Software for the alignment of multi-wire proportional chambers in the HARP experiment

Risto-Antti Paju, Queens' College, Cambridge

Vacation project for Part III Physics carried out at
CERN, Geneva, Switzerland, 11/07 - 21/09/2001
under the supervision of
Dr Alan Grant and Dr Petr Gorbounov

8th October 2001

Except where specific reference is made to the work of others, this work is original and has not been already submitted either wholly or in part to satisfy any degree requirement at this or any other University.

Risto A Paju

Abstract

Multi-wire proportional chambers (MWPCs) are commonly used in high energy physics, for the tracking of charged particles. The overall tracking resolution is determined by the separation of anode wires in each chamber.

In order to perform accurate beam tracking, the positions individual wires in MWPCs must be known to a precision preferably higher than the chamber resolution. This can be difficult as the wires are confined inside the chambers. The present work with the HARP experiment at CERN used chambers whose wire positions could only be measured to about 0.5 mm of lateral offset, and a few milliradians of rotation about the beam axis.

A significantly higher precision is obtained via software alignment, which finds the positions of chambers producing the best fit of the beam trajectory. However, a fundamental theorem asserts that at least two chambers must have known positions, in order to obtain unique values for the positions of the others.

A new method has been proposed by P. Gorbounov of the HARP group, whereby only one chamber needs to be transversely fixed. When data are recorded with this chamber in different longitudinal positions, the effect of multiple fixed chambers can be achieved: the other chambers' lateral positions must be consistent with all configurations of the reference chamber simultaneously.

The method has now been implemented by the author. The principle has been verified by simulations included in the software. It has been used with the HARP experiment to align four crossed-wire chambers to a precision of order 1 μm in offset and 1 mrad in rotation, when data from about 400,000 events were used. The procedure can be readily generalized for any situation where the alignment of tracking equipment is required.

Contents

1 Introduction **4**
1.1 HARP hadron production experiment 4
1.2 Operation of MWPCs 4
1.3 Beam tracking 5

2 Theoretical background **6**
2.1 Basic principle of alignment 6
2.2 A fundamental problem and its solution 7

3 Computational approach **9**
3.1 Mathematical description 9
3.2 Choice of language 9
3.3 Optimizations 10
3.4 Simulations 10
3.5 Debugging 11

4 Implementation **12**
4.1 General layout 12
4.2 Data structures 12
4.3 Simulation 13
4.4 Function fitting 14

5 Results and discussion **14**
5.1 Note on uncertainties 14
5.2 Verification of the principle by simulations 14
5.3 Results from the real data 15

6 Conclusions **15**

A Program source **19**

1 Introduction

1.1 HARP hadron production experiment

The HARP experiment utilizes the T9 beam from the Proton Synchrotron at CERN. The protons in the beam have a tightly defined momentum, which can be adjusted in the range between about 2 and 15 GeV/c. The beam is directed towards a nuclear target which is one of a choice of metals, and the produced secondary hadrons are investigated.

The principal goals of the experiment are related to neutrino physics. The yield of pions produced with protons of different momenta, and different targets, is one of the main topics of interest. That information will be of importance for future designs of proton-driven neutrino factories. These in turn can be used, in conjunction with neutrino detection facilities, to investigate neutrino masses.

Multi-wire proportional chambers (MWPCs) are used for the accurate determination of the incoming beam trajectories. This information is utilized in monitoring the beam envelope, which is important feedback for controlling the beam characteristics. In addition, it is used to find the proportion of particles that reach the target. Moreover, tracking the individual particles will provide the positions and directions at which they hit the target.

1.2 Operation of MWPCs

MWPCs are now commonly used for the tracking of charged particles. A brief explanation of a single proportional chamber is provided to explain their operation [1].

The chamber consists of a negative (cathode) sheath surrounding a positive one (anode), the two forming a capacitor in effect. The space in between is filled with a noble gas or some other suitable gas. As the charged particle enters the gas, the latter will be ionized. Electrons will then drift towards the positive anode wire, and the acceleration provided by the electric field of the capacitor may induce secondary ionization. Eventually electrons and ions are deposited on the anode and cathode respectively. A detectable signal is produced, as the potential of the capacitor changes with its charge. In the so-called proportional mode, the signal will be proportional to dE/dx of the particle.

An invention by Charpak (1968, Nobel prize in 1992), the MWPC achieves a reasonably accurate method of tracking by combining an array of proportional chambers into one unit. It is usually a grid of parallel anode wires sandwiched between two cathode planes. From the field line pattern (figure 1) it can be deduced that the wires define independent measuring units: if the particle only passes through the field lines from one wire, a signal will only be induced into that wire. Therefore no separate chambers are required and the wires can be spaced by e.g. 1 mm, as in this experiment, which is a satisfactory resolution in many cases.

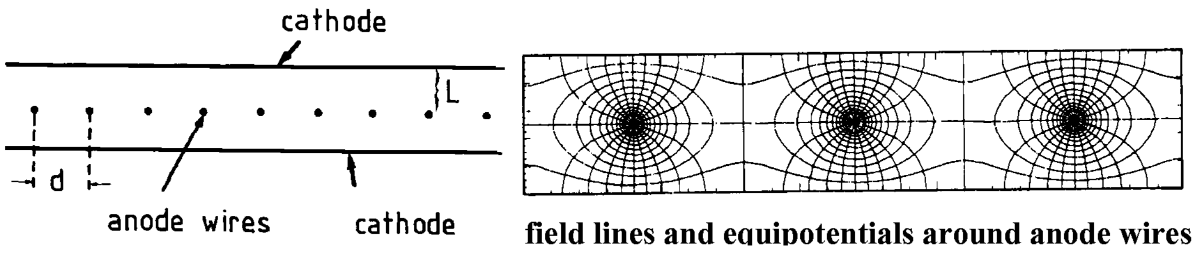

Figure 1: Basic arrangement of a MWPC (Adapted from [1])

For a wire spacing d, the standard deviation of the particle position (distance parallel to the wires, in the wire plane) is easily computed:

$$\begin{aligned} \sigma^2 &= \langle s^2 \rangle - \langle s \rangle^2 \\ &= \frac{1}{d}\int_{-d/2}^{d/2} s^2 ds - 0 \\ &= \frac{d^2}{12} \\ \Rightarrow \sigma &= \frac{d}{\sqrt{12}} \end{aligned}$$

It is relatively common to get signals from two adjacent wires even when it is believed that only a single particle passed through the wire plane. Naturally, it is possible that the particle passes through the border between adjacent wire-zones because its trajectory is angled relative to the normal to the plane. In this experiment, these events are accepted. Events with a higher number of signals (hits) per plane, of those pairs of signals not from adjacent wires, are rejected by the software trigger.

