Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep Informatica

A software package for reading out position sensitive light detectors

J.A.D. Cahill

begeleider: Prof.dr.ir. L. Spaanenburg

augustus 1996

RuG

Summary

In this report a software package is developed to read out different kinds of position sensitive detectors and build images from incoming data. This is achieved by using a DSP–board and a fast PC with a Windows operating system.
In the first part of this report the theoretical aspects of the gamma–ray detection mechanism are discussed and general algorithms are developed.

In the second part specific algorithms that apply to the practical problem are discussed together with the rest of the software required for both the DSP and the PC.

Contents

Introduction

A detector has been designed by D.E.P. at Roden in The Netherlands. A gamma–camera, incorporating this detector, is being developed at The University of Southampton in England. This camera is for applications in nuclear medicine. In short, it measures radioactive particles emitted from patients who have been injected with a radioactive isotope.

Multi pixel versions of these detectors are used in this project. Similar detectors from Hamamatsu in Japan are also evaluated in the experiments.

The purpose of the project is to create a real–time processing system capable of producing an image from the incoming gamma–photons. This will be realized using a DSP plug–in board and a fast PC with a Windows operating system. The system must be able to use various kinds of detectors with variable pixelshapes and sizes and also be fast.

Basically this system will allow a user to set up a particular hardware configuration, build a simple configuration file and start the application without rewriting algorithms. Also, in the event that software has to be changed the user has to be able to do this in a simple manner, without having expert knowledge and understanding of the complete package. Therefore, in addition to this report, a user–manual and a technical–manual will also be written. The reason for this is that this report will place more emphasis on the theoretical side of the project than on the practical side.

This report consists of 2 parts:

- A theoretical part in which the necessary algorithms are developed and evaluated.
  Here, general algorithms are developed and practical issues are only introduced when necessary.

- A practical part in which the complete system is described and developed.
  Here, algorithms are further developed for the practical system, based on the theory from part one.

PART I

THEORY

1. Gamma-ray detection principles

Before starting any theoretical studies, something has to be said about the gamma-ray detection mechanism. The detectors that are used in this project give the best response in visible light. Therefore gamma-photons cannot be detected without the aid of so-called scintillator crystals. These crystals are positioned in front of the detectors and produce optical-photons when a gamma-photon interacts with them. In most cases the crystal-arrays are designed in such a way that the created photons are restricted in their movement while traveling downwards. This ensures that only a predetermined width of the detector's entrance window will be hit.

When the light hits the detector's entrance window, the photons, entering at different angles, will refract and spread in width. Finally a lightspot will fall onto the detector's photocathode. The shape of this lightspot will be a circle and has a Gaussian distribution with the peak in the middle. See figure 1.1.

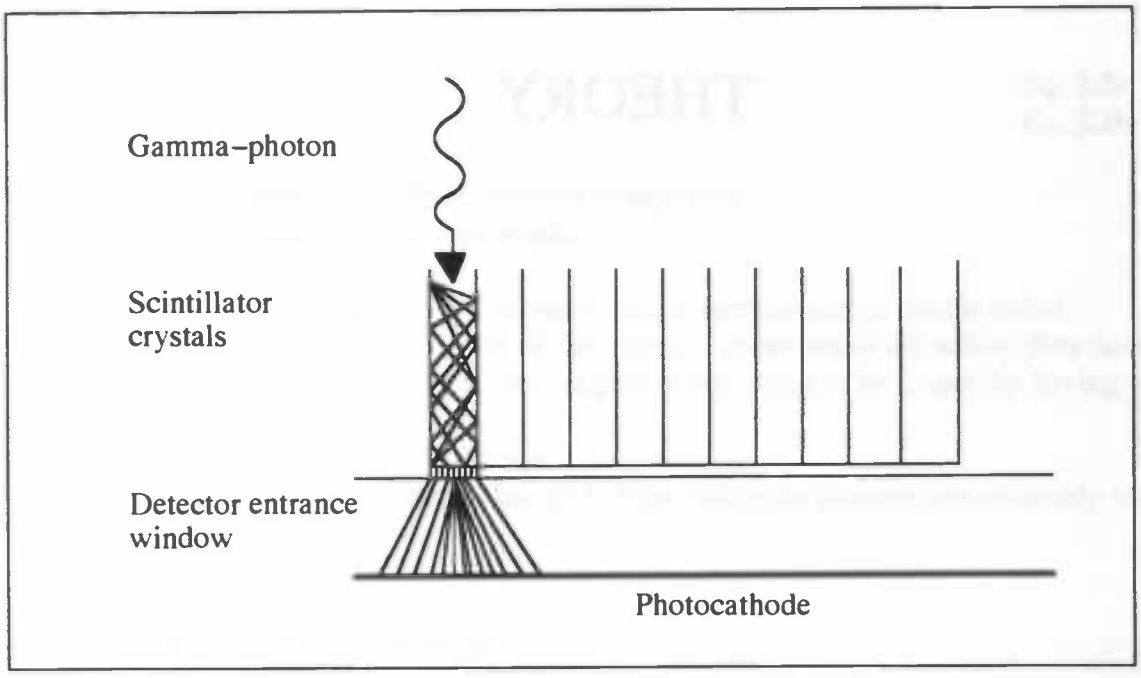

Figure 1.1: Light distribution through the use of scintillatorcrystals.

The distribution of light on the entrance window has a Gaussian shape. This means that most of the photons follow a more or less straight line and are concentrated in a relatively small area on the photocathode.

Upon leaving the entrance window on the other side, the photons interact with a photocathode layer. Here photoelectrons are released. They are accelerated in a strong electrical field and finally hit the photo–diodes (pixels), situated at the bottom, where they can be detected. See figure 1.2.

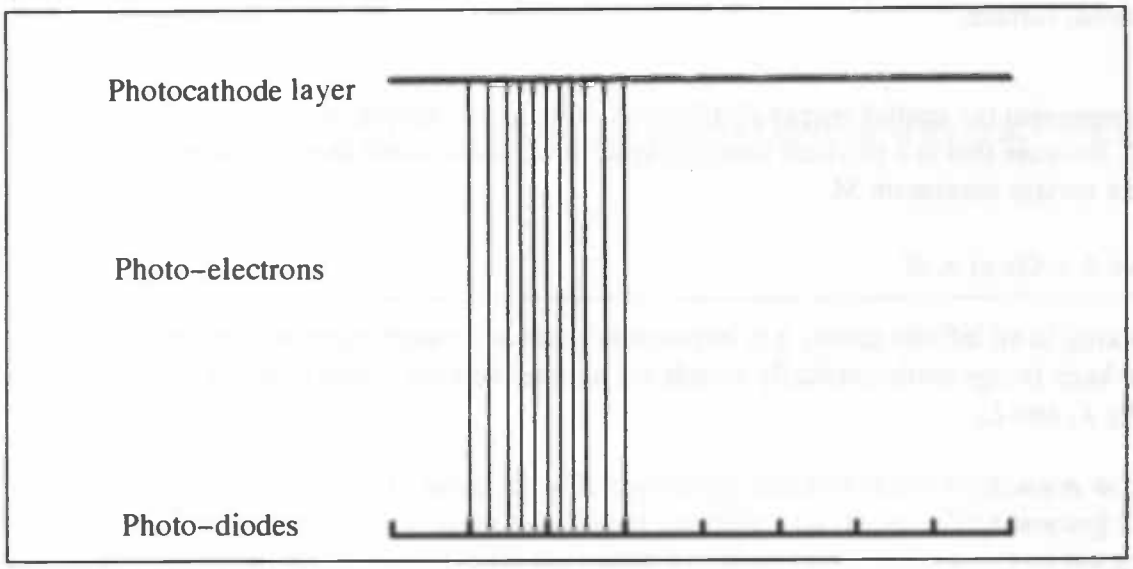

Figure 1.2: Inside the detector (HPD).

The problem discussed may be divided into 2 parts:

- Find the activated area and all activated pixels.
- Determine the position of the peak in the distribution as accurately as possible in order to find the original position of the gamma–photon. Also calculate the total energy of the gamma–photon by adding all hit pixels (= photo–diodes).

Algorithms providing solutions to both of these problems are developed and explained in this part of the report.

2. Finding an object on a 2D surface

Without paying too much attention to the given practical problems it is possible to develop general algorithms which can be used for this purpose. First the space of observation is defined and from that point on, different algorithms are developed that search for, and find, objects on a 2–dimensional surface.

Let $C(x,y)$ represent the spatial energy distribution of an image source of energy at spatial coordinates (x,y). Because this is a physical imaging system we can assume that the intensity is restricted to a certain maximum M.

So we have $0 < C(x,y) \leqslant M$ Eq. 2.1

When working in an infinite space, it is impossible to do any calculations within a reasonable time, so to keep things mathematically simple we assume that the image space is not infinite but bounded by L_x and L_y

So $0 \leqslant x \leqslant L_x$ Eq. 2.2a
$0 \leqslant y \leqslant L_y$ Eq. 2.2b

From now on all coordinates will be referred to as pixels.
A few important assumptions are also made:

- There is only one object on the grid. If one is found then the search can be ended.
- Every pixel that is active must be part of the object. If other pixels are active, they have to be distinguishable from activated pixels that are part of the object to be found (by having very low activation levels).
- Number of active pixels in an object is a.
- The object is completely situated on the grid. This makes the calculations relatively easy.

2.1 Method 1: search every position

The easy way to find an object on a bounded surface is by simply checking every pixel on the grid until all pixels a of an object are found.

This algorithm is simple and only needs a list of available pixel coordinates so that it can access each pixel on the grid. The word "available" is used because the space of observation does not have to be square, and not all coordinates in the grid have to exist. This will become clearer in the following chapters when different types of detectors are discussed.
It will find all active pixels whilst searching and calculations can be done either immediately during the search or after the search has been completed.

The algorithm in pseudo-code,

```
LOOP LIST OF X,Y COORDINATES UNTIL NUMBER OF PIXELS= a OR EOL
[
IF PIXEL(X,Y) = ACTIVE THEN
                [
                PIXELS = PIXELS + 1;
                CALCULATIONS OR SAVE COORDINATES FOR LATER;
                ]
]
```

Figure 2.1: The simple algorithm.

By using this approach, $L_x \times L_y$ positions will have to be checked in the worst case, depending on the number of active pixels and the position of the object on the grid. The running time of the search algorithm increases linearly with the number of pixels ($N_{total} = L_x \times L_y$) on the grid and the size of the object. Say we have K instructions per pixel. The algorithm needs at most $K \times N_{total}$ instructions. This clearly is an O(N) problem. If the number of pixels on the grid or the size of the object increases, then so will the running time of the algorithm (on the average).
So whilst this algorithm searches all pixels on the grid until all the active pixels are identified, it is quite fast because it identifies the active pixels on the run and needs little overhead to do this. So as long as the total number of pixels is relatively low, this algorithm will be very fast.

2.2 Method 2: search systematically

Another way to approach this problem is by searching the grid systematically. This implies that certain information about the objects that have to be found must be known in advance. What we need to know is :

•The shape of the object.
We know that the object is confined to a circle (as stated in chapter 1 explaining gamma-photon detection principles).

•The size of the object.
Assume that the size of the object is a pixels.

In a systematic search we have to sure that nothing can be missed. If the size of a circular object is known, it is possible to step through the pixels in the grid in such a way that one active pixels of an object will always be found. For example: assume a square grid of $N_{total} = 64$ pixels, so $N_x = N_y = \sqrt{64} = 8$ and a size of a=4 active pixels. This means $\sqrt{a} = 2$ pixels in the x–direction and $\sqrt{a} = 2$ pixels in the y–direction. To avoid missing an object, $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ columns should be checked in x and $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ rows in y direction, with a total of

$\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ pixels. Here it is 16 Eq. 2.3

Now the total grid can be systematically searched as follows (the black dots show the search pattern, the greyed areas show which pixels can be part of the object once a pixel has been found):

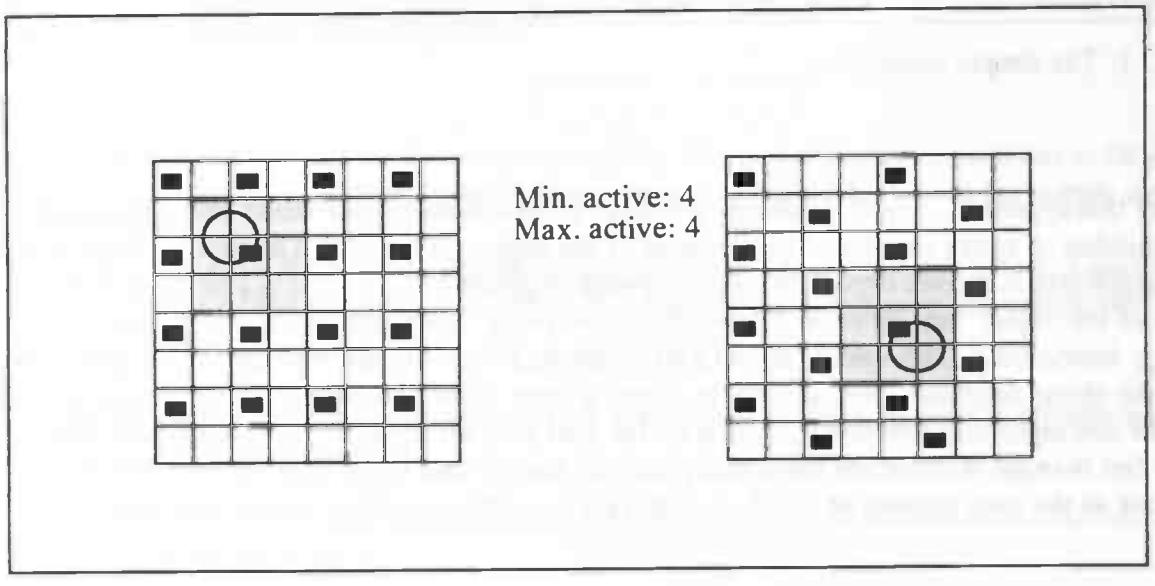

Figure 2.2: Search patterns.

