Measurement of the Reaction pp$\rightarrow$p$K^+\Lambda$ and its Analysis with a new Analysis Program: typeCase

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Katharina Ehrhardt
aus München

Tübingen

2011

Tag der mündlichen Prüfung:	31. Oktober 2011
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Heinz Clement
2. Berichterstatter:	PD Dr. Roland Speith

Abstract

Im Herbst 2004 wurde am COSY-TOF-Detektor ein dediziertes hoch-Statistik Experiment zur Suche nach dem Pentaquark-Teilchen Θ^+ unternommen. Als Strahlimpuls wurden $3.081 GeV/c$, entsprechend $T_p = 2.282 GeV$ Strahlenergie, in Proton-Proton-Stössen verwendet. Auch wenn für den Wirkungsquerschnitt für die Produktion des Θ^+ nur eine obere Schranke angegeben werden konnte [45], lassen sich die, während dieses Experiments gewonnenen Daten im Hinblick auf die Reaktion $pp \to pK^+\Lambda$ analysieren, nachdem diese Reaktion genauso wie die Reaktion $pp \to pK_s\Sigma^+$ in den Daten angereichert wurde. Mit der im Rahmen dieser Arbeit geschaffenen Analysesoftware **typeCase** wurden diese Daten analysiert.
Durch erschöpfende Kalibration konnte eine Auflösung von $\approx 6 MeV(FWHM)$ in den invarianten Massen erreicht werden. Der totale Wirkungsquerschnitt der Reaktion $pp \to pK^+\Lambda$ wurde zu $21.1 \pm 0.1_{stat} \pm 2.0_{sys} \mu b$ bestimmt und stimmt innerhalb der Fehler mit anderen Veröffentlichungen überein [41]. Durch die exklusiven Messungen konnten differentielle Wirkungsquerschnitte erstellt werden. Auch sie stimmen im Wesentlichen mit früheren Veröffentlichungen überein.
Durch die exzellente Auflösung wurde im invarianten Massen-Spektrum $M_{p\Lambda}$ die Struktur des Σ-Cusp sichtbar. Auf Grund der hohen Statistik konnten nun zum ersten Mal differentielle Wirkungsquerschnitte für den Σ-Cusp erstellt werden. Sein totaler Wirkungsquerschnitt konnte zu $\sigma_{cusp} = 1.2 \pm 0.1_{stat} \mu b$ bestimmt werden.
Die Spektren können durch Modelrechnungen der N^*-Resonanzen $S_{11}(1650)$,-$P_{11}(1710)$ und $P_{13}(1720)$ zuzüglich einer einfachen Rechnung für den Σ-Cusp gut beschrieben werden.

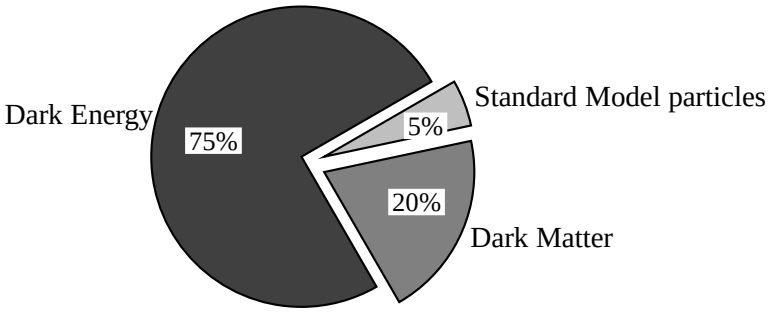
Contents

Chapter 1

Introduction

1.1 Structure of matter

As we know today, the universe is made up of dark energy, dark matter and what we call ordinary matter: Most of it (75%), the dark energy, is something we don't



have any idea what it is made of yet, though we know what it does: In the current phase of our universe it accelerates its expansion.
The next big part is the dark matter (20%) and at least here we can pin down some of the properties. It consist of massive particles interacting only weakly with "normal" matter. Both dark energy and dark matter are under close observation in science today, but not issue of this thesis, though mentioned for completeness.
The ordinary matter, the remaining five percent of the energy content of the universe, is what we know most of:
There are four fundamental interactions, two sets of particles each of three families made up of doublets. The interactions are:

electro-magnetic force coupling to the (single) electromagnetic charge. Its mediating boson is the massless photon γ.

strong force couples to the color-charge (threefold: red, green, blue). Its corre-

sponding bosons are the 8 (also color-charged) massless gluons.

weak force has three massive vector bosons: charged (W^+, W^-) and neutral (Z^0) current. This gives rise to the very short range of the interaction. It couples to the weak charge.

gravitation couples to the mass and was assigned a hypothetical particle, the (massless) graviton.

Theoretical descriptions for the electro-magnetic interaction exist ($U(1)$) and perturbatory calculations give good results. This is also true for the weak-force, that has been included into the theory now naming electro-weak interaction ($SU(2) \times U(1)$).

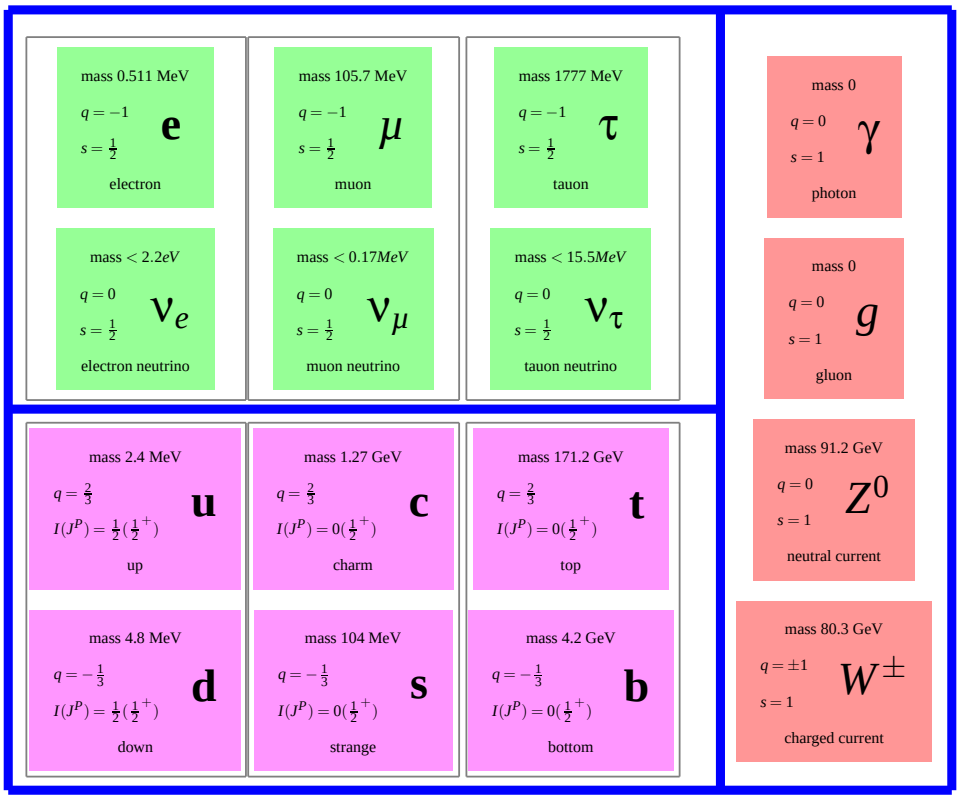
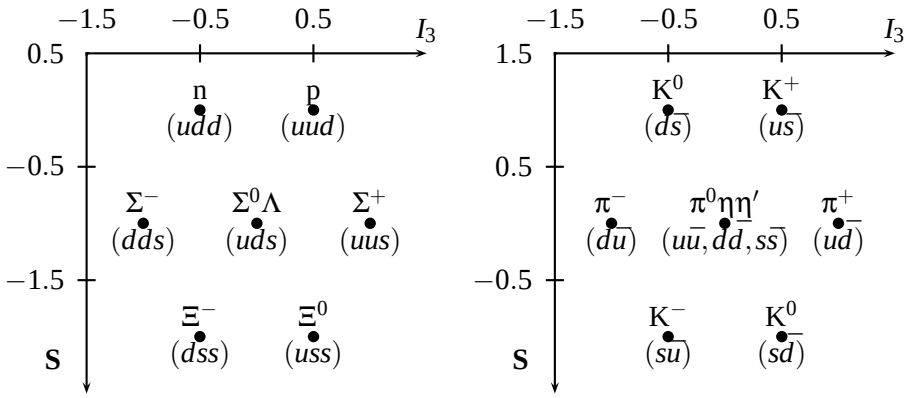


Figure 1.1: Particles in Standard Model. Green shows the leptons, magenta the quarks and in red the force mediating bosons. On each box there is written the mass of the particle, its electric charge, its spin, its abbreviation and its name (values [46], appearance [4]).

The theory could be extended to also include the strong force to "the standard model" ($SU(3) \times SU(2) \times U(1)$), but due to the charge of the gluons and the ris-



(a) Baryon-octett (b) Meson-nonett

Figure 1.2: The baryon-octett at spin-parity $J^P = \frac{1}{2}^+$ and the meson nonett at spin-parity $J^P = 0^-$ [22].

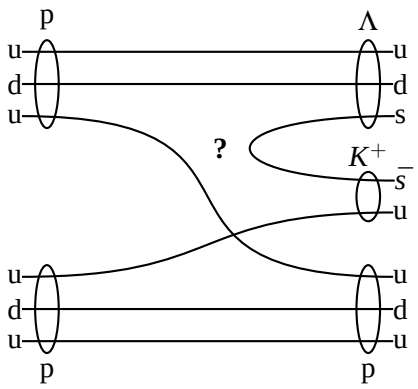


Figure 1.3: The reaction $pp \rightarrow pK^+\Lambda$. Quark-line-model.

ing potential the Quantum-Chromo-Dynamics, QCD is perturbatively very hard to solve and is subject to recent research efforts.

The particles however come as already mentioned in two groups, Quarks and Leptons. Quarks carrying color-charge and Leptons being color neutral. The leptons' three families are called electron, muon and tauon, where one part of the doublet has charge -1 and a mass between 0.511 *MeV* and 1777 *MeV* and the other is the corresponding neutrino with a very small mass and no electric charge. The quarks in their doublet have fractional electric charge ($\frac{2}{3}$, $\frac{-1}{3}$) and don't exist freely due to the nature of the strong force that they are subject to.

Today we know, that the particles we can observe such as protons, neutrons and, as described in this work, hyperons are made up of quarks and gluons. The gluons being charged cause the potential to be linearly rising. This is the reason that single quarks can not be observed but are confined in color-neutral objects. These color

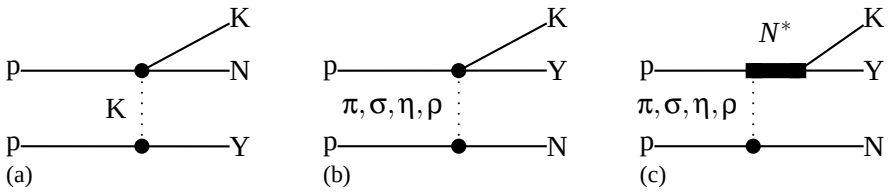


Figure 1.4: Meson exchange model using either Kaon (a) or non-strange mesons (b) including N^*-Resonnances (c).

neutral objects are either mesons ($q\bar{q}$) or baryons (qqq) or other exotic objects like ($qqqq\bar{q}$), ($(qq)(\bar{qq})$), ($q\bar{q}g$) etc.
Taking the lightest baryons with spin $\frac{1}{2}$ and parity $(-)$, they can be arranged, having the strangeness (number of strange quarks) on the y-axis and the z-component of the iso-spin on the x-axis, in an octet (fig. 1.2(a)). Here there are beside the nucleons (proton and neutron) the hyperons containing at least one strange quark. For the state with strangeness -1 spin and iso-spin of u- and d-quark couple to an iso-spin-singlet (Λ) and an iso-spin-triplet-state ($\Sigma^-\Sigma^0\Sigma^+$).
For the purposes of describing ordinary matter and matter, that is also easily accessible in medium-energy-experiments, it suffices to take into account only the three lightest quarks: u, d and s. Only for high-energy experiments the charm-quark has to be taken into account, adding another dimension to figures like fig. 1.2. For even higher energies, also bottom and top quarks start to play a role.
The same way as for the baryons the mesons can be arranged into a nonet (fig. 1.2(b)).
The mass of the naked quark – with $\approx 3MeV$ for u- and d-quark – is quite small compared to the mass of the nucleon of $\approx 1GeV$ or even the mass of the lightest meson, the pion ($m_{\pi^0} = 135MeV$). The rest of the mass is generated dynamically, by the gluon field, the sea-quarks and other effects.
Strangeness was first observed in 1947 ([37]) in an experiment to examine cosmic rays. Here "strange" particles were observed, that were generated only in pairs with lifetimes much longer than expected, meaning their decay mechanism being much different from their production-mechanism ([49]).
Today we know: Strange baryons (Hyperons) are produced, when a s-$\bar{s}$-pair is excited from the vacuum. Let's introduce a new quantum number – the strangeness; the strange-quark having $S = -1$ and the anti-strange-quark $S = +1$. Exciting a $s\bar{s}$-pair from the vacuum conserves strangeness: before S = 0 and afterwards $S = (-1)+(+1) = 0$. The decay on the other hand involves a transformation of an $S = \pm 1$-state to an $S = 0$-state and is only possible via the weak interaction that can transform between the families of quarks and leptons.
To get a closer insight into the interaction between nucleon and hyperon the reaction $pp \rightarrow pK^+\Lambda$ (figs. 1.4 and 1.3) is observed, especially close to threshold where the relative velocities of the particles are small and final state interactions are likely to happen ([17]).

1.1.1 Exotic particles

In spring 2003 a new resonance was reported by the LEPs-collaboration measured at SPRING8 ([43]). A baryonic-resonance with a mass of M=1.540GeV, width $\Gamma = 10$MeV, charge C=+1 and strangeness S=+1. In the usual 3-quark-baryon-model this resonance could not be described, S=+1 hinting to an anti-strange-quark. The only possible quark content would then be $(uudd\bar{s})$ – a minimum of five quarks is necessary giving the resonance its name: penta-quark.
Afterwards a number of papers were released by many collaborations reanalyzing old data and re-measuring the region of interest.
The outcome was twofold. About half the experiments, mainly the ones using photo-production of the hyperon-channel in question, claim to see this resonance. The other half, mainly using hadronic interaction for the production-process (e.g. pp-collisions), give negative statements.
Being especially suited for this kind of experiment, also the COSY-TOF collaboration took part in these efforts. First reanalyzing old data ([44]) and later, in autumn 2004 in a dedicated high statistics measurement ([45]). While the first, low statistic result was still positive with respect to the observation of the penta-quark, the later one only could give an upper limit of the production cross-section.
The trigger used to collect this data did not only enhance the abundance of events with the reaction $pp \rightarrow pK_s^0\Sigma^+$ which was thought to show the penta-quark in the pK_s^0-channel but also the reaction $pp \rightarrow pK^+\Lambda$. The background to the reaction $pp \rightarrow pK_s^0\Sigma^+$ was the reaction $pp \rightarrow pK^+\Lambda$, meeting the same trigger-conditions. This background channel was analyzed in the course of this thesis.

1.2 Motivation for the new analysis program: typeCase

The COSY-TOF-Collaboration is a very loosely bound collaboration, focusing the collaboration on detector development. That gave the reason that prior to the new analysis program **typeCase**, there existed three independent analysis programs and of each several incompatible sub-versions ([2], [27], [1]).
As always in such collaborations analysis programs tend to develop. Every person working with the program includes new parts and new concepts into the analysis software. And with time, there is no compatibility between the different versions of the one analysis program left. The only way to prevent that is a strict version-control with very few people working on one single part of the program at a time. And the possibility to modify the program-flow without modifying the code.
As one may say, the development of different analysis programs, that can analyze only part of the data taken with the detector, instead of only one that can analyze all data, was a waste of time and manpower. But on the other hand it gives us an important and really powerful tool for the analysis of the COSY-TOF data. Since all analysis programs use different analysis strategies, the comparison of the results of the analysis of the data (taking the same data set for all analysis programs), either on the histogram or the event by event comparison gives a deep insight into

the data as well as the analysis programs and their strengths and weaknesses.
The analysis program **typeCase** does not in any way implement new analysis strategies. It is designed as a platform, so one can run his or her analysis strategies, his or her algorithms.
The larger part of designing a new strategy or a new algorithm doesn't consist of actual programming (I'm not taking bug-shooting into account) but of designing a solution for a problem in some kind of pseudo-code. When an idea or a solution has manifested, it is so far not bound to a programming language, though C++, C and FORTRAN are the clear favorites in particle physics.
The new approach uses the main features of Object-Oriented-Programming ([4] "Object Oriented Programming"): Abstraction, encapsulation, modularity, messaging, inherritance and most important late binding. The use of these concepts enables us to create an extremely powerful analysis tool. The feature of late binding (or polymorphism) makes it possible to change the analysis strategy without compiling the program anew.
In previous approaches also using object-oriented-principles ([2], [27]), the analysis, better the calculation of the event-properties was done in member-functions of the data-structures and the detector geometries were, up to some point, hard-coded. This resulted in the problem, that it is not at all easy to switch between different analysis strategies and making modifications to the detector setup had to be cross-checked several times to be sure, that the modification one made was really the intended one and that it was made in all parts of the program consistently.
I followed a different approach:

- No global variables.
- All data – detector-setup and analysis-strategies – are read during run-time from external files.
- Data-structures and calculation of their values were separated.
- Data-structures were made simple and general.
- HTML-reference documentation on the Web.
- Calculation of the data-structures' values were encapsulated into classes: algorithm classes.
- Abstraction, inheritance and polymorphism are used for the algorithm classes.
- Graphical User Interface for easy use and easy extension.
- Version control.

This approach has several advantages: the analysis itself is strongly modular; the modules can be distributed among the available developpers for modification. The problem of two persons working on the same piece of code is reduced. Different

analysis-strategies are kept in parallel, so the strategies can be chosen depending on the actual detector-setup and reaction-in-question. The analysis software is independent of the detector-setup, though if other geometries shall be used it may be possible that the available shapes have to be restocked. It is easy to extend the analysis-software with additional modules or shapes.
The name "**typeCase**" was found suitable for the program concept.
The next two chapters will focus on the development of the analysis-program **typeCase**. In this course, the COSY-TOF-detector will be described at full length in contrary to the common practice only to describe the currently used setup. The way the detector developed over time, as well as its modularity and flexibility gave constraints to the design of the analysis-program as a framework. The concepts and strategies to design the **typeCase**-analysis-framework will be described in chapter 3.
The analysis of the measurement of the reaction $pp \rightarrow pK^+\Lambda$ will be described afterwards. Since the COSY-TOF-detector had been described in such detail earlier, the description of the actual setup during the measurement of the data analysed for this work will be a reference to the detector-components that were used; the actual geometry is listed in Appendix E.
Chapter 7 will present the results of the analysis, starting with the simulations done to describe the data, total cross-section considerations, description of the event selection, finishing with the differential distributions of the reaction $pp \rightarrow pK^+\Lambda$.
The Appendix contains used formulae, the description of the used kinematical fit, the – already mentioned – detector-geometry listings, the differential distribution of the unfitted values of the reaction $pp \rightarrow pK^+\Lambda$ and last but not least a more detailed program documentation for **typeCase**.

Chapter 2

COSY-TOF detector

The COSY-TOF detector is a very modular detector. Many components have been build and assembled in different ways to make the – then current – configuration of the COSY-TOF-detector (fig. 2.10). Unlike other theses I will not restrict myself to explaining the setup that was currently used, when the data I was analyzing for this work was recorded. The purpose for this is to show the modularity of the detector and the differences in the way the detector can look like. This mainly gave the reason for the way the typeCase analysis framework is designed and implemented: Flexibility.

2.1 COSY-accelerator

The COoler SYnchrotron accelerator is a storage ring, delivering a proton- or deuteron-beam with a particle momentum range (for protons) from 270 up to 3300 MeV/c. It is located at the Forschungszentrum Jülich. It's name shows already its main feature, the low momentum spread and low emittance which is achieved by phase space cooling.
Beginning with H^--ions or H_2^-, that are accelerated in the JULIC-cyclotron, which is used as pre-accelerator to an energy of 40 MeV, the protons (deuterons) are stripped off their electrons and injected into the COSY-ring for further acceleration. The accelerated protons or deuterons can be used for experiments either inside the ring such as COSY-11, ANKE or WASA, or extracted off the ring and used at an external experimental site, such as BIG-KARL or TOF (fig. 2.1).

2.2 TOF-detector

The TOF-detector is a non magnetic Time-Of-Flight-spectrometer, which delivers – after analysis – direction vectors and the flight time of the detected particles. It is located at an external beam-line. It consists of a vacuum tank, three meters in diameter. Its length can be varied from one meter up to three meters (nine meters was planned, but not yet built). The whole detector is a modular system; there

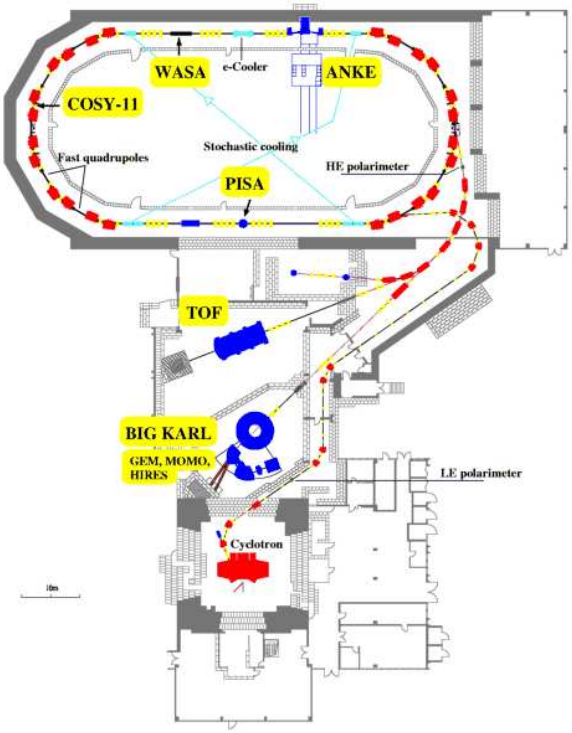


Figure 2.1: The cooler-synchrotron with the pre-accelerator, the COSY-Ring, internal and external experiments. [28]

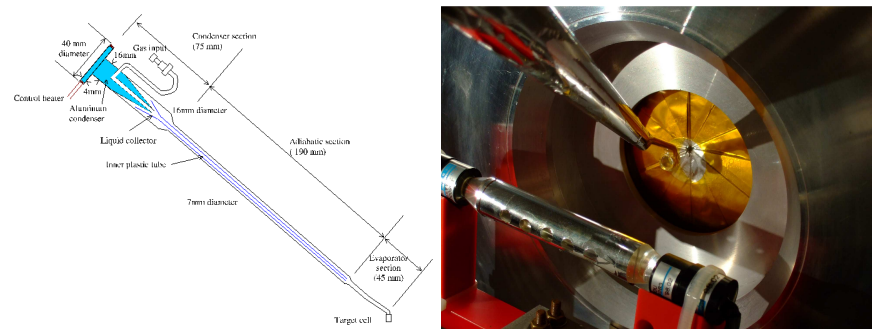
exist sub-detectors, that can be plugged in or left out according to the physics being studied. In any case one can separate three different regions. The first one is the target system, next the start region, that gives the start time for the time of flight measurement, and the stop region giving the stop signal for the time of flight measurement. There is also a calorimeter available, after the stop-region up to 50 cm radius around the beam-axis, giving also kinetic energy for stopped particles.

2.2.1 Target

Currently an unpolarized target is used. It is usually equipped with liquid hydrogen or deuterium, but solid materials (like lead) can be used as well, if density effects on probed reactions are of interest. An additional target was planed to be build for double polarized measurements (both beam and target being polarized), but not yet realized.

Unpolarized target

The unpolarized target system for the COSY-TOF-detector has been developed at the Forschungszentrum Jülich along with the target of the neighboring detector



(a) (b)

Figure 2.2: The target system as it is used in COSY-TOF. In 2.2(a) there is a schematic drawing of the target ([24]), 2.2(b) shows a photograph of the target region (Picture: M.Krapp, 2008 [12]).

BIG-KARL ([24]). There were several requirements that had to be met. First it should provide a high luminosity. The TOF-detector is located at an external beam line and therefore a very small beam spot (2 mm in Diameter) with an excellent duty factor is provided by super-slow resonance extraction. The requirement of high statistics of the specified reaction entail high luminosities and therefore high area densities.
A thick target would solve that problem, but on the other hand enlarge the energy loss the particles undergo in the target volume. So a thin target has to be used.
A small target on the other hand gives the big advantage of a well defined vertex position. This results in a better resolution of the directions of the particles and also gives the possibility to geometrically resolve delayed decays of unstable particles like hyperons and mesons.
Next requirement is that as little shadowing effects as possible shall take place. The windows through which the beam enters and exits, have to be as thin as possible but also planar. The cooling system has to occupy minium space and minimum solid angle.
The target thickness has to be constant, so the target volume has to be filled without bubbles. The shape of the target-system was defined by the need to cool the material down to a working temperature of 2.7K in as little time as possible and preventing connections and cooling to be in the way of the particles to the stop-detectors. The target cell is of cylindrical shape of 6 mm diameter and 4 mm length with 0.9 μm thick mylar windows at the front and the back side and 60μm copper-walls at side and bottom (fig. 2.2). It sits on the end of a copper-tube that is connected to the cooling copper-head. To prevent the generations of bubbles in the target volume, the copper tube is separated by a thin aluminum separator into an upper, up-flowing part for the gas and a lower, down-flowing part for the liquid. During operation the target region is evacuated with a remaining pressure of 10^{-7} mBar. To separate it from the vacuum tank with an operation vacuum of 10^{-3}

mBar, there exists a separation foil of 30μm thickness ([38]).

2.2.2 Sub-Detectors

The parts of the TOF-detector are, as earlier mentioned, designed to be exchangeable between beam-times, to use the minimum amount of material needed to probe the physics giving the motivation of the experiment. This is done to increase precision, since every layer of material changes the kinematical parameters of the particle going through. The detectors are divided into sub-detectors, which can be mounted or dismounted. Furthermore they are divided into two regions of position according to their functionality. There is the start and the stop region. For higher precision, the start region should better be divided into start and tracking region, but due to historical development this was not the case. The exact dimensions and other properties can be found in table E.1.

2.2.3 Start region

The start region gives the start-timing for the time of flight measurement. It does not trigger (better: start) the time-measurement in electronics, the stop detectors do this. A sketch of the start and tracking system used to record the data for this work is shown in fig. 2.3.

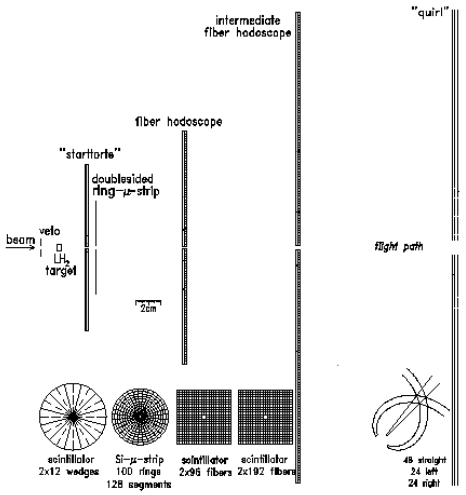


Figure 2.3: The “Erlangen” start system [12].

Start detector/ Start Torte

The Start detector, by some called Start Torte, is a circular plastic scintillator segmented into 12 wedges (pizza pieces or cake pieces: Torte = German for cake). It is located downstream from the target. It has two layers, each 1 mm thick, which are

rotated against each other by half a segment, which is 15 degrees. This results in an azimuthal resolution of $\Delta\phi = 15^{\circ}$. The outer radius is 76 mm, the inner radius, cut out for the beam, is 1 mm. The signals are read out by photomultipliers. The Start detector is used for the determination of multiplicity and therefore for trigger electronics for the specific event.

2.2.4 Tracking region

The following detectors (downstream the beam-line) are not used for time of flight measurement, but provide much more precise angular information of the tracks than the stop detectors can. Therefore I named these detectors tracking region. A sketch of the start and tracking system used to record the data for this work is shown in fig. 2.3. From 2008 on one could choose between several setups:

- stack of several hodoscopes[1] and micro-strip detectors
- straw-tube-chamber

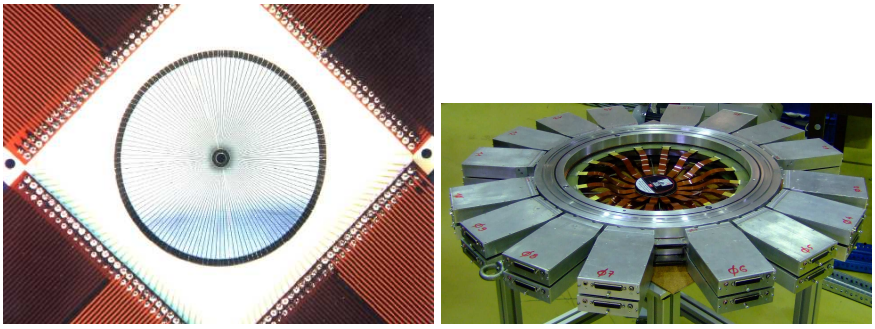
Micro-Strip (ring)

The silicon micro-strip detector is a 520μm thick circular silicon plate with an outer radius of 31.0 mm and an inner radius of 3.1 mm. It is segmented both in 128 wedges and in 100 concentric rings, read out at both sides, front and back (fig. 2.4(a)). These elements are connected via very thin gold wires and Kapton-bands to preamplifiers outside the vacuum tank. The segmentation gives 12800 pixel and therefore a very fine resolution. The energy information can be used for energy-loss calculation and thus for particle identification, though this is not very precise ([18]).

Micro-Strip (spiral)

In 2008 a new silicon micro-strip detector was added to the TOF-setup. It is also a 520μm thick circular silicon plate with an outer radius of 31.0 mm and an inner radius of 3.1 mm. This one is segmented into 256 archimedian spirals on both sides bent in different directions back and front. Connections are the same as for the Micro-Strip-Ring, but due to the enormous amount of channels, each two channels are read out together, resulting in a total of 256 channels for this detector to process (fig. 2.4(b)). The segmentation gives 65536 (16384) pixels. Due to construction these pixels are all the same size, not in Cartesian, but in spherical coordinates ([48]).

[1] A hodoscope is a planar stack of thin detector elements, extended in one spacial dimension. Here it is long box-like fibers.



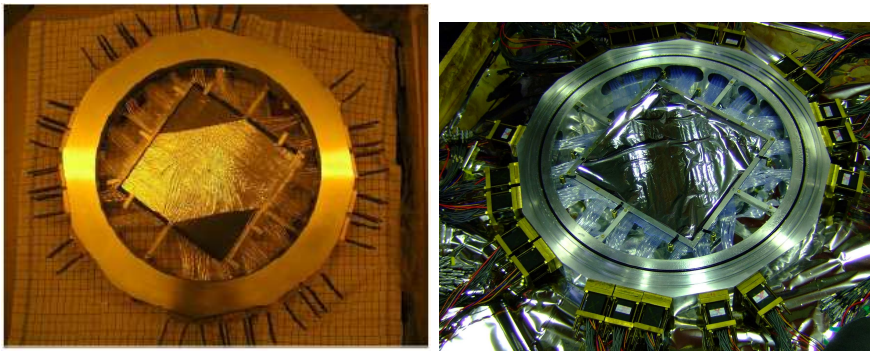
(a) (b)

Figure 2.4: Photographs of micro-strip detector before mounting ([30], M. Krapp [12]). 2.4(a) shows the bonding of the wedge shaped side of the Micro-Strip-Ring, 2.4(b) shows the one complete Micro-Strip-Spiral detector with bonding, support structure and preamplifiers [12].

Three-layered Hodoscope

This hodoscope has been modified several times in the past ([42]), the newest modification taking place before the beam-time in October 2004, where it has been extended in size and a third layer has been added (fig. 2.5(a)). The hodoscope now consists of three layers of plastic scintillator each one 2.0 mm thick, located about 100mm behind the target. All layers have fibers as elements, meaning they are box shaped with a quadratic base shape and a height, that defines the overall length of the sub-detector. The layers are rotated around the beam axis. The first layer viewed from target is the so called D-layer (D for diagonal). The 136 elements of this layer have all the same length of 272.2 mm giving the layer a quadratic shape. The elements are rotated by 45.375° around the beam-axis from the detectors x-axis.

The other two layers, called Y and X, are oriented perpendicular to each other, giving a rotation angle from the x-axis of 0.375 and 90.375°. The two layers are identical in shape except for their rotation angle. The elements have all different length, so each layer overlaps the D-layer by 75%. These layers have been the original layers of this hodoscope, with a quadratic shape with a length of 191 mm, so the D-layer covers the whole two existing layers. The elements of the X- and Y-layer have been extended to fit the edges of the D-layer. This results in half of the area covered by three layers, the other half by two layers. Additionally each layer has one element, right in the middle of the layer which is cut in half to allow the beam to pass through without signal. The elements are made of plastic scintillator and read out by photomultipliers. As signals the energy loss but no time signal is recorded.



(a) (b)

Figure 2.5: Photographs (M. Krapp [12]) of the three-layered hodoscope (2.5(a)) and the two layered hodoscope (2.5(b)) before mounting [12].

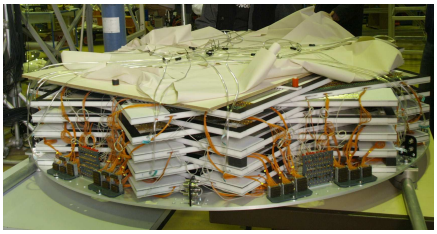
Two-layered Hodoscope

The two layered hodoscope is rather simple compared to the three-layered one, here all elements of both layers have the same length of 383.8 mm with width and thickness of the fiber elements of 2.0316 mm. This results also in a quadratic shape of the layer since here two elements are halved to form a beam hole (See fig. 2.5(b)). The layers of this hodoscope are also rotated from the x-axis around the beam-axis by -33.85° and 63.85°. The elements are made of plastic scintillator (BCF12) and read out by photomultipliers. As signals the energy loss is recorded ([29]).

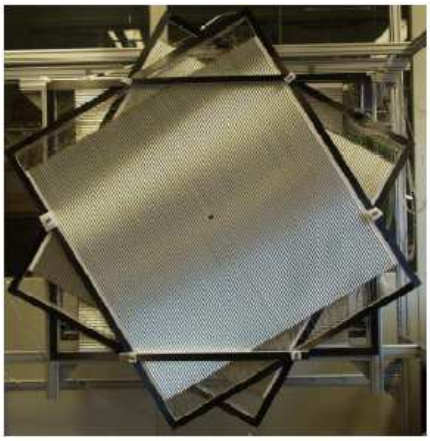
Straw-Tube-Chamber

The straw-tube chamber is an upgrade of the detector system, newly (2008) done to improve the resolution of decay vertices. It consists of 15 double layers of 208 proportional chambers each (fig. 2.6). The single straw is 1050 mm in length, 10 mm in diameter and has a wall thickness of 30 μm Mylar. Each double layer is tilted by an angle of 60° against the preceeding one, to allow three dimensional track reconstruction. Here, too, a beam hole of 15 mm diameter exists. Due to the rotation angle the active detector volume is almost cylindrical with a diameter of 1000 mm and a length of 300 mm. The need of heavy frames to keep wire tension is overcome by overpressure within the tubes and gluing the tubes together in one double layer. So the weight of the detector could be reduced to 15 kg for the detector and 15 kg for its supporting structure ([33], [13], [7], [36], [35], [34]).

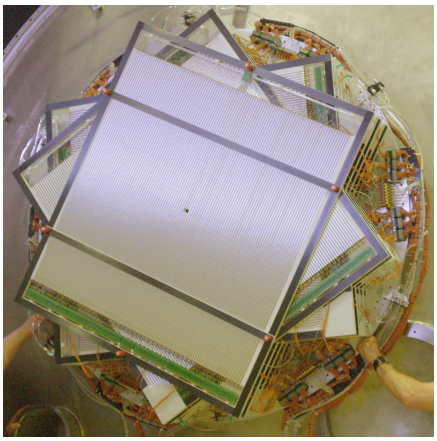
(a)



(b)



(c)



(d)

Figure 2.6: Straw tube tracker. (c) shows the three different orientations of the layers ([34]), in (a) one can see a dismantling single straw tube ([33]). In (b) and (d) one can see the 15 double-layers of the actual Straw tube tracker just before mounting and being fixed to the start cap of the TOF-detector ([12], [34]).

2.2.5 Stop region

The stop region consists of three different detectors, namely Quirl, Ring and Barrel detector, each giving a timing signal, that is used for the time of flight measurement and an energy-loss information. The hits in this sub detectors trigger data recording. Both Quirl and Ring detector are built in principally the same way, with different building parameters.

Quirl and Ring detector

The Quirl and the Ring detector are both three-layered plastic scintillators read out by photomultiplier tubes outside the vacuum tank. The sub-detector overall shape is a circular one. One of the layers has wedge shaped elements (as the Start detector

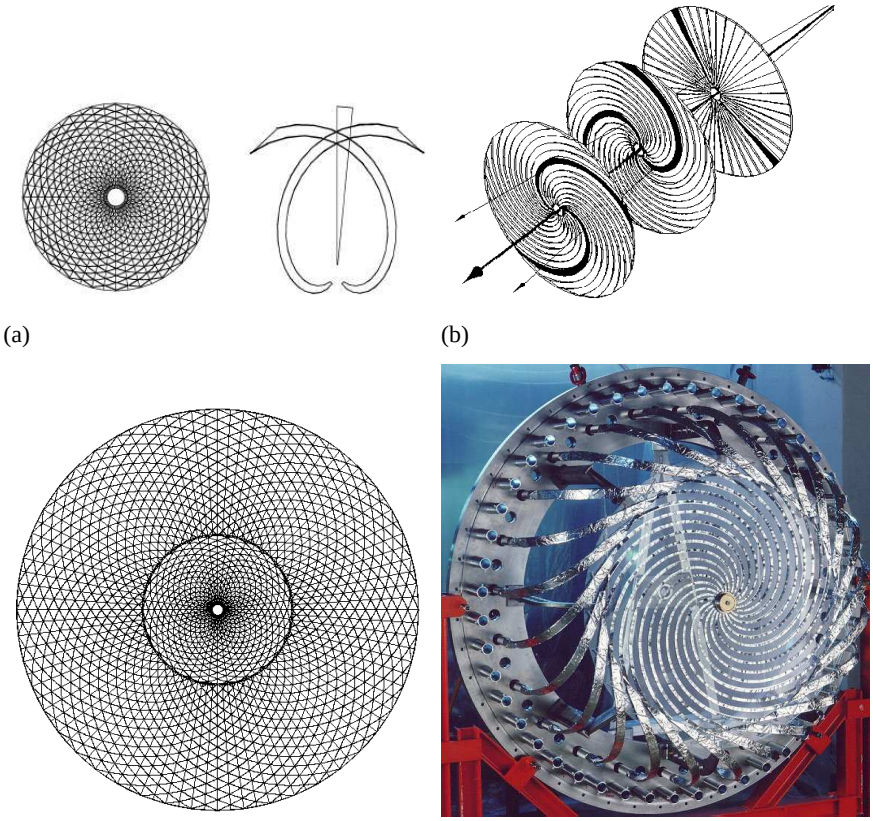


Figure 2.7: The Quirl and Ring detectors. (a) shows a sketch of the Quirl detector. Here a projection of all three layers and the construction of a pixel out of three hit elements in different layers is shown, more clearly even in (b). (c) shows the projection of both Quirl and Ring elements. A Photograph of the Quirl detector is shown in (d) ([40], [21], [25], [12])

and one of the Micro-Strip-Ring sides, see sec. 2.2.4 and sec. 2.2.3), giving only azimuthal but no polar angular information (fig. 2.7(b)). In order to get information about the polar angle θ, two additional layers have been added to construct pixels with fine polar- and azimuthal- angular information. These layers have elements that are shaped as Archimedian spirals with the following edge function:

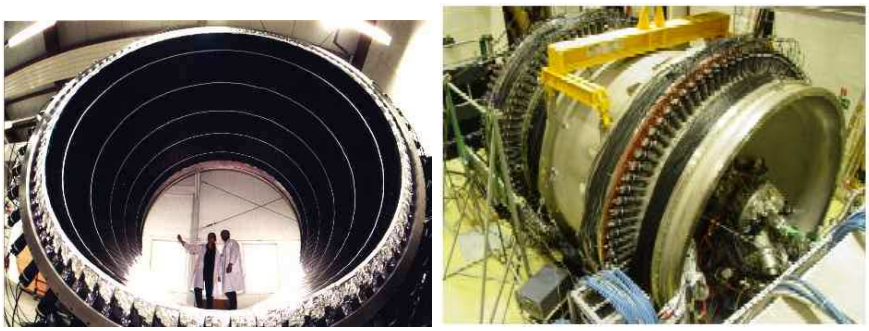
$$r_\varphi = \frac{\Delta r}{\Delta \varphi} * \varphi = \frac{r_{max}}{\varphi_{max}} * \varphi = b * \varphi, \tag{2.1}$$

where φ_{max} is the azimuthal angle from the center point of the sub-detector to the actual position. Two of the three layers have the same absolute bending b but different sign. This results in pixels with increasing area with θ, but that have a constant width in polar angle θ. The wedge shaped elements cut these pixels again in half

to increase resolution in azimuthal angle φ. Since the layer with the wedge shaped (straight) elements has twice as many elements as the other layers, the triangular pixels are unique in which elements they consist of.
The thickness of individual layers for both Quirl and Ring is 5 mm each. The Quirl detector consists of 48 elements in the straight layer and 24 elements each in the bent layers, its inner radius, for the beam, is 42 mm, its outer radius is 580 mm. The maximal φ is 180°, so the value for the bending $b = 184.62\frac{mm}{rad}$. Spacing between two layers is 6 mm (fig. 2.7).
The Ring detector has an inner radius of 568 mm and an outer radius of 1540 mm, so there is a Δr of 12 mm where Quirl and Ring are overlapping. Certainly only in central projection, (in projection) from target, for the short setup (see fig. 2.10(c)) there is a gap between Quirl and Ring. The bending b for the Ring is larger than for Quirl, namely 618.762 $\frac{mm}{rad}$, so the maximal bending angle for an element is 142.6°. The elements of the Ring detector begin at a bending angle of 52.6 °, so each element covers 90° and it does not intersect with all other elements but has only half of the number of intersection points: 24. This results in 2304 pixels.
Material-Quirl: Bycron-BC404: n_{mean} = 1.58, $\alpha_{tot,reflex}$ = 39.3°, c_{max} = 19.0cm/ns, density 1.032g/cm, H/C-Ratio = 1.1, $charge_{eff}$ = 5.612, A_{eff} = 11.157, Radiation length = 43.5cm, Absorption length = 125cm.
Material-Ring ([16]): Bycron-BC408:n_{mean}=1.58, $v_{inscinti,wedges}$ = 18.6cm/ns = 0.62c, $v_{inscinti,spirals}$ = 14.9cm/ns = 0.497c

Barrel detector

Originally the Barrel detector was planned as three 3 m long barrels, almost cylindrical in shape to extend the distance of target to Quirl/Ring-detector to up to 9 m. It was designed to be comparable to the Quirl and the Ring detector in design, also three-layered with one layer giving information about the azimuthal angle and two layers of bent elements to provide information about the polar angle. Instead of the flat design of the Quirl- and Ring- detector, the Barrel layers should be tilted, to form a cone-stump-mantle, close to a cylinder. The radius of the Barrel cylinder is larger close to the target, to make it possible to connect the three identical Barrel modules without the loss of acceptance (1553.5 mm to the front, 1488.5 mm at the end). Contrary to the plans, only one layer of plastic scintillator was realized instead of three. To gain information about the polar angle, this being the pizza piece layer, the 96 elements are read out on both ends. Each one is 2854 mm long, 15 mm thick and wedge-shaped (fig. 2.8). The position of the hit is calculated out of the time difference of both time-signals. The resolution therefore is quite poor, but with the other sub-detectors, this can be improved.



(a) (b)

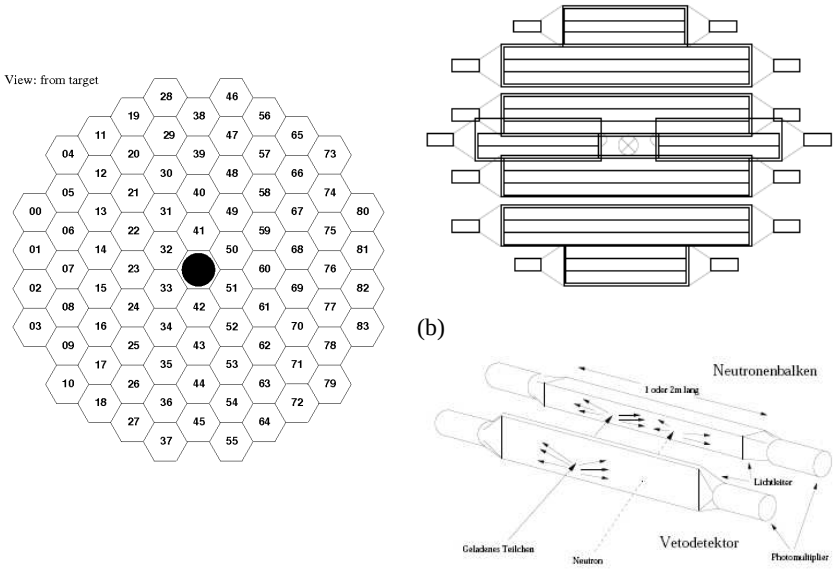
Figure 2.8: Photographs of the TOF-detector. (a) shows the inside of the Barrel detector, (b) the complete TOF-detector with front-cap to the right. The photomultiplier-tubes for the Barrel are visible as black spikes heading to the vacuum-tank ([12]).

Calorimeter

The calorimeter (fig. 2.9(a)) is the only sub-detector in the TOF-detector-system, that can be called a thick detector, a detector, that is able to stop the average charged baryon or meson. It is also the only sub-detector, apart from the COSY-NUS detector (see sec. 2.2.5) that can detect neutral particles. It is located directly behind the Quirl-detector and covers its area in central projection. It is made of 84 hexagonal prisms each 450 mm thick, with a key-width of 140 mm, leaving one prism in the center out for the beam. This shape and arrangement leaves an almost circular shape with a minimum diameter of 1260 mm and a maximum diameter of 1421 mm for coverage. Protons up to a velocity of 0.65 c can be stopped (pions: 0.88 c). The individual elements are made of plastic scintillator (BC-416) and read out by photomultipliers ([21]).

Pad detector

The pad detector is designed to increase acceptance in the region close to the beam-axis. The Quirl-detector has a minimum distance of 42 mm to the beam-axis. In the short TOF configuration of about 1 meter this results in a minimum polar angle of 2.2°. To be able to detect particles with a smaller polar angle, the pad detector has been installed, within the inner radius of the Quirl-detector. The inner radius of the pad-detector is 10 mm resulting in 0.5° in the short TOF configuration. The pad detector also is made of plastic scintillator, that is read out by photomultipliers. The shape of the individual elements is wedges, but in contrary to the other pizza shaped detectors, this one has an inner ring with an outer radius of 20 mm and an outer ring with an outer radius of 43 mm. The inner ring is segmented into 4 elements the outer ring into twice that number.



(a) (b) (c)

Figure 2.9: (a): Calorimeter. Sketch of the front areas of the elements, along with the numbering of the elements; view from target ([21]). (b) and (c): The COSY-NUS detector, a single element (c), the total arrangement (b) ([5]).

COSY-NUS

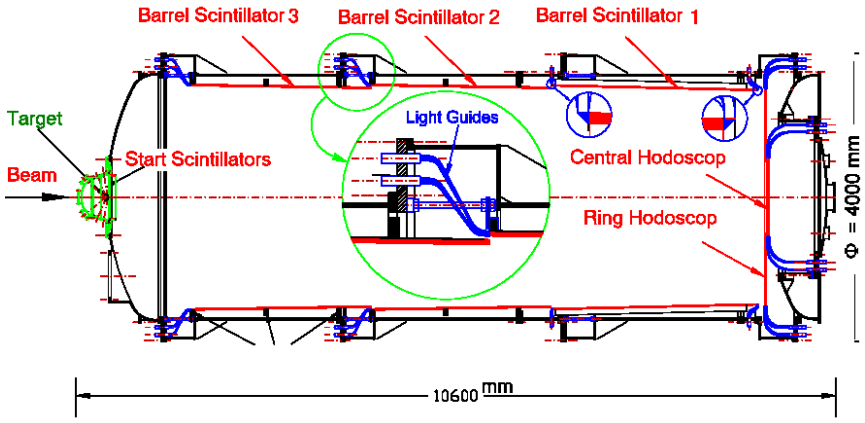
To be able to detect neutrons with the TOF-detector-system, there has been developed a separate detector concept for detecting neutrons at TU Dresden. It is a system of several scintillator blocks outside the vacuum tank (fig. 2.9(b)). A further description can be found in [5].

Beam hodoscope

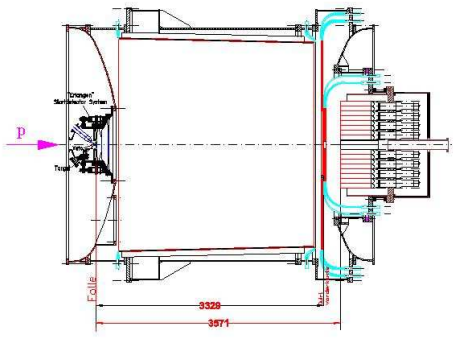
Behind the TOF-detector, there is a beam hodoscope installed, that is made of two layers of 32 elements of plastic scintillator fibers each. The fibers have as the other fiber hodoscopes a quadratic base shape with a edge length of 2 mm, that are read out by photomultipliers.

Veto detector

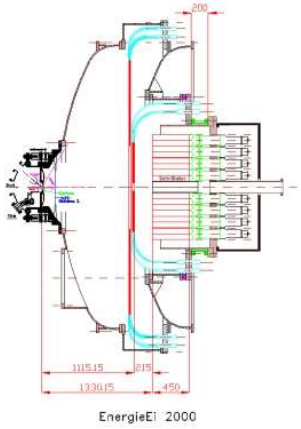
The reality of the beam shape is, that it is not only made up of a beam spot of 1 mm in diameter but also of a so called halo, that makes these beam particles pass sub detectors and create signals, that can falsely trigger an event. To prevent this the veto-detector-system has been installed. It is made of several rings of plastic scintillator, read out by photomultipliers. First ring is 900 mm in front of the target with a central hole of 15 mm, second is 500 mm in front of the target with an 8 mm central hole. These two are called ”Molnar”-vetos. At a distance of 50 mm in front of the target there is the third one, the ”Wolfi”-Veto, with a central hole that can be chosen from 1.5 to 3.5 mm.



(a)



(b)



(c)

Figure 2.10: Auto-CAD drawings for the different possible configurations. Subfigure (a) shows the nine meter version, (b) and (c) the actually built 3 meter and 1 meter versions ([12]).

Chapter 3

Program package typeCase: Data Analysis

3.1 Ananlysis

In particle physics, the results are sometimes less easy to find than in other fields of experimental physics, where results sometimes just can be read from a display of the measurement electronics. The data an experiment in particle physics provides today, raw and unprocessed, consists of signals of lots of detector-elements with different meanings. This data has to be recorded[1], perhaps already filtered by triggers. Only part of this data is what we are actually interested in, so we have to reconstruct what happened in the detector and more accurately filter for the events of interest (cuts). And even after that, data is still convolved with acceptance and efficiency.
The analysis is the engine that connects the unprocessed data to the final graphs of efficiency- and acceptance-corrected observables. It has to read raw data and calibrate the signals (assign physical meaning to so far arbitrary numbers). It finds tracks in the detector; signals on the path of a particle where it deposited energy in a detector-element. For these tracks, direction (preferably at production –vertex– position), velocity, momenta, energy are calculated if the available information suffices. It may be possible to identify the type of particle that generated that track. A kinematic fit may be used to force the measured properties to fulfill energy- and momentum-conservation. In the end the observables can be calculated out of the retrieved 4-momenta and filled into histograms.
And still these will not be the final results because these histograms are still convolved with the efficiency and acceptance of both detector and reconstruction. So simulations have to be made: particles are generated according to some model and transported through a virtual detector, where detector-response is simulated. And again the same procedure as for data is applied to the simulated events. Now you are in the position to generate a correction for your data and show the distributions

[1] done by the Data AQuisition system (DAQ) that is not subject of this thesis.

of your observables in a way, that they can be compared to other experiments.
All this is called The Analysis. Rather often there are different tools for each step that is mentioned above. Dedicated small programs that are created, designed and used by and for a single physicist. In the **typeCase** analysis framework all of these steps can be done consistently, combined in an integrated framework.[2].

3.2 Concepts

As described in the introduction, the restrictions on the actual program are tough. The main concepts of the program are:

Flexibility The user should be able to analyze all data from the COSY-TOF detector with a single program without compiling the code anew, each time the setup is changed. This requirement, consequently followed, **results in the feature that all possible data from nuclear and particle physics is analyzable with this program**.

Low divergency This requirement can only be met, if a very basic data structure is used, that every programmer can use and just stores all relevant information without processing it. The basic data structures are extremely simple, without any functionality. They act as container classes and have to be filled by other means. There is no need for any programmer to modify the container classes. As for the processing units of code, a modular approach was chosen. Here, an algorithm can be plugged in, if needed or left out, if not. This is possible since the algorithms are capsuled in classes. All are derived from the base class AAlgorithm, overwriting the process-method. By calling this method, one can access each algorithm.
These features keep divergency low, since single analysis parts, the algorithms, can be easily exchanged.

Easy to use Physicists tend to program software, that can only be used and understood by them and the persons who spend quite a long time studying the code. This software should be easy to use, therefore a Graphical User Interface (GUI) was designed. Here, the data to analyze can be selected, as well as the algorithms to use on the data or the detector setup. Help is supplied as for the detector setup, the geometry is graphically displayed, for the algorithms, a description is displayed (which can only be as detailed as the programmer of the algorithm designed it, but any programmer is strongly advised to make this description as understandable as possible, since it will be not only him who is using it) and the parameters of the algorithms to be used can be changed.

[2]Except for the virtual detector engine transporting simulated particles and generating simulated detector response. There are already rather good programs available like GEANT([11]).

Easy to adapt Since the container classes are so simple and well documented, existing algorithms can be easily modified and packed into algorithm classes to fit into the program.

Easy to extend The program is not designed to be finished at some point. It is designed in a way, that any algorithm that can be packed into the algorithm class structure, can be included into the program. An experienced programmer can add and modify algorithms by himself. There exists just a single method, that defines algorithms and that has to be modified, when an algorithm changes its call. For the non-experienced programmer, there is no need to modify the code, when adding a new algorithm. A Graphical User Interface asks about the properties of the algorithm and generates and inserts the code which is necessary. If this has been done already by someone else, the program offers the possibility to read a log-file, that reads the specifications of the algorithm, the user just has to supply a name.

These requirements result in some very powerful features: **portability**. Usually if one wants to give analyzed data to someone else, this is done giving histograms or – in best case – ASCII-files, containing a lot of numbers. Here, since one of the possible outputs are fixed, data can be analyzed to a certain point in one location, transfered to another and be analyzed further on in the other location. But portability of data is just one of the key features. Another is the portability of algorithms. If an algorithm is programmed in one site, it can be easily used in an other, especially if the install-logs (3.12.3) are used. This program can be used not only in the COSY-TOF-collaboration, but for many other experiments too.

3.2.1 Documentation

The components of the **typeCase**-analysis-framework have been documented. A reference documentation for the framework-classes as well as a general description is available on
`http://www.pit.physik.uni-tuebingen.de/~ehrhardt/KTOF`.

3.3 Principles of work

This software was designed using features of Object-Oriented-Programming (OOP). The main concepts of OOP are ([4]):

Encapsulation: Data can be combined into a single structure and can be equipped with methods (functions) that work on this data. But encapsulation is even more than that; it means that data can be hidden to the outside.
For **typeCase**, the information about one specific structure, let's say a hit, has been combined and encapsulated into the class TCalibHit. The variables themselves are not visible to the outside of the class but are only accessible via get-methods and set-methods (getters and setters), taking care that the

values the variables can hold are valid ones. In the **typeCase**-framework there are no public class variables.

Messaging: Here one object sends a message that changes properties or causes the execution of a member-method of an other object.

Modularity: A module is a list of commands that fulfill a certain task (or function) and that is bound into a procedure or function with a certain interface. Modules can be used repeatedly at different points in a program.
This is the main concept of functions or methods. It makes the code easier to read and to see what actually happens.

Data abstraction: In short, data abstraction is the definition of an interface, where only the definition is visible (or even defined) but not the actual implementation. This peaks in the definition of abstract data-types that define methods but do not implement them. You cannot instantiate an object of such an abstract class (but use pointers of that type), but you can inherit from one.
One of the main concepts used in **typeCase**: An interface is defined, components that many types of a kind should have – all shapes have a center-point (sec. 3.8).

Inheritance: A base class defines properties that other classes also have and that will be derived from it. It is a way of reusing code. Let's have a small example: we define a base class "animal". It may have a weight, a number of legs, ears and it can eat and make communicative sound. Other classes (sub-classes) like "pig" or "dog" can inherit these properties and functions from "animal" and declare new properties and functions (like fur color for the dog).
This very important feature gives us the possibility to define inheritance trees, where the child classes inherit all methods and variables from their ancestors. So a *circle* inherits the normal-vector from its ancestor *planeShape* and the center-point from its ancestor *base_shape*.