There exists another mechanism by which multiple hits per plane can be detected. The products of ionization may enter the field of an adjacent wire. These can be difficult to distinguish between the above events, particularly in this experiment where the signal strengths are not recorded due to the limitations of data processing. However, it is believed that such cases are sufficiently rare to be insignificant.

1.3 Beam tracking

It is often the case that MWPCs are arranged in pairs with the wire sets orthogonal to each other. This could be used to approximate the xy coordinates of the particle trajectory in a plane. Of course the two sets of wires cannot be physically in the same plane, but the approximation can be improved arbitrarily by having a sufficiently small z-separation. In HARP a separation of about 10 mm is achieved by having a common cathode plane for both sets of wires. That pair is then in fact a single chamber and called a MWPC as well. There are in total four such chambers in HARP, as illustrated in figure 2.

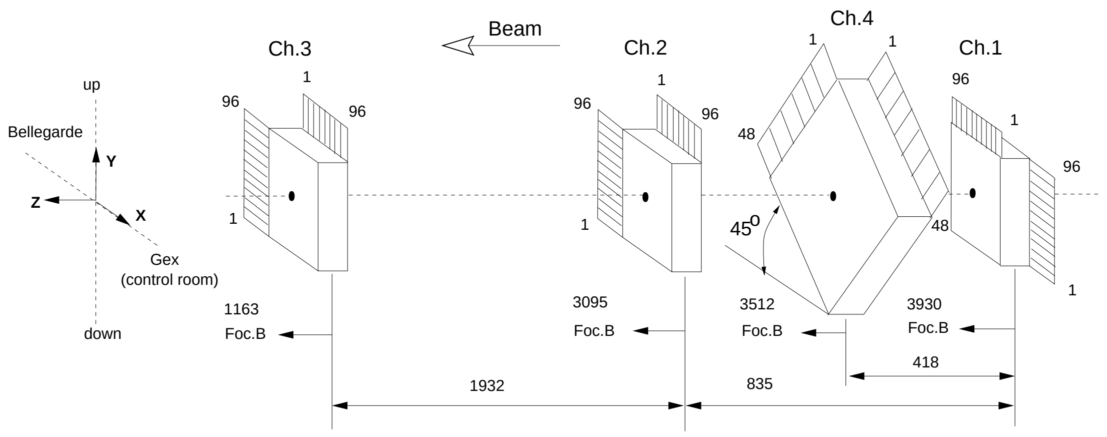

Figure 2: Layout of HARP MWPCs. The distances are in millimetres, and measured from the centres of chambers. (Adapted from [2])

However, it is not necessary to make the approximation with xy planes. The trajectory of the particle can be reconstructed by fitting a straight line, or perhaps a more general curve, to the pattern of wires that are hit. This is done via a least squares method where the residuals are the squared distances between the wires and the trajectory. Nevertheless, it is more convenient from the point of view of construction, to arrange the wire planes in pairs which share a cathode plane.

There is inevitably a level of interaction between the particle and the MWPC, hence the straight line trajectory is only an approximation. It is, however, an enormously complex task to compute the precise interactions and modify the straight-line picture. Moreover, the high momentum of the particle (around 2GeV/c in HARP) suggests that the deviations from straight line are negligible, especially considering the overall precision of tracking.

While the precision is dependent on $\sigma = d/\sqrt{12}$ for MWPCs with a wire spacing of d, it is improved with an increasing number of chambers.

2 Theoretical background

2.1 Basic principle of alignment

Alignment is the process of determining the positions (x, y, z and rotations) of the MWPCs. Basically it is done by direct mechanical measurements, but they can only performed to within about 0.5 mm. The problems arise mainly from the construction of MWPCs. The wires are not mechanically accessible from the outside, so their positions can only be approximated. Moreover, the relation (separation and angle of rotation) between the two wire planes in a chamber is not known exactly. Therefore it is essential to treat the two sets of wires as independent.

More precise alignment can be performed as follows. First we assume certain positions and rotations (collectively called 'alignment parameters') for all wire planes. Then for each event, trajectory fitting is performed, based on these parameters. The total sum of normalized squared residuals, χ^2, is computed to indicate the overall goodness of fit. The process is repeated using slightly different alignment parameters. The actual alignment parameters are found when χ^2 is at its minimum (best fit) value.

In practice, χ^2 is treated as a function of the alignment parameters, while the beam data are constant parameters. To minimize that function is a tedious calculation, so in practice a specialized software is used. At CERN the common choice has been to use the MINUIT [3] package.

2.2 A fundamental problem and its solution

It is easy to see that χ^2 is unchanged if the entire system of MWPCs is moved or rotated. The beam trajectories, which are computed on the basis of signal wires, will be shifted accordingly. The trivial answer to this problem is to have one chamber fixed with known coordinates. The alignment parameters of other planes are then computed relative to that. Nevertheless, unique parameters cannot be found simply by fixing one plane. Translations or rotations linear in z would still leave the parameters undefined, as depicted in figure 3. This was well exhibited by the simulations used to test the program.

Thus more than one of the planes are to be fixed. Moreover, as one wire plane is insensitive to the dimension along the wires, two or more planes must be fixed in each of two dimensions. In the usual arrangement with each MWPC consisting of two crossed wire planes, two fixed chambers could be used to achieve this.

The chambers may have non-zero lateral and rotational deviations, as long as they are known. But this is somewhat against the original argument, because the calculation of the alignment parameters would be unnecessary if we could measure them directly.

A solution suggested by P. Gorbounov has been to use one chamber, fixed in the lateral and rotational senses, in different longitudinal positions. As the MWPCs are mounted on metal rails, this can be done to a high precision. Even when the rails are not exactly parallel to the z-axis, the deviation is easy to measure when compared to measuring the actual positions of the wires.

The above procedure of alignment is then performed with sets of data, with the one chamber in different z-positions. χ^2 is the total sum of normalized squared residuals over all configurations, with the same alignment parameters used for the non-fixed planes in all cases. Naturally this introduces further complications to the method, but with well designed software utilizing sufficient computing power, the desired goal remains accessible.

