not at all, the values will be close to one. These two situations are illustrated in Figure 12.

As *wtsbtot* $(1 - CL_{sb})$ reaches the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, the corresponding values of *wtbtot* $(1 - CL_b)$, $\frac{wtbtot}{wtsbtot}$ and $2\ln Q$ are stored in arrays *m_cl_b_exp_sb*, *m_cl_b_p_exp_sb* and *xi2_exp_sb* respectively. The value of CL_b at CL_s=0.05 is stored in the variable *cl_b_exp_sb*.

And finally, in the last part of the while loop, the confidences for the different frequency contours of the test experiments are computed by comparing all the $2\ln Q$'s of the MC experiments ($2\ln Q_{MC}$'s) with the *xi2_exp_sigtest* array found earlier. When the $-2\ln Q_{MC}$'s reach the point where they are equal to the various entries of the *xi2_exp_sigtest* array, $1-CL_b$, CL_{sb} and CL_s are stored in the arrays *m_cl_b_exp_sigtest*, *cl_sb_exp_sigtest* and *cl_s_exp_sigtest* respectively.

The while loop contained an integration of $1 - CL$'s. To find the CL's, we need to execute a for loop iterating from zero up to the number of Monte Carlo simulated background experiments. As in the while loop, this is a integration process where the relative frequency, or probability density, of each $2\ln Q$ for the background and the signal+background hypotheses are found by dividing the weight of the $2\ln Q$ by the sum of all the weights. The variables *cls* and *clsb* now hold the updated values of the CL_b and the CL_{sb} areas under the background and signal+background p.d.f.'s respectively. The variable *qb* holds the corresponding $2\ln Q$ value. The $2\ln Q$'s are found in the

q_b_expts array, which was sorted in ascending order earlier in the method.

By comparing each $2\ln Q$ with the previous, the MC experiments with equal values of $2\ln Q$ are accessed in a nested for loop as one "block" of experiments. This is necessary to ensure that the integration to find the confidences include all the experiments for each step on the $-2\ln Q$ axis, and to make sure that identical experiments get identical values of integrated confidences. (see Figure 13). For each of the members in the block of equal $2\ln Q$ values, the average value of CL_b for the signal+background hypothesis and its square is calculated and stored in variables *clbtot* and *clbsq*. To find the false exclusion rates, the probability of excluding the signal and the signal+background hypotheses when they should be accepted, we identify the two experiments where CL_s and then CL_{sb} are approximately equal to 5%, and look at the value of CL_{sb}. The false signal exclusion rate, *clsb* at $cls \approx 0.05$, is stored in *fe_rate*, and the false signal+background exclusion rate *clsb* when *clsb* reaches 5%, is stored in *fe_rate_sb*. If the smallest CL_s or CL_{sb} is greater than 5% the corresponding false exclusion rate is of course zero.

The method proceeds by analysing the statistics of the signal+background experiments. The variables *wexpt_signal* and *wexpt_signal_sq* accumulates values to be used later to find the average value of $-2\ln Q$ if the signal+background hypothesis is true, and its square. An alternate version of CL_s used by the ALEPH collaboration, *cls_aleph* [13], is also calculated.

The value of *cls* is continuously tested to find the 90%, 95% and 99% exclusion ($CL_s \leq 10\%, 5\%, 1\%$). For each of these tests, the weight of the background hypothesis distribution, *wtb*, is stored in the variables *wt99*, *wt95* and *wt90*.

Next, we look at the statistics of the background experiments. We find the average

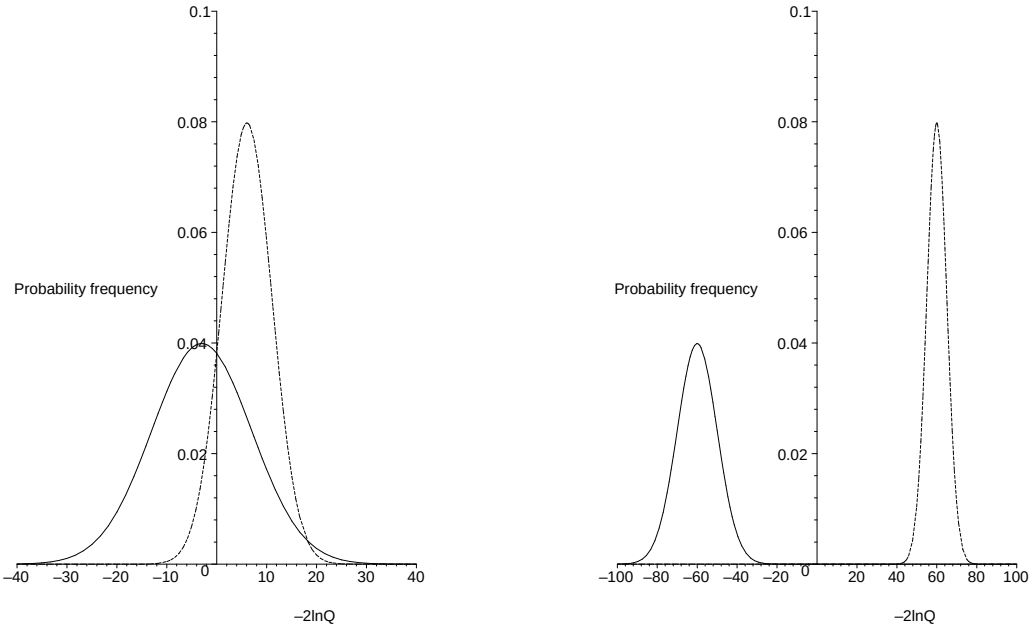

Figure 12: Left: The signal+background p.d.f. (solid line) and background p.d.f. (dashed line) overlap. The values of the discovery potentials are small. Right: The p.d.f.'s do not overlap, and the values of the discovery potentials are close to one. (The axes are arbitrary.)

