A "Region-Growing" Algorithm for Matching of Terrain Images

G. P. Otto
T. K. W. Chau

Department of Computer Science
University College London
Gower Street
London WC1E 6BT

This paper describes and discusses a new algorithm for stereo matching, which has been designed to work well with data from the SPOT satellite.[4] It is basically an extension of Gruen's adaptive least squares correlation algorithm,[11,12] so that whole images can be automatically matched, instead of just selected patches. Initial results on quality and speed are presented, together with a theoretical analysis of the potential speed on both conventional and multi-processor architectures.

1. OVERVIEW

This paper describes and discusses a new algorithm for stereo matching, which has been designed to work well with data from the SPOT satellite.[4] It is basically an extension of Gruen's adaptive least squares correlation algorithm,[11,12] so that whole images can be automatically matched, instead of just selected patches.

2. THE PROBLEM

We wanted an algorithm capable of producing accurate and dense disparity maps quickly and reliably using images produced by the SPOT satellite. (Ultimately, we want accurate digital elevation models, but this paper does not deal with the associated sensor calibration and orientation problems.) As a target, we want our system to be able to achieve height accuracies of 5 m or better (RMS) over all of the visible scene surface. Assuming SPOT images with a base-height ratio of about 1, this means that we need to match to better than 0.5 pixel† wherever possible.

The panchromatic scanner on the SPOT satellite is a linear 6000 element CCD array; 2-dimensional images are produced by composing a succession of 1-D images, taken at 1.5ms intervals, as the satellite orbits the earth at ~ 7km/s. The linear array is approximately perpendicular to the satellite's track, so the 2-D images are approximately square. (SPOT supply the data as 6000 × 6000 images.) For stereo work, the array is frequently tilted so that it is looking at angles off the vertical of up to 25° or so, thus allowing base-height ratios of just over 1. This, and the fact that 1 pixel corresponds to about 10m × 10m on the ground, gives the potential for deriving good quality DEM's from this data. For more information, see (e.g.) Chevrel et al.,[4] Gugan [13] and Dowman.[7]

3. POSSIBLE APPROACHES TO SOLVING THE PROBLEM

Let us start by summarising some important facts about our application area:

- The surface we are looking at is typically continuous. (Overhanging cliffs & such-like are rare.) On the other hand, there may well be cloud and haze between the sensors and that surface, so regions may be occluded or changed in contrast.
- The stereo pairs of images are taken at different times, so transitory phenomena such as clouds and wave-patterns will not match up between the two images.
- The images typically contain lots of small-scale texture (though often with poor contrast), but large-scale features are relatively sparse.
- There are often significant distortions between corresponding image patches — the base-height ratio of the sensor-scene combination is typically about 1 (which makes it easier to attain good height accuracy), and the scenes often contain steeply sloping regions.
- The initial disparity ranges can be large — up to about 1000 pixels (e.g. when trying to match two SPOT images, of base-height ratio ~ 1, of the Himalayas) — even for "ordinary terrain", ranges of 100-200 pixels are typical.
- Obtaining accurate sensor orientation information is relatively difficult/expensive, so we do not wish to use geometrical constraints for the matching unless necessary, or unless not-very-accurate information is adequate.
- Since SPOT uses a scanned line sensor, algorithms which rely upon the epipolar lines being known need to be converted into a successive refinement format (if they are to produce accurate disparities). See Otto[18] for a discussion of this point.

3.1. An area-based (correlation) algorithm is most appropriate for this task

People commonly divide stereo matching algorithms into two kinds: feature-based and area-based. In feature-based althorithms, the original pixel data is converted into some more abstract "features" (often line-segments) before matching, whereas in area-based algorithms the pixel data is compared directly (typically by minimising some measure of "mismatch" over small areas surrounding the points of interest). Feature-based algorithms are typically faster, since converting images into "features" reduces the quantity of data to be handled, and makes the comparison/matching between images easier. For this reason, we started by considering primarily feature-based algorithms.