Polymorphism: or late binding describe the way the lookup of methods is treated: during compilation of code (early-) or during run-time (late-binding). Late-binding provides the possibility to handle pointers of base classes (possibly abstract), pointing to any kind of sub classes; by calling an interface method of the base class, the appropriate method of the sub class is used. In the example above this would mean, that we have a pointer of class "animal". Somewhere else an object of a derived class has been allocated and the address of that object has been stored in the pointer. Our class "animal" probably has defined the method "sound", making one of the animals typical communicative sounds. Though our pointer does not know which animal is in the box, it may shake the box to see which sound may come out: "oink!" This might have been a pig!

Polymorphism is both the most difficult and the most important concept being used for **typeCase**. It is used in many ways throughout the framework. To start with, it is used for the volumes of the hit elements. Each hit element certainly has a volume, but it is unimportant and possibly impossible for the hit to know exactly how the shape looks like and how it behaves, as long as it is derived from the class *volumeShape*. This way it is not necessary to define separate hit-classes for each detector, but all hits from all detectors could be treated with just one class. The same holds for the *TPixel* class containing planar shapes. But most importantly it is used for the analysis strategies, making the program most versatile.

For the analysis-software the concept of polymorphism became the most important one. The class *AAlgorithm* was defined and all analysis-modules are defined as sub classes of the *AAlgorithm* base class. It defines a name and most important the method *progress* that will be called once for every event.

Since the algorithms are no more distinguishable after initialization, all parameters and the data-structures have to be passed via the constructor during initialization.
There are three main steps: initialization, execution and finalization:

Algorithm 3.1 Initialization process for algorithms

```
init setup and data
n←number of algorithms
allocate f :array of pointers of AAlgorithm[n]
for all algorithms i do
  f[i] ← get Algorithm(parameter[i])
end for
```

Initialization In this one step, the algorithm-type is known. All needed parameters and the needed part of the data-structure is passed to the algorithm, necessary by-algorithms are declared and connections (using SIGNAL-and-SLOT-mechanism of Qt) are made (algorithm 3.1).

Execution In this step, which is repeatedly done once for each event, the actual analysis takes place, for example reading from file or pixel-calculation in one sub-detector. Since the definition of the algorithms taking part in this step of the analysis is done dynamically, the actual type of the individual algorithm is not known (algorithm 3.2).

Finalization The last step of the analysis is not important for algorithms like calibration, pixel-calculation or tracking, but it is essential for writing algorithms, where data has to be written and files have to be closed. This step should not be omitted in any case since used memory is freed here (algorithm 3.3).

Algorithm 3.2 Execution of algorithms

```
f :array of pointers of AAlgorithm[n]
for all events i do
  if no more events to process or stop-flag then
    leave loop
  end if
  reset event
  for all algorithms j do
    f[j] → progress()
  end for
end for
```

Algorithm 3.3 Finalization process for algorithms

```
f :array of pointers of AAlgorithm[n]
for all algorithms i do
  free f[i]
end for
finalize setup and data
```

In the following sections the individual parts of the analysis are described, beginning with the most basic one, geometry, ending with the description of the Graphical User Interface.

3.4 Components

Analyzing the data requires two kinds of data-structures to be given to any analysis method: constant structures representing the detector-setup and event-based structures, that change with every event that is analyzed. These structures – virtual detector (*TSetup*, *TDetector*, *TTarget*, *TBeam* and *TMaterial*) and event (*TEvent*, *TRawHit*, *TCalibHit*, *THitCluster*, *TPixel*, *TCluster* and *TTrack*)– are combined into the container-package (sec. 3.9). It still misses the geometrical representation of the detector elements, but the shapes – volumes (*volumeShape*) and planar ones (*planeShape*) – are pooled in a package (sec. 3.8) by themselves to make it easier to extend it or use it by itself.
For definition of the detector-setup, the beam and target, as well as the shapes themselves, the parameter-classes (sec. 3.7) are necessary. They have been extended with descriptions for runs and beam-times as well as the algorithms and the way the detector shall be drawn.
Unfortunately, it proved necessary to implement classes for the geometrical description of the detector. They are combined into a separate package: geometry (sec. 3.5). As well as the other packages this package can be used without the others.

The algorithms themselves are combined in one package (sec. 3.10) for the same reason as the shapes are.
To identify reaction-pattern for events, the reaction classes have been developed. As the shapes they are stored in a separate package, but in order not to make a circular dependence, the basic reaction class was pooled with the basic algorithm classes into the basic-package.
The analysis itself is encapsuled in the class *tofAnalysis*. This class implements the three analysis-steps (algorithms 3.1, 3.2, 3.3), and defines the needed data structures. It can be included into different programs:

typeCase : the full Graphical-User-Interface program (sec. 3.12) with extended and user-friendly definitions of the analysis-parameters.

cl-**typeCase** : a command-line version, being a bit faster than the widget-based version, since the graphical overhead is missing.

- Feel free to implement an own version.

The package dependency (fig. 3.1) and the main outline of their content was designed with the top-down method, but the individual packages were implemented and tested using the bottom-up-method – starting with the most basic class in the most basic package – in order to have reliable data-structures when entering the more complex tasks.

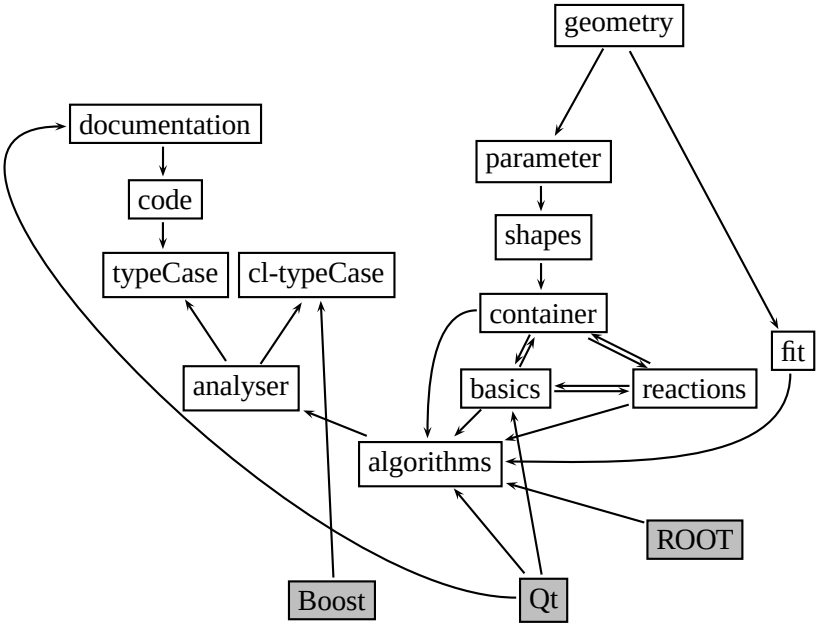


Figure 3.1: Dependence graph of the individual packages of the typeCase-analysis-program. Here the gray filled boxes are the ROOT, Boost and Qt libraries as prerequesites. The open boxes are packages of the typeCase-analysis-program.

3.5 Geometry package

The geometry package has been done especially because the possibilities of the root-mathematical-vectors and -matrices are somewhat limited and partially not reliable. It is a stand alone package and can be used by other programs alone.
The base class of the geometry package is the class geomObject. It contains a status variable. Earlier inheritance of this class from TObject has been removed making this package independent of root.
From this class points and vectors are derived, the latter ones also derived from points. There are general points and vectors, that can have any dimension and specialized points and vectors for 2D- 3D- and 4D- use. Operators for adding, subtracting, multiplying and scalar products have been included.
The specialized points and vectors have special functions to access the data, like Cartesian (X(), Y(), Z(), W()), polar (R(), Phi()), spheric (R(), Theta(), Phi()) and cylindric (Rho(), Phi(), Z()) representations. There exists also an operator to write the data to an output-stream, which is also sensitive to the current representation, as well as a toString()-method returning a string.
The 4D-classes have one prominent member, the momentum4D class. This class provides all functionality of the relativistic 4-momentum. Its *set*-operator is split according to the variable it gets, that can be mass or energy, 3-momentum or velocity. These properties can also be read again (Momentum(), Velocity(), Direction(), Mass(), Energy()). The operators, such as *, + and - have been adjusted. With the boost-method, one can perform a Lorentz-transformation and return the resulting vector. There exists a static method CM(), that returns the center-of-mass frame of the arguments (up to five 4-momentum-vectors).
The package also contains two different matrix types, one is symmetric 3-dimensional, the other one is the general type. The 3-dimensional matrix, matrix3D, is thought as rotation-matrix in 3-dimensional space (which is the most important one if coming to detector setup), but can be also constructed as dyadic product of two 3D-vectors. The general matrix (matrixNxM) doesn't have to be symmetric, as it's name says. This matrix is a mathematical matrix in the sense, that it can be multiplied, added and subtracted to other matrices, it can be transposed, inverted and, what is not possible in the root-library, multiplied to a vector of the right dimension and the right type, since the vectors distinguish between line- and column-vectors.
Last but not least, there are five more geometrical classes in the package, at least when considering 2- and 3-dimensional space. There is a plane, plane3D, that consists of a footing point (Foot()), two direction vectors and a normal vector. The other four are lines, one each limited straight line, and unlimited straight line in both 2D and 3D, lLine3D, lLine2D, sLine3D and sLine2D. lLine3D is the connection between two points and is therefore represented with P() and Q(), and it also has a defined length. sLine3D only holds a direction and one point, that can be shifted along the line. It has no length, since there is no end point to the line. The same holds for the 2D variants of these classes.
Since sLine3D, lLine3D and plane3D are especially done for the representation of

the geometry of detectors, there have been implemented several operators to compute distances of lines, points to lines, points to planes, and hit-points of lines and planes and lines.
To draw 3D-shapes to, let's say, a root-canvas it is necessary to perform a projection of the 3D-coordinates to 2D-space. To perform this a static class projector was defined, that can project in central-, parallel- and fish-eye- projection from an eye-point to a view-plane.

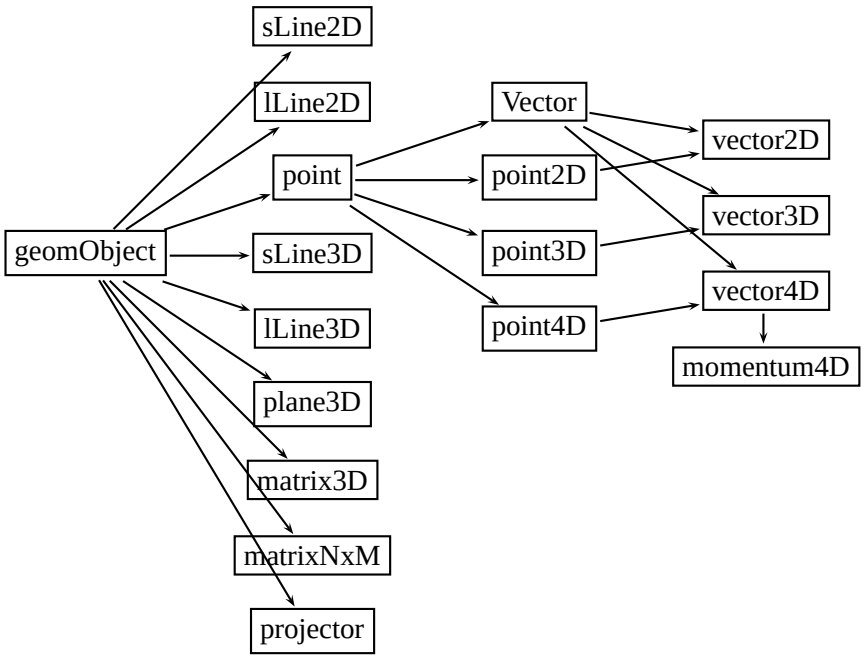


Figure 3.2: Class tree for the geometry package.

3.6 Fit package

This package is a rather small package containing only three but essential classes at the moment. There is a single line fit, fitting in 3D a straight line to a number of points. There is also a multiple line fit. This is interesting if several tracks emerge from the same point each with an own set of points. The vertexFit fits the individual lines to their points in parallel forcing them to have the same footing-point. Though it is possible to fit any number of lines to a common vertex, advice is not to use more than 4 lines at the same time ([6]).
Most essential and difficult one the kinematic fit, fitting three components of a particle with fixed mass in different possible representations (momentum components, energy and angles, momentum and angles, speed and angles, momentum

and normalized x- and y-momentum components, energy and normalized x- and y-momentum components, ...) using momentum and energy conservation. Any component can be set to fixed, measured or unmeasured and additional constraints can be set, invariant masses of decaying particles. This kinfit has been adopted from the kinfit used by the WASA-Collaboration written in FORTRAN. For more detailed information see sec. B.

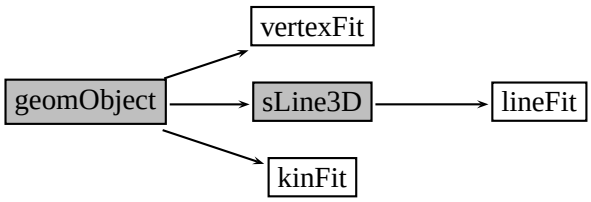


Figure 3.3: Class tree for the fit package. Classes in filled gray boxes are not part of the library.

3.7 Parameter package

The main use of the parameter package is to define in an easy way the parameters of the analysis (detector-setup, algorithms to use, their parameters, which runs to analyze etc.) and provide them with file-In-and-Output. They are used mainly before the actual analysis is launched, reading them from file. Then they are used to initialize the analysis-structures, like the setup and the algorithms.
There is one class, that capsules all possible shapes, *shape_parameter*, one that holds a detector, *detector_parameter*, one that holds a run, *run_parameter*, or a beam-time, *beamTime_parameter*, one that capsules the properties of the beam(s) and target, *reaction_parameter*, one for the materials used in the experiment, *material_parameter* and *element_parameter*, and most important the *algorithm_parameter* class, that holds the name, description, ID and parameter for all the different algorithms.

3.8 Shape package

The shape package is still a very small package, since it does not contain many shapes. The already defined shapes are build considering the COSY-TOF-detector and therfore it is possible that some detector cannot be described by the given volume shapes. But the package is extendable, if the new shapes are fully implemented children of the respective base-shapes. There are two main types of shapes in the package: volume- and plane-shapes.
The base shape of all shapes, volume- and plane-, is *base_shape*. It contains a

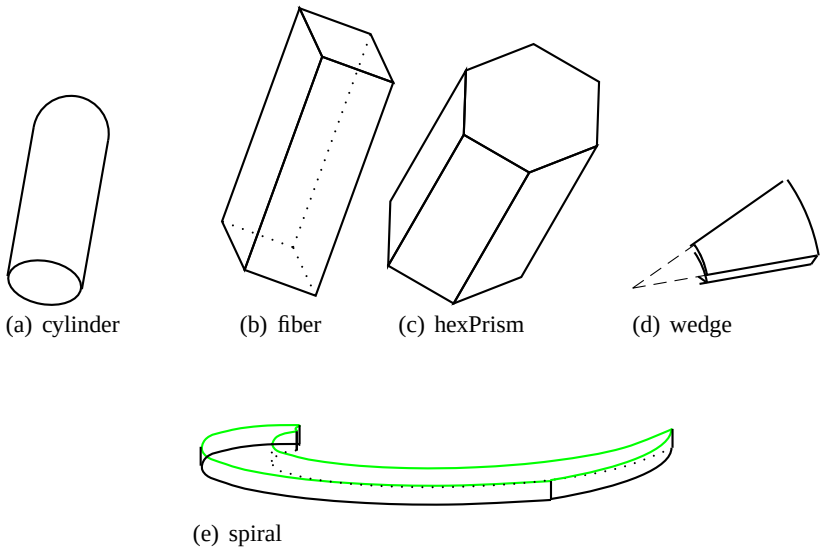


Figure 3.4: Volume shapes: 3.4(a) cylinder (identical with strawTube), 3.4(b) fiber, 3.4(c) hexPrism, 3.4(d) wedge and 3.4(e) spiral.

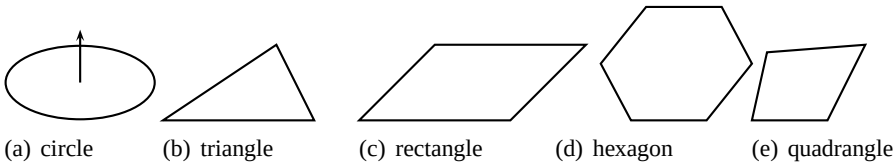


Figure 3.5: Planar shapes: 3.5(a) circle (including normal vector), 3.5(b) triangle, 3.5(c) rectangle, 3.5(d) hexagon and 3.5(e) quadrangle.

center-point.
The volume shapes are especially constructed for use as geometric representations of detector elements. The class *volumeShape* has some placeholder functions, that have to be implemented in derived classes to enable the tracking algorithms for example. The tracking algorithms don't implement the hit point calculation for an assumed track to the volume, but the shape classes do.
These hit-point-calculating methods are optimized for speed – using distance estimations, reducing the number of operations and making educated guesses about the hit surface –, since these methods are the bottle-neck in the analysis.
The planar shapes, the second part of the shape package, are all derived of the class *planeShape*, that declares as place-holder several methods, so the individual type doesn't need to be known. The most important one is getCenter(), that returns the center-point of the planar shape. But there are also angular ranges to be calculated for each shape, from an origin-point along a specified axis.

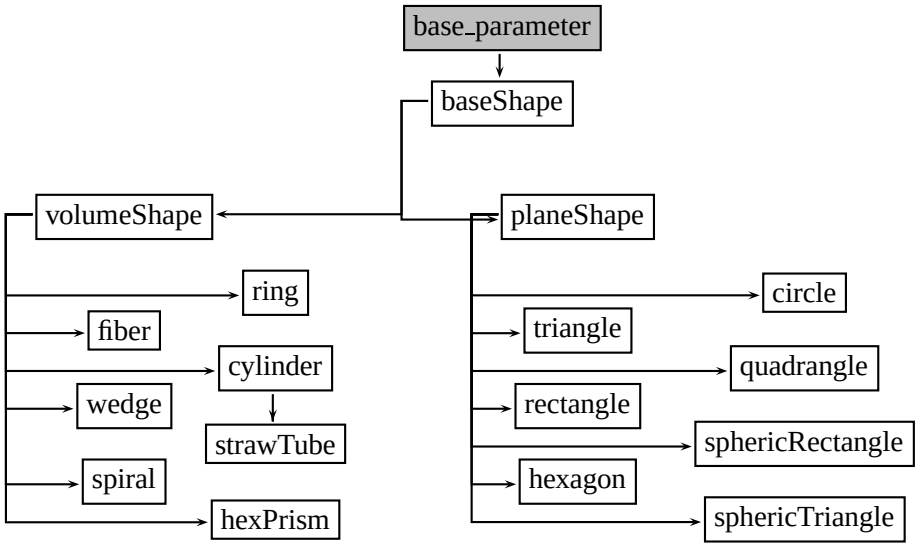


Figure 3.6: Class tree for the shape package. Classes in filled gray boxes are not part of the library.

The volumes that have been implemented already are the volumes, the COSY-TOF-detector is build of: fiber, wedge, spiral, ring, cylinder, strawTube, hexPrism, triangle, rectangle, hexagon, circle, sphericTriangle, sphericRectangle and quadrangle (see fig. 3.4 and fig. 3.5).
For a more detailed description of the shapes see sec. D.2.

3.9 Data container package

The container classes are designed and implemented to follow one single purpose: flexibility. Therefore, contrary to other analysis packages the container classes do not have any methods to generate their properties. One could think of different classes for the calibrated hits, for example, according to the element shape, holding their calibration data basis as static member field.
But that means, that for every new detector a new class has to be programmed, and therefore, a stable version of these classes won't be possible. The hits, pixels, tracks are the same data structure no matter where it occurs. Its content has to be generated outside the container data structure and then filled into the container class variable. This leaves, as only point to modify, the algorithms that do this generation and filling, opening the possibility to use different algorithms without further compilation on the same data structure to compare the results and find differences.
There are two different container class types: the event based container classes,

that are manipulated and generated anew every event, and the setup describing container classes, that are generated once, at the beginning (and whenever the setup of the experiment changes) and used as read only structures during analysis.

3.9.1 Setup describing classes

These classes describe the experimental setup and are therefore kept quasi constant. Their properties are, as for all container classes, not generated by themselves but by the controlling analyzer class prior to the analysis, having parameter classes as input. When the setup changes these setup classes are generated anew.

3.9.2 Event based container classes

The event based container classes are holding the properties of the single event, beginning with the very basic data, the raw hit data, the calibrated values of the hits, to the further processed ones, the pixels, clusters and tracks. In multi-thread application, these classes are created in one instance for each program thread that is used.
For a more detailed description see sec. D.3.

3.10 Algorithm package

The algorithm package consists of several base classes, namely:

- AAlgorithm
  The AAlgorithm is the very base class of all algorithms used in this package. It is derived from the basic Qt-class, QObject, to provide the very elegant SIGNAL-SLOT mechanism for messaging. The AAlgorithm class has one member, or property, that is the name of the algorithm, which has to be provided with the constructor (which means by declaring a variable of this class, a string has to be provided to the constructor containing the name of the algorithm), the methods to access this property and a virtual function that has to be overwritten by each class that is derived from AAlgorithm and that is the function called process, called to actually do the algorithm (see algorithm. 3.2).

- AFitAlgorithm
  The AFitAlgorithm is derived from AAlgorithm inheriting all of its properties. It describes the general fitting algorithm, providing apart from the process-function also the possibility to get data not only with the constructor, but at run time. This is kept quite general so the user should not feel limited by this feature. Last but not least it has a function called fit, that will

do, in derived classes, the actual fitting. With getResult(), the result of the fitting can be retrieved. The fit() as well as the getData() function are declared as SLOT, from the Qt-SIGNAL-SLOT mechanism, the fit() function emits a SIGNAL, so the fit-algorithm and the calling algorithm don't have to know each other.

- AElossAlgorithm
  The AElossAlgorithm is designed to fit the needs of the material classes, to calculate the energy loss of a particle traversing matter.

Subsequently all algorithms that are described here are derived from the AAlgorithm class. Not all algorithms that were developed in course of the analysis of the data for this thesis will be described, but most of the ones that are finally used. Find a complete list attached in sec. D.4.

3.10.1 Data read-in

On different stages of the analysis it may be necessary or useful to store the data acquired by the analysis routines to file, either to make the program more stable by using less algorithms at one step, or if one step is under development one might think of doing analysis up to that point, save, and start the analysis several times (development) with different parameters at that point anew, saving a lot computer time. Or there might be a so called Bottle-Neck-Algorithm, that takes a lot of time (like tracking). Save the found tracks to file and do the reaction recognition in a later step.
On different stages of the analysis, different types of data are available (raw-hits, calibrated hits, pixels, clusters, tracks, reactions,...), so different formats were chosen for write-out and again read-in.

The TADE-format

The data the COSY-TOF detector gives to the experimentalist is not as we would hope the 4-vectors of all passing particles. The detector delivers a voltage for each channel, not only for those that were hit. The voltage signals are fed into the data-acquisition-electronics (DAQ), that decides which voltage actually corresponds to a hit and transforms the voltage into an integer-number (channel) – more precisely the integrated current. Along with that, the arrival time of the signal, the rise of the voltage above a certain level, is recorded and also converted into an integer-number. The output of the DAQ is a binary file, where the data are not sorted by event, but in clusters and the hits are still numbered by input channel, not detector/element wise. This data-format is not read in, but is converted into an ASCII file, that is also readable by eye, by a program called ems2tade, that has been around for quite some time and is being used throughout the collaboration. The output format, the input format (or one of the formats that can be read by this analysis program) is the

TADE format (TDC[3]-ADC[4]-Detector-Element), that has an event header including the event number, the trigger information and the number of hits for this event. All hits are written successively line by line, giving four integer numbers each: the integer timing information, the TDC, the integer energy information, the ADC (or more often used QDC), the ID of the sub-detector and the number of the element, that has been hit.
This information is read in by an algorithm called **AReadFromTade** and stored into the container-data-structure for further processing.

The hit tree format

The root-framework was chosen for its visualization- but mainly for its data-compression-functionality when it comes to writing to file, so most file formats are root-files, containing *TTree*s. Writing the original event-data-structure to root-file was time and disk-space consuming and above all unstable.
So a new format, the hit tree format was developed for the data type hit, writable to a new file. Its main feature is that it writes to several *branches*, reducing disk-space occupancy significantly.

The track tree format

For the track structures there has lately been developed an alternative file structure, distinguishing between different track types (prompt, kink and vees). This file type has been spread (Analysis Meeting 09.2009) in the collaboration and is now a standard output format.

3.10.2 Calibration

To generate the correct energy and timing information out of the integer QDC- and TDC- values is called calibration. There are several steps of calibration:

- Before tracking
  - Apply cuts in QDC and TDC
  - Convert QDC to energy
  - Convert TDC to time
  - Apply a walk-correction to the time-information
- After tracking

[3]Time-to-Digital-Converter, the time signal the electronic provides, given in channels.
[4]Analogue-to-Digital-Converter, the integrated signal size, given in channels. It gives a measure of the deposited energy.

  - Correct the time-information for the run time of the signal in the material
  - Apply a pulse height correction
- After particle Identification: Do a quench-correction

In these algorithms the found parameters are applied to the data. The generation of calibration-parameters as well as the used functions are described in more detail in Chapter 6.

Teufel-Correction In the beam-time October 2004 there has been a significant problem with the QDC-electronics of the Start detector. There were shifts/jumps in the pedestal-position and the amplification almost event-based. The runs have been sorted into categories of their correctability. Note, that the QDC-jumps are critical, since this energy-information also modifies the walk-correction, which modifies the timing-information.
This has been done in the University of Erlangen by A. Teufel, giving the correction its name.
The correction works in steps of 10000 events, applying simply an offset and a factor to the QDC value, setting the pedestal to a value of 1000 and the data peak to 1500. This has to be done before the other calibration steps, since the calibration counts on the correct QDC values.

3.10.3 Pixel-calculation

Pixel calculation generates of several layers of thin, adjoining detectors, planar shapes, on a plane close to the detector. These shapes are characterized by the overlap of several elements, each in a different layer. The overlap area of the elements give the planar shape its shape. This makes it possible to have many pixels with few elements, few read out electronics. For example, two layers, each 100 elements, can, as fibers with perpendicular layers result in 10000 pixels with only the read out of 200 elements.

Pixels in Quirl-shaped detectors

The pixels in the Quirl-shaped detectors, such as Quirl and Ring (sec. 2.2.5) are, in their reconstruction quite simple, as an algorithm, not so simple as shape. The pixels are more or less triangular, so they can be approximated by a triangle. The algorithm provided for the Quirl-shaped pixels also allows a triangle approximation. Actually they are spherical triangles, since the edges of the bent layers of the Quirl don't have straight edges (see pic. 2.7(a) and 2.7(b)).
If any three elements of each a different layer (s... straight element number, l... left bent element number, r... right bent element number) form a pixel, it is defined by

a simple formula $\Delta\varphi = (l + r) - s$. This $\Delta\varphi$ has to be within predefined ranges, here between -1 and +1. The value s, by the way defines the azimuthal angle of the pixel; the value $(l - r)$, the polar angle, with $(l - r) = 0$ being the outermost pixel. The complete formula for the Quirl-pixels can be found in the sec. A.1.

Pixels in hodoscopes

Pixels in two layered detectors have one problem: an overlap of two elements in different layers is not enough to define a real track. Requiring three elements in a three layered detector diminishes these ambiguities, but reduces the efficiency, since each layer has an efficiency of less than one. The first fiber hodoscopes were both two layered, with perpendicularly aligned layers. So both pixels could be calculated by the same algorithm. Since the diagonal layer has been added to the now three layered hodoscope, a different algorithm had to be designed.

Two-layered hodoscope Pixels in the two layered hodoscope are the overlap of two elements from each of the two layers, which gives many pixels, that do not correspond to particle transitions. Nevertheless, the pixel center is the point of closest approach of the two center lines of the fibers in their length direction (the longest one), the shape is a rectangle, the edges are projections of the element shapes to the plane defined by the pixel center and the beam axis.
Unfortunately the shape of the two-layered fiber hodoscope's elements is not exactly box shaped but bent to the outside at the center. The ends of the individual fibers remain fixed. This effect is small but noticeable. It is corrected for the exact position of the pixels.

Three-layered hodoscope First of all the possible area for three element pixels is not the whole detector area, but just half of that (see sec. 2.2.4). So starting with an element of the diagonal layer, intersecting that with an element of one of the other layers, one has to see whether this possible pixel position is inside or outside the three-element-pixel-region. If it is outside, one can immediately define the pixel, with the center point being the point of closest approach of the middle lines, as done for the two layered hodoscope. If it is inside, one has to check whether there is an element hit in the remaining layer nearby. If not, there is no pixel. If there is, the resulting pixel shape can be quite complicated, from a triangle to a shape with five or six corners. This is quite complicated and in the programmers point of view not worth the effort, so the pixel is approximated by a circle with the diameter the same as the width of one of the originating elements.

Algorithm 3.4 Calculation of pixel in 3-layered hodoscope

```
for all hits in diagonal layer i do
  for all hits in x-layer j do
    if intersection is in 2-layered region then
      take pixel
    else
      for all hits in y-layer k do
        if intersection is close to k then
          take pixel
        end if
      end for
    end if
  end for
  repeat with x-y exchanged
end for
```

Pixels in Micro-Strip detector

The pixels in the Micro-Strip detector are quite simple, since only the overlap of two elements, a ring shaped and a wedge shaped have to be considered. The result is again a wedge with the inner and outer radius of the ring element and the φ and $\Delta\varphi$ of the wedge element. The resulting wedge is the projected on the detectors front plane, leaving the wedges front plane, that is called a spheric rectangle (sec. D.2).

Cluster search

When a particle flies through a detector, it may happen, that the particle passes not only one element per layer, but two or more neighboring elements, depending on angle of the particle and thickness of the detector layer. For these cases, there are many pixels calculated for just one passing particle. Here one can collect the pixels belonging to just one hit point, together. This is called a cluster.
The cluster search used in this package is quite a simple one (reflected in its name: Simple Cluster Search). It collects pixels, that contain neighboring elements, starting with the pixel with the largest energy sum of its elements, pixels with directly neighboring elements are added (the “directly” is a maximal element number difference specified with the algorithm-parameter at initialization, this has been set to 1 in the analysis used here). This can be done recursively for the added pixels. (This algorithm is not used for the actual analysis).

Hit cluster

Intuitively a cluster search is done on pixel-level, calculating pixels and then merging neighboring pixels to a cluster. However this is not the only possibility to form a cluster, it can be also done on hit level. While the pixel-cluster-search is a two-dimensional one, the hit-cluster-search is linear, using only element numbers instead of position.
The analysis software used at the Forschungszentrum Jülich uses this variation, because pixel-clusters were not expected at the design phase of the program. This is why neighboring hits are merged instead of neighboring pixels.
This feature was adapted for the new analysis package. For the hit-clusters, neighboring hits (element numbers) are searched, also taking into account the element number jump for the circular detectors at element number 0 to $n-1$.
The hit-clusters shape and TDC is taken from the hit with the QDC-weighted position of the participating hits and for the new QDC the individual ones are added.

3.10.4 Tracking algorithms

As already mentioned there exist more than one way to identify tracks depending on collaborator. The algorithm used at the Forschungszentrum Jülich was used as a starting point, but the others were implemented as well. For the prompt tracking the different algorithms produced output, that was very small in difference, so the version with the highest efficiency was used.
On the other hand the Vee-tracking, searching for neutral-decays, showed some differences.The two remaining versions: the Suspect-Vee-search and the Pixel-search have been compared. The Pixel-search, being a brute force method was considerably slower than the Suspect-Vee-search, but the efficiency was higher. The resolution was comparable.

Prompt Tracking

The tracking routine used in this work is derived from the routine used in the Jülich Analysis Program. There are several assumptions made:

- The tracks originate in the target, so one point of the straight line is fixed already: the target (0, 0, 0). This assumptions is valid for all prompt tracks due to the definition of prompt tracks.

- The second point to pin down the straight line is the center of a stop-pixel (in the Jülich Analysis Program this is fixed to Quirl, Ring, Barrel and a special angular range of large Hodoscope pixels; in this analysis, Quirl-, Ring-, Barrel- and Micro-Strip-pixels are used).

In this version, nothing so far distinguishes the pixels, so one has to supply the IDs of the pixels, which are to be stop-pixels.
The procedure is: There are some main differences between the original version

Algorithm 3.5 Prompt tracking algorithm used in Jülich

for all stop pixels j of type i **do**
  $line \leftarrow \left(\overrightarrow{pixel_{i,j}^{center}}\right) - (origin)$
  for all detectors d and elements $e \in d$ **do**
    distance $\leftarrow$ $line$ to $volume_{e,d}$
    if $volume_{e,d}$ is hit **then**
      take $element_{e,d}$
    else if $distance < max - distance$ **and** distance is smallest for all e in d **then**
      take $element_{e,d}$
    end if
  end for
  if $N_{hits} < N_{i,min}$ **then**
    reject track
  end if
  if $\chi^2_{line-fit} < \chi^2_{i,max}$ **then**
    reject track
  end if
end for
sort tracks by χ^2
eliminate too close tracks ($\alpha < \alpha_{max}$) or tracks with too many shared elements
sort tracks by θ

and the derived version that is used in this work: Most parameters are not built in, but provided by the algorithm_parameter passed during initialization: IDs of pixel-types, n_{common}, $dist_{max}$, $n_{j,min}$, $\chi_{j,max}$ and α_{max} as well as an array of detector-ids to search hits for.

Suspect search

The tracking algorithm described above has one major flaw: its speed. The most time consuming operation is the hit-point-calculation of the straight line with the volumes. Many accelerations for this operation have been done, especially in the geometry package, also including distance estimations for large distances. But the biggest problem remained: the number of hit-point calculation.
The new idea was to do the hit-point calculation not for every element for each track, but to do it once, asking the detector overall shape which element most probably would be hit by the line. Allowing for a specific deviation in element number (supplied for each detector and each biasing pixel) the hit-point calculation is now done but only once.
Next step was to define essential detectors, like Start detectors, that have to be on a track, for the track to make sense. The search for these essential detectors was

done first. If one was not found, the track is rejected.
For some detectors it may not be possible to define a suspect-function. Some may have every single element read in from file, some shapes may have the suspect-method not correctly implemented for a certain stack-type of elements. Here it is possible to have the algorithm do a conventional search for this special sub-detector.
With this modifications the speed was increased by a factor of ten, without noticeable difference in output.

Decay Tracking / Vertex Tracking

There are two kind of unstable particles, charged and neutral ones. For the neutral particles, the interesting decay channel is the one into two charged particles. This enables one to make some assumptions: There are two charged tracks, that intersect somewhere outside the target and that are in plane with the target and their point of closest approach. And the point of closest approach must have a minimum distance from the target (d_{min}) to be distinguishable from prompt tracks. The following routine takes advantage of the coplanarity of the decay plane with the target. This routine is also derived from the Jülich Analysis Program.
The prompt tracks have to be calculated beforehand, because this routine rejects decays that have too many hits in common with prompt tracks. The tracks are sorted in ϑ as the prompt tracks. Here also the parameters are taken from the algorithm_parameter instead of using hard-coded values.

suspect V-search

As described in 3.10.4 also the Vee-search adapted from the analysis software used at the Forschungszentrum in Jülich is creepingly slow. So the same modifications as for the prompt suspect search were made here.

pixel-search

Adapted from the analysis software in Dresden, this tracking routine is a brute force method. Here any two pixels are combined and form the bias of a track. Additional pixels close to that line are searched. Then an element search as for the suspect-search is done to find other elements on the track. A single line Fit is applied. Tracks too close to prompt tracks, with too less hits or too high χ^2 of the fit are rejected.
Then any two tracks are combined to find a vee. There are limits on the distance of the tracks to each other (α_{min}), χ^2-values of the combined fit and the coplanarity with the target to get a neutral decay into two particles.

Algorithm 3.6 Neutral-decay tracking algorithm used in Jülich

for any combination of stop-pixels i and j with $i \neq j$ **do**
  define $plane$ as $triangle(P_{target-center}, P_i, P_j)$
  for all intermediate pixels k **do**
    if distance $plane - P_k < d_{max}$ **then**
      take pixel k
    end if
  end for
  for all stored intermediate pixels l **do**
    for both i and j **do**
      search elements on $line_{i/j,l} = (P_{i/j}, P_l)$
      if $N^{hits}_{line_{i/j,l}} > N^{hits}_{min}$ **then**
        keep $line_{i/j,l}$
      end if
    end for
  end for
  combine any two $line_{i,x}$ and $line_{j,y}$ to a $\mathcal{V}$
  if $N^{hits}_{\mathcal{V}} > N_{min}$ **and** $\chi^2_{veefit} < \chi^2_{max}$ **and** $distance(vertex - target) > d_{min}$ **then**
    take $\mathcal{V}_{smallest\chi^2}$
  else
    no $\mathcal{V}$ for this combination i, j
  end if
end for

3.11 Reaction recognition

Reaction recognition is perhaps the most interesting part of the analysis, at least from the physicists point of view. Many reactions have differently tight restrictions on the properties of the tracks, defined by the kinematical restrictions of the reaction. Most stringent case is elastic scattering of two identical particles. Here the kinematical restriction on the tracks, the angles and the velocities is hardest.
As mentioned in the documentation (see sec. D.3.12) the reactions found are stored in the TEvent-structure. The reaction recognition does no real testing for these footprints itself, but passes this point down to the reaction classes themselves.

3.12 Graphical User Interface

The Graphical User Interface (GUI) is designed to help the physicist manage the incoming data, the detector setup and the algorithms that shall be used on the data. The user selects the order in which the algorithms are applied to the data and can watch histograms during analysis. After analysis he can display the results, fit histograms, apply cuts.

This is programmed on the basis of Qt [47], to use the SIGNAL-and-SLOT mechanism Qt provides and above all, the designer with its Graphical User Interface to produce the widgets (windows).
With **typeCase** version 2 also the graphical user interface of the software became very well structured.
An important feature of version 2 is the compatability both with Qt3 and Qt4, that makes at some points the code hard to read but also provides hereby portability and backward compatability.
There are:

main window that manages the file-IO for the parameters as well as the other windows/widgets. All special features that are native to the graphical user interface are launched here

help widget a simple window based on an HTML-viewer, that allows to browse through the help files

parameter widgets to modify the parameters of the analysis, see sec. 3.12.1

analysis widget here all parameters for the analysis come together to perform and view the analysis. Selected algorithms as well as selected runs have to be passed (and are passed via the main window) to this module. On a (dividable root-) canvas the event pattern/histograms/trees can be drawn

view widgets these widgets perform actions like simulations, kinfit or efficiency and acceptance corrections. Usually these parts of analysis are done in separate programs, but for **typeCase** these very different functionalities have been included.

utility widgets and dialogs

3.12.1 Parameter-widgets

The parameter-widgets (see parameter-package sec. 3.7) allow a parameter to be displayed and/or modified (figs. 3.7, 3.8). The list widgets display a list of parameters. This exists for the following parameter-types:

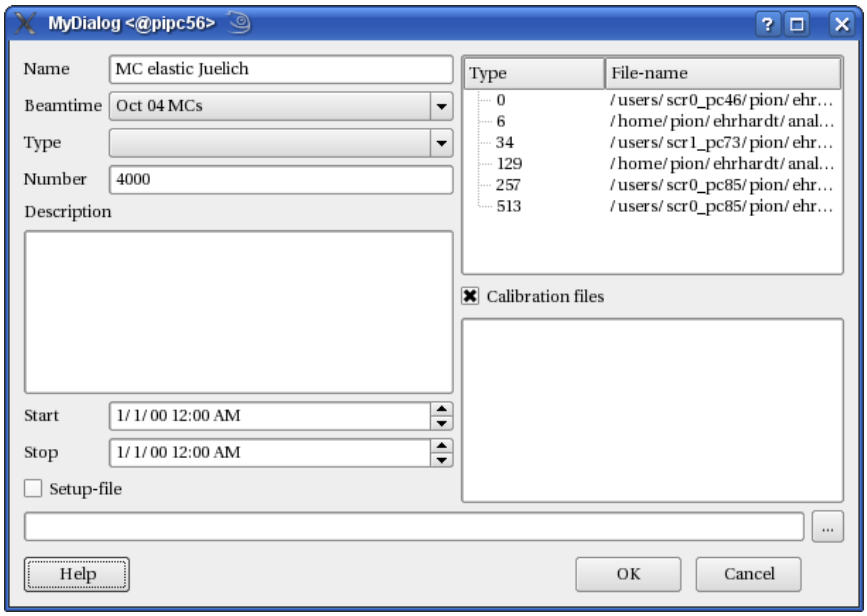
- *run_parameter*, *beamTime_parameter*
- *element_parameter*, *material_parameter*
- *shape_parameter*
- *detector_parameter*, *reaction_parameter*
- *algorithm_parameter*
- *paint_parameter*

As list widgets there are two defined for algorithms, one to simply display the algorithms and one to sort them for analysis. Run and beam-time widgets are organized into a data-basis-widget and detector-, material- and shape-lists are combined into a setup-widget. The complete set of parameter-widgets including dependencies is shown in fig. 3.9.

Figure 3.7: Widget for editing and displaying a detector and a run (b). For more pictures see sec. D.7

3.12.2 Utilities

Utility widgets are small, useful widgets, that are also used in other programs. They are packed into a separate library and can be linked to any other program. There is a widget, that manages three floating-point values, useful for points and vectors in three dimensions. There are dialogs for single values, as integers, floating-point-values, a single string, 3D-vectors/-points, and a whole vector set. There is also a dialog that asks for a single widget, the user can interact with, e.g. a parameter-widget.
Unfortunately the developers at Trolltech ([47]) removed the class QWizard at the transition from Qt3 to Qt4 with a remark of the type “if you want to have it do it yourself”. The class *QMyWizard* as result is also in this library. Last but not least there is a stack-widget combined with a combo-box, where the user can define which widget is shown by the selection of the combo-box.
To edit the properties that come from various root-objects, widgets for Input/Output for fill-, line-, marker-, text-, axis-, pad- and histogram-attributes as well as a colored button including a root-color selection dialog have been created (*AttributeWidget*, *AAttributeWidget*, *ColorButton*, *ColorSelectDialog*, *HistoWidget*). Since the root-class *TQtWidget* provides no real signals, *QCanvasWidget* has been declared for easier use. For division of canvases *DivideDialog* can be used to ask



(a)

Figure 3.8: Widget for editing and displaying a run . For more pictures see sec. D.7

for the number of divisions in x- and y-direction.

Qt-utilities

Some of the most used commands that have been changed from Qt3 to Qt4 are redefined in a small utility unit, to reduce the number of preprocessor commands in the actual code.

3.12.3 Install-wizards

For easy installation of additional algorithms and shapes to **typeCase**, meta-code has been generated that inserts the necessary commands into make-files, header-files, the analyzer-code files and, if necessary, to **typeCase**-code-files (see sec. D.5). The parameters – filenames mostly – can be handled with two installation wizards (fig. 3.10), that also do some kind of rough consistency check.
Having specified the installation parameters once, they can be saved to file and reloaded in a later session or can be passed along together with the corresponding header- and code-files for easy exchange of algorithms between developers and users.

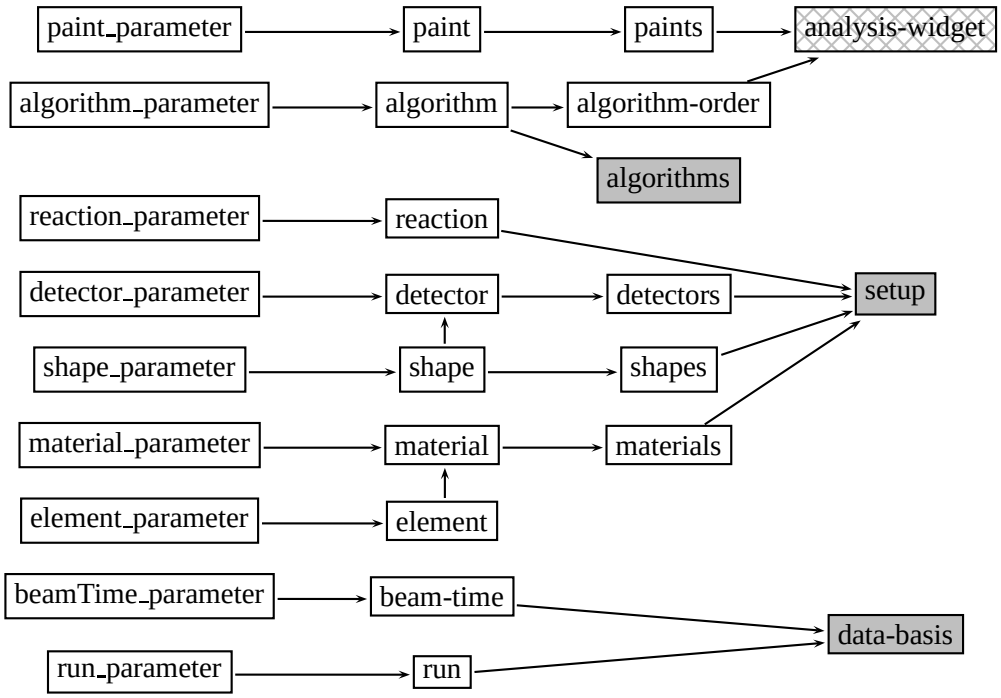


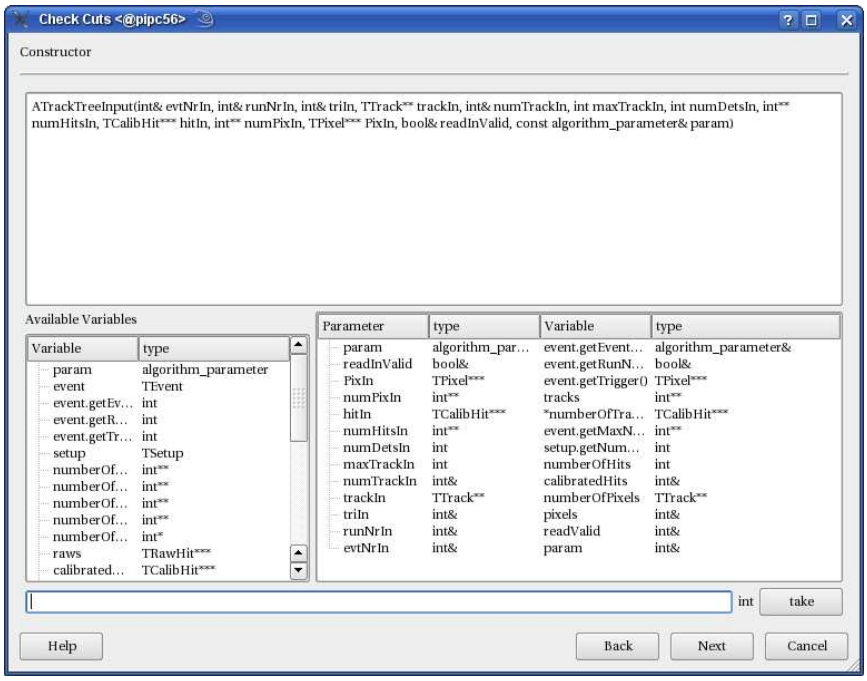
Figure 3.9: Dependency of the widget classes. First column shows the non-GUI-parameter-classes as used in the parameter package. Second column shows the single-parameter-widgets, third the list-widgets and fourth the combined widgets. The widgets filled in gray are directly accessible via the main window; the hatched analyser-widget is not included in this library.

3.12.4 IO-algorithm-widgets

A subset of algorithms is used for reading and writing data to and from file. For easier handling these algorithms are not per se listed under algorithms, but come with an own widget, that returns the parameter describing the algorithm. They can be selected this way along with reaction recognition and the order selection in the analyzer widget.

3.12.5 Analyzer-widget

The analyzer-widget needs the assignment of selected runs and algorithms, that can be passed – and are in the case of **typeCase** – using the Signal-and-Slot-mechanism from the data-basis-widget or the algorithm-widget.
The main part of the analyzer-widget is covered by a canvas, that can show the event-pattern, information about a current event, a histogram or a tree, that are filled during analysis (fig. 3.11).



(a)

Figure 3.10: Screen-shot of some widget of the install-wizards. Go to the picture-gallery (sec. D.7) for more pictures.

The left bar controls the analysis, using *init*, to initialize algorithms, *start* or *step* to run analysis, *stop* to halt the running analysis, finishing the current event, *final* to finalize analysis (necessary e.g. to close files), *help* to receive some help on the widget, *algorithms* to add input-, output- or reaction-recognition-algorithms and *view* to select the display type in the aforementioned canvas.
The bottom bar shows the status of the analyzer (initialized, running, stopped, . . .), the number of events to be analyzed per step and the number of selected runs.

3.13 User and developer guide

see section. 4.

3.14 Note on root

Initially it was planned to derive all classes from the root base class *TObject*. This would have provided writability to root-files and usability in the command-line-interpreter CINT. But this was omitted, when serious problems in the construction of the root-framework became visible.

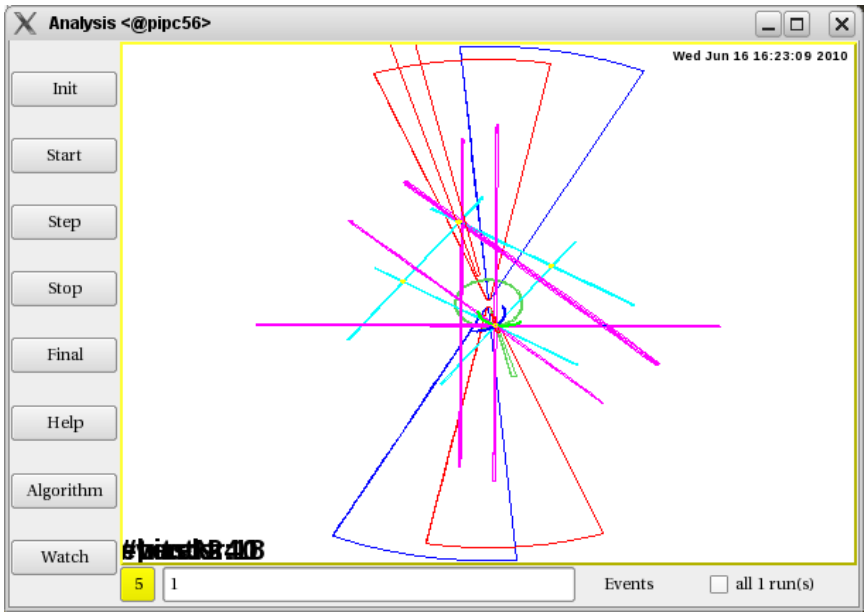


Figure 3.11: The analysis-widget, showing the control buttons on the left and the display of the currently analyzed event (pp-elastic-scattering) as a point-projection of the hit detector elements (detector-ID is color-coded) in a root-TCanvas in the center.

First of all the memory management was reprogrammed overwriting the *new* and *delete* operators. This may have been a reasonable idea in the first years of C++-compilers, but meanwhile doing so is for a long time no more appropriate since the compilers memory management is good.
However the aforementioned operators were redeclared as class members of *TObject* making it impossible to privately or protectedly inherit from this class.
The actual implementation of the root memory management leads to serious problems concerning portability of code (what may work on one machine may fail completely on an other), large programs (memory is sometimes not freed correctly) and polymorphism which was made impossible due to the actual implementation of *TStorage*.
After many unexplainable segmentation faults, the dependence was removed and surprise: it works now reliably. But the problem remains for the root-defined classes. It is strongly recommended that the polymorphism feature is not used with root-classes.

Chapter 4

Guides

4.1 The very short User Guide

4.1.1 Get the program

Download the zip file at `http://www.pit.physik.uni-tuebingen.de/~ehrhardt/KTOF/download/typeCase.tar.bz2`.
Unpack the file using

```
> tar --bzip2 -xf typeCase.tar.bz2
```

Prerequisites

You need one of the following configurations:

	configuration 1	configuration 2
compiler: gcc version	3.x	4.x
root-library version	$<$5.18.x	$>$5.18.x
Qt version	3.x	4.x
Boost	any will do	

Environment variables Next, you need to set a number of parameters necessary for the compilation process. These are mainly path variables, of where the individual variables are located:

Variable	meaning	default
CDIR	where is your compiler	
LD_LIBRARY_PATH	path including all directories for shared libs	
PCCODE	identifier for your computer	
KTOFPACK	**typeCase**-package directory	
BOOST_ROOT	Boost main-directory	
BOOSTINCLUDE	Boost-header directory	$BOOST_ROOT/include
BOOSTLIB	Boost-library directory	$BOOST_ROOT/lib$PCCODE
QTDIR	where is your Qt	
QT_VERSION	Qt-version	
QT_INCLUDE	Qt-header directory	
QT_LIBS	linker command for Qt-libraries	
ROOTSYS	where is your root distribution	
ROOTLIB	root-libraries	$ROOTSYS/lib/root
ROOTINCLUDE	root header directory	$ROOTSYS/include/root

You certainly need the directories for your libraries, like Boost ($BOOST_ROOT), Qt ($QTDIR) and root ($ROOTSYS) and since the variables are native to the libraries, they may already be set. The include and library paths were added since you can specify these paths to be different from the default values during the installation process. The version of Qt ("3"—"4") is necessary for the generation of the meta-objects of some of the widgets.
Use the $PCCODE variable to identify the machine you are doing the compilation on, with this you can make libraries for different configurations in the same base directory: the $PCCODE will be appended to the name of the object- and executable-files. This can be the value stored in $HOST, you can even leave it empty if you want to compile only one version of binaries.
To be able to choose between different compilers the $CDIR variable was added if you want to use the default g++-compiler use

```
> which g++
```

to see where it is located and set the path-variable $CDIR accordingly.
$LD_LIBRARY_PATH (but I recommend having $KTOFPACK too) is the only variable that is important to have at run time. Else you won't be able to execute the program. Like the variable $PATH it contains paths separated by colons and may already be set to some value; append the library paths of Boost, root, Qt and **typeCase**.
When you extracted the tar-file, there is a file *typeCaseVariables.init* including the definition of the needed variables for bash (for shell you will need *setenv* instead of *export*). Open the file and edit the variables; save them for later use. Type (for bash)

```
> source typeCaseVariables.init
```

Compilation

Go to the package directory of *typeCase*, here you type

```
> make FIRST
```

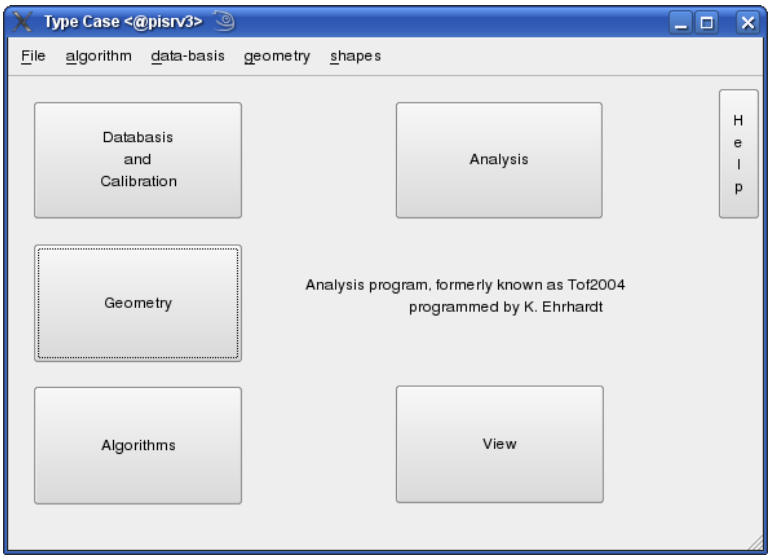
With this command, the packages will be compiled; this may take some time. Then you go to the *gui*-directory and type again

```
> make INSTALL
```

There you are.

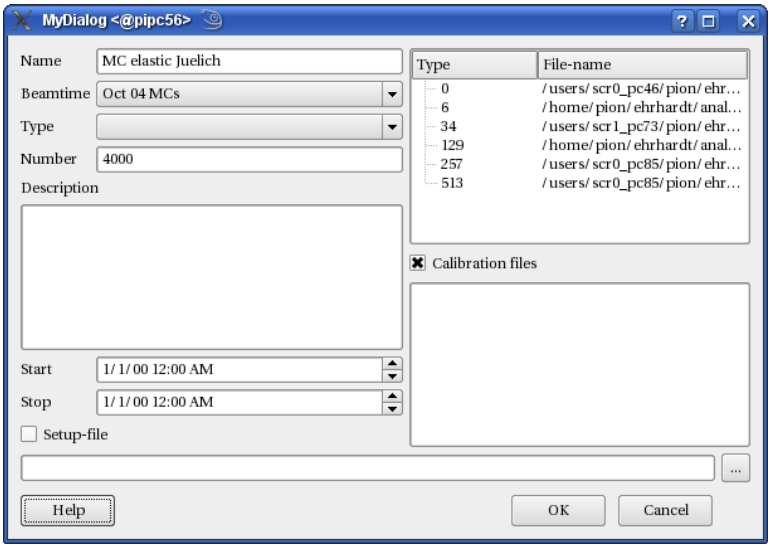
4.1.2 Start typeCase

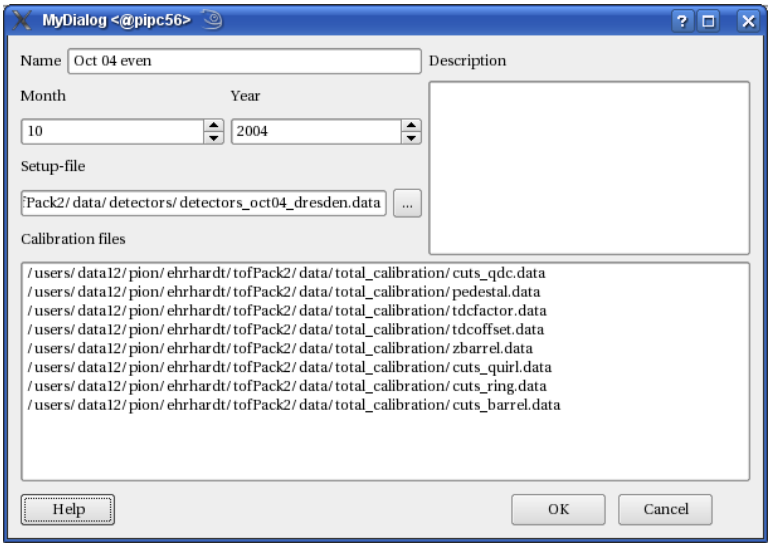
```
> typeCase$PCCODE
```



If you start it for the first time, it will create a new directory in you $HOME-directory: *.typeCase*.
If you need some help on the widgets or the reference-documentation of the classes of **typeCase**, use the **help**-button.
Clicking on any button on the main-window will open an independent window, you can work in different ones in parallel.
The menu is mainly for file-management, here you can load and save the parameters of the analysis. The “save”, “save as” and “load” commands refer to the complete data-base of that type, so with “load” you will load a complete different data-base from file, omitting the previously used (so take care to have “save”d it before). If you want to add a single parameter from a file to the existing data-base, use “add”. Also the installation-wizards (sec. 3.12.3 and Appendix D.5) are accessible here.