In this way only $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ pixels have to be checked. If one assumes that there is only one object, then the search can be ended at this point. In this way the number of pixels to be checked is reduced considerably. In the case of a=4 and $N_{total} = 64$, the total number of pixels checked is 16 at the most (worst case), depending on the position of the object on the grid. This is 1/4 of the total number of pixels and quite an improvement on the simple algorithm. As opposed to the previous algorithm, where the execution time was linearly dependent on the size of the object and the position, this one performs better if the size of the object grows. Of course the rest of the pixels still have to be identified, but the search speed increases if the size of the object increases:

The following graph shows the maximum number of pixels that have to be checked when there are 1000 pixels on the (square) grid and the (square) object's size varies from 1 to 30 pixels (the Matlab file to generate this graph can be found in appendix A):

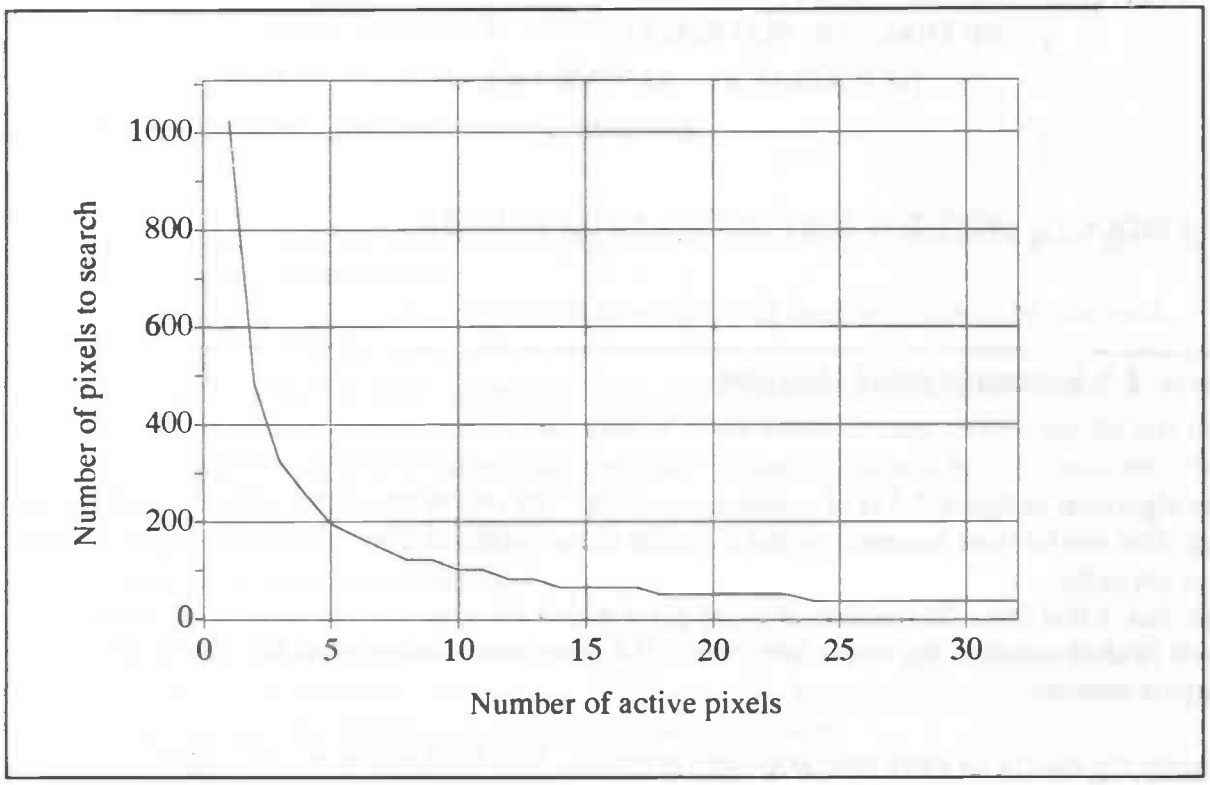

Figure 2.2: Number of pixels that have to be checked by the systematic search algorithm.

Obviously this graph is not complete. The systematic algorithm still has to find all other active pixels.

To jump through the grid like this, a list of available pixels is not enough. A table(matrix) of all coordinates (including the ones that are not actually there) has to be provided. This is because the algorithm must be able to jump to every position on the grid with a minimum of delay.
If the complete matrix is not provided, calculations will have to be done in order to find certain positions and that would be too costly.

First the search algorithm:

```
STEP THROUGH MATRIX(Y) UNTIL FOUND OR Y=Ly
        [STEP THROUGH MATRIX(X) UNTIL FOUND OR X=Lx
                [IF PIXEL(X,Y) = ACTIVE THEN FOUND = TRUE;]
        ]

FIND(X,Y);  //GET THE REST OF THE ACTIVE PIXELS
```

Figure 2.3: Systematic search algorithm.

The algorithm in figure 2.3 is of course incomplete. The rest of the object still has to be identified. This can be done recursively, but it entails extra overhead from which the simple algorithm does not suffer.
The idea is that from the position of every pixel which the algorithm finds active, a search is made in all directions (up, down, left, right). If it finds a new active pixel the search starts again. If not it returns.

Besides the matrix an extra field is needed per pixel. This field can be used to register whether a pixel has already been searched.

```
FUNCTION FIND(X,Y)
  [IF EXIST THEN IF NOT VISITED THEN
        SET VISITED
        IF ACTIVE THEN
                [
                CALCULATION OR SAVE COORDINATES+VALUES FOR LATER
                IF X<>1 THEN FIND(X-1,Y);   //STEP LEFT
                IF Y<>Ly THEN FIND(X,Y+1);  //STEP DOWN
                IF X<>Lx THEN FIND(X+1,Y);  //STEP RIGHT
                IF Y<>1 THEN FIND(X,Y-1);   //STEP UP
                RESET VISITED;
                ]

  ]
```

Figure 2.4: Finding all activated pixels.

Clearly this algorithm takes relatively more time to complete. It only becomes interesting with larger numbers of pixels. It is still an O(N) problem. At most all pixels will be searched and if more pixels are introduced on the grid, the algorithms running time will grow linearly with the number of pixels.

2.3 A speed comparison, method 1 versus method 2

At what point does it become interesting to use method 2 (fast search, recursive pixel gathering) over method 1 (slow, linear search).
To calculate at which total number of pixels this occurs, both methods have to be analysed.

We assume that the object is small compared to N_{total}, for instance $a = 4$ or $a = 9$ pixels. This means that the object activates a square area, even if it has a round shape. It also means that in the worst case, during a systematical search (algorithm 2), the object will not be found until the last pixel is checked.

Now we carry out a worst case analysis:

Let us assume that all instructions can be carried out in 1 clock cycle. We will also assume that each basic operation is equal to 1 instruction. This is possible because 2 algorithms are being compared, so that only the relative execution times are interesting.

Algorithm 1

In the worst case the object will be situated at the bottom right hand position on the grid, so N_{total} pixels will have to be visited and for a pixels the instructions in the if statements will be entered. The total number of basic instructions in the loop is 4, the assignment in the if–statement is 1 operation. The calculations are not counted here, for they are the same in both algorithms.
We then have

$$N_{total} \times 4 + a \times 1 \text{ total instructions} \qquad \text{Eq 2.4}$$

Algorithm 2

As in algorithm 1, in the worst case the object is on the bottom right hand position on the grid and therefore Eq. 2.3 gives the number of pixels to visit during the search:

$\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ pixels. So for the outer loop, the total instructions are $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x 3, for the inner loop $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ x 3 and in the IF–statement 1 instruction is carried out for every pixel visited.

In total this gives: $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x 3 + $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ x 3 + $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ x 1 instr. Eq. 2.5

The recursive pixel gathering has to be added to eq. 2.5:

Each active pixel will execute 4 new function calls. This means $4 \times a$ calls. Each line with a function call will execute 3 instructions, two tests and one call. In each function, after each call, 3 operations are carried out. So we have $4 \times a \times (4 + 3)$ instructions. When coming out of the recursion, $4 \times a$ visited resets are done. This gives: $4 \times a \times (4 + 3) + 4 \times a$
Eq. 2.6
Adding eq. 2.5 and eq. 2.6 gives a total of

$\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x 3 + $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ x 3 + $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil$ x $\left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ x 1 + $4 \times a \times (4 + 3) + 4 \times a$ Eq. 2.7

To compare the 2 algorithms the number of pixels on the grid is plotted against the execution time for a fixed object size (the Matlab file can be found in appendix A).
The following plots are for object sizes $a = 4$ and $a = 9$ pixels:

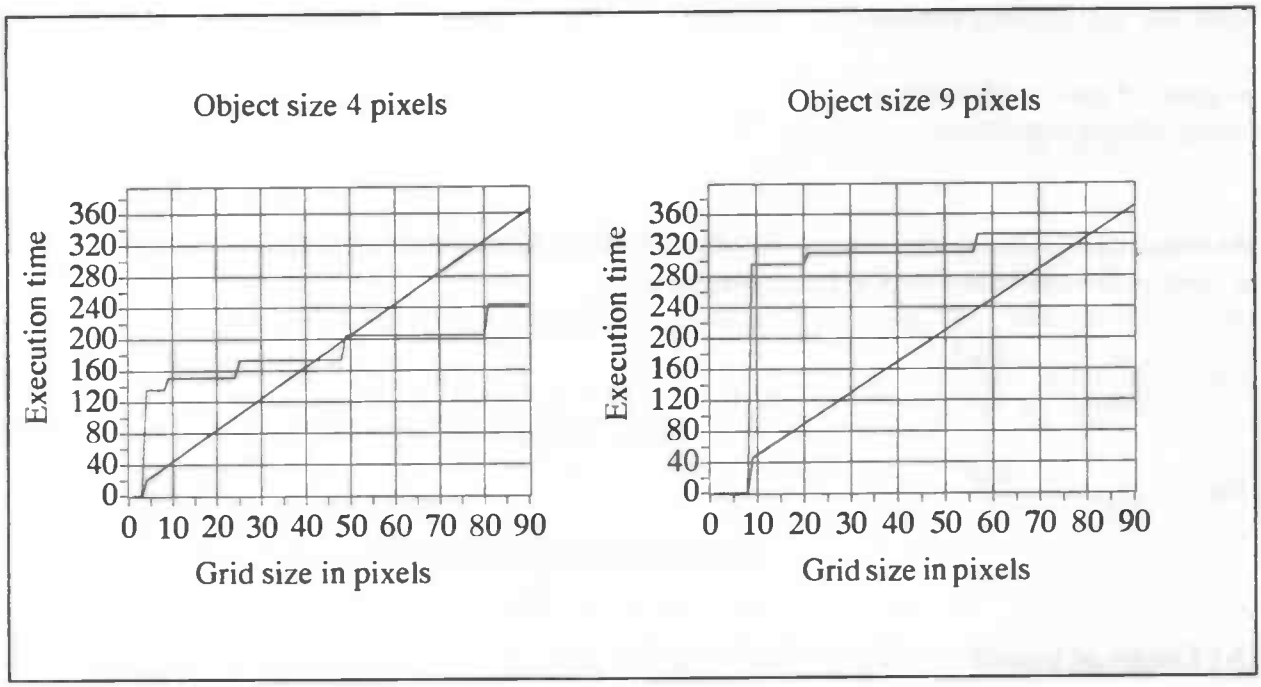

Figure 2.5: 4 and 9 pixel object, method 1 versus method 2.

From these plots it follows that with an object size of 4 pixels, method 2 outperforms method 1 at a grid size of approximately 40–50 pixels. With an object size of 9 pixels the grid size is approximately 80–90 pixels. Thus for each configuration of grid and object size the best method can be calculated.

3. Finding the peak

To determine the peak position of a distribution $C(x,y)$ different methods can be used. Basically there are 2 interesting methods:

- Center of gravity algorithms
- Peak–fitting algorithms

As stated earlier, the spatial energy distribution $C(x,y)$ has a Gaussian shape. If we separate the x– and y–directions and look at both marginals:

$$C(x) = \frac{1}{\sqrt{2\Pi\sigma}} \times e^{\frac{-(x-\mu)^2}{2\sigma^2}} \qquad \text{Eq. 3.1}$$

$$C(y) = \frac{1}{\sqrt{2\Pi\sigma}} \times e^{\frac{-(y-\mu)^2}{2\sigma^2}} \qquad \text{Eq. 3.2}$$

3.1 Centre of gravity

This algorithm is based on the following principle:
We want to find the maximum likelihood (ML) estimator $\hat{\mu}$ for μ. This can be calculated in the following manner:

Assume $x_1, \ldots\ldots, x_n$ are independently distributed from a normal $\mathcal{N}(\mu,\sigma)$ distribution. Then

$$C(x) = \frac{1}{\sqrt{2\Pi\sigma}} \times e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2} \qquad \text{Eq. 3.3}$$

The log–likelihood is then given by

$$C(x) = -\frac{n}{2}\log 2\pi - n\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 \qquad \text{Eq. 3.4}$$

The ML–estimator is calculated

$$\frac{\delta}{\delta\mu} = 0 \text{ , so } \sum_{i-1}^{n} x_i - n\mu = 0$$

Thus the ML–estimator $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$ Eq. 3.5

In our case we have a slightly different situation. To determine a peak position the weighted average is used. This means that every position on the grid is weighted by the number of photoelectrons that fall on it (= the total energy that falls on to a pixel). The peak is calculated as follows:

$$X_{centre} = \frac{\sum_{x} x * E_{x,y}}{\sum_{x} E_{x,y}} \qquad \text{Eq. 3.6a}$$

$$Y_{centre} = \frac{\sum_{y} y * E_{x,y}}{\sum_{y} E_{x,y}} \qquad \text{Eq. 3.6b}$$

$$E_{total} = \sum_{x,y} E_{x,y} \qquad \text{Eq. 3.6c}$$

Effectively the same principle as eq. 3.5 is used to calculate the average. Eq. 3.6a can be written as:

$$X_{centre} = \frac{(x_{1_1} + x_{1_2} + ... + x_{1_{E1}}) + (x_{2_1} + x_{2_2} + ... + x_{2_{E2}}) + ... + (x_{1_1} + x_{x_2} + ... + x_{x_{Ex}})}{E_1 + E_2 + ... + E_x} \qquad \text{Eq. 3.7}$$

where,

x_{i_j} = photoelectron j at x–position i.

E_x = total number of photoelectrons (energy) at position x.

So by averaging all measured values, the peak will be identified correctly. This of course is in an ideal situation in which there are an infinite number of measurements. In real life this doesn't happen.

So the peak μ will not always be correctly estimated by $\hat{\mu}$ but will be biased. Simulations can be carried out to discover how accurate the calculations will be.

The algorithm is a very fast one. The following pseudo–code does the trick:

```
LOOP LIST OF ALL ACTIVE PIXELS (ALREADY COLLECTED OR ON THE FLY)
        [Xtotal = Xtotal + (Xposition x value);
         Ytotal = Ytotal + (Yposition x value);
         Etotal = Etotal + value;
         ]

//CALCULATE POSITIONS: (at the end)

Xpeak = Xtotal/Etotal
Ypeak = Ytotal/Etotal
```

Figure 3.1: Determining centroid.

This again is an O(N) problem: the algorithm has an execution time that grows linearly with a. The maximum number of pixels activated is N_{total}.

This algorithm can be used either on the fly, doing calculations as each active pixel is collected or after all pixels have been collected. Using the second possibility implies keeping a list of active pixels which will slow down the algorithm, therefore this is not recommended. Incorporating this algorithm with one discussed in chapter 2 is faster.

3.2 Peak fitting

Another possible method for determining the peak position is by making use of the fact that we have a Gaussian (normal) distribution. A Gaussian curve is determined by a minimum of 3 points. There are several methods of peak–fitting. Most of them mean fitting a number of detected points to a Gaussian distribution function with estimated parameters and calculating the least square error of the fitted function. With each iteration the parameters of the Gaussian function are estimated to get a function that is closer to the Gaussian that was detected. It is already quite clear that this algorithm is very slow compared to the centre of gravity algorithm because several iteration steps are needed. As the highest speed possible is required here, these types of algorithms are not interesting.

Therefore peak fitting will not be explored any further.

4. Analysis of accuracy

It is interesting to know how well the centre of gravity algorithm performs. There are 2 reasons for this:

- To find out how accurate this method is.
- To find out how many pixels have to be active to give the best results. There are 2 problems connected with the number of pixels used:
  - More pixels means more calculation time, thus lower speed performance.
  - More pixels means that less area on the grid can be used effectively.

4.1 Centre of gravity method

To assess the accuracy of the centre of gravity method, a simulation is done in Matlab. The Matlab files used in the simulations can be found in appendix A. The simulation is set up as follows (1 dimensionally):

- Create a Gaussian distribution of a number of photoelectrons that fits exactly on a number of pixels (assumption: width of total Gaussian is 6σ).
- Sort the generated photoelectrons into bins that represent pixels.
- Use the algorithm to calculate the peak position.

The Matlab files for these simulations can be found in appendix A.

3 questions were asked:

- How does the number of activated pixels (with the same number of photoelectrons) influence the accuracy?
- How does the number of photoelectrons change the accuracy?
- How does the position of the activated spot on the grid affect the accuracy?