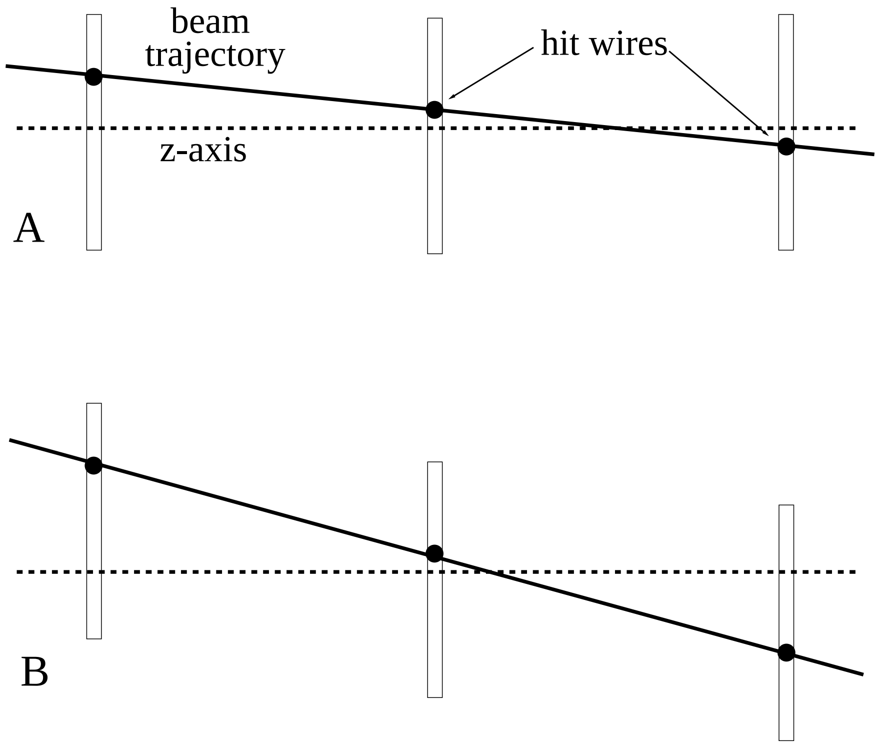

Figure 3: A simplified analogy of the fundamental problem. The hits on each plane are unchanged. These are only two possible beam trajectories, equally well fitted to the hits.

3 Computational approach

3.1 Mathematical description

The following notation has been adapted from P. Gorbounov's previous work on the subject [2]. Each wire in the ith wire plane is represented by the equations

$$\begin{aligned} z &= z_i \\ -x \sin\theta_i + y\cos\theta_i + \Delta_i(w) + \delta_i &= 0 \end{aligned}$$

where θ_i is the angle of the wire w.r.t. the x-axis and Δ_i the intended offset from the z-axis, normal to the wire direction. The latter is calculated from the wire number w via $\Delta_i = s_i(C_i - w)$ using constants s_i, C_i. Also $\theta_i = \theta_{0,i} + \delta\theta_i$. Thus δ_i and $\delta\theta_i$ are the deviations from ideal, or alignment parameters.

Chamber	plane	θ_0/degrees	s/mm	c	z/mm
1	1	0	$+1$	48.5	$3930+5$
	2	90	$+1$	48.5	$3930-5$
4	3	-135	-4	24.5	$3512+5$
	4	135	-4	24.5	$3512-5$
2	5	-90	-1	48.5	$3095+5$
	6	0	$+1$	48.5	$3095-5$
3	7	-90	-1	48.5	$1163+5$
	8	0	$+1$	48.5	$1163-5$

Table 1: The current parameters of the MWPCs in HARP (adapted from [2])

It is approximated that the wire planes are normal to the z-axis and any tilts in this sense can be neglected. Naturally it would be possible to include these deviations into the alignment procedure as well. However, their effect is estimated to be considerably smaller than that of the above parameters δ_i and $\delta\theta_i$.

3.2 Choice of language

The author was initially familiar with FORTRAN 90. It also seemed the most appropriate language for numerical applications of the required nature. On one hand, this is due to the intrinsic functions and the powerful syntax e.g. for array handling. On the other hand, it was expected that a wealth of number-crunching algorithms would be readily available as recompiled libraries. Probably the most important of these is the MINUIT function minimization package.

At CERN it first appeared that only the GNU compiler for FORTRAN 77, g77, would be available. Most of the code for HARP beam analysis was written in F77, as were the libraries. Work on this program was started in F77 which was a disappointment, because this version of FORTRAN lacks many of the syntactical features that make F90 a powerful physics language. There were also great difficulties in dynamical allocation of memory. Other languages such as C were considered at this point, however a PGI F90 compiler was eventually found and it even supported g77 libraries.

3.3 Optimizations

The function that is to be minimized performs a large number of calculations on each call. In this experiment, there are six sets of data: three different positions of one wire chamber, and two types of beams in the sense of focusing. Each set has initially had 100,000 events, and after removing ambiguous events via a software trigger, about 70,000. Therefore the function performs about 400,000 least square fits in producing the χ^2-value. In the minimization stage, hundreds or thousands function calls are executed. Therefore it is crucial to optimize the performance of the program by any means necessary, while not compromising numerical accuracy.

FORTRAN 90 has a powerful syntax with 'array expressions' and these were used as widely as possible, both to make the source code simpler and to increase performance. The program was developed on Linux on a dual Pentium III workstation, so the compiler could utilize the parallelity of the source code by delegating work to both processors. In addition, the vector instructions (SSE) were accessible with the compiler to provide further optimization.

As the rotational deviations would be something in the order of milliradians, it was expected that differential formulae should be used to compute the sines and cosines of angles. They would provide sufficient precision and they were believed to be faster. However, using the real trigonometric functions turned out slower only by a few percent, and it was decided to use them instead for better precision. This was a good sign proving that there actually is a section called 'math coprocessor' in the CPU, providing assembler instructions of the trigonometric functions.

It can be argued that there are many obvious optimizations to be done. One reason is, of course, that the program needs to be used in a production environment. The code cannot then be optimized indefinitely, and there are more important bug-fixes and integrity checks to be made. Less obvious is the fact that, after some parallelization is made to fully employ both processors of the PC, any further parallelization is in vain because all of the computing power is already in use. There have been many decisions, thus, to leave the code with ordinary `do` loops instead of array expressions even if the latter would have been syntactically possible.

3.4 Simulations

From the beginnings of the project it was clear that a section for producing simulated data was to be included. Generally, in projects of this kind, simulations are a useful 'tool' for debugging the code. Naturally this method requires that the simulation part itself is sufficiently free of bugs.

On the other hand, the simulated data generation involves the inverses of many routines of the actual program. For example, the conversion of positional coordinates to wire patterns is exactly the opposite of what is done by the fitting routine. Therefore, a bug in either is only left unnoticed if exactly the same mistake is made in both. The probability for this is believed to be significantly lower than that for a single bug.