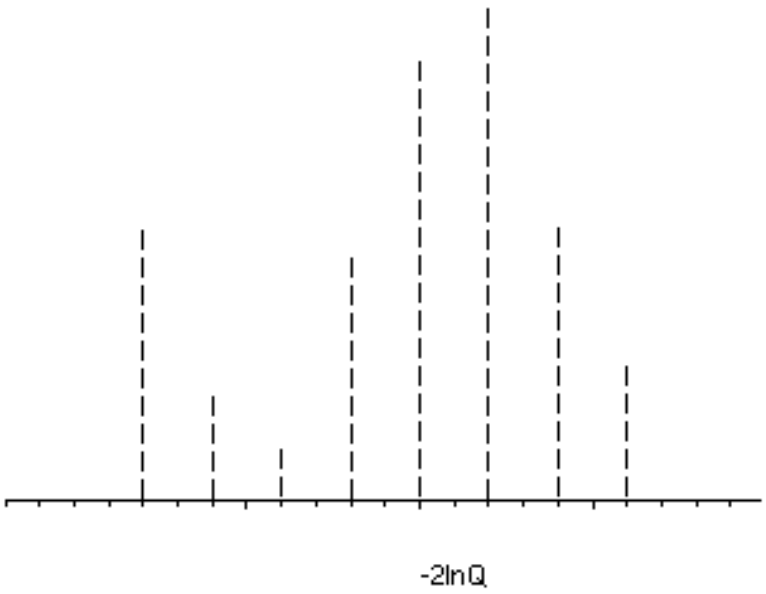

Figure 13: A histogram showing the frequencies with which the randomly generated $-2\ln Q$'s occur. A numerical integration consists of finding the sum of all the $-2\ln Q$'s frequencies. All experiments with equal values of $-2\ln Q$ must therefore be included in each integration step.

value of CL_{sb} for the background hypothesis by accumulating *clsb*wtb* in variable *clsbtot*. In variable *clbbtot*, the average value of CL_s for the background hypothesis is found by accumulating *cls*wtb*, and *clstot_aleph* holds the sum of all *cls_aleph*wtb*'s. We find the average value of CL_s given that the background hypothesis is true, in variable *wexpt_infty*, .

Continuing our analysis of the background experiments, we check the value of *clb* to find the five points where it reaches the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations. At each of these points, we store the values of CL_{sb}, CL_s and $2\ln Q$ in arrays *cl_sb_exp_b*, *cl_s_exp_b* and *xi2_exp_b* respectively.

In the last part of method "getClImpunb", we make the final calculations of expected values and uncertainties. As an example, to find the expected background confidence for signal+background experiments and its estimated uncertainty from the accumulated statistics, we divide *clbtot* with the sum of all the background experiment weights. The result is stored in variable *cl_b_infty*. The uncertainty, or the standard deviation, is found by employing the formula $\sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}}$, where n is the sum of all the background weights and x and $\bar{x}$ are, in this situation, *clb* and its average value. Most of the other variables found during the execution of method "getClImpunb()"are treated the same way; they are divided by the sum of the appropriate weights to find average values, and their uncertainties are calculated.

The output

When method “getClImpunb()” is finally finished and all TTree branch variables are set, the results need to be displayed in some way. The method “exclude()” calls two other methods to make outputs of the results. The first one is method “excludeFile()”, a method that prints all variables of interest to file “exclude.res”. The second method is called “finishHistos()”, and it fills the TTree and writes it to the Root file “likemc.root”. The results are also displayed on the computer screen, in the shell or in the form of a pop-up window, depending on which interface the user has chosen.

7 Test results

In this chapter I will present the results of two analyses performed with both the `alrmc` and the `alrmc++` programs, and one analysis performed with the `alrmc++` program only. The goal is both to show that `alrmc++` works, and that it reproduces the results of the original Fortran program.

7.1 Test analysis

While it was still under construction, I used a simple test analysis of six channels to test the `alrmc++` program. Each channel had values for the observed number of candidates, whether or not the channel should be used when running an analysis, whether the mass information should be used, the number of background candidates, the efficiency, the branching ratio and the luminosity. The input is shown below (the input variables and the format of the `alrmc++` input file is described in detail in Appendix B).

Observed	1	0	1	5	0	1
Use channel	1	1	1	1	1	1
Use mass	0	0	0	0	0	0
Background	0.675	0.440	0.583	5.340	0.410	0.730
Efficiency	0.0256	0.0425	0.1217	0.5223	0.0125	0.0226
Branching ratio	0.1	0.1	0.1	0.1	0.1	0.1
Luminosity	1.0	1.0	1.0	1.0	1.0	1.0

The analysis was performed with 10000 Monte Carlo experiments. The output of both programs are shown in the table below. The output variables are explained in Appendix C.