The following results were obtained form 1000 simulations per distribution. Figure 4.1 shows the results with 100 photoelectrons, figure 4.2 with 200 photoelectrons and finally figure 4.3 with 1000 photoelectrons. All three were tested with 1 pixel to 10 pixels activated. In all graphs the average errors and the maximum errors in pixel percentage are plotted vertically, whilst the number of pixels is plotted horizontally. The minimum errors are always 0.

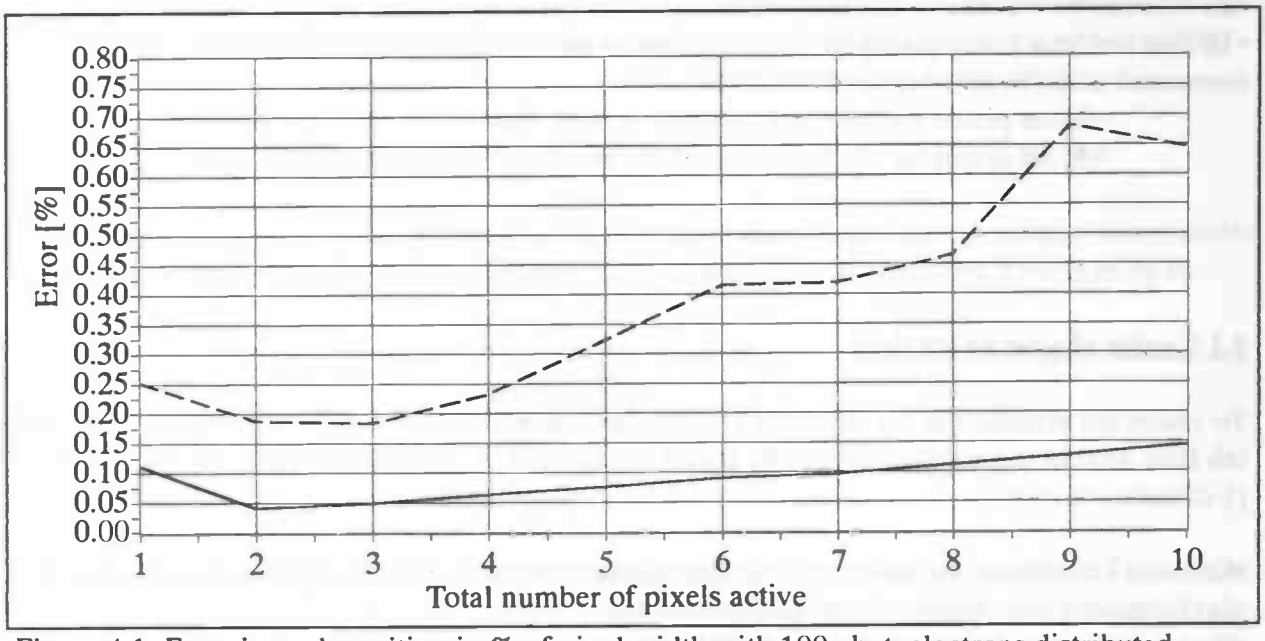

Figure 4.1: Error in peak position in % of pixel width with 100 photoelectrons distributed.

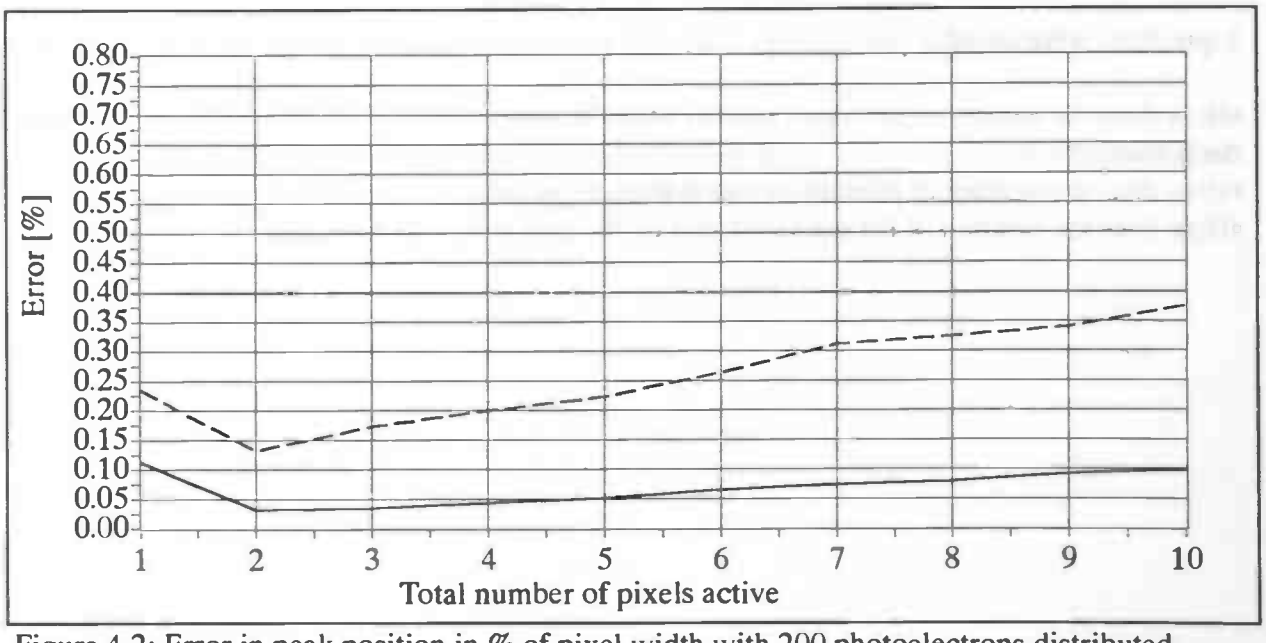

Figure 4.2: Error in peak position in % of pixel width with 200 photoelectrons distributed.

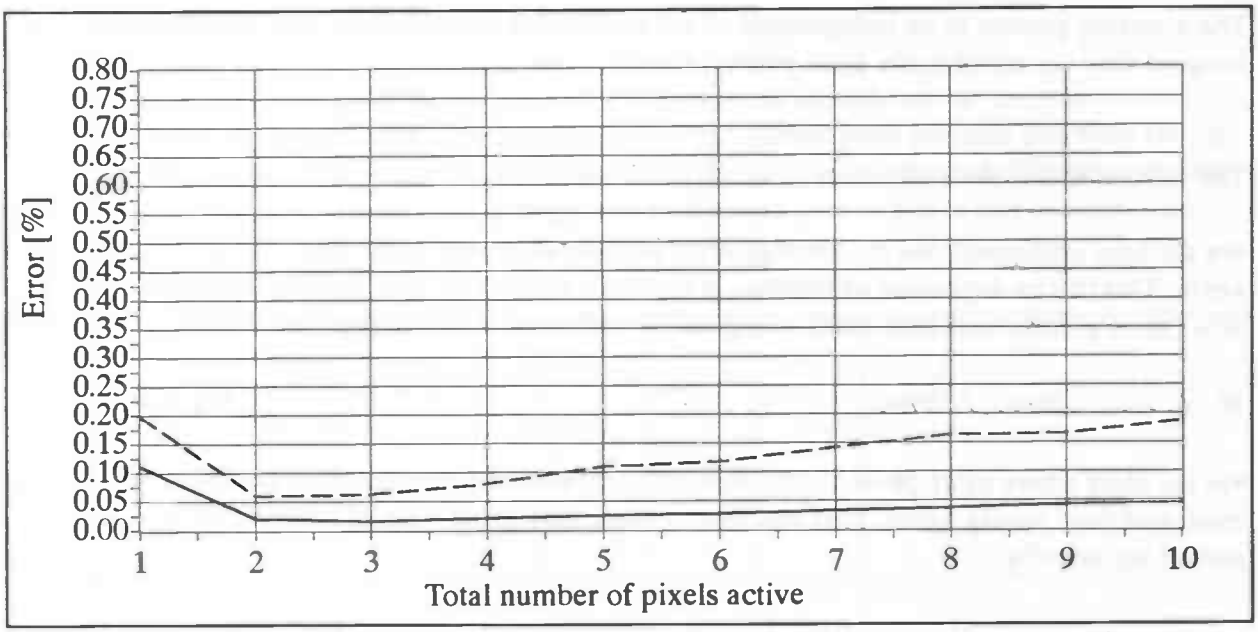

Figure 4.3: Error in peak position in % of pixel width with 1000 photoelectrons distributed.

To make a better comparison, here are all 3 averages in one plot:

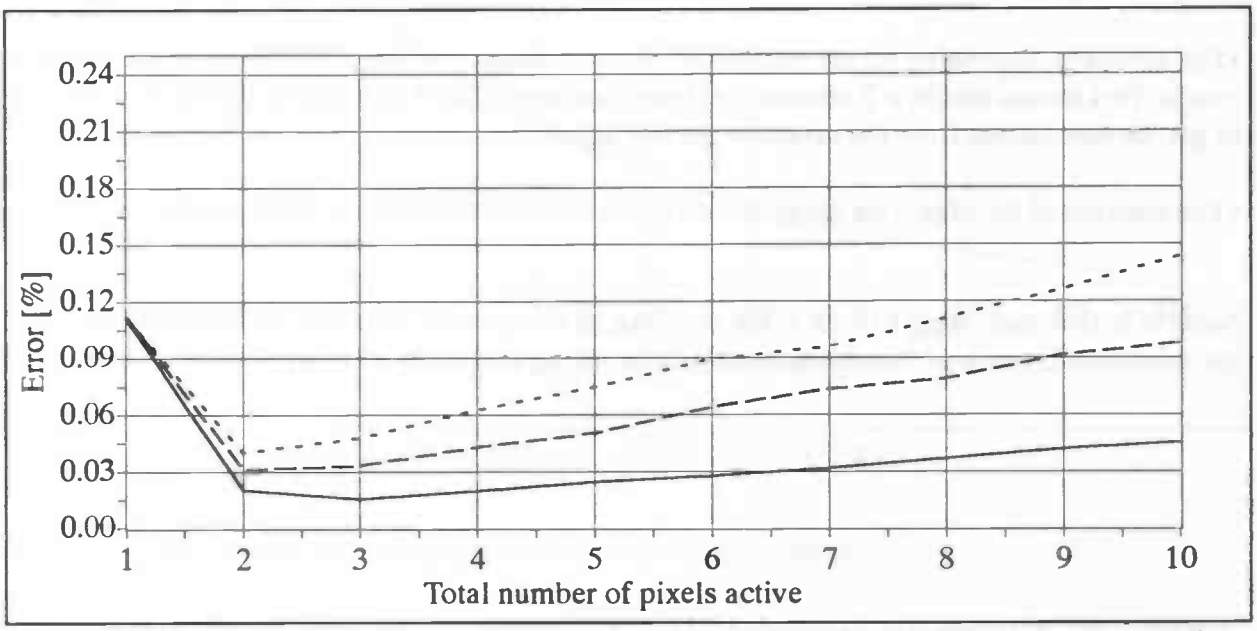

Figure 4.4: Error in peak position in % of pixel width for 100, 200 and 1000 photoelectrons distr.

The accuracy proved to be independent of the position of the activated spot (results not shown because they are equal to the plots printed above).

The following things can be seen in the plots:

•In all three configurations the average error initially decreases quite fast, but then starts rising again. This can be explained as follows: if the same amount of data is distributed over a wider area (more pixels) then each pixel will give less information about the distribution.

$$IF \quad \sigma_1 > \sigma_2 \Rightarrow Error_{\sigma_1} > Error_{\sigma_2} \qquad \text{Eq. 4.1}$$

•In the plots where more photoelectrons were generated, the average error reaches a lower minimum and rises slowly again. This means that more data gives a better result. This is to be expected statistically.

Conclusions

3 conclusions can be drawn from these results:

•The accuracy of the algorithm improves as the number of photoelectrons increases. This means that efforts should be made to ensure that as many photoelectrons as possible are generated.

•The accuracy, depending on the number of photoelectrons, reaches a minimum value at 2 or 3 pixels. This means that in a 2–dimensional case not more than 4 or 9 pixels should be activated to get the best results from the centre of gravity algorithm.

•The position of the object on the grid does not influence the accuracy significantly.

Note: it is also interesting to look at the variance of every error. This will rise much faster after the minimum is reached, but the optimal number of active pixels will stay the same.

4.2 Usable area

If part of the information is missing because the object is not completely on the grid (at the edge), the centre of gravity method will not calculate the correct peak position and total energy. Therefore it has to be ensured that the complete object is situated on the grid. This also implies that if the activated number of pixels is large, the usable area (where peaks can be calculated) decreases.

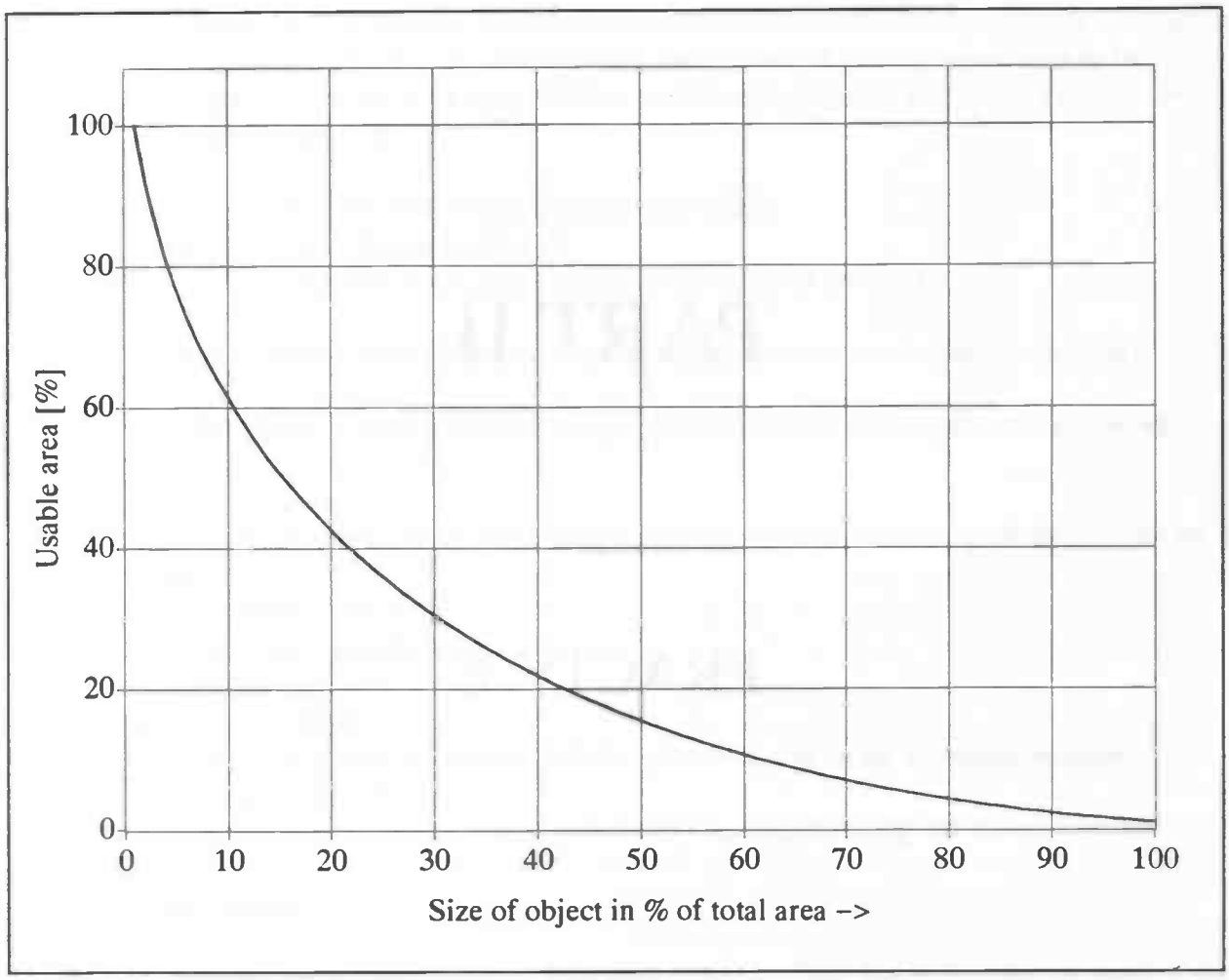

Figure 4.5: Usable area in % of total area with different object sizes.