However, we came to the conclusion that an area-based (correlation) algorithm was more appropriate for our needs, because of the difficulty of simultaneously attaining the accuracy & density goals with a feature-based algorithm. This difficulty arises from the need to be able to locate a "feature" to high accuracy (at least 0.2 pixel) near any position in an image, in order to be able to obtain correspondences accurate to better than 0.5 pixel "densely" over the scene.† (Large regions of the SPOT images we have been working with (views of Aix-en-Provence) have few good edges; in addition, there is significant noise (much of it linear!) in the images we have.)

Area-based algorithms require assumptions about the smoothness of the surface being viewed (e.g. that, locally, it is approximately flat) so that disparities at neighbouring points can be related. However, where those assumptions are valid (which they usually are in this application), the area-based algorithms can produce very accurate answers since they make use of all of the local data. Furthermore, area-based algorithms will perform reasonably well even in regions which are nearly homogeneous.

The area-based algorithm we chose to concentrate on was one described by Gruen,[11,12] because it was claimed to achieve very high accuracy[12,10] (though that was only on selected regions of images), and because it could be made to run at an acceptable speed (see appendix)

† We need "better than", so that we have some margin for error in the conversion from disparities to heights, and for homogeneous and difficult-to-match areas.

† Note that it is difficult to "post-process" the correspondences to increase accuracy, because of the lack of large-scale regularity in our scenes. On the other hand, post-processing can be effectively used to remove "blunders", i.e. grossly inaccurate matches.

even though it allowed for distortions caused by (e.g.) the different viewing angles.

3.2. Outline of Gruen's Algorithm for matching image patches

Gruen's algorithm is an adaptive least-squares correlation algorithm — the basic idea being to minimise the sum-of-the-square-of-the-differences between two image patches, with the minimisation being over a set of parameters specifying how the patches (and their grey-levels) are allowed to be distorted between images. Such distortion can arise from many causes (e.g. perspective distortion); Gruen[11] allows for an affine transformation between coordinates in the images, and an additive distortion to the grey-levels. This geometric distortion is more-or-less equivalent to assuming that the viewed surface is approximately planar (within the region visible in a patch), and that each patch subtends a small angle at the sensor (so that non-linear terms in the perspective distortion can be ignored). Note that it does not assume anything about the angle of inclination of the surface, and that it will cope with large base-height ratios.

Finding the best fit is essentially a multi-parameter optimization problem. Gruen solves this iteratively, by making initial estimates of the parameters, and then linearising the problem in such a way that he has to solve a set of over-constrained linear equations at each stage. (See his papers, and the appendix, for more detail. Alternatively, Chau and Otto[3] give a concise & precise formulation.)

This algorithm is capable of producing high-accuracy results at a moderate computational cost, but its radius of convergence is small — e.g. it needs to be given starting values for disparities which are within a few pixels of the true values.[12]

3.3. Solving the "search" problem

The correlation algorithm can accurately match points in the two images — but it needs to be given good approximations to start with. Rather than attempt some form of general search (over a potentially large range of disparities and distortions), this seemed to be an excellent time to exploit the continuity of the surface being viewed. This led to the following algorithm:

4. DESCRIPTION OF THE REGION-GROWING ALGORITHM

The essence of the algorithm is simple: start with an approximate match between a point in one image and a point in the other; use Gruen's algorithm to produce a more accurate match and the distortion parameters, and use this to predict approximate matches for points in the neighbourhood of the first match. Then use Gruen's algorithm to refine these matches, and so on.