Variable name	`alrmc++`	`alrmc`
$\mathbf{s_{tot}}$	0.07472	0.0747
$\mathbf{b_{tot}}$	8.178	8.1780
$\mathbf{CL_{sb}}$	0.546093	0.551525
$\mathbf{CL_b}$	0.557638	0.562975
$\mathbf{CL_s}$	0.979296	0.979661
$\mathbf{-2 \ln Q}$**, from user input**	-0.00296659	-0.00296658
CL_s_infty	0.975514	0.975605
-2lnQ_b_infty	-0.000238418	-0.000896416
1-CLb	0.442362	0.437022001
Discovery potential, 3 sigma	0.00300535	0.0031
Discovery potential, 4 sigma	0.000200356	0.0002
Discovery potential, 5 sigma	0.000100178	0.0001

We see that the values of s_{tot} and b_{tot} for the two different programs were equal. The `alrmc++` values of the other variables came close to the `alrmc` values, but they were not exactly the same. This was expected for several reasons. The `alrmc` and the `alrmc++` programs use different random generators to produce random numbers, which of course results in different random numbers. Also, the variable types have been changed. This is something that comes about in the cause of translation; the REAL, REAL*4, INTEGER and so on of Fortran 77 have been translated mostly into the C++ variable types `double` and `int`. In addition, some of the values that are calculated during an analysis are very small, meaning that they are greatly influenced even by small fluctuations in the random numbers.

7.2 Simple analysis test

A simple check to see if the program is working, is to set the number of observed candidates to zero. From Chapter 5, we see that the $-2\ln Q$'s of the Monte Carlo generated experiments are calculated using random numbers produced by a Poisson distribution to find the number of candidates for the channels. We know from Equation (2) and (4) that when the number of candidates of the user input is equal to zero, only the $-2\ln Q$'s where the random number of candidates are also equal to zero will satisfy the condition $-2\ln Q \geq -2\ln Q_{obs}$.

The probability of obtaining a specific number of candidates , n_i, from the random number generation is given by the Poisson probability function

$$P(n_i) = \prod_{n_i} \frac{e^{-(s_i+b_i)}(s_i+b_i)^{n_i}}{n_i!}. \tag{13}$$

The symbols of this equation are explained in Chapter 4. We see that

$$P(n_i = 0) = \prod_{n_i} e^{-(s_i+b_i)}. \tag{14}$$

For the signal+background hypothesis,

$$P(n_i = 0) = \prod_{n_i} e^{-(s_i+b_i)} = e^{-(s_{tot}+b_{tot})}, \tag{15}$$

and for the background hypothesis,

$$P(n_i = 0) = \prod_{n_i} e^{-(b_i)} = e^{-b_{tot}}. \tag{16}$$

The ratio of these two probabilities, which corresponds by definition to the ratio of $\frac{CL_{s+b}}{CL_b}$, is $e^{-s_{tot}}$. From Equation 12 we see that when the luminosity, cross section, branching ratio and efficiency are all equal to one, as we assume is the case here, s_{tot} = number of channels, N_{chan}. As a result, CL_s will have the value $e^{-N_{chan}}$. An example of three channels would give $CL_s \approx 0.05$ (5%). The `alrmc++` input values of this example is shown below.

Observed	0	0	0
Use channel	1	1	1
Use mass	0	0	0
Background	1.0	1.0	1.0
Efficiency	1	1	1
Branching ratio	1.0	1.0	1.0
Luminosity	1.0	1.0	1.0

The analysis was performed with 100000 Monte Carlo experiments. The results of `alrmc` and `alrmc++` is displayed in the table below.

Variable name	`alrmc++`	`alrmc`
$\mathbf{s_{tot}}$	3	3
$\mathbf{b_{tot}}$	3	3
$\mathbf{CL_{sb}}$	0.00269843	0.00247875
$\mathbf{CL_b}$	0.054324	0.0497871
$\mathbf{CL_s}$	0.0497871	0.0497871
$\mathbf{-2\ln Q}$**, from user input**	-6	-6
CL_s_infty	0.263642	0.265931
-2lnQ_b_infty	-1.86152	1.83558
1-CLb	0.945676	0.950212955
Discovery potential, 3 sigma	0.128108	0.1226
Discovery potential, 4 sigma	0.0188541	0.0185
Discovery potential, 5 sigma	0.001267	0.0013

Both the `alrmc` and the `alrmc++` program finds approximately 5% confidence in the signal hypothesis. As in the previous example, there are some differences between the two programs in the values of the other variables.

7.3 Neutrino oscillations

The NOMAD neutrino oscillation $\nu_e \to \nu_\tau$ search [3] is a good example of an experiment where the `alrmc` program can be used in the final stage of the analysis. The data consists of thirteen different channels (using the `alrmc++` definition of Chapter 5). All the channels have very few or no observed candidates. The experimental results and the NOMAD method of calculating the signal confidence are described in [3] and [9]. To see what results the CL_s method and the `alrmc++` program give compared to [3], we need to insert the experiment data into the program.

In [3], the results are given in the form of an upper limit on the probability of a ν_e oscillating to a ν_τ at 90% confidence: $P_{osc}(\nu_e \to \nu_\tau) < 2.6 \times 10^{-2}$ at 90% CL. The CL

value is the average value of the confidence in the signal hypothesis. The sensitivity is given as $P_{osc} = 4.3 \times 10^{-2}$.

Because the `alrmc` program was originally designed to make use of one analysis method in particular, its input variable names do not match the ones of the analysis method of [3]. For example, the variable P_{osc} had to be inserted in the cross section variable of the input file to "scale" the input control variables to match the NOMAD data. This is a weakness of the original program that has been, unfortunately, passed on to the new `alrmc++` program. The experimental results of [3] had to be interpreted so that the their values could be introduced to the program through the correct control variables.

In the `alrmc++` program, the median of the confidence in the sensitivity [9] (when there is no signal, only background) is represented by the third element of array *cl_s_exp_b*, and the average value is represented by the variable *CL_s_infty*. The confidence in the signal hypothesis is, as usual, CL_s.

