Infrared jitter imaging data reduction algorithms

Nicolas Devillard - European Southern Observatory

ABSTRACT — **Jitter imaging (also known as *microscanning*) is probably one of the most efficient ways to perform astronomical observations in infrared. It requires though very efficient filtering and processing methods to produce the best possible output from raw data. This paper discusses the various issues attached to each data processing step and offers several algorithms, having in mind an automatic data processing in pipeline mode (i.e. without user interaction) as intended for the VLT.**

1 Introduction

Jitter imaging (also known as *microscanning*) is an efficient method used in astronomical infra-red observations to take care of sky background issues with a minimum loss of observing time. By observing the same spot on the sky with small offsets around a central position for each exposure, it is hopefully possible to deduce the sky background variations directly by filtering the images, and separate astronomical from sky signal.

A typical acquisition in jitter mode consists in a set of 10 to 100 frames. The first frame is assumed centered on the point of interest, the following frames are slightly offseted from the first position with offsets not bigger than a reasonably small proportion of the detector size (15% for SOFI/ISAAC). The data reduction process is doing sky estimation and subtraction, plane recentering, and finally frame co-addition to a single frame. Each processing step is studied here in the context of automatic data reduction in pipeline mode (i.e. without user interaction).

2 Sky background estimation

In the following discussion, it is assumed that the input is a data cube or set of frames containing a reasonable number of successive planes taken on the same position of the sky, with relatively small offsets. It is also reasonable to assume that flat-field and dark corrections have been applied to this cube in case they have a significant impact on the signal quality. The aim is to remove the sky contribution from the input set of frames by means of filtering. As usual in such cases, some assumptions must be made on typical sky and astronomical signal in order to be able to define criteria for signal/sky discrimination.

The main assumption is that most of the observed signal is indeed sky background. This is the case when observing point-like sources but is not suitable for extended objects. In the latter case, a better choice would be to combine jittered observations with sky background acquisitions taken with a wider offset outside of the observation field, the immediate drawback being a loss of observing time. This method is referred to as *jitter+offset*, and can be considered as a subset of jitter mode with few additional needed reduction steps.

A derived assumption is that for a vast majority of pixels, what is seen along a time line is mostly sky background signal. This can be achieved if one assumes an approximately homogeneous distribution of objects on the sky, and an homogeneous repartition of the offsets used for jittering. The following paragraph describes a method to generate optimal random offsets for sky background filtering in jitter mode.

2.1 Poisson offset generation

Points generated with a Poisson distribution law share a property (let us name it *Poisson property* here), which specifies that no couple of points is closer to each other than a minimal distance. This distance is computed by taking into account the size of the (rectangular here) domain we want to generate points in, and the number of points to generate.

The minimal distance between any two points for a set of n points generated in a rectangle of size $S_x \times S_y$ is chosen as:

$$d(S_x, S_y, n) = \sqrt{\frac{S_x S_y}{n\sqrt{2}}} \qquad (1)$$

Points are randomly generated, only the ones satisfying the Poisson property are stored, until the requested number of points has been reached. Figure 1 shows an example of 100 generated points scattered according to a Poisson distribution law.

In the case of jitter offsets, we are interested in generating points which share the Poisson property for any h consecutive points (should ideally by equal to the half size of the running time filter described in the sky filters

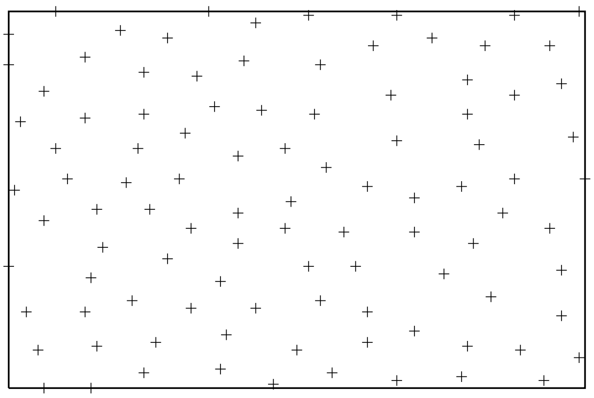

Figure 1: Poisson distribution law

below) out of a group of n (total number of points to generate). The problem is now described as:

Generate n points, for which any h ($h < n$) consecutive points share the Poisson property. The algorithm we use can be described as:

- Generate $h - 1$ points sharing a Poisson property for h points on the requested domain (minimum distance is $d(S_x, S_y, h)$).
- Iterate the following step until n points have been generated:
- Generate a point sharing the Poisson property with the previous $h - 1$ points (minimum distance is $d(S_x, S_y, h)$. Append this point to the list of generated points. Iterate for the next point with the $h - 1$ previous points.

This way, we can be sure that h consecutive points will be homogeneously scattered over the requested domain, and that the sky estimation will be done with optimized chances of finding mostly sky variations.

A Poisson point generator simply called `poisson` dedicated to this task has been written using only standard ANSI C features, allowing compilation on any station without need to download any specific library. It has been included in SOFI/ISAAC templates, when random offset generation is requested to be done by the instrument (*AutoJitter* template).