Another use for simulations is perhaps of even greater importance. The method for circumventing the fundamental problem of alignment has not been used before, as far as the HARP group is aware. Simulated data with known deviations from the ideal positions provides a rigorous way of testing the procedure. The wire patterns generated this way are fed into the minimization routine, and the resulting alignment parameters can easily be compared to the actual deviations.

3.5 Debugging

The use of simulations as debugging aids has already been mentioned. However, they are generally the more useful, the higher level of code is being analyzed. For more obscure details and syntactical mistakes, different approaches are required.

An obvious starting point for debugging is the compiler itself, with the error messages it produces, along with the relevant line in the source file. However, there are invariably more subtle mistakes that do not interrupt the compilation, and are manifested as runtime errors. In these cases, it has proved instructive to print the values of some variables at certain critical points of the execution. At the very least, these would indicate the point in the code where the program crashes, and also produce specific information on the state of the program.

A particular problem with this project has been the use of variables of different precision. Most of the library subroutines use 64-bit precision and this was also chosen as the default precision for most variables. However, some of the library routines were only available as 32-bit versions. This was not a great problem, as FORTRAN 90 handles different types intelligently: for example the code

```
real*8 :: pi = 3.141592653589793
real*4 x

x = 2*pi
```

is valid FORTRAN 90 and the variable `x` is assigned the correct value.

The trouble arises, for example, when a `real*8` variable is passed to a subroutine which uses `real*4` variables only. The library subroutine receives a string of 32 bits and treats them as a 32-bit real variable.

Printing values of intermediate variables is good practice then, because unexpected values can be quickly recognized. This is most obvious when the variables are known to be positive only, and the printed values come out negative.

It should also be noted that 32-bit precision would be adequate if we only judged by the uncertainties in the final results. Using `real*4` variables only would certainly improve the execution time. However, the choice of 64-bit precision is to minimize rounding errors in the many intermediate stages. With the large number of both the data, and the number of function calls, the cumulative errors might be significant, had we only used 32 bits for real variables.

4 Implementation

4.1 General layout

The program makes extensive use of the `module` structure of FORTRAN 90. It provides namespaces that can be accomplished with `common` blocks in F77, but with more flexibility. A particular improvement over modules is the ability to use dynamically allocatable arrays.

Many parameters and even the event data are passed on via modules. Firstly, this is essential for memory management because one prefers not to make a copy of the data array with about 1.4 million `real*8` variables. On a PC with 256 MB of RAM, only one copy could be used without having to resort to virtual disk-swapping memory. This alternative was proved very slow by running two copies of the program simultaneously.

Moreover, the function to be minimized would have a very complicated interface if every parameter was to be included as its arguments. It should be noticed that in terms of minimization, the event data are in fact constant parameters. The only variables then are the sixteen alignment parameters: δ_i (`d(i)`) and $\delta\theta_i$ (`dtheta(i)`) with $i = 1...8$. As discussed before, some of these will be fixed, but this is made at a higher level to keep the program as general and modular as possible.

Another positive consequence of the `module` structures is that the program can be thoroughly divided into subroutines to increase the clarity and flexilibity of the code. Otherwise the subroutines could become complicated due to the large number of arguments, which might compromise the intended clarity and maintainability.

4.2 Data structures

The structure for containing the wire hit patterns was adapted from related programs by P. Gorbounov. For one event, it is an integer array of rank 2, with the contents described by the following comment:

```
! hit(i, 1) = no. of hits per i'th plane
! hit(i, j+1) = wire number of j'th hit
```

As mentioned above, a 'good' event can have a maximum of two hits per plane, and these must be adjacent. Therefore $j \leq 2$. For the entire set of data, further indices are introduced to denote the event number in a configuration, and the configuration number.

For the purpose of least squares fitting in the minimization routine, the hit patterns are converted into spatial coordinates. For a single event we use a rank 2 array, named thus to separate it from the lowercase delta:

$$\begin{aligned} \texttt{big\_d}(i, 1) &= \Delta_i \\ \texttt{big\_d}(i, 2) &= \text{plane number} \end{aligned}$$

This structure was chosen because of the fitting routine; there can be 0, 1 or 2 hits per plane, but there is exactly one linear equation per each hit i. While the conversion from the hit pattern is slightly complicated, it is better for the overall performance to have the fitting stage as fast as possible. As above, two further indices are introduced to account for the entire data set. However, this means that each 'slice' of `big_d(i, j)` has a fixed i dimension. Although one can count the non-zero plane numbers of each event, it has proved faster to store the number of hits per event in the dedicated array `num_hits`.

As the solution to the fundamental alignment problem requires, there are sets of data taken with different positions z_i. Although the z-positions have several values only for one chamber i.e. two planes, it has been decided for clarity to derive all z_i from a rank two array:

```
do j=1, nconfs
   z = zconf(:, j)

   ! (simulation or fitting over all data in the conf.)

end do
```

4.3 Simulation

The subroutine `simulation` is used to generate a random event. It starts by two random points, each located on one of the extreme wire planes. They can be anywhere on the rectangular planes instead of being confined to the wires. Parameters for the straight line joining them are computed. They are used to find the hit wire numbers on each wire plane, by finding the wire closest to the beam line on each $z = z_i$ plane.

To produce a realistically large set of data, the wrapper routine `sim_set` first produces random values for the deviations δ_i and θ_i for each plane i. With these as a basis, `simulation` is then run in a loop through a number of times to generate a number of events in each configuration. The relevant set of z_i is used for each configuration, as described in section 4.2.

A library subroutine [4] is used to generate the random numbers. UNIX time is chosen as the random seed, as it is readily available and non-repeating.

4.4 Function fitting

The subroutine `beam_fit` is used as a wrapper around the `TLS` [5] library routine, which is a general least squares fitting algorithm. The residuals in this case are the normalized distances between the beam trajectory and the wires with induced signals. `TLS` performs the minimization of the sum of the squared residuals, to produce the parameters of the beam trajectory (straight line). However, the most important outcome for the alignment is the minimized sum of squared residuals.

With `TLS` one has the choice to solve several systems of linear equations at once [5]. It is possible that this would increase performance noticeably, when compared to looping over individual processes. However, this feature could not be used because in the real data, the number of hits per event is not fixed. As each hit corresponds to one equation in the trajectory fitting routine, each event will have to be treated separately.