To find the median value of the sensitivity at 90% CL using the `alrmc++` program, the value of the cross section control variable was first adjusted until the value of the output variable *cl_s_exp_b[2]* was approximately 0.10 (10%). Later the same adjustments were made to find the cross section values where *CL_s_infty* and $CL_s \approx 0.10$.

The input of the `alrmc++` program which gives *cl_s_exp_b[2]* ≈ 0.10 is shown below.

Observed	0	1	4	0	0	3
2	0	5	5	0	1	0
Use channel	1	1	1	1	1	1
1	1	1	1	1	1	1
Use mass	0	0	0	0	0	0
0	0	0	0	0	0	0
Background	1.19	0.42	3.01	1.45	0.28	2.70
0.50	1.80	5.0	6.5	0.5	0.1	0.4
Efficiency	3.9	4.5	12.1	10.9	23.3	12.6
4.5	20.1	45.7	25.9	1.8	2.1	1.8
Branching ratio	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0
Luminosity	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0
Cross section	0.04267	0.04267	0.04267	0.04267	0.04267	0.04267
0.04267	0.04267	0.04267	0.04267	0.04267	0.04267	0.04267

The output, where *cl_s_exp_b[2]* is very close to 0.10, is shown below. The program was performed with 100000 Monte Carlo experiments.

Variable name	alrmc++
$\mathbf{s_{tot}}$	7.21976
$\mathbf{b_{tot}}$	23.85
$\mathbf{CL_{sb}}$	0.0267006
$\mathbf{CL_b}$	0.371901
$\mathbf{CL_s}$	0.0717951
$\mathbf{-2\ln Q}$**, from user input**	-3.88761
CL_s_infty	0.145795
-2lnQ_b_infty	-2.75927
1-CLb	0.628099
Discovery potential, 3 sigma	0.224919
Discovery potential, 4 sigma	0.0483331
Discovery potential, 5 sigma	0.0052645
cl_s_exp_b[2]	0.100076

The value of the median of the sensitivity at 90% confidence according to the `alrmc++` program analysis, is about 0.0427. This is approximately the same value as the NOMAD value of 0.043. When the value of the average CL (CL_s_infty)reached 10%, however, the sensitivity value was 0.05035. This value is greater than the NOMAD value.

`alrmc++` gave an upper limit on the probability $P_{osc}(\nu_e \to \nu_\tau)$ at 90% CL of 0.038, which is a greater value than the NOMAD value of 0.026.

We see that the upper limit on the probability of the neutrino oscillation $\nu_e \to \nu_\tau$ becomes greater when we use the `alrmc++` program to analyse the NOMAD data. Still, the point of this example was not to re-analyse the NOMAD data, but to show that it is actually possible to use the `alrmc++` program to analyse the data from such experiments. I will therefore not try to explore this topic further, or draw any conclusions as to which statistical analysis method is the better to use in this case.

8 Conclusions and outlook

From the test results of the previous chapter, and from other tests conducted in the process of translation from Fortran 77 to C++, it seems clear that the `alrmc++` works, and is able to reproduce the results of the old Fortran version. However, there are certain aspects of the translated version that need to be mentioned.

8.1 Problems and results

The `alrmc++` program is mainly object oriented in its structure, although some parts of the code are not quite optimal in this respect. For example, the calculations of s_k and b_k of Formula (5) in method "altlnq()" could be delegated to the channel objects. The reason that this has not been done, is mainly that changing these parts of the code would make such a large impact on the program structure that it would be too time consuming for a cand. scient. assignment. Being at least mostly object oriented, however, the program is modular and can easily be changed or expanded.

The user interface, written in Java, is optional and is expected to make it easier for new users to understand and use the program. It has a help section providing information about the input file and the analysis. The Java interface window is shown in Figure 7.

There have been problems for new users related to the connection between the C++ program and the Java interface. The most common problem seems to be that the Java program is unable to find the C++ shared library (see Appendix A). The compilation of the interface and `alrmc++` combination can also be difficult, and is likely to be dependent on operating system and compiler versions.

Not all features of the C++ program version have been tested yet. There are possibilities in the program for letting the user provide error values and error sources, but this has not been fully implemented. Another feature not yet tested is the possibility of letting the user choose to include information about, for example, particle masses. This would require that signal and background probability distributions be specified in the methods "sigprob()" and "backprob()". These distributions will, hopefully, be included in future expanded versions of the program.

One of the disadvantages of object oriented programming, mentioned in Chapter 3, is that the computational speed may be less than for non-OO structures. This is true for the `alrmc++` program, which was about twenty times slower than the `alrmc` program in April 2002. Because of the restrictions in the duration of a cand. scient. assignment, I have not had time to optimise and improve the C++ program, so that when the translation was finished, there was a lot that could be done to make the program work faster [2].

[2] My supervisor, Alex Read, has done some optimising and has managed to get the difference in computational speed down from 20 times slower than the Fortran program to about 1.2 times slower. Many of the main bottlenecks were located at points where arrays were declared and deleted, and moving these operations proved to be quite effective.

8.2 The future

So far only one analysis type (MC computations of confidences for counting experiments at a fixed point of the model parameter(s)) has been implemented, but the program is designed for more. The idea is that the super class Analysis (see Chapter 3) should contain all the common methods that all analysis types use, while its sub classes should contain the specific methods of the various analyses. Figure 14 shows super class Analysis with three sub classes, each of them associated with an object containing the analysis type methods.