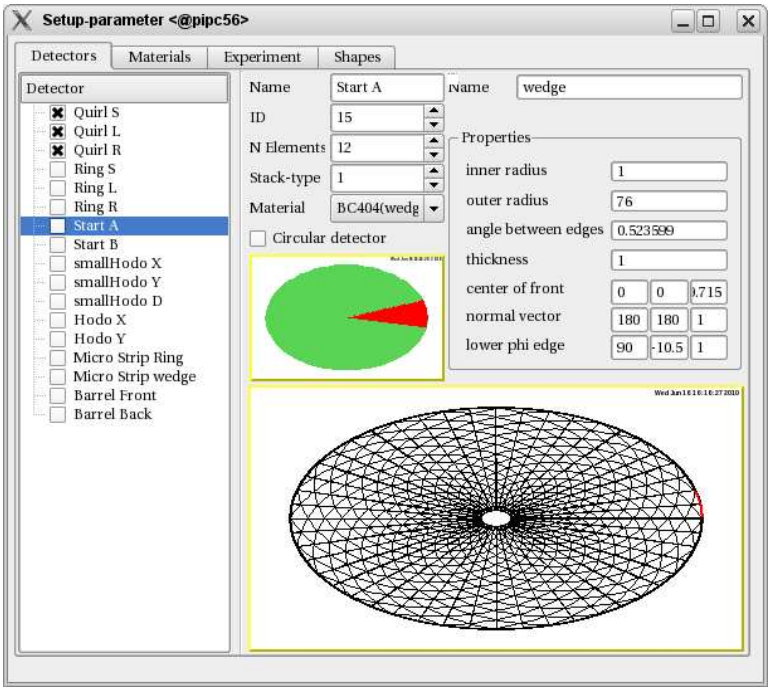
The Data-Base

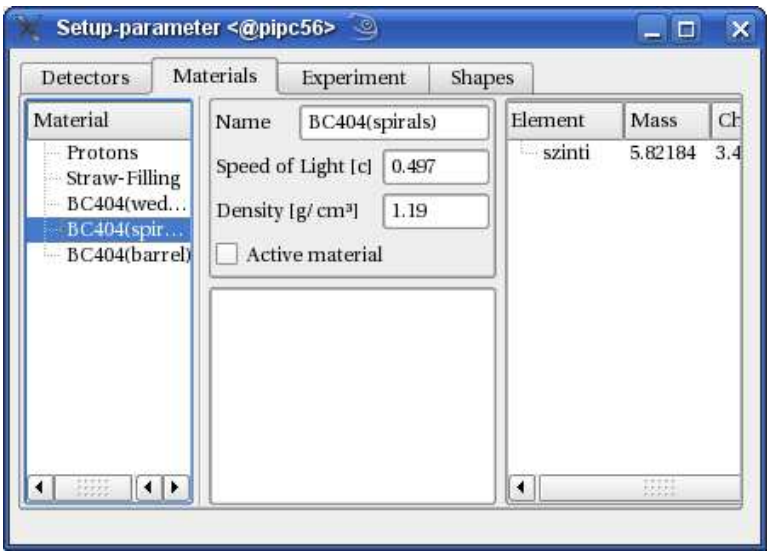




Clicking on the button labeled “Databasis and Calibration”, the data-basis-widget opens, showing the defined beam-times and runs. The runs are equipped with check-boxes, so if you check one it is marked for analysis.
With a *right-click* you access the context menu, that allows you to inspect (read-AND-write) the selected run/beam-time, to delete the selected run/beam-time or to create a new one.

The setup





Access the setup by clicking on the button labeled “Geometry”. This window has four tabs.

The last one contains the defined shapes. Here you can view the defined shapes, their properties and the way they look like when plotted.

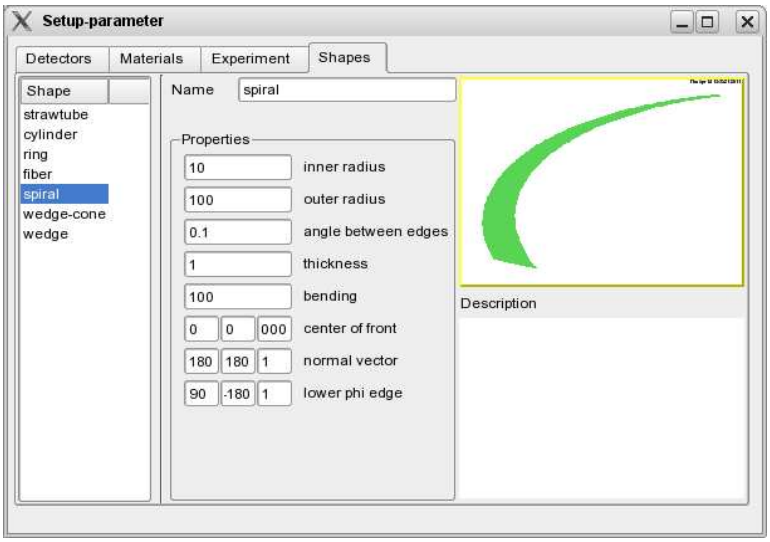
The tab called “Experiment” shows the settings for beam and target.

In “Materials”, the materials of the detector-setup can be viewed and modified.

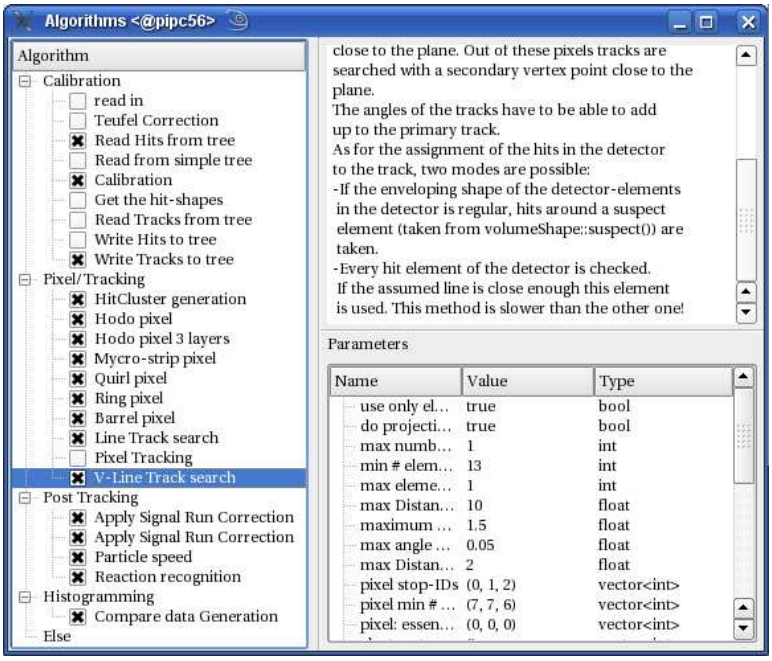
The first tab shows the defined detectors. A checkmark can be added to detectors in the list; selected this way, the checked detectors will be drawn in the lower canvas. In the upper part of the window, the current detector is displayed and can be changed here.

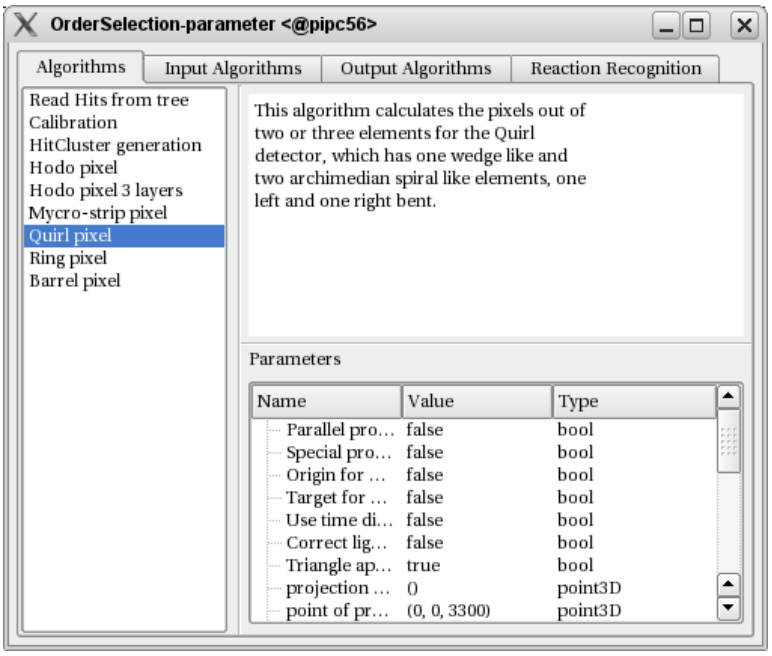
Right-clicks on the lists open pop-up-menus, enabling adding of items and removing items.

Make sure, that you saved the modified setup before you start the analysis, because the setup is read from file according to the file specified for the beam-time or run.


Setup-parameter
Detectors
Materials
Experiment
Shapes
Shape
strawtube
cylinder
ring
fiber
spiral
wedge-cone
wedge
Name
spiral
Properties
10
inner radius
100
outer radius
0.1
angle between edges
1
thickness
100
bending
0
0
000
center of front
180
180
1
normal vector
90
-180
1
lower phi edge
Description

The algorithms





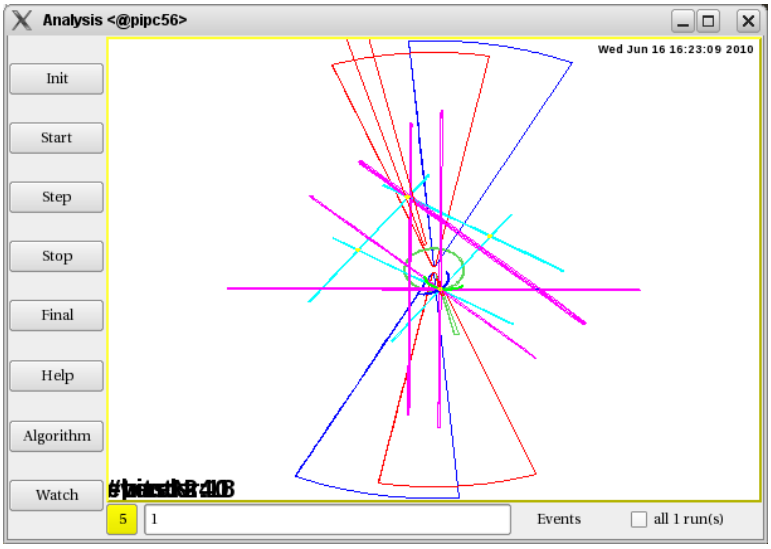
The button labeled “Algorithms” opens the algorithm-window. On the left there is the list of all available algorithms, here – as with the data-basis – a check-mark means that the algorithm is selected for analysis. On the right, the current algorithm is described including the parameters, ready for modification.
At this point the algorithms are not sorted in any way. You cannot really do that here, though enabling them in the right order may help.

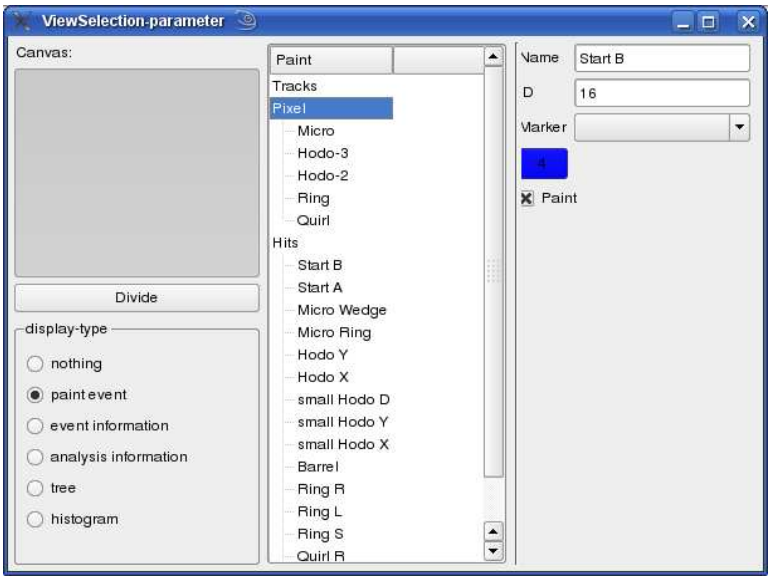
The sorting

Having now edited and saved the detector-setup, chosen the runs to analyze and the algorithms to use, we can now proceed to the actual analysis by clicking on the “Analysis”-button.
On the bottom you can see – on the very right – the number of runs you chose in the data-basis-window. The color-button on the left is still red (uninitialized).
Before you go ahead and initialize the analysis, let’s first bring some order into the selected algorithms. So click on the button “Algorithms”.
In the window that opens you can see the selected algorithms. Select one to see (and modify) its parameters on the right side. To move it click on the description and type “u” for up, “n” for down and “d” for un-select. As you perhaps noticed,

there is no input-algorithm so far. Let's change that by clicking on the "input Algorithms" tab and choose one of the input algorithms from the combo-box on top. Click on insert, when all the parameters fit. Do the same for output-algorithms. Then check again for the sorting of the algorithms. You don't need to close the window to proceed.

The analysis procedure





On the left of the analysis window, you see the control buttons. Press "init" to initialize analysis. You cannot run an un-initialized analysis, as well as you cannot stop a not running analysis. Enter the number of events to do in one step in the line-edit on the bottom of the window.

The button on the left of the bottom line shows color coded the state of the analysis. Red is uninitialized, green is initialized but not running, yellow is running. If you finalize, the button will turn red again.

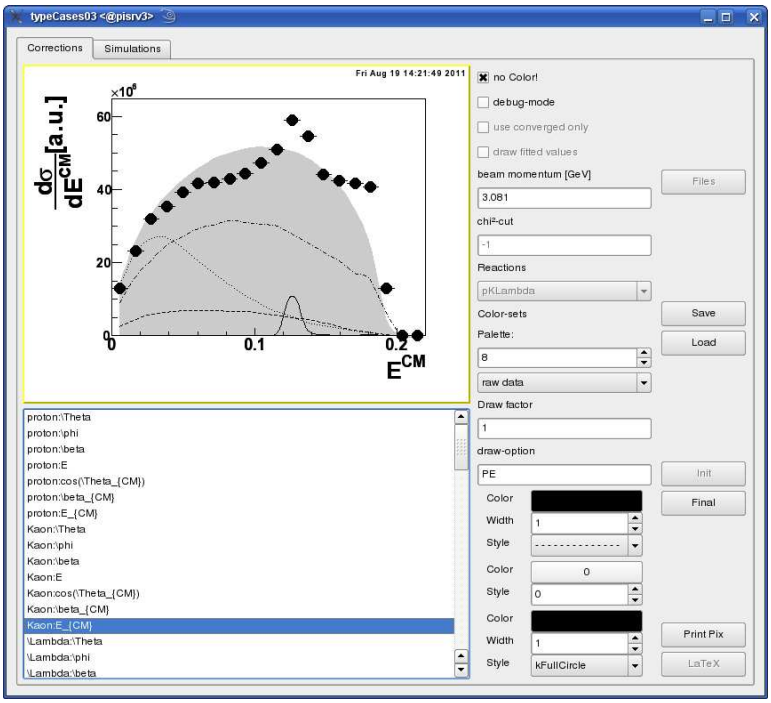
After analysis you should finalize the analysis-engine using the "final"-button. This will cause important last-minute-stuff to happen, like closing files, generating calibration constants (only if calibration-generation was selected as algorithm) or simply freeing memory that was allocated.

Viewing the progress

To see such an event-display as shown in the picture on the left side, you click on the “Watch”-button. An other window opens. Select a pad. Select what to watch. If you want to view a histogram or a tree-branch, you have to select it after initialization. Again no need to close the window, you can change it at any time.

Corrections, Simulations and Kinfit

Generating these nice plots as seen in this work, the plotting engine has been added to the software recently, as well as the simulation and some rudimentary kinfit-engine.
To access these, go to the Main-Window and press the “View”-button.

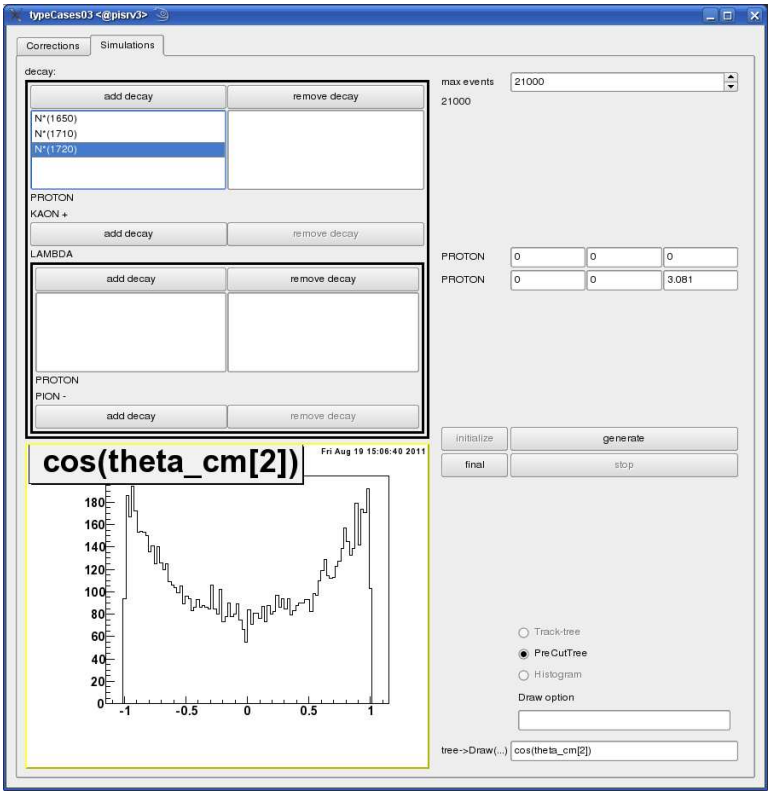


Corrections and plotting Select the “Corrections”-tab at the upper part of the widget.
For the purpose of performing corrections and displaying the data nicely, you define different data-sets: raw-data for uncorrected data, corrected-data, different simulations like phase-space or model simulations, simulations that passed a virtual detector and efficiency (simulation-through-detector over pure simulation). Each data-set will need at least one input-file (specify histogram-files, files with track-tree-format, PreCutTree-format or ascii-dat-format) and how it should be displayed (line-, marker- and fill- -color, -style and -size, draw-option).
The central part considers the plotting of the data. The lower part of this column is dedicated to the color-schemes of the data-sets, as well as the draw-option and the relative size of the set (data is always 1). On the top you can select between the two paint-modes (color and monochrome, though you can also use colors for the monochrome-draw) and switch on the debug mode, where all data-sets are drawn with factor 1, not only the specified ones. You can specify cuts and a beam-momentum, essential if you are reading from tree- or dat-format. In the center of

this column you can select between the different reaction types. Each reaction type defines different graphs, so take care that – especially for histogram-input – the input files match the selected reaction.
The left part of the widget is covered by a canvas to draw the graphs and a list to browse through the list of available histograms. On the right edge, you have the control buttons:

- **Files**: Here you open a window that displays the files you can read in for each data-set
- **Save**: Save the file-names and color-options to file.
- **Load**: Load file-names and color-options from file.
- **Init**: Define the data-structures and read input-files. For histogram-files this is rather fast but it can take a while for the tree- and dat-formats, since the histograms are filled event-wise, also applying cuts.
- **Final**: Clear the data-structures again to read other data from other files or perhaps change to a different reaction type.
- **Picture**: Draw all histograms with the specified colors and plot modes to a directory of your choice.
- **LaTeX**: Produce a LaTeX-file, that includes all plots, previously plotted into a specified directory. The file is not fed through the LaTeX-engine, you will have to do that yourself going into the specified directory and using *latex* and *dvips* or *dvipdf*.

Simulation Select the “Simulations”-tab at the top of the widget.



The right part of this widget is filled with controls: set the number of events to simulate, the initial system and show the currently simulated number of events. The button “initialize” initializes the simulation engine asking for the type of output you want to have (you can also have PreCutTree-format if you select a reaction type). “Finalize” will save the simulated events to file. “generate” starts the simulation. Since the simulation will run in a different thread you can stop(“Stop”-button) the simulation before it reaches the maximum number of events.
On the left side of the widget, you see the simulated particles. Top-level is the decay of the initial system. It frames the particles it decays in. On the top part of the frame, there are two buttons enabling you to add or remove particles (opening a menu). Below that, there are the modifications to the phase-space-weight, every event receives: You may add resonances (Breit-Wigner-like) or Final-State-Interactions. This is followed by the particles. If one of these particles is non-stable, you may add the decay of this particle into some other particles. Note: There is no consistency-check done whether the masses of the decay-particles is smaller than that of the decaying particle or whether charge or lepton-number for example are conserved.
On the bottom there is a canvas and an input line. If you specify a valid tree-branch to draw, it will be drawn to canvas.

4.1.3 Command-line-typeCase

The command-line-**typeCase** (or cl-typeCase) is quite small and easily compiled: go to the *cl-typeCase*-directory and type

```
> make
```

For a faster performance use the command

```
> make STATIC
```

Since the packages, the program depends on, were already compiled, the procedure is finished rapidly.
The program takes some command-line arguments, type

```
> cl-typeCase --help
```

to see which.
The parameters “a”, “d” and “m” define the parameter-files for the used algorithms, the data-basis and the materials. Select a run with “r”, probably you should enclose the parameter and the name in quotes, especially if the run’s name contains blanks. Alternatively you can specify a file containing the names of the runs (“runs”, one line per run name).
“n” sets the number of events to analyze, it is 1 by default. “l” passes a start entry number to input-algorithms, that support this feature. “j” defines the number of threads the analysis shall run with (defaults to 1), don’t use more than you have processors in your machine.
“o” specifies the path where output-algorithms write their files to.
With the options “local”, “local-directory” and “nonLocal” you can ask the input- and output-algorithms to copy input files to a local directory, do the analysis locally and copy the output files back. Most input-algorithms clean-up their copies afterwards, but it pays to check.
Use the “paint”-option to generate a post-script file with the projected event-pattern of the analyzed events, but use this option only for a small number of events, the file will grow very large else. The options “H” and “I” provide you with additional information about the analysis.
All parameters can be specified using the command-line, but also using the file “analysisInit.init” in the current directory. An example may be

```
> cl-typeCase --H --I "--r=elastic run" --a=algorithms.data
--o=out-directory/first- --d=beamTimes.data --m=materials.data
--n=1000000 --nonLocal
```

Progress file The analysis-program will write a file to the directory “/dev/shm/$USER´ if it exists. In this file it will save the progress of the analysis. This directory was chosen, since it doesn’t exist on disk – all files will be deleted on a reboot –, it is much faster than writing to disk. And since the file is just a few hundred bytes in size no matter how much you analyze it won’t fill up the memory.

Parameter files

There are three parameter files necessary to run an analysis.

1. A data-basis-file; here the defined beam-times are saved. For each beam-time, there exists a separate file with the runs for this beam-time.

2. A material-file containing the defined materials.

3. An algorithm-file; here the algorithms that are going to be used on the data are described, in the right order. Only algorithms with the set used-flag will be considered.

The detector-setup will also be read from file, but you cannot specify it separately. It comes with the data-base, where for each beam-time the setup-file-name is saved. All these parameter-files are ASCII-files – human-readable. Most values are named and therefore easy to identify.
Fortunately you won't have to write them all from scratch by yourself. **typeCase** did that already for you. Especially for the data-basis and the materials it is best and much easier to use the Graphical User Interface to define the necessary parameters for the runs and materials. Later-on, as experienced user, you can open the parameter-files with an editor (like vi or emacs) and modify the content, for the runs, it is rather easy. Use the complete data-base as input and specify a run name for the analysis.
For the algorithms, use the "save analysis algorithms" menu, that is accessible as pop-up-menu in the algorithm-order-widget from the analysis-widget. Here you can save the current algorithm-set to file. Starting with this file you can easily modify the parameters if necessary using the editor of your own choice.

4.2 The extremely short developers' guide

You are a C++-programmer and you want to extend the **typeCase** analysis-framework. You are welcome to do so.
You are a FORTRAN-programmer and you want to include an algorithm to the **typeCase** analysis-framework. Please proceed to sec 4.2.2.
You are a C-programmer and you want to extent the **typeCase** analysis-framework. You are welcome to do so, though you will have to learn something about object oriented programming first.
First you have to decide what you want to do

- Modify an existing shape
- Modify an existing algorithm
- New detector-shape
- Interface to black-box/library
- New algorithm
- Create Install-log
- Apply Install-log
- GUI-Maintenance

Class 4.1 Class definition of class volumeShape.

```
class volumeShape:public base_shape
{
 protected:
  vector3D res;
 private:
  float maxDist;
 public:
    volumeShape(string n="undefined");
    volumeShape(const volumeShape &s);
    virtual ~volumeShape();
    virtual volumeShape* getClone();
    float getMaxDistance() const;
    void setMaxDistance(float value);
    virtual float getFlightPathInShape(const sLine3D &line);
    virtual Vector HitParams(const sLine3D &line);
    virtual Vector Hitting(const sLine3D &line);
    virtual Vector HitParams(const planeShape &shape, point3D origin);
    virtual volumeShape *getNext(int times, int stackType);
    virtual volumeShape *getEnvelope(int times,int stackType);
    virtual int suspect(const sLine3D &line, int stackType);
};
```

4.2.1 Shapes

The shapes come in two different types: the volumes and the planar shapes. The volumes are mainly used to express the shape of a detector or detector-element, while the planar shapes are mainly used for the projected pixel-shapes. This reflects the interface of the two types (see class definitions 4.1 and 4.2).
The volumes need the copy constructor as well as the clone function. The function *getFlightPathInShape()* returns the distance between entrance- and exit-point (needed for $\Delta E/\delta x$-calculations). The functions *HitParams()*, *Hitting()* and *suspect()* are needed for tracking purposes and return information about the hit of a line with the volume (for more information see D.2 and the reference documentation in [23]). Though not shown here, I recommend to define and implement a constructor taking a shape_parameter to define the properties of the volume. If you want your shape to be drawn to a root-canvas, you will have to overwrite the function(s) *Draw()*.
The planar shapes define a plane in which they exist. The footing point of the plane is always the center-point defined in the base class base_shape; the normal vector is defined in planeShape. The planar shapes define a number of corner points and an area-calculation-function. They return the shape_parameter containing their prop-

Class 4.2 Class definition of class planeShape.

```
class planeShape: public base_shape
{
 protected:
    vector3D normal;
    float circRadius;
 public:
  planeShape(const string &n);
  planeShape(const shape_parameter &description);
  virtual ~planeShape();
  vector3D getNormal()const;
  void setNormal(const vector3D & v);
  plane3D getPlane()const;
  void setPlane(const plane3D &p);
  virtual int getNumberOfPoints()const;
  virtual point3D getPoint(int num)const;
  virtual float area()const;
  virtual float angularRange_Phi(const point3D &origin=point3D(0,0,0),
     const vector3D &zDirection=vector3D(0,0,1))const;
  virtual float angularRange_Theta(const point3D &origin=point3D(0,0,0),
     const vector3D &zDirection=vector3D(0,0,1))const;
  virtual vector3D distancePlane(const plane3D &p);
  virtual vector3D distance(const sLine3D &line);
  virtual float circumscribedRadius()const;
  virtual shape_parameter description()const;
  static shape_parameter getDescription();
};
```

erties. Also a static function is defined that returns the properties of the shape into a parameter. The distance calculating functions *distancePlane(), distance(), angularRange_Theta()* and *angularRange_Phi()* are of importance for the tracking process.

Modify an existing shape

If you want to modify an existing shape, simply go to "$KTOFPACK/shapes/include", here you find the header files of all existing shapes. The code files are in "$KTOFPACK/shapes/src" and are named as the shapes but in lower case letters. Here you can modify the shapes functions, but take care, that you keep in mind the definition of the shape in general!

New detector-shape

The creation of a new shape needs the modification of quite some files, including a make-file, an additional header-file and the “getShape”-file. If you don’t want to do it by hand, use the install-wizard by **typeCase**. This does not implement the actual shape for you, but helps you to insert it into the framework.

Class 4.3 A new plane-Shape class example.

```
class myNewShape:public planeShape
{
  protected:
//put here your new variables
  public:
  myNewShape();
  myNewShape(const myNewShape &shape);
  myNewShape(const shape_parameter &description);
  myNewShape( ... necessary parameters ...);
  virtual ~myNewShape();
  virtual planeShape* getClone();
  virtual shape_parameter description()const;
  static shape_parameter getDescription();
  virtual int getNumberOfPoints()const;
  virtual point3D getPoint(int num)const;

  virtual float angularRange_Phi(const point3D &origin=point3D(0,0,0),
    const vector3D &zDirection=vector3D(0,0,1))const;
  virtual float angularRange_Theta(const point3D &origin=point3D(0,0,0),
    const vector3D &zDirection=vector3D(0,0,1))const;
  virtual vector3D distancePlane(const plane3D &p);
  virtual vector3D distance(const sLine3D &line);

  virtual void Draw(const point3D &eye, const plane3D &plane,
    vector4D* boundingBox, int lColor, int fColor=8,
    int fStyle=1001)const ;
  virtual void Draw(const point3D &eye=point3D(0,0,0),
    const plane3D &plane=plane3D(point3D(0,0,1),vector3D(0,0,1)),
    vector4D* boundingBox=NULL, TObject **ident=NULL, int lColor=1,
    int fColor=8, int fStyle=1001)const;

//put here the necessary getter and setter methods
};
```

planeShape Define your new planeShape as in class definition 4.3. Define the properties, like some corner-points or vectors as “protected” or “private”, define constructors and destructors in the shown pattern (default-constructor, copy-constructor(s) and the normal constructor), define parameter-getters.
Most importantly: overwrite the distance functions, that they fit your shape!
Overwrite the *Draw()* functions to provide painting functionality to root-canvas.

volumeShape Define your new volume shape as shown in example 4.4. Define the constructors and destructor (default- , copy-constructors, normal-constructor) and the description parameter-getters.
The *getClone()*-, *getNext()*- and *getEnvelope()*-functions are very important. In the *getNext()*-function, you define, how out of one shape a neighboring one will be generated.
Overwrite the hit-point-calculating functions *Hitting()*, *entrance()*, *distance()*, *Normal()*, *HitParams()*, *suspect()* and *getFlightPathInShape()* for the use in tracking.
If you want the shape to be drawn, overwrite the *Draw()*-functions.

Important For the implementation, use a new file in the $KTOFPACK/shapes/src/-directory. Use the reference documentation [23] to help you with the implementation in the source file.
If in doubt, how you should implement something, open one of the other shape-source-files – best something with a rather similar shape –, copy the code to your file and use it as a starting point.
If the new shape you have in mind resembles an already defined shape, think about inheriting from that shape. Perhaps you can extend the already existing shape to fit your needs.

4.2.2 Algorithms

In class definition 4.5 there is the class-definition of the class AAlgorithm, the base-class of all algorithm-modules. It is derived of the class QObject [47] to enable the SIGNAL-and-SLOT-mechanism. Apart from default- and copy-constructor and destructor it has a name-getter. It defines methods to retrieve histogram- or tree-pointers, if defined, for external viewing (see sec. 4.1.2). The static *getDescription()*-method returns an algorithm_parameter, that defined all necessary variables and the description of the algorithm.
The most important function is the *process()*-method.

Modify an existing algorithm

There exist the following major modification-types

Class 4.4 Definition of a new volume.

```
class myNewShape: public volumeShape
{
 protected:
//put here your new variables
 public:
  myNewShape();
  myNewShape(const volumeShape &s);
  myNewShape(const shape_parameter &description);
  myNewShape( ... necessary parameters ... );
  virtual ~myNewShape();
  virtual shape_parameter description()const;
  static shape_parameter getDescription();

  virtual volumeShape* getClone();
  virtual volumeShape *getNext(int times,int stackType);
  virtual volumeShape *getEnvelope(int times,int stackType);

  virtual Vector Hitting(const sLine3D &line);
  virtual point3D entrance(const sLine3D &line);
  virtual vector3D distance(const sLine3D &line);
  virtual sLine3D Normal(const sLine3D &line);
  virtual bool cut(const fiber &f);
  virtual Vector HitParams(const sLine3D &line);
  virtual Vector HitParams(const planeShape &shape, point3D origin);
  virtual float getFlightPathInShape(const sLine3D &line);
  virtual int suspect(const sLine3D &line, int stackType);

  virtual void Draw(const point3D &eye, const plane3D &plane,
    vector4D* boundingBox, int lColor, int fColor=8,
    int fStyle=1001)const;
  virtual void Draw(const point3D &eye=point3D(0,0,0),
    const plane3D &plane=plane3D(point3D(0,0,1),vector3D(0,0,1)),
    vector4D* boundingBox=NULL, TObject **ident=NULL, int lColor=1,
    int fColor=8, int fStyle=1001)const;

//put here the necessary getter and setter methods
};
```

Modify a parameter: this can be done online, no need to change the code or to recompile. Use the Graphical User Interface to modify the parameter or edit the algorithm-parameter-file to give the parameter the desired value.

Class 4.5 The definition of the class AAlgorithm

```
class AAlgorithm: public QObject
{
 private:
  string name;//!
 public:
  AAlgorithm(string n);
  AAlgorithm(const AAlgorithm &a);
  ~AAlgorithm();
  string getName()const;

  virtual void *process(void*ptr);

  virtual vector<string>histogramNames();
  virtual TH1* histogram(string histoName);

  virtual vector<string>treeNames();
  virtual TTree* tree(string treename);

  static algorithm_parameter getDescription();
};
```

Modify the defaults for a parameter: Go to the algorithm-source file and there to the implementation of the constructor. Here you can add or change the defaults of a parameter. You will have to recompile.

Add a parameter: Go to the algorithm-header file. If there has already been defined a variable to hold the new parameter, proceed, else define a new variable.
Go to the algorithm-source file. Assign the value of the new parameter to the variable.
Recompile the algorithm package AND the analysis package (needed because header file changed).
Don't forget to edit the algorithm-parameter-files to add the new parameter.

Change the behavior of an algorithm: Go to the algorithm-source file and change the implementation of the *process()*-method.
Recompile the algorithm-package.

Add a function: Go to the algorithm-header file and add the function to the class definition.
Go to the algorithm-source file and implement it.
Recompile the algorithm package AND the analysis package (needed be-

cause header file changed).

Change the amount of data the algorithm receives: Go to the algorithm-header file and modify the constructor-definition according to the modified needs: e.g. add , *TPixel* pixels, int &numberOfPixels* probably you'll have to add or remove variables.
Go to the algorithm-source file, apply the modifications also here to the constructors implementation and change the code to fit the new specification.
Go to the algorithm-init file of the analyzer. Change the init-call of the algorithm according to the modifications you made for the constructor.
Recompile the algorithm package AND the analysis package.

Interface to black-box/library

Let's say, you have an algorithm defined by somebody else. Unfortunately this algorithm was not defined in the **typeCase** framework, but as FORTRAN-, C- or C++(other than **typeCase**)-algorithm. You have some library or black-box-function, but the interface has a valid specification. So you use the following steps:

1. define a new algorithm as in class definition 4.6
2. define the constructor to get the necessary variables, like hits for calibration, hits and pixels for pixel calculation, hits, pixels and tracks for tracking, tracks for after-tracksearch-calibration, etc.
3. use the constructor to initialize the data structures (probably pointers or arrays), copy the parameter values to class-variables and to initialize your black-box, if necessary.
4. the *process()*-method is the place to put the actual conversion between the **typeCase**-data-structures and the input into your black-box. Run your black-box and copy again the outcome to the **typeCase**-data-structures.
5. use the install-wizards to insert it into **typeCase**.

New algorithm

If you want to implement a new algorithm,

- you better first take some pencil and paper and put down some notes about it:
- What shall it do.
- What data does it need.
- Will it need information from time to time (e.g. like switching to a new run)
- Define the algorithm on paper in some kind of pseudo-code.

Class 4.6 Definition of a new algorithm

```
class AMyNewAlgorithm:public AAlgorithm
{
Q_OBJECT
  private:
//class variables
 public:
 AMyNewAlgorithm(... needed parameters ...,
    const algorithm_parameter& param);
 virtual ~AMyNewAlgorithm();

 virtual void *process(void*ptr);

 //virtual vector<string>histogramNames();
 //virtual TH1* histogram(string histoName);

 //virtual vector<string>treeNames();
 //virtual TTree* tree(string treename);

 static algorithm_parameter getDescription();

 signals:

 public slots:
   virtual void onNewRun(run_parameter &run);
};
```

When you are ready, define your algorithm as shown in class definition 4.6. The tree- and histogram-functions are not essential, so you only need to overwrite them if you define histograms or trees that you want to be seen outside. The analyzer (sec. D.1) defines signals, that can be caught by the algorithm using the SIGNAL-and-SLOT-mechanism. If you define signals or slots you will have to add the "Q_OBJECT" macro on top, don't do it, if you don't need it.
When implementing, I recommend you to start with the *getDescription()*-method. Here you can insert, what you put on paper before. Describe what the algorithm will do and what it will need for it. Define which parameters it will need.
Then go to the constructor and implement the initialization, like copying the parameter-values to member-variables and initializing references.
Immediately after the constructor you should implement the destructor, freeing the allocated memory. Do it now, you'll perhaps forget it later.
Implement the algorithm in the *process()*-method. For more complicated algorithms it pays to modularize the process by defining member-functions to do recur-

ring or coherent parts in a separate block.
Use the install-wizard to insert the algorithm into **typeCase**.

Class 4.7 Definition of the QAlgoritmDefineWidget, the base-class of the IO-algorithm-widgets.

```
class QAlgorithmDefineWidget: public QWidget
{
Q_OBJECT
 protected:
   QPushButton *insertButton;
   algorithm_parameter ap;
   int ID;
   bool isInput;
 public:
#if QT_VERSION < 0x040000
   QAlgorithmDefineWidget( QWidget* parent = 0,
      const char* name = 0, WFlags fl = 0 );
#else
   QAlgorithmDefineWidget( QWidget* parent = 0,Qt::WindowFlags f = 0 );
#endif
   ~QAlgorithmDefineWidget();
   int getID()const;
   void setID(int v);
 signals:
   void insertClick(algorithm_parameter *a, bool inputAlgorithm);
 public slots:
   virtual void resize(int w, int h);
 virtual void resize(const QSize &s);
 virtual void resizeEvent(QResizeEvent *e);
 virtual void setRuns(vector<run_parameter*> *selectedRuns);
 protected slots:
   virtual void OnInsertButtonClick();
};
```

Input-output-algorithms Some of the algorithms are not for processing data, but for reading data from file or electronics or to write it to disk or other devices. In principle these algorithms are in no way distinguishable from any other algorithm, but for the use in the Graphical User Interface, they are treated a tiny bit different, when it comes to selection. They are not listed with the other algorithms, but you can select them in the ordering-window (see sec. 4.1.2).
If you are adding an IO-algorithm, you probably will need an IO-algorithm-widget for that. It has to be derived from QAlgorithmDefineWidget (class definition 4.7)

Class 4.8 Definition of a new IO-algorithm-widget.

```
class QAMyNewAlgorithmWidget: public QAlgorithmDefineWidget
{
  Q_OBJECT
    protected:
   //for each parameter you need an input widget
  void languageChange();

 public:

#if QT_VERSION < 0x040000
  QAMyNewAlgorithmWidget( QWidget* parent = 0, const char* name = 0,
    WFlags fl = 0 );
#else
  QAMyNewAlgorithmWidget( QWidget* parent = 0,Qt::WindowFlags f = 0 );
#endif
  ~QAMyNewAlgorithmWidget();

 public slots:
   //slots to copy changed parameter values to the algorithm_parameter
};
```

like in this example 4.8.
For each parameter you will need an input widget: check-boxes for boolean, line-edits for floating point, integer and strings, 3D-inputs for vectors and points, list-boxes for the vector-types and AlgorithmDisplays for the algorithm-type-parameters. For each input-widget you will need at least one slot, to catch the changes in the properties, the user performs. Copy them directly to the inherited variable *ap*. Don't forget the connections for these slots. You don't have to free them, if you define them as children of the widget, Qt will do that for you.
Use the install-wizard to insert the widgets into **typeCase**.

4.2.3 Install-logs

The install-wizards give you the possibility to insert algorithms and shapes into your copy of **typeCase**. I really recommend using them for you may easily forget some file where to make changes and I, myself am using them.

Create Install-log

To create an install-log, go to the main-window and select from the menu either “algorithm”–“install algorithm” or “shape”–“install shape”.

Algorithm

The first page already asks you for the install-log, omits that for the moment and select the category and the information about being IO-algorithm.
Page two displays the algorithm-parameter that will be saved for the algorithm.
On page three you provide the files of the algorithm. First go for the header file and select the algorithm you want to use. Then add the source file(s). You will be asked if it should parse for a description. If you defined the *getDescription()*-method, say yes here. If you go back to page two you will see the changes.
Page four shows connections and call-frequencies. Here you can connect the slots you defined to the signals the analyzer provides.
Page five makes you define the assignment of the available variables to the constructors parameters as you will need for the initialization process.
Page six is the finish page. If you proceed on finish, the algorithm will be installed. But you want to save an install-log first, so after having provided the wizard with all information, you go back to page one and click on the “create”-button.

Shape

For the shapes it works as for the algorithms. The install-log-box is defined on page one, but you go through the wizard, supplying all information about the shapes files, parameters and the way, it is created out of a shape_parameter. Then you go back to page one and create the install-log.

Apply Install-log

Applying the install-log is rather simple. There are two ways. First, you choose either “algorithm”–“load install-log” or “shape’–“load install-log” and specify the name of the install-log file. Secondly you can open the install-wizard and click on page one on the “read”-button for the install-logs. This way you can still see and modify the parameters before you click on finish on the last page to install the component.

4.2.4 Plot reactions

Another part is the filling and drawing of histograms for a specific reaction. Most of the functionality has been formulated in a very general way, independent of the type of reaction. But some things remain, that have to be redone every time you define a new reaction-type. Fortunately you can copy most of it from an existing one.

As for the algorithms and shape you'll also here have an abstract base class, that defines the general functionality, derived classes define the reaction specific functionality. The base class is called *reaction_type* and the class definition is shown in secs. D.4 and D.5. These are the methods you **won't** have to modify. But they are using methods that are strongly reaction dependent and these you will have to implement in your derived class.
Your derived class can look like class definition 4.9.
You will have to implement the following methods, each of them defined virtual:

1. int* getIDs()
2. float *getMasses()
3. void setBaseUse()
4. vector<string>getCutNames()
5. string makeComment(bool use[20], int nCuts)
6. void fillCuts(writeOutStruct &stru, int cuton, int nCuts, bool *cuts, float *masses, bool converged)
7. void getStrings(string strings[20], bool use[20], int cuton, float* masses)
8. void qualityFill(TH1F*** histos1, TH2F*** histos2, writeOutStruct &stru, int which, int cuton, int nCuts, bool *cuts, bool use[20], float *masses, bool converged, float weight=1)
9. void observablesFill(TH1F*** histos1, TH2F*** histos2, writeOutStruct &stru, int which, float weight=1)
10. void addDrawingLines(histoStack1 *h1, histoStack2 *h2, float *masses)
11. void setHistoPropertiesDefault(const momentum4D &inputmomentum)
12. bool leafToStruct(TLeaf** leav, writeOutStruct &stru, momentum4D &cms, momentum4D &inM, momentum4D moment[4][4], momentum4D inter[4], momentum4D Pcms[2][3], momentum4D Jmoment[2][3][3], momentum4D jbm[2][3], float *mass,int id[4],vector3D &lDir)
13. bool leafToStruct(TLeaf** leav, writeOutStruct &stru, momentum4D &cms, momentum4D &inM, float *mass,int *id)
14. bool trackTreeStruct2WoStruct(trackstruct &tr, writeOutStruct& stru, momentum4D &cms, momentum4D &inM, float *mass)
15. void event2Pstruct(istream &input, writeOutStruct& tracks, const momentum4D &initSystem, int *particleids)
16. void setTexFile(texFileMakeup &makeup)

For a reference see Appendix D.6.6 or use
`http://www.pit.physik.uni-tuebingen.de/~ehrhardt/KTOF/`.
Important again is the initialization of some variables in the constructor:

fN2D	number of 2D histograms
fN1D	number of 1d histograms
fNparticles	number of particles
fNangles	number of angles to store for PreCutTree-format
fNmissingMasses	number of missing-masses to store for PreCutTree-format
fNinvariantMasses	number of invariant-masses to store for PreCutTree-format
fkinfitNdF	number of degrees of freedom for kinematic fit
HistogramDefinitionDefault	=new histoProperties[fN2D+fN1D] will be deleted by base class
HistogramDefinition	=new histoProperties[fN2D+fN1D] will be deleted by base class

But best advice I can give is to copy the source file from a related reaction and change the code to fit your needs.
In class definition. 4.9, you can see the example definition of a simple plot-reaction including only the essential functions.

Class 4.9 Definition of a plot-reaction-class

```
class example_reaction: public reaction_type
{
public:
 example_reaction();
 ~example_reaction();
 virtual int* getIDs();
 virtual float *getMasses();
 virtual void setBaseUse();
 virtual vector<string> getCutNames();
 virtual string makeComment(bool use[20], int nCuts);
 virtual void fillCuts(writeOutStruct &stru, int cuton,
    int nCuts, bool *cuts, float *masses, bool converged);
 virtual void getStrings(string strings[20], bool use[20],
    int cuton, float* masses);
 virtual void qualityFill(TH1F*** histos1, TH2F*** histos2,
    writeOutStruct &stru, int which, int cuton, int nCuts,
    bool *cuts, bool use[20], float *masses, bool converged,
    float weight=1);
 virtual void observablesFill(TH1F*** histos1, TH2F*** histos2,
    writeOutStruct &stru, int which, float weight=1);
 virtual void addDrawingLines(histoStack1 *h1, histoStack2 *h2,
    float *masses);
 virtual void setHistoPropertiesDefault(
    const momentum4D &inputmomentum);
 virtual bool leafToStruct(TLeaf** leav, writeOutStruct &stru,
    momentum4D &cms, momentum4D &inM, momentum4D moment[4][4],
    momentum4D inter[4], momentum4D Pcms[2][3],
    momentum4D Jmoment[2][3][3], momentum4D jbm[2][3],
    float *mass,int id[4],vector3D &lDir);
 virtual bool leafToStruct(TLeaf** leav, writeOutStruct &stru,
    momentum4D &cms, momentum4D &inM, float *mass,int *id);
 virtual bool trackTreeStruct2WoStruct(trackstruct &tr,
    writeOutStruct& stru, momentum4D &cms, momentum4D &inM,
    float *mass);
 virtual void event2Pstruct(istream &input,
    writeOutStruct& tracks, const momentum4D &initSystem,
    int *particleids);
 virtual void setTexFile(texFileMakeup &makeup);
};
```

Chapter 5

Measurements

The measurements analyzed for this work were recorded in October 2004 with the COSY-TOF-detector (see fig. 2.10(b)) at the Forschungszentrum Jülich.
The measurements were dedicated to the search for the $\theta^+_{(1520)}$-particle, with a positive signal reported in several publications ([43], [44], [50]). To serve this purpose the beam-momentum was chosen to be $3.059 GeV/c$ translating to a beam-kinetic-energy of 2.261 GeV. A closer look at the data revealed the real beam momentum $P_{beam} = 3.081 GeV/c$ and $E_{beam} = 2.282 GeV$. At this energy the θ^+-resonance was expected to show up in the center of the invariant mass distribution of the reaction-products proton and K_s. However no positive signal could be derived from the data, but an upper limit to the cross-section [44].

5.1 Detector setup

For this experiment the detector setup of the 3m-version was chosen, including Start (sec. 2.2.3), Micro-Strip (sec. 2.2.4), two Hodoscopes (sec. 2.2.4 and 2.2.4), Barrel (sec. 2.2.5), Quirl and Ring (sec. 2.2.5). For details see Tables in Appendix E.
Unfortunately for the particle identification the calorimeter was cut off for that measurement due to missing QDC-modules.

5.2 Trigger

In collision experiments the wanted reaction is seldom the most abundant one. Also the beam intensity and target density give a reaction rate that is far too high to be handled by the Data AcQuisition system (DAQ). A selection has to take place even before recording, to enrich the data with the wanted reaction.
This is the task of the trigger system. It gives a positive signal when the hits in the detector show a desired pattern. For elastic scattering this would be two hits in a stop detector and two hits in each layer of start. This is called a multiplicity of two in both start and stop detector. For the reaction $pp \rightarrow pK_s\Sigma^+$ the pattern looks

as follows: the proton passes the detector generating signals in all layers – one in each start and stop –, the Σ^+ – decaying with a decay length of 24.04 mm – gives a signal in start and the charged one of it's decay products gives a signal in the stop detector; the K_s decays with a decay length of 26.84 mm gives no signal in start but its charged decay products ($\pi^+\pi^-$) give two signals in the stop detector. The trigger was set to have a multiplicity of 2 in start and a multiplicity of 4 in stop.
For calibration purposes there was also a two-track trigger switched on but with a pre-scaling of 400.
These trigger conditions were not only met by the reaction $pp \rightarrow pK_s\Sigma^+$ but also by the reaction $pp \rightarrow pK^+\Lambda$ with both proton and K^+ traversing the detector; the K^+ without decay (decay length for K^+ is 3712 mm). This gives two start and two stop signals. Λ is neutral, so there is no signal in the start (decay length 78.9 mm) but its charged decay into $p\pi^-$ (64%) gives two signals in stop fitting exactly into the trigger. Unfortunately, considering the luminosity or, in other words, the time between the individual proton beam-bunches, the gates for the stop detector were set too long leading to a mixing of events. If in two successing beam bunches interactions took place with the target and there were two tracks each, there were stop signals fed into the trigger-system for each of the four tracks but, since the gate for start was considerably smaller, only two signals for the start detector. This gave rise to a high amount of recorded elastic interactions and single pion production events within the data even with the hyperon trigger.

Chapter 6

Calibration

The data acquisition system of the detector returns raw data, that indicates which detector element has generated a signal and provides information on signal-amplitude (QDC) and timing (TDC). These QDC and TDC values are delivered as integer values that have to be converted into more suitable and comparable units like – as used for this work – nano-seconds and GeV. This conversion along with the decision which signal is a valid one is called calibration.
The following procedures have to be applied:

- Before tracking
  - Apply cuts in QDC and TDC
  - Convert QDC to energy
  - Convert TDC to time
  - Apply a walk-correction to the time-information
- After tracking
  - Correct the time-information for the run time of the signal in the material
  - Apply a pulse height correction
- After particle identification
  - Do a quench-correction

6.1 Common calibration

In spring 2005 the TOF-collaboration agreed on a well-defined calibration method, a common set of calibration parameters and a common file format for distribution of these parameters. The following formulae are from this agreement:

$$t_{counter} = \quad Offset_{TDC} - ((TDC + rnd) * f_{TDC} - walk - t_{inDetector}) \quad (6.1)$$

$$walk = \quad \frac{A}{(QDC - P_{mean}) + B} + D * \log(QDC - P_{mean}) \quad (6.2)$$

$$E_{counter} = \quad (QDC - P_{mean}) * f_{QDC} \quad (6.3)$$

The values $Offset_{TDC}$[ns], f_{TDC}[ns], A[ns], B[channels], D[ns], P_{mean}[channels], f_{QDC}[GeV], TDC_{high}[channels], TDC_{low}[channels], QDC_{high}[channels], QDC_{low}[channels] are calibration constants. They have to be determined for every detector element separately and will not be given as an Appendix to this work ($\sim$1400 elements $\times$ 11 constants $\times$ 500 runs = too much numbers). But the calibration-data-basis can be retrieved via the **typeCase**-home-page ([23]).
In the first calibration-step the following steps are done:

$$\begin{aligned} t_{counter} &= Offset_{TDC(event-number)} - ((TDC + rnd) * f_{TDC} - walk) \\ walk &= \frac{A}{(QDC - P_{mean}) + B} + D * \log(QDC - P_{mean}) \\ E_{counter} &= (QDC - P_{mean}) * f_{QDC} \end{aligned}$$

Here, also cuts are applied, setting hits as invalid, when the QDC- or TDC-value is not within certain limits. The QDC-cuts are to cut away the so called pedestal, which is in fact the noise the detector and its photomultiplier/preamplifier produce. There appear signals, with a Gaussian distribution in pulse-height, that do not correspond to any real hit.
In an experiment, there were always certain runs, that record the noise of the detector-system. These distributions are fitted and one applies, already during the experiment a threshold on the voltage, to suppress this pedestal and record only the real signals, with higher energy. However, this pedestal suppression only sometimes works perfect, there can be a lot of garbage hits in the data, that have to be cut away. On some channels however, the thresholds can be to high cutting into the desired data; this has to be taken into account for simulations.
For the TDC cuts, one has to consider, that it may be necessary to use only hits in a special time window for analysis. The TDC cuts in analysis have been set very wide for the stop detectors and to $\pm 2\sigma$ of a Gaussian distribution for the Start detectors.
The other corrections are applied after tracking, when the complete angular information of the track is available. Here

$$t_{counter,mod} = t_{counter} - t_{inDetector}$$

and

$$E_{counter,mod} = E_{counter} * f_{QDC}(\theta)$$

with $t_{inDetector}$ being the time the signal needs to propagate from the generation-point in the detector to the photomultiplier for example and $f_{QDC}(\theta)$ a function in dependence of θ or alternatively r, the signal run path to readout are applied. The run-time-correction is only necessary for the detectors where the timing-information is recorded, namely Start, Quirl, Ring and Barrel, where Barrel has a correction due to the pixel calculation, taking the mean of the TDC at both ends of the element. The Start detector has such a short run path for the light, that the correction lies far below the timing resolution, leaving Quirl and Ring.
For the pulse-height-correction, the correction for the fact, that in wedge or spiral shaped detectors the light collection is dependent on the polar angle of the track, this is done also just for the Quirl and the Ring detector. The parameters here come as coefficients of a polynomial of the 5th or 6th order in the distance-to-photomultiplier, which is multiplied to the energy information. Since the QDC-information is not used to gain energy-information on the particle, this algorithm is not applied to the data ([10]).

6.2 Beam time "October 2004"

The data of the analyzed beam-time was special in many respects. One of the big changes compared to earlier and later periods of data-taking was, that a common effort of the whole collaboration was made to analyze this data with respect to the reaction $pp \rightarrow pK_s^0\Sigma^+$. The main goal was to learn something about the penta-quark particle Θ^+, though this goal was only reached by giving an upper limit to its cross-section (see [44]).
But nevertheless, the measured data contains a lot of events of the reaction $pp \rightarrow pK^+\Lambda$ to be analyzed for this thesis and a common set of calibration parameters were generated to be applied to the data. This calibration parameters, along with the formulae to apply them have been supplied by senior members of the collaboration and are also in this analysis applied to the data.
After initial difficulties this calibration worked nicely at least for the QDCs and cuts, since the QDC values are not used to extract actual energies but only to have a binary switch whether a detector element was hit or not. Also the problems coming with the QDC-jumps in the Start-counter could be corrected applying the Teufel-correction (see sec. 3.10.2).
Unfortunately this is not the case for the TDCs. Being a Time-Of-Flight-spectrometer, timing and with it the TDCs are an essential tool and have to be calibrated as exact as possible to make sure that the errors of the extracted velocities are minimal. Here the calibration parameters supplied by the collaboration showed being not sufficient.
Binning-correction seems to be constant over the whole period of data-taking, but the offsets and walk aren't. Drifts and jumps of the offsets can be observed over the whole period. A closer look at the hit pattern of the wedge shaped detectors gives a

hint to the beam direction and magnitude, that also changed during the beam-time. So it is necessary to do a calibration based on the individual run (i.e. approximately one hour of data-taking).

6.3 Geometry calibration

The geometry is crucial for the reconstruction of the tracks the measured particles take through the detector. The better the geometry is defined, the higher the reconstruction efficiency, the better the resolution for the directions of the 4-momentum vectors of the measured particles. There geometry plays an even more important role than the timing.
To define the geometry of the detector, two assumptions are made:

1. The mean interaction point is the origin $(0,0,0)$,
2. The direction of the beam is along the z-axis $(0,0,1)$.
3. By general agreement, the x-axis is at nine O'clock when looking in beam-direction.