In pseudo-code this becomes:

```
{ INPUTS: two images; 1 or more approximate matches between the images }
set list_to_be_grown_from to empty
for each approximate match
    run Gruen's algorithm
    if it converges
        store result in list_to_be_grown_from
while list_to_be_grown_from is not empty
    pick an item from the list (& remove it from the list)
    for each "neighbour" of the selected match
        if "neighbour" not already matched
            use selected item to predict match
            run Gruen's algorithm using prediction
            if it converges (and satisfies any constraints we might impose)
                store result in list_to_be_grown_from
```

This pseudo-code doesn't mention output — clearly, the results can either be output as they are generated, or at the end, when they can be ordered in whatever fashion is convenient.

Another thing not defined by this pseudo-code is what a "neighbour" is. Since Gruen's algorithm is only applicable when the scene surface is approximately planar†, it does not seem worthwhile to use it directly to attempt to match every pixel in (say) the left hand image to the corresponding point in the right-hand image — if we use Gruen's algorithm on every 5th or 10th pixel, we can predict quite accurately what the matches are for intervening points. For this reason, our current implementation of this algorithm allows us to specify a grid of regularly spaced pixels in the left-hand image, which are the points for which Gruen's algorithm will be used. Then the "neighbours" are the four nearest points on this grid.

4.1. Selecting which match should be used to grow from ...

The remaining major item not specified by this pseudo-code is which match should be selected from **list_to_be_grown_from** on each iteration. Our first attempt used a stack (last-in first-out list), because it was simple and would keep the list size to a minimum. However, this version turned out to be susceptible to mismatches, which could occur when a region of the image was relatively homogeneous, or was obscured by cloud. (See below for more discussion.) To cure this, we decided to use a "best-first" strategy.

The "best-first" strategy is based on assigning a measure of "goodness" to each match we obtain using Gruen's algorithm. Then, when selecting a match to use for prediction, we use the "best" match left in **list_to_be_grown_from**. (This requires the list to be a priority queue — but this is not a notable overhead if a data-structure such as a heap[1] is used.)

4.1.1. Which match is "best"?

Gruen's algorithm can fail in two main ways:

(i) it fails to converge;

(ii) it converges, but to the wrong match.

The first of these failure modes does not trouble the region-growing algorithm too badly, since

(a) it knows that it has failed, and

(b) it is likely to have several more attempts, from other directions.

This latter point is important, since if the failure to converge was due to a bad prediction (because, for example, a ridge or other discontinuity has been crossed), there is a good chance that the predictions from at least one of the other directions will be better. (E.g. the one(s) from the other side of the ridge.)

On the other hand, the second mode of failure could be very serious, for two reasons:

(a) the algorithm does not know that a bad match has occurred, and

(b) it may lead to worse matches in future, if the surrounding regions of the images lack sufficient distinguishing features to ensure that Gruen's algorithm either produces the correct match or nothing at all.

(The first, depth-first, implementation of this algorithm went wrong for this reason — when it attempted to grow through a nearly homogeneous region, it wandered sufficiently far from the correct match that later matches converged wrongly.)

There are two ways of curing this:

- devise tests which ensure that a match is "correct";
- grow from the matches which are (believed to be) most accurate first.

The first method is desirable, but it is not obvious how to do this thoroughly. However, relatively simple tests can detect many errors; for example, using the sensor geometry one can constrain the possible matches to a line — anything too far from this is a blunder.

The second method is relatively easy, so long as one can produce some number which is likely to correlate with the accuracy of the match. A number which seems plausible, and which has succeeded well so far, is the value of the largest eigenvalue of the 2×2 matrix composed of the estimated (co)variances of the line & sample disparities. (See Gruen[11] for the derivation of this matrix.) This number should correlate well with the magnitude of the error in disparity for that match, but the derivation of the estimated (co)variances depends on assumptions about

† Though it can be generalised somewhat — see discussion in section 4.2.

the image noise characteristics which are only approximately true, at best, so there may be better measures of "goodness".

4.2. Choosing the set of allowable distortions

The accuracy of our solution is a trade-off between the following factors:

(i) how well the distortions we allow (or correct for) model the real distortions (within an image patch);

(ii) how many parameters we have to estimate (the more parameters we have to estimate from a given set of data, the less accurate the estimate of any parameter becomes, in general);

(iii) and how big we can make our patches (so that we can "average over" more data, and thus obtain more accurate answers).