2.2 Sky combination

Sky combination is a method used to filter out low-frequency sky variations from a set of jittered images. The algorithm is based on a non-linear filtering of pixels along time. One estimated sky frame is estimated per input raw frame.

The method is iterating on each pixel position on the detector along a time line. The algorithm can be described for each pixel position as:

- Loop on all planes. For each plane, do the following:
- Loop on all pixel positions on the detector. For each position, do the following:
- Extract an array of $\pm h$ pixel values in time around the current pixel (this brings up to $2h + 1$ values).
- Scale (divide) each pixel in the array by the (filtered) average value of the plane it was extracted from.
- Sort this scaled array by increasing pixel value.
- Reject the *rmin* and *rmax* pixel values in the sorted array.
- Average the remaining pixel values and assign the result to the same pixel position in the same current plane in the output cube.

Notice that if $rmin = rmax = h$ this method is equivalent to a running median filter: only the central sorted value is kept. Do not use such settings for *rmin* and *rmax* though, since it would leave edge planes with undefined values.

Edge effects are not negligible: the first and last h planes do not use $2h + 1$ raw planes for sky estimation, but between $h + 1$ and $2h + 1$. This lower amount of information is propagated to a less precise sky estimation and thus more noise in the subtracted frames.

User-provided parameters are:

- h the halfwidth of the running time filter. Setting too small a value makes the filter useless. On the opposite, too high a value would prevent taking into account the temporal sky background variations and thus skew the results. Setting this value automatically would require to take into account the exposure time and sky temporal variation statistics in the current wavelength. It is not yet implemented. The default value is 7, which means that in the best cases 15 time values (minimum 8) are used for each pixel to estimate sky variations.
- *rmin* and *rmax*, the number of rejected upper and lower pixels. Their sum shall not exceed h. Rejecting low-valued pixels compensates for very noisy values. Rejecting high-valued pixels is the mechanism which extracts the astronomical signal. The default for both values is currently set to 3.

This method is sensitive to an initial assumption: namely, when a pixel position sees mostly an object signal and not a sky background variation, it becomes difficult (if possible) to discriminate sky from object signal. This is the case for extended and bright objects: when jittered around, the offsets are never big enough to prevent skewed estimations. The resulting artefacts are usually dark crowns surrounding bright objects in the final image. They are directly affecting photometry and signal-to-noise measurements.

3 Frame offset detection

The next step is to measure or acquire the offsets between the frames. It is reasonable to assume that the acquisition orders for telescope pointing are available as ancillary data. If they are not, there should be an automatic way of finding these offsets with a minimal chance of error. Last, users may want to use their own method to detect offsets and feed their input into the algorithm for ultimate refining. We discuss here a cross-correlation method used both for offset detection and refining.

3.1 Detecting features for cross-correlation

The quality of the cross-correlation criterion depends on the amount of interesting signal present in the measurement area. Intuitively, if we try to cross-correlate two flat (blank) images, the returned measurement will have no meaning since any pixel could be a valid candidate.

The peak detector described below is used to localize bright features. Because the first frame in the stack has had less frames to estimate its sky background, it is usually more affected by noise and thus not so reliable to use for bright object detection. A good indicator of bright object position can be computed cheaply by taking the difference between the first 2 raw frames in input. This way, we have a frame with excellent sky and dark subtraction, but also negative objects. If we specify to the peak detector that it should not try to detect negative peaks, the same results are found as by using the first sky-subtracted frame, but since it is less noisy it is possible to find fainter objects, thus more correlating features.

Peak detection algorithm From an input image, estimate a mean m and σ value. Since it is likely that there will be bright signal in the input image, we use for estimated mean and sigma the actual *median* and *average absolute deviation from the median.* This way, we are less sensitive to pertubations in estimating these parameters.

A binary image of the same size is created by assigning one to all pixels which value lie above $m + K\sigma$ in the input image, and zero else. The definition of K depends on the amount of noise and the number of objects present in the image. For very noisy images, it is best to lower it to a value between 0.1 and 1 (default is 2).

This thresholding creates a crude map of bright regions in the input image. To clean bad pixels and other such small defects, a binary morphological closing is then performed. To learn more about binary morphological operations in image processing, the reader is referred to any complete image processing book, such as **[Russ 1995]**. Connected components in this binary map are then identified and assigned a unique ID, then sorted out by distance to the center. The algorithm used to classify features is non-recursive and ensures that computations are done in a finite time with a reasonable amount of memory used.

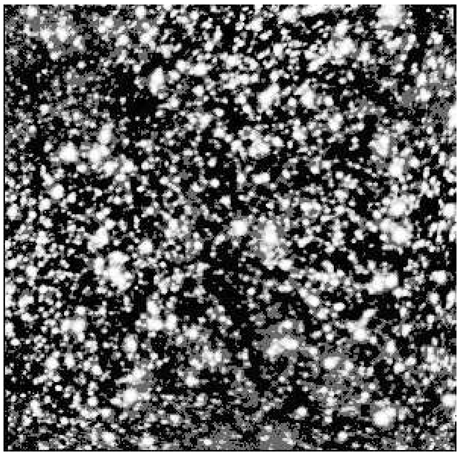

Figure 2: Input image for the peak detection algorithm

Figure 3: Peak filtering with $K = 1$

Figure 2 shows a typical input image for feature detection. Figure 3 shows what the algorithm sees for $K = 1$. This binary map is the result of binary thresholding and morphological closing.