With these assumptions the detector definition is done in four steps:

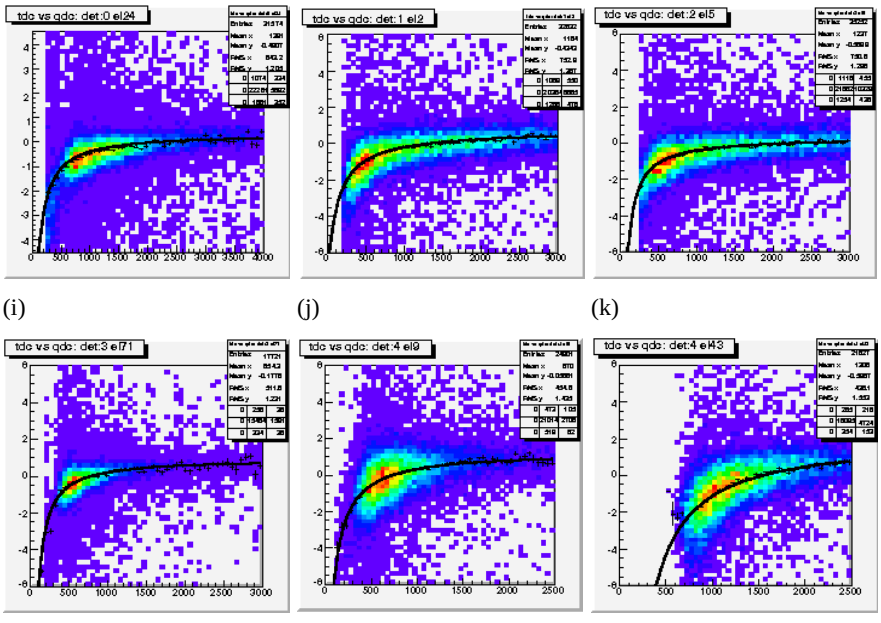
1. Position in x-y-plane:
   For any two particle reaction, the φ-difference of the two particles is 180°. Take a detector with a wide range in θ (here the 2-layered-Hodoscope detector was used). Searching for a two track event means searching for two hits in each layer. Combine these hits to pixels. Shifting the center point of the detector in x-y-plane also shifts the peak in the $\Delta\varphi$ distribution. The correct x-y-position can be found by searching the point where the peak has a minimal width and a position close to 180°.
2. Position in z-direction:
   As in step (1): take two-particle-events, use events with $\Delta\varphi$ close to 180° and plot the value $\frac{1}{\sqrt{\tan\theta_1 \tan\theta_2}}$ which corresponds to the γ-value of the beam in elastic events (see eq. A.24). γ is fixed due to fixed beam-momentum. Modify the z-value of the pixel to fit the mean of the γ-peak to the specified value.
3. Stacking order of the detectors in the tracking-region:
   Take a pixel in Quirl or Ring. Connect the center-point of this pixel to the target to get a straight line. For each detector in the tracking region calculate for each hit of the considered event the distance of the line to the element's volume. Plot this distance versus element number. A line appears in this plot. If it is flat at zero, the detector is in the right position, an offset translates into a shift in direction perpendicular to the elements length, a slope translates into a shift in z-direction. Take only z positions into account in this step.

4. Using the pixels from step (3), plot the θ-difference of the Micro-Strip to the pixels in hodoscopes versus φ of the Micro-Strip pixel. Modify the x-y-position to make the graph flat. Do the same for Quirl and Ring respectively. Plot the θ-difference of the hodoscopes to the pixels in the other detectors over θ. Modify the z-position to make the graph flat.

5. Iterate steps (1) to (4) until there is no further modification.

Note that a tilt of the beam-axis to the symmetry-axis of the detector is not taken into account!
The geometry of the detectors and their layers are at least for each of them given by construction. This fixes the actual size. The position and alignment direction is however not fixed a priori. By construction the three layers of Quirl and Ring each are fixed on each other, as well as the two layers of the Start detector.

6.4 Walk-correction



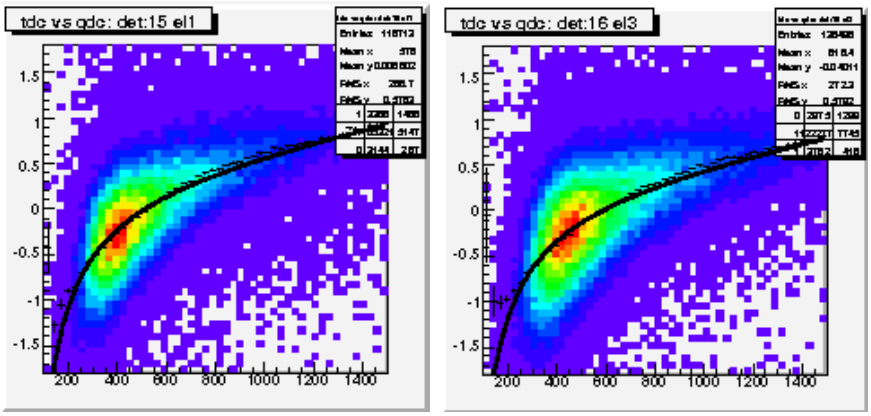
(i) (j) (k)

(l) (m) (n)

Figure 6.1: These plots are generated during calibration as control plots. They show preliminarily corrected TDC-differences [ns] in dependence of QDC [channels] for detectors QuirlS (a), QuirlL (b), QuirlR (c), RingS (d), RingL (e), RingR (f). The black line is a fit function. Its parameters are used later-on for correction.

The walk occurs for small signals. For a signal to be recorded, it has to have a height larger than a certain threshold. The moment when this threshold is reached

is taken as time. The fraction of the voltage-at-timing to voltage-at-maximum is different for small signals and large signals since the aforementioned threshold is constant. To compensate this dependency of time to signal height the walk correction is applied.
Plotting the time [ns] vs. QDC-value, this dependency can be seen and fitted with an appropriate function.
There are a lot of discussions of which function can be considered as appropriate



(a) (b)

Figure 6.2: These plots are generated during calibration as control plots. They show preliminarily corrected TDC-differences [ns] in dependence of QDC [channels], for StartA (a) and StartB (b). The black line is a fit function. Its parameters are later used for correction.

and also whether a single TDC should be used for this purpose. The single TDC has no real meaning in the experiment, but the TDC-difference between two detectors on a single track has. But taking the time-of-flight here mixes the walk of two detectors and the correction has to be done several times iteratively.
There are two possible constructs: tracks and pixels on which this calibration can be done. The pixels in Quirl and Ring are unambiguous and therefore timing is considered within the pixel. The two Start layers are taken from a known track. For Barrel it becomes a bit more difficult, since there exists only one layer that is read out on both sides. Time-difference between front and back contains still position information, but this can be determined beforehand and subtracted for the

walk plots. This leaves the following:

$$\begin{aligned}
TDC^{Q/R}_{left} - TDC^{Q/R}_{right} \quad & -vs- \quad QDC^{Q/R}_{left} \\
TDC^{Q/R}_{right} - TDC^{Q/R}_{left} \quad & -vs- \quad QDC^{Q/R}_{right} \\
TDC_{startA} - TDC_{startB} \quad & -vs- \quad QDC_{startA} \\
TDC_{startB} - TDC_{startA} \quad & -vs- \quad QDC_{startB} \\
TDC_{Barrel_{back}} - TDC_{Barrel_{front}} - t_{position} \quad & -vs- \quad QDC_{Barrel_{front}} \\
TDC_{Barrel_{front}} - TDC_{Barrel_{back}} - t_{position} \quad & -vs- \quad QDC_{Barrel_{back}}
\end{aligned}$$

The applied procedure is:

- for all events
  - for all tracks/pixels in the event
    - fill the defined histograms for all (Start, bent Quirl/Ring, Barrel)-detector elements on track.
- for all considered detector elements
  - produce a maximum histogram with QDC-axis.
  - fit the maximum histogram with

$$f^i_{(QDC)} = \frac{A}{B+QDC} + C * \log(D+QDC) + E$$

  - save parameter for next iteration.

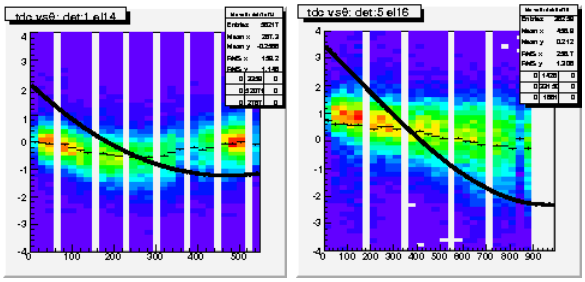
Since the two bent layers of each Quirl- and Ring-pixel have the same signal-run-time –canceling out in the TDC-difference– walk-corrections can be calculated before the signal-run-time correction. For the straight layers the formula is:

$$TDC_{straight} - \frac{TDC_{left} + TDC_{right}}{2} \quad -vs- \quad QDC_{straight}$$

This uses both bent and straight elements. Here the signal-run-calibration (sec. 6.5) is needed and has to be generated before the – in this case – non-iterative walk correction. The parameter after this last step are saved as calibration parameters.

6.5 Signal-run-correction

The time the signal of a hit needs to come from the creation point in the detector element to the element readout has to be corrected for. This signal is – in case of the scintillators – the light-flight-time. In the wedge shaped elements it was found that this time is simply the distance of the entrance point of the particle to the read-out divided by the speed of light in the material.



(a) (b)

Figure 6.3: These plots are generated during calibration as control plots. They show preliminarily corrected TDC-differences [ns] in dependence of signal-run-path [mm], for QuirlL (a) and RingL (b). The black line is the overall correction function. Do not feel disturbed, that the line does not lie on top of the points. This is due to the fact, that the plot contains already some correction.

As for the bent elements of Quirl and Ring the situation is worse, a linear correction is not sufficient. Here a distance-to-PMT(photomultiplier-tube)-dependent offset has to be applied. A fourth order-polynomial has been found suitable.
This is done along with the walk-calibration, one step before the straight elements of Quirl and Ring are calibrated and once after.
The distributions that are fitted are

$$(TDC_{bent} - TDC_{straight}) \quad -vs- \quad (distance - to - PMT)$$

6.6 TDC-Offset

The offset-calibration comes in two steps. The offset-calibration is the last calibration step, so all other TDC-calibration-types (binning, Walk, signal-run-calibration) are applied before the offsets are calculated.
In the first step the TDC peak positions of all elements are shifted to a common value (500 ns is used in this work, but it doesn't matter, since it cancels out in TDC-difference). Here the TDC of elements on known tracks (or pixels in the case of Quirl and Ring) are used.
Drifts of the peak-position of the TDC with event-number are quite common (see fig. 6.4(b)), in some runs however, jumps in the TDC-baseline happen (see fig. 6.4(c)). The jumps happen for all elements of all detectors at roughly the same position, but the height of the jump as well as the drift slope are quite different. For some elements it almost vanishes. This results in the problem that, if the time-of-flight is correctly adjusted for the first event in the run, the time-of-flight at the end of the run can be off by 10% or more. This can be overcome by firstly introducing an event number dependent TDC-offset, here a straight line was found to be sufficient, and secondly dividing the run into sections that have their own offset-parameter-set.

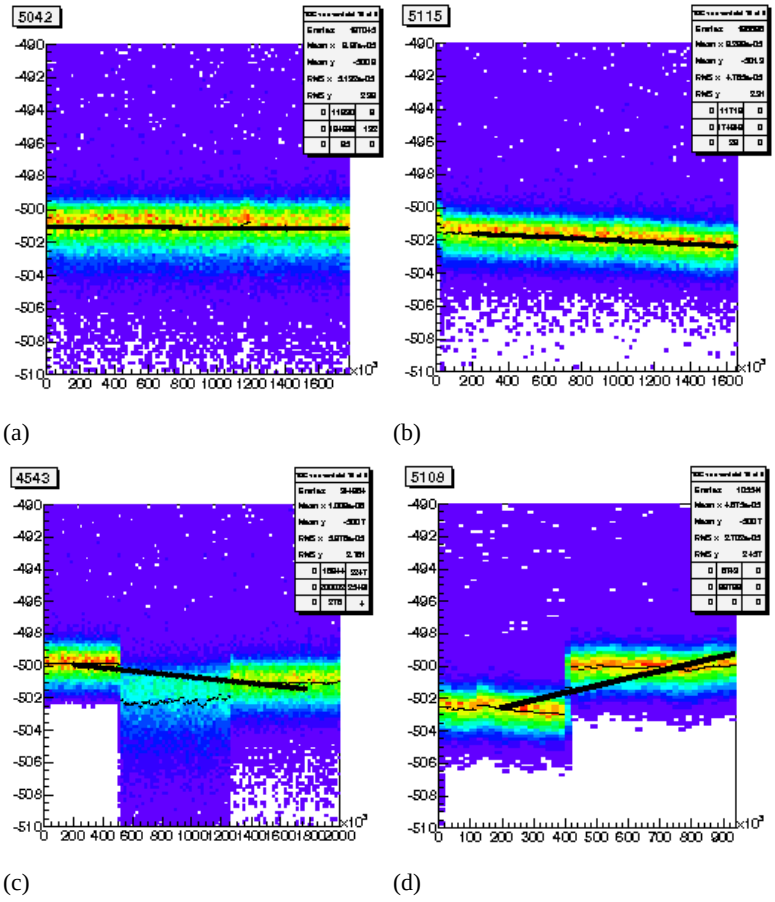
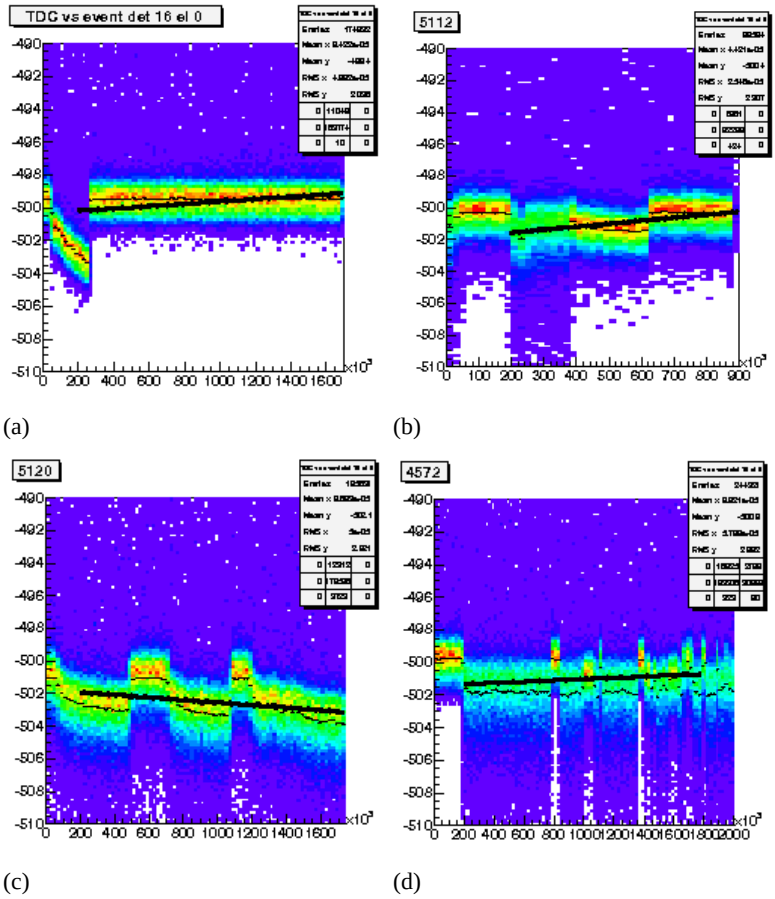


Figure 6.4: TDC distribution in dependence of event-number (record time) for runs 5042 (6.4(a)), 5115 (6.4(b)), 4543 (6.4(c)) and 5108 (6.4(d)). Due to the small number of jumps and constant slope runs of these types can be used for analysis.

The second step uses events of the type pp-elastic, where with two known polar angles the complete kinematic is known (eq. A.23) with even tight restrictions on these angles (eqs. A.24). The routine applies tight cuts on the coplanarity of the two tracks (2°) and the fulfillment of the elastics-condition. In these events velocities of the two protons can be calculated, with them the flight-path and eventually the time-of-flight.
The difference of the time-of-flight calculated from the angles (TOF_a) is then compared to the time-of-flight which is measured (TOF_m). Out of this plot $TOF_a - TOF_m$ two values can be extracted: the mean value of the Gaussian shaped distribution is the absolute offset of the stop detector to the Start detector and has to be added to the offsets found in step one and secondly the width of the distribution gives the detector-time-resolution.



(a) (b)

(c) (d)

Figure 6.5: TDC distribution in dependence of event-number (record time) for runs (6.5(a)), 5112 (6.5(b)), 5120 (6.5(c)) and 4572 (6.5(d)). Runs of these types cannot be corrected using the standard procedures due to the sheer number of jumps and the nonlinear behavior.

6.7 Calibration procedure

The complete TDC-calibration procedure is done in the Module ATDCcalibration, which is part of the typeCase analysis software. It performs the following steps (as described above) and writes the result to text-file:

1. element-offset-calibration (preliminary)
2. position dependence of TDC-difference in Barrel (preliminary)
3. walk-calibration iterative
4. signal-run-correction (preliminary)
5. walk-calibration non-iterative

6. signal-run-correction
7. position dependence of TDC-difference in Barrel
8. element-offset-calibration
9. detector-offset-calibration
10. determination of the TDC-cuts for the Start detector

Several calibration steps are done repeatedly as the element-offset-calibration, which is necessary in the beginning to center the following calibration histograms or to cancel dependencies in other variables. This improves accuracy (see sec. 7.7.2).

6.7.1 QDC-cuts

In the common calibration format there are four values defined for the use of cuts for each detector element. These parameters are QDC_{low}, QDC_{high}, TDC_{low} and TDC_{high}. A valid hit has to have a TDC between TDC_{low} and TDC_{high} and a QDC between QDC_{low} and QDC_{high}.
QDC_{high} is a very large number because all hits having a large energy can be counted as real hits. For the hits with small energy, some threshold has to be applied, to suppress the noise of the detector. A small signal is likely to be no signal at all. To record only a reasonable amount of garbage, thresholds were applied already during data-taking, and software-cuts for the QDC were supplied (one set for the whole beam-time) by the collaboration. These QDC_{low}-values should be the minimum of the QDC-distribution, between the right tail of the Gaussian shaped noise and the Landau shaped data-signal (see fig. 6.7). But even after applying the supplied software-cuts for most elements there remained a lot of noise. Unfortunately the plots are far from being easily fitted and the shapes (amount of noise, distance between noise and signal, ...) quite different, so the cuts are supplied by hand. Due to a drift of the QDCs of up to 100 Channels per week, these cuts are to be generated run-wise as the other calibration are. On the other hand some of the elements had QDC-spectra where the threshold during data-taking was set too high and most of the signal was not recorded at all.
This was only done for the stop detectors, being track-defining detectors. For the Start detector this was not necessary because the noise suppression worked well.
The TDC-spectra of the Start detector are Gaussian shaped, therefore a cut can be easily generated during the TDC-calibration-procedure, cutting away side-peaks.
The TDC-spectra of the stop detectors are again Landau-shaped; here we generate only an upper limit at the steep side of the Landau-distribution, cutting away side-peaks. These are generated during the generation and selection of the QDC-cuts, which is essential for the Quirl detector, the side-peaks being quite dominant here.

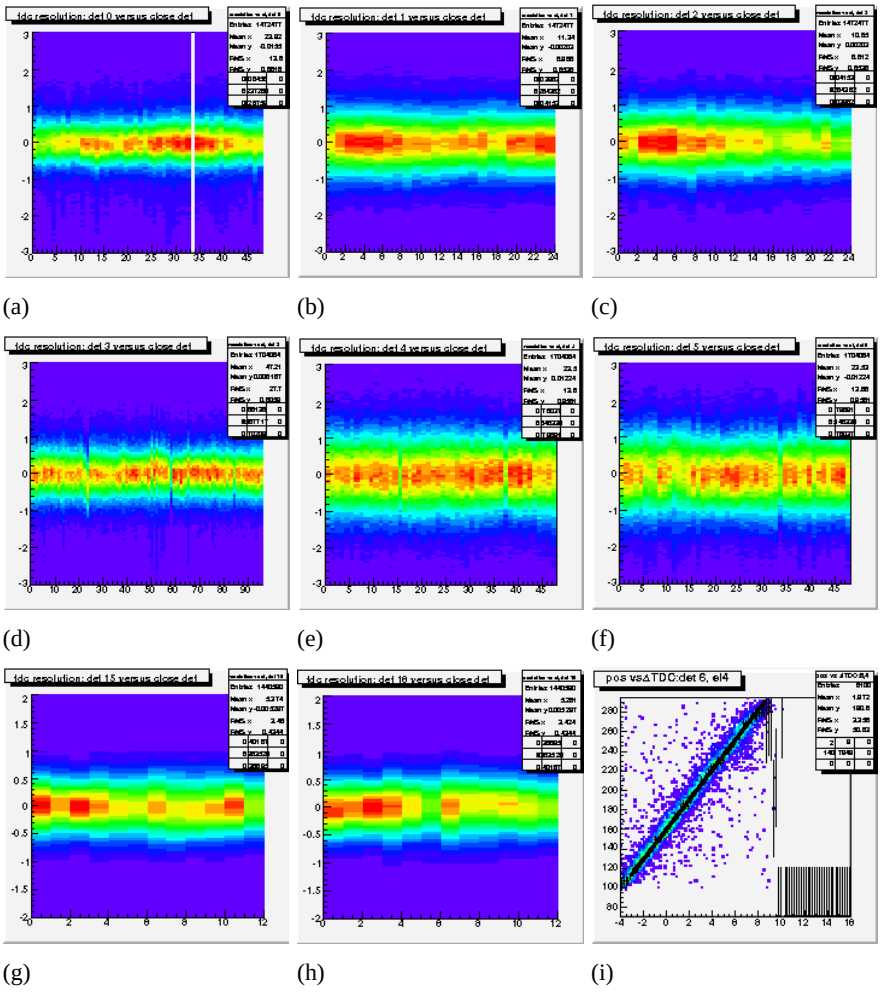


Figure 6.6: Plots (a) to (h) show TDC-difference in dependence of element number after calibration. The difference is taken between two neighboring layers of detectors and should be (close to) zero. Plot (i) shows the calibration plot of the position of a barrel hit. The x-axis shows the TDC-difference between back- and front-readout, the y-axis shows the distance of the entrance-point of the track to the front-readout. You can nicely see the linear dependence. Note: the y-axis in this case is given in cm.

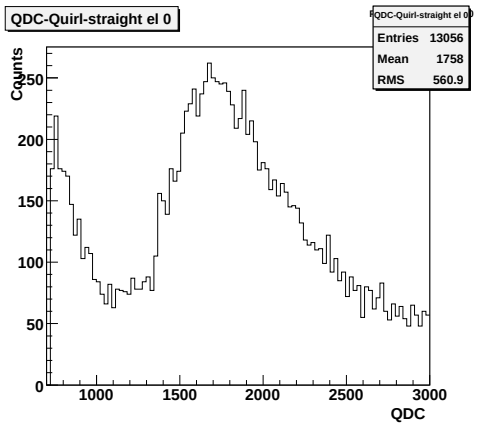


Figure 6.7: QDC-spectrum of the first element of the wedge shaped layer of Quirl. Clearly a rest of noise can be seen on the left side. The optimal cut will be at about Channel 1200.

6.7.2 Parameters

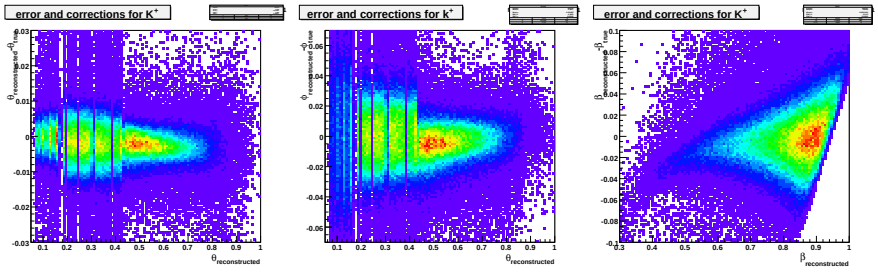
The calibration parameters determined that way can be downloaded as a zip file at `http://www.pit.physik.uni-tuebingen.de/~ehrhardt/KTOF/download/parametersOct04.tar.bz2`
The calibration files are in Common-Calibration-format. The archive contains also the beam-time-parameter file as well as the setup-definition file and the Teufel-correction files.

6.8 Velocity corrections and error determination

6.8.1 Error determination

For a kinematical fit (Appendix. B) a determination of errors is necessary. This is done, using Monte-Carlo simulations, that were passed through virtual detector and later-on through the analysis-software in the same way as data is. After tracks and final (for MCs) corrections, the tracks are written to file (sec. 3.10.1).
Using this file and the purely simulated events (the ones that were fed into the virtual detector) as input, the reconstructed events can be assigned to the original events, the true values of the angles and the velocities can be assigned to the measured values. With this assignment it is possible to generate distributions of the difference of some measured value from its true value. These distributions are Gaussian in shape. The mean value of the fitted function is saved as a correction value and the sigma value can be used as error for the kinematical fit.
It is not sufficient to take errors only dependent on particle and property type, but the errors are quite different for different parts of the detector. So a error-lookup-

table has been implemented and generated, having errors for each particle and property to fit dependent on polar angle θ for the directions and β for velocities (see fig. 6.8).



(a) (b) (c)

Figure 6.8: Deviations of (a)θ, (b)ϕ and (c)β for K^+ from true values in dependence of θ (a,b) and β (c).

6.8.2 Velocity corrections

Unfortunately the corrections generated as described in sec. 6.8.1 and the calibration with pp-elastic-scattering in the calibration routine were not sufficient. So an additional correction step was applied. Here both pp-elastic and $pK^+\Lambda$-events were used to cover the complete detector.
For pp-elastic the process was the same as in the calibration routine:

1. identify pp-elastic using coplanarity ($||\phi_1 - \phi_2| - \pi| \langle 0.05$) and γ-elastic-cuts ($\left|\frac{1}{\sqrt{\tan\theta_1 * \tan\theta_2}} - \gamma_{calc}\right| < 0.075$)
2. calculate complete kinematic out of geometry using A.23
3. calculate velocities using $\beta = \frac{p}{\sqrt{p^2+m^2}}$
4. calculate time-of-flight using $tof_{calc} = \frac{path}{\beta}$

For the $pK^+\Lambda$-events it is possible to calculate the complete kinematic out of the geometry of the event, assuming the Λ-direction as connection between primary- and decay-vertex is sufficiently well known:

1. take exact track pattern: 2 charged prompt tracks, 1 neutral decay vee
2. determine momenta of prompt charged and prompt neutral track using momentum conservation
3. determine particle identity using energy conservation using the combination that fulfills it best

4. do a cut on energy conservation
5. calculate velocities using $\beta = \frac{p}{\sqrt{p^2+m^2}}$
6. calculate time-of-flight using $tof_{calc} = \frac{path}{\beta}$

Then the point of entrance of the particles in the individual sub-detectors is calculated, the time-of-flight difference of the previously calculated tof_{calc} to the measured $tof_{measured}$ is plotted in dependence of the distance of this entrance point to the detector readout. A fifth-order polynomial is assumed and fitted to the profile of this 2D-distribution.

Chapter 7

Results

7.1 Simulations

After the analysis of the measured data including calibration, tracking, velocity reconstruction and reaction recognition, the resulting spectra are still convolved with the detectors acceptance and efficiency of both detector response and reconstruction. To get rid of these effects simulations are done.
First step is to generate events of the reaction to investigate, usually these first events are phase space distributed. Then these particles are transported through a virtual detector generating simulated detector response. In best case the output of the simulation software is the same format as data. Then this output is, together with prepared (quite often dummy) calibration, fed into the analysis software. In the optimal case the analysis software is unaware of the origin of the events.
To get the efficiency and acceptance, spectra are generated for both purely simulated events and simulated events, that passed the virtual detector. The ratio of these spectra (purely simulated over passed virtual detector), the correction spectra are to be multiplied to the data spectra to get the efficiency and acceptance corrected spectra:

$$\begin{aligned} \mathsf{H}_{\mathsf{data}} &\quad \dots \textit{ Data Spectra} \\ \mathsf{H}_{\mathsf{sim}} &\quad \dots \textit{ Simulated Spectra} \\ \mathsf{H}_{\mathsf{mc}} &\quad \dots \textit{ Simulated Events trough virtual detector} \\ \mathsf{H}_{\mathsf{efficiency}} &= \frac{\mathsf{H}_{\mathsf{mc}}}{\mathsf{H}_{\mathsf{sim}}} \\ \mathsf{H}_{\mathsf{corrected}} &= \mathsf{H}_{\mathsf{data}} / \mathsf{H}_{\mathsf{efficiency}} \end{aligned}$$

Fortunately the acceptance of the COSY-TOF-detector is rather large and due to the Lorentz-boost almost 4π are covered in the center of mass frame for the reaction $pp \rightarrow pK^{+}\Lambda$. This gives the reason why it is mostly unimportant which kind of reaction mechanism is used for the generation of the MonteCarlo-Simulation for the efficiency and acceptance corrections. Corrections have been generated for

both phase-space-distributed events and a simulation close to the one described in sec. 7.2. The final spectra differ only in the parts of the spectra where the relative efficiency is rather low, which occurs only at the very edges of spectra like $\cos\theta^{cm}$. Therefore the differences are negligible. Nevertheless the spectra shown in this work are corrected using the model-simulations and not the phase-space distributed events.

7.1.1 Simulation

The generation of phase-space-distributed events is done by the GIN phase space generator built into the GEANT simulation software. It generates a random physical event of the specified type along with a phase space weight. Afterwards a Laplacianly distributed random number is generated. If the weight is smaller than the random number, the event is skipped.
Instead of using the simple phase space weight, a modification can be multiplied to it, modifying the differential cross-sections. This can be a resonance using a Breit-Wigner-term (see sec. A.4) modifying the invariant mass spectra, a final state interaction (FSI) or a propagator that also modifies the angular distributions.
The internal GEANT phase-space-generator GENBOD is used in the case of phase-space distributed simulations and for simple cases like elastic scattering. For the more complicated simulations an external program has been written that generates phase-space distributed events, using the ROOT phase-space generator TGenPhaseSpace. This program is written in C++ and therefore much easier to read and to debug. The events generated with this program are written to file(s) and later used as an input for the GEANT simulation program.

7.1.2 Virtual detector

The virtual detector is a computer description of the physical detector. The Fortran based GEANT 3 package developed at CERN is used. The program was originally programmed in Jülich, but further refined in the course of this work.
First step, after initialization of basic data structures, is to define materials. Then the detector volumes are defined as active volumes, few supporting structures as dead material.
Then the event generator is launched, generating a previously specified number of valid events. The particles generated for these events, are transported through the detector volume, taking into account the energy loss by electromagnetic- and hadronic-interactions (the latter can be switched off), decay of unstable particles, until the particles (primary as well as decay products) are either stopped or leave the detector volume.
If some energy loss occurred in one of the volumes defined as active volumes, a timing and energy loss output similar to the output the DAQ[1] produces is generated. This output is written to file. Also the generated particles are written to file

[1] Data AQuisition system

with their 3D-momentum components and the particle ID. Last but not least there are "track" files generated that record – apart from the generated particles – the start vertex, the stop vertex, the particles ID and the initial and the final momentum of each particle processed in the course of the event.
Using track files of data, purely simulated and analyzed simulated events it is possible to determine the reconstruction efficiency and the reconstruction resolution.
For the sake of precision the virtual detector has to be as close to the real detector as possible.

7.1.3 Calibration for Simulation

As mentioned above the output of the virtual detector is fed into the analysis program the same way as real data is. In the course of this analysis a calibration is applied. In the case of simulation a dummy calibration is applied as usual in the COSY-TOF-Collaboration. To have more precise results, a de-calibration can be applied to the simulated values to be able to apply the same calibration constants as for data.
Fortunately processing the whole procedure is not necessary. Already during the event transport smearing is applied for the timing. Since the QDC-signal is used as a mere switch whether the signal is a valid one, an energy-de-calibration is not necessary. This leaves us with the cuts. Cuts in data are necessary to suppress the noise but they are always cutting away valid signals, sometimes significantly. To simulate the way the small signals are treated in data, cuts for simulated events were generated by comparing the spectra of the simulated QDCs and the rest-QDC-spectrum of data and choosing a value, where the relative height (compared to the peak-position and -height) of the spectra is the same. This procedure can be applied after generating simulated events.
There are runs where some parts of the detector have reduced efficiency, that does not depend on event-number, θ-angle, QDC, TDC or other values. To have corresponding events for this already quite large part of the measured data, this has already to be taken care of during simulation. So the simulation code was altered for a number of simulated events to reduce the efficiency for the appropriate parts of the detector.

7.1.4 Background

Since any measurement involves not only the desired reaction but also other so called background reactions, that have track-pattern similar to the one of the studied reaction, these background reactions have to be simulated to make an estimation on the background content of the real spectra and to give an efficiency estimate of the applied cuts.
Studies for background reactions have been made and the results are listed in table 7.1 and 7.2.
The main background however could not reasonably be simulated or even reason-

reaction	σ [mb]	simulated events	track-pattern	after cuts
Data		3046587	662765	34869
$pp \to pK^+\Lambda$	see sec. 7.4	8000000	611208	381308
$pp \to pK^+\Sigma^0$	0.005	500000	10356	100
$pp \to pp\pi^+\pi^-$	2.5	500000	1728	5
$pp \to pp\pi^0\pi^0$	1	500000	472	1
$pp \to pp\pi^0$	3	500000	271	0
$pp \to pn\pi^+$	15-16	500000	109	0
$pp \to pn\pi^+\pi^0$	4	500000	238	0

Table 7.1: Simulated reactions in phase-space distribution. Expectation of contamination in the data-set.

able estimated: The background due to event-mixing. As discussed in the trigger section (sec. 5.2), the gates of the stop detectors were set too long, which made it possible, that a lot of originally two-charged-track events were mixed with a previous two-charged-track event (e.g. two succeeding elastic events or a single pion-production-event and an elastic-event). This background was reduced with tight cuts and a kinematic fit.

7.2 Simulated Resonances and Final State Interactions

Previous studies ([41]) show, that in the energy region of interest mainly three N^*-resonances contribute. Dominantly visible is the P_{11} resonance N^*_{1710} with a width of $\Gamma_{1710} = 100\ MeV$. But there are also contributions from the S_{11} resonance N^*_{1650} with a width of $\Gamma_{1650} = 165\ MeV$, the P_{13} resonance N^*_{1720} with a width of $\Gamma_{1720} = 200\ MeV$ ([46]) and the final state interaction of the two heavy particles in the exit channel of this reaction: p and Λ ([17]).
The resonances are described each with a Breit-Wigner-term (see. eq. A.4). This term changes the Dalitz-Plots and therefore the invariant mass spectra and also the angular spectra, but not sufficiently to describe the data.
According to the meson exchange model, a propagator for the exchanged meson

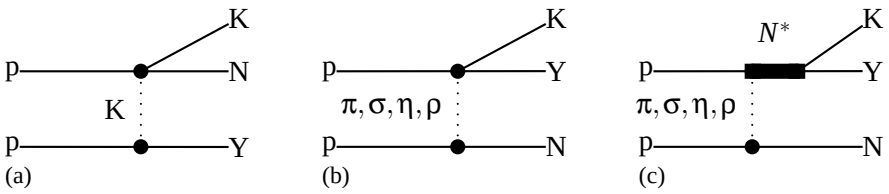


Figure 7.1: Meson exchange model. (a) shows an exchange of a strange meson (Kaon) without he excitation of a resonance. The exchange of a non-strange meson without (b) and with (c) the excitation of a N^*-resonance is shown.

reaction	remaining % after						
	tracking	converged	cut 1	cut 2	cut 3	cut 4	cut 5
Data	3046587	662765 (22%)	50727 (1.7%)	240749 (7.9%)	1650328 (54.2%)	545789 (17.9%)	391153 (12.8%)
$pp \to pK^+\Lambda$	652320	611208 (93.7%)	490042 (75.1%)	477244 (73.2%)	599613 (91.9%)	499463 (76.6%)	565985 (86.8%)
$pp \to pK^+\Sigma^+$	471	220 (46.7%)	3 (0.6%)	53 (11.3%)	100 (21.2%)	94 (20%)	104 (22.1%)
$pp \to pp\pi^+\pi^-$	1728	501 (29%)	5 (0.3%)	104 (6%)	192 (11.1%)	231 (13.4%)	253 (14.6%)
$pp \to pp\pi^0\pi^0$	472	153 (32.4%)	1 (0.2%)	34 (7.2%)	89 (18.9%)	83 (17.6%)	71 (15%)
$pp \to pp\pi^0$	272	51 (18.8%)	0 (0%)	20 (7.4%)	26 (9.6%)	30 (11%)	34 (12.5%)
$pp \to pn\pi^+\pi^0$	238	38 (16%)	0 (0%)	6 (2.5%)	20 (8.4%)	13 (5.5%)	15 (6.3%)
$pp \to pn\pi^+$	109	18 (16.5%)	0 (0%)	2 (1.8%)	11 (10.1%)	5 (4.6%)	11 (10.1%)

Table 7.2: Simulated reactions in phase-space distribution. Work of cuts. Cut 1 is $\chi^2 < 8$-cut, cut 2 is $\alpha_{P_{miss,PK^+},vee} < 4^\circ$, cut 3 is $M_{\Delta p\pi} < 0.04$ GeV, cut 4 is $MM_{PK^+} < 0.07$ GeV and cut 5 is $MM_{PK^+\Delta P\pi} < 0.2$ GeV.

has to be added. Since we are dealing with N^*-excitations, σ-exchange appears very reasonable. For the Roper resonance $N^*_{(1440)}$ it has been shown to be the dominant exchange process for the re-scattered nucleon [8], [9].
Before we continue let's make a short break to discuss the nomenclature of the momenta used in the following formulae: On the production vertex of the N^*-resonance q denotes the momentum-exchange between the two baryons. That is the momentum of the exchanged meson. In superscript the frame in which this momentum is given is provided (q^{cm} is meson-momentum in overall-center-of-mass, q^{N^*} in the N^* rest-frame); q_μ denotes a Lorentz-invariant 4D-momentum vector, $\vec{q}$ a 3D-momentum vector and q the length of the 3D-momentum-vector. On the decay vertex of the N^*-resonance, k is the momentum of the emitted meson (Kaon). Finally for the final state interaction p is the momentum between the two interacting particles (here only proton and Λ have been taken into account).
The meson propagator is then:

$$P = \frac{1}{\left(q^{cm}_\mu\right)^2 - m^2_{meson}} \tag{7.1}$$

This term introduces an angular dependence, as visible in the measured data.
The enhancement at the low invariant mass of proton and Λ compared to phase-space is due to some final state interaction at low relative momentum of the two

heavy articles. The final state interaction can be described

$$\sigma \sim \frac{1}{4}|M|^2 \frac{(p^{cm})^2 + \beta^2}{(p^{cm})^2 + \alpha^2} \tag{7.2}$$

with $|M|^2$ the singlet production matrix element squared (singlet only since the triplet state doesn't contribute according to [17]) and p the proton or Λ momentum in the $P\Lambda$-system. The values α and β are defined as follows :

$$\alpha = \frac{1 - \sqrt{1 - 2\frac{r}{a}}}{r}$$
$$\beta = \frac{1 + \sqrt{1 - 2\frac{r}{a}}}{r}$$

using as parameters the scattering length a and the effective range r; see [17] for further details.
Resonance terms: The resonance amplitudes are described by a Breit-Wigner-term:

$$f_{resonance} \sim \frac{-M\Gamma_{resonance(q)}}{M^2 - M^2_{resonance} + iM\Gamma_{resonance}} \tag{7.3}$$

For each vertex there are terms necessary depending on the spin/iso-spin of the resonance and the exchanged meson. The σ-exchange-vertex contributes with a constant factor. For the three contributing resonances we have the following:

$$f_{resonance} \sim \frac{-M\Gamma_{resonance(q)}}{M^2 - M^2_{resonance} + iM\Gamma_{resonance}} \begin{cases} const. & N^*_{1650}(S_{11}) \\ \left(\vec{\sigma}\cdot\vec{k}\right) & N^*_{1710}(P_{11}) \\ \left(\vec{s}\cdot\vec{k}\right) & N^*_{1720}(P_{13}) \end{cases} \tag{7.4}$$

The momentum dependent width in the numerator and the vertex-factors can be combined into the following expression:

$$c = \sqrt{\frac{\Gamma_f}{k^{N^*}} \frac{\Gamma_i}{q^{N^*}}} \tag{7.5}$$
$$= \sqrt{\Gamma^0_f \Gamma^0_i} \sqrt{\frac{1}{q^{N^*} k^{N^*}}} \sqrt{\left(\frac{k^{N^*}}{k_R}\right)^{2l_f+1} \left(\frac{k_R^2 + \delta^2}{(k^{N^*})^2 + \delta^2}\right)^{l_f+1} \left(\frac{q^{N^*}}{k_R}\right)^{2l_i+1} \left(\frac{k_R^2 + \delta^2}{(q^{N^*})^2 + \delta^2}\right)^{l_i+1}}$$

$$\Gamma_{f,i} = \Gamma^0_{f,i} \left(\frac{k^{N^*}}{k_R}\right)^{2l+1} \left(\frac{k_R^2 + \delta^2}{(k^{N^*})^2 + \delta^2}\right)^{l+1} \tag{7.6}$$

$$\delta^2 = (m_{N^*} - m_\Lambda - m_K)^2 + \frac{\Gamma^2_{N^*}}{4} \tag{7.7}$$

$$\tag{7.8}$$

The complete expression for the cross-section then looks as follows (M the decay particles invariant mass, $\vec{\sigma}$ the spin-operator vector, $\vec{s}$ the spin):

$$\begin{aligned}\sigma \sim \quad & PS \quad * \quad FSI \quad * \quad |Propagator \quad * \quad (aN^*_{1650} + bN^*_{1710} + cN^*_{1720})|^2 \\ =& PS * \underbrace{\left(\frac{1}{4}|\mathsf{M}|^2 \frac{(p^{cm})^2 + \beta^2}{(p^{cm})^2 + \alpha^2}\right)}_{FSI} * \underbrace{\left|\frac{1}{(q^{cm}_\mu)^2 - m^2_\sigma}\right.}_{propagator} * \\ & \left(\underbrace{e^{i\phi_{1650}} c_{1650} \left(\frac{-M}{M^2 - M^2_{1650} + iM\Gamma_{1650}}\right)}_{S_{11}\ N^*_{1650}} + \underbrace{e^{i\phi_{1710}} c_{1710} \left(\frac{-M}{M^2 - M^2_{1710} + iM\Gamma_{1710}}\right)}_{P_{11}\ N^*_{1710}} \right. \\ & \left. + \underbrace{e^{i\phi_{1720}} c_{1720} \left(\frac{-M}{M^2 - M^2_{1720} + iM\Gamma_{1720}}\right)\right|^2}_{P_{13}\ N^*_{1720}} \right) \end{aligned} \tag{7.9}$$

with k_R being the momentum of either particle at the resonance pole position and $\Gamma^0_{f,i}$ –better the relative strength a_i – a fit parameter.
To find the right parameters (listed in table 7.3), a fit was performed using the MINUIT ([19], [20]) minimization tool. The used version was version 2 integrated into the root-analysis-framework programmed in C++. The fitted spectra were the invariant mass spectra of each two prompt particles of the reaction $pp \to pK^+\Lambda$ (figs. 7.17(a), 7.17(b) and 7.17(c)). Using the named three resonances N^*_{1710}, N^*_{1650} and N^*_{1720}, the spectra could be reasonably well described though in the center of the invariant mass spectrum of $p\Lambda$ (7.17(b)) there is a structure known as cusp-effect (see sec. 7.8.3). To describe this structure, a Breit-Wigner-term was introduced due to its peak-like structure, no resonance-like process whatsoever was intended though using this formula! This term was added as intensity, not on amplitude-level.

7.3 Normalization

In order to get the total cross-section σ_{tot} and the correctly scaled differential cross-sections, a normalization has to be done. The process of normalization results normally in a single value with the unit $\frac{barn}{N_{corrected}}$. Multiplying this number to the efficency and acceptance corrected graphs and the corrected total number of events, returns the total cross-section and the correctly scaled differential cross-sections.
To retrieve this number, a reference reaction has to be chosen with a known cross-section.

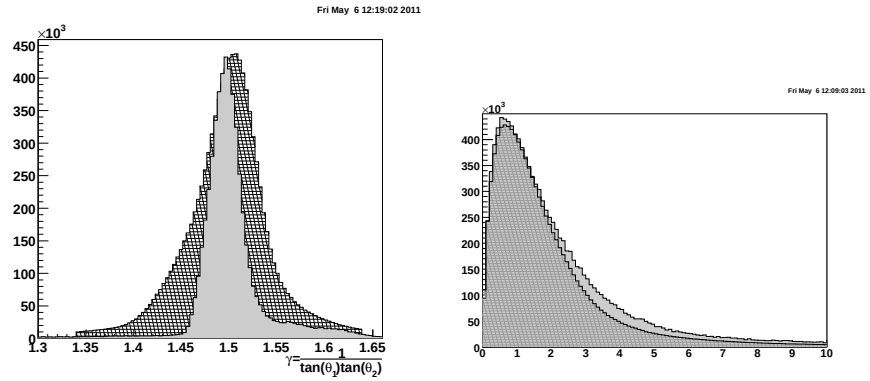
Resonances		Value
M_{1650}	S_{11}	1.653GeV
Γ_{1650}		0.1683GeV
ϕ_{1650}		0.514419
a_{1650}		0.166
M_{1710}	P_{11}	1.712 GeV
Γ_{1710}		0.0986GeV
ϕ_{1710}		0.278949
a_{1710}		0.17
M_{1720}	P_{13}	1.731 GeV
Γ_{1720}		0.3826GeV
ϕ_{1720}	(fixed)	0
a_{1720}		0.61
CUSP		
M_{CUSP}		2.138 GeV
Γ_{CUSP}		0.025GeV
a_{cusp}		0.078
FSI		
a		-2.23 fm
r		1.4 fm
Meson exchange propagator		
m_{meson}		0.400GeV (σ)

Table 7.3: Parameters used for the described model. The relative strengths a_i used as fit parameters are dimensionless and $\sum a_i = 1$

Here pp-elastic-scattering was chosen, where the differential cross-section depending on the polar angle in the CM-system (A.5.1) is well known [39].
As for the procedure, a number of events was generated using the aforementioned cross-section retrieved from [39]. These events were fed through a virtual detector and then analyzed as was the procedure for all simulated runs.
From the data there were elastic events extracted using the same data-base as for extracting the $PK^+\Lambda$-events. Here the following cuts were made:

$$\begin{aligned} &N_{tracks} = 2 \\ &-0.05 < |\phi_1 - \phi 2| - \pi < 0.05 \\ &-0.2 < \frac{1}{\sqrt{\tan\theta_1 * \tan\theta_2}} - \gamma_{beam} < 0.2 \end{aligned} \tag{7.10}$$

The resulting $\frac{1}{\sqrt{\tan\theta_1 * \tan\theta_2}}$ distribution is shown in eq. 7.2(a). The elastic gamma peak is clearly visible in the distribution, though there remains background. This can be removed using a closer cut on this distribution and a kinematic fit (eqs. 7.11). Furthermore the correct beam-momentum can be extracted out of this plot, since



(a) (b)

Figure 7.2: (a) $\frac{1}{\sqrt{\tan\theta_1 * \tan\theta_2}}$, (b) χ^2-distribution are shown for both data (hashed) and Monte-Carlo (solid gray).

the γ here is the relativistic γ of the initial system. This γ was found to be $\gamma = 1.488$ which corresponds to a beam-momentum of $P_{beam} = 3.081\ GeV/c$.
After the extraction the data is fed into the kin-fit-engine with the correct beam-momentum as well as analyzed Monte-Carlo-events are. The events are then used to generate graphs for purely simulated, simulated through virtual detector and data, applying the following cuts on all three data-sets:

$$\begin{aligned} N_{tracks} &= 2 \\ -0.05 < |\phi_1 - \phi 2| - \pi &< 0.05 \\ -0.075 < \frac{1}{\sqrt{\tan\theta_1 * \tan\theta_2}} - \gamma_{beam} &< 0.075 \\ \chi^2 &< 1.5 \end{aligned} \tag{7.11}$$

For further study the $\cos\theta_{cm}$-graphs are used. As for the $PK^+\Lambda$-reaction, the plots are corrected using purely generated, Monte-Carlo and measured data (7.1).
Some ranges in $\cos\theta_{cm}$ are chosen for the normalization (fig. 7.3). For these ranges the integral over the literature distribution is calculated and divided by the corrected number of entries retrieved from the histogram.

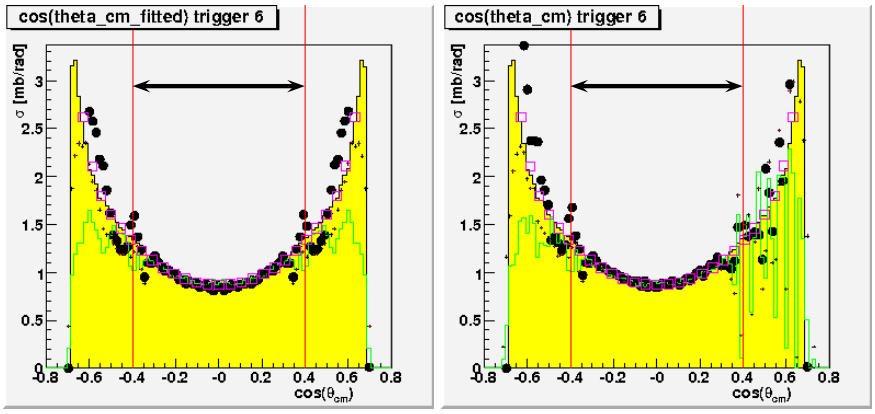
$$f_{normalisation} = \frac{\int^{ranges} \frac{\delta\sigma}{\delta\theta_{cm}} d\theta_{cm}}{\sum^{ranges} N_{corrected}} \tag{7.12}$$

7.3.1 Trigger

Here again the question of triggers arises. There was a dedicated elastic-trigger, requiring two hits in start and two again in stop (start2-stop2). This trigger was pre-scaled by a factor of 400, only every four-hundredth event was recorded.
On the other hand there is an enormous amount of elastic-events in the regular

data-set (see sec. 5.2).

Trigger	elastic-trigger (6)
Number	$1.85 \cdot 10^{-8} \frac{mbarn}{evt}$
Error	$0.85 \cdot 10^{-10} \frac{mbarn}{evt}$



(a) kinfitted (b) non kinfitted

Figure 7.3: $\cos\theta_{cm}$-distribution for pp-elastic scattering. The regions used for normalisation are the ones indicated by arrows.

7.4 Total cross-section

To determine efficiency eight milion events were simulated using the previously described model (sec. 7.2). After virtual detector, analysis and cuts, 4.7% remained. The measurable decay-channel of the Λ into two charged particles is not the only possible decay, it has only a branching ratio of 64% ([46]). Together with the data-events, that passed the cuts and the luminosity derived from pp-elatic-scattering, the total cross-section can be calculated:

	Value	Error
Simulated events N_{GIN}	8e+06	2828.4
through detector N_{MC}	381308	617.5
Data events N_{DATA}	34869	186.7
efficiency: $\varepsilon = \frac{N_{MC}}{N_{GIN}}$	0.04766	7.9e-05
branching ratio into charged decay	0.639	0.005
derived from elastic $\left(\frac{\delta\sigma}{\delta N}\right)$ [mb/Entr]	1.847e-08	8.4e-11

The errors are either from text-book (branching ratios), purely statistical ($\Delta N =$

$\sqrt{N}$) or calculated using Gaussian error-propagation.

$$\sigma_{PK^+\Lambda} = \left(\frac{\delta\sigma}{\delta N}\right)\frac{N_{DATA}}{\varepsilon_{PK^+\Lambda}}\frac{1}{BR} \tag{7.13}$$
$$= 0.0211417mb$$

$$\Delta\sigma_{PK\Lambda} = \sqrt{\left(\frac{N_{DATA}}{\varepsilon_{PK\Lambda}}\frac{-1}{BR}\Delta\left(\frac{\delta\sigma}{\delta N}\right)\right)^2 + \left(\left(\frac{\delta\sigma}{\delta N}\right)\frac{1}{\varepsilon_{PK\Lambda}}\frac{-1}{BR}\Delta N_{DATA}\right)^2} \tag{7.14}$$
$$\overline{+\left(\left(\frac{\delta\sigma}{\delta N}\right)N_{DATA}\frac{-1}{BR}\frac{-1}{\varepsilon^2_{PK\Lambda}}\Delta\varepsilon_{PK\Lambda}\right)^2 + \left(\left(\frac{\delta\sigma}{\delta N}\right)N_{DATA}\frac{-1}{\varepsilon_{PK\Lambda}}\frac{-1}{BR^2}\Delta BR\right)^2}$$
$$= 0.000225497mb$$

$$\sigma_{PK^+\Lambda} = 21.1 \pm 0.2_{stat} \pm 2.0_{sys}\mu b \tag{7.15}$$

As systematic error for the total cross-section 10% of the absolute value were assumed. This is roughly the value used in other publications as well and the amount the total cross-section varied in dependence of different values for the different applied cuts.
The previous publication by M. Schulte-Wissermann ([41]) claimed a total cross-section of $23.9\pm0.3\mu$b, which is comparable within error-bars to the value presented in this work. The main difference lies in the total number of events, which is much lower for this work than in the publication by M. Schulte-Wissermann. With his analysis, he has twice the reconstruction efficiency I have, but with the limitation of having no kinematic fit. In total, especially since the total-cross-sections are so close, the result can be viewed as comparable.
The main reason for the rather small number of events may be found in the already mentioned non-working trigger. A number of 300,000 to 400,000[2] events was expected to be reconstructed from such a long beam-time. Due to this trigger-malfunction, most of the hyperon-trigger-events were just garbage – two successing events with two charged prompt tracks each. The reconstruction of events omitting the time-of-flight information is possible, though many events may survive the applied cuts, that are not really $pp \to pK^+\Lambda$ but probably two single-pion-production events.

7.5 Legend

All one-dimensional plots shown in this chapter have the following color code:

[2] considering expected crossection, requested luminosity, target thickness, efficiency of detector and reconstruction and assumed DAQ-efficiency.

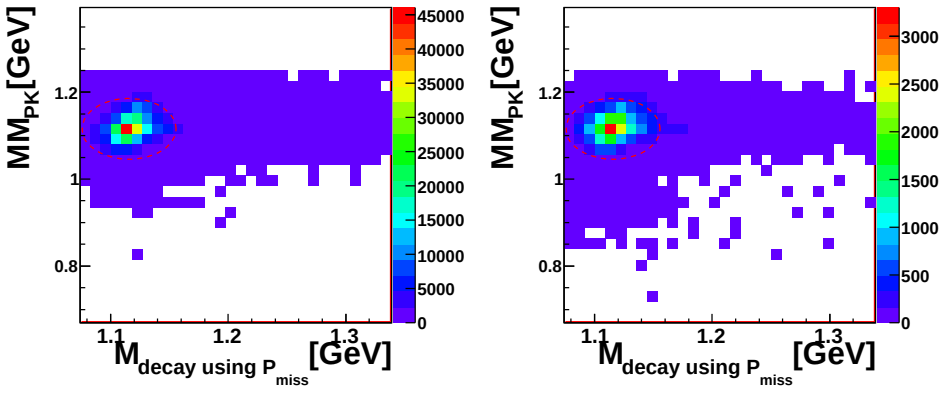
Color	data set
yellow fill	phase space
black dots with error bars	corrected data
blue dotted line	Simulation N^*_{1720}
green dotted line	Simulation N^*_{1650}
magenta dotted line	Simulation N^*_{1710}
red dotted line	Simulation of cusp
red line	**complete Simulation**
gray fill	MC simulation through virtual detector

Plots with observables are shown as phase-space with corrected data and models, plots showing cuts display only uncorrected data along with the Monte-Carlo passed through virtual detector.
Two-dimensional plots are shown in two steps since plotting two 2D-plots on top of each other can be gruesome.

7.6 Analysis

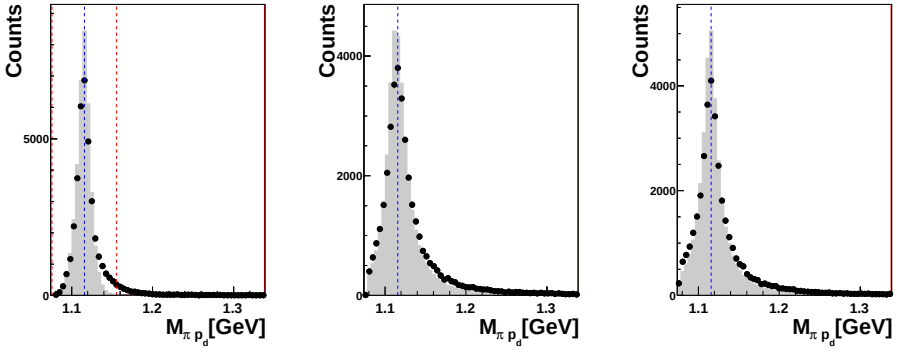
The analysis-program **typeCase**was used for the calibration and analysis of the data. For the complete list of parameters see Appendix. F.

7.7 Selection of $PK^+\Lambda$-events



(a) MC (b) Data

Figure 7.4: The figures (a) and (b) show – before the kinematical fit – the invariant mass of decay-particles p and π^- vs. the missing mass of the two prompt charged particles. Both describe the Λ-particle and the peak is clearly visible at the Λ-mass on both axes.