This leads to two design questions:

- what are the significant distortions within a given size of patch?
- what is the smallest set of parameters which will compensate for those distortions? (In particular, it may be possible to correct for some of the distortions using just (e.g.) the sensor geometry, rather than measuring them from the data in a patch.)

It is convenient to divide the distortions into two classes: radiometric (which affect the measured grey-level at any corresponding point), and geometric (which affect the positions of corresponding points). The radiometric distortions will arise from causes such as variations in sensor gain, atmospheric haze & so forth. The geometric distortions will arise because of the projection of the viewed surface onto two different viewpoints.

4.2.1. Radiometric distortions

Variations in atmospheric haze alone can cause significant variation in grey-levels. (A factor of two or more in contrast for corresponding patches.) Thus, we need to compensate for changes in contrast between images. In addition, there is often a significant additive offset, so we currently allow for an additive and a multiplicative distortion between left and right images patches. This has worked well so far.

4.2.2. Geometric distortions

The geometric distortions model the way in which the disparities vary in a local region of the images. The disparities will, in general, have both an x and y component†. The x disparity can vary, almost arbitrarily, due to changes in scene height. However, because the earth's surface is relatively smooth (at the scales we are considering), the variation can be modelled quite well (locally) by a linear variation. The y disparity can, in theory, be determined once the sensor orientation is known accurately *and the x disparity is known*. (The y disparity cannot be determined accurately without knowing the scene height (or equivalently, the x disparity) — see Otto.[18]) However, the SPOT satellite's attitude varies slightly & not very predictably during its orbit, so that it is very difficult to calculate the y disparity to better than a pixel or so, unless hundred's of control points are used. Thus, it is easier, and more accurate, to measure the y disparity than to calculate it. For the sake of computational convenience (and because that is the way Gruen[11,12] did it), we also measure, rather than pre-calculate, the variation in y disparity across a patch. However, we would expect some improvement in quality (and speed!) if we utilised the geometric information from the SPOT headers, so that these distortions could be expressed as functions of the other distortions. We intend to experiment with this fairly soon.

This model of the geometric distortion gives us 6 parameters ($x\text{-}disparity$, $y\text{-}disparity$, $\frac{\partial x\text{-}disparity}{\partial x}$, $\frac{\partial x\text{-}disparity}{\partial y}$, $\frac{\partial y\text{-}disparity}{\partial x}$, $\frac{\partial y\text{-}disparity}{\partial y}$) — i.e. we are modelling the disparities (locally) using the first-order terms of a Taylor series.

† We will assume that x increases from left to right across an image, and y increases downward.

This model will break down at discontinuities (such as a ridge), or where the terrain is very rough (e.g. craggy peaks). This has not yet proved to be a significant problem, but more experimentation is required before we can be confident of this. If this algorithm were to be applied to aerial photographs (which have a much higher resolution), then we would need to do some modification to cope with (e.g.) urban areas, with their many sharp changes in slope. In such areas, it may be appropriate to augment our matcher with some edge-based matching, or to allow "folded plate" distortions in the correlations.

5. GENERATING APPROXIMATE MATCHES FOR THE REGION-GROWING TO START FROM

This region-growing algorithm requires a few approximate correspondences to start growing the regions from. These correspondences need to be accurate to about 1-2 pixels, and, ideally, there should be at least one such match in each "isolated region" of the images. (By "isolated region" we mean a region which is surrounded by nearly homogeneous or obscured regions, so that the region-growing algorithm has no good path to follow from other well-textured areas.) We have used only small numbers of approximate correspondences to seed our algorithm (typically 3 or 4); this has worked well so far, but we will probably use more when we automate this stage properly.