Because the structural element for the binary morphological operation is a 3×3 kernel, no bright object strictly smaller than this can be detected, this to avoid detecting bad pixels as bright sources. It might be the case (e.g. for focus images) that such small sources should be detected. A smearing filter (flat 5×5 kernel) is then applied to the input image previous to the peak detection. Small sources and bad pixels are then detected.

This algorithm is offered separately as an **eclipse** command called `findobjs`. It can be used to autodetect sources in a FITS image, to compute Full Width at Half Maximum or estimate photometry. This command supersedes the previous `detpeak`, `photfits` and `fwhm` commands in the **eclipse** package.

Peak detection strategy Detecting enough peaks in any image in an automatic way is not easy to generalize. A strategy has been established to try to gather as many features as possible. Input parameters are: min_p and max_p, resp. minimum and maximum number of points to be detected, K the threshold factor as described above in the peak detector. In the following items, as soon as peak detection has returned a correct number of points we exit and return the found points. If not enough points have been found, we proceed to next step. If more points than requested have been found, we declare the search successful and return the max_p brightest points found in the image.

- Apply the peak detector with an initial value for K.
- Apply a low-pass filter to the image (e.g. a flat 5x5 kernel), apply the peak detector with the initial value for K.
- On the low-pass filtered image obtained at the previous step, try peak detection using $K/2$.
- On the low-pass filtered image, apply detection with $K/5$.
- If we reach this point, it means no list of points can be found in the input image. If $max_p > 1$ return a failure code, else go ahead.
- If the image maximum pixel is usable (far enough from the edges to enable cross-correlation), return it.
- At this point, no point was found and $max_p = 1$, issue a warning about the poor image quality and return the image center.

Normally, in a decent astronomical image there should always be sources to detect for this algorithm. The 3 parameters to set for object detection are specified in a specific section of the initialization file for the jitter program.

```
[DetPeak]
Threshold           = 1.0 ;
MinDetectPoints     = 1 ;
MaxDetectPoints     = 10 ;
```

Users can also provide their own lists of features through a configuration ASCII file, if they do not want to rely on this automatic feature extraction. Indeed, for images containing only faint sources (e.g. narrow-band imaging), the above strategy may fail to find anything but noise in the images, thus corrupt the offset measurements.

3.2 Cross-correlation

The DataFlow environment on the VLT ensures that data such as sky position pointed to by the telescope should be present in every FITS header. However, no telescope pointing can be precise up to subpixel precision and expected errors are in the range of a few pixels. For this reason, telescope offsets are taken as a first estimate of the place to look for, but not as an absolute reference. This estimate is fed into the offset precise detection algorithms described hereafter. Users can also provide their own offset estimations if they have other tools to compute them. Last, if no first estimate is provided, a default null vector will be used, with a larger correlation search domain.

Cross-correlation is a very efficient tool to match images. It is quite robust to noise, and can be normalized to allow pattern matching independently of scale and offset in the images. It has serious drawbacks however, in the case images of faint sources. We discuss here the implementation of a cross-correlation tool, the parameters which are used to configure it, and try to define which are the best suited parameters for a given search.

3.2.1 Definitions

The cross-correlation of two images I_1 and I_2 is defined as the product:

$$\sum_{p_1 \in w_1} \sum_{p_2 \in w_2} p_1 \otimes p_2 \qquad (2)$$

where p_1 is the pixel index running over the domain of interest w_1 in the image I_1, and similarly p_2 a running 2-dimensional index over the domain of interest w_2 in the image I_2. The cross-correlation product denoted by $\otimes$ can be defined by several possible functions. Common product definitions are:

Sum of squared differences:

$$p_1 \otimes p_2 = \sum_{w_1, w_2} (p_1 - p_2)^2 \qquad (3)$$

Sum of products:

$$p_1 \otimes p_2 = \sum_{w_1, w_2} (p_1 p_2) \qquad (4)$$

The first criterion describes a measure of difference between the two interest regions of the images. For a pattern-matching application, this criterion is the quantity to be minimized to find the best matching point. On the opposite, the second criterion describes a measure of resemblance between the two interest regions, it is a quantity which has to be maximized in order to find the best correlating point.

Other cross-correlation criteria differ from these two by the fact that they can include scale and offset normalisation in luminance. This is the case when we have:

$$I_1(i,j) = \alpha I_2(i + dx, j + dy) + \beta \qquad (5)$$

However in the jitter case, the previous processings have normally taken care of rescaling the data to set

$\alpha = 1$ and $\beta = 0$ (all planes share the same photometry).

In this implementation for the jitter command, it has been chosen to look for a minimum of the measure of difference.

Also, no rotation or scale difference is assumed, although this could easily be included in the linear fit. The reason behind is that trying to match a rotation or scale would probably give non-null results, whereas we know that none of these can occur. This would bias the quality of the offset estimation.

The reference image is the first image in the frame list, it is considered to be the best centered image on the object. This puts a requirement on the acquisition procedure definition: when acquiring data in jitter mode, the first frame shall be the best centered position on the field of interest. This is the case for SOFI/ISAAC.