(a) using all β

(b) using only β_p and secondary vertex

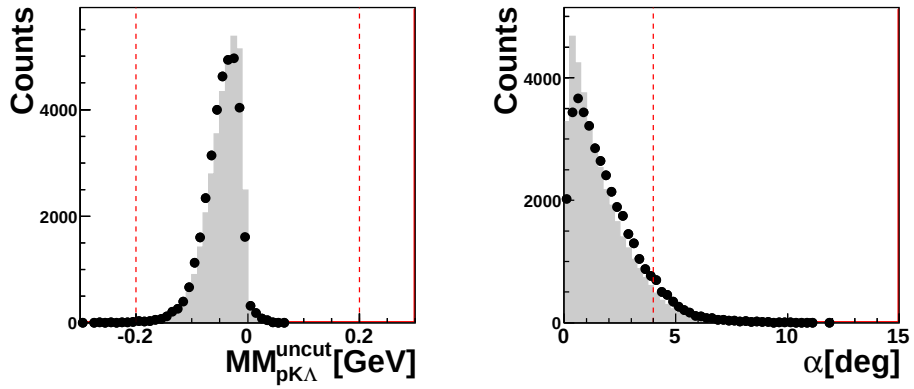
(c) using no β of decay particles

Figure 7.5: Different spectra of $p\pi^-$ invariant mass distributions before kinematical fit. (a) used no geometrical information about the secondary vertex, (c) used purely geometry for calculation ([1]), (b) is a hybrid calculation method. Blue dotted line indicates the Λ-mass, red dotted lines show where cuts are applied. Shaded gray is MC, points are data.

Cuts need to be applied to the data to enrich the data sample with the wanted reaction and reduce background. Nevertheless, there will still remain a non-vanishing background content.
The most stringent cut applied to the data is requesting the exact track-pattern of two charged prompt tracks and one neutral decay. This reduces the data sample by a factor of 500. It should be considerably less considering the applied trigger, but since it didn't work as designed, this is the best that could be achieved.
Next cut is the "forward cut": Applying momentum conservation, the momenta of the three prompt particles (2 charged, 1 neutral) can be calculated by using:

$$\begin{aligned} \vec{P_{initial}} &= p_1\widehat{v_1} + p_2\widehat{v_2} + p_3\widehat{v_3} \\ \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}^{initial} &= \begin{pmatrix} \widehat{v_x^1} & \widehat{v_x^2} & \widehat{v_x^3} \\ \widehat{v_y^1} & \widehat{v_y^2} & \widehat{v_y^3} \\ \widehat{v_z^1} & \widehat{v_z^2} & \widehat{v_z^3} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \\ \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} &= \begin{pmatrix} \widehat{v_x^1} & \widehat{v_x^2} & \widehat{v_x^3} \\ \widehat{v_y^1} & \widehat{v_y^2} & \widehat{v_y^3} \\ \widehat{v_z^1} & \widehat{v_z^2} & \widehat{v_z^3} \end{pmatrix}^{-1} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}^{initial} \end{aligned} \tag{7.16}$$

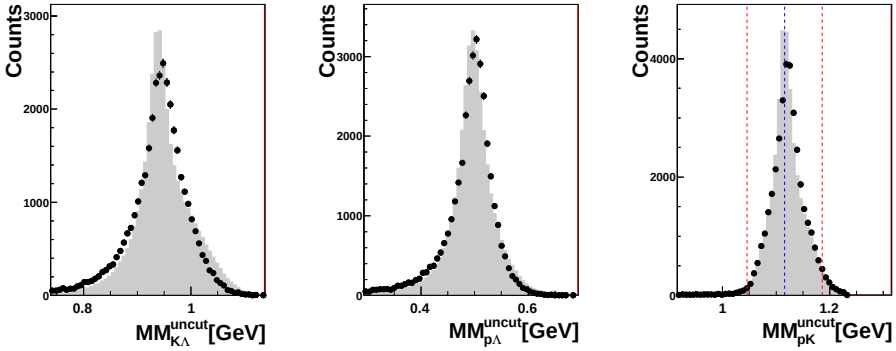
By the fact that all observed particles are forward going, all resulting momenta have to be positive.
Even more stringent cuts on the momenta ranges of the particles can be applied using the so called "Phase-Space-Cut" used in the work of Wolfgang Schröder



(a) $M^{miss}_{pK^+\pi^-p}$ (b) $\alpha^{vee}_{MP_{pk}}$

Figure 7.6: Overall missing mass and $\alpha^{vee}_{MP_{pk}}$, both before kinematical fit. Red dotted lines show where cuts are applied. Shaded gray is MC, points are data.

([49]). Here after particle identification (which is done using energy conservation: $\Delta E = E_{initial} - (E_1 + E_2 + E_3)$ using the combination that produces a smaller ΔE) cuts on the phase-space-limit are applied, allowing only for events where all momenta are in the kinematically allowed momentum-range.
We do not follow this cut method here, but rather follow a more sophisticated procedure: Up to this point particle identification was not necessary. A kinematic fit is applied with all four different permutations, keeping the mass-(or particle)-assignment with the smallest χ^2.
After particle identification, cuts are performed on the missing mass of the two prompt charged particles (using the unfitted values; using the fitted ones result in delta-distributions) to be close to the literature-value of the Λ-mass, as well as the invariant mass of the two decay particles of the neutral particle.
An additional cut is applied on the angle between the connection line from the primary to the secondary vertex and the missing momentum of the two charged prompt particles (see fig. 7.6(b)).
The next cut is applied on the overall missing mass (see fig. 7.6(a)).
Last but not least a cut on the *chi*^{2}-distribution of the kinematic fit is applied (see fig. 7.8(a)). The χ^2-distribution peaks at $(n-2)$ where n is the number of degrees of freedom, that is the number of over-constraints that are imposed on the kinematic due to measuring more properties of the participating particles than necessary to reconstruct all 4-momenta. Not counting the Λ but its decay particles, there are four particles, each with 4 properties (two angles, one mass, one energy/-momentum/velocity) summing up to 16 variables to be determined. The angles of all particles are measured (8), the masses of all particles are assumed (4), the velocities of all particles are measured (4), momentum- and energy-conservation add

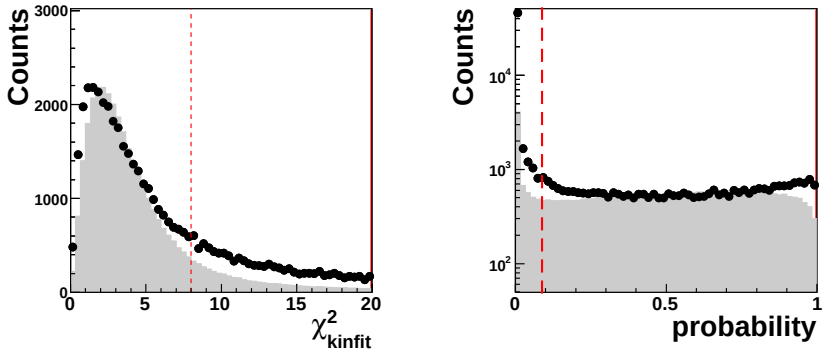


(a) $M^{miss}_{pK^+}$ (b) $M^{miss}_{p\Lambda}$ (c) $M^{miss}_{K^+\Lambda}$

Figure 7.7: Missing Masses of pK^+(c), $p\Lambda$(b) and $K^+\Lambda$(a) before kinematical fit. Red dotted lines show where cuts are applied. Shaded gray is MC, points are data.

another 4 constraints to the equation. Imposing the Λ-mass as additional constraint on the invariant mass of the decay particles p and π^-, give five over-constrains in total.
As a measure of the quality of the data the missing masses of any two prompt particles is plotted in fig. 7.7.

7.7.1 Kinematical fit



(a) χ^2 (b) Probability

Figure 7.8: Properties of kinematic fit, 7.8(a) shows the χ^2 distribution, 7.8(b) is the probability distribution.

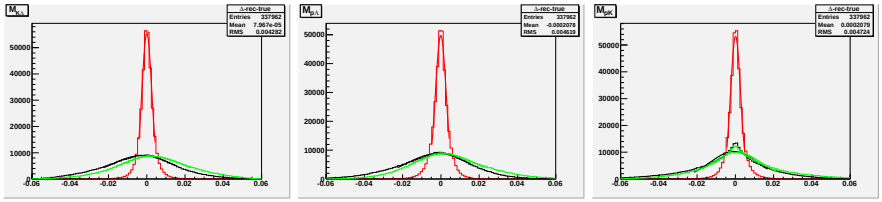
7.7.2 Resolution

Before proceeding to the differential cross-sections, let's make a short stop to investigate the resolution of these distributions, especially here the invariant masses. Of interest here are the central values in table 7.4, these are the differences between the fitted and the true values. For the missing masses these values are naturally tiny, since they are minimized during the kinematic fit. For the invariant masses the resolution is FWHM[3] is about 6 MeV, which is comparable with the values retrieved from more recent COSY-TOF-beam-times using the Straw-Tube-Chamber [26].

	reconstructed-fitted	true-fitted	true-reconstructed
	$M_{K\Lambda}$		
$\sigma\ [GeV]$	0.0165	0.0025	0.0172
FWHM $[GeV]$	0.039	0.0059	0.040
	$M_{p\Lambda}$		
$\sigma\ [GeV]$	0.0168	0.0026	0.0171
FWHM $[GeV]$	0.040	0.0062	0.040
	M_{pK}		
$\sigma\ [GeV]$	0.0130	0.0025	0.0135
FWHM $[GeV]$	0.030	0.0058	0.032
	$M_{p\pi}$		
$\sigma\ [GeV]$	0.0063	(0.00071)	0.0063
FWHM $[GeV]$	0.015	(0.0017)	0.015
	$MM_{K\Lambda}$		
$\sigma\ [GeV]$	0.026	(0.00061)	0.026
FWHM $[GeV]$	0.060	(0.0014)	0.060
	$MM_{p\Lambda}$		
$\sigma\ [GeV]$	0.022	(0.00034)	0.022
FWHM $[GeV]$	0.052	(0.00080)	0.052
	MM_{pK}		
$\sigma\ [GeV]$	0.017	(0.00071)	0.017
FWHM $[GeV]$	0.040	(0.0017)	0.041

Table 7.4: Resolution of the invariant- and missing- mass spectra.

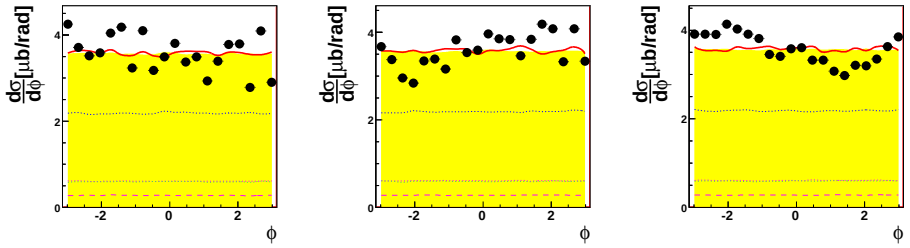
[3] full width half maximum



(a) $M_{K^+\Lambda}$ (b) $M_{p\Lambda}$ (c) M_{pK^+}

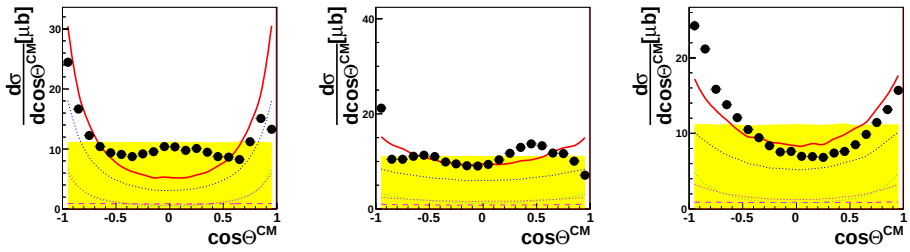
Figure 7.9: Monte-Carlo resolution histograms. Black lines are (reconstructed-fitted)-values, green lines are (reconstructed-true)-values and red (fitted-true)-values, for (a) $M_{K^+\Lambda}$, (b) $M_{p\Lambda}$ and (c) M_{pK^+}

7.8 Kinfitted Graphs



(a) ϕ_p (b) ϕ_{K^+} (c) ϕ_Λ

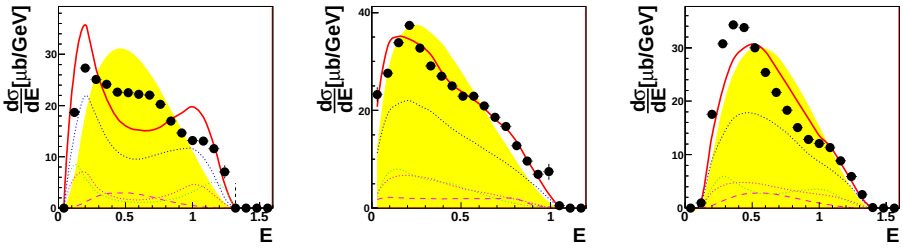
Figure 7.10: The ϕ^{lab}-distribution for proton, K^+ and Λ.



(a) $\cos\theta_p^{cm}$ (b) $\cos\theta_{K^+}^{cm}$ (c) $\cos\theta_\Lambda^{cm}$

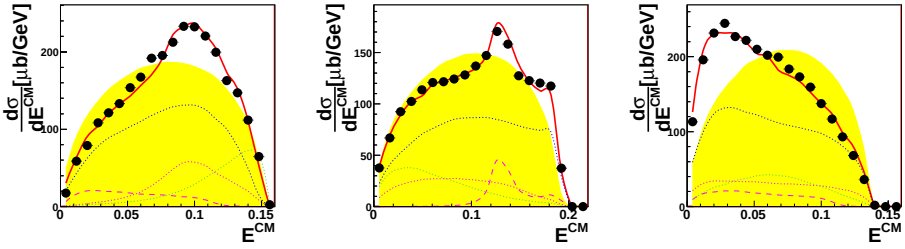
Figure 7.11: The $\cos\theta^{cm}$-distribution for proton, K^+ and Λ.

In this section we discuss the graphs with the kinematically fitted observables. All plots are normalized to σ_{tot} derived in sec. 7.3.
The $\cos\theta_{cm}$-graphs (fig. 7.11) can be used as a measure of the quality of the data reduction. Since the entrance channel consists of identical particles, the exit channel has to be symmetric in $\cos\theta_{cm}$ around 90° or 0. Looking at the θ_{cm}-distributions of the individual particles – proton, K^+ and Λ – one can see, that they are largely symmetric to 90°. For the proton and the Λ the distributions deviate clearly from phase-space.
The ϕ^{lab}-distributions (fig. 7.10) are flat as expected.
The center-of-mass energy distributions (E^{cm}, fig. 7.13) are nicely described by the model.



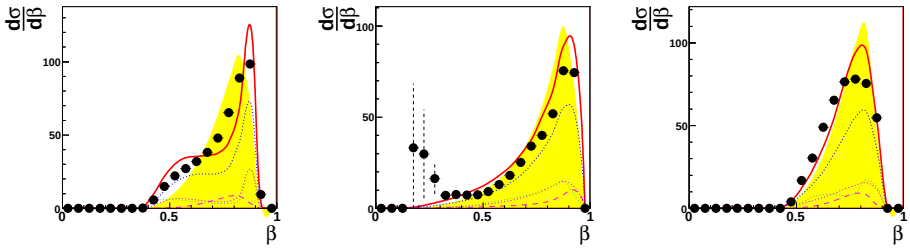
(a) E_p^{lab} (b) $E_{K^+}^{lab}$ (c) E_Λ^{lab}

Figure 7.12: The laboratory Energy-distribution for proton, K^+ and Λ.



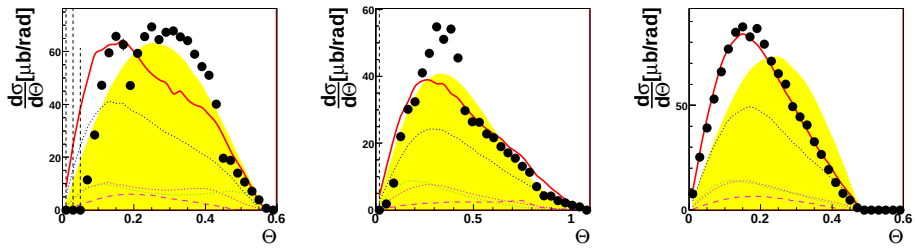
(a) E_p^{cm} (b) $E_{K^+}^{cm}$ (c) E_Λ^{cm}

Figure 7.13: The center-of-mass Energy-distribution for proton, K^+ and Λ.



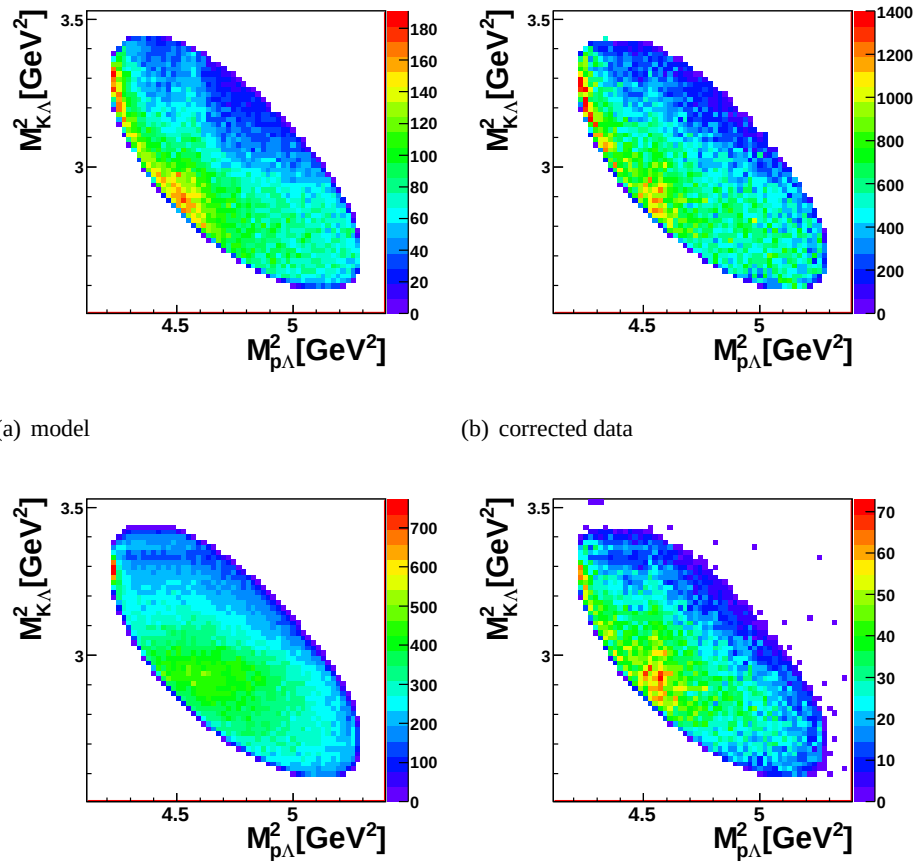
(a) β_p^{lab} (b) $\beta_{K^+}^{lab}$ (c) β_Λ^{lab}

Figure 7.14: The β^{lab}-distribution for proton, K^+ and Λ.



(a) θ_p^{lab} (b) $\theta_{K^+}^{lab}$ (c) θ_Λ^{lab}

Figure 7.15: The θ^{lab}-distribution for proton, K^+ and Λ.



(a) model

(b) corrected data

(c) MC

(d) raw data

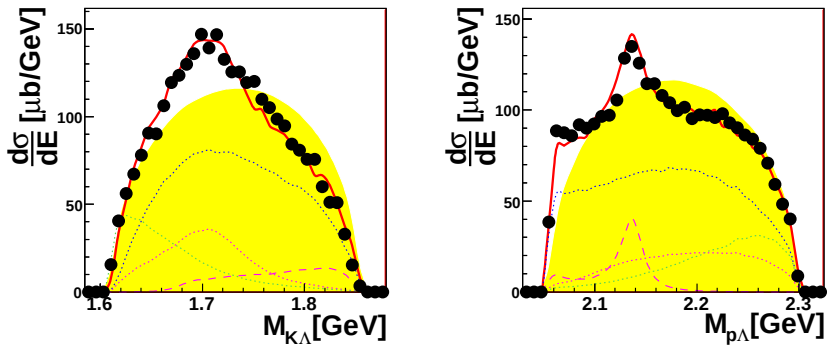
Figure 7.16: Dalitz-plots for Simulation (a), corrected data (b), MC (c), data (d).

7.8.1 Dalitz-plot and projections

The corrected Dalitz-plots (fig. 7.16(b)) agree quite well with the model calculations (fig. 7.16(a)). Here, at the left upper part of the ellipse – at low invariant mass of $p\Lambda$ – the enhancement due to Final-State-Interaction is visible. The bulk enhancement at the lower part, that is almost parallel to the $M_{p\Lambda}$-axis, is the expression of the N^*-excitations described earlier (sec. 7.2). A vertical line in the center is the expression of the Σ-cusp-effect

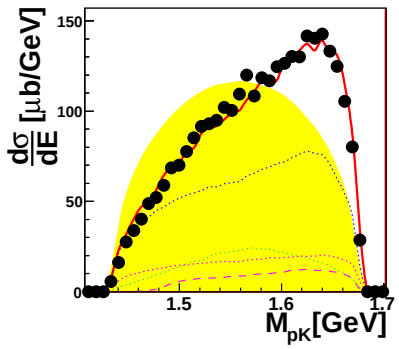
A closer look at the projections reveals more detailed information to the eye.

As well as the Dalitz-plot, the invariant mass spectra (figs. 7.17(a), 7.17(b) and 7.17(c))are nicely described by the model simulations; in the invariant mass spectrum of $M_{p\Lambda}$, there is an enhancement at 2.14GeV, which is due to the cusp-effect (sec. 7.8.3).In fig. 7.17(d), the model-simulation is done using the literature values for the Final State Interaction. Here it is clear, that these parameters are not able to

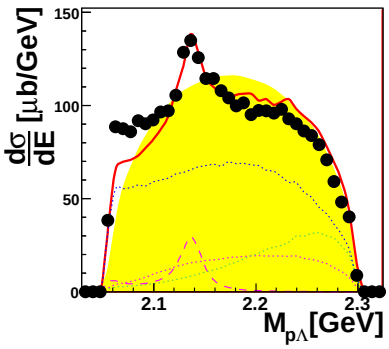


(a) $M_{K^+\Lambda}$

(b) $M_{p\Lambda}$



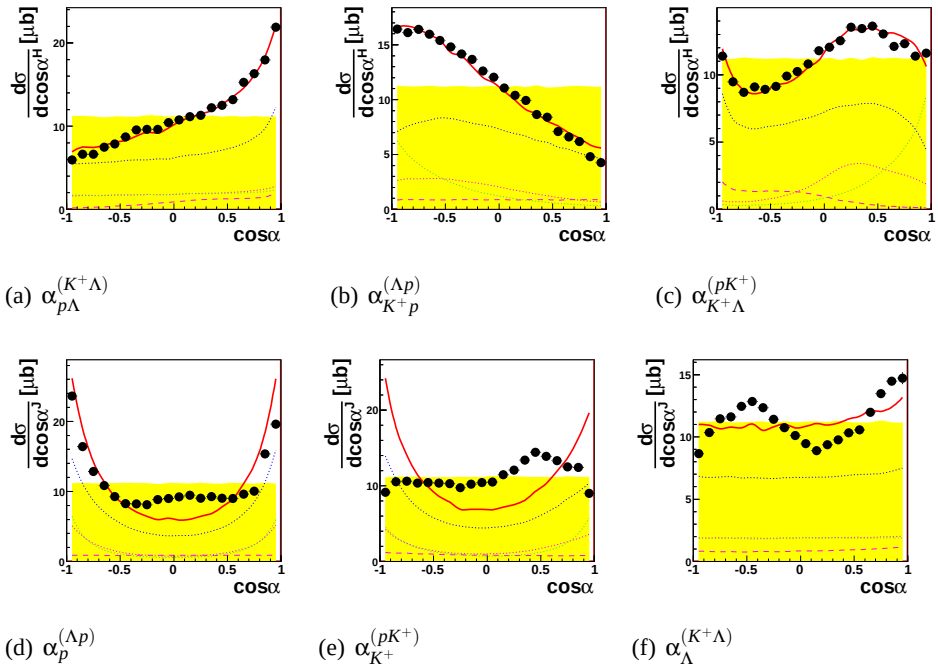
(c) M_{K^+p}



(d) $M_{p\Lambda}$

Figure 7.17: Invariant Masses of $K^+\Lambda$ (a), $p\Lambda$ (b) and K^+p (c). (d) contains for the simulation of FSI the literature values, (b) is adjusted values.

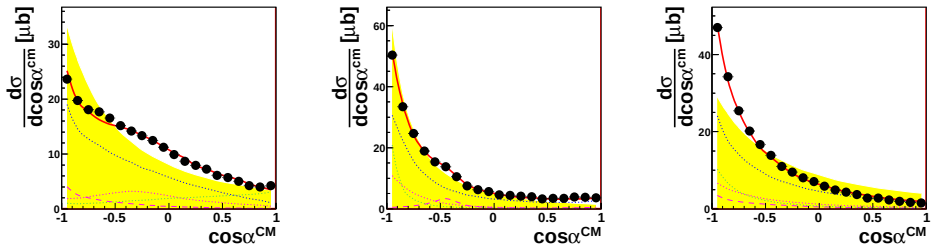
discribe the FSI in full, but they have to be adjusted (2σ-deviation, fig. 7.17(b)). The helicity-frame-angles – another possible projection of the Dalitz-plot – also agree nicely with the model-simulation. Also in this distribution, the contributing resonances are visible, as well as the final state interaction.



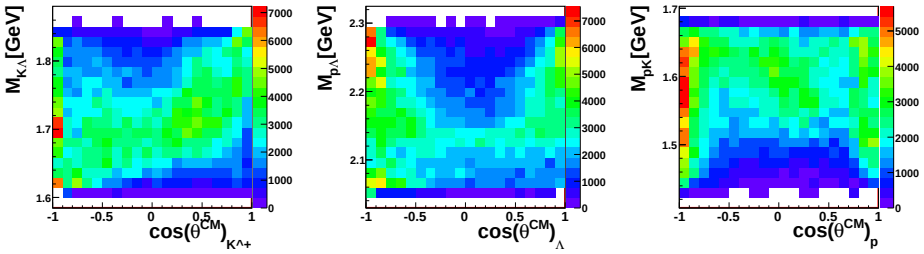
(a) $\alpha_{p\Lambda}^{(K^+\Lambda)}$ (b) $\alpha_{K^+p}^{(\Lambda p)}$ (c) $\alpha_{K^+\Lambda}^{(pK^+)}$

(d) $\alpha_{p}^{(\Lambda p)}$ (e) $\alpha_{K^+}^{(pK^+)}$ (f) $\alpha_{\Lambda}^{(K^+\Lambda)}$

Figure 7.18: Helicity- and Jackson-Frame-angles.



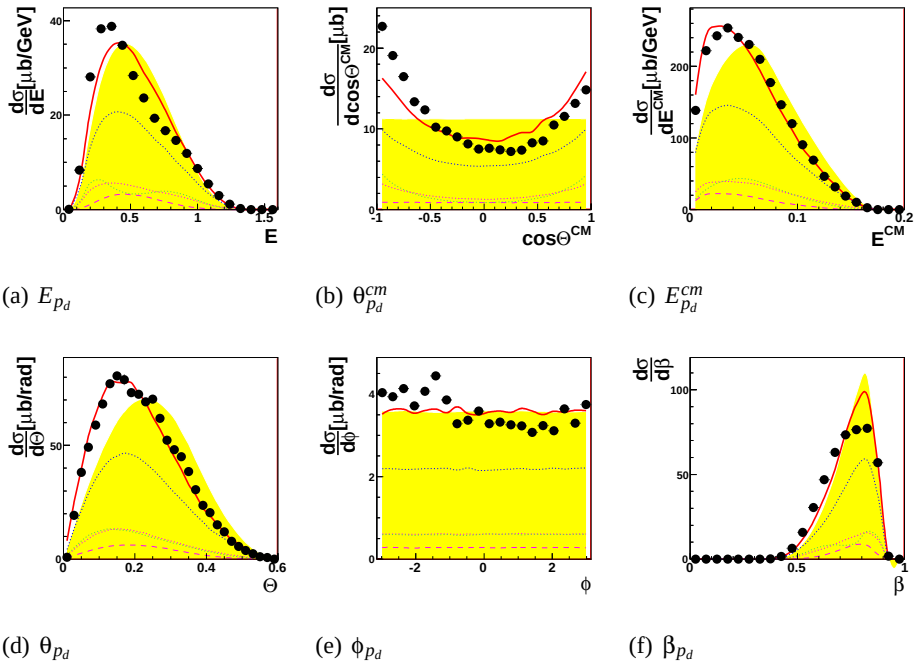
(a) $\alpha^{cm}_{K^+\Lambda}$ (b) $\alpha^{cm}_{\Lambda p}$ (c) $\alpha^{cm}_{pK^+}$



(d) $M_{K^+\Lambda}$ vs. $\cos\theta^{cm}_{K^+}$ (e) $M_{\Lambda p}$ vs. $\cos\theta^{cm}_{\Lambda}$ (f) M_{pK^+} vs. $\cos\theta^{cm}_{p}$

Figure 7.19: CM-angles. (a), (b) and (c) show the opening angles of each two particles. (d), (e) and (f) invariant mass of each two particles versus the center of mass angle of one of them.-

7.8.2 Λ-decay-particles

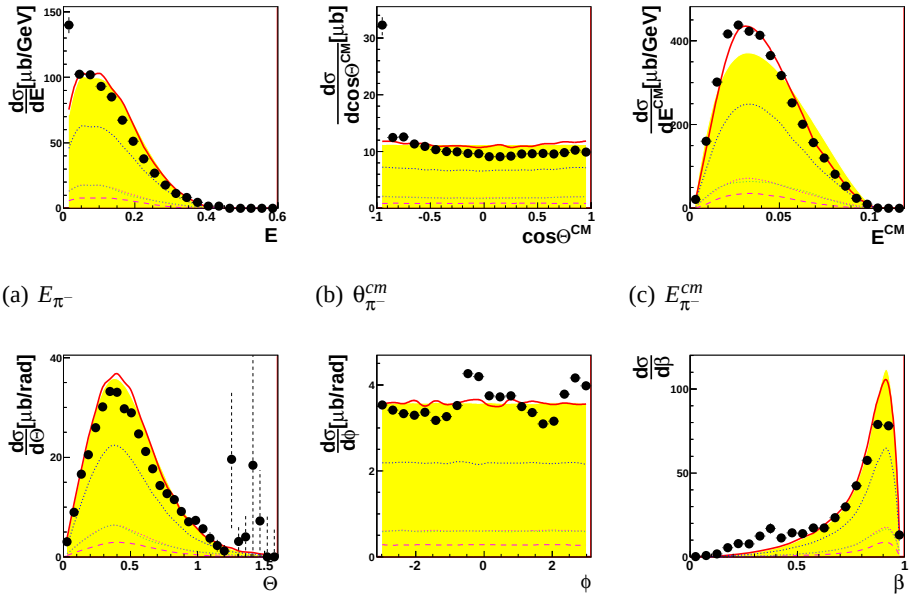


(a) E_{p_d} (b) $\theta^{cm}_{p_d}$ (c) $E^{cm}_{p_d}$

(d) θ_{p_d} (e) ϕ_{p_d} (f) β_{p_d}

Figure 7.20: Λ-decay particle: proton

The Λ that could be reconstructed decayed into proton and π^-. With the rather small excess momentum of 101 MeV/c ([46]), the proton, being almost as heavy as the Λ, carries almost the same momentum as the Λ it decayed off. This is the reason, that all effects, that the Λ-observables are subject to, are also affecting the proton-observables. Therefore the proton laboratory and center-of-mass distributions are rather well described.
The θ^{lab}-distributions of all measured particles show steps, at the transition from one stop sub-detector to another (Quirl-Ring at 0.17*rad*, Ring-Barrel at 0.43*rad*). This is caused by the fact, that the detector is not as well described by the simulation-software as expected, the features of the raw θ-spectra, both of Monte-Carlo-model-simulations and data, have qualitatively the same shape, but differ quantitatively. Here some inefficiencies of the real detector have not been correctly transformed to the virtual detector.



(a) E_{π^-} (b) $\theta^{cm}_{\pi^-}$ (c) $E^{cm}_{\pi^-}$

(d) θ_{π^-} (e) ϕ_{π^-} (f) β_{π^-}

Figure 7.21: Λ-decay particle: π^-

7.8.3 Σ-cusp

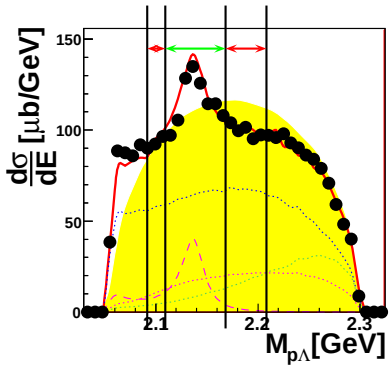
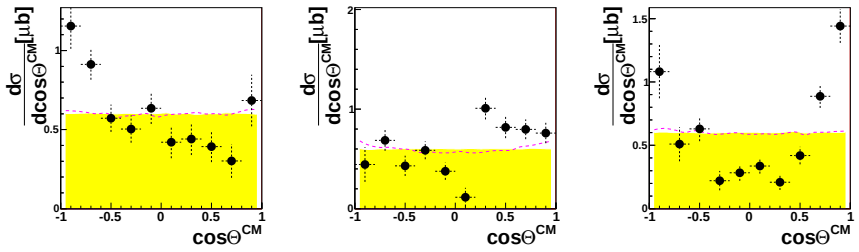


Figure 7.22: The invariant mass spectrum $M_{p\Lambda}$, including the lines on which the cusp-cut is made. Red arrows indicate the side-bands, greenthe main-band.

The Σ-production reactions $pp \rightarrow pK^+\Sigma^0$ and $pp \rightarrow nK^+\Sigma^+$ are suppressed in this data-sample.
Σ^0 decays electromagnetically with a mean life-time of $\tau = 7.4 * 10^{-20} s$ [46] into $\Lambda\gamma$ (BR = 100%). It gives the same signature in the COSY-TOF-detector as the reaction $pp \rightarrow pK^+\Lambda$, but can be removed using the χ^2-cut (sec. 7.7) and the cut on $\alpha_{P_{miss,PK^+},vee}$. The later one $pp \rightarrow nK^+\Sigma^+$ shouldn't even trigger the data aquisition and minor remnants are removed due to the applied cuts.
Nevertheless these Σ-production reactions are still visible in the measured data, but not as a final but as an intermediate state. With small relative momentum between nucleon and Σ, a conversion between Σ and Λ via hadronic interaction can occur (collision damping). Having now proton, kaon and Λ in the final state this event is no more distinguishable from the rest of the $pK^+\Lambda$-events, but it generates a narrow vertical structure in the Dalitz-plot and the projection on $M_{p\Lambda}$-axis. Here it is visible as a peak at Σ-threshold. This is called the Σ-cusp-effect.
The hadronic nature of the interaction gives rise to the rather broad width of this structure of 25 MeV.

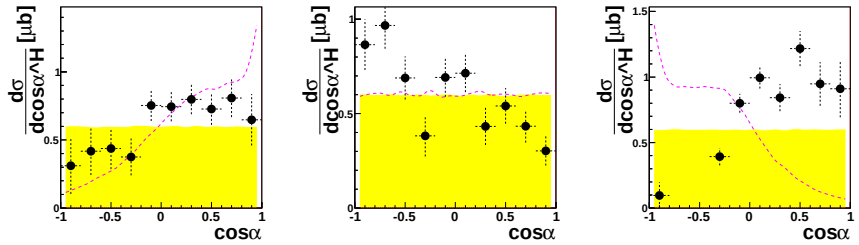
Having this large number of events for the reaction $pp \rightarrow pK^+\Lambda$, and the Σ-cusp showing up so nicely, we are in the position to calculate a total cross-section and generate differential distributions for the effect using side-band-subtraction.
For the side-band-subtraction, the events were separated according to their $M_{p\Lambda}$. The side-band-events (left and right) were filled in one set of histograms, the main- or cusp-band-events into another (see fig. 7.22). To gain the final set of histograms, the side-band-histograms were subtracted from the main-band-histograms[4].

[4]Including a factor of 0.899 to take the different area in model – without cusp – into account.



(a) $\cos\theta_p^{cm}$ (b) $\cos\theta_{K^+}^{cm}$ (c) $\cos\theta_\Lambda^{cm}$

Figure 7.23: This figure shows the $\cos\theta^{cm}$-distribution for proton, kaon an Λ-particles for the Σ-cusp-region.



(a) $\cos\alpha_{p\Lambda}^{(K^+\Lambda)}$ (b) $\cos\alpha_{K^+p}^{(\Lambda p)}$ (c) $\cos\alpha_{\Lambda K^+}^{(pK^+)}$

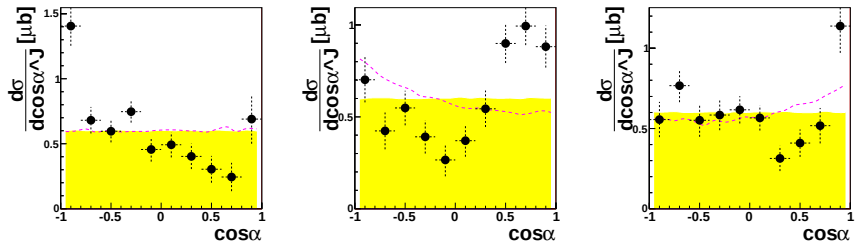
Figure 7.24: This figure shows the helicity-frame angles for proton, kaon an Λ-particles for the Σ-cusp-region.

	side-band left	cusp-band	side-band right
$M_{p\Lambda}$-ranges $[GeV]$	2.085 2.11	2.11 2.175	2.175 2.21
N_{events}^{Data}	4361	15213	7332
$N_{events}^{simulation}$	1326950	2804269	2335188
$N_{events}^{through-detector}$	39167	148147	93300

These numbers together with the total cross-section (eq. 7.15) result in a total cross-section for the cusp of:

$$\sigma_{cusp} = 1.2 \pm 0.2 \mu b \tag{7.17}$$

The following differential cross-sections have been produced: angular distribution in center of mass frame $\cos\theta^{cm}$ (fig. 7.23) for all three prompt particles, Jackson- and helicity-frame-angles respectively (figs. 7.25 and 7.24).
Contrary to the pictures showing the complete data-set, the cusp-region-graphs have considerably larger error-bars due to significantly smaller statistics. The $\cos\theta_\Lambda^{cm}$-distribution is definitely not phase-space-like, but is rather a smiling distribution. The kaon and proton center-off-mass angular distributions show a rather



(a) $\cos\alpha_p^{(\Lambda p)}$ (b) $\cos\alpha_{K^+}^{(pK^+)}$ (c) $\cos\alpha_\Lambda^{(K^+\Lambda)}$

Figure 7.25: This figure shows the Jackson-frame angles for proton, kaon an Λ-particles for the Σ-cusp-region.

flat behavior as expected due to relative s-wave close to threshold.
Fig. 7.25(a) gives access to the cusp-system, being the proton angle in the $p\Lambda$-system. This distribution is also definitely non-flat, but showing a decrease for forward going protons. Also the helicity-frame-angle between proton and kaon in the respective system shows such a behavior.
From these figures it is clear, that the toy-simulation used for these figures does not describe the data. What can be described are the effects of the Σ-cusp in the Dalitz-plots and projections of the full reaction, but not the extracted pure Σ-cusp spectra. Here detailed theoretical calculations are needed.

Chapter 8

Discussion

As expected, the results for the events of the reaction $pp \rightarrow pK^+\Lambda$ held no real surprises. The contributions of the N^*-resonances – as seen also in previous publications – are clearly visible and can be described with the used model. For this model, three N^*-resonances (N^*_{1650}, N^*_{1710}, N^*_{1720}) were added coherently including a meson-propagator due to σ-exchange. Along with the Final-State-Interaction, the data can be well described.
In contrary to the previous publications no non-resonant term was included into the model. N^*_{1720} is a very broad resonance. It is used instead of the non-resonant phase-space contribution.
Due to the excellent resolution a deviation in the invariant mass of proton and Λ ($M_{p\Lambda}$) – between 2.11 GeV and 2.175 GeV – became visible.This Σ-Cusp region can – at least in the Dalitz-plot and projections – be described using an additional Breit-Wigner-term, though no implications what so ever towards a resonance shall be made with this choice of function, for the Σ-cusp is a peak-like structure in this invariant mass spectrum.
Though having rather large error-bars, the figures in chapter 7.8.3 show already, that the distributions of the Σ-Cusp are neither phase-space-distributed nor can they be described with a Breit-Wigner-term alone.
The center-of-mass-frame angles for both proton and kaon are rather flat but the Λ-angle is definitely not. A meson-exchange is unlikely to happen, a simulation including meson-exchange could not describe the data. As for the helicity- and Jackson-frame angles, here also the distributions are not phase-space-like. Though the helicity-frame angle $\cos\theta^{(\Lambda p)}_{pK^+}$ can be reasonably well described by the Breit-Wigner-distribution, the other helicity-frame angles can not.
Providing now for the first time exclusive measurements showing a complete set of observables for the Σ-cusp-region, detailed theoretical calculations are now needed to describe those spectra appropriately.

Appendix A

Formulae

A.1 Pixels in Quirl and Ring

The pixels in Quirl and Ring detector – and later in the SQT – have a very special form due to the projected shape of the elements forming them ([32]).
The variables Δ_r and Δ_l are the point projection factors relative to the layer with the wedge shaped elements:

$$\begin{aligned} \Delta_r &= \frac{z_r}{z_w} \\ \Delta_l &= \frac{z_l}{z_w} \end{aligned}$$

In case of Quirl and Ring detector the bending b is defined as:

$$b = \frac{r_{max}}{\varphi_{max}}$$

The edges of the bent elements can be described as

$$r = \frac{r_{max}}{\varphi_{max}}(\varphi - \varphi_{offset}) = b(\varphi - \varphi_{offset}) = b\tilde{\varphi}$$

as for the straight element $\varphi = const$ is valid. The value φ_{offset} can be calculated for the lower-φ-edge for element number n_i in layer i with N_i total elements as:

$$\varphi_{n_i} = \varphi_{global} + 2\pi\frac{n_i}{N_i}$$

for the straight elements and

$$\varphi_{n_i} = \varphi_{global} - 2\pi\frac{n_i}{N_i}$$

for the bent ones. Lets define an intersection point $P_{intersection}$ in cylindrical coordinates. The z-component is quite easy to determine: P_z is the z-component of the

straight layer. Assuming the bending of both bent layers being of same absolute value, the intersection point of two edges E_{l,n_l} and E_{r,n_r} is:

$$\begin{aligned} P_{bb\{n_l,n_r\},\varphi} &= -\frac{(\Delta_l\varphi_{n_l}+\Delta_r\varphi_{n_r})}{\Delta_l+\Delta_r} \\ P_{bb\{n_l,n_r\}\rho} &= b\Delta_l\left(P_{bb\{n_l,n_r\}\varphi}-\varphi_{n_l}\right) \end{aligned}$$

A pixel out of two bent layers, has four such intersection points: $P_{bb\{n_l,n_r\}}$, $P_{bb\{n_l+1,n_r\}}$, $P_{bb\{n_l,n_r+1\}}$ and $P_{bb\{n_l+1,n_r+1\}}$. The intersection of a bent and a straight layer is simpler:

$$\begin{aligned} P_{bs\left\{n_{l/r},n_s\right\},\varphi} &= \varphi_{n_s} \\ P_{bs\left\{n_{l/r},n_s\right\},\rho} &= b\Delta_{l/r}\left(\varphi_{n_s}-\varphi_{n_{l/r}}\right) \end{aligned}$$

There are three categories of pixels:

Three corners, pointing right (condition $n_l+n_r-n_s>0$)
Here are the intersection points $P_{bb\{n_l,n_r\}}$, $P_{bs\{n_l,n_s+1\}}$, $P_{bs\{n_r,n_s+1\}}$. The area S of the pixel is:

$$\begin{aligned} S &= \frac{b^2}{2}\left[\frac{1}{3}\left(\Delta_l^2-\Delta_r^2\right)\varphi^3+\left(\Delta_l^2\varphi_{n_l}-\Delta_r^2\varphi_{n_r}\right)\varphi^2\right. \\ &\quad \left.+\left(\Delta_l^2\varphi_{n_l}^2-\Delta_r^2\varphi_{n_r}^2\right)\varphi\right]_{\varphi_{n_s+1}}^{P_{bb\{n_l,n_r\}\varphi}} \end{aligned} \tag{A.1}$$

The center point C in cylindrical coordinates is

$$\begin{aligned} c_\varphi &= \frac{b^2}{8S}\left[\left(\Delta_l^2-\Delta_r^2\right)\varphi^4+\frac{8}{3}\left(\Delta_l^2\varphi_{n_l}-\Delta_r^2\varphi_{n_r}\right)\varphi^3\right. \\ &\quad \left.+2\left(\Delta_l^2\varphi_{n_l}^2-\Delta_r^2\varphi_{n_r}^2\right)\varphi^2\right]_{\varphi_{n_s+1}}^{P_{bb\{n_l,n_r\}\varphi}} \end{aligned} \tag{A.2}$$

$$\begin{aligned} c_r &= -\frac{b^2}{3S}\left[\frac{1}{4}\left(\Delta_l^3+\Delta_r^3\right)\varphi^4+\left(\Delta_l^3\varphi_{n_l}+\Delta_r^3\varphi_{n_r}\right)\varphi^3\right. \\ &\quad \left.+\frac{3}{2}\left(\Delta_l^3\varphi_{n_l}^2+\Delta_r^3\varphi_{n_r}^2\right)\varphi^2+\left(\Delta_l^3\varphi_{n_l}^3+\Delta_r^3\varphi_{n_r}^3\right)\right]_{\varphi_{n_s+1}}^{P_{bb\{n_l,n_r\}\varphi}} \end{aligned} \tag{A.3}$$

Three corners, pointing left (condition $n_l+n_r-n_s<0$)
Here are the intersection points $P_{bb\{n_l+1,n_r+1\}}$, $P_{bs\{n_r+1,n_s\}}$, $P_{bs\{n_l+1,n_s\}}$. The area S of the pixel is:

$$\begin{aligned} S &= \frac{1}{2}b^2\left[\frac{1}{3}\left(\Delta_r^2-\Delta_l^2\right)\varphi^3+\left(\Delta_r^2\varphi_{n_r+1}-\Delta_l^2\varphi_{n_l+1}\right)\varphi^2\right. \\ &\quad \left.+\left(\Delta_r^2\varphi_{n_r+1}^2-\Delta_l^2\varphi_{n_l+1}^2\right)\varphi\right]_{\varphi_{n_s}}^{P_{bb\{n_l+1,n_r+1\}\varphi}} \end{aligned} \tag{A.4}$$

The center point C in cylindrical coordinates is

$$
\begin{aligned}
c_\varphi &= \frac{b^2}{8S}\left[\left(\Delta_r^2-\Delta_l^2\right)\varphi^4+\frac{8}{3}\left(\Delta_r^2\varphi_{n_r+1}-\Delta_l^2\varphi_{n_l+1}\right)\varphi^3\right. \qquad \text{(A.5)}\\
&\quad \left.+2\left(\Delta_r^2\varphi_{n_r+1}^2-\Delta_l^2\varphi_{n_l+1}^2\right)\varphi^2\right]_{\varphi_{n_s}}^{P_{bb\{n_l+1,n_r+1\}\varphi}}\\
c_r &= -\frac{b^2}{3S}\left[\frac{1}{4}\left(\Delta_r^3+\Delta_l^3\right)\varphi^4+\left(\Delta_r^3\varphi_{n_r+1}+\Delta_l^3\varphi_{n_l+1}\right)\varphi^3\right. \qquad \text{(A.6)}\\
&\quad \left.+\frac{3}{2}\left(\Delta_r^3\varphi_{n_r+1}^2+\Delta_l^3\varphi_{n_l+1}^2\right)\varphi^2+\left(\Delta_r^3\varphi_{n_r+1}^3+\Delta_l^3\varphi_{n_l+1}^3\right)\right]_{\varphi_{n_s}}^{P_{bb\{n_l+1,n_r+1\}\varphi}}
\end{aligned}
$$

Six corners (condition $n_l+n_r-n_s==0$)
Here we have the intersection points $P_{bb\{n_{l+1},n_r\}}$, $P_{bb\{n_l,n_{r+1}\}}$, $P_{bs\{n_l,n_s\}}$, $P_{bs\{n_{r+1},n_s\}}$, $P_{bs\{n_r,n_s\}}$, $P_{bs\{n_{l+1},n_s\}}$ Depending whether $P_{bb,\{n_l,n_r+1\},\varphi} > P_{bb,\{n_l+1,n_r\},\varphi}$, the limits for the integration change:

$$
\begin{aligned}
\phi_1 &= \varphi_{n_s}\\
\phi_2 &= \begin{cases} P_{bb,\{n_l,n_r+1\},\varphi} & P_{bb,\{n_l,n_r+1\},\varphi} > P_{bb,\{n_l+1,n_r\},\varphi}\\ P_{bb,\{n_l+1,n_r\},\varphi} & P_{bb,\{n_l,n_r+1\},\varphi} < P_{bb,\{n_l+1,n_r\},\varphi}\end{cases}\\
\phi_3 &= \begin{cases} P_{bb,\{n_l+1,n_r\},\varphi} & P_{bb,\{n_l,n_r+1\},\varphi} > P_{bb,\{n_l+1,n_r\},\varphi} = A\\ P_{bb,\{n_l,n_r+1\},\varphi} & P_{bb,\{n_l,n_r+1\},\varphi} < P_{bb,\{n_l+1,n_r\},\varphi} = B\end{cases}\\
\phi_4 &= \varphi_{n_s+1}
\end{aligned}
$$

$$
\begin{aligned}
S &= \frac{b^2}{2}\left(\left[\frac{1}{3}\left(\Delta_l^2-\Delta_r^2\right)\varphi^3+\left(\Delta_l^2\varphi_{n_l}-\Delta_r^2\varphi_{n_r}\right)\varphi^2+\left(\Delta_l^2\varphi_{n_l}^2-\Delta_r^2\varphi_{n_r}^2\right)\varphi\right]_{\phi_1}^{\phi_2}\right.\\
&\quad +\left[\frac{1}{3}\left(\Delta_r^2-\Delta_l^2\right)\varphi^3+\left(\Delta_r^2\varphi_{n_r+1}-\Delta_l^2\varphi_{n_l+1}\right)\varphi^2+\left(\Delta_r^2\varphi_{n_r+1}^2-\Delta_l^2\varphi_{n_l+1}^2\right)\varphi\right]_{\phi 3}^{\phi_4}\\
&\quad \left.+\begin{cases}\Delta_l^2\left[\left(\varphi_{n_l}-\varphi_{n_l+1}\right)\varphi^2+\left(\varphi_{n_l}^2-\varphi_{n_l+1}^2\right)\varphi\right]_{\phi_2}^{\phi_3} & ,A\\ \Delta_r^2\left[\left(\varphi_{n_r}-\varphi_{n_r+1}\right)\varphi^2+\left(\varphi_{n_r}^2-\varphi_{n_r+1}^2\right)\varphi\right]_{\phi_2}^{\phi_3} & ,B\end{cases}\right.
\end{aligned}
$$

$$
\begin{aligned}
c_\varphi &= \frac{b^2}{2S}\left(\left[\frac{1}{4}\left(\Delta_l^2-\Delta_r^2\right)\varphi^4+\frac{2}{3}\left(\Delta_l^2\varphi_{n_l}-\Delta_r^2\varphi_{n_r}\right)\varphi^3+\frac{1}{2}\left(\Delta_l^2\varphi_{n_l}^2-\Delta_r^2\varphi_{n_r}^2\right)\varphi^2\right]_{\phi_1}^{\phi_2}\right.\\
&\quad +\left[\frac{1}{4}\left(\Delta_l^2-\Delta_r^2\right)\varphi^4+\frac{2}{3}\left(\Delta_l^2\varphi_{n_l+1}-\Delta_r^2\varphi_{r+1}\right)\varphi^3+\frac{1}{2}\left(\Delta_l^2\varphi_{n_l+1}^2-\Delta_r^2\varphi_{n_r+1}^2\right)\varphi^2\right]_{\phi_3}^{\phi_4}\\
&\quad \left.+\begin{cases}\left[\Delta_l^2\left(\frac{2}{3}\left(\varphi_{n_l}-\varphi_{n_l+1}\right)\varphi^3+\frac{1}{2}\left(\varphi_{n_l}^2-\varphi_{n_l+1}^2\right)\varphi^2\right)\right]_{\phi_2}^{\phi_3} & ,A\\ \left[\Delta_r^2\left(\frac{2}{3}\left(\varphi_{n_r+1}-\varphi_{n_r}\right)\varphi^3+\frac{1}{2}\left(\varphi_{n_r+1}^2-\varphi_{n_r}^2\right)\varphi^2\right)\right]_{\phi_2}^{\phi_3} & ,B\end{cases}\right) \qquad \text{(A.7)}
\end{aligned}
$$

$$
\begin{aligned}
c_r \;=\; & \frac{b^3}{3S}\left(\left[\frac{1}{4}\left(\Delta_l^3+\Delta_r^3\right)\varphi^4+\left(\Delta_l^3\varphi_{n_l}+\Delta_r^3\varphi_{n_r}\right)\varphi^3\right.\right. \\
& \left.+\frac{3}{2}\left(\Delta_l^3\varphi_{n_l}^2+\Delta_r^3\varphi_{n_r}^2\right)\varphi^2+\left(\Delta_l^3\varphi_{n_l}^3+\Delta_r^3\varphi_{n_r}^3\right)\varphi\right]_{\phi_1}^{\phi_2} \\
& -\left[\frac{1}{4}\left(\Delta_r^3+\Delta_l^3\right)\varphi^4+\left(\Delta_r^3\varphi_{n_r+1}+\Delta_l^3\varphi_{n_l+1}\right)\varphi^3\right. \\
& \left.+\frac{3}{2}\left(\Delta_r^3\varphi_{n_r+1}^2+\Delta_l^3\varphi_{n_l+1}^2\right)\varphi^2+\left(\Delta_r^3\varphi_{n_r+1}^3+\Delta_l^3\varphi_{n_l+1}^3\right)\varphi\right]_{\phi_3}^{\phi_4} \\
& \left\{\begin{array}{ll}
+\Delta_l^3\left[\left(\varphi_{n_l}-\varphi_{n_l+1}\right)\varphi^3+\frac{3}{2}\left(\varphi_{n_l}^2-\varphi_{n_l+1}^2\right)\varphi^2+\left(\varphi_{n_l}^3-\varphi_{n_l+1}^3\right)\varphi\right]_{\phi_2}^{\phi_3} & ,A \\
-\Delta_r^3\left[\left(\varphi_{n_r+1}-\varphi_{n_r}\right)\varphi^3+\frac{3}{2}\left(\varphi_{n_r+1}^2-\varphi_{n_r}^2\right)\varphi^2+\left(\varphi_{n_r+1}^3-\varphi_{n_r}^3\right)\varphi\right]_{\phi_2}^{\phi_3} & ,B
\end{array}\right)
\end{aligned}
\tag{A.8}
$$

A.2 Bethe-Bloch-Formula

When a particle traverses matter it interacts with the matter leaving some (or all) of its kinetic energy in the matter. This basic feature allows us to identify the particles. In the case of heavy charged particles, the only measurable particles in the case of COSY-TOF, the interaction of the particle in matter can be characterized by:

1. Inelastic scattering on electrons of the electron shell of atoms.
2. Elastic scattering on nuclei.
3. Nuclear reactions .
4. Bremsstrahlung.
5. Čherenkov radiation.

The reactions 3, 4 and 5 happen quite seldom in nature and are therefor neglected also the elastic scattering of the particles on the nucleus can be neglected due to the, compared to the inelastic scattering on shell-electrons, small cross-section and energy loss.
The remaining process of inelastic scattering on electrons of the electron shell of atoms has been described by H. BETHE and F. BLOCH [15]:

$$
-\frac{dE}{dx}=\frac{4\pi}{m_e c^2}\frac{nz^2}{\beta^2}\left(\frac{e^2}{4\pi\varepsilon_0}\right)^2\left[\ln\left(\frac{2m_e c^2\beta^2}{I\left(1-\beta^2\right)}\right)-\beta^2\right] \tag{A.9}
$$

with the following definitions

ze ... particle charge
e ... electron charge
ε_0 ... vacuum permissivity
c ... speed of light
$n = \frac{N_A Z \rho}{A}$ ... electron density
$\beta = \frac{v}{c}$ ... particle speed
m_e ... electron mass
$I \approx (10eV)\,Z$ ... mean excitation potential

A.3 Invariant mass / missing mass

A.3.1 Invariant mass

In the analysis of a reaction it is important to consider the sum of the 4-momentum vectors of all or part of the participating particles. An important observable is the invariant mass, the mass of the summed up 4-momentum-vectors.
The invariant mass of n particles is calculated via:

$$M_{inv} = \sqrt{\left(\sum_{i=1}^{n} \frac{E_i}{c^2}\right)^2 - \left(\sum_{i=1}^{n} \frac{p_i}{c}\right)^2} \tag{A.10}$$

This reduces in the two particle case to:

$$M_{inv} = \sqrt{\left(\frac{E_1 + E_2}{c^2}\right)^2 - \left(\frac{p_1 + p_2}{c}\right)^2} \tag{A.11}$$

Using energy- and momentum-conservation the number of necessary input components reduces in the two particle case to five.

Invariant mass for Λ-decay

In the case of Λ-reconstruction, two of these components are the angles of the decay-products to the Λ. Due to the big mass difference of the decay particles (proton and π^-), the masses of the decay particles are fixed as well. The track with the larger opening angle is always the π^-, the one with the smaller opening angle the proton. With these considerations there remains one value to be determined.