The `MINUIT` package is used to minimize the total sum of squared residuals by varying the alignment parameters. The function to be minimized is coded as the subroutine `FCN` with a definite kind of interface as required by `MINUIT`. The input arguments of `FCN` are in an array of 16 elements: $\delta\theta_i$ and δ_i with $i = 1...8$. Inside `FCN`, `beam_fit` is called in a loop over all configurations, and all events in each configuration. For each configuration, the correct set of z_i is used. In addition, the correct deviations of the fixed planes are set per configuration.

`MINUIT` requires initial guess values for the alignment parameters in order to know where to start the search. It should be stressed that these are either all zero, or randomly chosen. This is important when testing the algorithm with simulated data: the minimization routine does not know the real values of alignment parameters. Therefore the results presented in section 5.2 reflect the integrity of the minimization routine. This is essential in the case of real data when the actual deviations cannot be known.

5 Results and discussion

5.1 Note on uncertainties

With MINUIT, the default scheme for determining the parameter errors is based on standard chi-squared methods. The minimized value of χ^2 is defined to have unit uncertainty, from which the corresponding uncertainties in alignment parameters are derived.

5.2 Verification of the principle by simulations

Initially, simulations were made with only one or zero fixed chambers. As was expected, the resulting alignment parameters had no relation to the simulated deviations. They were not even consistent, in the sense that the same set of deviations would produce different alignment parameters on different runs of the minimization program. Based on this, the author was convinced of the severity of the fundamental problem.

With the scheme of one chamber in three positions in operation, the outcomes of simulations suddenly changed into something consistent. An example of a full-sized simulation/minimization run is provided below. There are six configurations with 76,000 events each, with planes 3, 4, 7 and 8 fixed.

At this time there was no data available from the chamber with planes 3 and 4 so it has been omitted in the simulation as well. Planes 7 and 8 are in the chamber with different z-positions, and different deviations in each case, so their alignment parameters are not computed.

It can be seen from the above figures, that the minimization results agree with the initial deviations within uncertainty limits. Moreover, the final value of χ^2 is approximately half of the initial, even though the latter is based on the 'ideal' hits from simulated trajectories, showing another aspect of the success of the minimization.

5.3 Results from the real data

Test runs were carried out using only a part of the data. The noted consistency between the alignment results from different sets of data indicates a further level of success for the method. The set of alignment parameters obtained is provided in table 3; the data from planes 3 and 4 are included here.

6 Conclusions

A new method for the alignment of multi-wire proportional chambers, proposed by P. Gorbounov, has been implemented and tested by the author. The improvement over conventional alignment procedures is in that the one chamber that is laterally fixed, is employed in a number of longitudinal positions. The overall effect is the same as that achieved with multiple fixed chambers, with the advantage that the lateral coordinates need to be known only for one chamber.

Both simulation runs, and tests on actual data, have verified that the method is successful in solving the fundamental problem of alignment. It has eventually been used with the HARP experiment to align four crossed-wire chambers. The precision obtained with about 400,000 events is of order 1 μm in lateral offset and 1 mrad in rotation.

References

[1] C. Joram: Particle Detectors, CERN Summer Student Lecture Series 2001

[2] P. Gorbounov, L. Scotchmer: MWPC: Layout and geometry,

`http://harp.web.cern.ch/harp/Classified/Sub_detectors/MWPC/`

```
Simulated deviations

       plane   dtheta/rad
           1   6.7297556E-02
           2   4.8374962E-02
           3   0.0000000E+00
           4   0.0000000E+00
           5   3.7635576E-02
           6   3.2903196E-03
           7   0.0000000E+00
           8   0.0000000E+00

       plane   delta/mm
           1    1.213176
           2   -3.092195
           3   0.0000000E+00
           4   0.0000000E+00
           5   -3.283027
           6   -3.383858
           7  -0.4517817
           8    1.638722

Alignment parameters from MINUIT

       plane   dtheta/rad      +/-
           1   6.7256272E-02   5.3191496E-05
           2   4.8352595E-02   6.8815985E-05
           3   0.0000000E+00   0.0000000E+00
           4   0.0000000E+00   0.0000000E+00
           5   3.7636429E-02   5.6742949E-05
           6   3.2561109E-03   4.3560252E-05
           7   0.0000000E+00   0.0000000E+00
           8   0.0000000E+00   0.0000000E+00

       plane   delta/mm        +/-
           1    1.213263       2.1140520E-03
           2   -3.092112       2.2907092E-03
           3   0.0000000E+00   0.0000000E+00
           4   0.0000000E+00   0.0000000E+00
           5   -3.283680       1.4097507E-03
           6   -3.384363       1.3028856E-03
           7   0.0000000E+00   0.0000000E+00
           8   0.0000000E+00   0.0000000E+00
```

Table 2: Simulated deviations and the corresponding results of alignment

```
Alignment parameters from MINUIT

       plane   dtheta/rad      +/-
           1   5.0818520E-03   4.2462247E-04
           2   4.8763724E-03   8.4756484E-04
           3  -4.2760791E-03   1.7678936E-04
           4  -8.9576663E-03   1.6403430E-04
           5   2.8447525E-03   2.9622624E-04
           6   5.4293168E-03   3.6899984E-04
           7   0.0000000E+00   0.0000000E+00
           8   0.0000000E+00   0.0000000E+00

       plane   delta/mm        +/-
           1   9.0514682E-02   1.8791633E-03
           2   -2.900985       6.3912272E-03
           3    2.436198       2.2991272E-03
           4   0.5246968       3.5192200E-03
           5    1.233972       2.0006932E-03
           6  -0.7262853       1.1148308E-03
           7   0.0000000E+00   0.0000000E+00
           8   0.0000000E+00   0.0000000E+00
```

Table 3: Alignment parameters from real data

[3] CERN Program Library Long Writeup D506: MINUIT,

`http://wwwinfo.cern.ch/asdoc/minuit/minmain.html`

[4] CERN Program Library Writeup V113: Fast Uniform Random Number Generator,

`http://wwwinfo.cern.ch/asdoc/shortwrupsdir/v113/top.html`

[5] CERN Program Library Writeup E230: Constrained and Unconstrained Linear Least Squares Fitting,