The domains of interest to use for cross-correlation shall be parts of the image containing enough signal to produce clear local minima. The choice of the best points of interest to be used for cross-correlation is described in a further section. We select as many points as possible in the first image (or first difference image, as explained below in the section about points of interest) and in all consecutive planes use all possible points for matches. This way, any point visible both in the first frame and the current frame will be used to estimate the shift, points that disappear due to the jitter are simply not taken into account. For each plane is established a list of offsets, each of them having been estimated from one correlation point. A median offset is computed for each direction, and the selected offset is the closest to this median offset, the distance criterion being a sum of absolute values of the differences between offsets.

If $(m_x, m_y) = (median(x_i), median(y_i))$, and $d_i = |x_i - m_x| + |y_i - m_y|$, the selected offset has the minimal d_i distance to the median offset.

3.2.2 Example

Figure 4 shows an example of a cross-correlation function for a sum of squared differences. The reference image is a bright galaxy, and the compared image is a shifted version of the reference with a translation vector of $(+5, -5)$. This plot clearly shows that the cross-correlation criterion is minimal at the expected point. A search for a local minimum will yield the right translation vector.

At this point, a subpixel offset detection is applied by fitting a parabola to the cross-correlation signal in x and y, and looking for the subpixel minimum. This method is theoretically precise up to a 1/100 pixel precision in the ideal case of noiseless images. The precision obtained with SOFI data is about 1/10 pixel.

3.2.3 Parameter setup

Parameters to setup for the cross-correlation algorithm are in all cases:

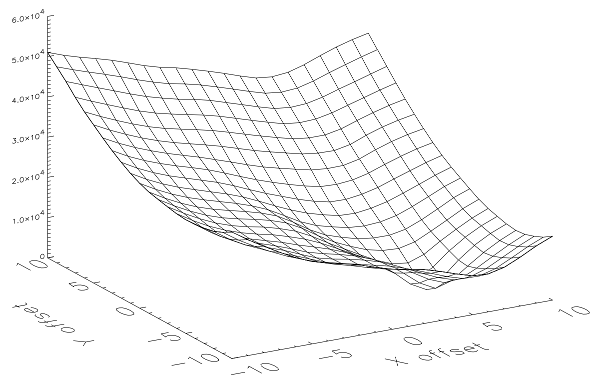

Figure 4: Cross-correlation matrix example

- The size of the search area in x and y. Given a starting point, how far will be searched in pixels around the point to find for a matching point? Setting this value depends on the expected level of accuracy for the provided positions.
- The size of the measurement area in x and y. Given a candidate position for a match, how many pixels should be used to compute the cross-correlation criterion? Figure 5 shows the importance of choosing a right measurement area size.

As specified, parameter settings depend on the quality of the input estimates for the offsets. The implementation gives the following possibilities for input estimates: header offsets, file offsets, or no offsets (automatic search). Each case is described hereafter.

One word about edge effects: computing a cross-correlation criterion for a point lying on an image edge would mean a loss in precision and probably artefacts. For this reason, it has been decided not to compute any cross-correlation if the involved point lies too close to an edge in one of the two input images. This sets a constraint on the choice of correlating point only.

Header offsets Offsets can be acquired from the FITS header for each file (this prevents the use of a single FITS file containing all jitter data, for simple implementation purposes). The user needs to specify what are the keywords to look for in each frame header to get the offset information in x and y, in pixels.

NTT offsets as found in SOFI data have been shown to be reliable to subpixel accuracy in the best cases, usually in the range of about 5 pixels in error, and never more than 10. Differences are mainly due to the optics and wavelength range. The default search area in this case is set to ± 10 pixels in x and y, which should cover more than really needed. The default measurement area is set to ± 25 pixels, which amounts the total amount of pixel operations to:

$(2 \times 10 + 1)^2 \times (2 \times 25 + 1)^2 = 1,147,041$

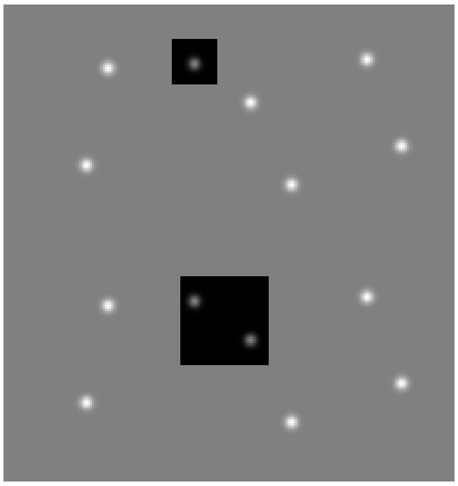

Figure 5: Importance of a right definition for the measurement area. The measurement areas are shown as overlapping dark boxes over the image. Above: it is too small, as it would induce a periodical pattern effect (any other star is a good candidate). Below: it looks large enough to prevent false detections.

File offsets Offsets can be acquired from a user-specified text file. The default setup for search and measurement areas are as for header offsets. This needs to be adapted to the quality of the provided offset measurements.

More information about input offset file format can be obtained by launching the `jitter` command with the `--offset` option.