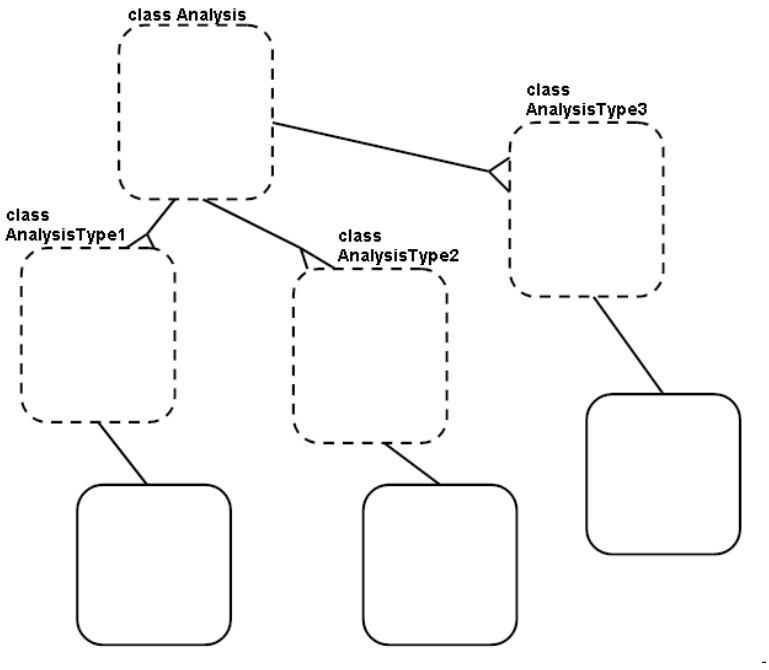

Figure 14: The super class Analysis with three sub classes, their objects containing analysis type methods.

When using the command line version of the C++ program, it is the "main()" method that initialises the various objects and calls the major methods. To adapt the program to work with a different form of input or slightly different procedures, the "main()" method could be replaced by a customised version.

A test was made to see if the results of a simulation of the Higgs search process $H \to \gamma\gamma$ [11] with approximately 80 channels (with several thousand events per channel) could be analysed by the `alrmc++` program. The test showed that this could not be done by the current program in a finite amount of time, and so adapting the program to read such input and analyse it is a challenge for the future.

Also for the future is the an addition to the program of a feature which allows the user to feed information about the parameters of the test signal MC experiments. At present date, the user can only provide the observed values of the various parameters.

The compilation and run-time problems of the Java user interface might be solved in the future by making a completely new interface, or providing a more user friendly

compilation "recipe". A suggestion of using scripting [14] to solve the problems seems promising.

The `alrmc++` program is a work in progress. It has been re-organised and object oriented, while still reproducing the results of the "exclude_signal" analysis of the Fortran version, but there is a lot more to be done. There is still a need for further optimisation and object orienting, and for expansion and adaption. Hopefully, this work will be continued in the future.

A How to use the alrmc++ program

A.1 Preparations

The program uses the external libraries of Root [6] and CLHEP [7], both available at the CERN home pages. The paths to these libraries must be set in the environment variable LD_LIBRARY_PATH and in the compilation command or Makefile.

When using the Java interface version, the C++ files must be compiled as a shared library [2]. The path of this shared library must be set in the Java file AnalysisJava.java.

A.2 Compilation and the Makefile

When all the necessary files have been copied and the library paths have been found and set, it is time for the compilation. The easiest way would be to compile only the C++ files first to get a working sample of the command line version. This is a normal C++ compilation, using a C++ compiler with the external libraries of Root and CLHEP included. The program has so far been compiled with the g++ compiler, and all standard C++ libraries have been included. The procedure is described in most C++ tutorials.

To use the Java interface, some changes must be made from the command line version by the user. The compilation will be different, as the Java files must be compiled with a java compiler, the C++ files with a C++ compiler (with the option -shared) and the "in between" class AnalysisJava.class (produced by the java compiler) with the javah command with the option -jni ("javah -jni AnalysisJava"). The "shared" option makes a shared library of all the C++ files. The shared library is loaded at the start of each session in the object of class AnalysisJava. As mentioned in the Preparations, *this means that the name and path of the shared library must be correct in this file.*