Such correspondences can be generated by hand (e.g. any ground control features used for sensor orientation), or automatically, by an algorithm such as Barnard and Thompson's algorithm.[2] Kevin Collins has modified this algorithm to work with SPOT data — see Collins et al.,[5] Day and Muller,[6] and Muller et al.[17]

Another method, which shows promise, but which hasn't been fully tested yet, is based on the idea of picking isolated "features" using some straightforward feature detector (e.g. critical points of the intensity surface, or Moravec[16] or Forstner[8]), finding the largest set of matches which are geometrically consistent with the a priori data, and with each other, and then keeping only those points which are uniquely matched and which correlate well (using e.g. Gruen) with each other.

6. HOW GOOD IS THE REGION-GROWING ALGORITHM?

6.1. How good are the results?

We only have preliminary results about the quality and robustness of this algorithm. One example will give the flavour of the results so far:

Using the region-growing algorithm to produce a 30m DEM over a pair of 240 × 240 subimages extracted from SPOT images of Aix-en-Provence, we obtained plausible matches for over 99% of the matchable points. (Some points were not visible in both images, or were obscured by cloud.) Comparing the matched points against a DEM derived from underflight photography, we found:

Mean difference = -3.716 m
Standard deviation = 7.955 m
RMS difference = 8.780 m

(No blunder detection was used.) The DEM derived from the underflight photography is estimated to have an RMS height error of about 1-2m; in addition, much of the remaining error is due to the sensor model used. The remaining error is that due to the matching algorithm; this error appears to be significantly less than 0.5 pixels (RMS), possibly as low as 0.1 pixels (RMS). One notable cause of error appears to be the smoothing effect of the largish correlation patches used (21 × 21 pixels), since in the area being viewed there were several narrow ravines. We expect these results to improve when we have incorporated various refinements into the algorithm and the sensor model.

Colleagues in the Department of Photogrammetry and Surveying are going to do a thorough analysis of the quality of results obtained by this algorithm. See Day and Muller[6] for a description of some of their early results.

6.2. How fast is this algorithm?

The speed of the algorithm depends partly on the implementation, and partly on the parameters used (e.g. patch size and grid spacing). Since we are still in the process of evaluating and tuning it for real data, we cannot yet give a definitive answer. Thus, this section will just outline our knowledge so far.

6.2.1. Speed of Gruen's algorithm

On a single SUN 3/180, using a 68881 floating point coprocessor, our current implementation takes just under ½ sec to do one iteration of Gruen's algorithm, using a patch size of 21 × 21, or about ¼ sec for a patch size of 15 × 15. (The time is approximately proportional to the area of the patches used.) Recoding to use integer arithmetic wherever possible would probably speed this up by a factor of about 4. However, a 20 MHz T800 Transputer† should be able to reduce these times to about 0.06 and 0.03 seconds respectively, if good code is produced for the critical sections. (See appendix.)

For any given pair of patches, these times need to be multiplied by the number of iterations required to get accurate convergence — our initial experiments indicate that typically only one or two iterations are needed when Gruen's algorithm is used as part of the region-growing algorithm.

Overall, then, matching one pair of patches on a Transputer will typically take between 0.05s and 0.2s, depending on the parameters used, and the image characteristics.

6.2.2. Parallelizing the region-growing algorithm

Since Gruen's algorithm requires significant computation (at least 50 ms worth) on comparatively little data (two image patches, together with the initial values of the parameters add up to a few kbytes), an easy and efficient way of parallelising the region-growing algorithm is to have one central "master", which manages the priority queue & the image data, and many "workers", which just apply Gruen's algorithm to whatever patches they are given, and return their results to the "master". Since we wish to match many more points than we have Transputers, and Transputer links are fast enough to transfer the data faster than the CPU's can process it, this automatically leads to good load-balancing and a high processor utilization, with little "parallelization" overhead. i.e. We expect the speedup to be linear in the number of processors. Furthermore, the "workers" do not need much memory — 100-200kbytes is ample.