In the analysis of other groups of the collaboration, the Λ-momentum is used, calculated out of the incoming momenta and the two prompt particles using the missing mass (eqn. A.14) formula. This leads to the following formula:

$$
\begin{aligned}
M^2_{p\pi} = & \qquad (A.12)\\
& m^2_\pi + m^2_p + p^2_\Lambda \left(\frac{2\tan^2\theta_p \tan^2\theta_\pi + \tan^2\theta_p + \tan^2\theta_\pi}{(\tan\theta_p + \tan\theta_\pi)^2} - 1 \right) \\
& + 2\sqrt{\left\{ m^2_\pi m^2_p + p^2_\Lambda \left(\frac{\tan^2\theta_p \tan^2\theta_\pi \left(m^2_\pi + m^2_p\right) + m^2_\pi \tan^2\theta_\pi + m^2_p \tan^2\theta_p}{(\tan\theta_p + \tan\theta_\pi)^2} \right) \right.} \\
& \left. \overline{+ p^4_\Lambda \left(\frac{\tan^2\theta_p \tan^2\theta_\pi (\tan^2\theta_p \tan^2\theta_\pi + \tan^2\theta_p + \tan^2\theta_\pi + 1)}{(\tan\theta_p + \tan\theta_\pi)^4} \right) \right\}}
\end{aligned}
$$

θ_p is the angle between the proton- and the Λ-direction, θ_π is the angle between the π^-- and the Λ-direction, m_p and m_{pi} are the masses of proton and π^- and p_Λ the calculated momentum of the Λ-particle.
This method is biased by the primary particles of the reaction. It is preferred to have a method to calculate the invariant mass only from the measured values of the decay particles.
Fortunately the mass of the proton is quite close to the mass of the Λ and so the direction- and velocity-change of the baryon in the decay is not too large. Also the distance of the Start-detector to the range where the Λ-decay can be reconstructed is small compared to the complete distance to the stop-detectors. The velocity of the proton can be determined by using the stop-timing of the hit stop-detector and the global start-timing, here in this case the mean start of the prompt proton and the prompt K^+.
This gives the following formula:

$$
\begin{aligned}
M^2_{p\pi} &= m^2_p + m^2_\pi \qquad (A.13)\\
&+ \left(m_p \frac{\beta_p}{\sqrt{1-\beta^2_p}} \right)^2 \left(1 + \sin^2\theta_p + \cos^2\theta_p \frac{\tan^2\theta_p}{\tan^2\theta_\pi} - \cos^2\theta_p \frac{(\tan\theta_p + \tan\theta_\pi)^2}{\tan^2\theta_\pi} \right) \\
&+ 2\sqrt{m^2_p m^2_\pi + \left(m_p \frac{\beta_p}{\sqrt{1-\beta^2_p}} \right)^2 m^2_\pi} \\
&\overline{+ \left(m_p \frac{\beta_p}{\sqrt{1-\beta^2_p}} \right)^2 \left(\sin^2\theta_p + \cos^2\theta_p \frac{\tan^2\theta_p}{\tan^2\theta_\pi} \right) \left(m^2_p + \left(m_p \frac{\beta_p}{\sqrt{1-\beta^2_p}} \right)^2 \right)}
\end{aligned}
$$

With β_p being the velocity of the proton.

A.3.2 Missing mass

The missing mass is a version of the invariant mass, taking incoming and outgoing particles, apart from the sign, alike:

$$M_{miss} = \sqrt{\left(\sum_{i}^{ingoing} E_i - \sum_{j}^{outgoing} E_j\right)^2 - \left(\sum_{i}^{ingoing} \vec{p}_i - \sum_{j}^{outgoing} \vec{p}_j\right)^2} \tag{A.14}$$

Usually the square of the missing mass is shown, it is not restricted to positive values and can give a hint on which side of the reaction the mass is missing:

$$M_{miss}^2 = \left(\sum_{i}^{ingoing} E_i - \sum_{j}^{outgoing} E_j\right)^2 - \left(\sum_{i}^{ingoing} \vec{p}_i - \sum_{j}^{outgoing} \vec{p}_j\right)^2$$

It can be calculated for the complete reaction, where it incorporates conservation of energy and momentum and should be zero. Using only some of the reaction particles, it can give important information about reaction processes, produced resonances etc.

A.3.3 Dalitz-plot

In a three-particle decay one can plot a so called Dalitz-plot, a scatterplot representing the complete kinematic of the reaction. One of the axes is the square of the invariant mass (A.10) of e.g. particle A and B while the other axis is the square invariant mass of particle B and C.
In a pure phase-space distribution the Dalitz-plot is flat, but a resonance being produced in the reaction will show up in the Dalitz-plot, if the statistic permits sometimes even more pronounced than in the invariant mass plots themselves. A resonance decaying into two of the three outgoing particles will show up as a line, while a resonance involving all three will show up as a point.

A.4 Breit-Wigner-Formula for Resonances

The description of the shape of resonances, e.g. showing up in (invariant) mass- or energy-spectra is given by the Breit-Wigner-formula [14]

$$f_{resonance} \sim \sqrt{\frac{2j+1}{(2s_1+1)(2s_2+1)}} \frac{-\sqrt{s}\Gamma_{(s)}}{s - M^2 + i\sqrt{s}\Gamma} \tag{A.15}$$

As for the nomenclature: $\sqrt{s}$ is the invariant mass (see. A.10) of the decay particles, M the mass of the resonance. Further-on: Γ is the width of the resonance at pole-position, Γ_i and Γ_f partial widths, with $\sqrt{\Gamma_i^0 * \Gamma_f^0}$ the strength of the resonance. k is the meson-momentum of the outgoing (decay) vertex in the resonance rest-frame,

q is the momentum at the ingoing (production) vertex also in the resonance rest-frame. There are three things to be taken into account:
First, the width Γ_s in the numerator is the product of the partial widths of the production and the decay of the resonance and therefore momentum dependent.
Second is the production vertex, where the exchanged meson and the spin of the resonance determine the operator that has to be applied as a factor. σ-exchange contributes only with s-wave, resulting in a constant factor. π-exchange is p-wave (l=1):

$$operator = \begin{cases} const. & \sigma - exchange \\ \begin{cases} \vec{\sigma}\cdot\vec{q} & J=\frac{1}{2} \\ \vec{S}\cdot\vec{q} & J=\frac{3}{2} \end{cases} & \pi - exchange \end{cases}$$

Third is the decay vertex, where – as for the production vertex – spin and parity of the resonance as well as the angular momentum of the decay products define the necessary operator.

$$operator = \begin{cases} const. & s-wave \\ \vec{\sigma}\cdot\vec{k} & p-wave,\ J=\frac{1}{2} \\ \vec{S}\cdot\vec{k} & p-wave,\ J=\frac{3}{2} \end{cases}$$

These operators and the momentum dependent width can be expanded to the following formula:

$$\begin{aligned} \Gamma_{f,i} &= \gamma^0_{f,i}\left(\frac{k}{k_r}\right)^{2l_{f,i}+1}\left(\frac{k_r^2+\delta^2}{k^2+\delta^2}\right)^{l_{f,i}+1} \\ c_{(k,l_f,q,l_i)} &= \sqrt{\frac{\Gamma_i\Gamma_f}{qk}} \\ &= \sqrt{\frac{\Gamma_i^0\Gamma_f^0}{q^{N^*}k^{N^*}}}\sqrt{\left(\frac{k}{k_R}\right)^{2l_f+1}\left(\frac{k_R^2+\delta^2}{k^2+\delta^2}\right)^{l_f+1}\left(\frac{q}{k_R}\right)^{2l_i+1}\left(\frac{k_R^2+\delta^2}{q^2+\delta^2}\right)^{l_i+1}} \end{aligned} \tag{A.16}$$

with the angular momentum both at the production (l_i) vertex and at the decay (l_f) vertex.
Now the formula has the following form:

$$f_{resonance} \sim \sqrt{\frac{2j+1}{(2s_1+1)(2s_2+1)}}\, c_{(k,l_f,q,l_i)}\frac{-\sqrt{s}}{s-M^2+i\sqrt{s}\Gamma} \tag{A.17}$$

If more than one resonance is excited, the resonance amplitudes have to be added, including a phase (multiply a factor of $e^{i\phi}$).

A.5 Frames

Particles, directions, velocities, energies and momenta are measured in the lab system. But there exist other inertial systems, that can provide interesting features for i.e. angular distributions.

A.5.1 CM-Frame

The Center of Momentum System (CMS) or overall center of momentum system, is the rest frame of the center of momentum. Here both beam and target particle have the same absolute momentum in different directions:

$$\vec{p}_{beam} = -\vec{p}_{target} \tag{A.18}$$

If the entrance channel is symmetric (here 2 protons), the exit channel has to give symmetric angular distributions in CMS around 90°.

A.5.2 Subsystems

The Jackson Frame as well as the Helicity Frame are not overall center of mass systems but correspond to different subsystems. For three-body decays of the type $ab \rightarrow 123$, one can define three different subsystems: $(2,3)$, $(3,1)$, $(1,2)$.

Jackson-Frame

In one of these sub-system-frames (i.e. $(2,3)$), the Jackson Angle is defined as the angle between particle 3 and the beam direction in this sub-system-frame: $\angle(\vec{p}_b, \vec{p}_3)$.
This angle connects the exit (3) and the entrance (beam) channel and gives information not accessible by means of the Dalitz plot. Define the Jackson Angle as:

$$\theta_{b3}^{R23} = \angle(\vec{p}_b, \vec{p}_3)^{R23} \tag{A.19}$$

where the superscript denotes the subsystem, the subscript indicates the angle of particle 3 with respect to the beam *b*.

Helicity-Frame

For two-particle decays such as the Λ-decay, the helicity frame is defined as the Λ-rest frame. The angle here between the decay particles and the Λ-direction gives a hint to the Λ-polarization.

$$\theta_{decay\Lambda}^{\Lambda} = \angle(\vec{p}_{decay}, \vec{p}_{\Lambda}) \tag{A.20}$$

But for three-body-decays we can define subsystems and a helicity angle:

$$\theta_{13}^{R23} = \angle(\vec{p}_1, \vec{p}_3)^{R23} \tag{A.21}$$

A.6 Reactions

A.6.1 2-particle reactions

For any reaction of the pattern $AB \rightarrow CD$ there are tight restrictions on the kinematics of the reaction.
For simplicity let's consider the lab-frame with A being the beam-particle (moving along the z-axis), B the target. Here the following equation is valid:

$$|\phi_C - \phi_D| = 180^\circ \tag{A.22}$$

The momentum of particles C and D can be calculated from the θ-angles and the sum-momentum of particles A and B:

$$\begin{aligned} P_t &= P_{A+B}\frac{\tan\theta_C * \tan\theta_D}{\tan\theta_C + \tan\theta_D} \\ P_{l,C/D} &= P_{A+B}\frac{\tan\theta_{D/C}}{\tan\theta_C + \tan\theta_D} \\ P_{C/D} &= \sqrt{P_t^2 + P_{l,C/D}^2} \\ &= P_{A+B}\frac{\tan\theta_{D/C}}{\tan\theta_C + \tan\theta_D}\sqrt{\tan^2\theta_{C/D} + 1} \end{aligned} \tag{A.23}$$

The longitudinal momentum P_l is parallel, the transverse momentum P_t is perpendicular to the sum-momentum of A and B. This also holds if A and B are not two particles but one that decays into two particles (eg. Λ-decay).

Elastic scattering

The elastic scattering $pp \rightarrow pp$ has the additional restriction that the particles C and D have the same mass. This provides an additional restriction on the lab-angles:

$$\gamma = \frac{1}{\sqrt{1 - \beta_{beam}}} \tag{A.24}$$

apart from the coplanarity condition A.22

Deuteron-π^+

The reaction $pp \rightarrow d\pi^+$ is another example of a binary reaction. Here also the equations A.22 and A.23 hold. Contrary to the elastic scattering of protons, for the $d\pi^+$-reaction the identity of the particles has to be determined using the $P_t - P_l$-plot, where the transverse momentum is plotted versus the longitudinal momentum. In this plot two bands are visible forming ellipses; one for the deuteron, one for the pion. Only in the small overlapping region the distinction of the two masses is impossible.

Appendix B

KinFit

Measured values of particles, like directions (in Cartesian, spheric or cylindric coordinates) and energy, momentum or velocity always are smeared due to limited resolution of the detector. This results in the fact, that –for measured values– momentum and energy is not necessarily conserved and that the values for different observables may exceed beyond the kinematically allowed region. To shift these values back to the kinematically allowed region, a kinematical fit (abbr. kinFit) is performed taking the errors of the measurement into account.
The kinematic fit procedure used for this work was adopted from the kinFit used by the WASA-Collaboration ([3]). The value to be minimized is the deviation from energy and momentum conservation:

$$\begin{aligned} F &= (\Delta P_x)^2 + (\Delta P_y)^2 + (\Delta P_z)^2 + (\Delta E)^2 = \left| \begin{pmatrix} \Delta P_x \\ \Delta P_y \\ \Delta P_z \\ \Delta E \end{pmatrix} \right|^2 \\ &= \left| \begin{pmatrix} \sum^j P_x^j \\ \sum^j P_y^j \\ \sum^j P_z^j \\ \sum^j E^j \end{pmatrix} - \begin{pmatrix} P_x^i \\ P_y^i \\ P_z^i \\ E^i \end{pmatrix} \right|^2 = minimal \end{aligned} \tag{B.1}$$

For minimalisation the gradient method using Lagrangian multipliers is used. Let's make some definitions: Each Particle gives 3 variables (mass counts as fixed). These variables can be either measured, unmeasured or fixed (useful for beam and target).

Variables Let's have m measured variables and u unmeasured variables:

$$X = (x_0, x_1, \ldots, x_m) \tag{B.2}$$

$$U = (x_{m+1}, x_{m+1}, \ldots, x_{m+u}) \tag{B.3}$$

Transform to n Particles (transformation see Table B.1):

$$P_i = \begin{pmatrix} P_{x,i}(X_j,X_k,X_l) \\ P_{y,i}(X_j,X_k,X_l) \\ P_{z,i}(X_j,X_k,X_l) \\ E_{i}(X_j,X_k,X_l) \end{pmatrix} \tag{B.4}$$

Define an error Matrix G_M for the measured variables as $diag\left(\Delta x_i^2\right)$

$$G_M = \begin{pmatrix} \frac{1}{\Delta x_0^2} & 0 & \dots & 0 \\ 0 & \frac{1}{\Delta x_1^2} & \dots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \dots & \frac{1}{\Delta x_m^2} \end{pmatrix} \tag{B.5}$$

Do a Taylor-expansion:

$$F_{last} = \begin{pmatrix} \sum P_{x,i} \\ \sum P_{y,i} \\ \sum P_{z,i} \\ \sum E_i \end{pmatrix} + \sum_{j=1}^{m} \varepsilon_j \frac{\delta F_\mu}{\delta x_j} = F + \sum_{j=1}^{m} \varepsilon_j \frac{\delta F_\mu}{\delta x_j} = F + B\varepsilon \tag{B.6}$$

using derivative matrix B_M (see Tables B.2, B.3, B.4, B.5, B.6, B.7, B.8). Since variable x_j only contributes to the calculation of the 4-momentum of particle κ, the expression for the derivation matrix reduces somehow.

$$B_M = \frac{\delta F_\mu}{\delta x_j} = \begin{pmatrix} \frac{\delta \sum_i P_{i,x}}{\delta x_0} & \frac{\delta \sum_i P_{i,x}}{\delta x_1} & \dots & \frac{\delta \sum_i P_{i,x}}{\delta x_m} \\ \frac{\delta \sum_i P_{i,y}}{\delta x_0} & \frac{\delta \sum_i P_{i,y}}{\delta x_1} & \dots & \frac{\delta \sum_i P_{i,y}}{\delta x_m} \\ \frac{\delta \sum_i P_{i,z}}{\delta x_0} & \frac{\delta \sum_i P_{i,z}}{\delta x_1} & \dots & \frac{\delta \sum_i P_{i,z}}{\delta x_m} \\ \frac{\delta \sum_i E_i}{\delta x_0} & \frac{\delta \sum_i E_i}{\delta x_1} & \dots & \frac{\delta \sum_i E_i}{\delta x_m} \end{pmatrix} = \left(\frac{\delta \sum_l P_{l,i}}{\delta x_j}\right)_{i,j} = \left(\frac{\delta P_{\kappa,i}}{\delta x_j}\right)_{i,j} \tag{B.7}$$

No unmeasured variables Define Lagrangian multiplier λ, variable modificator ε and χ^2

$$\lambda = \left(BG^{-1}B^T\right)^{-1} F_{last} \tag{B.8}$$

$$\varepsilon = -G^{-1}\left(B^T\lambda\right) \tag{B.9}$$

$$\chi^2 = \varepsilon^T G \varepsilon \tag{B.10}$$

With unmeasured variables Define for m measured variables: error matrix G_M, derivative matrix B_M, modificator ε_M.

Define for u unmeasured variables: weighting matrix G_U, derivative matrix B_U, modificator ε_U.

$$G_B = \left(B_M G_M^{-1} B_M^T\right)^{-1} \tag{B.11}$$

$$\varepsilon_U = -\left(G_U + B_U^T G_B B_U\right)^{-1} B_U^T G_B F_{last} \tag{B.12}$$

$$\lambda = G_B\left(B_U \varepsilon_U + F_{last}\right) \tag{B.13}$$

$$\varepsilon_M = -G_M^{-1}\left(B_M^T \lambda\right) \tag{B.14}$$

$$\chi^2 = \varepsilon_M^T G_M \varepsilon_M + \varepsilon_U^T G_U \varepsilon_U + 2\lambda\left(F_{last} + B_M \varepsilon_M + B_U \varepsilon_U\right) \tag{B.15}$$

Representations It is not necessary to do the fit in the simple P_x, P_y, P_z, representation. Other representations of the particles may reflect the detector-setup more precisely, like in the case of COSY-TOF β, θ, ϕ. To take account for the different representations different formulae have to be applied to calculate the 4-vectors out of the variables (see table B.1) and the derivative matrices B_M and B_U (see tables B.2, B.3, B.4, B.5, B.6, B.7, B.8).
It is interesting to examine the dependence of the outcome of the kinFit with respect to the representation used. To have comparable errors as input, these also have to be transformed.

Representation	ID	P	P_x	P_y	P_z	E
P_x, P_y, P_z	1	$\sqrt{P_x^2 + P_y^2 + P_z^2}$	P_x	P_y	P_z	$\sqrt{P^2 + m^2}$
E_{kin}, θ, ϕ	2	$\sqrt{E_{kin}\left(E_{kin} + 2m\right)}$	$P\sin\theta\cos\phi$	$P\sin\theta\sin\phi$	$P\cos\theta$	$E_{kin} + m$
β, θ, ϕ	3	$\frac{m\beta}{\sqrt{1-\beta^2}}$	$P\sin\theta\cos\phi$	$P\sin\theta\sin\phi$	$P\cos\theta$	$\frac{m}{\sqrt{1-\beta^2}}$
P, θ, ϕ	4	P	$P\sin\theta\cos\phi$	$P\sin\theta\sin\phi$	$P\cos\theta$	$\sqrt{P^2 + m^2}$
$E_{kin}, \delta x, \delta y$	5	$\sqrt{E_{kin}\left(E_{kin} + 2m\right)}$	$\frac{P\delta x}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P\delta y}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P}{\sqrt{\delta x^2+\delta y^2+1}}$	$E_{kin} + m$
$P, \delta x, \delta y$	6	P	$\frac{P\delta x}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P\delta y}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P}{\sqrt{\delta x^2+\delta y^2+1}}$	$\sqrt{P^2 + m^2}$
$\beta, \delta x, \delta y$	7	$\frac{m\beta}{\sqrt{1-\beta^2}}$	$\frac{P\delta x}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P\delta y}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{P}{\sqrt{\delta x^2+\delta y^2+1}}$	$\frac{m}{\sqrt{1-\beta^2}}$

Table B.1: Transform measured properties into 4D-momentum.

	P_x	P_y	P_z	E
$\frac{\delta}{\delta P_x}$	1	0	0	$\frac{P_x}{\sqrt{P^2+m^2}}$
$\frac{\delta}{\delta P_y}$	0	1	0	$\frac{P_y}{\sqrt{P^2+m^2}}$
$\frac{\delta}{\delta P_z}$	0	0	1	$\frac{P_z}{\sqrt{P^2+m^2}}$

Table B.2: Derivatives for representation 1: P_x, P_y, P_z

δ	P_x	P_y	P_z	E
$\frac{\delta}{\delta E_{kin}}$	$\frac{E_{kin}+m}{P}\sin\theta\cos\phi$	$\frac{E_{kin}+m}{P}\sin\theta\sin\phi$	$\frac{E_{kin}+m}{P}\cos\theta$	1
$\frac{\delta}{\delta\theta}$	$P\cos\theta\cos\phi$	$P\cos\theta\sin\phi$	$-P\sin\theta$	0
$\frac{\delta}{\delta\phi}$	$-P\sin\theta\sin\phi$	$P\sin\theta\cos\phi$	0	0

Table B.3: Derivatives for representation 2: E_{kin},θ,ϕ, $P=\sqrt{E_{kin}\left(E_{kin}+2m\right)}$

	P_x	P_y	P_z	E
$\frac{\delta}{\delta\beta}$	$\frac{\delta P}{\delta\beta}\sin\theta\cos\phi$	$\frac{\delta P}{\delta\beta}\sin\theta\sin\phi$	$\frac{\delta P}{\delta\beta}\cos\theta$	$\frac{m\beta}{(1-\beta^2)^{\frac{3}{2}}}$
$\frac{\delta}{\delta\theta}$	$P\cos\theta\cos\phi$	$P\cos\theta\sin\phi$	$-P\sin\theta$	0
$\frac{\delta}{\delta\phi}$	$-P\sin\theta\sin\phi$	$P\sin\theta\cos\phi$	0	0

Table B.4: Derivatives for representation 3: β,θ,ϕ, $p=\frac{m\beta}{\sqrt{1-\beta^2}}=m\beta\gamma$, $\frac{\delta P}{\delta\beta}=m\frac{1+\frac{\beta^2}{1-\beta^2}}{\sqrt{1-\beta^2}}=m\gamma\left(1+\beta^2\gamma^2\right)$

Errors

$$\delta x = \tan\theta\cos\phi \tag{B.16}$$

$$\delta y = \tan\theta\sin\phi \tag{B.17}$$

$$\theta = \arccos\frac{1}{\sqrt{\delta x^2+\delta y^2+1}} \tag{B.18}$$

$$\phi = \arccos\frac{\delta x}{\sqrt{\delta x^2+\delta y^2}} \tag{B.19}$$

	P_x	P_y	P_z	E
$\frac{\delta}{\delta P}$	$\sin\theta\cos\phi$	$\sin\theta\sin\phi$	$\cos\theta$	$\frac{P}{\sqrt{P^2+m^2}}$
$\frac{\delta}{\delta\theta}$	$P\cos\theta\cos\phi$	$P\cos\theta\sin\phi$	$-P\sin\theta$	0
$\frac{\delta}{\delta\phi}$	$-P\sin\theta\sin\phi$	$P\sin\theta\cos\phi$	0	0

Table B.5: Derivatives for representation 4: P,θ,ϕ

	P_x	P_y	P_z	E
$\frac{\delta}{\delta E_{kin}}$	$\frac{E_{kin}+m}{P}\frac{dx}{l}$	$\frac{E_{kin}+m}{P}\frac{dy}{l}$	$\frac{E_{kin}+m}{P}\frac{1}{l}$	1
$\frac{\delta}{\delta dx}$	$P\frac{l^2-dx^2}{l^3}$	$-P\frac{dydx}{l^3}$	$-P\frac{dx}{l^3}$	0
$\frac{\delta}{\delta dy}$	$-P\frac{dxdy}{l^3}$	$P\frac{l^2-dy^2}{l^3}$	$-P\frac{dy}{l^3}$	0

Table B.6: Derivatives for representation 5: E_{kin},dx,dy, $P=\sqrt{E_{kin}\left(E_{kin}+2m\right)}$, $l=\sqrt{dx^2+dy^2+1}$.

$$\Delta\beta = \frac{s}{c}\left|\frac{1}{t^2}\Delta t\right| \tag{B.20}$$

$$\begin{aligned}\Delta P &= m\left|\gamma\left(1+\beta^2\gamma^2\right)\Delta\beta\right| \\ &= \left|\frac{E}{P}\Delta E\right| \end{aligned}\tag{B.21}$$

$$\begin{aligned}\Delta E &= \left|\frac{dE}{d\beta}\Delta\beta\right| = m\left|\beta\gamma^3\Delta\beta\right| \\ &= m\left|\frac{P}{E}\Delta P\right| \end{aligned}\tag{B.22}$$

$$\begin{aligned}\Delta\theta &= \sqrt{\left(\frac{\delta x}{l\sqrt{1-\frac{1}{\delta x^2+\delta y^2+1}}}\Delta x\right)^2+\left(\frac{\delta y}{l\sqrt{1-\frac{1}{\delta x^2+\delta y^2+1}}}\Delta y\right)^2} \\ &= \frac{1}{l\sqrt{1-\frac{1}{l^2}}}\sqrt{\delta x^2\Delta\left(\delta x\right)^2+\delta y^2\Delta\left(\delta y\right)^2} \end{aligned}\tag{B.23}$$

$$\begin{aligned}\Delta\phi &= \sqrt{\left(\frac{\left(\delta x^2+\delta y^2\right)^{-\frac{3}{2}}}{\sqrt{1-\frac{\delta x^2}{\delta x^2+\delta y^2}}}\delta y^2\Delta\left(\delta x\right)\right)^2+\left(\frac{\left(\delta x^2+\delta y^2\right)^{-\frac{3}{2}}}{\sqrt{1-\frac{\delta x^2}{\delta x^2+\delta y^2}}}\delta x^2\Delta\left(\delta y\right)\right)^2} \\ &= \frac{\left(\delta x^2+\delta y^2\right)^{-\frac{3}{2}}}{\sqrt{1-\frac{\delta x^2}{\delta x^2+\delta y^2}}}\sqrt{\left(\delta y^2\Delta\left(\delta x\right)\right)^2+\left(\delta x^2\Delta\left(\delta y\right)\right)^2} \end{aligned}\tag{B.24}$$

$$\Delta\left(\delta x\right) = \sqrt{\left(\cos\phi\left(1+\tan^2\theta\right)\Delta\theta\right)^2+\left(\tan\theta\sin\phi\Delta\phi\right)^2} \tag{B.25}$$

$$\Delta\left(\delta y\right) = \sqrt{\left(\sin\phi\left(1+\tan^2\theta\right)\Delta\theta\right)^2+\left(\tan\theta\cos\phi\Delta\phi\right)^2} \tag{B.26}$$

	P_x	P_y	P_z	E
$\frac{\delta}{\delta P}$	$\frac{dx}{l}$	$\frac{dy}{l}$	$\frac{1}{l}$	$\frac{P}{\sqrt{P^2+m^2}}$
$\frac{\delta}{\delta dx}$	$P\frac{l^2-dx^2}{l^3}$	$-P\frac{dydx}{l^3}$	$-P\frac{dx}{l^3}$	0
$\frac{\delta}{\delta dy}$	$-P\frac{dxdy}{l^3}$	$P\frac{l^2-dy^2}{l^3}$	$-P\frac{dy}{l^3}$	0

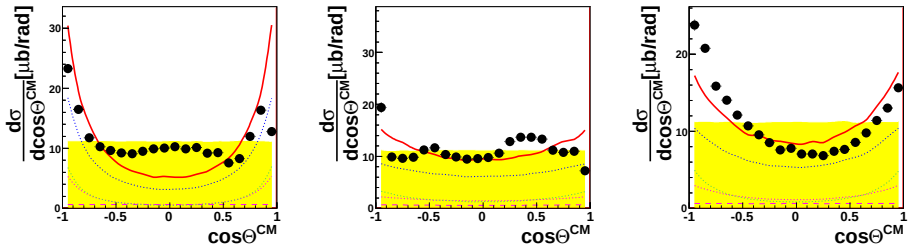
Table B.7: Derivatives for representation 6: P, dx, dy, $l = \sqrt{dx^2+dy^2+1}$.

	P_x	P_y	P_z	E
$\frac{\delta}{\delta\beta}$	$\frac{\delta P}{\delta\beta}\frac{dx}{l}$	$\frac{\delta P}{\delta\beta}\frac{dy}{l}$	$\frac{\delta P}{\delta\beta}\frac{1}{l}$	$\frac{m\beta}{(1-\beta^2)^{\frac{3}{2}}}$
$\frac{\delta}{\delta dx}$	$P\frac{l^2-dx^2}{l^3}$	$-P\frac{dydx}{l^3}$	$-P\frac{dx}{l^3}$	0
$\frac{\delta}{\delta dy}$	$-P\frac{dxdy}{l^3}$	$P\frac{l^2-dy^2}{l^3}$	$-P\frac{dy}{l^3}$	0

Table B.8: Derivatives for representation 7: β, dx, dy, $l = \sqrt{dx^2+dy^2+1}$, $P = m\beta\gamma = \frac{m\beta}{\sqrt{1-\beta^2}}$, $\frac{\delta P}{\delta\beta} = m\frac{1+\frac{\beta^2}{1-\beta^2}}{\sqrt{1-\beta^2}} = m\gamma\left(1+\beta^2\gamma^2\right)$

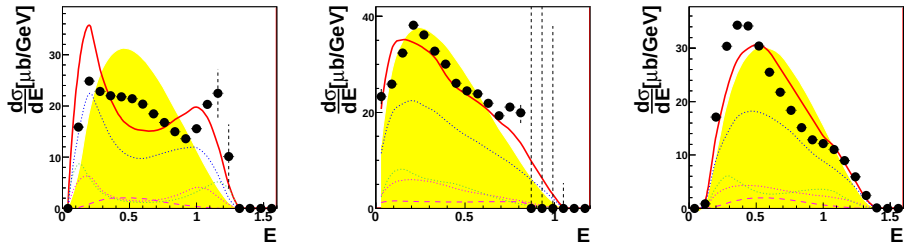
Appendix C

Unfitted and additional Graphs



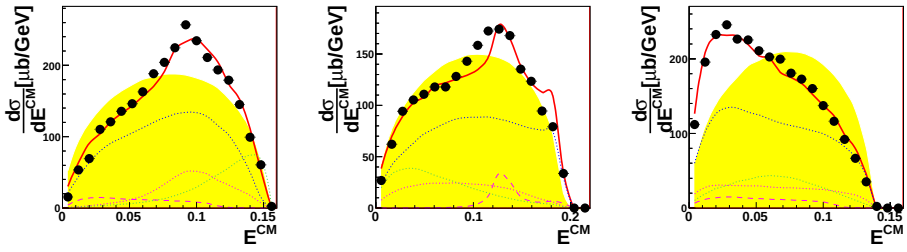
(a) (b) (c)

Figure C.1: The $\cos\theta^{cm}$-distribution for proton, K^+ and Λ.



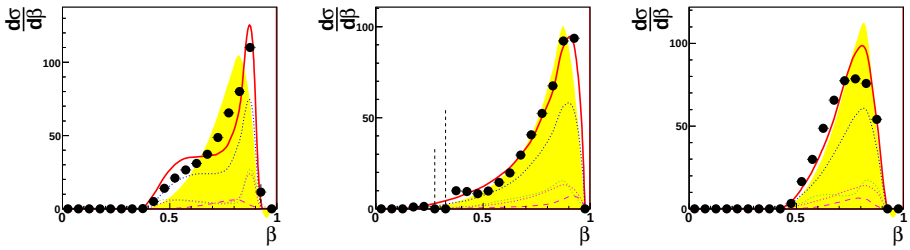
(a) (b) (c)

Figure C.2: The laboratory Energy-distribution for proton, K^+ and Λ.



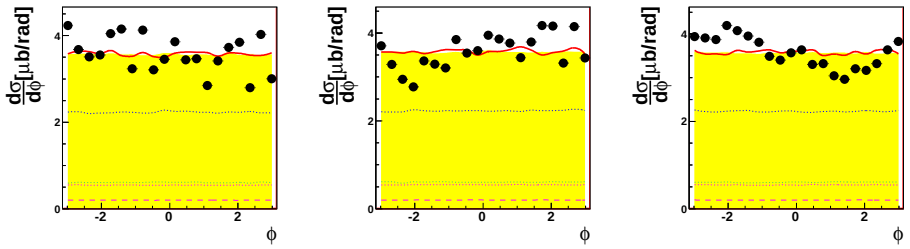
(a) (b) (c)

Figure C.3: The center-of-mass Energy-distribution for proton, K^+ and Λ.



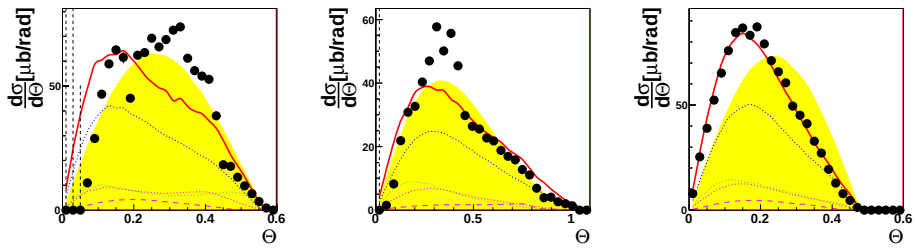
(a) (b) (c)

Figure C.4: The β^{lab}-distribution for proton, K^+ and Λ.



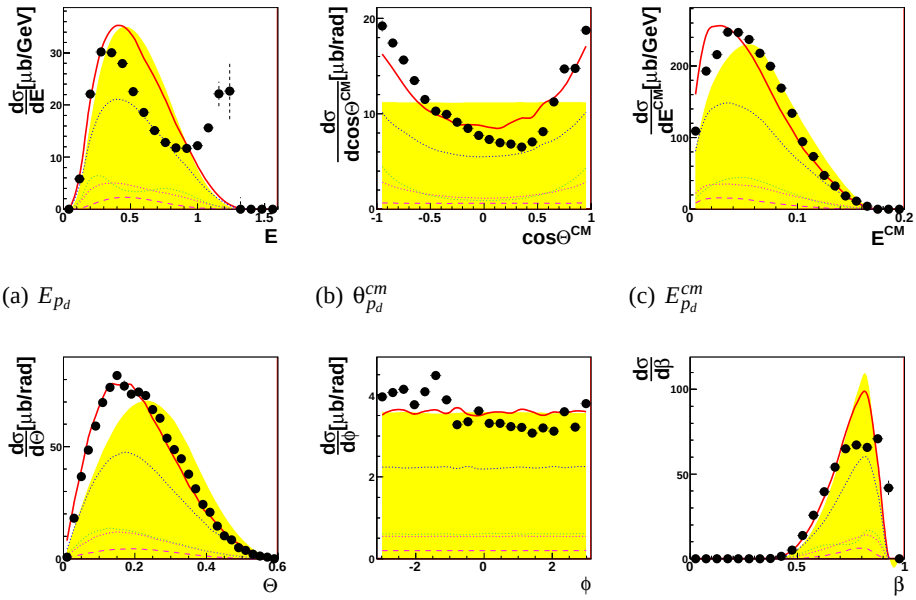
(a) (b) (c)

Figure C.5: The ϕ^{lab}-distribution for proton, K^+ and Λ.



(a) (b) (c)

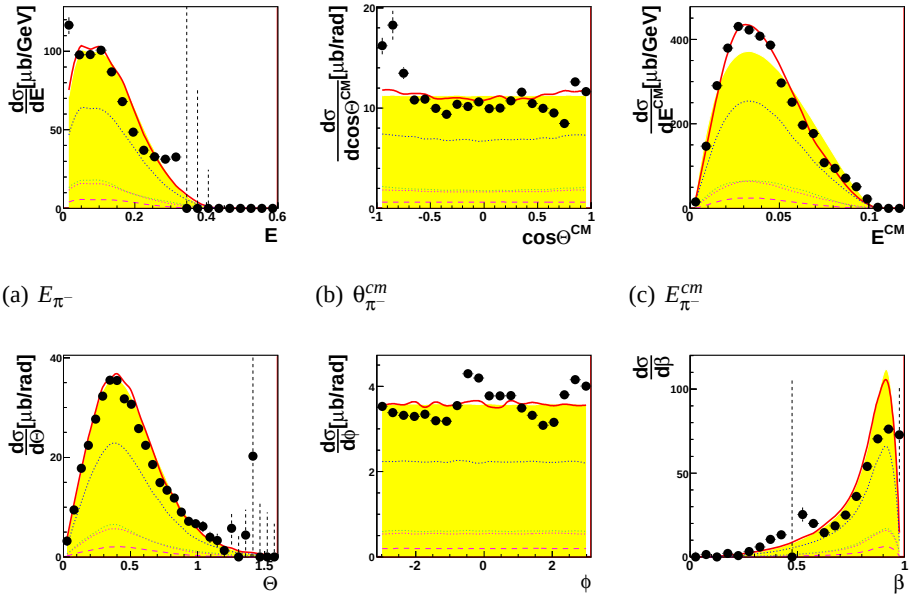
Figure C.6: The θ^{lab}-distribution for proton, K^+ and Λ.



(a) E_{p_d} (b) $\theta_{p_d}^{cm}$ (c) $E_{p_d}^{cm}$

(d) θ_{p_d} (e) ϕ_{p_d} (f) β_{p_d}

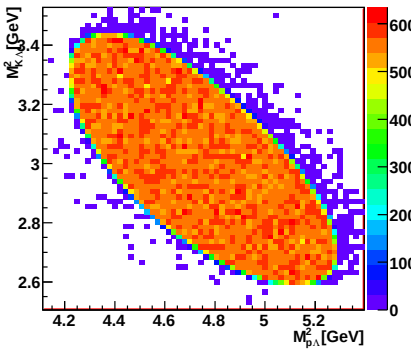
Figure C.7: Λ-decay particle: proton



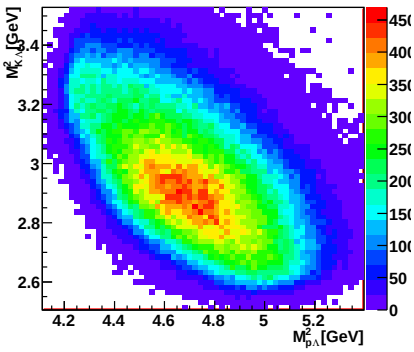
(a) E_{π^-} (b) $\theta_{\pi^-}^{cm}$ (c) $E_{\pi^-}^{cm}$

(d) θ_{π^-} (e) ϕ_{π^-} (f) β_{π^-}

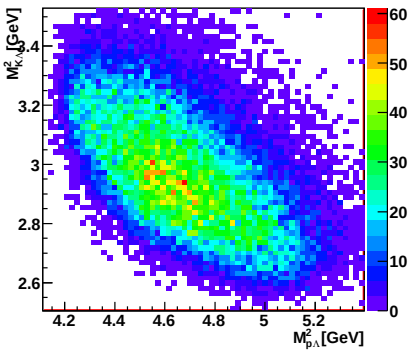
Figure C.8: Λ-decay particle: π^-



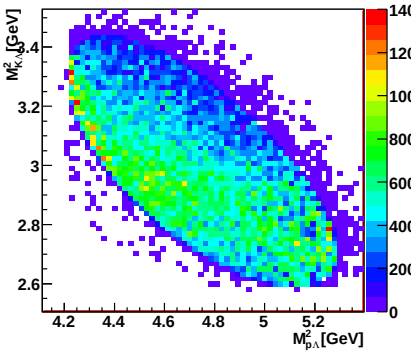
(a) model



(b)

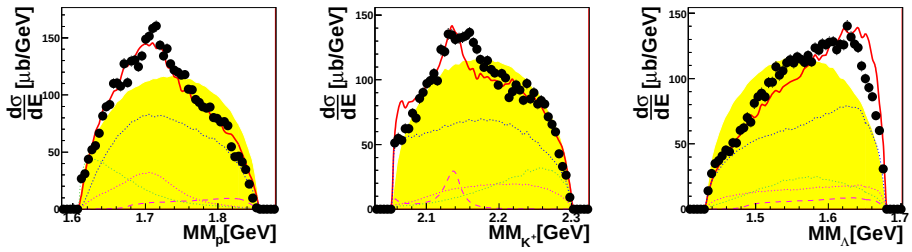


(c)



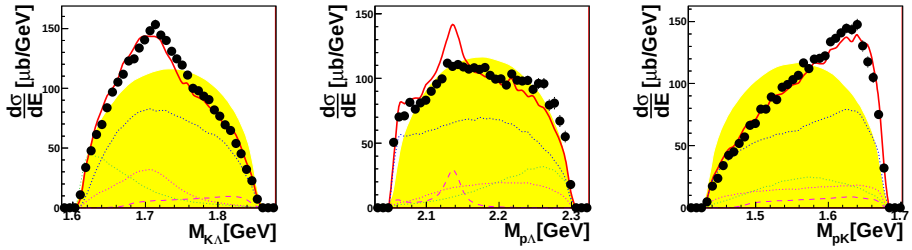
(d) corrected data

Figure C.9: Dalitz-plots for Simulation (a), MC (b), data (c), corrected data (d).



(a) M_p^{miss} (b) $M_{K^+}^{miss}$ (c) M_Λ^{miss}

Figure C.10: Missing Masses of proton (a), K^+ (b) and Λ (c).



(a) $M_{K^+\Lambda}$ (b) $M_{p\Lambda}$ (c) M_{K^+p}

Figure C.11: Invariant Masses of $K^+\Lambda$ (a), $p\Lambda$ (b) and K^+p (c).

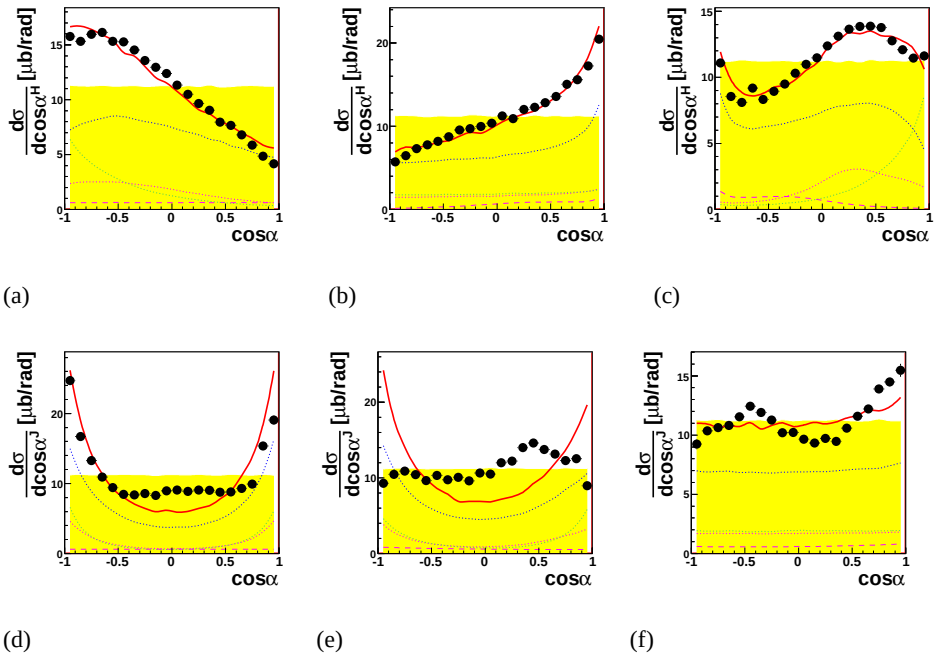


Figure C.12: Helicity and Jackson-Frame-angles.

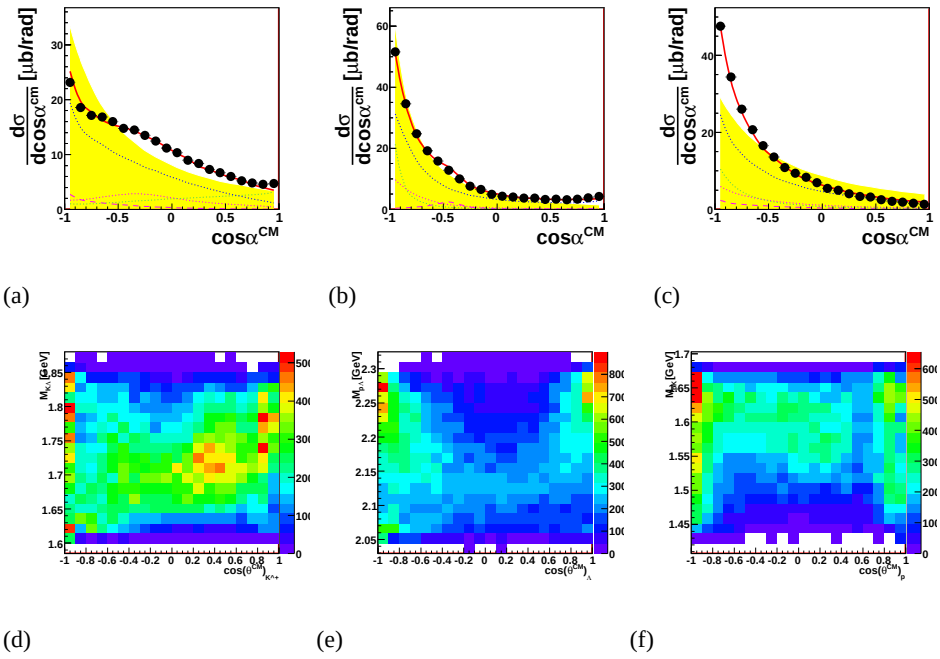


Figure C.13: CM-angles. (a), (b) and (c) show the opening angles of each two particles. (d), (e) and (f) invariant mass of each two particles versus the center of mass angle of one of them.-

C.1 Cusp

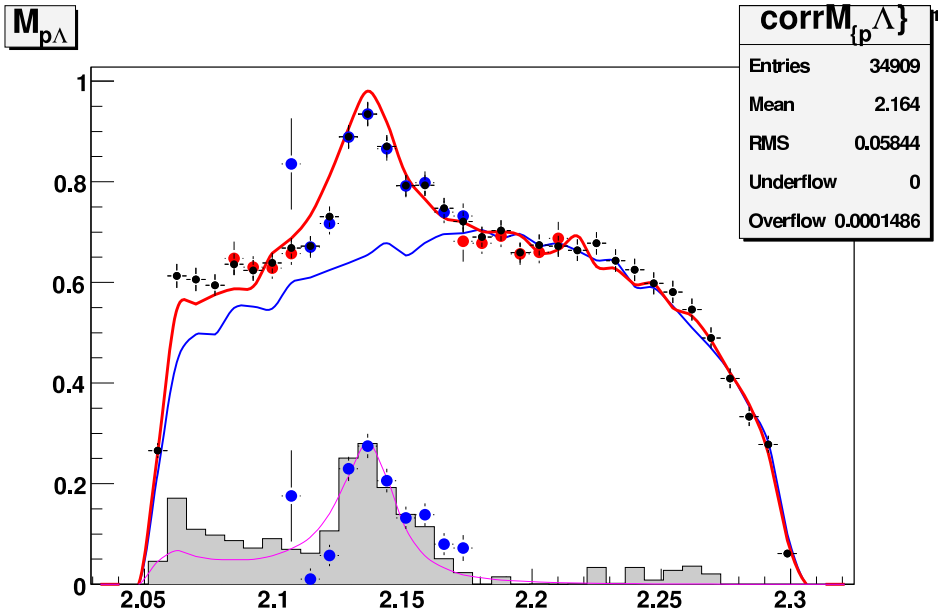


Figure C.14: Considerations to the Σ-cusp total cross-section. Graph shows $M_{p\Lambda}$. Dots is corrected data: black all, blue (up) cusp-band, red side-band. Lines are simulations: red including cusp, blue widthout cusp. Red line is area-normalized to black points, blue line is area-normalized to red line in the range $M_{p\Lambda} > 2.2GeV$. Magenta line is the difference between the two simulations, resulting in a cross-section of $\sigma_{cusp} = 1.65\mu b$.
Gray fill is the difference of all corrected data to simulation without cusp resulting to a cross-section of $\sigma_{cusp} = 2.0\mu b$.
The lower blue dots are calculated using the mean of the red dots and subtracting that from the upper blue dots. This results in a cross-section of $\sigma_{cusp} = 1.2\mu b$.
Scale of the y-axis is dependent on the bin-width d_{bin}: $\mu b/GeV \times d_{bin}$.

Appendix D

typeCase: Program documentation

Here is some documentation on the different components that make up the **typeCase**-Analysis-program. A class reference is available on the net [23] or with the help function in **typeCase** itself. The complete reference documentation would go beyond the scope of page-limits for this thesis.
The program itself consists of several components

D.1 Analyzer

The analyzer is the class, that can do the complete analysis, though, the algorithms and container-structures don't essentially rely on it.
It comes with a large number of methods, some of them, several times with different parameters, provided for convenience.
The usual procedure is

1. initialize using *initStep()* or
   (a) *initData()*
   (b) *createSetup()* (or *defineMaterials()*, *defineDetectors()*, *defineReaction()*
   (c) *initAlgorithms()*
2. process
   - *process()* or
   - *step()*
3. finalize using *endStep()* or
   (a) *killAlgorithms()*
   (b) *killSetup()*

(c) *killData()*

If you are not satisfied with the way the event-loop is handled here, use the *getAlgorithm()*-method to allocate the algorithms and do the actual event-loop yourself. To provide trees and histograms as well as the event structure for drawing (GUI), appropriate functions and signals have been implemented.
For process control there are slots that add and remove runs from a runs-to-analyze-list, to request specific events and to stop the analysis at some point.
The most critical point of this class is the *getAlgorithm()*-method, where the actual algorithms are allocated according to their IDs given by the algorithm_parameter provided as parameter to the method. This is the only point, where the algorithms actual type is selected and it is treated according to its type. Later the algorithms are not anymore distinguishable. For easy installation of new algorithms an installation-procedure has been implemented (see D.5).

- Init methods
  - *tofAnalysis(int max=100,int maxDets=100, int NThreads=1)*
    Constructor sets the maximum number of detectors, the maximum number of hits and pixels per detector and the number of threads the analysis shall be done with.
  - *∼tofAnalysis()*
    Destructor. Finalizes the analysis if not done before and frees allocated memory.
  - *void initStep(const vector<algorithm_parameter> &p, vector<beamTime_parameter> &beamParam, vector<run_parameter> & runParam,vector<detector_parameter> &dets, vector<material_parameter> &mats, reaction_parameter col)*
    Initializes the analysis for some given algorithm-parameters (cannot be changed during analysis, you have to reinitialize for that), some databasis and setup.
  - *void initData()*
    Initializes the data-structure.
  - *void defineMaterials(const string &fileName,const string &fileName2)*
    Reads material-parameter from file and initializes the materials.
  - *void defineMaterials(vector<material_parameter> &mats,vector<detector_parameter>&dets)*
    Initializes used materials.
  - *void defineDetectors(const string &fileName)*
    Reads detectors from file and initializes detectors.
  - *void defineDetectors(vector<detector_parameter>&dets)*
    Initializes detector-setup.
  - *void defineReaction(reaction_parameter col)*
    Initializes experiment-parameters.

- *void createSetup(vector<detector_parameter> &dets, vector<material_parameter> &mats, reaction_parameter col)*
  Initializes complete setup by calling above functions.
- *void createSetup(const string &setupFileName, const string &materialFileName)*
  Reads setup from file and initializes setup by calling above functions.
- *void initAlgorithms(const vector<algorithm_parameter> &p, const run_parameter & runParam)*
  Initializes algorithms for all threads by calling the static *getAlgorithm()* function.

- finalize methods
  - *void killAlgorithms()*
    Finalizes algorithms by calling destructors.
  - *void killData()*
    Frees data-structures.
  - *void killSetup()*
    Frees setup-data-structures.
  - *void endStep()*
    Finalizes complete analysis by calling above functions.

- process control methods
  - *bool step(int num=1)*
    Makes the analysis run for *num* events, if the end of one run is reached, the newRun-signal is emitted for the subsequent run and the analysis continues.
  - *bool process()*
    Makes the analysis run until there is no more data to process, if the end of one run is reached, the newRun-signal is emitted for the subsequent run and the analysis continues.
  - *void doRequestEvent(int eventNumber, int runNumber)[SLOT]*
    Emit the requestEvent-Signal, that can be caught by an input algorithm.
  - *void doRequestNextEvent()[SLOT]*
    Emit the requestNextEvent-Signal, that can be caught by an input algorithm.
  - *int stopAnalysis()[SLOT]*
    Stops the analysis after the current event.
  - *void *thread_run(void *ptr)*
    Function, that is called for each thread, that processes events until a specified number of events is reached or the input is empty.

- io-methods
  - *vector<vector<string>> getHistogramNames()*
    Returns for each initialized algorithm, that defines histograms the array of histogram names headed by the algorithm name.
  - *TH1 *getHisto(const string &name)*
    Returns the histogram of name *name*, defined by some initialized algorithm, or NULL if a histogram of this name is not defined.
  - *vector<vector<string>> getTreeNames()*
    Returns for each initialized algorithm, that defines trees the array of tree names headed by the algorithm name.
  - *TTree *getTree(const string &name)*
    Returns the tree of name *name*, defined by some initialized algorithm, or NULL if a tree of this name is not defined.
- static methods
  - *static vector<string> getVariables()*
    Returns a vector with variable names, their types and a comment on whether they are defined once or once per thread, that are accessible during initialization of an algorithm in the function *getAlgorithm()*.
  - *static algorithm_parameter getAlgorithmParameter(int ID)*
    Returns for an algorithm ID the description the algorithm-class provides.
  - *static int getAlgorithm(AAlgorithm ***out,int &executeUpTo,const algorithm_parameter &param,const run_parameter &firstRun, TEvent &event, TSetup &setup, int **numberOfHits, int *numberOfTracks, int **numberOfPixels, int **numberOfClusters, int **numberOfHitClusters, TRawHit ***raws, TCalibHit ***calibratedHits, THitCluster ***hitClusters, TTrack **tracks, TPixel ***pixels, TCluster ***clusters, TMaterial **materials, TDetector **detectors, bool &readValid,int &readInID, QObject *stearIt,bool &eventRequesting)*
    Basic algorithm initializing method. Returns the total number of algorithms that have been allocated for algorithm *param*. These algorithms are stored in *out*, *executeUpTo* gives the number of algorithms that should be called in the event-loop, the rest can be such things as fitting algorithms. Apart from the data-structures, there are *readValid* that is given to reading-algorithms that will switch it to false if the number of remaining events for the current file is zero, *readInID* which is obsolete but kept for backward compatability, *stearIt* the class, that provides the essential signals for e.g. new runs, *eventRequesting* that provides a switch, that of several different reading algorithms one (not necessarily the first one) can act as an event-input-list.

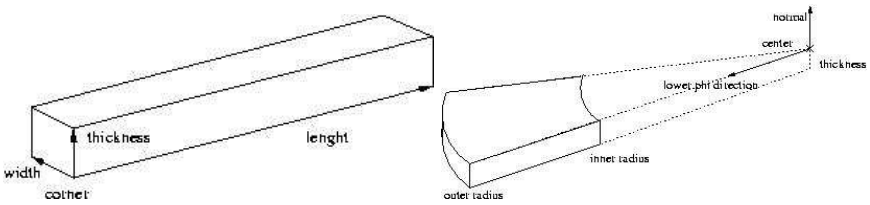
- slots
  - *void addRun(run_parameter &rp)*
    Adds a run to the list of runs to analyze.
  - *void removeRun(run_parameter &rp)*
    Removes a run from the list of runs to analyze. Cannot remove current run.
  - *void showNewRun(run_parameter & run)*
    Debugging output. Writes Run information to analysisLog.
- signals
  - *newRun(run_parameter&r, beamTime_parameter&b)*
    This signal is emitted whenever a new run will be necessary, right at the beginning of the analysis and whenever the current run has no events left to analyze and there is a new run in the runs-to-analyze-list. If your algorithm needs to do something with every new run (e.g. read some run-specific calibration) connect a slot to this signal.
  - *newEvent(int nrEvent)*
    After each thousand events in step- or process-mode, after each event in single-step-mode this signal is emitted providing the number-of-analyzed-events or the event-number (single-step-mode).
  - *changeEvent(TEvent *ev)*
    After each thousand events in step- or process-mode, after each event in single-step-mode this signal is emitted providing the event-data-structure. Useful for drawing purposes.
  - *algorithmInit(int algoID)*
    Signal is emitted, when an algorithm of ID *algoID* is going to be initialized.
  - *algorithmInited(int algoID)*
    Signal is emitted, when an algorithm of ID *algoID* has been initialized.
  - *analysisFinished(int numEvents)*
    Signal is emitted, when the analysis stopped after *numEvents* events.
  - *initStateChanged(const char *msg)*
    Signal is emitted, when some step in the initialization process is finished.
  - *finishStateChanged(const char *msg)*
    Signal is emitted, when some step in the finalization process is finished.
  - *requestEvent(int eventnumber, int runnumber)*
    Signal is emitted to request some specific event in analysis.
  - *requestNextEvent()*
    Signal is emitted to request the next event, when using some algorithm as event-input-list.

Some of the methods that are still member of the class are kept for convenience keeping backward compatability, but are not documented here.