An example of a Makefile for compilation of both the command line and the Java interfaces is included below. The shared library is called nativeC.so, and the executable command line file is called alrmc++:

```
# The Macros
#MAKEFILE = makefile
    FILE = alrmc++
  OFILES = nativeC.so
      CC = g++

      LN = -lm  -lCLHEP  -lCint -lCore -lTree -ldl -lg2c -lstdc++
    LIBS = -L./ -L/usr/lib -L/mn/susy/particle/clhep-1.7.0.0/i386_redhat71/lib
   -L/mn/susy/particle/root-v3.01.06/i386_redhat71/lib/
 INCLUDE = -I./ -I/mn/susy/particle/clhep-1.7.0.0/i386_redhat71/include
   -I/mn/susy/particle/root-v3.01.06/i386_redhat71/include/
  IDIRS  = -I/mn/helicity/local/fys/epf/jdk1.3.1_02/jre/lib/i386/
   DEBUG = -g -O0

# Object files, it pays of in structure to truncate with backslash (\)
```

```
OBJC = main.o channel.o analysis.o exclude.o histogram.o
OBJJ = channel.o analysis.o exclude.o histogram.o cppJava.o AnalysisJava.o

INCC = analysis.h exclude.h histogram.h channel.h arrayComp.h
INCJ = cppJava.h jni.h AnalysisJava.h

.SUFFIXES: .cc .o

# Rule for each subroutine
.cc.o:
$(CC) $(LIBS) $(INCLUDE) $(DEBUG)  -c $<

MAIN = $(FILE)

# target
all: $(MAIN) java lib

# MAINs dependencies, compile each subroutine according to Rule
$(MAIN): $(OBJC) $(MAKEFILE)

$(OBJC): $(INCC)

$(OBJJ): $(INCC) $(INCJ)

# Compile Main and link
$(MAIN): $(OBJC)
$(CC) $(STDOPT) $(DEBUG) $(INCLUDE) $(LIBS) -o $@ $(OBJC) $(LN)

#Make libraries
$(OFILES): $(OBJJ)
 $(CC) $(CCFLAGS) $(INCLUDE) $(LIBS) -shared -o nativeC.so
 $(OBJJ) $(LN)

lib: libAnalysisJava.so

libAnalysisJava.so: AnalysisJava.h $(OFILES) AnalysisJava.o
 $(CC) $(CCFLAGS) $(INCLUDE) $(LIBS) -shared -o libAnalysisJava.so
 $(OFILES) $(CPPDEFINES) $(CCLIBS) $(IDIRS) $(LN) AnalysisJava.o

java: JavaCpp.class AnalysisJava.class AnalysisJava.h

JavaCpp.class: JavaCpp.java
javac JavaCpp.java

AnalysisJava.h: AnalysisJava.class
```

```
javah -jni AnalysisJava
touch AnalysisJava.h

AnalysisJava.class: AnalysisJava.java
javac AnalysisJava.java
```

A.3 Special problems

There are some factors that the user should be aware of when using the `alrmc++` program. These are listed below.

- The compilation of shared libraries and of javah -jni is likely to be dependent on the operating system. The program has so far been tested only within a Linux Red Hat environment.
- If the program can't find the shared library, it might be necessary to add the path of this library to the LD_LIBRARY_PATH or similar.
- There is a strange difference between the command line and the Java interface version. The format of the input file is the same in both cases, but the commas of the "double" variables must be written as "." for the command line and "," for the Java version. This is a strange difference that originally did not seem to have any logical explanation. Just lately, it has been suggested that this is due to the locale feature of the Java language, which adjusts the comma standard according to the country of residence.
- There is a problem if a variable that is defined as an integer in the program is given as a double in the input. If this happens, the program will read the number as zero, causing problems in the computation.

A.4 Output

When the program has finished, there will be two output files; one text file containing the variables computed by the program, and a .root file containing a Root TTree filled with result data. The latter can be opened and viewed in Root. An example of a Root script that opens the file and displays the variable "cl_sb" as a histogram is shown below. The script is contained in a file called "display.c". The file is executed in the Root environment by typing ".x display.c".

```
int display()
{
  //Reset all variables in session:
    gROOT->Reset();
  //Open file:
    TFile *p = new TFile("likemc.root","READ");
  //View file contents
     p->ls();
  //Read file Tree to new session Tree:
    TTree *tree = T;
  //Make a canvas to display histograms:
    TCanvas *can = new TCanvas("c","C",0,0,600,400);
  //Draw variable cl_sb as a histogram:
    tree->Draw("cl_sb");

  return 0;
}
```

B Input file format

The Java interface should be used for global information input. This could be the number of Monte Carlo experiments you want the program to simulate, the scan range (not yet implemented), etc of the analysis type you want to run. The input file should contain information about the individual channel(s) you want to analyse.

The first line of the input file should contain the number of channels you want to analyse. The rest of the lines should all start with a control variable (see list of control variables below), and then the values of that variable for each channel, with a space or tab in between. The variables are not case sensitive.

B.1 The control variables:

Efficiency: The nominal signal detection efficiency per search channel. Default = 1.0.

Background: The nominal integrated background rate per search channel.
Default = 0.0.

Observed: The number of candidates observed per channel. Default = 0.

Luminosity: The integrated luminosity per search channel. Default = 1.0.

Branching: The branching ratio per search channel. Default = 1.0.

Cross section: Cross section per search channel. Default = 1.0.

Usechan: True/False per channel to enable/disable it in the analysis. This way, it is possible to study individual channels or subsets of channels once a multichannel search has been configured. True = 1, false = 0. Default = 0.

Usemass: True/False per channel to enable/disable the use of the p.d.f. of the discriminant for enabled channels. Usemass is a synonym for usepdf (reconstructed mass is a typical discriminant) in the Fortran version. True = 1, false = 0. Default = 0 .
The "true" option is *not yet implemented.*

Betype: The background error type per channel 0=No errors, 1=Normal distribution, 2=Poisson distribution, 3=Binomial distribution. Default = 0.
Not yet implemented.

Bep1: First parameter for generating background errors per channel (the interpretation depends on the error type, see betype): Normal distribution (1) = Standard deviation. Binomial distribution (2) = Number of events in parent sample. Poisson distribution (3) = Number of events selected from parent sample. Default = -1.
Not yet implemented.

Eetype: The efficiency error type per channel: 0 = No errors, 1 = Normal distribution, 2 = Binomial distribution, 3 = Poisson distribution. Default = 0.
Not yet implemented.

Eep1: First parameter for generating efficiency errors per channel. The interpretation depends on the error type (see eetype); Normal distribution (1) = Standard deviation, Binomial distribution (2) = Number of events in parent sample, Poisson distribution (3) = Number of events selected from parent sample. Default = -1.
Not yet implemented.