`http://wwwinfo.cern.ch/asdoc/shortwrupsdir/e230/top.html`

APPENDIX

A Program source

```
! HARP MWPC Alignment by Risto-Antti Paju <risto.a.paju@iki.fi> 2001
! with guidance from my supervisors at CERN: Alan Grant, Petr Gorbounov

! Also thanks to Frederick James of CERN for help on MINUIT

! Notes:
!
! subroutine get_hits (reading data files) adapted from code by P. G.

module common_params
  implicit none

  ! #planes, #wires/plane, max #beams/config
  integer, parameter :: nplanes = 8, max_ndata = 76000

  ! 3 different z-positions of one chamber
  ! * 2 types of beam for each
  integer, parameter :: nconfs = 6

  integer ndata(nconfs)

  ! max #hits/plane
  integer, parameter :: maxhpl = 2
  integer, parameter :: maxhits = maxhpl * nplanes

  real*8, dimension(nplanes) :: z
  real*8, dimension(nplanes, nconfs) :: zconf

  ! start & end areas at z(1) and z(8) are squares:
  real*8, parameter :: width = 96 ! of planes in mm

  real*8, parameter :: pi = 3.14159265358979323846

  ! worst fraction of d**2 to discard from sum(d**2)
  real*8, parameter :: discard = 0.02

  real*8, dimension(maxhits, 2, max_ndata, nconfs) :: delta_set
  integer, dimension(max_ndata, nconfs) :: num_hits ! total per beam

  real*8, dimension(nplanes) :: iangle, norm
```

```
end module common_params

module data_files
  use common_params

  character(len=11) :: file_prefix = "data/align."
  character(len=4), dimension(nconfs) :: file_nos = (/"3897", "3900", "3902",

end module data_files

module conversion_params
  use common_params

  real*8, parameter, dimension(nplanes) :: &
       & s = (/1, 1, 4, 4, 1, 1, 1, 1/), &
       & c = (/48.5, 48.5, 24.5, 24.5, 48.5, 48.5, 48.5, 48.5/)
  ! s is also used for the normalization constants

end module conversion_params

module minim_params
  use common_params
  ! step, limit for changing dtheta (radians) & d (mm)
  ! zero limit = unbounded
  ! 1 degree = 0.017 radians
  real*8 :: dtstep = 0.005, dtlimit = 0, dstep = 0.1, dlimit = 0

  ! +/- limits for randomly chosen init values
  real*8 :: dtinit = 0, dinit = 0
  ! ditto for simulated deviations
  real*8 :: dt_slimit = 0.2, d_slimit = 7

  integer, parameter :: nfixed = 2
  integer, dimension(nfixed) :: fixed = (/7, 8/)

  ! the lateral position of chamber 3 varies with z.. complication++
  real*8, dimension(nplanes, nconfs) :: dt_fixed, d_fixed

  character*10 dtname(nplanes), dname(nplanes)

  integer narg, npari, nparx, istat
  integer ierflg, ivarbl

  real*8 xval(2*nplanes), error, fmin, fedm
  ! change of min. function value that determines param. errors
  real*8 :: up = 1 ! FVAL is sum of norm. squared residuals

end module minim_params

subroutine initialize_params
  use minim_params  ! includes common_params
  use conversion_params

  real*4, dimension(2*nconfs) :: ranx

  ! normalization constants for distances
  !  = grid separations in mm
  norm = abs(s) / dsqrt(12.)

  ! ideal, intended angles of wires w.r.t. the x-axis
  iangle(1) = 0
  iangle(2) = pi / 2
  iangle(3) = -3 * pi / 4
  iangle(4) = 3 * pi / 4
  iangle(5) = -pi / 2
  iangle(6) = 0
  iangle(7) = -pi / 2
  iangle(8) = 0

  !     z-positions of wire planes
  zconf(:, 1) = (/3934+5, 3934-5, 3512+5, 3512-5, 3095+5, 3095-5, 1270+5, 1270

  zconf(:, 2) = zconf(:, 1)
  zconf(:, 3) = zconf(:, 1)

  zconf(7:8, 2) = zconf(7:8, 1) + 553.5
  zconf(7:8, 3) = zconf(7:8, 2) + 552

  zconf(:, 4:6) = zconf(:, 1:3)

  ! later, some of these will be non-zero
  dt_fixed = 0
  d_fixed = 0

  d_fixed(7, 2) = -0.3 ! flipped sign due to positive s
  d_fixed(7, 3) = -0.6
  d_fixed(8, 2) = 0.22
  d_fixed(8, 3) = 0.5
  d_fixed(:, 4:6) = d_fixed(:, 1:3)

  num_hits = 0
  ndata = max_ndata
  delta_set = 0

end subroutine initialize_params

subroutine get_delta(hit, big_d, nhits)
  use conversion_params
  integer i, j, k

  ! hit(i, 1) = no. of hits per i’th plane
  ! hit(i, j+1) = the wire number of j’th hit
  integer, intent(in), dimension(nplanes, 1+maxhpl) :: hit
  real*8, intent(out), dimension(maxhits, 2) :: big_d

  k = 1
  do i=1, nplanes
     do j=2, hit(i, 1)+1
        big_d(k, 1) = s(i)*(c(i) - hit(i, j))
        big_d(k, 2) = dble(i)
        k = k + 1
     enddo
  enddo

  nhits = sum(hit(:, 1))

end subroutine get_delta

subroutine get_hits(hit_set)
  use data_files

  integer i, j, k
  integer*2 ibuf(9)
  integer temp(8)

  integer, intent(out), dimension(nplanes, 1+maxhpl, max_ndata, nconfs) :: hit
  external get_hits

  hit_set = 0

  do k=1, nconfs
```

```
     open(1, file=trim(file_prefix)//file_nos(k), status="old", form="unformat
     temp = 0
     do i=1, max_ndata
        read(1, end=999) ibuf

        temp(1)=ibuf(2)
        temp(2)=ibuf(3)

        temp(3)=ibuf(8)
        temp(4)=ibuf(9)

        temp(5)=ibuf(4)
        temp(6)=ibuf(5)
        temp(7)=ibuf(6)
        temp(8)=ibuf(7)

        do j=1, nplanes
           if (temp(j) .gt. 0) hit_set(j, 1, i, k) = 1
           hit_set(j, 2, i, k) = mod(temp(j), 100)
           hit_set(j, 3, i, k) = temp(j) / 100
           if (hit_set(j, 3, i, k) .gt. 0) hit_set(j, 1, i, k) = 2

           ! bug hunt (trigger)
           if (hit_set(j, 3, i, k) .gt. 0) then
              if (abs(hit_set(j, 2, i, k) - hit_set(j, 3, i, k)) .gt. 1) then