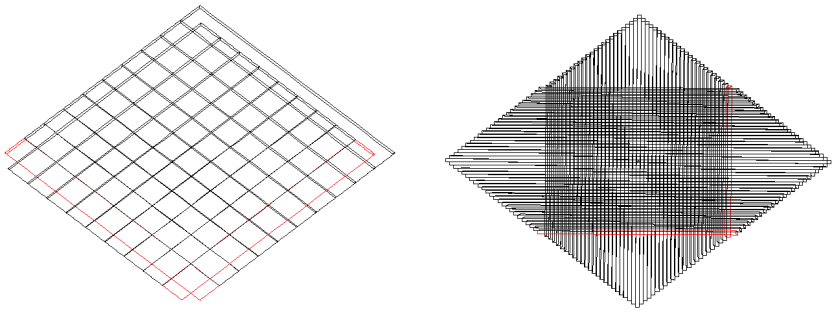
D.2 Shapes classes

The shapes package defines some methods that have to be implemented for the shapes to be usable for e.g. the tracking algorithms. Most of them are only used for the volumes, but methods like *Draw* or *getClone()* are also useful for planar shapes. The essential methods are:

- *entrance(const sLine3D &line)*
  that returns the entrance point of a straight line into the volume,
- *distance(const sLine3D &line)*
  that returns the distance vector of the line to the volume, which is the zero-vector if the volume is hit,
- *getClone()*
  that returns a copy of the volume, that has to be deleted by the user,
- *suspect(const sLine3D &line, int stackType)*
  returning the most probable element number, a straight line may hit, for a given stack-type considering this shape as element number zero,
- *HitParams(const sLine3D &line)*
  that calculates the parameters needed for tracking,
- *DrawProjected(const point3D &eye, const plane3D &plane, TCanvas* canvas, int color1=1, int color2=0, int style1=6,int ptt=-1)*
  that projects the surfaces of the volume and draws them on a canvas; this is actually depreciated use
- *Draw(const point3D &eye, const plane3D &plane, vector4D* boundingBox, TObject **ident, int lColor, int fColor, int fStyle)*
  instead, here you also get a bounding box for the shape and a pointer to a TObject, that my identify the drawing on a canvas,
- *getNext(int times, int stackType)*
  that calculates the next element out of the own properties and the stackType,
- *getEnvelope(int times,int stackType)*
  that calculates the enveloping volume of the whole detector,
- *Hitting(const sLine3D &line)*
  that calculates all the properties of a possible hit between a straight line and the volume, and



(a) fiber (b) wedge



(c) stacked fibers (d) stacked fibers

Figure D.1: (a) Sketch of a fiber, showing the corner point and the three direction vectors. This shape is used to describe the two TOF-hodoscopes. (b) Sketch of a wedge, showing the center point, the lower phi edge direction vector, the normal vector, thickness, inner and outer radius properties. This shape is used to describe the TOF-start detector, one layer of the micro-strip detector, one layer of each Quirl and Ring detector and the Barrel detector, using an additional property, the tilt angle between the normal vector and the symmetry axis. (c) and (d) show two different ways to stack fibers. (c) is stack-type 2, (d) stack-type 17.

- *Normal(const sLine3D &line)*
  that returns the entrance point and the vector normal to the surface in that point; this method is especially useful, if you want to do ray-tracing on the volumes.

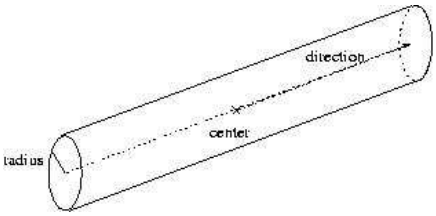
These methods, especially the methods used for tracking are and should be optimized for speed, since these methods are the bottle-neck in the analysis. For the hit calculating methods, it pays to do the code for hit calculation once, in Hitting, and call this method from the others (like entrance, distance, etc.), this prevents too many bugs and makes the code clearer and easier to read. For the tracking methods, note, that only hit-point-calculation is done for the surfaces the line can actually hit to speed this calculation up, so the straight line is chosen as to be a half-limited straight line, pointing from the footing point into a direction (e.g. target outwards).

Fiber The fiber is a box like shape, that consists of six planar surfaces, each two opposing surfaces being parallel (fig.D.1(a)). It is represented by a corner-

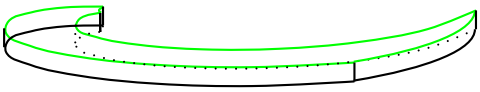
point and three direction vectors. In special case, the fiber is a rectangular box, but it doesn't need to be. There are several stacking parameters available, as rectangular stacking and extended rectangular stacking. The rectangular stacking stacks the fibers in the direction of the second direction vector, with no space in-between. There is a hole left out, when the number of halved rows is larger than zero. This is to include something like a beam-hole. The difference between rectangular and extended rectangular stacking is that the fibers are elongated by twice the shifting vector times the element number, up to the position of the beam-hole and back, so that a stretched hexagonal shape results as envelope. This stacking was implemented to pay respect to the hodoscope that was installed in autumn 2004, that has two layers of that shape and a third layer in rectangular stacking (figs.D.1(c), D.1(d)).

Wedge The wedge in principal looks like a pizza- or cake- piece (fig. D.1(b)). It has a center point, that is not the center of mass of the volume, but the center of the circle, of the front plane of the cake, the piece was cut out. It has a normal vector, to define the front plane and a vector that defines one of the edges. It also has a phi range, an angle, by which the edge vector is rotated around the negative normal vector to result in the other edge and a thickness. There are two radius variables, an inner and an outer radius. This shape describes nicely the start detector and the straight layers of the Quirl- ,Ring- and the Micro-Strip-detector. The individual elements of the Barrel detector are also wedge shaped, but with a far away center point and radii that are close together; also the elements are arranged in a way that they result more in a cone-shape mantle than a cake. So there is an additional variable, that accounts for the distance of the outer circle of the envelope to the center-line of the envelope. There are four stacking types available: circular stacking, that results in a cake shaped envelope, extended circular stacking, where the elements that come after completing a circle are added on the outer radius of the circle, cone stacking, that takes the tilt of the Barrel into account, and line stacking, that stacks the wedges in line with the negative normal vector, the result here is a thicker wedge (fig. D.2(d)).

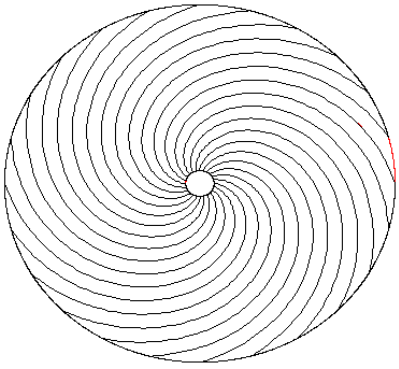
Spiral The spiral shape is in general like the wedge, except of the bending of the edges. The edges are not straight lines but archimedian spirals, with the equation: $r = \frac{r_{max}}{\phi_{max}}\phi$, where r is the distance from the center-point on the plane, and phi is the angle between the edge-vector (fig. D.2(b)). The archimedian spirals are build in the Micro-Strip-Spiral, the Quirl- and Ring detector. In the latter there are one wedge shaped layer, and two spiral-layers in opposite bending. The result of this is pixels of each three elements, that cover all the same angle in space. For the spiral shapes, there exists only the



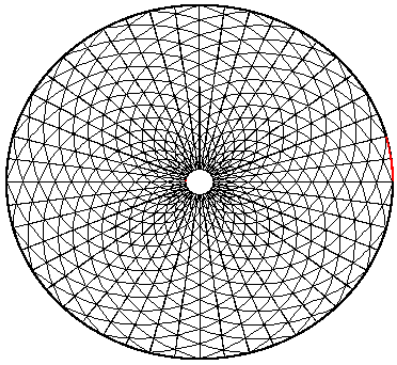
(a) cylinder



(b) spiral

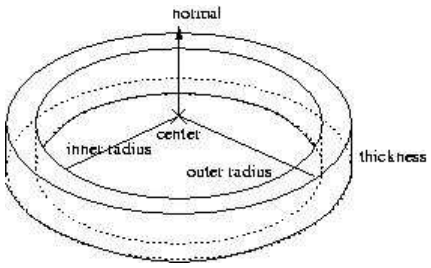


(c) spiral stacking

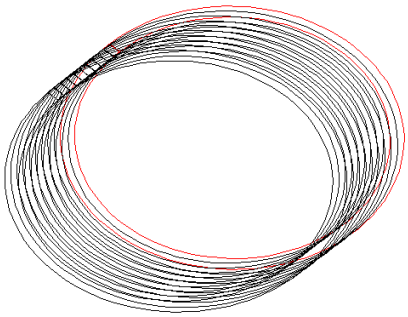


(d) spiral and wedge stacking

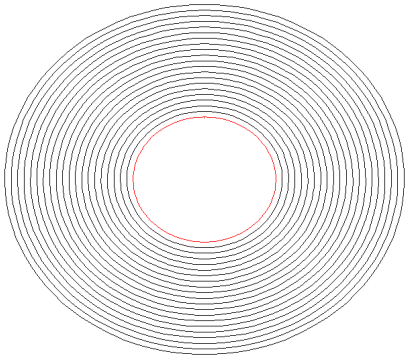
Figure D.2: (a) Sketch of a cylinder, showing center point and direction vector along with the radius property. This shape is used to describe the target volume and the straw-tube-chamber. (b) Sketch of a spiral, showing the center point, the lower-phi-edge vector, the normal vector, the thickness, the outer and inner radius and the bending. This shape is used to describe the bent layers of both Quirl and Ring detector. (c) and (d) show the stacking of spirals and wedges to form the Quirl detector. In both cases stack-type 1 is used.



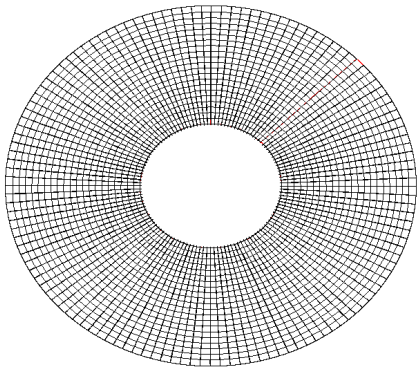
(a) ring



(b) stacked ring



(c) stacked ring



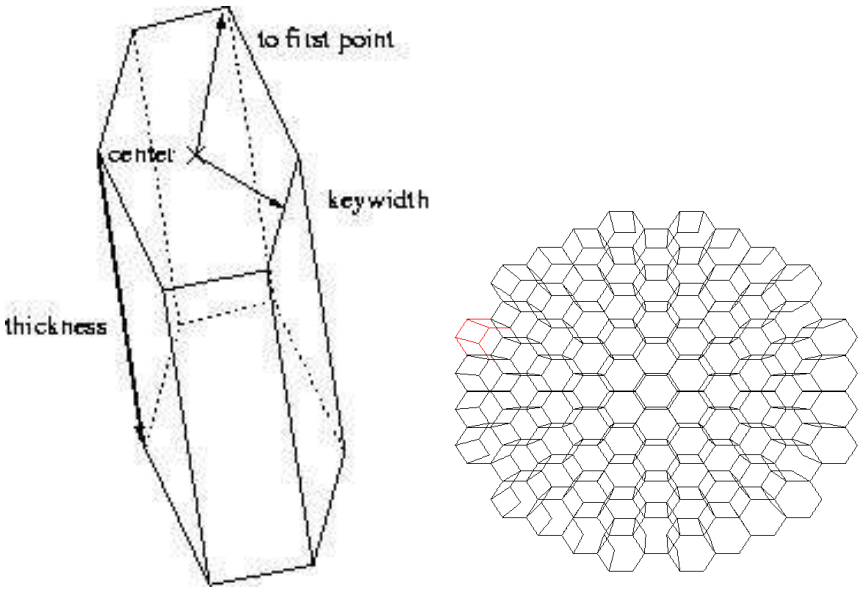
(d) stacked microstrip

Figure D.3: (a)Sketch of a ring, showing the center point, the normal vector, the thickness, the inner and outer radius properties. This shape is used to describe one layer of the Micro-Strip detector. (b) and (c) show the different ways the ring can be stacked with (b) having stack-type 2 and (c) stack-type 1. (d) shows ring and wedges combined as used for the Micro-Strip-detector (sec. 2.2.4).

circular stacking, to result in a cake-shape (figs. D.2(c), D.2(d)).

Cylinder The cylinders consist of a center-point, a direction vector, that describes the shift of the center-point to one of the ends of the cylinder, a radius and a preference direction, that is used for stacking (fig. D.2(a)). Here exist two types of stacking, rectangular stacking, that results in a box like envelope and circular stacking, that is similar to the extended circular stacking of the wedge. The cylinder is used for the target volume.

Ring The ring is a generalized form of the wedge and a more specialized form of the cylinder (fig. D.3(a)). Generalized in the way, that it has no edge-vectors, it covers 2π range, specialized in the way, that it has an inner and an outer radius, it is not filled. There are two types of stacking implemented: circular



(a) hexPrism (b) stacked hexPrism

Figure D.4: (a) Sketch of a hex-prism, showing the center point, the key-width vector, the first-point-vector and the thickness vector. (b) shows how the hexPrisms are stacked to form the calorimeter.

stacking, that modifies the radii of the rings, such that they increase the radius of the envelope (fig. D.3(c)), and line stacking, that results in a thicker ring (fig. D.3(b)).

Hexprism The hex-prism is the most complicated of the volume shapes. It has eight planar surfaces, always two of them in parallel (fig. D.4(a)). The front and the back surface are a regular hexagon, the backward simply shifted by the thickness vector. The other surfaces connect the back and the front plane. The front plane contains as center of mass point the center-point, the vector from the center-point to one of the corner-points and a key-width vector, that is in plane but normal to the other vector. This key-width vector also gives the direction for stacking. There is line stacking, that generates next elements in a row, and snake-stacking, that generates a hex-prism shaped envelope (fig. D.4(b)).

StrawTube The straw-tube is derived from the cylinder, inheriting all properties and methods, but there is an additional parameter, the isochrone-radius, that makes it necessary to implement a new class.

triangle with three corners. It calculates its center itself.

sphericTriangle To pay respect to the more unusual shape of the pixels in Quirl and Ring, this spheric triangle has been developed. The center property has to be set from outside.

rectangle represented by two four points. The angles between two connected edges are 90deg.

quadrangle Any four points in plane.

spheric rectangle This planar shape could be the front and back side of a wedge. The center point here is calculated using the center point of the underlying wedge shifting it into the direction of the sum of both edges of the wedge by the mean of the inner- and outer radius.

hexagon A regular hexagon, with a center, a normal vector and a so called first point A.

circle A planar circle.

D.3 Container classes

D.3.1 TMaterial

This is a class, that is not used much so far, but is generated for completeness and perhaps future use. It contains the properties of a material, a detector or dead material[1] can consist of, such as density, radiation length, speed of light and the possibility to enter the elements, with mass (in GeV) and charge, the material consists of.
Lately it finds use in the correction of the timing due to run time of the signal (light for scintillators) from the generation-point (where the particle passes through the material) to the collection point (e.g. photo-multiplier).

D.3.2 TDetector

The class TDetector is designed to hold one single detector layer, but can handle more if necessary. It holds the geometry of the detector in question and a pointer to the material the detector consists of. Each element is kept in an array and can be used by the appropriate method. Note, that here a pointer to the original data is returned, so if modified, the volume shape of the element is modified. The geometry is not defined by this class, but is put in as the shape of the first element, that has to be generated outside and an integer parameter that holds the way, the other elements are generated out of the first one. But still this generation is not done

[1] Dead material is the opposite of detector material. It is not connected to the DAQ, but distorts never-the-less the measurement simply by its presence.

in the TDetector class but by the volume shape itself (see 3.8). The possibility to modify the shapes gives you the possibility to alter the shapes according to your needs after generation. Additionally this class holds an ID value, corresponding to the ID of the detector it reflects.
Later it will be possible to read in the shapes of the individual elements from file, if they are aligned so irregularly that they cannot be generated from the first one. This is not necessary for the TOF-detector, but may be useful for other detectors (like WASA, central calorimeter e.g.).

D.3.3 TTarget

The TTarget class is a small class to hold the properties of the target. It has a volume shape, reflecting the target shape, a temperature variable and a four momentum vector to describe the particle the target is made of. This will have to be modified in the future to be usable for other than elementary (proton or deuteron) targets. As for the TDetector class the TTarget class has to be given an outside generated volume shape as target volume.

D.3.4 TBeam

This class also is still not perfect, since not much used. It consists of a plane shape, that corresponds to the beam spot, the projection of the beam on the target surface, a value for the emittance and a four momentum vector for the beam particle.

D.3.5 TSetup

The TSetup class is the class that holds the setup of the whole experiment, detectors, beam(s) and target. It has arrays for materials (TMaterial) and detectors (TDetector) and variables for beam and target. Furthermore, it provides the possibility to have a two beam experiment, a collider experiment, so you can supply a second beam. All properties have to be generated outside the class and added to the TSetup variable to hold them.

D.3.6 Raw hit data: TRawHit

The TRawHit is the most basic class, it simply contains four integer variables, naming detector and element of the detector that is hit and the hit properties, the QDC and TDC information, in channels, as it is read out by QDC and TDC cards. If one of the properties is not needed, one can set it to -1.

D.3.7 Calibrated hit data: TCalibHit

The TCalibHit class contains the hit data after calibration, which is a variable of type TRawHit, for the unprocessed hit and additionally two floating point variables for energy and time information and a volume shape that reflects the detector geometry. The volume shape pointer is saved as assigned, but only a clone can be retrieved for further processing. This makes it possible to save a volume shape once for any number of events, when written to file. But makes it also less vulnerable to programming errors. As mentioned above, the TCalibHit structure doesn't contain any calibration data, only calibrated data, and it doesn't retrieve its volume shape itself. It has to be assigned by appropriate algorithms like ACalibration (3.10.2) and AGetTheShape (D.4). Furthermore it has a property, that makes it possible to distinguish between hits, that have passed all cuts and requirements, and those that have not: the valid-property.

D.3.8 TPixel

The TPixel class is the next class in reconstruction hierarchy. Very seldom, the detectors are built in an intrinsic pixel structure. At most times, the pixels with fine resolution in x- and y- or θ- and φ- are reconstructed out of hit detector elements, that intersect somehow in a projection to a plane or a sphere. The way this reconstruction is done is not relevant for the TPixel class, the concrete algorithm has to be applied from the outside. But the TPixel structure saves the elements that contribute to the pixel and a planar shape, that gives the shape of the pixel.

D.3.9 TCluster

Often when a particle traverses a detector layer, it deposits not only energy in one element per layer, but perhaps in two or more, depending on the specific direction and the thickness of the detector layer. Here a simple pixel structure, that saves only one element per layer is not enough anymore. The TCluster handles a number of TPixels, that correspond to one particle, that went through this detector, along with a three-dimensional point that is the center of the cluster shape. To have the exact planar shape is out of question, since it can be very complex.

D.3.10 THitCluster

This class is added quite lately in the development process, to implement the whole functionality of the analysis software used at the Forschungszentrum Jülich. Here due to design features of the software, the generation of pixel-clusters was not possible, so the cluster search is done on hit level, merging several neighboring hits into one, deleting the non used ones.

The principle of this package is never to delete any information, so a new class was implemented. It is derived from TCalibHit inheriting all properties and functions. It can be used alternatively to the TCalibHit objects, making it possible to make as less modifications to the existing algorithms as possible with the full functionality. Pass TCalibHit or THitCluster to an algorithm requiring TCalibHit, the algorithm won't even notice the difference.

D.3.11 TTrack

The TTrack structure has a lot of properties to describe. First of all the path of the particle, the track describes. Here the sLine3D structure from the geometry package (3.5) was chosen, since it has a footing point Foot and a direction vector Direction, that describes the particles flight at its vertex point. In a future package this may be extended to a variable, that can hold any class object, that is derived of sLine3D, to enable curved lines. The TTrack structure can hold any TCalibHit element, that was hit by the particle going through the detector, along with all the pixels. You can set a track to be a prompt track or a secondary track, that doesn't origin in the target, but on the path of an other track, due to a decay, and pointer to secondary tracks, in which this particle decayed. Additionally there are some other analysis relevant properties, like errors for the track properties, that can be used, for example a kinematic fit, and a χ^2 value reserved for the fit of the path to the elements/pixels. Finally there are kinematically relevant properties as speed of the particle, its kinetic energy, its charge (this one is an integer value) and certainly a four momentum representation of the particle.

D.3.12 TEvent

The TEvent class combines all other event based classes. Here all hits, pixels, clusters and tracks are saved. This class allocates the memory for the TCalibHit, TPixel, TCluster and TTrack objects. The references to these objects can be retrieved by the.member functions. The TEvent class provides place for event specific information, such as event number, run number and an integer trigger information, additionally it has a three-dimensional vector for polarization information. Finally there is an array saving information about reactions, that could have taken place in this event.

D.4 Algorithm classes

Most has been discussed already in section 3.10, though I will provide a list for completeness:
AAlgorithm is the base class for all algorithms. An algorithm, that is not derived from AAlgorithm (not necessarily directly) cannot be used in this analysis. Its

constructor asks for a string containing the name of the algorithm but this can also be a description, no essential need to be unique. The parameter-list of the derived classes usually is much more lengthly than that. Its properties are:

- *string getName()*
  Returns the name of the algorithm specified in the constructor.

- *void *process(void*ptr)*
  Method that will be called in the event-loop. Anything you want to happen then put it here.

- *vector<string>histogramNames()*
  This method returns a vector of names for histograms that are defined in the algorithm. By default it is empty. It is called by the GUI, to make the histograms accessible and drawable during analysis. If you define debugging or write-out histograms do not hesitate to overwrite this method and make them available here.

- *TH1* histogram(string histoName)*
  Returns the histogram of name *histoName* or NULL if none such exists. If you inserted a histogram-name into the return-vector of the method above, also add some commands here to make the histogram available for viewing.

- *vector<string>treeNames()*
  This method returns a vector of names for trees that are defined in the algorithm. By default it is empty. It is called by the GUI, to make the trees accessible to the GUI during analysis. If you define debugging or write-out trees do not hesitate to overwrite this method and make them available here.

- *TTree* tree(string treename)*
  Returns the tree of name *treeName* or NULL if none such exists. If you inserted a tree-name into the return-vector of the method above, also add some commands here to make the tree available for viewing.

- *static algorithm_parameter getDescription()*
  This static member function returns an algorithm_parameter that holds all parameters (not necessarily with sensible values), the algorithm requests. This method is not essential, but overwriting it in derived classes makes things more comfortable.

As derived algorithms there exist:

- IO-algorithms

  AReadFromTade The most basic input algorithm, reading from the ASCII TADE-format described in sec. 3.10.1

  AConversion Place holder class. Here you can define your own conversion from your data format to the **typeCase** data format.

AHitTreeInput and AHitTreeOutput Write and read hits to/from root-TTree-file. The hit tree contains branches for detector and hit element, the raw TDC and QDC as well as for the calibrated values.

ATrackTreeInput and ATrackTreeOutput Read and Write Tracks to/from root-TTree-file. The track-tree contains branches for the general track properties, but also for the hit elements assigned to the track. Apart from the pixel-information, there is no information lost when writing to file.
Prompt-, kink- and vee-tracks are stored in different trees. For the input, there exist a whole range of options starting from using a directory on the local machine to reading only specific event-pattern.

AMultipleTreeInput and AMultipleTreeOutput An older version of ATrackTreeInput/Output with fixed branch-length. This class is obsolete, but maintained for format compatability reasons.

- Generation algorithms

AGenerateEvents This algorithm generates events using the root TGenPhaseSpace phase-space particle generator to generate prompt tracks. If desired you can apply a modulation function/graph/histogram to implement some model.

AGenerateEventsWithDecay This algorithm not only generates prompt tracks, but also simulates decays of a specified number of tracks. To all vertices separate modulation functions can be applied.

AVirtualMachine The two algorithms above only generate the particles. This algorithm here is an extremely simple virtual machine, generating hits in the detector volume causing energy loss for the particles. It does not do real scattering, so the angles are conserved.
This is a toy virtual machine, do not use it as realistic simulation of your detector, but feel free to upgrade it into one.

- Calibration algorithms

AGetTheShape Assigns the shape from the detector-structure to the hit-structure. This is only needed if the input-algorithm doesn't do it itself (e.g. AReadFromTade).

ACalibration Calibration algorithm. Applies Offset-, Walk- and Binning-calibration-parameters to the hits QDC and TDC and applies cuts for QDC and TDC.

ATeufelCorrection As mentioned in sec. 3.10.2, the Teufel-correction is applied here. Note that it is applied to the raw QDCs. All other calibration steps leave the raw values untouched.

- Pixel and Cluster algorithms

BarrelPixel For a detector with two-sided-read-out it defines a pixel, with the pixel center defined by the difference of the TDCs between the back- and the front-read-out.

HodoPixel Pixel calculation for 2 perpendicular fiber-shaped hodoscope layers.

AHodo3Pixel Pixel calculation for 3-layered fiber-hodoscope, first two layers have to be perpendicular, the third in 45° to these.

MicroPixel Pixel calculation for 2-layered detector with one side wedges and rings on the other.

Ringpixel Pixel calculation of detectors of type Quirl or Ring see sec. 3.10.3, A.1 and figures 2.7(a) and 2.7(b).

AHitClusterSearch Cluster search on hit level.

MicroCluster Cluster search on pixel level.

- Tracking algorithms

TüTrackSearch First approach to implement the Jülich prompt-tracking routine into **typeCase**. Since it is quite slow it is in use no more (see sec. 3.10.4).

TüVSearch First approach to implement the Jülich vee-tracking routine into **typeCase**. As for the prompt version it is too slow to be in use anymore (see sec. 3.10.4).

ALineTrackSearch This is the improved prompt tracking algorithm using suspect-search described in sec. 3.10.4.

AVLineTrackSearch This is the improved vee-tracking algorithm using suspect-search described in sec. 3.10.4.

APixelTracking This algorithm implements the pixel-vee-search used in Dresden. It is a brute force method, combining any two pixels to possible tracks. It is quite slow, but has a high efficiency. It combines intersecting tracks to decays. Can be switched to find also prompt tracks.

APromptHistoTracker This algorithm implements the projection-search for prompt tracks used in Erlangen. Since it works on histograms instead of 2D-function-lines, it is definitely slower than the suspect-search, and the output is comparable.

AKinkSearch This is a kink search, finding charged decays between two detector layers.

ATofStrawTrackFinder This algorithm is an interface to the TofStrawTrack-Finder implemented at the Forschungszentrum Jülich. To use this algorithm properly you have to download and install the TofStrawTrack-Finder from the tof-home-page. The hits are fed into the track-finder and the found tracks are converted to the **typeCase**-format.

AAssignHitsToTracks This algorithm is useful, if a track-finder was applied, that did the track-search not on the complete detector but on a subset of layers. This algorithm adds the missing detector layers to the track using suspect-search.

AFindDecaysInTracks This algorithm is needed, when a track-finder was applied, but no specification was given whether the found tracks are primary or secondary. This algorithm defines the tracks into prompt, kink, vee, any-decay, adding neutral tracks if necessary.

- Post-tracking-calibration algorithms

  AapplyLRC To finally calibrate the TDCs, they have to be corrected for the signal-run-time in the detector, before reaching the photo-multipliers for example. This algorithm calculates the hit point of the tracks in their hit elements to a distance to readout. The correction is then a function dependent on this path. Several options can be chosen for correction-functions.

  ATofPixCorrection This algorithm does essentially the same as the one above, but can be applied already on pixel-level, as it is done in Jülich.

  APulsHeightCorrection Corrects the QDCs for the signal-run-path in the detector.

  ACalculateTrackProperties Calculates the betas out of the TDCs. Specify start- and stop-detectors; the mean is taken each.

- Other algorithms

  ACompareToGIN This algorithm helps to compare the analyzed Monte-Carlo simulation to the generated input values, leading to resolution, efficiency and acceptance.

  AWriteHistogramsToRootFile This algorithm produces an awful lot of histograms, that are written to file or can be viewed during run-time. It is still maintained, but the information contained here can be retrieved in a more elegant way using the trees of e.g. ATrackTreeOutput.

- ABetheBloch: A small class calculating the energy-loss of a particle in matter using the Bethe-Bloch-formula. It can be added to a material.

- ATDCcalibration: Big algorithm generating calibration for anything with a TDC. It does offset, walk, signal-run-correction, Barrel-pixel-position and also some TDC-cuts. Needs pixels and/or tracks.

D.5 Meta-code

Per definition meta-code is code that generates code.
To make installation of new algorithms and shapes more comfortable and less error

generating, this part of the analysis-program has been developed. It inserts commands into all necessary files to include the new part of the analysis.
Specifications have to be made about the constructor of the class in question and in the case of algorithms also about SIGNAL-SLOT-connections and GUI-interactions. These can be saved to file, reread and executed later.
It contains search functions for every part of the installation process to prevent double installation.
Note: This part of the analysis-program does not prevent you from programming the new class. It only makes it (much) easier to install it into **typeCase**.

D.5.1 algorithmWriter

The class algorithmWriter provides all necessary functionality to insert a new algorithm into the analysis package. Apart from the getters and setters, that provide the necessary information about the algorithm in question, like header- and source-file-names, class name, constructor call, connections to the main algorithm class (sec. D.1), informations for IO-algorithms (widget header and source, etc.) it provides the following functions (table D.1).
Operators to write the properties to file and reread them from file are supplied. These files can be traded along with the new algorithm to other sites.

void	insertToMakeFile()		
void	insertToIOMakeFile()		
void	insertToHeader()		
void	insertToIOHeader()		
void	insertToIO()		
void	insertToAnalyser()		
bool	headerAvail()	const	included in common header
bool	makeAvail()	const	included in makefile
bool	libAvail()	const	included in library
bool	analyserAvail()	const	ready for analysis
bool	ioAvail()	const	ready for IO
bool	ioMakeAvail()	const	included in IO-makefile
bool	ioHeaderAvail()	const	included in IO-header
int	getNextID()		
int	getInstalledID()		

Table D.1: These methods check up to which state the algorithm is already installed and are able to insert the necessary code into the analysis-code-files and make-files.

D.5.2 shapeWriter

The class shapeWriter works quite similar, though it is a lot simpler, the shapes having a much less complicated calling structure than the algorithms. Here again some getters and setters provide you with the information necessary to install the new shape (see table D.2).
Operators to write the properties to file and reread them from file are supplied. These files can be traded along with the new shape to other sites.

void	insertToMakeFile()		
void	insertToHeader()		
void	insertToShapes()		insert to analyzer
bool	shapeAvail()	const	ready for analysis
bool	headerAvail()	const	included in common header
bool	makeAvail()	const	included in makefile
bool	libAvail()	const	included in library

Table D.2: These methods check up to which state the shape is already installed and are able to insert the necessary code into the analysis-code-files and make-files.

D.6 the plot-reaction classes

To generate the pretty plots seen in chapters 7 and C, several classes had to be designed and implemented.
There are some help classes, like *colorScheme*, *filesManager*, *histoProperties* and *texFileMakeup*, that help to define the color scheme of a single histogram type, to manage the files for input and output, histogram properties, like name, title, axis titles, etc and the way a LaTeX-file containing all plots and information of interest should be made off. Additionally there are the classes *histoStack1* and *histoStack2*, that contain all the histograms that should be plotted into one picture and do the actual scaling and plotting.
When looking closer to a specific reaction, you may be interested not only in the total cross section, but also in the differential cross sections and you want to see the quality of your data and the resolution of your setup. For this purpose the plot-reaction classes have been developed.
The main purpose of these classes is to define and fill histograms for a specific reaction. But they can also apply cuts, correct data with simulations, write a TeX-file containing all interesting plots and define and fill a tree (the “PreCutTree”-format) with all values that are filled into the histograms.
If you think: That’s a tough task to define a class that alone can do this for all possible reactions; you are right.
What is defined – as you have already seen for algorithms and shapes – is an interface, an abstract class that defines the functions but does not implement them.

There are two classes derived from this base classes so far: implementing the plotting a.s. for the reactions $pp \to pK^+\Lambda$ and $pp \to pn\pi^+$.

D.6.1 colorScheme

The struct colorScheme is a rather small one, containing information about line-, fill- and marker- -color, -style and -size. For information about the color-pallete and styles visit the root web-page (eg. `http://root.cern.ch/root/html524/TAttMarker.html`). The most important functions are:

D.6.2 texFileMakeup

The class texFileMakeup is there to make a nice LaTeX-file containing the plots of the desired reaction. It capsules the number of pages for the one-dimensional histograms. For each page it saves the number of rows and columns and for each of these slots the ID of the plot to draw in this space. The "sub"-string defines the aspect ratio of the plots: "a" for golden-ratio, "b" for squared.

D.6.3 histoProperties

This small struct combines the properties of one histogram. It contains the name (which will not be displayed but is used to write to root-file), the title, the title of the axes, the number of bins for the x-axis (as well as for the y-axis for 2D-histograms), the ranges of the axes (xMin to xMax, yMin to yMax), the position of the title box in canvas NDC[2]. It has a flag that is true, if only raw data and simulation through detector shall be plotted, showing e.g. detector resolution, and that is false for observable histograms, where corrected data and pure simulations are shown. The *tp* variable holds the dimension of the histogram and is negative if the axes-ranges can be overwritten by an input from file. The *scaling* parameter gives the relative scaling of the histogram in the picture, >1 leaves space on top, <1 cuts away the uppermost point(s).

D.6.4 filesManager

The filesManager class is a class containing only static variables and methods. It is no use defining an object of that type.
It keeps the information on the files for the different data sets you may want to plot. The following possible data-sets are defined:

- **Data**: raw data
- **Simulation**: simulated events
- **Through Detector**: simulated events that passed a virtual detector and were reconstructed as data was.

[2]normalized canvas coordinates: (0,0) at lower left corner, (1,1) at upper right.

- **Corrected**: recently added it is now possible to read the corrected data histograms from file.

Of type Data or Corrected you may have only one set, but you may have any number of sets of Simulations and Through Detector histograms, assigning Through Detector to a Simulation-set to make it possible to calculate an efficiency out of that combination.
Each Simulation is identified by a name or an ID.
Additionally it is possible to define a *marking* that is added to a LaTeX-file in the footer of each page. Many methods ask for a character parameter type, that can be "d" for data, "c" for corrected data, "s" for simulation or "m" or "t" for Through Detector sets (case doesn't matter). The most important methods are:

D.6.5 histoStack1 and histoStack2

The drawings in chapters 7 and C contain in one picture a stack of histograms. There is the yellow-filled phase-space, the black dots for corrected data and many colored lines, that correspond to different simulations. They are all stored in the histoStack-classes, that not only do the drawing and correct scaling of the different histograms, but also assign the color-schemes and calculate efficiency and corrected histograms.

histoStack

In principle one base-class *histoStack* would have been enough, perhaps a derived class for the 2D-histograms, because the error-propagation has to be done by hand there and also the draw-method is different. But unfortunately the root-classes do not permit polymorphism and therefore the histogram-variables as well as the getters and setters had to be implemented for both classes separately.
Each histogram-stack contains a property-variable (sec. D.6.3), that defines the axes-ranges and titles. It also contains for each histogram in the stack the root-drawing-option and a relative scaling factor. There is a flag for logarithmic (y-axis for 1D, z-axis for 2D) plotting, the color for applied cuts and hits (e.g. Λ-mass in fig. 7.5(a)) and most important which simulation to use for correction and which simulation-set corresponds to which Through-Detector-set.

histoStack1

This class contains a stack on 1-dimensional histograms, with two colorSchemes for each set (one common, one for black-and-white printing). Each object contains a list of x-positions for vertical lines (first two will be of *cutColor*, the rest will have *hintColor*). There exists the possibility to normalize the histograms not to common integral, but to have a common height at some position; for this purpose you can set the special normalizer-flag and set the x-position at which the histograms all

have the same height.
It can produce corrected data-histograms if all necessary histograms (raw-data, simulation and corresponding simulation-through-detector) are available; it applies the right colorSchemes and scales and draws the histograms to a selected canvas.

histoStack2

The class histoStack2 contains a stack of histograms of type TH2F. As the 1D-version, this class can also produce corrected-data-histograms as well as draw the histograms to canvas. But for 2D-histograms, no colorScheme is needed, all histograms are drawn with the option "col". This reduces the disc-space of the resulting ".eps"- or ".ps"-files and time to draw enormously compared to scatter-plots.

D.6.6 Reaction-types

The reaction-types do the filling of the histograms for the differential cross section spectra, they can apply cuts, define a tree-format containing all interesting information about this reaction, file-IO for data and methods to plot the graphs and do some nice LaTeX-file to include the plots, say something about the data-sources and calculate the total cross-section.
For full flexibility an abstract base class *reaction_type* was defined. It implements the general functions of these classes, as well as static functions to allocate installed plot-reactions. The general –implemented– methods are:

- **void setBeamMomentum(float value)** Set the momentum of the beam-particle. There are 2 particles in the initial state: target-proton and beam-proton. Target is at rest. In units of GeV
- **float beamMomentum()const** Returns the momentum of the beam-particle in GeV.
- **void setCrossectionPerEntry(float value,float error)** Sets the cross-section per entry in the data-histograms (probably determined using some reference reaction eg. pp-elastic-scattering) along with its error in mBarn.
- **float crossectionPerEntry()const** Returns the cross-section for each event in data in mBarn.
- **float crossectionPerEntryError()const** Returns the error for the cross-section per entry-value.
- **reaction_type(string iname)** Constructor. Won't ever be called directly but only via derived classes.
- **virtual reaction_type()** Destructor.
- **string name()const** Returns the name of the reaction. Set variable *fname* in your constructor.
- **int nParticles()const** Returns the number of particles, that will be displayed for this reaction.
- **int nAngles()const** Returns the number of angle-parameters that will be saved in PreCutTree-format for this reaction.

- **int nMissingMasses()const** Returns the number of missing-mass-parameters that will be saved in PreCutTree-format for this reaction.
- **int nInvariantMasses()const** Returns the number of invariant-mass-parameters that will be saved in PreCutTree-format for this reaction.
- **int n1D()const** Returns the number of 1D-histograms defined for this reaction.
- **int n2D()const** Returns the number of 2D-histograms defined for this reaction.
- **void setUse(int pos, bool value)** Set the use of the pos^{th} cut.
- **static vector<string> reactionNames()** Static function that returns the names of all defined reactions.
- **static reaction_type *getReaction(const string& name)** Returns a pointer to an object of the reaction of the specified name. Returns the NULL-pointer if name is not valid. Take care that you delete the object after use.
- **void resetStruct(trackstruct& stru)** Sets the values of the structure to default values.
- **void initWriteOutTree(TTree *tree, writeOutStruct &str, string pre)** Initializes a tree for the PreCutTree-format.
- **bool getHistos(TFile *f, TH1F ***histos1, TH2F ***histos2, string pres)** Retrieves the histograms for this reaction from file and saves them in the arrays for 1D-histograms and 2D-histograms. Specify a string that can be prepended to the name of the histograms.
- **void fillWriteOutTree(TTree* tree, writeOutStruct &str,TTree* inTree)**
- **string precuttree_eventList_quality(bool use**[20]**, int cuton, float* masses)** If you have a tree in PreCutTree-format, you can apply cuts easily using the generation of a TEventList-object with the TTree::Draw-method. This method returns the cut-string for all events where at least one of the used cuts passed.
- **string precuttree_eventList_observables(bool use**[20]**, int cuton, float* masses)** If you have a tree in PreCutTree-format, you can apply cuts easily using the generation of a TEventList-object with the TTree::Draw-method. This method returns the cut-string for all events where all of the used cuts passed.
- **bool getPreCutTreeLeaves(TLeaf ** hdr, TLeaf ** lvs, TTree *tree, string pre)** Extracts from a tree in trackTree-format the leaves of the hearder
- **void refillStruct(TLeaf** hdr, TLeaf** leaves, writeOutStruct &str, int offsetPos)** Copies the current content of the leaves (retrieved via **getPreCutTreeLeaves()**) to the struct.
- **void addEventsToPrecuttree(TTree *inTree, TTree* outTree, writeOutStruct &str,string pre)** If *inTree* and *outTree* are both of PreCutTree-format this method appends the entries from *inTree* to *outTree*. *pre* can be some string that is prepended to the leaf-names in the *inTree*.
- **void getAllMyDataFromFile(histoStack1 *histo1D, histoStack2 *histo2D, bool fitted, bool show=true)** This very potent method uses all files defined in class *filesManager* for each of the data-sets defined in *filesManager* and

histoStack and reads the content from the input files (possibly applying cuts) and saves it in the defined output-files and most important in the histostack-arrays, that have to have the right dimension for this reaction.

- **void plotToDirectory(histoStack1 *histo1D, histoStack2 *histo2D,string pathname)** For each 1D-hisogram this method produces one .eps-file in the specified directory containing the canvas-Print()-output, with the drawn histogram-stacks. The numbering of the files is successive:
  pathname/drawings_$i_$ASPECT.eps. The $ASPECT can be "a" for an aspect-ratio of $\sqrt{2}$ or "b" for an aspect-ratio of 1.
  2D-histograms will be plotted each histogram in a separate file:
  pathname/drawings_$(i+$N_{1D}$)_$j_b.eps, with

0	raw data
1	corrected data
2	efficiency
$3 \cdots N_{sim}$	simulations
$3+N_{sim} \cdots 3+N_{sim}+N_{throughDetector}$	simulation through detector

- **void makeTexFile(string pathname, int nData, int nSim, int nThrough)** Generates a LaTeX-file that contains all plots and other available information for this reaction and the input-data. You will have to apply LaTeXthis file later.
- **void fillCompareTree(TTree* tree, writeOutStruct &inStruct, writeOutStruct& outStruct, TChain* inTree, TChain* outTree, int tp1, int tp2, bool ch1, bool ch2,string pre1,string pre2)** This method will combine a simulation-data-set (*inTree*) and a simulation-through-detector-set (*outTree*) to a single PreCutTree (*tree*). The trees can have different format (*tp1*, *tp2*, where 0 means track-tree-format, 1 PreCutTree-format), possibly be TChains (*ch1*, *ch2*) and have some string prepended to the leaves in the trees. The only events written will be the ones where both trees have an entry for the same event number.
- **void produceCompareTree(vector<string> filesMC, vector<int> typesMC, vector<string> filesGIN, vector<int> typesGIN, string outfile)** Produce a tree conaining simulation and simulation-through-detector-data. Calls **fillCompareTree(...)**.
- **void add_pkl(TTree *tree, TH1F ***histos1, TH2F ***histos2, Float_t initM, float chiCut, bool invalues, bool converged, int filetype)** Adds events from a track-tree to a set of histograms, applying cuts. Uses **leafToStruct(...)**, **observablesFill(...)**, **qualityFill(...)**.
- **void fillHistosFromPrecuttree(TTree *tree,TH1F ***histos1, TH2F ***histos2,bool invalues,bool converged, string preH, string preS)** Adds events from a PreCutTree to a set of histograms applying defined cuts. Uses **leafToStruct(...)**, **observablesFill(...)**, **qualityFill(...)**.
- **void fillData2PTtrees(istream &input, writeOutStruct& Ptracks, TTree *Ptree, trackstruct &Ttracks, TTree *Ttree, const momentum4D &init-**

System) Read from a dat-input-stream. This will be filled to a track-tree and a PreCutTree. Uses **event2Pstruct(...)** and **copyPTstructs(...)**.

- **void fillData2histosTree(istream &input, writeOutStruct& tracks, trackstruct &Ttracks, TTree *tree, const momentum4D &initSystem,TH1F*** histos1, TH2F*** histos2,int which, int cuton, bool converged)**Read from a dat-input-stream. This will be filled to a track-tree and the histograms. Uses **event2Pstruct(...)**, **copyPTstructs(...)**, **observablesFill(...)**, **qualityFill(...)**.
- **void fillData2histos(istream &input, writeOutStruct& tracks, const momentum4D &initSystem,TH1F*** histos1, TH2F*** histos2,int which, int cuton, bool converged)** Read from a dat-input-stream. This will be filled to histograms. Uses **event2Pstruct(...)**, **observablesFill(...)**, **qualityFill(...)**.
- **void fillData2Ptree(istream &input, writeOutStruct& tracks, TTree *tree, const momentum4D &initSystem)** Read from a dat-input-stream. This will be filled to a PreCutTree. Uses **event2Pstruct(...)**.

To specify the individual behavior for each reaction, you have to derive new classes from *reaction_type* implementing the purely virtual methods and perhaps defining new methods and variables. Each of the following methods have to be overwritten (using keyword *virtual*):

- **int* getIDs()** Allocate an array of integer, fill it with the GEANT-IDs of the particles in this reaction and return the pointer to it. Don't return a pointer to a local object.
- **float *getMasses()** Allocate an array of floats, fill it with the masses of the particles in this reaction and return the pointer to it. Don't return a pointer to a local object.
- **void setBaseUse()** Set the base-use array for the 20 cuts with default values.
- **vector<string> getCutNames()** Return a vector of strings with the names of the individual cuts. Will be used in GUI and for display.
- **string makeComment(bool use[20], int nCuts)** Returns a LaTeX-string with all the cuts, that are currently used. Will be added as caption to the graphs on each page of the LaTeX-file.
- **void fillCuts(writeOutStruct &stru, int cuton, int nCuts, bool *cuts, float *masses, bool converged)** Assuming the current event is filled into *stru*, the *cuts*-array shall be filled with boolean values whether the corresponding cut was passed. *cuton* defines the index to use for the cuts of the *stru* arrays (0: unfitted, 1: fitted).
- **void getStrings(string strings[20], bool use[20], int cuton, float* masses)** Since in PreCutTree-format all important values have already been calculated and stored, an event-list can be generated using a cut string. This function shall return the strings for the individual cuts so they can be used in the TTree::Draw()-method.

- **void qualityFill(TH1F*** histos1, TH2F*** histos2, writeOutStruct &stru, int which, int cuton, int nCuts, bool *cuts, bool use[20], float *masses, bool converged, float weight=1)** This method calls the *fillCuts(...)*-method and shall fill all histograms showing the quality applying all cuts but the one shown in this histogram.
- **void observablesFill(TH1F*** histos1, TH2F*** histos2, writeOutStruct &stru, int which, float weight=1)** This method doesn't care about cuts, that's done in the method calling this one. It simply fills all histograms defined as observable histograms –the ones that will be drawn in nice color.
- **void addDrawingLines(histoStack1 *h1, histoStack2 *h2, float *masses)** Perhaps you want to indicate cuts you are applying, or guide the eye by putting a line to a position where a peak shall be. This is the function to add some line to the corresponding histogram-stack.
- **void setHistoPropertiesDefault(const momentum4D &inputmomentum)** This method defines the default description of the histograms you want to fill and draw. You can read the properties from file, but here in this method you define the default, that doesn't need any files.
  Each histogram-property is defined by name, title, axes-titles, binning of the axes (will be multiplied by 10), axes-ranges, the position of the title-box and the draw-mode (true for observable, false for quality-draw).
- **bool leafToStruct(TLeaf** leav, writeOutStruct &stru, momentum4D &cms, momentum4D &inM, momentum4D moment[4][4], momentum4D inter[4], momentum4D Pcms[2][3], momentum4D Jmoment[2][3][3], momentum4D jbm[2][3], float *mass,int id[4],vector3D &lDir)** Depreciated function, see description for method below.
- **bool leafToStruct(TLeaf** leav, writeOutStruct &stru, momentum4D &cms, momentum4D &inM, float *mass,int *id)** This method fills the struct *stru* with all properties, that will be drawn later from the leaves of track-tree-format-leaves. Very basic method, take care of this one!
- **bool trackTreeStruct2WoStruct(trackstruct &tr, writeOutStruct& stru, momentum4D &cms, momentum4D &inM, float *mass)** For this method you copy the basic properties from a track-tree-struct to a PreCutTree-struct and calculate all important properties for this reaction.
- **void event2Pstruct(istream &input, writeOutStruct& tracks, const momentum4D &initSystem, int *particleids)** This method reads the event from an dat-input-stream and converts it into the PreCutTree-struct, calculating not only center-of-mass values and velocity but also invariant- and missing-masses and angles.
- **void setTexFile(texFileMakeup &makeup)** With this method you define how your LaTeX-file shall look in the end. How many pages and how many rows and columns shall the individual page have. Which plots to show in which slot and with which a aspect-ratio. This is what you define here.

Class D.1 Definition of class histoStack.

```
class histoStack
{
 protected:
  histoProperties property;
  bool flogarithmic;
  static int fcorrectWith;
  static vector<int> mcAssign;
  static int nBins;
#ifdef USE_LEGEND
  TLegend* legend;
#endif
 public:
  static vector<float> drawSimulation;
  static vector<float> drawMC;
  static float drawData;
  static float drawCorrection;
  static float drawEfficiency;
  static int cutColor;
  static int hintColor;
  static vector<string> simulationOption;
  static vector<string> mcOption;
  static string dataOption;
  static string correctedDataOption;
  static string efficiencyOption;
  static vector<string> simulationName;
 public:
  void setProperties(const histoProperties &hs);
  string name()const;
  string title()const;
  static int bins();
  static void setBins(int b);
  static int correctionSim();
  static int correctionMC();
  static string option(char tp, string iname);
  static void setOption(char tp, string iname, string c);
  static float drawFactor(char tp, string iname);
  static void setDrawFactor(char tp, string iname, float v);
  void setLogarithmic(bool v);
  bool logarithmic()const ;
  static int correctWith();
  static void setCorrection(string name);
  static void setCorrection(int num);
};
```

Class D.2 Definition of class histoStack1.

```
class histoStack1:public histoStack
{
 public:
  static vector<colorScheme> simulationColor;
  static vector<colorScheme> mcColor;
  static colorScheme dataColor;
  static colorScheme correctedDataColor;
  static colorScheme efficiencyColor;
  static vector<colorScheme> simulationColorALT;
  static vector<colorScheme> mcColorALT;
  static colorScheme dataColorALT;
  static colorScheme correctedDataColorALT;
  static colorScheme efficiencyColorALT;
 public:
  static void cleanStuff();
  static void setDataColor(colorScheme c,string drawOption);
  static void setDataColor(colorScheme c, colorScheme ac,string drawOption);
  static void setDataColorALT(colorScheme c);
  static void setCorrectedDataColor(colorScheme c,string drawOption);
  static void setCorrectedDataColor(colorScheme c, colorScheme ac,string drawOption);
  static void setCorrectedDataColorALT(colorScheme c);
  static void setEfficiencyColor(colorScheme c,string drawOption);
  static void setEfficiencyColor(colorScheme c, colorScheme ac,string drawOption);
  static void setEfficiencyColorALT(colorScheme c);
  static void addSimulationColor(string name, colorScheme c,string drawOption);
  static void addSimulationColor(string name, colorScheme c, colorScheme ac, string drawO
  static void addSimulationColorALT(string name, colorScheme c);
  static void addMCColor(string name, colorScheme c, colorScheme ac,string drawOption);
  static void addMCColorALT(string name, colorScheme c);

  static colorScheme color(char tp, string iname);
  static colorScheme colorALT(char tp, string iname);
  static void setColor(char tp, string iname, colorScheme c);
  static void setColorALT(char tp, string iname, colorScheme c);

  histoStack1();
  ~histoStack1();
  histoStack1(const histoProperties& hs);
  TH1F *dataHisto()const;
  TH1F *mcHisto()const;
  TH1F *simHisto()const;
  int nHistograms()const;
  void setSpecialNormalizer(bool v);
  void setNormalizerX(float x);
  void addLine(float xvalue);

  TH1F *makeHistogram(string pres);
  void setDataHistogram(TH1F* histo);
  void addSimulationHisto(TH1F* histo,string name,string pres);
  void addMChisto(TH1F* histo,string name,string pres);
  void addMChisto(TH1F* histo,int assignsTo,string pres);

  void addHistogram(string name, string pre, char tp, TH1F* histo);
  void setHistogram(int num, char tp, TH1F* histo);
  TH1F* histogram(int num, char tp);

  void correct();
  void draw(bool debugMode, float sigmaEntry);
  void update();
  void updateALT();
};
```

Class D.3 Definition of class histoStack2.

```
class histoStack2: public histoStack
{
 public:
  static void addSimulationOption(string name, string drawOption);
  static void addMCOption(string name, string drawOption)
  static void setDataOption(string drawOption);

  histoStack2();
  ~histoStack2();
  histoStack2(const histoProperties& hs);
  TH2F *dataHisto()const;
  TH2F *mcHisto()const;
  TH2F *simHisto()const;
  int nHistograms()const;

  TH2F *makeHistogram(string pres);
  void addHistogram(string name, string pre, char tp, TH2F* histo,
         bool draw=true);
  void setHistogram(int num, char tp, TH2F* histo);
  TH2F* histogram(int num, char tp);
  void setDataHistogram(TH2F* histo);
  void addSimulationHisto(TH2F* histo,string name,string pres,
         bool draw=true);
  void addMChisto(TH2F* histo,string name,string pres,
         bool draw=true);
  void addMChisto(TH2F* histo,int assignsTo,string pres,
         bool draw=true);

  string nameOfSet(int which);
  void add(float xvalue,float yvalue, float rx, float ry);
  void correct();
  bool draw(int which, float sigmaEntry);
  void update();
};
```

Class D.4 Definition of general part of the class reaction_type.

```
class reaction_type
{
 protected:
  int fN2D;
  int fN1D;
  int fNparticles;
  int fNangles;
  int fNmissingMasses;
  int fNinvariantMasses;
  int fkinfitNdF;
  float BEAM_MOMENTUM;
  float SIGMA_ENTRY;
  float SIGMA_ENTRY_ERROR;
  void copyPTstructs(trackstruct& tracks, writeOutStruct& stru);
  void getTrackTreeLeaves(TLeaf ** lvs,TTree *tree);
  TTree* getTreeFromFile(TFile *myfile, int &fileType);
  void checkLeaves(bool isChain, TChain* tree, int &testTree, int tp, TLeaf **leaves, TLeaf **headers,string pre);
  void event2momenta(istream &input, vector<vector3D> &momenta, vector<int> &ids, int& eventNumber, int &runNumber,
     int &trigger, float &chi, int &iterations);
  void event2momenta(istream &input, vector<vector3D> &momenta, vector<int> &ids, int& eventNumber, int &runNumber,
     int &trigger, float &chi, int &iterations, float &beam);
  void copyFromLeaves(trackstruct &tracks, TLeaf **leaves);
  bool baseUse[20];
  void setUse(bool uses[20]);
 public:
  histoProperties *HistogramDefinitionDefault;
  histoProperties *HistogramDefinition;
  float cutValueArray[20];
 public:
  void setBeamMomentum(float value);
  float beamMomentum() const;
  void setCrossectionPerEntry(float value,float error)
  float crossectionPerEntry() const;
  float crossectionPerEntryError() const;
  reaction_type(string iname);
  virtual ~reaction_type();
  string name() const;
  int nParticles() const;
  int nAngles() const;
  int nMissingMasses() const;
  int nInvariantMasses() const;
  int n1D() const;
  int n2D() const;
  void setUse(int pos, bool value);
  static vector<string> reactionNames();
  static reaction_type *getReaction(const string& name);

  void resetStruct(trackstruct& stru);
  void initWriteOutTree(TTree *tree, writeOutStruct &str, string pre);
  void fillData2PTtrees(istream &input, writeOutStruct& Ptracks, TTree *Ptree, trackstruct &Ttracks, TTree *Ttree,
     const momentum4D &initSystem);
  void fillData2histosTree(istream &input, writeOutStruct& tracks, trackstruct &Ttracks, TTree *tree,
     const momentum4D &initSystem,TH1F*** histos1, TH2F*** histos2,int which, int cuton, bool converged);
  void fillData2histos(istream &input, writeOutStruct& tracks, const momentum4D &initSystem,TH1F*** histos1,
     TH2F*** histos2,int which, int cuton, bool converged);
  void fillData2Ptree(istream &input, writeOutStruct& tracks, TTree *tree, const momentum4D &initSystem);
  bool getHistos(TFile *f, TH1F ***histos1, TH2F ***histos2, string pres);
  void fillWriteOutTree(TTree* tree, writeOutStruct &str,TTree* inTree);
  void add_pkl(TTree *tree, TH1F ***histos1, TH2F ***histos2, Float_t initM, float chiCut, bool invalues,
     bool converged, int filetype);
  string precuttree_eventList_quality(bool use[20], int cuton, float* masses);
  string precuttree_eventList_observables(bool use[20], int cuton, float* masses);
  bool getPreCutTreeLeaves(TLeaf ** hdr, TLeaf ** lvs, TTree *tree, string pre);
  void refillStruct(TLeaf** hdr, TLeaf** leaves, writeOutStruct &str, int offsetPos);
  void fillHistosFromPrecuttree(TTree *tree,TH1F ***histos1, TH2F ***histos2,bool invalues,bool converged,
     string preH, string preS);
  void addEventsToPrecuttree(TTree *inTree, TTree* outTree, writeOutStruct &str,string pre);
  void getAllMyDataFromFile(histoStack1 *histo1D, histoStack2 *histo2D, bool fitted, bool show=true);
  void plotToDirectory(histoStack1 *histo1D, histoStack2 *histo2D,string pathname);
  void makeTexFile(string pathname, int nData, int nSim, int nThrough);
  void fillCompareTree(TTree* tree, writeOutStruct &inStruct, writeOutStruct& outStruct, TChain* inTree,
     TChain* outTree, int tp1, int tp2, bool ch1, bool ch2,string pre1,string pre2);
  void produceCompareTree(vector<string> filesMC, vector<int> typesMC, vector<string> filesGIN,
     vector<int> typesGIN, string outfile);
```