If we have enough "workers", then the one "master" will become a bottleneck. Initial analysis has shown that this will not happen until there are at least 30 "workers", and probably more. Even then, it would be relatively straightforward to partition the master over two or more processors.

We have already parallelised our algorithm on a network of SUN workstations, and speedup is, as predicted, linear up to 15 processors (which was all we could get our hands on!). A Transputer implementation has just begun (in collaboration with the Royal Signals & Radar Establishment, Malvern).

6.3. Summary — speed

Using 30 T800 Transputers, and the algorithm above, it should be possible to produce a high-quality, dense disparity map from a pair of SPOT images in about 2 hours.‡ Clearly, this time will vary, depending upon the image chacteristics and the algorithm parameters; but this time should be fairly realistic and representative.

† One 20MHz T800 Transputer can manage between 1 and 1.5 MFlops on floating-point intensive code. This, together with the fast serial links, makes the Transputer a very suitable building block for this application.

‡ This figure is based on: each point requiring about 0.2 CPU-seconds to match (on average); matching all the points on a square grid consisting of every 6th pixel horizontally or vertically (ie. 1 in 36 pixels of the original); 6000 × 6000 images.

7. SUMMARY AND CONCLUSIONS

We have described an algorithm which is capable of producing high-quality, dense range-maps, which runs at a reasonable rate on conventional processors, and which can achieve linear speedup on multi-processor architectures. It is applicable when the scene being viewed has significant texture, but few discontinuities, and a full range-map is required.

The speed is independent of the range of disparities present in the images, and the algorithm does not require any knowledge of the sensor geometry. (Though such knowledge can be used to get more accurate matches, and to assist in the initial "seeding" of the algorithm with approximate matches.)

8. ACKNOWLEDGEMENTS

This work was done as part of a collaborative* research project, funded by the Alvey Directorate. We would also like to thank our colleagues for help and useful discussions; most notably, Kevin Collins (of RSRE) helped us with Transputer-related issues, while Tim Day and J-P. Muller (both of UCL-P&S) provided us with input & test data, and helped with the quality assessment.

APPENDIX — Speed of Gruen's algorithm on one processor

This section analyses the speed potential of Gruen's stereo matching algorithm,[11,12] running on a single processor machine. It assumes that it is being used for matching **one** patch in one image to **one** patch in the other. This is because Gruen is likely to be parallelized (for a multi-processor system) by doing such matches concurrently on separate processors, with little or no interaction between them. See main text for a fuller discussion.

A1. Assumptions, parameters and notation used in this analysis

The (resampled) image patches will be regarded as rectangular, of size $N \times M$. The number of parameters to be determined will be denoted by p. The notation will generally follow Gruen;[11] exceptions will be noted where they occur.

Edge (of matrix/image) effects will not be discussed since they would complicate the analysis, but don't significantly affect the speed.

A2. "Basic" Gruen

This section discusses a "no bells or whistles" version of Gruen's algorithm (the core of his 1985 paper) — the next section will discuss the possible refinements, and their effects upon the speed.

As mentioned above, the algorithm is iterative, so we will begin by discussing the steps required within one iteration.

A2.1 Resampling an image patch

The algorithm works by matching image patches — typically rectangles of between 15×15 and 30×30 pixels. The left-hand patch is just a subwindow of the left-hand image, and stays constant throughout.† The right-hand patch is a distorted and resampled subwindow from the right-hand image. The distortion is an affine transformation (which includes the disparity information), together (typically) with a radiometric adjustment of some kind. (E.g. a constant added to each point of the patch‡.) Gruen uses bilinear interpolation.

Essence of the resampling:

* The collaborating partners are the Department of Photogrammetry and Surveying at University College London, the Royal Signals and Radar Establishment, Laser-Scan Laboratories and Thorn-EMI Central Research Laboratory.