The Matlab file to generate this graph can be found in appendix A.

Care has to be taken that objects are not too large. From chapter 4.1 it is clear that the optimum number of pixels that an object should consist of is 4 or 9. At 4 or 9 pixels the usable area of the grid is about 65 – 80%. So to have a usable area of 80%, the object should consist of not more than 4 pixels.

Using not more than 4 or 9 pixels would benefit speed performance, because fewer calculations have to be done.

PART II

PRACTICE

5. Objectives

The project has the following objectives:

1. To enable serial data to be read out from two different types of detectors (PSPMT & HPD) using a fast ADC system. The on-board DSP will be used for data-handling/reduction. The PC will be used for display and optimization of the images.

2. The development of a flexible system, ideally selected from the front-end prior to running the data collection software. For the PSPMT the number of x and y wires have to be interchangeable. For use with the HPD the number of hexagonal pixels or square pixels has to be interchangeable.

3. The development of an event reconstruction algorithm:
   two methods: peak fitting (optional)
   centroid with variable threshold(centre of gravity)

   Optional: a testmode to display x and y profiles per selected event.(diagnostic only).

4. The incorporation of energy correction by using calibration data to generate a look-up table.

5. Display of image using image processing techniques such as (updating image as often as possible):
   - maximum entropy
   - maximum likelihood

6. With options to display:
   energy spectrum with the possibility of being able to set an energy window.
   x and y projections
   display image using the data selected by setting/changing the energy window

Other important points:

•The aim is to design a real-time system. This means that the DSP has to be used for most of the data processing (points 3 & 4). The host-PC is only used for tasks such as displaying, etc.

•Hardware: a DATEL ADC/DSP board with complimentary software will be provided.

•The programming language should be C (there is a C-compiler provided with the DSP). For the front-end software a software package such as Visual C would be more appropriate.

•Because it is likely that for certain applications the source code has to be changed, it is important that the software is well documented. Also it would be of great help if part of the program could also be used on more sophisticated systems with VME-interfacing.

6. The project

The aim of the project is to develop software that can read data from different kinds of position sensitive detectors and build images from that data. An image is generated by calculating the peak positions of incoming gamma–photons and counting how many events occur at each position. The total number of photons that occurred at a certain position gives a good idea of the activity of the source at that position.

The complete system can be schematically shown as follows:

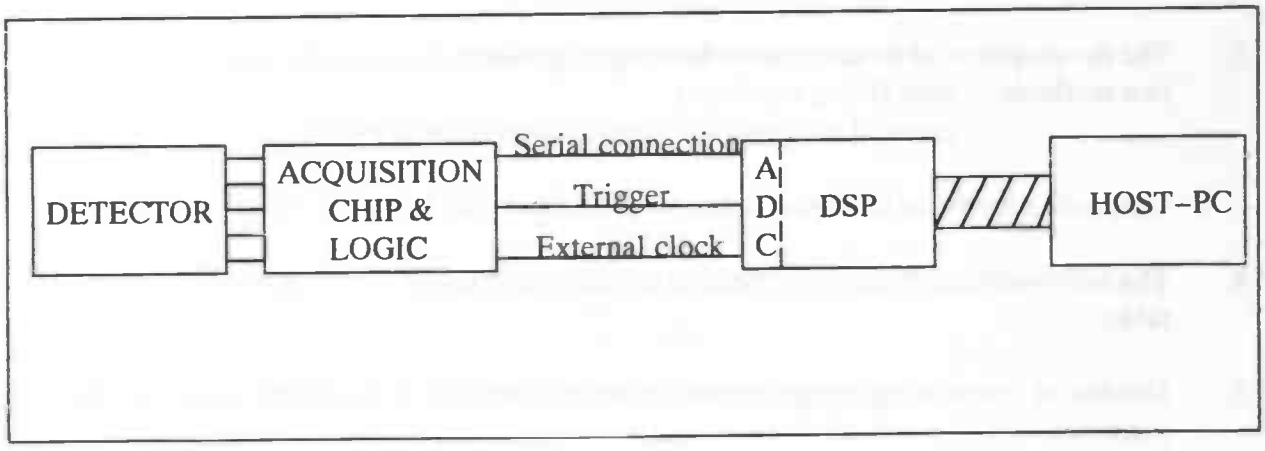

Figure 6.1: Schematic overview of the system.

The whole process:

- The detector detects an event(gamma–photon).
- The signal from each pixel is amplified (parallel) and shaped in the acquisition (VA/HX2)–chip.
- The acquisition–chip & logic produce a trigger and a clock and emit the data serially.
- The ADC on the DSP reads in the data and processes it (calculates positions + energy).
- The PC reads in the processed data from the DSP and builds an image.

6.1 The detectors

Initially the system must be capable of processing data from three different types of multi-pixel detectors:

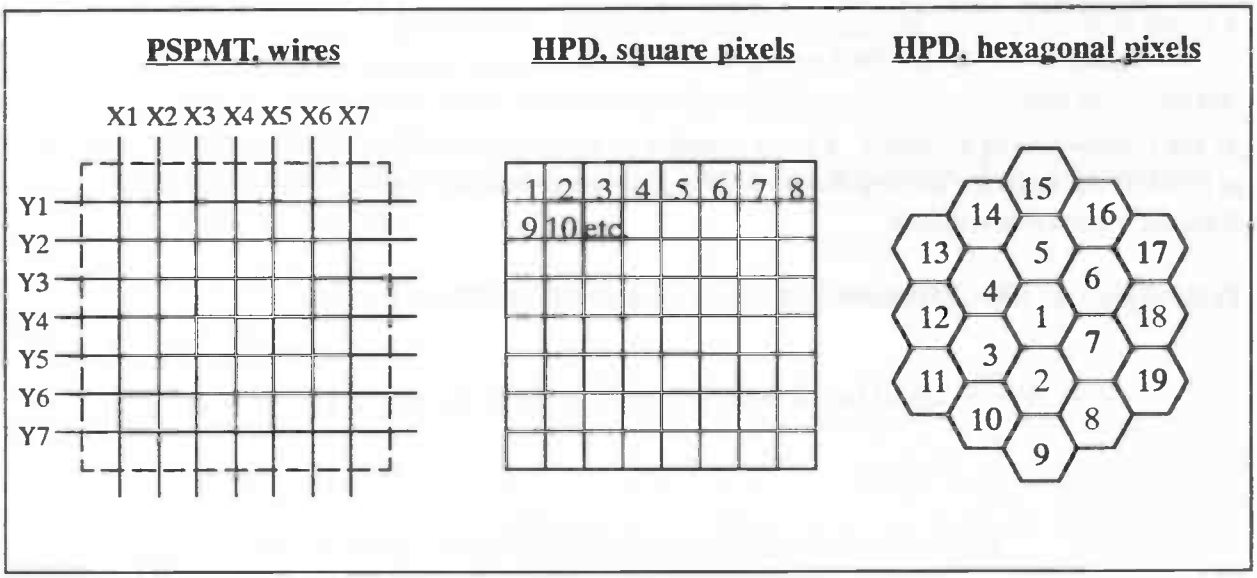

Figure 6.2: The detectors.

1. PSPMT (wires).

   This detector consists of a number of vertical and horizontal wires, crossing at right angles, that can be hit. A number of x-wires en y-wires will be hit at the occurrence of each event (the release of a gamma photon).

   Response: not uniform (100 – 300%).

   The response difference between two points on the detector surface can sometimes be 4 to 1. This occurs because, due to the production process, the photocathode layer produces more electrons at certain positions than at others (at the same light intensity).

2. HPD (square pixels).

   These detectors consist of a number of square pixels that can be hit. A number of pixels will be hit at the occurrence of each event.

   Response: very uniform (< 1% difference).

   The response is very uniform over the whole detector surface. This is because of a better production process of the photocathode layer.

3. HPD (hexagonal pixels).

These detectors consist of a number of hexagonal pixels that can be hit. A number of pixels will be hit at the occurrence of each event.

Response: very uniform.

See 2.

At the occurrence of an event, a large number of light–photons are created using the scintillators in front of the detector's entry window. The photons produce a lightspot on the (multi–pixel) detector's entrance window.

Typically about 100–200 photoelectrons are deposited during each event.

6.2 Acquisition–chip & Logic

The output of each of the wires or pixels is enhanced using a pre–amplifier. The resulting signal is then converted into a Gaussian–curve by a shaper. The VA–chip provides these services and also allows the incoming (parallel) data from the various wires/pixels to be transmitted serially to the DSP.
The wires or pixels can be transmitted in a predetermined sequence. At the start of each event a trigger–pulse is transmitted in order to activate the ADC on the DSP board.
An external clock is integrated in order to allow synchronization of the serial–data flow and the sampling of the flow by the ADC.
The voltage of the samples can range from 0–2 [V]. This range represents about 100 discrete levels (0 – 100 photoelectrons).

6.3 The ADC/DSP

The ADC samples the data using an external clock and an external trigger (see 6.2).

The DSP has a number of tasks:

- Energy correction using a look–up table generated by calibration data.

  At a fixed light intensity the number of photoelectrons emitted at each position on the photocathode can vary and must therefore be corrected. A correction of the calculated energy value will be applied at the moment that an x, y–position is determined. This is done through a look–up table.

The HPD response is fairly uniform over the whole photocathode surface making energy correction much less important.
The PSPMT response is not uniform. The ratio between responses from 2 points (at a maximum distance from each other) on the photocathode surface can sometimes reach a value of 1:4. Energy correction in these circumstances is extremely important.

- sample mapping.

  The sampling sequence is not identical for each detector (with the exception of the wires: first all the x–wires, and then all the y–wires in sequence). This sequence is unsuitable for the processing algorithm. Because of this problem it is necessary to generate a table with all the sample positions within each event. The algorithm, which is employed to process the samples, can consult this table in order to access them in the required sequence.

- threshold detection.

  An certain amount of noise will always be present on every pixel. This must be filtered out. Through low–noise electronics it is ensured that the noise levels of pixels are much lower than the level a pixel has when it gets hit. By simply setting a threshold, the pixel checking algorithm is able to distinguish between pixels with only noise on them and pixels that are active because they were hit.

- active pixel counting.

  To get a reasonable peak position from the algorithm a certain minimum number of pixels have to be active in an object. If the number is too low, useful calculations cannot be done.

  Conversely if too many pixels are active, it means that there are probably 2 objects present on the grid. If these objects overlap, the algorithms would not give the correct results.

  By counting the number of active pixels the algorithm can decide whether to use the data or not, depending on the minimum and maximum pixel counts that have been entered by the user.

- determining the energy level.

  The energy level must be determined for each event. This is the value produced when all the wires or pixels are added together.

- determining the x,y position.

  To determine the x,y position that the emitted gamma–photon had, the peak position for the event (light spot) must be determined.

6.4 The Host PC

The last step in the process involves the transmission of data to the host–PC. This data consists of an x–position, an y–position and an energy level for each event. After this the host–PC starts processing the received data.

The PC has a number of tasks (These tasks will be executed under the Windows operating system):

- visualization of the x,y positions.(building the image)

  The PC displays the x,y positions with the highest possible update frequency and the least possible data loss. The energy levels will be shown, for example, in grey tints or in colour. The picture is composed as clearly as possible and in the shortest possible time using this method. The image is built by counting gamma–photons on each position.

- displaying the energy spectrum.

  The capability to display an energy spectrum from the incoming data must be available. The x– axis represents the various energy levels and the y–axis the number of occurrences.

- choice of energy window.

  The user must be able to choose the energy window to be processed in the image. In this way undesirable energies can be omitted. Sometimes gamma–photons get scattered. When this happens, the energy level drops considerably and the position of the gamma–photon will be different from the original position. This data should not be used in the image because it would give false results.

  A typical energy response from a pixel caused by an incoming gamma–photon looks like this:

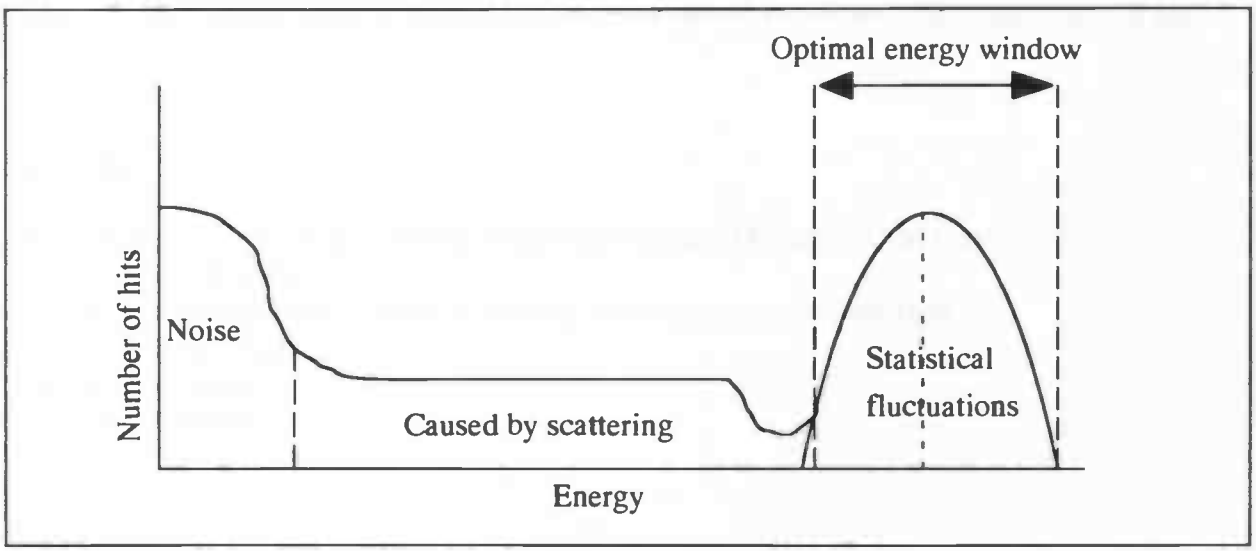

Figure 6.3: Typical energy response caused by gamma–photons.

- smoothing the image.

  Because the image can contain a great deal of extreme colour transitions between the pixels it must be possible to smooth the image by the application of image processing. These transitions occur because certain positions get large photon counts while others get none, depending on the object that is scanned by the detector and the acquisition time.

7. Taking stock

Before developing any practical software we look at the hardware and software that is available and evaluate both.

7.1 Hardware

The following hardware is presently being used :

Detectors:

•PSPMT: Format:
Number of x–wires: 28 (5") and 18 (3")
Number of y–wires: 28 (5") and 16 (3")

Number of active pixels: 3 to 9

•HPD: Format:
Number of square pixels: 25*

•HPD: Format:
Number of hexagonal pixels: 7, 19, 37*, 61*

Number of active pixels on 19 pixel HPD: min. 3

* Not available yet.

VA–chip & logic:

Supplies serial data at circa 20 Kevents per second (this is the target processing speed of the system).

ADC/DSP:

Datel DSP board (PC plug in board, ISA BUS) with a 10 MHz ADC on board.
DSP: TI C30–40 MHz.
512 Kb Dual Ported Ram (1 byte = 32 bits), memory mapped in the Host PC.
8 Kb expansion RAM.
2 Kb internal RAM.
1024 sample FIFO.