Class D.5 Definition of abstract part of class reaction_type.

```
  virtual int* getIDs()=0;
  virtual float *getMasses()=0;
  virtual void setBaseUse()=0;
  virtual vector<string> getCutNames()=0;
  virtual string makeComment(bool use[20], int nCuts)=0;
  virtual void fillCuts(writeOutStruct &stru,int cuton, int nCuts,
     bool *cuts, float *masses, bool converged)=0;
  virtual void getStrings(string strings[20], bool use[20],
     int cuton, float* masses)=0;
  virtual void qualityFill(TH1F*** histos1, TH2F*** histos2,
     writeOutStruct &stru, int which, int cuton, int nCuts,
     bool *cuts, bool use[20], float *masses, bool converged,
     float weight=1)=0;
  virtual void observablesFill(TH1F*** histos1, TH2F*** histos2,
     writeOutStruct &stru, int which, float weight=1)=0;
  virtual void addDrawingLines(histoStack1 *h1, histoStack2 *h2,
     float *masses)=0;
  virtual void setHistoPropertiesDefault(
     const momentum4D &inputmomentum)=0;
  virtual bool leafToStruct(TLeaf** leav, writeOutStruct &stru,
     momentum4D &cms, momentum4D &inM, momentum4D moment[4][4],
     momentum4D inter[4], momentum4D Pcms[2][3],
     momentum4D Jmoment[2][3][3], momentum4D jbm[2][3],
     float *mass,int id[4],vector3D &lDir)=0;
  virtual bool leafToStruct(TLeaf** leav, writeOutStruct &stru,
     momentum4D &cms, momentum4D &inM, float *mass,int *id)=0;
  virtual bool trackTreeStruct2WoStruct(trackstruct &tr,
     writeOutStruct& stru, momentum4D &cms, momentum4D &inM,
     float *mass)=0;
  virtual void event2Pstruct(istream &input, writeOutStruct& tracks,
     const momentum4D &initSystem, int *particleids)=0;
  virtual void setTexFile(texFileMakeup &makeup)=0;
};
```

Class D.6 Definition of class filesManager.

```
class filesManager
{
 public:
  static void setMarking(string m);
  static string marking();

  static void addDataFile(string filename, int fileType);
  static void addDataFile(string filename, int fileType, colorScheme c);
  static vector<string> dataInputFiles(vector<int> &types);
  static vector<string> dataOutputFiles(vector<int> &types);

  static void addSimulationFile(string filename, int fileType, string name);
  static void addSimulationFile(string filename, int fileType, string name, int to);
  static void addSimulationFile(string filename, int fileType, string name,
     colorScheme c);
  static void addSimulationFile(string filename, int fileType, string name, int to,
     colorScheme c);
  static int nSimulations(){return simulationFiles.size();}
  static vector<string> simulationInputFiles(int num, vector<int>& types);
  static vector<string> simulationOutputFiles(int num, vector<int>& types);

  static int nThroughDetectors(){return throughDetectorFiles.size();}
  static vector<string> throughDetectorInputFiles(int num, vector<int>& types);
  static vector<string> throughDetectorOutputFiles(int num, vector<int>& types);
  static void addThroughDetectorFile(string filename, int fileType, string name);
  static void addThroughDetectorFile(string filename, int fileType, int assi);
  static void addThroughDetectorFile(string filename, int fileType, string name,
      colorScheme c);
  static void addThroughDetectorFile(string filename, int fileType, int assi,
     colorScheme c);
  static void setThroughDetector(int pos, string name);
  static void setThroughDetector(int pos, int fromSim);
  static int correctionID();
  static int throughToSim(int pos);
  static void setCorrection(int c);

  static float draw(char tp, int num);
  static void setDraw(char tp, int num, float value);
  static int entries(char tp);
  static vector<string> InputFiles(int num, char tp, vector<int>& types);
  static vector<string> OutputFiles(int num, char tp, vector<int>& types);
  static string names(int num, char tp);
  static void removeSimulation(int num);
  static void removeThroughDetector(int num);
  static void addFile(string name, string filename, int filetype, char tp,
     colorScheme c);
  static void removeFile(int num, string filename, char tp);
  static bool hasType(int num, char tp, int filetype);
  static void readFromFile(string filename, const vector<int>& commandlineIDs,
     const vector<int>& commandlineTypes, const vector<string>& commandlineFiles,
     bool show=true);
  static void status(ostream &output=cout, bool texMode=false);
  static void clean();
};
```

Class D.7 Definition of struct histoProperties.

```
struct histoProperties
{
  string name;
  string title;
  string titleX;
  string titleY;
  int nBinsX;
  int nBinsY;
  float xMin;
  float xMax;
  float yMin;
  float yMax;
  float posX;
  float posY;
  bool drawAll;
  float scaling;
  int tp;
  histoProperties();
  histoProperties(string nme, string titl,
    string titlx, string titly, int nB, float fr, float to);
  histoProperties(string nme, string titl,
    string titlx, string titly, int nB, float fr, float to,
    float px, float py);
  histoProperties(string nme, string titl,
    string titlx, string titly, int nB, float fr, float to,
    int nB2, float fr2, float to2);
  histoProperties(const histoProperties& pr);
  histoProperties& operator=(const histoProperties& pr);
};
```

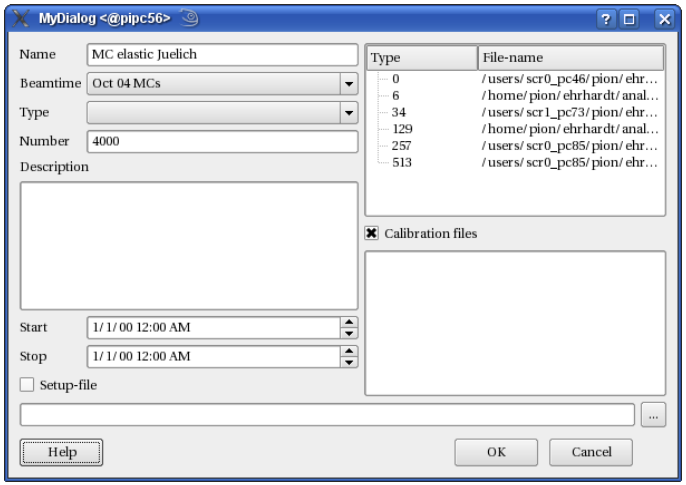
Class D.8 Definition of class texFileMakeup.

```
class texFileMakeup
{
public:
  texFileMakeup();
  ~texFileMakeup();
  void setDefinition(vector<pair<int,int>> rowsAndColumnsOnPages,
     vector<int> pages2D);
  int nPages()const;
  int nRows(int page)const;
  int nColumns(int page)const;
  string subString(int page)const;
  int ID(int page, int row, int column)const;
  int n2Dpages()const;
  int ID2d(int page)const;
  void setRows(int page, int value);
  void setColumns(int page, int value);
  void setSubString(int page, string value);
  void setID(int page, int row, int column, int value);
  void setID2d(int page, int value);
  void setPage(int page, int id1, int id2, int id3, int id4,
     int id5=-1, int id6=-1, int id7=-1, int id8=-1);
};
```

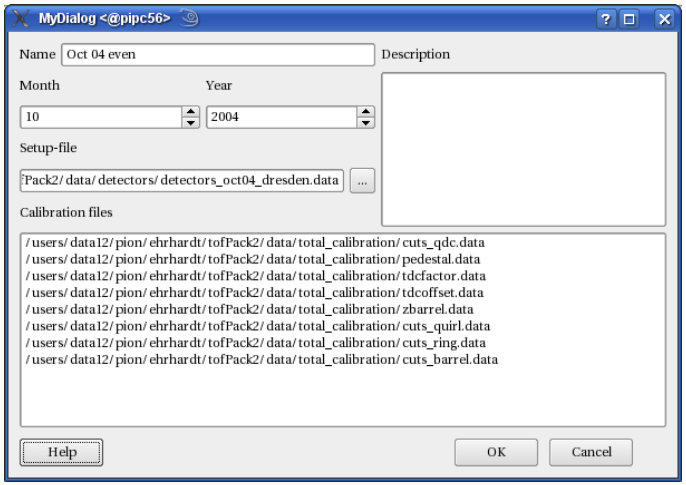
Class D.9 Definition of struct colorScheme.

```
struct colorScheme
{
  int lineColor;
  int lineStyle;
  int fillColor;
  int fillStyle;
  int markerColor;
  int markerStyle;
  float size;
  colorScheme(int lc, int ls, int fc, int fs, int mc, int ms, float mz);
  colorScheme();
  colorScheme& operator=(const colorScheme&c);
  colorScheme (const colorScheme&c);
  void applyToHisto(TH1F* histo, const histoProperties &hp);
  bool operator==(const colorScheme&c);
};
```

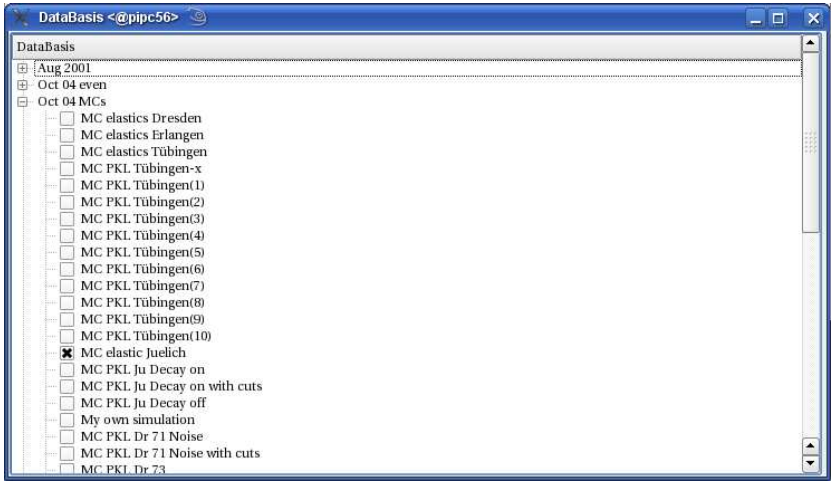
D.7 Picture Gallery



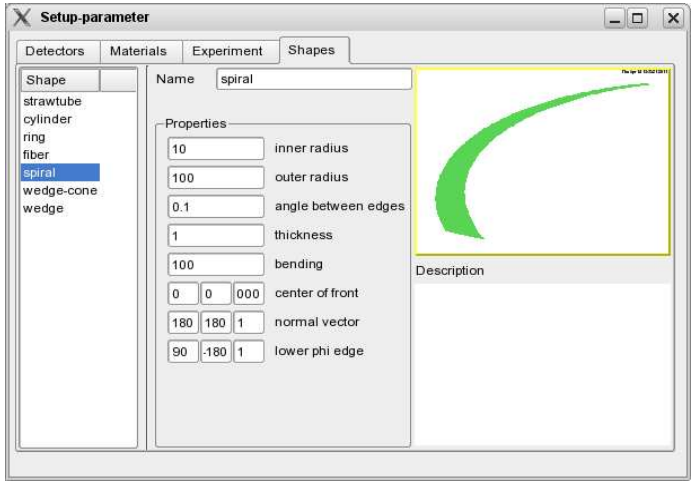
(a) Run-widget



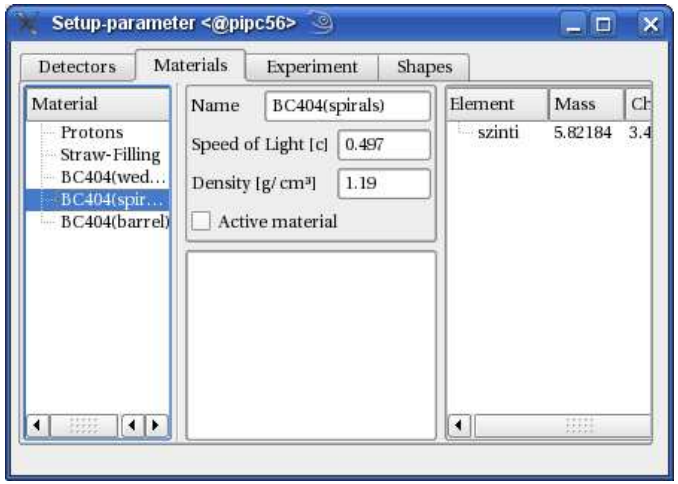
(b) Beam-time-widget



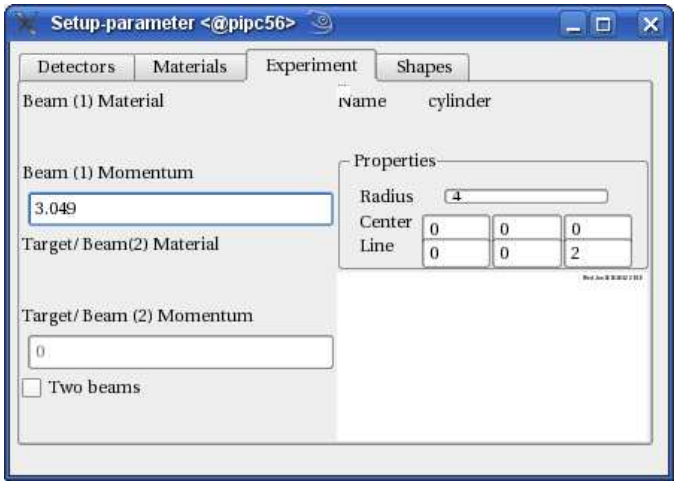
(c) Data-Basis-widget



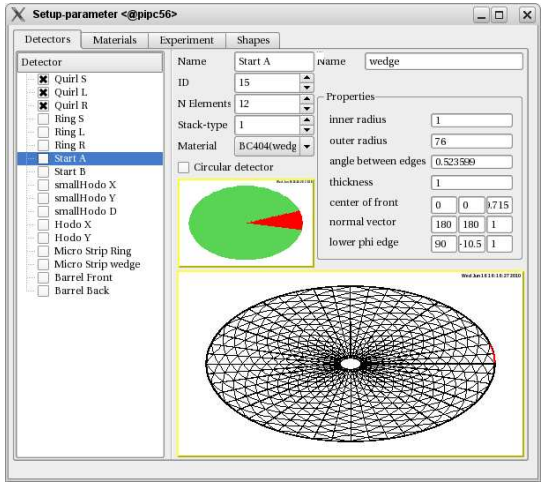
(d) Default-shapes-widget



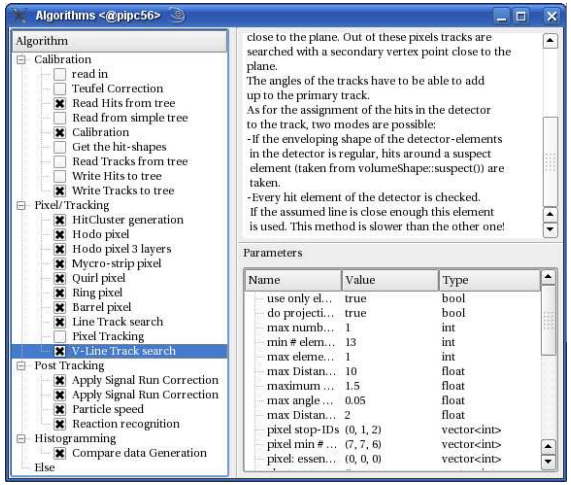
(e) Material-widget



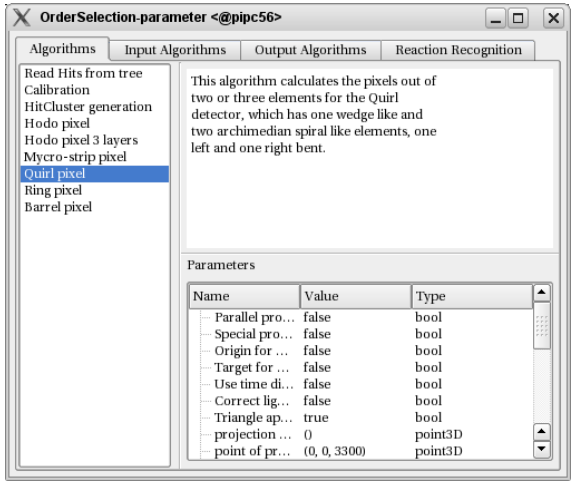
(f) Experiment-widget



(g) Detector-Geometry-widget

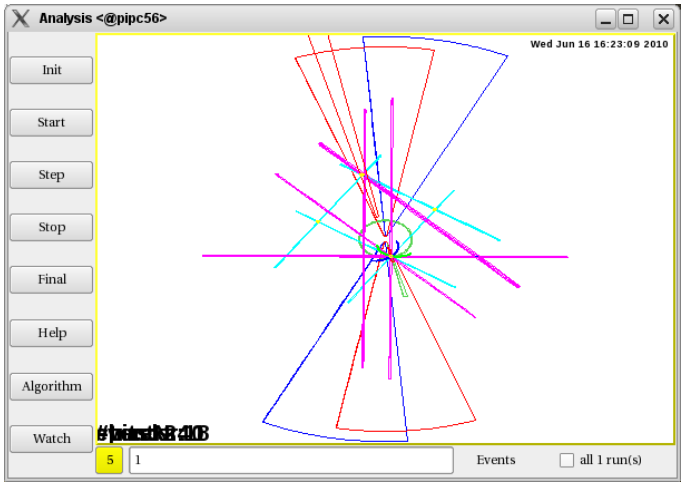


(h) Algorithms-widget

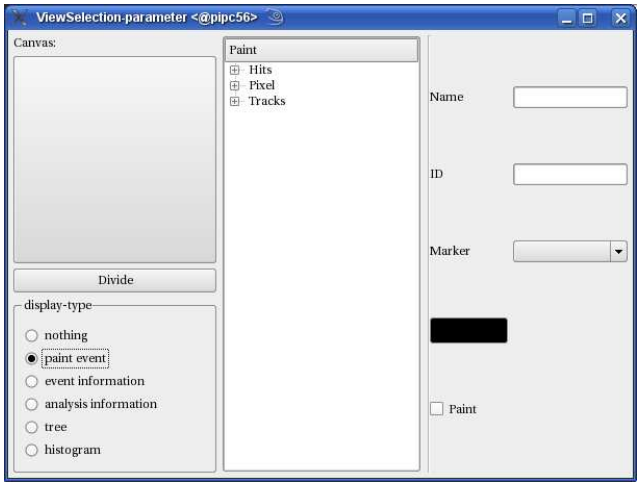


(i) Analysis-order-widget

(j) Input-Algorithm-widget

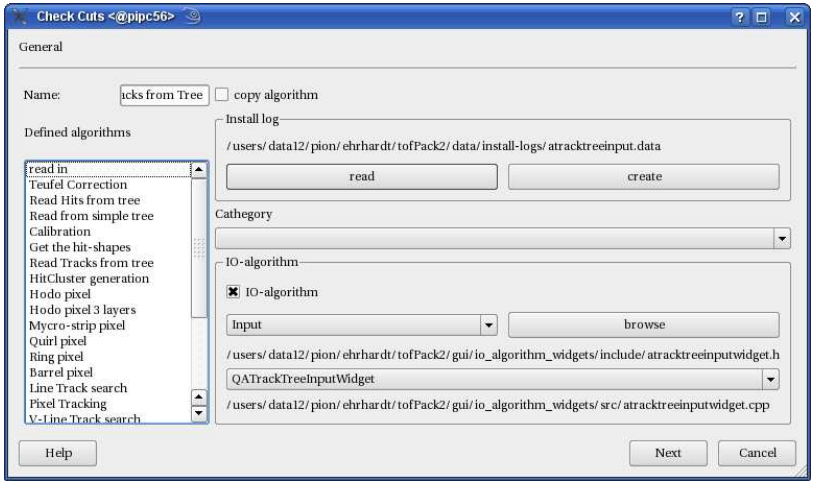


(k) Analysis-widget

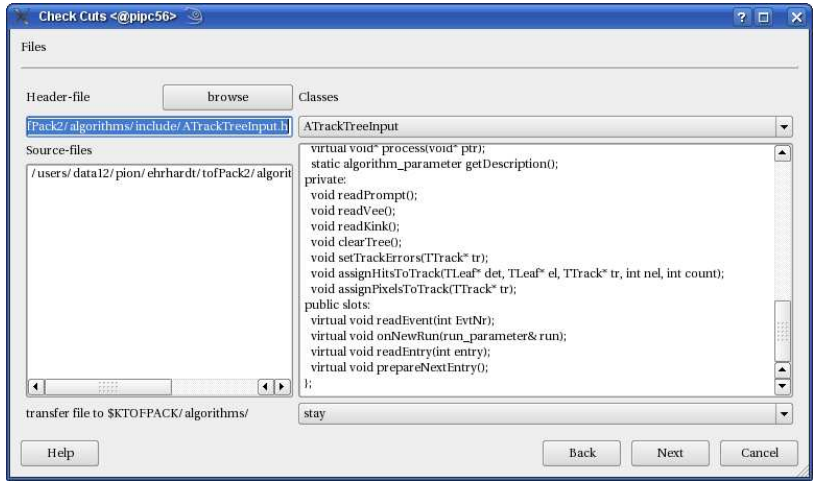


(l) View-Selection-widget

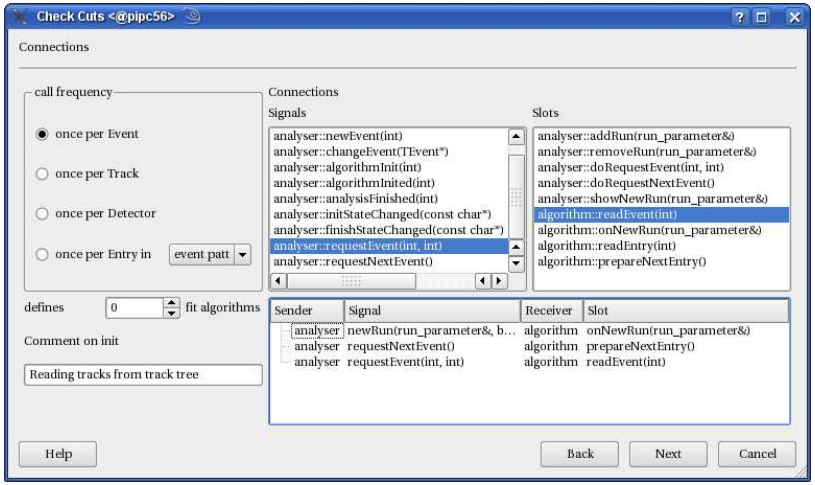
Figure D.5: Analysis widgets



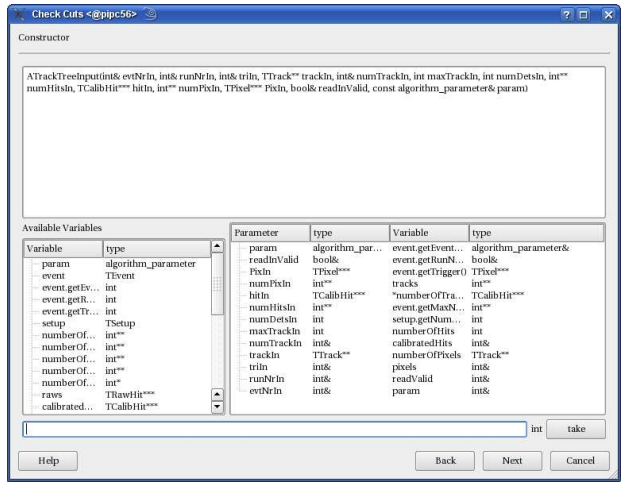
(a) Algorithm-installation page 1



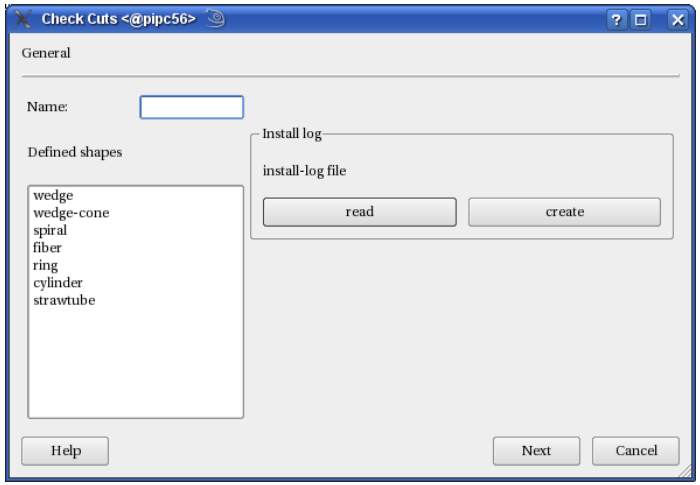
(b) Algorithm-installation page 2



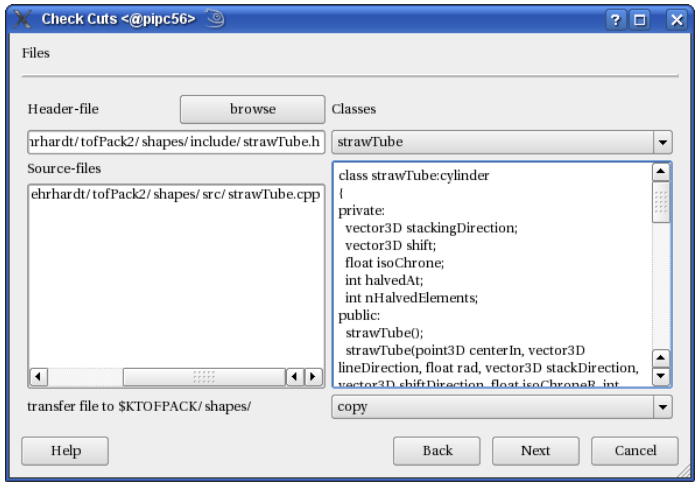
(c) Algorithm-installation page 3



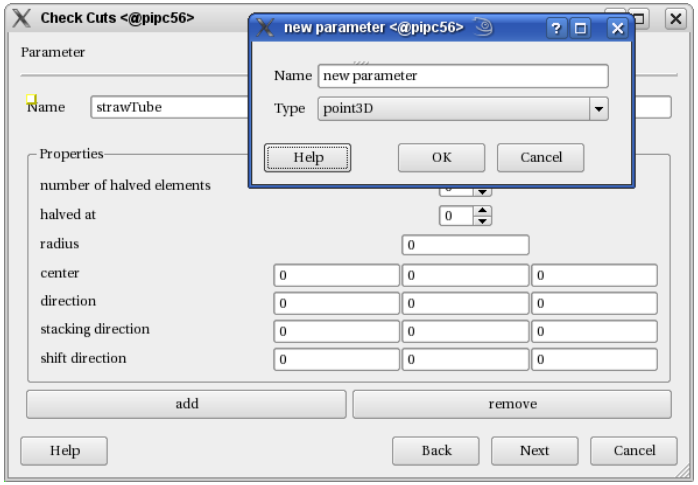
(d) Algorithm-installation page 4



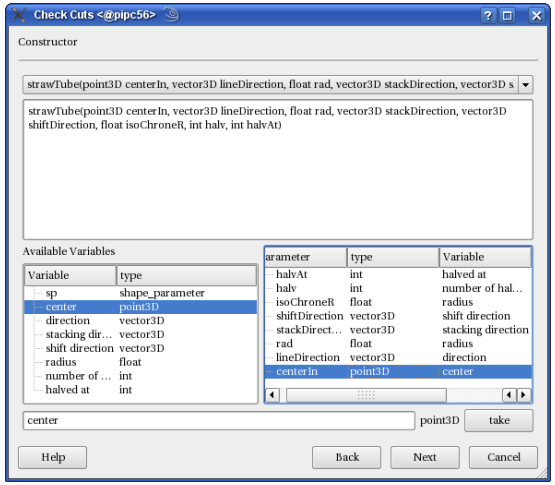
(e) Shape-installation page 1



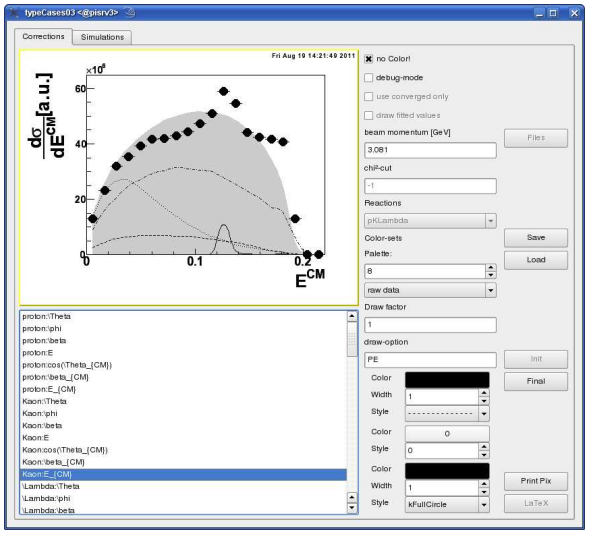
(f) Shape-installation page 2



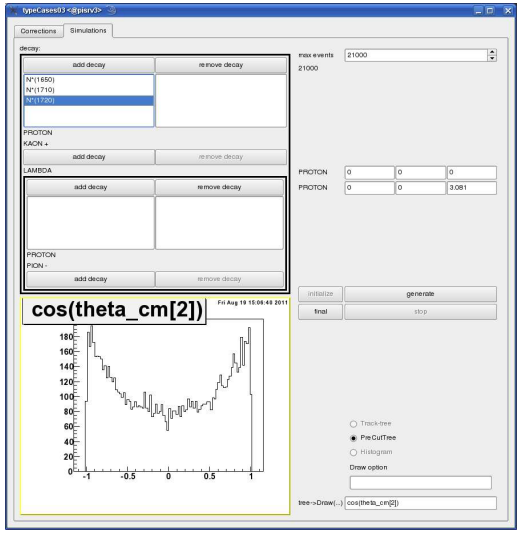
(g) Shape-installation page 3



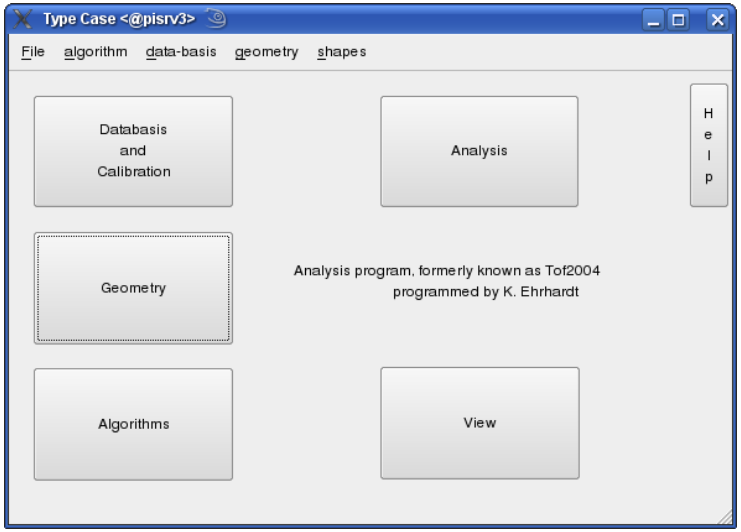
(h) Shape-installation page 4



(i) correction performing widget



(j) simulation widget



(k) main widget

(l) detector widget

Appendix E

Detector dimensions and materials

E.1 Quirl detector

Straight layer: wedges, Left and right layers: spiral.

elements straight	48
elements left bent	24
elements right bent	24
inner radius	42 mm
outer radius	580 mm
thickness	5 mm
bending	184.62 mm/rad
read out	QDC&TDC
zPosition S	3356mm
zPosition L	3367mm
zPosition R	3378mm
phi of 1st element	0

E.2 Ring detector

Straight layer: wedges, Left and right layers: spiral.

elements straight	96
elements left bent	48
elements right bent	48
inner radius	568 mm
outer radius	1540 mm
thickness	5 mm
bending	618.807 mm/rad
read out	QDC&TDC
zPosition S	3317.38 mm
zPosition L	3326.38mm
zPosition R	3335.38mm
phi of 1st element	0

E.3 Barrel detector

One wedge-cone layers. Readout at both sides.

elements	96
radius base	1553.5 mm
radius at cut	1488.5 mm
length of cone-stump	2854 mm
thickness	15 mm
read out	QDC&TDC
zPosition base	367 mm
phi of 1st element	0deg

E.4 Start detector

Two wedge-layers.

elements A	12
elements B	12
inner radius	1 mm
outer radius	76 mm
thickness	1 mm
read out	QDC&TDC
zPosition A	19.715 mm
zPosition B	20.715 mm
phi of 1st element A	0deg
phi of 1st element b	15deg

E.5 2-layered Hodoscope

Two fiber-layers.

elements X	192
elements Y	192
length	524 mm
width	2.0316 mm
thickness	2 mm
read out	QDC
zPosition X	192.344mm
zPosition Y	194.35mm
angle from x-axis X	-33.9deg
angle from x-axis Y	56.1deg

E.6 3-layered Hodoscope

Three fiber-layers, X and Y extended rectangular.

elements X	96
elements Y	96
elements D	136
length	382 mm
width	2 mm
thickness	2 mm
read out	QDC
zPosition X	98.72 mm
zPosition Y	96.72 mm
zPosition D	94.72 mm
angle from x-axis X	89.657deg
angle from x-axis Y	-0.343deg
angle from x-axis D	134.657deg

E.7 Micro-Strip-ring detector

Straight layer: wedges, ring layer: rings. Note: rotation sense is opposite to other circular detectors.

elements phi	128
elements R	100
inner radius	3.1 mm
outer radius	31 mm
Δr	0.279 mm
thickness	0.5mm
read out	QDC
zPosition Phi	26.485 mm
zPosition R	26.465 mm
phi of 1^{st} element	44.819 deg

E.8 Micro-Strip-spiral detector

Left and right layers: spiral.

elements left bent	128
elements right bent	128
inner radius	2.8mm
outer radius	31mm
thickness	0.52mm
bending	480mm/rad
read out	TDC
zPosition L	-
zPosition R	-
phi of 1^{st} element	-

Name	ID	element shape	material	readout	thickness [mm]	zPosition [mm]
Quirl-Straight	0	wedge	scintillator	QDC&TDC	5	3356
Quirl-Left	1	spiral	scintillator	QDC&TDC	5	3367
Quirl-Right	2	spiral	scintillator	QDC&TDC	5	3378
Ring-Straight	3	wedge	scintillator	QDC&TDC	5	3317.38
Ring-Left	4	spiral	scintillator	QDC&TDC	5	3326.38
Ring-Right	5	spiral	scintillator	QDC&TDC	5	3335.38
Barrel-Front	6	wedge	scintillator	QDC&TDC	15	367
Barrel-Back	7	wedge	scintillator	QDC&TDC	15	367
Start-A	15	wedge	scintillator	QDC&TDC	1	19.715
Start-B	16	wedge	scintillator	QDC&TDC	1	20.715
Hodoscope X	17	fiber	scintillator	QDC	2.02	98.72
Hodoscope Y	18	fiber	scintillator	QDC	2.02	96.72
Hodoscope D	23	fiber	scintillator	QDC	2.02	94.72
int. Hodoscope X	19	fiber	scintillator	QDC	2.02	192.344
int. Hodoscope Y	20	fiber	scintillator	QDC	2.02	194.35
Micro-Strip-Phi	21	wedge	silicium	QDC	0.51	26.485
Micro-Strip-Rad	22	ring	silicium	QDC	0.51	26.485
Calorimeter	25	hexprism	scintillator	QDC&TDC	450	$\sim$3390

Table E.1: Table with detector properties of the COSY TOF detector

Appendix F

Analysis

The following analysis-steps have been performed on the data:

1. conversion to hit-tree-file
2. calibration-generation for the individual run
3. prompt tracking
4. vee tracking
5. extraction
6. kinematical fit
7. luminosity calculation
8. plotting

F.1 Hit-tree-file generation

This step contains five algorithms:

parameter name	value
Read from tade	
Teufel Correction	
file for reference data	Teufel/stable/mean_values_run5140.log
Detectors to correct	15,16
Calibration	
Correct QDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 21, 22, 23
Correct TDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16
Get the hit-shapes	
Write Hits to tree	
use local directory	false

F.2 Calibration-generation

Read Hits from tree	
search for event	false
Use as event input list	false
Use local directory	false

Calibration	
Correct QDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 21, 22, 23
Correct TDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16

Quirl pixel	
pixel ID	**0**
first layer	0
second layer	1
third layer	2
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 – 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10mm

Ring pixel	
pixel ID	**1**
first layer	3
second layer	4
third layer	5
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 – 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10mm

Micro-strip pixel	
Use hit-cluster	false
PixelID	**5**
ID Ring shaped detector	21
ID Wedge shaped detector	22

Line Track search	
use vertex as start	true
use angular distance	false
max number of elements in 2 tracks	2
pixel stop-IDs	0, 1, 5
pixel min # elements on track	9, 9, 9
pixel: essential detector IDs	(0: 15, 16), (1: 15, 16), (5: 15, 16, 6)
start-pixel IDs	none
detectors	0 1 2 3 4 5 6 7 15 16 17 18 19 20 21 22 23
element search mode	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
max element distance	0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 0 0 0 0 0 0 2 2 1 1 3 3 3 3 1 1 3
check pixel IDs	5
cluster stop-IDs	none
cluster min # elements on track	none
cluster: essential detector IDs	none
pixel: max chi squared for track	25, 25, 50
theta restriction for stop	(0.062-0.18), (0.16-0.45), (0.4-0.84)
phi restriction for stop	(-π, π),(-π, π),(-π, π)
max distance	4mm, 4mm, 8mm
cluster: max chi squared for track	none

Calibration algorithm	
do geometry calibration	false
do beam calibration	false
do WALK calibration	true
do TDC-Offset calibration	true
do Barrel calibration	true
do on event basis	false
kill file on end	false
read from file	false
n Events for cal	100000
number of iterations	4
print pattern	401
detectors to calibrate	15, 16, 6, 7, 1, 2, 4, 5, 0, 3
pixel based	-1, -1, -1, -1, 0, 0, 1, 1, 0, 1
measures against	1, 0, 3, 2, 5, 4, 7, 6, 45, 76
is Stop pixel	0, 0, 1, 1, 1, 1, 1, 1, 1, 1
has 2-sided readout	-1, -1, 3, 2, -1, -1, -1, -1, -1, -1,
pixels dets	-1, -1, -1, -1, 1, 2, 1, 2, 0, 0
pixel det references to det	-1, -1, -1, -1, 6, 7, -1, -1, 8, -1
walk do only single iteration	0, 0, 0, 0, 0, 0, 0, 0, 1, 1
do light-run-correction	0, 0, 0, 0, 1, 1, 1, 1, 0, 0
reference pixels	0 1
mean, width for tdc offset	(-60 6), (-60 6), (-78 4), (-78 4), (-60 20), (-60 20), (-60 20), (-57 20), (-80 15), (-77 20)
min, max for qdc	(100 1500), (100 1500), (0 2000), (0 3000), (0 3000), (0 3000), (0 2500), (0 2500), (0 4000), (0 3000)
min, max for lrp	(0 80), (0 80), (0 3550), (0 3550), (0 300), (0 300), (0 300), (0 300), (0 3000), (0 3000),
calibration Output Path	$KTOFPACK/data/calibration/Calibration
author	K. Ehrhardt

F.3 Prompt tracking

Read Hits from tree	
search for event	false
Use as event input list	false
Use local directory	false

Calibration	
Correct QDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 21, 22, 23
Correct TDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16

Hodo pixel	
Pixel ID	**4**
ID first layer	19
ID second layer	20

Quirl pixel	
pixel ID	**0**
ID first layer	0
ID second layer	1
ID third layer	2
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10

Ring pixel	
pixel ID	**1**
ID first layer	3
ID second layer	4
ID third layer	5
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10

Barrel pixel	
PixelID	**2**
ID Front channel	6
ID Back channel	7
Pixel size	25

Line Track search	
use vertex as start	true
use angular distance	false
max number of elements in 2 tracks	2
pixel stop-IDs	0, 1, 2, 4
pixel min # elements on track	9, 9, 6, 8
pixel: essential detector IDs	(0:15, 16), (1: 15, 16), (2: 15, 16), (4: 15, 16, 6)
start-pixel IDs	none
detectors	0 1 2 3 4 5 6 7 15 16 17 18 19 20 21 22 23
element search mode	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
max element distance	0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 1 1 5 5 5 5 5 2 5 0 0 0 0 0 0 3 3 1 1 3 3 3 3 1 1 3
check pixel IDs	5
cluster stop-IDs	none
cluster min # elements on track	none
cluster: essential detector IDs	none
pixel: max chi squared for track	25, 25, 100, 50
theta restriction for stop	(0.07-0.2), (0.14-0.45), (0.8-1.36), (0.1-1.1)
phi restriction for stop	(-π,π), (-π,π), (-π,π)
max distance	4, 4, 20, 8
cluster: max chi squared for track	none

Particle speed	
do charged prompt	true
do neutral prompt	false
do min angle secs	true
do min+1 angle secs	true
ID of closest start detector	15
ID of start detectors	15, 16
ID of stop detectors	0, 1, 2, 3, 4, 5, 6, 7

Write Tracks to tree	
Search for read event	false
Use local directory	false

F.4 Vee tracking

Read Hits from tree	
search for event	false
Use as event input list	false
Use local directory	false

Calibration	
Correct QDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 21, 22, 23
Correct TDC	0, 1, 2, 3, 4, 5, 6, 7, 15, 16

Read Tracks from tree	
Search for read event	false
use as event input list	true
read event pattern	false
Use local directory	false
event pattern	2 0 0

Hodo pixel	
Pixel ID	**4**
ID first layer	19
ID second layer	20
Use hit-cluster	false
Use middle plane	false
Use phi modulation function	false

Hodo pixel 3 layers	
Pixel_ID	**3**
ID_D_layer	23
ID_X_layer	18
ID_Y_layer	17
Use hit-cluster	false

Micro-strip pixel	
PixelID	**5**
ID Ring shaped detector	21
ID Wedge shaped detector	22
Use hit-cluster	false

Quirl pixel	
pixel ID	**0**
ID first layer	0
ID second layer	1
ID third layer	2
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 – 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10

Ring pixel	
pixel ID	**1**
ID first layer	3
ID second layer	4
ID third layer	5
Parallel projection	false
Special projection plane	false
Origin for projection point	false
Target for projection point	false
Use time difference	false
Correct light flight time	false
Triangle approximation	true
min-max delta phi	-1 – 1
minimum number of elements	2
number of spiral-spiral-crossings	23
maximum time difference	10

Barrel pixel	
PixelID	**2**
ID Front channel	6
ID Back channel	7
Pixel size	25

V-Line Track search	
use only elements unused of Prompt	false
do projection search	false
max number of elements in 2 tracks	1
min # elements on track-vertex-track	13
max elements in common with prompt for pixel	3
max Distance Vertex-Plane	10
maximum chi for track-vertex-track	1.5
max angle diff pri to secondary	0.05
max Distance pixel-plane	2
pixel stop-IDs	0, 1, 2
pixel min # elements on track	7, 7, 6
pixel: essential detector IDs	0, 0, 0
cluster stop-IDs	none
cluster min # elements on track	none
cluster: essential detector IDs	none
pixel start-IDs	3, 4
cluster start-IDs	none
min start objects per plane	2, 2
detectors	17, 18, 19, 20, 23
element search mode	0, 0, 0, 0, 0
must not detectors	22
max suspect-element distance	2 2 2 2 2 1 2 2 2 2 2 1 4 4 4 4 4 1
pixel: max chi squared for track	4, 4, 10
cluster: max chi squared for track	none
max distance	2, 2, 5
vertex z-position from target	25, 100

Apply Signal Run Correction	
detectors to apply	0, 1, 2, 3, 4, 5, 6, 7
type of calibration	1, 101, 101, 1, 101, 101, 100, 100

Particle speed	
do charged prompt	true
do neutral prompt	false
do min angle secs	true
do min+1 angle secs	true
ID of closest start detector	15
ID of start detectors	15, 16
ID of stop detectors	0 1 2 3 4 5 6 7

Write Tracks to tree	
Search for read event	true
Use local directory	false
Write only pattern	false

F.5 Extraction

The extraction – or first-step data-reduction – is, as well as the next steps, done in a separate executable. The extractor reads an ascii-file containing the names of the runs to process. The data-basis is read and the run-data of all runs to process is stored.

initialize output-track-tree-file(s)
for all runs *i* **do**
  open hit and track files
  synchronize hit and track trees
  for all events in run[*i*] **do**
    if track-pattern other than for pp-elastic or $pK^+\Lambda$ **then**
      skip event
    end if
    read hits of event
    calibrate hits of event
    recalculate β of each track
    fill tree
  end for
end for
close files

The TDC-calibration applied here not only contains the usual offset- and binning-calibration and the signal-run-time (lrp) correction for the archimedian spirals but also the lrp-correction for the wedge shaped elements, that was fixed with pp-elastic-scattering and the geometrically reconstructed $pK^+\Lambda$-events.
Command-line-options are:

help—-h	show this help and exit
DB=FILENAME	specify a data-basis-file
runs=FILENAME	specify a file containing runs to use (all file specified this way will be used)
elastic=FILENAME	write elastic events in sample to file
pkl=FILENAME	write PKL-candidates in sample to file
lrp=FILENAME	read and do lrp-corrections from file FILENAME
makeLRP=FILENAME	generate lrp-corrections from sample and write them to FILENAME
tmpTree=FILENAME	write a temporary tree to file
show	show specified files and parameter

F.6 Kinematical fit

The kinematical fit as described in Appendix B was encapsuled into a separate executable. The properties for the fit are read from file, these are the number of particles to fit, the assignment to the track-structures (prompt or vees), the mo-

mentum and the particle ID of beam and target particles and, for each particle, the particle IDs (GEANT), the representation in which to fit, the way the property is treated (measured, unmeasured, fixed) and the participation in any additional constraint (e.g. decayed off Λ). Additionally it contains the maximum number of iterations and the ε-value, the maximum deviation of the sum of 4-momenta from momentum and energy conservation.
There are a lot of different ways to perform the fit, especially how the velocities of the particles shall be treated. As Command-line-parameters there are defined:

option	description
help	produce help message
show	show set parameters
input-file arg	input files
i arg (=kinfit.init)	init file
d arg	setup file
n arg	maximum number of events to process
start arg (=0)	start at entry arg
progress	show a progress bar
error-branch	write branches with errors
sigma-branch	write branches with sigmas
true-branch	write branches with true values (MC-only)
check-LAMBDA	add constraint for lambda decay
SIGMA-LAMBDA	check for sigma and lambda reaction both
erlangenIDs	do purely geometric particle identification
erlangenPlus	do purely geometric particle identification including phase-space-cuts
reconstructBadBetas	ignore betas if larger than 1
reconstructBadDproton	ignore beta of decay proton
reconstructBadPion	ignore beta of decay pion
geometry	do only geometrical reconstruction ignore betas
vee-fits	do vee-fits on each vertex
errors-from-tree	use errors from tree if available
ERRORS arg	file with errors and corrections
Correction arg	external correction file
largePerror arg (=1)	multiply prompt track errors by arg
largeDerror arg (=1)	multiply decay track errors by arg
geoTest	fit and write only data that passes geometry test
cuts	fit only reasonable pkl-data
dataReduction	write only converged data to file
o arg (=out.root)	out file name

The used command is

```
> kinfitter --i=kinfit.init --ERRORS=errorfile.data --show
--progress --reconstructBadBetas --cuts --o=outputFile.root
--check-LAMBDA inputFile.root
```

The file "kinfit.init" contains the aforementioned particle information, the file "errorfile.data" contains an error-lookup-table for the $pK^+\Lambda$-reaction, derived from monte-carlo-simulations (see sec. 6.8.1). The velocities with $\beta > 1$ will be reconstructed using energy and momentum conservation, reducing the number of over-constraints for this event. The constraint of Λ-decay is added and only events that match – very loose – cuts will be fitted at all.

F.7 Luminosity calculation

The luminosity in mBarn/event is calculated using pp-elastic scattering. The extractor (sec. F.5) produces not only a file for the $pK^+\Lambda$-events but also a file containing all 2-track, coplanar ($\Delta\phi < \pi \pm 0.05 rad$) pp-elastic ($\frac{1}{\sqrt{\tan\theta_1 \tan\theta_2}} < \gamma \pm 0.1$) events. These are kinfitted (using geometrically reconstructed velocities, the elastic init-file and no additional constraints). The actual luminosity then is calculated in a small program, that reads three track-tree input-files: one for fitted data (pp-elastic), one for purely simulated pp-elastic (using the SAID [39] differential cross-section) and one with these simulated events having passed the virtual detector and the analysis program. For each of these the $\cos\theta_{cm}$-distribution is generated. The SAID differential cross-section for this distribution has been hard-coded into the program. The two simulated files allow for an efficiency correction of the data. Normalizing the corrected data-distribution to the SAID-differential cross-section, the luminosity in mBarn/event can be extracted.
Command-line-options are:

Option	description
B=STRING	set the name of the branch to plot
S=FILENAME	simulation file name
M=FILENAME	simulation through detector file name
D=FILENAME	data file name
F=FILENAME	histogram file name
min=NUMBER	set the minimum-x of the histograms
max=NUMBER	set the maximum-x of the histograms
bin=NUMBER	set the number of bins for the histograms
O=STRING	set the out-file-pattern
PPRINTER	set the print-flag and the printer for printing
b=NUMBER	set the x-position, where to normalize the histograms
beam=NUMBER	set the beam-momentum in GeV/c
range=NUMBER#NUMBER	add a range to the list of ranges use at least one!

The actual command was:

```
> luminosity --B="cos(theta_cm_kf)" --F=fitted-elastic.root
--b=37 --range=-0.68#-0.43 --range=0.43#0.68 --range=-0.28#-0.18
--range=0.18#0.28 --min=-0.8 --max=0.8 --O=luminosityFile
```

F.8 Plotting

The final cuts and the plotting are done in an additional external program, reading readily filled histograms, files with track-tree-format or the PreCutTree-format, that also includes missing- and invariant-masses, in essence all values that will be plotted to histogram (This format has been introduced since it is much faster to plot and still contains all event data, so cuts can still be applied).
Meanwhile this functionality has been included into the **typeCase** GUI.

List of Algorithms

Bibliography

[1] The Erlangen analysis program.

[2] The Jülich-Tübingen analysis program: tof++.

[3] The WASA kinFit.

[4] Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org`.

[5] A. Böhm. *Untersuchung der Reaktion pp → pnpi+: Aufbau des Detectors COSY-NUS und erste Ergebnisse*. PhD thesis, TU Dresden, 1998.

[6] A. Filippi, R. Geyer, D. Hesselbarth. Track fitting and pattern recognition in the COSY-TOF Experiment. Technical report, COSY-TOF, 2001. Jülich Internal Note.

[7] A. Ucar. *Developments for the TOF Straw Tracker*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn, 2006.

[8] L. Alvarez-Ruso. Two-pion decay modes of the N*(1440) in $np \to dpipi$. 2000.

[9] W. Brodowski, R. Bilger, H. Calén, H. Clement, C. Ekström, K. Fransson, J. Greiff, S. Häggström, B. Höistad, J. Johanson, A. Johansson, T. Johansson, K. Kilian, S. Kullander, A. Kupść, P. Marciniewski, B. Morosov, W. Oelert, J. Pätzold, R. J. M. Y. Ruber, M. Schepkin, W. Scobel, I. Seluzhenkov, J. Stepaniak, A. Sukhanov, A. Turowiecki, G. J. Wagner, Z. Wilhelmi, J. Zabierowski, and J. Zlomanczuk. Exclusive measurement of the $pp \to pp\pi^+\pi^-$ reaction near threshold. *Phys. Rev. Lett.*, 88:192301, Apr 2002.

[10] C. Rohlof. Protokoll der Vermessung eines geraden und eines gewundenen Szintillatorstreifens aus dem Quirl. Technical report, COSY-TOF, 1995. COSY-TOF-NOTES, BO-11-1995.

[11] CERN, 1211 Geneva 23, Switzerland. *GEANT-Detector Description and Simulation Tool*, 1993. CERN Program Library Long Writeup W5013.

[12] COSY-TOF-Collaboration. The cosy-tof home-page. web. `http://www.fz-juelich.de/ikp/COSY-TOF`.

[13] F. Schmidt. Untersuchung des Gasgemisches Ar-CO2-CF4 der HERMES-Driftkammern BC 1-4. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1996.

[14] G. Breit, E. Wigner. Capture of slow neutrons. *Phys. Rev.*, 49:519, 1936.

[15] H. Bethe. Zur Theorie des Durchgangs schneller Korpuskularstrahlen durch Materie. *Annalen der Physik*, 397:3, 1930.

[16] H. Kämmerling. TOF-Szintillator-Ringdetektor. Technical report, Zentralabteilung Technologie, Forschungszentrum Jülich, 1999.

[17] HIRES Collaboration. High resolution study of the Λp final state interaction in the reaction $p + p \rightarrow K^+ + (\Lambda p)$. *Physics Letters B*, 687:31–35, 2010.

[18] J. Wächter. Ein doppelseitiger Ringmikrostreifendetektor für COSY-TOF. Master's thesis, Universität Erlangen, 1998.

[19] F. James. MINUIT - Function Minimization and Error Analysis - Reference Manual. http://wwwasdoc.web.cern.ch/wwwasdoc/minuit/minmain.html, 2000.

[20] F. James and M. Roos. Minuit: A system for function minimization and analysis of the parameter errors and correaltion. 1975. Comput. Phys. Commun. **10**, 343-367.

[21] Jochen Kress. *Entwicklung und Installation eines Zentralkalorimeters und Messungen der Reaktion p p → p p pi+ pi- mit spin-polarisierten Protonen am Flugzeitspektrometer COSY-TOF*. PhD thesis, Universität Tübingen, 2003.

[22] J.R. Christman. SU(3) and the quark model, 2001. Project PHYSNET. University of Michigan.

[23] K.Ehrhardt. *The typeCase home-page*. `http://www.pit.physik.uni-tuebingen.de/~ehrhardt/KTOF`.

[24] M Abd el-Bary. *Development of a Cryogenic Target System wit Optimal Access to Reaction Detectors*. PhD thesis, Forschungszentrum Jülich, 2004.

[25] M. Dahmen et al. The quirl scintillator. *Nuclear Instruments and Methods in Physics Research*, A 348:97, 1994.

[26] M. Röder. Private communications, 2011.

[27] M. Schulte-Wissermann. The Dresden analysis program TofRoot. [31].

[28] Maier, R. et al. Cooler synchrotron COSY. *Nucl. Phys.*, A626:395c–403c, 1997.

[29] Marc Wagner. Entwicklung und Bau eines intermediären Szintillatorfaser-Hodoskops für COSY-TOF. Master's thesis, Universität Erlangen, 1997.

[30] Marc Wagner. *Assoziierte Strangeness-Produktion in der Reaktion $pp \to K^0\Sigma^+ p$ am COSY-Flugzeitspektrometer*. PhD thesis, Universität Erlangen, 2002.

[31] Martin Schulte-Wissermann. *Investigation of Meson Production at COSY-TOF Using the Analysis Framework TofRoot*. PhD thesis, Universität Dresden, 2004.

[32] P. Ringe. Quirl-Geometrie. Technical report, COSY-TOF, 1994. COSY-TOF-NOTES, BO-9-1994.

[33] P. Wintz et al. Status Report of the Straw Tracking Detector for COSYTOF. Technical report, Institut für Kernphysik, Forschungszentrum Jülich, 2001. Jahresbericht des IKP des Forschungszentrums Jülich 2001.

[34] P. Wintz et al. The New Straw Tracker For COSY-TOF. Technical report, Institut für Kernphysik, Forschungszentrum Jülich, Jülich/Germany, 2003. IKP Annual Report 2003.

[35] P. Wintz et. al. Resolution and Eciency of the Straw Tracker for COSY-TOF. Technical report, Forschungszentrums Jülich, 2004. Jahresbericht des IKP des Forschungszentrums Jülich.

[36] P. Wintz et al. Aging Tests of Straw Tubes for the PANDA Experiment. Technical report, Forschungszentrums Jülich, 2007. Jahresbericht des IKP des Forschungszentrums Jülich.

[37] Rochester G.D., et al. Evidence for the existance of new unstabe elementary particles. *Nature*, page 855, 1947.

[38] S. Wirth. *Erste Experimente am COSY-Flugzeitspektrometer zur Untersuchung des assoziierten Strangeness-Produktion im Proton-Proton-Stoss*. PhD thesis, , 1995.

[39] SAID, R. A. Arndt et al. . **S**cattering **a**nalysis **i**nteractive **di**alin.
telnet/ssh: said.phys.vt.edu oder said-hh.desy.de, Username "said"
WWW: http://said.phys.vt.edu oder http://gwdac.phys.gwu.edu/.

[40] P. Schönmeier. *Untersuchung der assoziierten Strangenessproduktion in der Reaktion $pp \to K^+\Sigma^+ n$ mit dem COSY-Flugzeitspektrometer*. PhD thesis, Technische Universität Dresden, 2003.

[41] M. Schulte-Wissermann. Production of Λ and Σ^0 hyperons in proton-proton collisions. *EPJ A*, 2010.

[42] T. Czarnecki. Bau und Test eines Szintillationsfaserhodoskops für ein Hyperonexperiment bei COSY. Master's thesis, Universität Erlangen, 1994.

[43] T. Nakano et al. Evidence for a Narrow S=+1 Baryon Resonance in Photoproduction from the Neutron. *Phys. Rev. Lett.*, 91(012002), 2003.

[44] The COSY-TOF Collaboration. Evidence for a narrow resonance at 1530 MeV/c2 in the K^0p – system of the reaction $pp \to \Sigma^+K^0p$ from the COSY-TOF experiment. *Physics Letters B*, 595:127, 2004.

[45] The COSY-TOF Collaboration. Improved study of a possible Theta+ production in the $pp \to pK^0\Sigma^+$ reaction with the COSY-TOF spectrometer. *Physics Letters B*, 649:252, 2007.

[46] The Particle Data Group. Booklett, 2010.

[47] Trolltech. The Qt project. web: `http://trolltech.com`.

[48] W. Gast. The Silicon Microstrip Quirl Telescope SQT. Technical report, Institut für Kernphysik, Forschungszentrum Jülich, 2008. Jahresbericht des IKP des Forschungszentrums Jülich 2008.

[49] Wolfgang Schröder. *Untersuchung der assoziierten Strangeness-Produktion in den Reaktionen $pp \to K^+\Lambda+$ und $pp \to K^+\Sigma^0p$ am Flugzeitspektrometer COSY-TOF*. PhD thesis, Universität Erlangen, 2003.

[50] Shi-Lin Zhu. Pentaquarks. *Int.J.Mod.Phys. A*, 2004.