Automatic offsets In this case, the algorithm should try to determine automatically the offsets between frames. It will look around the same position in all consecutive planes. The default search area is then increased to ±80 pixels, relying on the assumption that the offsets lie within 15% of the detector size. For SOFI/ISAAC, this means ±75 pixels. The measurement area is kept at ±25 pixels.

The total number of pixel operations in this case amounts to:

$(2 \times 80 + 1)^2 \times (2 \times 25 + 1)^2 = 67,420,521$

which is roughly a factor 60 in computation time compared to header offsets. It is easy to understand why it is preferrable to have reliable offsets in input (10 seconds computation time) rather than use an automatic search (10 minutes computation time). The automatic mode is also more prone to false detections, because the search area is bigger, thus the probability of mistaking positions increases.

3.2.4 Working domain

The cross-correlation tool is extremely robust to noise and very precise in the offset estimation. It is however sensitive to periodic patterns and unable to make the difference between one star and another similar point-like object for example. This can be avoided by enlarging the measurement area to a wider surface.

A typical issue is: when and how to determine that the detected local extremum is the right one? When matching two very different images, the procedure will always have an answer for the question: what is the best correlating place? There is clearly a need to define a threshold to validate a detected match or not. This strongly depends on the input signal and the expected amount of noise in the image. To get an idea, this implementation always prints out the value of the cross-correlation function at the local extremum, i.e. the minimum of sum of squared differences between both regions of interest. The smaller this value, the better the match quality (null for an ideal, integer shift without noise). As a first estimate, the quality of the matched point could be evaluated by saying (σ being this measure of similarity):

$\sigma < 10^2$	excellent
$\sigma < 10^3$	good
$\sigma < 10^4$	poor
$\sigma > 10^4$	false detection?

This rule of thumb of course varies wildly with the signal present in the input frames. The more signal there is in input, the more these thresholds are likely to get higher, without indicating a bad matching case. A possible enhancement of the algorithm would be to set this value automatically from a quick statistical analysis of the input signal for example.

3.3 Point pattern matching

Another method to register images efficiently makes use of point-pattern matching. The peak detector is applied to all input planes, and the search for offsets is then performed by using only on geometrical point positions. Among many algorithms comparing lists of points to find similar geometries, the fastest and most reliable one was identified as the one described by F. Murtagh in **[Murtagh 92]**. See references for other point-pattern matching algorithms.

The algorithmic problem can be described as: two lists of point coordinates are related by the fact that in each list it is possible to extract a subset containing matching points, i.e. points which are bound pairwise by the following transformation:

$$\begin{cases} x' = x + dx + N_x(x,y) \\ y' = y + dy + N_y(x,y) \end{cases}$$

where (dx, dy) is a fixed vector for all point pairs, and $N_x(x,y)$, $N_y(x,y)$ are two noise functions depending on the point position. (x', y') is the matching point associated to (x, y). The amount of added noise in the position by N_x and N_y should remain small, typically less than 10 pixels over a 1024^2 image.

Actually, transformations supported by Murtagh's algorithm include scale changes and rotations, but these are irrelevant to the jitter case and thus have been suppressed from the algorithmic search to limit possibilities of errors. The main issue is, given the two lists, to identify candidate transformations and associated lists of matching points. The point-pattern matching algorithm does not actually try to compute the transformation, it rather tries to identify point patterns with criteria which are independant from scale, offset, and rotation changes.

Differences brought to this algorithm for our implementation are the following:

- The definition of the *world-view* of a point has been extended to yield more precision (a factor 10) in the matching evaluation.
- Only geometrical point positions are used for pattern matching. It is reasonable to think that by adding some more informations about points to match such as star colour or relative brightness, one could improve the detection by excluding impossible matches. Provision has been taken to easily include that kind of information in the code.

Some more work has been done in the direction of removing impossible matching pairs, e.g. a point which is found is several pairs. The algorithm is run several times with rejection criteria, only the most stable points are retained. Matching efficiency is greatly increased this way.

Once the points have been matched they are sorted out by pairs. A least-square algorithm is applied to estimate the best translation vector between both sets.

Example: figure 6 shows an example of point-pattern matching over 50 points. The first list has been randomly generated, the second one has been shifted from the first by an offset vector of $(12.34, 34.56)$, and then a gaussian noise of maximum 10 pixels amplitude has been added to the point positions in the second list. Points have then been randomly removed from each list, leaving 47 points in the first list and 43 in the second one. The matcher indicates 41 matching point pairs, and an estimated offset of $(12.30, 34.34)$.

This method has been proved extremely precise because there are always many points of interest in SOFI/ISAAC images, which allows building a least-square estimation from a high number of point pairs. However, the reliability of the pattern matching is only about 80% at this stage, which excludes the possibility to use it without user interaction (cross-correlation reaches 100%. Some more work has to be done to fine-tune it automatically and bring its reliability to a higher value. This technique might also be used together with cross-correlation as a first estimate for the place to look for matches. For the moment, none of this algorithm is actually present in the code implementing the jitter reduction process.

4 Image recombination

All planes now need to be properly realigned with the first frame, knowing the offsets up to subpixel precision. The resampling operation (also known as *Image Warping*) is a classical problem in image processing. The subsequent image stacking operation is again some kind of temporal filter.