Host PC:

A fast Pentium computer.

7.2 Software

The following software can be used:

For the DSP board:

•Datel Scheduler.

This is a programme which, with the help of a function library, can execute simple scripts generated by the user. Only the scripts need to be programmed. All the hardware can be adjusted using this software: AD sample rate, number of samples per trigger, allocation of input– and output– buffers to contain the data, processing of the data etc. etc.
A source code and a TI C–compiler is included in the package. This can be used to make changes in the DSP–code and even to add new functions to the library.

For the Host–PC:

•Datel Commander.

This software makes provision for the placing and execution of the Scheduler in the DSP memory. A script, made by the user, will also be placed in the DSP memory.
Commander also provides for communication with the DSP board and also for data transport from the DSP board to the PC.

The Commander software is available for DOS and Windows 3.x.

Operating system: Windows 3.11 (16 bits) or Windows 95 (32 bits).

Programming Language: Visual C++ 1.52 (16 bits) or Visual C++ 4.0 (32 bits).

7.3 Considerations

Development Time:

In order to limit the development time use can be made of the available Datel software.
This applies to the DSP board and the PC.
A dedicated library function must be added to the DSP. Windows DLLs can be used for the PC (
If necessary these can be adapted with Borland C 3.1).

Speed of data processing:

If use is made of the existing DSP software the speed of data processing will be limited by the Scheduler overhead. Newly developed software can provide more speed.

Application of the existing Windows (Commander) software, means that 16 bit DLLs will be employed. If these DLLs are used under a 32–bits operating system this will result in severe delays (caused by "thunking", when calls are converted from 32 to 16 bits and 16 to 32 bits). This already limits the design to Windows 3.x.

The highest data processing speed would be obtained by working under Windows 95 (32 bits). This creates a new problem because new Commander software would have to be written.
In order to cope with possible interrupts and the communication with the DPR by the PC a new Device driver Developer Kit (DDK) would need to be purchased as an addition to Visual C++.

Financial considerations:

Purchase of the available Windows (Commander) software will cost approx. Fl. 2500,–
Purchase of a DDK kit would cost approx. Fl. 1000,– (+ lot of extra development time).

The future:

If the software is written under 32 bits Windows it will be "up to date". This means that the application will have the complete Windows 95 'look'.
Written under 16 bits Windows the software can be considered as dated.
It is of course possible that Datel will shortly make a 32 bits version of the Commander available. It would then be no problem to update from 16 to 32 bits at little cost.

Conclusions:

•The least time consuming and probably cheapest approach is to:

– Use the existing DSP software and adapt it where necessary.

– Use the available Windows Commander software to write a dedicated application.

Disadvantage : A slower system, working under Windows 3.X.

•The most time consuming and most expensive approach is to:

– Create completely new DSP software.
– Design new 32 bits Windows software.

Advantage: A faster system, working under Windows 95.

Because the system has to be up and running as soon as possible, it is best to use the existing software. This includes the Scheduler on the DSP–board and the Commander software for Windows 3.X.

8. Algorithms and processing on the DSP-board

Here, the different algorithms that will run on the DSP-board are developed.

8.1 Working principles

Before any algorithm is written, the scheduling-system on the DSP has to be described. The DSP-software will work schematically as shown below:

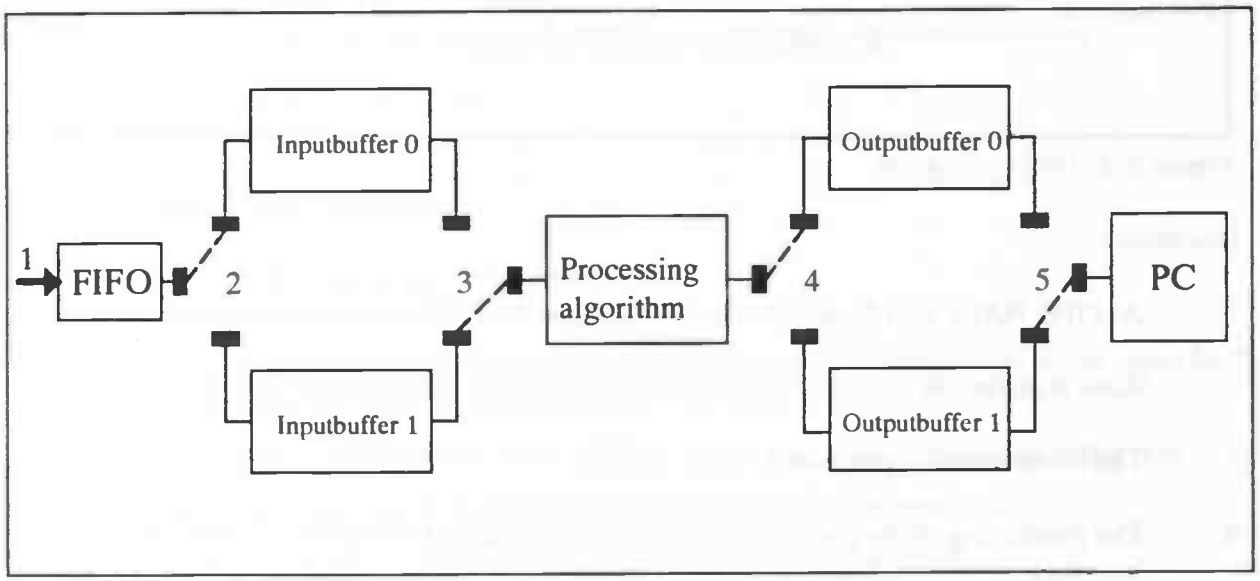

Figure 8.1: DSP-processing.

1. Data is transmitted via the ADC to the FIFO. When the FIFO is half full an interrupt is generated. At this point the FIFO must be emptied. This is to avoid the possibility of losing data. While being emptied, the FIFO is also receiving new data.

2. The FIFO half full ISR (Interrupt Service Routine) always fills the buffers alternately. If the capacity of the input buffer is greater than the FIFO, more FIFO transfers to one buffer can occur before the buffer is filled completely.

3. The algorithm will not begin processing until an input buffer is filled with data. A status bit is set at that moment to indicate that the input buffer is being processed. To avoid loss of data it is essential that the processing of the buffer is faster than the filling of the other buffer with new data. The FIFO will not wait for the processing algorithm to finish.

Timing diagram (|inputbuffer| = |2xFIFO| is assumed):

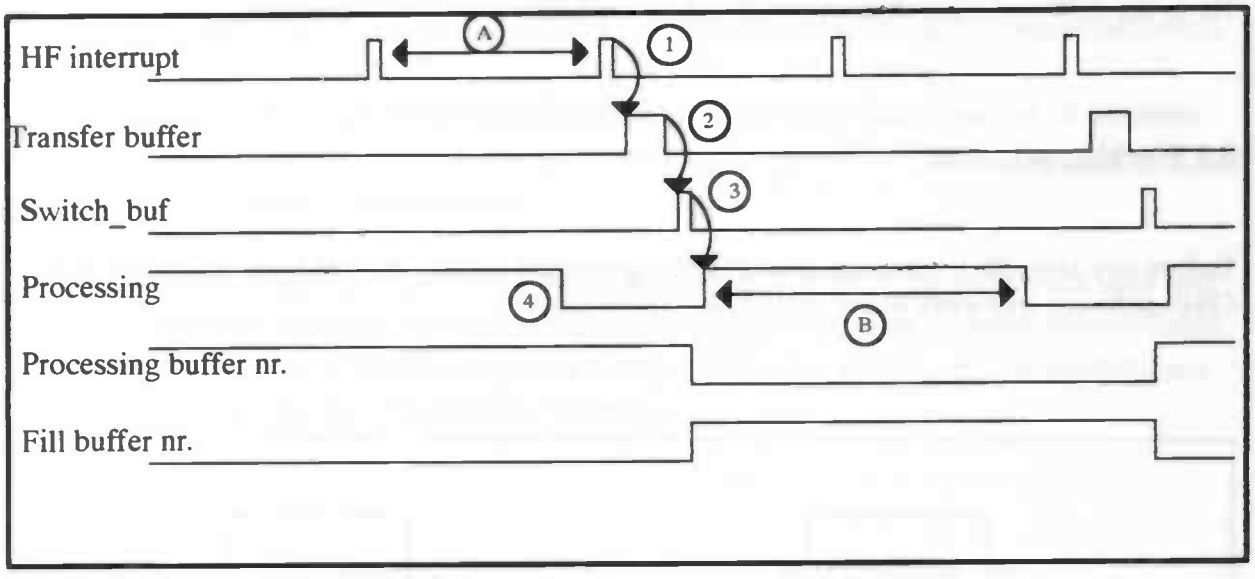

Figure 8.2: Timing diagram.

Discussion:

1. At FIFO HALF FULL, an interrupt occurs and the FIFO contents is transferred.

2. After transfer, the inputbuffer is switched in order to fill the other buffer.

3. The filled buffer is processed.

4. The processing of the previous buffer has to be finished before the second FIFO interrupt.

A. Time between interrupts

Event rate: ER events/second, where ER = 20.000 (aim).
Event size: S pixels or wires, where $7 \leqq S \leqq 61$ for now.
FIFO size: 1024/2 = 512 samples interrupt at FIFO half full).

Now from this we can calculate the time between two FIFO interrupts:

The data rate DR will be : $DR = ER \times S \quad [samples/s]$
This means an interrupt frequency of $\frac{DR}{512} = \frac{ER \times S}{512} \quad [interrupts/s]$
So the time between two interrupts will be $\frac{512}{ER \times S} \quad [s]$

In the worst case (20.000 events/s at 61 pixels) this would be :

$$\frac{512}{20.000 \times 61} = 0.42 \times 10^{-3} \quad [s]$$

B. Processing time (roughly)

CPU speed: 40 Mhz gives about 20 MIPS.

We will assume that the processing time that is available is independent of the input buffer size, and that the |FIFO| = |input buffer| .
The processing algorithm will have $\frac{512}{ER \times S} \quad [s]$ to carry out calculations. After this amount of time the data is lost because it gets overwritten by new data.
So we have $20 \times 10^6 \times \frac{512}{ER \times S}$ instructions do calculate $\frac{512}{S}$ events.
This means $20 \times 10^6 \times \frac{512}{ER \times S} \times \frac{S}{512} = \frac{20 \times 10^6}{ER} \quad [instructions/event]$

In the worst case (20.000 events/s at 61 pixels) this gives

$$\frac{20 \times 10^6}{20000} = 1000 \quad [instructions/event]$$

From the previous equation it seems to follow that the number of instructions per event is independent of the size of the input buffer and that the choice of the size is free. This is not true :

•The input buffer size must always be a multiple of the FIFO size (software limitation).

•The size of the input buffer must be chosen in such a way that a number of complete events will exactly fit into it. The processing algorithm will not be able to cope with partial events (speed consideration).

•The processing algorithm has an initialization part that has to be carried out each time the algorithm is called. The algorithm is called once every time an inputbuffer is filled. This implies that it is better to choose a large buffer size. In this way the overhead per event, caused by the initialization, is limited.

Points 4 & 5 (from diagram 4.1) will be discussed on the following pages.

Interrupt driven versus polling

There are two possible methods of transferring data buffers from the DSP's DPR to the PC memory.

•Interrupt driven

The DSP generates an interrupt to the PC if an output buffer has been filled and a status bit has been reset by the PC to indicate that the previously downloaded output buffer has been processed. If the bit has not been reset, the PC is still busy processing and the same output–buffer will be overwritten. If the bit has been set by the PC, an interrupt is generated to the PC and the status bit is set again. At this point the PC gets the address of the buffer, downloads it and starts processing it.

•Polling

This works almost the same as interrupt driven buffer transfer. The difference is that in this case the PC doesn't get interrupted, but instead, when finished with processing a buffer, the PC starts polling the DSP–board for the next full buffer. The problem here is that when the PC polls the DPR directly, access to the DPR by the DSP is not possible (and vice versa). Arbitration logic on the DSP–board is used to decide which side gets access. This will probably cause speed reductions in data processing. A connection can be made between an unused (internal) command–register bit in the DSP and a PC–IO register bit. This can be used to indicate that a buffer is ready. This IO register is not part of the DPR on the DSP–board and will therefore cause no speed problems. The PC can poll this bit until it is set high and then download a buffer.

ISA–BUS transfer rate

The PC bus operates on a frequency of 8 MHz. If 8 bit transfers are done, this is fast enough to download roughly 8 Mb of 8 bit bytes to the PC. The DPR on the DSP–board works with 32 bit bytes, so the transfer speed would be 8/4 = 2 Mb of 32 bit bytes/second. At a preferred data processing rate of 20.000 events/second with each event giving an X position, Y position and Energy, this implies 3 x 20.000 = 60.000, 32 bits, bytes/ second. This is much smaller than the maximum bus rate.

The following timing diagrams describe the polling mechanism on the PC–host and the DSP

Timing diagram (polling, PC side):

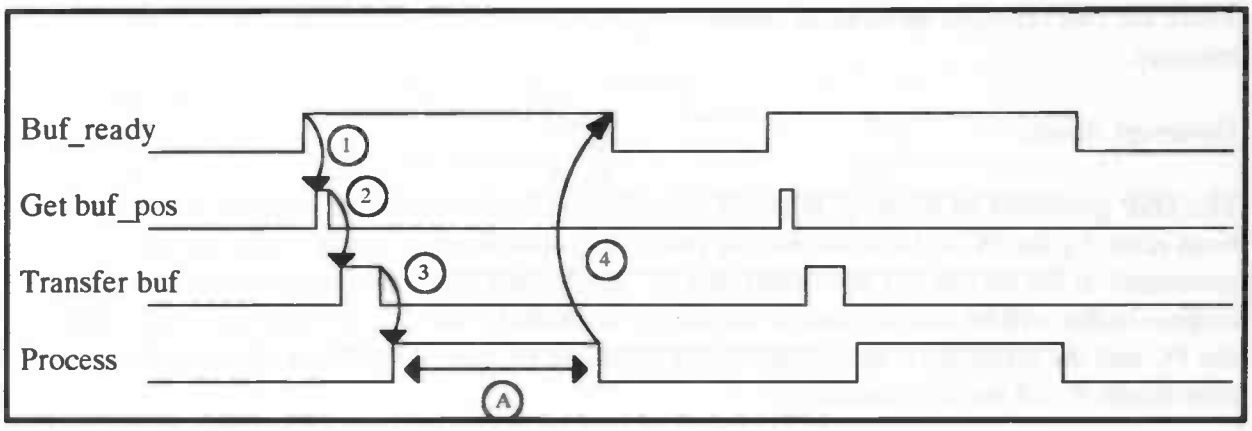

Figure 8.3: Polling, PC–side.

1. When buf_ready becomes high (read by polling), the buffer position is downloaded.

2. Transfer the buffer to the PC memory.

3. Start processing the data.

4. Set buf_ready low to indicate that the PC is ready for the next buffer

A. PROCESSING TIME (PC side).

This is achieved by using the following algorithm on the PC :

```
IF buf_ready THEN
        [TRANSFER BUFFER;
         PROCESS DATA;
         buf_ready = 0;
        ]
ELSE WAIT/POLL  // until interrupt or buf_ready
```

Figure 8.3a : the interrupt/polling algorithm on the PC side.