                 print *, "TRIG ERROR:", j, k, hit_set(j, 2, i, k), hit_set(j,
              end if
           end if

        enddo

        ! bug hunt
!        print *, i, temp(3:4)

     end do
     close(1)

999  ndata(k) = i - 1

  end do

end subroutine get_hits


subroutine get_delta_set (hit_set)
```

```
  use common_params

  external get_delta, trigger

  real*8, dimension(maxhits, 2) :: big_d
  integer, intent(in), dimension(nplanes, 1+maxhpl, max_ndata, nconfs) :: hit_
  integer hit(nplanes, 1+maxhpl), nhits, i, j, k

  delta_set = 0
  do j=1, nconfs
     k = 0
     do i=1, ndata(j)
        hit = hit_set(:, :, i, j)
        call get_delta(hit, big_d, nhits)
        k = k + 1
        delta_set(:, :, k, j) = big_d ! passed on..
        num_hits(k, j) = nhits ! ..via module common_params
     enddo

  end do

end subroutine get_delta_set

subroutine beam_fit(a, b, d, big_d, fit_params, nhits)
  ! TLS assumes abnormal convention of row/column order!!!
  ! and uses 32-bit precision !!!

  use common_params

  integer m1, m, l, ier, n
  COMMON /TLSDIM/ m1, m, n, l, ier

  real*8, intent(in), dimension(nplanes) :: a, b, d

  real*8, dimension(maxhits, 2), intent(in) :: big_d
  integer, intent(in) :: nhits

  real*8, dimension(5), intent(out) :: fit_params

  ! note the precision
  real*4, dimension(4, maxhits) :: dm
  real*4, dimension(maxhits) :: rhs
  real*4 eps, lp(4), aux(8)

  integer i, plane
  integer, dimension(4) :: ipiv

  n = 4
  l = 1
  eps = 0

  do i=1, nhits
     plane = nint(big_d(i, 2))

     rhs(i) = -(big_d(i, 1) + d(plane)) / norm(plane)
     dm(1, i) = a(plane)*z(plane) / norm(plane)
     dm(2, i) = a(plane) / norm(plane)
     dm(3, i) = b(plane)*z(plane) / norm(plane)
     dm(4, i) = b(plane) / norm(plane)
  enddo

  m = nhits ! # linear equations

  call TLS(dm(:, 1:m), rhs(1:m), AUX, IPIV, EPS, lp)

  fit_params(1:4) = lp

  fit_params(5) = aux(1)

end subroutine beam_fit

subroutine simulation (hit, fit_params, dtheta, d)
  ! Simulation with given deviations from ideal angles.
  ! Only one hit per plane.
  use conversion_params

  external get_delta, get_d2

  real*8, dimension(nplanes) :: a, b, x, y
  real*8, dimension(nplanes), intent(in) :: dtheta, d
  real*8, dimension(maxhits, 2) :: big_d, const
  real*8, dimension(5), intent(out) :: fit_params
  integer, dimension(nplanes, 1+maxhpl), intent(out) :: hit

  integer, dimension(nplanes) :: w ! note this is a local variable

  real*4, dimension(nplanes):: ranx ! note the precision

  integer plane, i, j, nhits
```

```
call ranmar(ranx, 4)

x(1) = (ranx(1) - 0.5)*width
x(8) = (ranx(2) - 0.5)*width
y(1) = (ranx(3) - 0.5)*width
y(8) = (ranx(4) - 0.5)*width

!     the line has the parametric equation
!     x = ax*z + bx
!     y = ay*z + by

ax = (x(1) - x(8)) / (z(1) - z(8))
ay = (y(1) - y(8)) / (z(1) - z(8))

bx = x(1) - ax*z(1)
by = y(1) - ay*z(1)

fit_params(1) = ax
fit_params(2) = bx
fit_params(3) = ay
fit_params(4) = by

! init #planes
big_d(1:nplanes, 2) = (/(dble(i), i=1, nplanes)/)

big_d(nplanes+1:maxhits, 2) = 0

x = ax*z + bx
y = ay*z + by

! Differential formulae would be _slightly_ faster, but we want precision he
a = -sin(iangle + dtheta)
b = cos(iangle + dtheta)

big_d(1:nplanes, 1) = - (a*x + b*y + d)

!     convert to actual wire numbers (inverse of get_delta !-)
w = nint(c - (big_d(1:nplanes, 1) / s))

hit = 0

hit(1:nplanes, 1) = 1
hit(1:nplanes, 2) = w

! quantize big_d for calculation of d**2
```

```
  ! (and check consistency of get_delta :-)

  call get_delta(hit, big_d, nhits)

  const = 0
  const(1:nplanes, 1) = big_d(1:nplanes, 1) + d
  const(1:nplanes, 2) = big_d(1:nplanes, 2)

  call get_d2(fit_params, a, b, const, nhits)

end subroutine simulation

subroutine get_d2(fit_params, a, b, const, nhits)
  use common_params

  real*8, dimension(nplanes), intent(in) :: a, b
  real*8, dimension(maxhits, 2), intent(in) :: const
  real*8 fit_params(5)
  integer i, plane
  integer, intent(in) :: nhits

  fit_params(5) = 0
  do i=1, nhits
     plane = nint(const(i, 2))
     fit_params(5) = fit_params(5) + ((a(plane)*(fit_params(1)*z(plane) + fit_
  enddo

end subroutine get_d2


subroutine fcn (npar, GRAD, FVAL, XVAL, IFLAG, futil)
  ! to be used with minuit
  use minim_params

  external beam_fit

  integer i, j, k
  integer, intent(in) :: npar, iflag

  real*8 fit_params(5)
  real*8, dimension(2*nplanes), intent(in) :: xval

  real*8, intent(out) :: fval

  real*8, intent(out), dimension(npar) :: grad
```

```
    real*8, dimension(nplanes) :: dtheta, d, a, b

    real*4 d2_set(max_ndata) ! for flpsor

    dtheta = xval(1:nplanes)
    d = xval(nplanes+1:2*nplanes)

    fval = 0
    do j=1, nconfs
       z = zconf(:, j)

       ! (fixed is used as an array subscript :-)
       dtheta(fixed) = dt_fixed(fixed, j)
       d(fixed) = d_fixed(fixed, j)

       ! We could use differential formulae, but the true ones are not that slow
       a = -sin(iangle + dtheta)
       b = cos(iangle + dtheta)

       do i=1, ndata(j)

          call beam_fit(a, b, d, delta_set(:, :, i, j), fit_params, num_hits(i,
          d2_set(i) = fit_params(5)
       enddo

       ! discard the worst fraction
       call flpsor(d2_set, ndata(j))
       k = nint((1. - discard) * ndata(j))