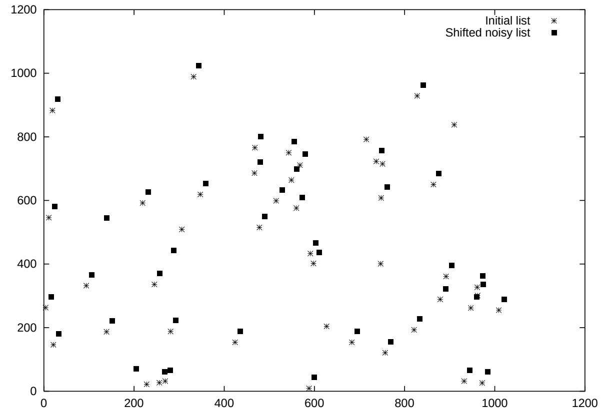

Figure 6: Point Pattern Matching: example

4.1 Image Resampling

To shift an image by a subpixel offset, it is necessary to use resampling methods. Provided some assumptions about the input signal cut-off frequency, an image is considered as the discretization of a continuous band-limited signal. According to Shannon's theorem, it is possible to retrieve completely this continuous signal from the knowledge of enough samples. It is then easy to *resample* the image, i.e. construct a new image with samples taken a new position of interest, namely the required offset. Interested readers are referred to **[Wolberg 90]**.

Designing an interpolation kernel can either be done in image or Fourier space. There is no *best* kernel for all purposes, but rather a zoo of possible filters, the output quality depending on the frequency contents of the input signal.

In the present implementation, we chose to resample images in the image space. The interpolation kernel is based on the 16 closest pixel values, there are 6 offered interpolation kernels described here.

sinc and square sinc Sampling a continuous signal is not harmless for the signal's spectrum: it multiplies it by a comb function, making it periodic. If Shannon's theorem is not respected, it is even worse since the signal's central spectrum is replicated over itself, creating aliasing artifacts.

If we assume a correctly sampled signal, then the ideal filter in Fourier space is a *Rect* function, which produces a sinc interpolation kernel in spatial domain.

$$\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

This kernel is far from being ideal in spatial domain, since it has infinite ringing converging towards 0 as $1/x$,

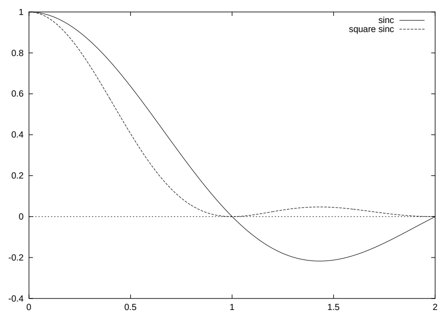

Figure 7: sinc and square sinc kernels

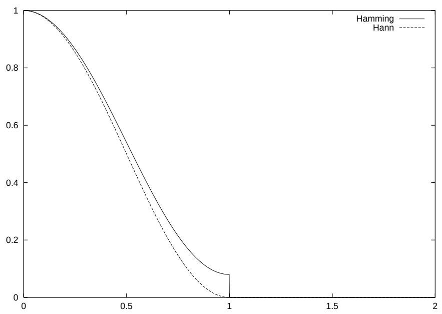

Figure 8: Hann and Hamming kernels

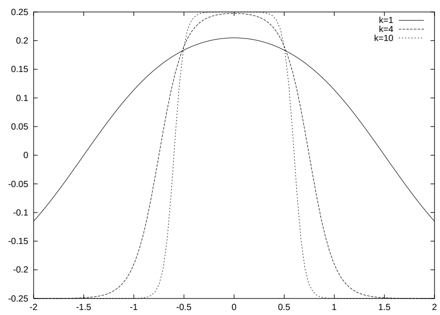

Figure 9: Scaled hyperbolic tangent function

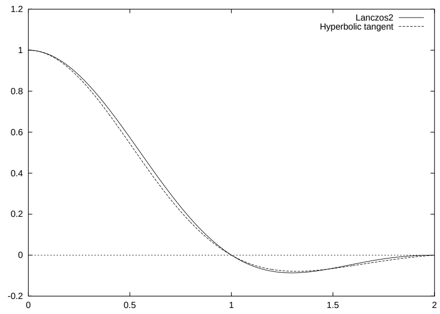

Figure 10: Lanczos and hyperbolic tangent kernels

which is too slow for small kernels such as a 4×4 pixel neighbourhood. A square sinc may be a better idea, converging as $1/x^2$, and being always positive. Figure 7 shows both sinc and square sinc interpolation kernels.

Hann and Hamming This family of interpolation kernels are defined from the same function, namely:

$$H(x) = \begin{cases} \alpha + (1-\alpha)\cos(\frac{2\pi x}{N-1}) & |x| < \frac{N-1}{2} \\ 0 & \text{otherwise} \end{cases}$$

where N is the number of samples in the windowing function. The two functions differ in the choice of α. For Hann, $\alpha = 1/2$ and for Hamming, $\alpha = 0.54$.

Figure 8 shows both kernels. They have the interesting property of always being positive, which limits the Gibbs phenomenon in edge neighbourhoods. Notice that the Hamming kernel is discontinuous. These kernels are widely used in electronics for digital filter design, more information can be found about them in any book related to digital signal processing.