Usesmear: True/False per channel to enable/disable the systematic uncertainties for enabled channels. True = 1, false = 0. Default = 0.
The "true" option is *not yet implemented.*

Example file

Below is an example file of six channels, using some of the control variables.

```
6
OBSERVED 1 0 1 5 0 1
USECHAN 1 1 1 1 1 1
USEMASS 0 0 0 0 0 0
BACKGROUND 0.675 0.440 0.583 5.340 0.410 0.730
EFFICIENCY 0.0256 0.0425 0.1217 0.5223 0.0125 0.0226
BRANCH 0.1 0.1 0.1 0.1 0.1 0.1
BETY 1 1 1 1 1 1
BEP1 0.26 0.21 0.24 0.73 0.20 0.27
EETY 1 1 1 1 1 1
EEP1 0.015 0.015 0.015 0.015 0.015 0.015
LUMI 1.0 1.0 1.0 1.0 1.0 1.0
```

C Lists of methods and variables

Following is a reference list of the most important methods of class Analysis, and their purpose:

readFile: Reads the input file (see Appendix B), makes the channel objects of class Channel and stores the user input in the variables of the channel objects.

unsmearBackground: Sets the background to the nominal (user input) background.

getSbTotals: Finds s_{tot} and b_{tot}, the expected number of all signal and background candidates of all channels.

getClImpunb: Finds all confidences and their uncertainties.

unsmearEfficiency: Sets the efficiency to the nominal (user input) efficiency.

altlnq: Finds the sum of Equation (4).

generateMcTrial: Generates random values for the number of signal and/or background candidates per channel.

smear_corr_eff_and_bg: Generates efficiencies and number of background candidates with a set of random fluctuations if there are any error sources defined.
First call: Makes a list of all user input error sources.

generate_sigtest_trial: As generateMcTrial, for unweighted signal experiments.

The output file "exclude.res" contains a lot of variables and their values. This is a reference list of these output variables:

s_exp: The number of expected signal candidates for all channels. The variable s_{tot} of Equation (2).

b_exp: The number of expected background candidates for all channels.

CL_sb: The probability that the results are less signal-like than the observed values, given that the signal+background hypothesis is true. The definition of this variable is given in Equation (6).

CL_b : The probability that the results are less signal-like than the observed values, given that the background-only hypothesis is true. The definition of this variable is given in Equation (8).

CL_s: The probability that the results are less signal-like than the observed values, given that the signal hypothesis is true. This is not a true confidence level, but a ratio

of confidences that provides a good approximation. The definition of this variable is given in Equation (9).

-2lnQ observed: The value of $-2\ln Q$ calculated using the user input (observed) values.

CL_s_infty: The expected average value of CL_s for the background only hypothesis.

-2lnQ_b_infty: The corresponding value of $-2\ln Q$.

Discovery potentials; 3,4,5 sigma (p_disc_3s, p_disc_4s, p_disc_5s): The probabilities of making discoveries $(1 - CL_{sb})$ when $1 - CL_b$ (the significance) is equal to the standard normal distribution probabilities at 3, 4 and 5 standard deviations, if the signal+background hypothesis is true.

1-CLb (cl_b_comp): See CL_b. The definition of variable CL_b is given in Equation (8).

cl_b_exp_sb: The expected average value of CL_b given that the signal+background hypothesis is true.

m_cl_b_exp_sb[]: The expected value of $1 - CL_b$ when $1 - CL_{sb}$ is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the signal+background hypothesis is true.

m_cl_b_p_exp_sb[]: The expected values of $\frac{1-CL_b}{1-CL_{sb}}$ when $1 - CL_{sb}$ is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the signal+background hypothesis is true.

m_cl_b_p_exp_b[]: The expected values of $\frac{1-CL_b}{1-CL_{sb}}$ when $1 - CL_{sb}$ is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the background hypothesis is true.

cl_sb_exp_b[]: The values of CL_{sb} when CL_b is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the background hypothesis is true.

cl_s_exp_b[]: The values of CL_s when CL_b is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the background hypothesis is true.

xi2_exp_b[]: The expected values of $-2\ln Q$ when CL_b is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the background hypothesis is true.

xi2_exp_sb[]: The expected values of $-2\ln Q$ when CL_{sb} is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations, if the signal+background hypothesis is true.

cl_s_exp_sb: The expected value of CL_s for the median value of $1 - CL_{sb}$ if the signal+background hypothesis is true.

xi2_exp_sigtest[]: The values of $-2\ln Q$ when the test signal distribution is equal to the standard normal distribution probabilities at -2, -1, 0, 1 and 2 standard deviations.

m_cl_b_exp_sigtest[]: The values of $1 - CL_b$ when the values of the simulated background $-2\ln Q$'s are equal to the xi2_exp_sigtest[]'s.

cl_sb_exp_sigtest[]: The values of CL_{sb} where the values of the simulated background $-2\ln Q$'s are equal to the xi2_exp_sigtest[]'s.

cl_s_exp_sigtest[]: The values of CL_s where the values of the simulated background $-2\ln Q$'s are equal to the xi2_exp_sigtest[]'s.

p_disc_5s_p, p_disc_4s_p, p_disc_3s_p and p_disc_2s_p: Discovery potentials. The values of $1-CL_{sb}$, when $\frac{1-CL_b}{1-CL_{sb}}$ is equal to the standard normal distribution probabilities at 2, 3, 4 and 5 standard deviations, given that the signal+background hypothesis is true.

cl_b_infty: The average value of CL_b for the background only hypothesis.