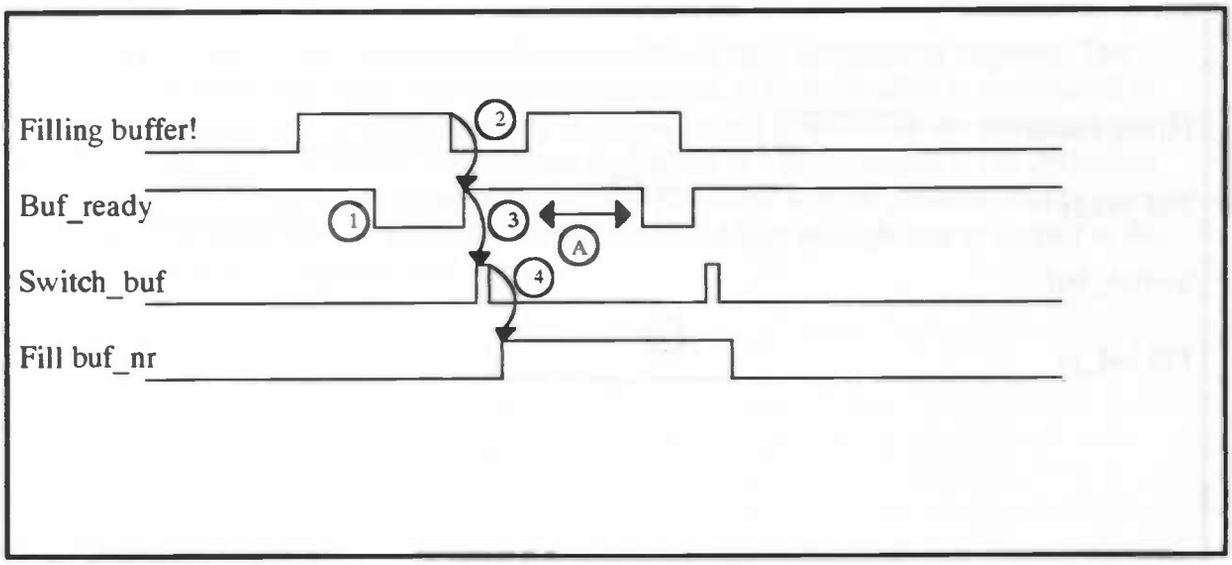

Figure 8.4: Polling, DSP side, PC finished in time for next buffer.

1. Buf_ready becomes low, indicating that the PC is ready to receive a new buffer.

2. When the buffer is filled, buf_ready is set high to indicate this to the PC.

3. The buffer is switched so that the other buffer will be filled.

A. PC PROCESSING PERIOD.

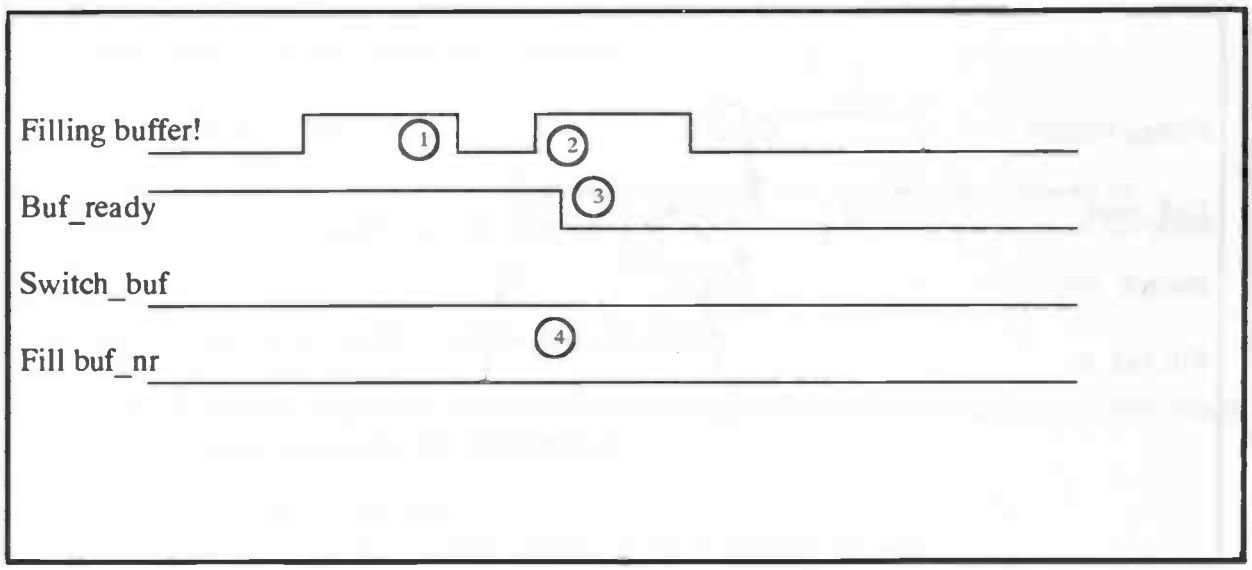

Figure 8.5: Polling, DSP-side, PC not finished in time for next buffer.

1. A buffer has been filled, but the buf_ready bit is high, so the same buffer gets filled again.

2. A buffer has been filled, but the buf_ready bit is high, so the same buffer gets filled again.

3. Buf_ready becomes low, indicating that the PC is ready to receive a buffer.

4. No buffers are switched and the fill buf_nr stays the same.

The following algorithm on the DSP is called when output_buffer is full:

```
IF !buf_ready THEN
        [buf_ready =1;
        // interrupt the PC;
        SWITCH o_buf;  // switch output buffers;
        ]
ELSE  // fill same buffer again from the beginning
        o_buf_overflow = o_buf_overflow + 1; //indicate an overflow occurred
```

Figure 8.5a : the interrupt/polling algorithm on the DSP side.

8.2 Energy correction table

The PSPMT detectors need energy correction because of their not uniform response. The corrections are made after a position has been calculated. A look–up table is constructed as follows: a lightbeam with a certain intensity is pointed at all positions on the photocathode. The detector response is collected for all positions and stored in a table (matrix). The correction resolution is determined by the number of positions recorded. If peak positions can be calculated at for instance 10% of a pixel, the correction table should hold enough data to correct at this resolution. If not, the resolution will degrade.

8.3 Sample mapping

The sample mapping is necessary for 2 reasons:

•The variable sample sequence.

Because the sample sequence can be different for each detector, a kind of map has to be constructed to make it possible for the algorithm to access every detector position in the input buffer.

•To create a mapping independent of the detector type.

To make the processing algorithm the same for every type of detector, a mapping is constructed that creates the same properties for all detectors.

The mapping is created as follows:
The user draws a grid over the detector surface in the following manner:

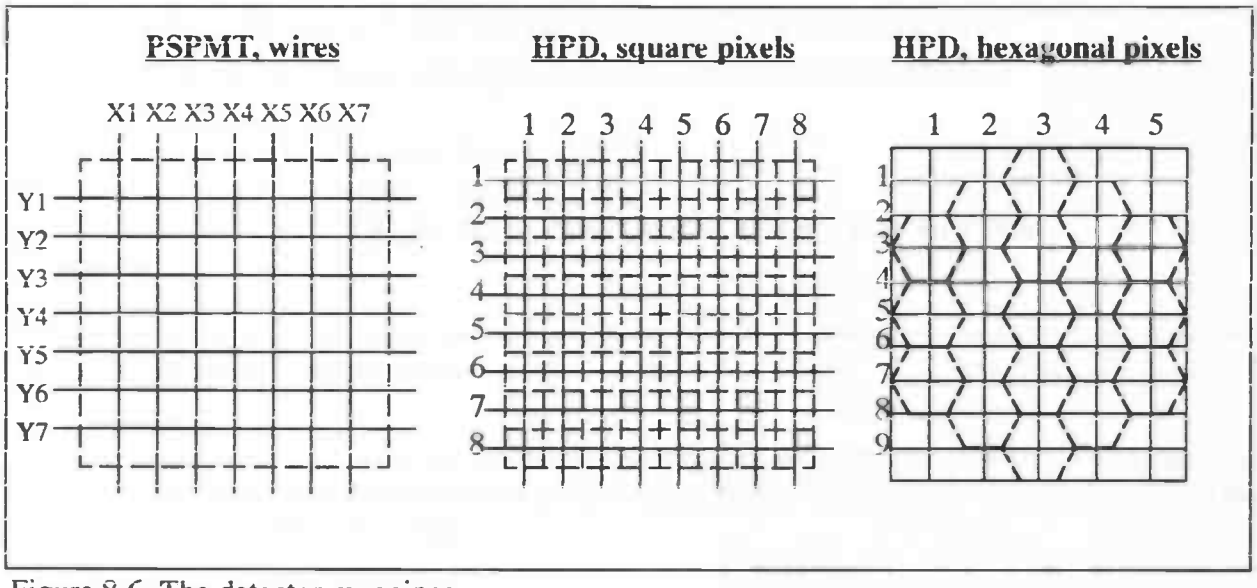

Figure 8.6: The detector mappings.

Depending on the type of algorithm that is used, two kinds of mappings can be created (see chapter 2.1 & 2.2):

•Method 1:

A list is made, specifying coordinates in the grid where pixels are present, with the pixel number they contain (this is the number in the sample sequence). This is done from the left to the right, starting at the top row. This creates a list of coordinates with pixel numbers. This list can be uploaded to the DSP–board.

•Method 2:

A matrix is made, specifying all coordinates on the grid, whether there are pixels present or not, with the pixel number they contain. In this manner, the algorithm can access any coordinate on the grid if necessary. This matrix can be represented by a list and then uploaded to the DSP–board, where the algorithm can access it as a matrix.

8.4 The processing algorithm

The processing algorithm will provide the following services:

•Determination of total energy: all the relevant x– and y–values added together.

•Determination of the peak position: determine weighed average of the wires/pixels.

The centre of gravity is determined as follows:

$$X_{centre} = \frac{\sum_{x} x * E_{x,y}}{\sum_{x} E_{x,y}} \qquad Y_{centre} = \frac{\sum_{y} y * E_{x,y}}{\sum_{y} E_{x,y}}$$

$$E_{total} = \sum_{x,y} E_{x,y}$$

An algorithm that can be applied to all types of detectors is preferred. For method 1 this is easily done by using the mapping from section 8.3. We have a list of coordinates with sample positions within an event. The mapping makes the algorithm independent of the type of detector. With method 2, it is much more difficult, as the pixels have a different layout on the grid. The systematic scanning of the grid is different for all types and so are the recursive jumping directions. In this way it is not possible to use exactly the same algorithm for all three detector types without adjustments. Although one general algorithm could be written for all three detector types, its efficiency would be severely curtailed by extra overhead. A better choice therefore, is to write a dedicated algorithm for each detector type.

Method 1

The algorithms in figure 2.1 & 3.1 can be combined so that calculations are done on the fly, as the active pixels are encountered. The search algorithm also has a threshold checking function.

```
LOOP LIST OF X,Y COORDINATES UNTIL EOL OR PIXELS = PCOUNTMAX
[
IF PIXEL(X,Y) > THRESHOLD THEN
                    [
                     PIXELS = PIXELS + 1;
                     Xtotal = Xtotal + (Xposition x value);
                     Ytotal = Ytotal + (Yposition x value);
                     Etotal = Etotal + value;
                    ]
]
// CHECK NR. OF ACTIVE PIXELS AND CALCULATE POSITION
IF PIXELS > PCOUNTMIN AND PIXELS < PCOUNTMAX THEN
                    [
                     Xpeak = Xtotal / Etotal
                     Ypeak = Ytotal / Etotal
                     WRITE Xpeak, Ypeak, Etotal TO OUTPUT BUFFER
                    ]
```

Figure 8.7: Method 1, the algorithm.

If, at the end of an event, it appears to be a double event or there are not enough active pixels, the values are not used. Considering that the percentage of double events amounts only 1 or 2% of the total, this does not occur very often.

Note:

For the PSPMT the mapping will always have 1 coordinate that is 0. In this way the algorithm calculates the peak correctly. For instance: [(1,0), (2,0), ... , (n,0)] for the x–wires.

Method 2

As discussed at the beginning of this chapter all 3 detectors will have separate algorithms. First the HPD with square pixels is discussed:

•HPD, square pixels.

This algorithm is straightforward because it has been designed in section 2.2 for the same grid shape. The algorithms described in figure 2.3 & 2.4 cannot be combined. The search algorithm will have to call the recursive one for each event. The algorithm from figure 3.1 can be incorporated in algorithm 2.4. With slight changes the algorithms will look like this:

```
STEP THROUGH MATRIX(Y) UNTIL FOUND OR Y=Ly
        [STEP THROUGH MATRIX(X) UNTIL FOUND OR X=Lx
                [IF PIXEL(X,Y) > THRESHOLD THEN FOUND = TRUE;]
        ]

IF FOUND THEN
        [
        FIND(X,Y);  //get the rest of the active pixels
         IF PIXELS > PCOUNTMIN AND PIXELS < PCOUNTMAX THEN
                [
                Xpeak = Xtotal / Etotal;
                Ypeak  = Ytotal / Etotal;
                WRITE Xpeak, Ypeak, Etotal TO OUTPUT BUFFER;
                 ]
         ]
```

Figure 8.8: Systematic search algorithm.

If there are not enough active pixels for a calculation, this search algorithm might often ignore the event. Events that have too many or too few active pixels are filtered out when the recursive algorithm returns.

```
FUNCTION FIND(X,Y)
  [IF EXIST THEN IF NOT VISITED THEN
      SET VISITED
      IF ACTIVE THEN
          [
          PIXELS = PIXELS +1;
          Xtotal = Xtotal + (Xposition x value);
          Ytotal = Ytotal +(Yposition x value);
          Etotal = Etotal + value;
          IF X<>1 THEN FIND(X-1,Y);   //STEP LEFT
          IF Y<>Ly THEN FIND(X,Y+1);  //STEP DOWN
          IF X<>Lx THEN FIND(X+1,Y);  //STEP RIGHT
          IF Y<>1 THEN FIND(X,Y-1);   //STEP UP
          RESET VISITED;
          ]
  ]
```

Figure 8.9: Finding all activated pixels.

The stepsize through the matrix can be calculated from eq. 2.3.

•HPD, hexagonal pixels.

This algorithm is very similar to the previous one. There are two differences:

•Jumping through the grid works differently. This can be seen in the mapping in figure 8.6 : left and right jumps are not possible. The stepping sequence in this type of grid is: LEFT–UP, RIGHT–UP, RIGHT–DOWN, LEFT–DOWN.

•The steps in the systematic search are different as can be seen in the next figure:

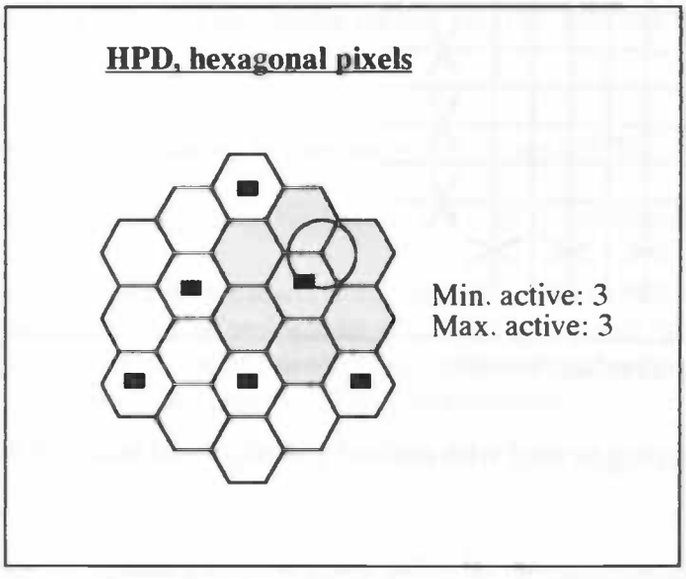

Figure 8.9: Scanning the pixels.

The algorithms are not shown here because the changes are minor.

•PSPMT, wires.