Lanczos and hyperbolic tangent These filters are defined differently but are very similar in spatial domain. The Lanczos function is defined by:

$$L(x) = \begin{cases} \text{sinc}(x)\text{sinc}(x/2) & \text{if } |x| < 2 \\ 0 & \text{otherwise} \end{cases}$$

It shows very strong anti-aliasing properties in Fourier domain, and is easily computed in spatial domain. It is possible to extend this function to superior orders if the kernel is enlarged to use more neighbouring pixels.

The *tanh* kernel derives from the approximation of an ideal box filter by a product of symmetric *tanh* functions. Let us define in Fourier space:

$$H_k(f) = (\frac{tanh(k(x+0.5))+1}{2})(\frac{tanh(k(-x+0.5))+1}{2})$$

Figure 9 shows H_k for several values of k.

Increasing k tends to make H_k converge towards the ideal box filter, thus leads to more ringing in spatial domain. The main difference is that this function decays towards 0 as e^{-x}, which is extremely fast and allows using small pixel neighbourhoods for resampling.

It is not easy to calculate the formal expression of the inverse Fourier transform for this filter. Instead, its expression is numerically computed by digital Fourier transform on enough points to have the kernel on the required number of tabulations.

Figure 10 shows Lanczos and scaled hyperbolic tangent kernels in the image domain.

4.2 Frame coaddition

Reaching the last step of jitter imaging data reduction: from a set of sky-subtracted and aligned images, concatenate all planes to a single image. Several methods are offered.

Linear average Input planes are simply averaged to a single one.

$$I_{out} = \frac{1}{n}\sum_{k=0}^{n} I_k \tag{6}$$

Median average For each pixel position over the detector, a time line is extracted, sorted, and the central (median) value is assigned to the output plane. This non-linear filtering allows extraction of bad pixels and cosmic rays, but does not increase the signal to noise ratio for faint objects, since they tend to be rejected by the median filtering.

Filtered average For each pixel position over the detector, a time line is extracted, sorted, and a rejection is performed on this line in terms of high and low percentages of rejected pixels. This allows extraction of spurious pixel values and still adds up faint signals to increase the overall signal to noise ratio.

Default rejection percentages are set to 5% in both directions.

4.3 Algorithmic optimizations

A quick study of the overall jitter algorithm shows that most of the execution time is spent into pixel sorts and median filters. A detailed study of several methods to reduce the number of operations needed for these two specific steps has been done. The fast sorting problem has received a great deal of attention, and there are too many good methods to try to introduce them here. It is probably enough to know that the quicksort version used in the jitter implementation (`pixel_qsort`) is always at least twice faster as the local `qsort()` function found in the local standard C libraries on all tested Unix flavours.

Searching an array to find the median value is usually done by sorting out the array and then returning the central value. However, it is intuitive that more work is actually done than really needed. Instead of ending up with a sorted array, it should be possible to limit the amount of permutations needed to bring the median value in the center of the array with all values below being smaller than the median, and all values above being greater. Some permutation schemes are known for most used cases (3, 5, 7, 9, 25 values), but unfortunately there is no generic formula giving the best permutation scheme for any given number of values.

Figures11, 12 and 13 show the permutation networks needed to get the median value out in the minimal time. A permutation of two elements a and b is here defined as:

```
pix_swap(a,b)
begin
    if a>b then swap(a,b)
    else do nothing.
end
```

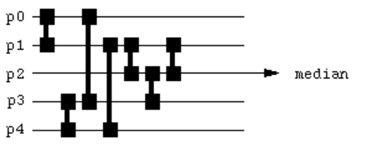

Figure 11: Permutation network for 5 elements

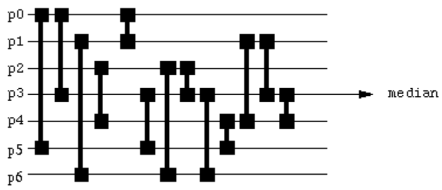

Figure 12: Permutation network for 7 elements

In these figures, a is always the above element, b the bottom one. Example: the permutations needed for 3 values `(p0,p1,p2)` are:

```
pix_swap(p0,p1)
pix_swap(p1,p2)
pix_swap(p0,p1)
```

In the more generic case of N elements in input, from which the median needs to be found, several algorithms have been reviewed. A complete survey of all methods, their results, an implementation for each algorithm in ANSI C and other pointers can be found from the following web site:

`http://www.eso.org/~ndevilla/median.html`

5 Extension to jitter+offset mode

The jitter+offset mode for SOFI and ISAAC must be used in case a fundamental assumption cannot be trusted for the normal jitter mode, namely that a pixel sees mostly sky background along time. This is the case for extended objects for example.