d_cl_b_infty: The error (standard deviation) of cl_b_infty

cl_bb_infty: The average value of CL_b if the background only hypothesis is true.

d_cl_bb_infty: The error (standard deviation) of cl_bb_infty

wexpt_sigtest: The average value of $-2\ln Q$ for the test signal experiments.

wexpt_sigtest_rms: The error (standard deviation) of wexpt_sigtest.

wexpt_signal: The average value of $-2\ln Q$ if the signal+background hypothesis is true.

wexpt_signal_rms: The error (standard deviation) of wexpt_signal.

p_excl_90: The probability of excluding the null hypothesis for $CL_s = 10\%$ given that the signal hypothesis is false.

p_excl_95: The probability of excluding the null hypothesis for $CL_s = 5\%$ given that

the signal hypothesis is false.

p_excl_99: The probability of excluding the null hypothesis for $CL_s = 1\%$ given that the signal hypothesis is false.

p_excl_95_aleph: The sum of the background weights of the Monte Carlo generated experiments for $CL_s\_aleph = 5\%$

cl_sb_infty: The average value of CL_{sb} for the background only hypothesis.

d_cl_sb_infty: The error (standard deviation) of cl_sb_infty.

cl_s_infty: The average value of CL_s for the background only hypothesis.

d_cl_s_infty: The error (standard deviation) of cl_s_infty.

wexpt_infty: The value of CL_s when the background hypothesis is true.

wexpt_infty_rms: The error (standard deviation) of wexpt_infty.

fe_rate: False exclusion rate; the value of CL_{sb} when CL_s is 5%.

fe_rate_sb: False exclusion rate; the value of CL_{sb} when CL_{sb} is 5%.

d_cl_sb: Not yet implemented.

d_cl_b: Not yet implemented.

cl_s_aleph: A specialised value of CL_s for the ALEPH experiment.

cl_s_aleph_infty: The expected value of CL_s_aleph when the background hypothesis is true.

References

[1] P. Abreu et al. (2001): *Search for the Standard Model Higgs boson at LEP in the year 2000*, EP 2001-004.

[2] C. Anderson (1997): *Putting a Java interface on your C, C++ or Fortran code* [on line]. Available at: http://www.math.ucla.edu/ anderson/JAVAclass/JavaInterface/JavaInterface.html [Viewed 12. September 2001].

[3] P. Astier et al. (1999): *Limit on $\nu_e \to \nu_\tau$ Oscillations from the NOMAD experiment*, CERN-EP-99-151.

[4] ATLAS Collaboration (1999): *Detector and Physics Performance Technical Design Report, Vol 2, CERN/LHCC/99-15* .

[5] G.K. Bhattacharyya and R.A. Johnson (1977): *Statistical concepts and methods* John Wiley & Sons, New York.

[6] R. Brun, F. Rademakers, S. Panacek, D. Buskulic, J. Adamczewski, M. Hemberger, N. West (2002): *ROOT User's Guide 3.02b* [on line]. Available at: http://root.cern.ch/root/RootDoc.html [Viewed 20. April 2002].

[7] CERN - Information Technology Division, Application Software & Databases, group IT/ASD: *The Anaphe home page* [on line]. Available at: http://anaphe.web.cern.ch/anaphe/ [Viewed 5. June 2002].

[8] O. Couet (1999): *PAW tutorial* [on line]. Available at: http://wwwinfo.cern.ch/asd/paw/tutorial/tut_index.html [Viewed 19. June 2002].

[9] G.J. Feldman, RD Cousins: Phys. Rev. D 57 (1998) 3873.

[10] P. Field (1996): *An introduction to object-oriented design* [on line]. Available at: http://www.accu.org/acornsig/public/articles/ood_intro.html [Viewed 27. April 2001].

[11] U. Fuskeland (2002): *Simulation of a search for the Standard Model Higgs boson in the $H \to \gamma\gamma$ channel at LHC/ATLAS*, Master thesis in experimental particle physics, University of Oslo.

[12] F. Gianotti (1999): *Collider Physics: LHC0*, ATLAS Conference Lecture ATL-CONF-2000-001.

[13] S. Jin, P. McNamara (2000): *The Signal Estimator Limit Setting Method*, in F. James, L. Lyons and Y. Perrin (eds.), *Workshop on Confidence Limits*, CERN Yellow Report 2000-005, p103. Available through weblib.cern.ch.

[14] H.P. Langtangen (2002): *Scripting Tools for Scientific Computations* [on line]. Available at: http://www.ifi.uio.no/ hpl/VitSimScripting/ [Viewed 12. June 2002].

[15] D. Nourie (2001): *Building an application* [on line]. Available at: http://developer.java.sun.com/developer/onlineTraining/new2java/divelog/ [Viewed 12. February 2002].

[16] A.L. Read (2000): *Modified frequentist analysis of search results (the CL_s method)*, in F. James, L. Lyons and Y. Perrin (eds.), *Workshop on Confidence Limits*, CERN Yellow Report 2000-005, p 81. Available through weblib.cern.ch.

[17] A.L. Read (1997): DELPHI 97-158 PHYS 737. Available at: http://www-h1.desy.de/h1/www/h1work/bsm/Stat.html.

[18] N. Smirnov, F. Carena, Tz. Spassoff (1998): *Object model of DELPHI Data*, DELPHI Internal Note 98-157 PROG 235.

[19] B. Stearns: *Java native interface* [on line]. Available at: http://java.sun.com/docs/books/tutorial/native1.1/index.html [Viewed 10. Mar ch 2002].