For this detector a different approach is required. A 2–dimensional mapping cannot be created because there are no coordinates.

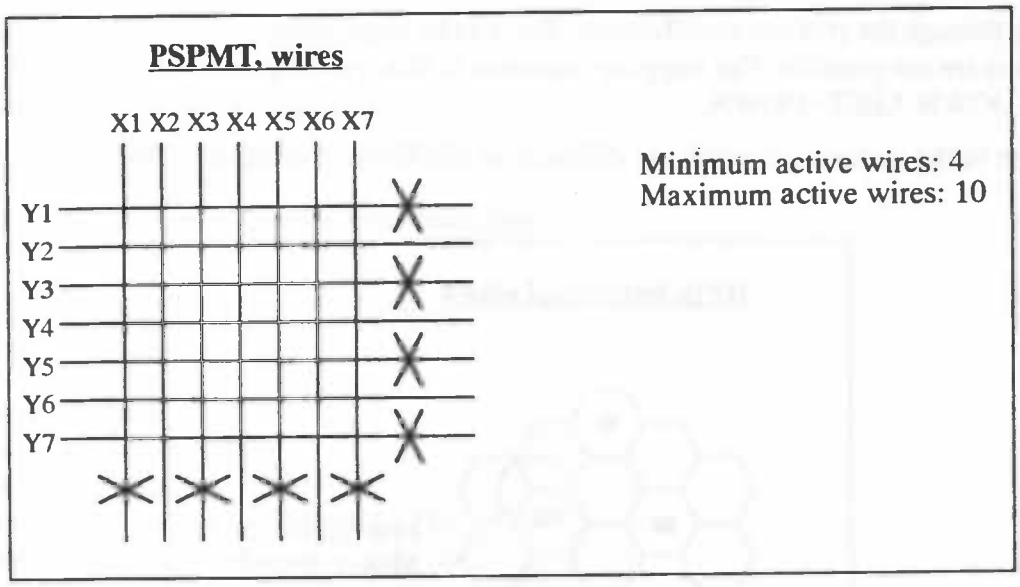

Figure 8.7: Scanning the wires.

A 1–dimensional mapping as used with method 1 is employed here, but a simplified algorithm of method 2 is used.

Assume that the minimum number of active wires is a, on a square grid this is $\sqrt{a}$ horizontally and $\sqrt{a}$ vertically. To ensure that no event is overlooked, $\left\lceil \frac{L_x}{\sqrt{a}} \right\rceil + \left\lceil \frac{L_y}{\sqrt{a}} \right\rceil$ wires have to be checked. When an active wire is detected, the algorithm steps back $\left\lceil \frac{L_x \text{ or } L_y}{\sqrt{a}} \right\rceil - 1$ wires (could be part of the event) and from that wire on checks the next *PCOUNTMAX* wires for the threshold parameter. Each wire that exceeds the threshold is used for the centroid calculation. This has to be done horizontally and vertically.

This algorithm will be the fastest of all: it uses 1–dimensional mapping, systematically searches the grid and is not burdened with the same overhead as the recursive method..

8.5 Reciprocal table

The DSP cannot calculate divisions in 1 instruction cycle. The division is actually quite time consuming. For this reason a reciprocal table is put into the DPR, to ensure fast divisions. Divisions are only carried out at the end of the algorithm, when the x,y positions are calculated. To determine the number of reciprocals needed, the maximum possible value for the divisor has to be calculated:

Depending on the number of pixels, the maximum value of the denominator differs.

$MAXPIXEL = 61$

We know that the maximum value for the energy on one pixel or wire can be

$E_{max} = \frac{2}{5} \times 2048 = 820$

Thus the maximum possible total energy is $Totalmax = E_{max} \times MAXPIXEL = 820 \times MAXPIXEL$

The worst case for $MAXPIXEL = 61$, gives $Totalmax = 820 \times 61 = 50020$ reciprocals.

This is a very large number, so 4 bitwise shifts to the right are done to both the denominator and the numerator. This way the number of reciprocals is divided by 16, and only about 3200 are needed. The division still yields the same answer (only with less accuracy, but this is not important because the 3 LSBs are not significant (see next chapter).

8.6 Accuracy and noise

The accuracy of the system is determined by the following factors:

•Noise in the system

this noise is caused by different factors:

- •Noise introduced by the detector
- •Noise introduced by the electronics

The total noise is described by the following formula:

$$N^2 = \frac{4kTC^2_{in}}{g_m \tau_p} + \left(2eI_{d+}\frac{4kT}{R_p}\right)\frac{\tau_p}{3} + A_{\frac{1}{f}} \quad \text{where}$$

e = electron charge [C]
k = Bolzmann constant [J/K]
T = Temperature [K]
g_m = transconductance of the first stage FET [S]
R_p = parallel equivalent noise resistance [Ω]
I_D = photodiode leakage current (dark current) [A]
C_{in} = total input capacitance (= C_{PD} + C_{FET}) [F]
τ = peak time of output pulse [s]
$A_{\frac{1}{f}}$ = coefficient determining the magnitude of the 1/f noise

This adds up to a noise of about 200 electrons per pixels independent of the number of photoelectrons. Considering that each photoelectron creates about 3000 electron/hole pairs, the noise has a maximum at 1 photoelectron which is 200/3000 = 0.07 photoelectron.

•ADC accuracy

The ADC samples at 12 (signed) bits. This at an input range of –5 to 5 [V].

The input range from the acquisition chip is from 0 to 2 [V], so the sign bit is not used here. Also due to ADC noise, the 3 LSBs cannot be used.

So we have 0 – 5 [V] on 8 bits effectively. This means 256 values over 5 [V].
We have a 0 – 2 [V] range, so this gives 2/5 x 255 = 102 values over 2 [V]. This is exactly what we need, for we have about 100 values that need to be distinguished (see 6.2).

Accuracy of calculations due to noise

The noise has a maximum value of 0.07 photoelectrons. 1 level on the ADC represents 5/256 = 0.0195 [V]. So 1 photoelectron is represented by 0.0195 [V]. This means that the noise of 0.07 photoelectrons is represented by 0.07x0.0195 = 0.0014 [V]. So quantization by the ADC is an advantage in this case because the errors get filtered out and will not have any influence on the calculations.

8.7 Implementation

Some information on the DSP- board , depending on the hardware configuration, will have to be set up via the PC.

•x_wires,y_wires, total_xy	:The number of x and y wires/pixels and their total.
•threshold	:Determines which minimum value a wire or pixel must have in order to be considered as part of an event.
•samplemap (list or matrix)	:A list of sequential references to wire or pixel positions within an event.
•reciprocal (table)	:A list of reciprocal values for divisions. This will be used because the DSP is unable to execute a division in 1 cycle.
•pcount_min, pcount_max	:To test whether there are enough pixels active to calculate an accurate position and not more than 1 object is present (double event).
•energycorrect (matrix)	:A table to correct energies.

A large part of the existing Datel software can be used without making many changes. A few functions will have to be added to perform centroid calculations and the scheduling of the buffer filling. Data transfer scheduling to the host-PC will also have to be changed.

Typical program operation is as follows:

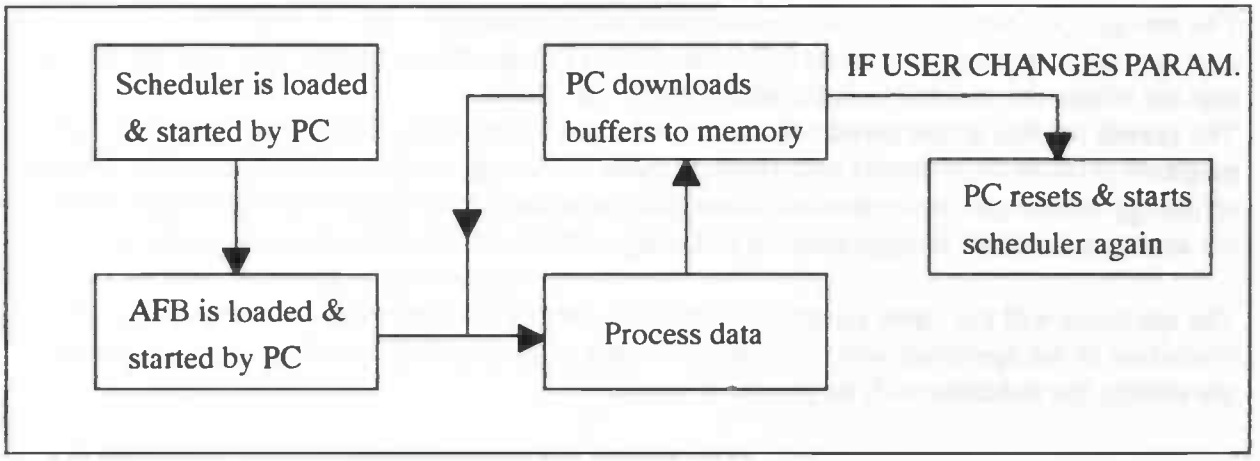

Figure 8.8: Typical program operation.

9. Algorithms and processing on the PC

The following features will be implemented:

•printing of x,y positions (average or total, when more than one event has occurred in the same position), as often as possible.

•printing an energy spectrum (a histogram of the various energies).

•possibility of choosing an energy window with which an image of the x,y positions will be generated. A number of options can be chosen by the user.

9.1 Displaying the x,y positions/levels

The x,y positions are displayed in a window. The colour of the pixels will indicate the energy value they represent. The colours should give an intuitive representation of the photon counts. So the highest count would be shown as red, then orange, yellow, green, blue, violet etc.
All calculated positions with an energy that falls within the selected energy window count as 1 hit on that position and will be used in the image. The image is built by counting the hits on every position. The position with the highest number of hits will have the brightest colour. The position with the least number of hits will have the darkest colour.

Before anything is displayed, data will have to be collected for a certain length of time. In this way enough data can be stored to form an image. The time needed to build an image depends on many factors and will have to be determined experimentally.

9.2 Displaying the energy spectrum

The energy spectrum is employed to determine which energies occur at which frequencies. The user can select an energy window (min, max) from this spectrum. In this way, only the energies that are within the window will be used to build the image.
The reason for this is that certain energies can be irrelevant to the image (see chapter 6.4). Some might even be noise. If these energies are left out, the image quality will improve. By choosing an energy window, experiments can be carried out to determine the best energy range. Each time the user selects a new energy window, the image will be rebuilt using the new energies.

The spectrum will not show all possible energies, simply because there are too many. Instead the resolution of the spectrum will be reduced in order to display it. Depending on the resolution of the screen, the reduction will be greater or smaller.

9.3 Resolution

When the x,y positions are displayed on the screen, it has to be ensured that a whole number of pixels is used to display a position. If this does not happen, the accuracy of the image will degrade. This is achieved by setting up a data display window, that consists of exactly the same number of pixels as the image to be displayed, or a multiple of that number. In this manner, one pixel will be displayed as one physical pixel or as a multiple of 1 (2x2, 3x3, 4x4 etc..). It has been determined in part I of this report, that the average error will always be around 10% of the pixels width (optimum), so it is not useful to display images with a resolution of more than 1/10 of a pixel on the detector. This will only involve more calculations and will not enhance the image quality.

9.4 Image processing

In order to enhance the printed image, a pixel weighting algorithm is used to make smoother pixel-transitions. Because the use of this algorithm can entail loss of detail, provision is made for an on/off function.

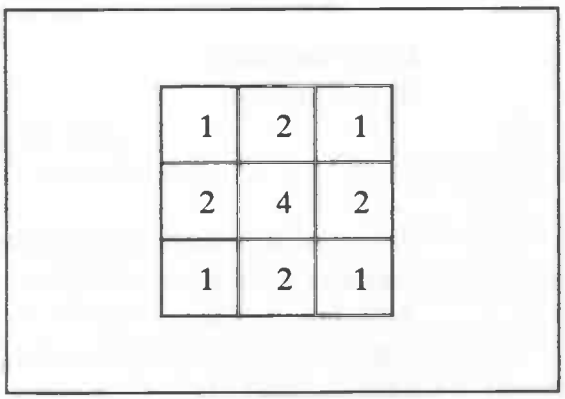

Figure 9.1: Pixel weights.

The middle of the grid is slid over all pixels that have to be displayed on the screen. The intensities calculated for each grid of the nine subpixels will then be averaged so that the center subpixel is weighted by a factor of 1/4; the top, bottom and side subpixels are each weighted by a factor of 1/8; the corner subpixels are each weighted by a factor of 1/16. In this way a new intensity for the middle pixel is calculated.

9.5 Implementation

The PC software will be written in Visual C++ (1.51, 16 bit version).

The software will include the following menus:

FILE

•Open:	Load a bitmap file.
•Save:	Save a bitmap file.
•Save as:	Save a bitmap file with a new filename.

DIAGNOSTICS

•Statistic	Bring up window with statistic about current configuration.
•Histogram	Bring up window with energy histogram to select energy window.

SAMPLING

•Start:	Begin sampling.
•Stop:	Stop sampling.

SETTINGS

•Energy window:	Adjustment of the energy window.
•Threshold:	Adjustment of the threshold.
•PCount:	Adjustment of max. and min. active pixels/event.
•Load:	Load a configuration file
•Save:	Save a configuration file.
•Save as:	Save a new configuration file.

DISPLAY

•Zoom:	Adjustment of the zoom factor.
•Update frequency:	Adjustment of the image refreshing speed.
•Smoothing:	Smoothing on/off.
•Clear:	Clear the image from the screen.

Certain information remains on–screen (statistics).

•Hardware configuration
•Threshold
•Pcountmin, Pcountmax
•Energywinmin, Energywinmax
•Update frequency
•Data loss
•Data/second
•Acquisition time
•FIFO overflow (boolean)
•Outputbuffer overflow

Main data structures:

•Reserved memory for downloading buffers.
Initially the raw data is downloaded from the DPR on the DSP–board to the host–PC. This will consist of:

- •Xposition
- •Yposition
- •Energy

•Structure to collect histogram data.
An energy histogram is built from the raw data. The positions are not used here.

•Matrix to hold x,y data for image processing.
In this matrix the counts on all positions are kept. Here the data is still unchanged.

•Memory bitmap to hold displayed image (after image processing).
Before being displaying from the matrix, the data is smoothed. The data is then transferred to the bitmap and can be displayed. This smoothing only occurs when the image is updated on the display. During normal data collecting periods, only the standard matrix is updated continuously.