       fval = fval + sum(d2_set(1:k))
    enddo

  end subroutine fcn

  subroutine sim_set(hit_set, d2)
    use minim_params

    external simulation

    integer i, j, k
    integer, intent(out), dimension(nplanes, 1+maxhpl, max_ndata, nconfs) :: hit
    real*8 fit_params(5)
    integer, dimension(nplanes, 1+maxhpl) :: hit
    real*8, intent(out) :: d2
    real*8 dtemp
```

```
real*8, dimension(nplanes) :: dtheta, d

real*4 ranx(2*nplanes) ! note the precision

print *, "lateral deviations of planes:"
call ranmar(ranx, nfixed*nconfs)
do i=1, nfixed
   do j=1, nconfs
      d_fixed(fixed(i), j) = 20.*(ranx((i-1)*nconfs+j)-0.5)
      print *, "plane/config/dev:", fixed(i), j, real(d_fixed(fixed(i), j))
   end do
end do
print *, ""

call ranmar(ranx, 2*nplanes)
do i=1, nplanes
   dtheta(i) = (ranx(i) - 0.5) * dt_slimit
   d(i) = (ranx(i+nplanes) - 0.5) * d_slimit
enddo

d2 = 0
do j=1, nconfs
   dtemp = 0
   z = zconf(:, j)

   ! (fixed is used as an array subscript :-)
   dtheta(fixed) = dt_fixed(fixed, j)
   d(fixed) = d_fixed(fixed, j)

   do i=1, max_ndata
      call simulation (hit, fit_params, dtheta, d)
      dtemp = dtemp + fit_params(5)
      hit_set(:, :, i, j) = hit
   enddo
   d2 = d2 + dtemp
enddo

open(1, file="sim.dev", status="replace")
write(1, *) "Simulated deviations"
write(1, *) ""
write(1, *) "       plane   dtheta/rad"
do i=1, nplanes
   write(1, *) i, real(dtheta(i))
end do
```

```
    write(1, *) ""
    write(1, *) "       plane   delta/mm"
    do i=1, nplanes
       write(1, *) i, real(d(i))
    end do
    close(1)

  end subroutine sim_set

  subroutine print_args
    use minim_params

    real*8 bar

    real*8, dimension(nplanes) :: dtheta, d, dterr_pb, derr_pb

    real*8, dimension(nplanes):: dterr_p, dterr_m, derr_p, derr_m

    ! get arguments and their errors
    do i=1, nplanes
       call mnpout(i, dtname(i), dtheta(i), error, -dtlimit, dtlimit, ivarbl)
       call mnpout(i+nplanes, dname(i), d(i), error, -dlimit, dlimit, ivarbl)
       call mnerrs(i, dterr_p(i), dterr_m(i), dterr_pb(i), bar)
       call mnerrs(i+nplanes, derr_p(i), derr_m(i), derr_pb(i), bar)
    enddo

    call mnstat(fmin, fedm, error, npari, nparx, istat)

    open(1, file="align.out", status="replace")

    write(1, *) "Alignment parameters from MINUIT"
    write(1, *) ""
    write(1, *) "       plane   dtheta/rad      +/-"!       +err            -err
    do i=1, nplanes
       write(1, *) i, real(dtheta(i)), real(dterr_pb(i))!, real(dterr_p(i)), rea
    enddo
    write(1, *) ""

    write(1, *) "       plane   delta/mm        +/-"!       +err            -err
    do i=1, nplanes
       write(1, *) i, real(d(i)), real(derr_pb(i))!, real(derr_p(i)), real(derr_
    enddo

    close(1)
    print *, "minuit’ed:", real(fmin), "+/-", real(fedm)

  end subroutine print_args

  subroutine minimization
    use minim_params

    external fcn, print_args

    real*8, dimension(2*nplanes) :: arglis
    real*4 ranx(2*nplanes)
    real*8, dtl, dl

    logical, dimension(nplanes) :: is_fixed = (/nplanes*.false./)

    is_fixed(fixed) = .true.

    call ranmar(ranx, 2*nplanes)
    ranx = 2*(ranx - 0.5)

    arglis = 0

    call mninit(5, 6, 7)
    call mnseti("The alignment of MWPCs in HARP")

    ! prepare limits for variables
    dtname = "dtheta_"
    dname = "d_"

    do i=1, nplanes
       ! these names work only if nplanes .le. 9
       dtname(i)(8:) = achar(48+i)
       dname(i)(3:) = achar(48+i)

       if (is_fixed(i)) then
          dtl = 0
          dl = 0
       else
          dtl = dtinit*dble(ranx(i))
          dl = dinit*dble(ranx(i+nplanes))
       endif

       call mnparm(i, dtname(i), dtl, dtstep, -dtlimit, dtlimit, ierflg)
       call mnparm(i+nplanes, dname(i), dl, dstep, -dlimit, dlimit, ierflg)
    enddo
    arglis = 0

    call mnexcm(fcn, "set nog", arglis, 0, IERFLG, 0)

    do i=1, nfixed
       arglis(2*i - 1) = fixed(i)
       arglis(2*i) = fixed(i) + nplanes
    end do

    call mnexcm(fcn, "fix", arglis, 2*nfixed, IERFLG, 0)

    arglis(1) = up
    call mnexcm(fcn, "set err", arglis, 1, IERFLG, 0)

    call mnexcm(fcn, "mig", arglis, 0, IERFLG, 0)

    ! this doesn’t seem to work, and the parabolic errors are good enough
  !  call mnexcm(fcn, "mino", arglis, 0, IERFLG, 0)
    call print_args

  end subroutine minimization


  program harp_align
    use common_params

    integer, dimension(:, :, :, :), allocatable :: hit_set

    real*8 d2, yy
    integer input, iflag, i, j, k

    real*8 xval(2*nplanes), fval, dtheta(nplanes), d(nplanes)

  !----------------------
    print *, "Random seed input:"
    read *, input
    print *, input

    ! we need the following random numbers:
    ! 4*max_ndata*nconfs to simulate beams
    ! 2*nplanes to simulate deviations
    ! 2*nplanes for initial deviations in the minimization stage
    ! 2*nconfs for initial deviations of the fixed planes
    call rmarin(input, 4*max_ndata*nconfs + 4*nplanes + 2*nconfs, 0)
    call initialize_params

    allocate(hit_set(nplanes, 1+maxhpl, max_ndata, nconfs))

  !  call sim_set(hit_set, d2)
    call get_hits(hit_set)

    call get_delta_set(hit_set)
    deallocate(hit_set)

    call minimization

    print *, "sim:", d2

  end program harp_align
```