Jitter+offset samples the sky background at offsets at least greater than the detector size. Notice that with an 8-m class telescope, there is indeed no *empty* sky frame, i.e. the photon-collecting power is such that no

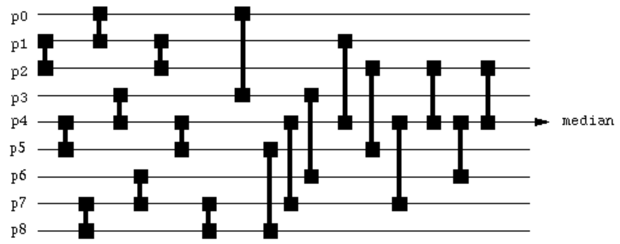

Figure 13: Permutation network for 9 elements

frame contains truly only sky signal. It is often necessary to filter out spurious astronomical objects from sky frames, by means of a 3d median filter for example. In the case of a `jitter+offset` observation, the jitter process will perform a median 3d average of all input sky frames (removing effectively spurious astronomical objects), then subtract this sky from all input object frames. The process then goes on as for normal jitter: feature detection, offset estimation, frame registration, and frame stacking.

The software is always working in normal jitter mode, and will switch itself to `jitter+offset` as soon as at least one frame has been declared as a sky plane in the input list of frames.

6 Implementation details

The above algorithms are all included in a single Unix command called `jitter`, made part of the **eclipse** distribution. This command is configured through an ASCII initialisation file (`jitter.ini`). Implementation choices taken for this command are described in this section.

6.1 Command call

Calling the command with the `-g` option will generate a default configuration file, containing all necessary information to understand how to modify the setup, having in mind the present documentation.

More details about how to call the command and the meaning of each command-line parameter can be found in the associated man page (called by `man jitter` once the `MANPATH` has been correctly updated).

6.2 Portability issues

The `jitter` command has been entirely implemented in ANSI C to allow easy portability over most Unix platforms. Support is actually provided for all POSIX-compliant operating systems including the real-time extensions. This covers at least Sun OS, Solaris, HP-UX, Linux, Ultrix, OSF/1, and IRIX. Other Unix flavours have not been tested yet.

Things to consider when using **eclipse** on different machines are mainly the endian-ness of the local processor (Intel or Motorola type). FITS data are stored in Motorola format, and it improves quite a lot the amount of disk accesses to work on Motorola like processors (Sun stations, PowerPCs, HPs, etc.), since it is possible to work with the input data without need of binary conversion. Intel compatibility is supported, but the price to pay is a (huge) overhead in data I/O due to binary conversions.

To know which kind of processor family your station belongs to, use the `endian` command from **eclipse** .

6.3 Memory issues

Reducing SOFI/ISAAC data means juggling with huge amounts of pixels. A typical jitter template produces 64 frames, using about 256 megabytes of disk space just to store raw frames. For data reduction, it is safer to be able to access about twice the input data size just for temporary storage of intermediate data. In this case, it means that about 800 free megabytes are needed to reduce one single frame set. VLT workstations have been scaled accordingly to allow such heavy processing, but it might not be the case on any user workstation. If you do not have enough RAM on your machine, the process will create its own swap spaces on your disk. You need then a decent workspace available on a local disk to allow processing.

The `jitter` command uses both memory and available disk space to store intermediate data products such as estimated sky. Much use is done of the `mmap()` system call, which belongs to the real-time POSIX extensions. This allows to work on virtually any machine no matter how much RAM+swap is available, as long as there is enough free user disk space available.

6.4 Execution times

For SOFI/ISAAC data, the execution times on an UltraSparc or a similar HP station are about 30 seconds per frame (CPU time). A detailed analysis of the procedure with a profiler shows that reducing 64 SOFI frames reaches an order of 1 billion pixel operations to run the whole algorithm. Most time is spent in sorting out pixel lists for non-linear filtering, which has lead to a detailed analysis of fast algorithms for median search and pixel quick sort (see above paragraph about algorithmic optimizations).

6.5 Code download

eclipse can be downloaded from its WWW site: `http://www.eso.org/eclipse`

7 Acknowledgements

This summary of algorithmic knowledge has been made possible with great help from SOFI/ISAAC people: J.G. Cuby, P.A. Duc, C. Lidman, A. Moorwood. Great help has been brought from the EIS (ESO Imaging Survey) people, to test out the code and detail all algorithms. Algorithms were discussed with too many other people to quote them all here, thanks to all of them for their help and support.

References

[1] Murtagh F., A new approach to point-pattern matching, Publications of the Astronomical Society of the Pacific, #104, 1992, pp 301-307.

[2] Wolberg G., Digital Image Warping, 1990, IEEE Computer Society Press ISBN 0-8186-8944-7.

[3] Wong A.K.C., Salay R., An algorithm for constellation matching, IEEE proceedings of 8th International Conference on Pattern Recognition, 1986, pp 546-554.

[4] Lavine D. et al, Recognition of Spatial Point Patterns, Pattern Recognition Vol. 16, No 3, pp 289-295, 1983.

[5] Hong J., Tang X., A new approach to point-pattern matching, IEEE proceedings of 9th International Conference on Pattern Recognition, 1988, vol. 1, pp 82-84.

[6] Edward J. Groth, A pattern-matching algorithm for two-dimensional coordinate lists, The Astronomical Journal, Vol. 91, Number 5, May 1986, pp 1244-1248.

[7] William Pence, FITSIO a FITS file subroutine library, `http://legacy.gsfc.nasa.gov/fitsio/`

[8] John C. Russ, The image processing handbook (second edition), CRC Press 1995.