Typical program operation

Once the package is started up, the following will happen:

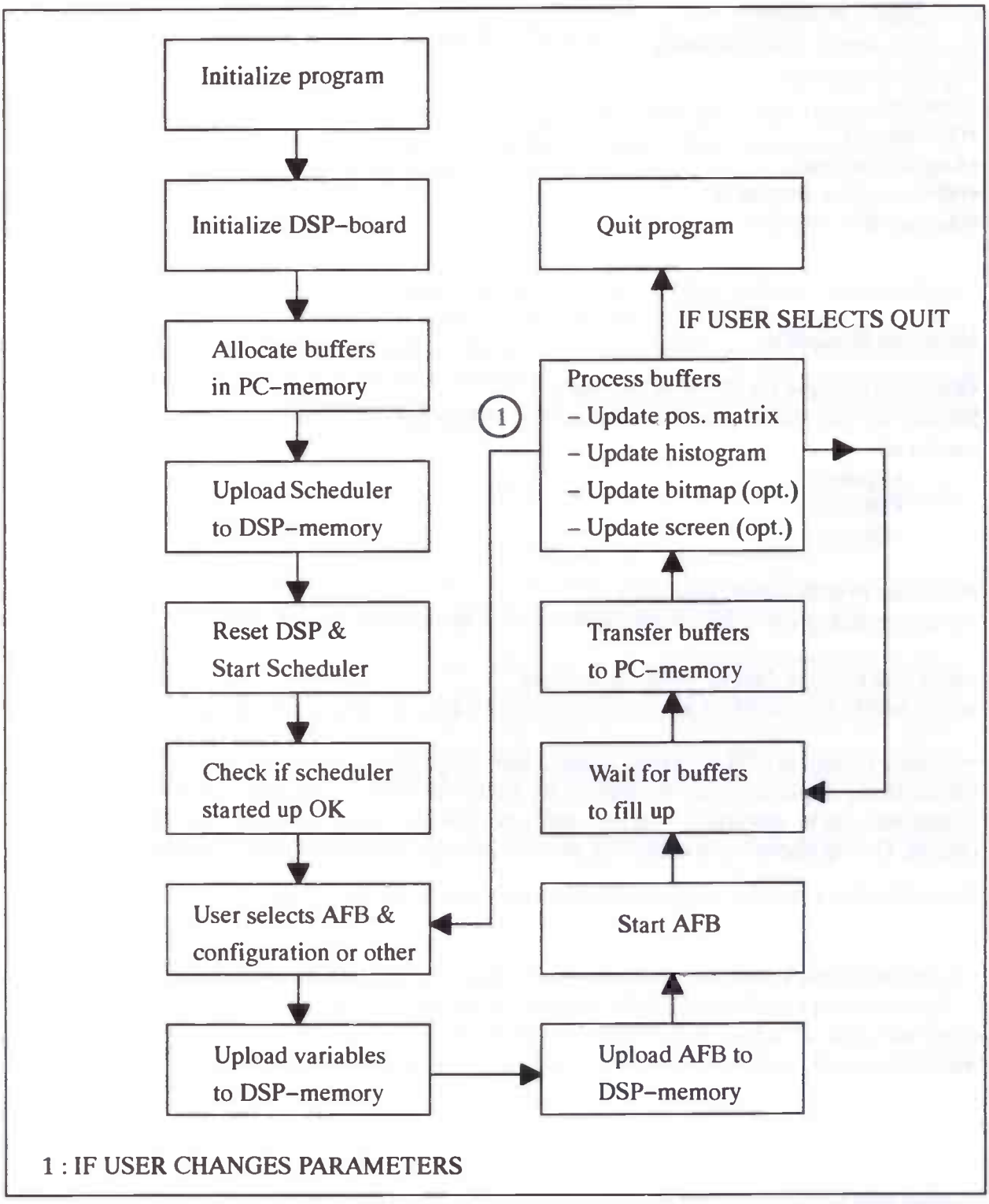

Figure 9.2: Typical program operation.

10. Conclusions & recommendations

A package has been developed to read out position sensitive detectors. Two search algorithms have been designed that find and scan a complete object on the detector grid:

•The linear search (method 1).
With this method every pixel on the grid is checked until all active pixels have been found. It has little overhead and is relatively fast. It uses a 1–dimensional pixel map (list) to access all available pixels. The algorithm can be used for every type of detector that has to be read out, without change.

•The systematical search (method 2).
With this method the grid is searched systematically. If an active pixel is found, the rest of the active pixels are found recursively. This method has more overhead and only becomes interesting with higher numbers of pixels on the grid. It uses a 2–dimensional pixel map (matrix) to access all available pixels. The standard algorithm does not work with every type of detector. So to ensure high speed processing, a slightly different algorithm is developed for every detector that has to be read out..

A speed comparison revealed that for objects with 4 and 9 active pixels the following choices should be made:

- •Object size 4 pixels, number of pixels on the grid < 40–50: use method 1.
- •Object size 9 pixels, number of pixels on the grid < 80–90: use method 1.
- •With higher numbers of pixels on the grid, method 2 should be used.

For peak calculation, the centre of gravity algorithm has been explored and simulated. Peak fitting is not interesting because of the long execution times. The results show that with events where 100–200 photoelectrons are deposited, object sizes of 4 or 9 pixels are preferred. With greater sizes the error in the calculation of the peak position grows too much. Using the sizes mentioned above, the error in the position in x– or y–direction will stay below 10% of the pixel width.
Using these object sizes also means that 65 – 80% of the detector grid can be used to determine peak positions.
To make sure that these object sizes are created, the scintillatorcrystals will have to be designed in the correct way.

Processing 20.000 events/second might be possible with detectors that have a small number of pixels, but the limit will be reached quickly on the existing DSP–board. Only a maximum of 1000 instructions per event can be used, so the more calculations that have to be done, the sooner the 1000 instructions are used up. This will also happen if the number of pixels on the detectors increase. The solutions are:

•Faster hardware.
•Turn down the event rate.
Both solution increase the number of instructions that can be used per event but the first option is the best because high event rates are preferred.

To write the software it is best to use the existing Datel DSP & Windows 3.x software.

The implementation of the software on both the DSP–board and the host–PC are discussed.

For future development a few points might be of interest:

•Find hardware that processes data even faster. This will of course eventually happen anyway. This includes:

- •Faster processors to do more calculations per second.
- •Special hardware implementations to find peak positions very fast.

•As processing rates increase because of new hardware development, it will become increasingly interesting to develop algorithms that can also calculate peaks of events that are partially on the detector surface. In this way 100% of the detector surface would be usable. Detecting peaks of double events is also worth investigating. Both can be achieved using peak fitting algorithms.

•Investigate possibilities of object detection and recognition in the images that are collected. Several existing techniques, including fuzzy logic or artificial neural networks could be employed for this purpose.

APPENDIX A

The following MATLAB files were used to do simulations:

•DO.M

USAGE: [mu, list, actpix, firstbin] = DO(pos, pixels, photo)

PURPOSE: Create a distribution of *photo* photoelectrons at position *pos*, on *pixels* pixels

•COFG.M

USAGE: [centre, errors] = COFG(mu, list, actpix, firstbin)

PURPOSE: Calculate peak with centre of gravity method

•SIMULATE.M

USAGE: [averagel, maxl, minl] = SIMULATE(pos, nrpix, nrsims, photo)

PURPOSE: Simulate *photo* photoelectrons at position *pos* on 1 to *nrpix* pixels for *nrsims* times and calculate the centroid using COFG each time.

•GO.M

USAGE: [averages, minima, maxima] = go()

PURPOSE: A list of simulations using SIMULATE

•AREA.M

USAGE: [list] = AREA(pixels, xp, yp)

PURPOSE: Calculate the usable area of the detector surface, with size *xp* x *yp* if the object on the grid has a size of *pixels* pixels

•AMOUNT.M

USAGE: [list] = AMOUNT(total)

PURPOSE: Calculate the number of pixels to search if the square grid has *total* pixels. The object size varies form 1 to *total* pixels

•RECUR.M

USAGE: [listrec,listnorm] = RECUR(a, total)

PURPOSE: Calculate the execution times for method 1 and 2 with different grid and object sizes (*total* and *a*)

```
function [m,opp,actpix,firstbin] = do(pos,pixels, photo)
%Simulates a distribution of a number of photoelectrons on a number of pixels at a position.
%[mu,opp,actpix,firstbin] = do(pos,pixels, photo).
%pos = avg. position of real mu.
%pixels = number of pixels to cover.
%photo = number of photoelectrons to create.
%opp = list of pixels with nr of counts.
%m = mu generated.
%actpix = number of active pixels.
%firstbin = first bin that contains data.

%assumption: all data contained in 3*sigma.
%So 6*sigma = 100% of curve.
maxbin=100;
sigma = pixels/6;
%create random mu between pos-0.5 and pos+0.5.
m = rand(1)+pos-0.5;
%create list of photoelectrons
list = normrnd(m,sigma,1,photo);
%Calculate nr of pixels that are hit
pix = round(max(list))-round(min(list))+1;
%now build histogram of photoelectrons on different pixels(bins)
bin=zeros(1,maxbin);
for count = 1:photo
        bin(round(list(count)))=bin(round(list(count)))+1;
end
opp =bin;
pixact=pix;
%now all bins are filled (from 1 to pixels)+offset
%find first filled bin (can also be calculated)
count=1;
while bin(count)==0,
        count=count+1;
end
firstbin=count;
actpix=pix;
```

```
function [centre, error] = cofg(mu,list,actpix,firstbin)
%Calculates centroid with centre of gravity algorithm.
% [centre, error] = cofg(mu,list,actpix,firstbin).
% mu = real mu.
% list = bins representing pixels with photoelectrons in them.
% actpix = number of pixels hit (is calculated by DO.M).
% firstbin = first pixel that has photoelectrons on it (calc. by DO.M).
% centre = centre of gravity.
% error = error % (of pixel width) of calculated mu.
total=0;
energy = 0;

for count = firstbin:actpix+firstbin-1,
        total= total+list(count)*count;
        energy= energy +list(count);
end

centre= total/energy;
error = abs(centre-mu);
```

```
function [list,maxl,minl] = simulate(pos,nrpix, nrsims, photo)
%Simulates distributions of a number of photoelectrons on different numbers of pixels at a
position.
%[list,maxl,minl] = simulate(pos,nrpix,nrsims,photo).
%pos = avg. position of real mu.
%nrpix = simulate 1 to nrpix pixels.
%nrsims = simulate each possibility (1,2,..,nrpix), nrsims times.
%photo = nr of photoelectrons that hit the detector.
%list = list of average errors.
%maxl = list of maximum errors.
%minl = list of minimum errors.
avg=0;
avglist = zeros(1,nrpix);
minlist = zeros(1,nrpix);
maxlist = zeros(1,nrpix);
minlist=minlist+10;
for p = 1:nrpix,
sims=0;
avg=0;
templist = zeros(1,nrsims);
        while sims<nrsims,
                [mu,opp,actpix,firstbin] = do(pos,p,photo);
                if (actpix >= p) & (opp(round(mu))>0),
                        [centre,error] = cofg(mu,opp,actpix,firstbin);
                        sims=sims+1;
                        templist(sims)=error;
                        if error>0.5,
                        end
                end
        end
        for count=1:nrsims,

                nr = templist(count);
                if nr<minlist(p),
                        minlist(p)=nr;
                end
                if nr>maxlist(p),
                        maxlist(p)=nr;
                end
                avg=avg+nr;
        end
        avglist(p)=avg/nrsims
end
list=avglist;
maxl=maxlist;
minl=minlist;
```

```
function [averages,minima,maxima] = go()
%Does a number of simulations, using the SIMULATE function.
%[averages,minima,maxima] = go().
%averages = list of average error lists.
%minima = list of minimum error lists.
%maxima = list of maximum error lists.

%Do simulations

averages= zeros(10,12);
minima= zeros(10,12);
maxima= zeros(10,12);

%Every simulation = 1000 times

%pos=25, photons=100
[list,maxl,minl]=simulate(25,10,1000,100);
averages(:,1)=list';
minima(:,1)=minl';
maxima(:,1)=maxl';

%pos=25, photons=200
[list,maxl,minl]=simulate(25,10,1000,200);
averages(:,2)=list';
minima(:,2)=minl';
maxima(:,2)=maxl';

%pos=25, photons=1000
[list,maxl,minl]=simulate(25,10,1000,1000);
averages(:,3)=list';
minima(:,3)=minl';
maxima(:,3)=maxl';

%pos=50, photons=100
[list,maxl,minl]=simulate(50,10,1000,100);
averages(:,4)=list';
minima(:,4)=minl';
maxima(:,4)=maxl';

%pos=50, photons=200
[list,maxl,minl]=simulate(50,10,1000,200);
averages(:,5)=list';
minima(:,5)=minl';
maxima(:,5)=maxl';
```

```
%pos=50, photons=1000
[list,maxl,minl]=simulate(25,10,1000,1000);
averages(:,6)=list';
minima(:,6)=minl';
maxima(:,6)=maxl';

%pos=75, photons=100
[list,maxl,minl]=simulate(75,10,1000,100);
averages(:,7)=list';
minima(:,7)=minl';
maxima(:,7)=maxl';

%pos=75, photons=200
[list,maxl,minl]=simulate(75,10,1000,200);
averages(:,8)=list';
minima(:,8)=minl';
maxima(:,8)=maxl';

%pos=75, photons=1000
[list,maxl,minl]=simulate(75,10,1000,1000);
averages(:,9)=list';
minima(:,9)=minl';
maxima(:,9)=maxl';


function [xlist] = area(pixels,xp,yp)
%Calculates usable area versus the number of pixels used for lightspot
%assuming that 100% of lightspot is on the detector surface.
%[xlist] = area(pixels,xp,yp)
%pixels = from 1 to pixels (in x  and y) are tested
%xp = number of xpixels
%yp = number of ypixels
%xlist = % of usable detector surface versus nr of pixels


list = zeros(1,pixels);
for count=1:pixels,
        xpixe = sqrt(count);
        ypixe = sqrt(count);

        xstraal=xpixe/2;
        ystraal=ypixe/2;

        eff=(xp-xstraal*2+1)*(yp-ystraal*2+1);
        list(count)=(eff/(xp*yp))*100;
end
xlist=list;
```

```
function [list]=amount(total)
%Shows relation between total nr of pixels and nr pixels that
%have to be checked with the systematic algorithm.
%[list]=amount(total).
% total = number of pixels on the grid ( N_x × N_y).
% list = list of nr of pixels that have to be checked.

for a= 1:sqrt(total)
        list(a)= round(sqrt(total)/sqrt(a))*round(sqrt(total)/sqrt(a));
end


function [listrec,listnorm] = recur(a,ntotal)
%Calculates the execution times of both methods with object size a and different grid sizes.
lin=zeros(1,ntotal);
lir=zeros(1,ntotal);
for tel=a:ntotal
%method 1
        lin(tel) = tel*4 +a;
%method 2
        lir(tel) = round(sqrt(tel)/sqrt(a))*3 + round(sqrt(tel)/sqrt(a))*round(sqrt(tel)/sqrt(a))*3 +
round(sqrt(tel)/sqrt(a))*round(sqrt(tel)/sqrt(a)) +4*a*(4+3)+ 4*a;
end

listrec=lir;
listnorm=lin;
```

GLOSSARY OF ABBREVIATIONS

ADC	Analog Digital Convertor.
AFB	Application Function Block, a script file with simple instructions to be carried out on the DSP card.
DPR	Dual Ported Ram.
DSP	Digital Signal Processor.
event	The occurrence of a lightspot on the detector surface.
FIFO	First In First Out.
HPD	Hybrid Photomultiplier Detector.
ISR	Interrupt Service Routine.
LSB	Least Significant Bit.
MIPS	Million Instructions Per Second.
photocathode	Electrons are freed form this material when optical photons enter it.
photoelectron	An electron emitted from the photocathode layer as a result of an incoming gamma–photon.
PSPMT	Position Sensitive Photo Multiplier Tube.
RAM	Read Only Memory.
scattering	process where gamma–photons divert from their original route through interaction with other objects, and lose energy.

BIBLIOGRAPHY

1. Mark Andrews, Learn Visual C++ Now, Microsoft press, 1996.
2. H.G. Dehling, Dictaat inleiding statistiek, Vakgroep wiskunde, 1992.
3. Gordon Gilmore & John Hemingway, Practical gamma–ray spectrometry, John Wiley & sons, 1995.
4. David Harel, Algorithmics, Addison–Wesley, 1992
5. David J. Kruglinski, Inside Visual C++, Microsoft press, 1994.
6. Howard Mark & Jerry Workman, Statistics in spectroscopy, Academic press, 1991.
7. Charles Petzold, Programming Windows 95, Microsoft press, 1996.
8. W.K. Pratt, Digital image processing, John WIley & Sons, 1991.
9. DATEL PC–430 User Manual, DATEL, 1995.
10. TI Floating–point DSP assembly language tools, TI, 1995
11. TI Floating–point DSP optimizing DSP compiler, TI, 1995