† The 1986 paper also allows the left-hand patch to move, so that the final disparity estimate is obtained for a fixed x,y position with respect to the ground. This isn't relevant to us, so we will ignore it here.

‡ Such an offset doesn't need to be done during the resampling. All it requires is the corresponding distortion parameter, and appropriate design matrix entries.

```
FOR x = 0 TO N−1 DO              /* coords in resampled patch */
    FOR y = 0 TO M−1 DO
        x_from = a*x + b*y + c; /* a, b, c constants */
        y_from = d*x + e*y + f; /* d, e, f constants */
        patch[x,y] = interpolate(rhimage,x_from,y_from);
```

At first glance, this seems to require 4 multiplications and several additions within the inner loop to calculate x_from & y_from. However, by moving constant expressions out of the loop & so forth, the new values of x_from, y_from can be calculated from the old with just one addition each. So, let us concentrate on the interpolation part for a while.

The basis of the (bilinear) interpolation is:

```
int_x = ⌊x_from⌋ ;
int_y = ⌊y_from⌋ ;
fract_x = x_from − int_x;
fract_y = y_from − int_y;
temp1 = (1−fract_x)*rhimage[int_x,int_y] +
        fract_x*rhimage[int_x+1,int_y];
temp2 = (1−fract_x)*rhimage[int_x,int_y+1] +
        fract_x*rhimage[int_x+1,int_y+1];
result = (1−fract_y)*temp1 + fract_y*temp2;
```

This can be rearranged into:

```
[ calculate int_x, int_y, fract_x, fract_y ]
temp1 = rhimage[int_x,int_y] + fract_x*(rhimage[int_x+1,int_y] −
        rhimage[int_x,int_y]);
temp2 = rhimage[int_x,int_y+1] + fract_x*(rhimage[int_x+1,int_y+1] −
        rhimage[int_x,int_y+1]);
result = temp1 + fract_y*(temp2−temp1);
```

In addition to calculating int_[xy], fract_[xy], and some array accesses, this requires 3 multiplications each time, and 6 additions/subtractions.† int_[xy], fract_[xy] can be calculated directly instead of calculating x_from, y_from — thus 4 additions are needed within the loop to calculate these. (In practice, int_x will stay constant for many iterations of the inner loop, so this could probably be reduced to 3 additions, on average.)

The array accesses can be speeded up by using pointers and registers (in a language like C[14]) or, in occam,[19] "IS" and local variables, so we will just regard them as part of the "housekeeping". The multiplications and additions/subtractions can all be done using integer arithmetic, if the numbers are scaled suitably. Thus, we can assume that integer arithmetic has been used, if it is faster on the system being used.

Overall then, the resampling will require about

$$MN.(3\ \textit{multiplications} + 10\ \textit{additions} + \textit{some housekeeping})$$

A2.2. Forming the "design matrix" and "observation equations"

The "design matrix" (called A in Gruen's papers) is a matrix with MN rows (one for each x,y position within the resampled image patches), and p columns. In the 1985 version, each row of the matrix has the form:

$$(g_x \;\; x.g_x \;\; y.g_x \;\; g_y \;\; x.g_y \;\; y.g_y \;\; 1)$$

where g_x denotes $\frac{\partial g}{\partial x}$, and g_y denotes $\frac{\partial g}{\partial y}$, evaluated at the appropriate (x,y) coordinates within the patch. To form each row of this, we need to calculate g_x, g_y (essentially two subtractions to form the first difference approximations), and then multiply these by x, y appropriately (4 multiplications). The time to insert the final 1 will be regarded as part of the house-keeping.

With care & implicit decimal points, these calculations can also be done using integer arithmetic. Overall then, forming the design matrix will require about

$$MN.(4\ \textit{multiplications} + 2\ \textit{additions} + \textit{some housekeeping})$$

† Further simplification is possible if the remapping is a pure translation, since then fract_[xy] are constants.

Also required for the observation matrix is the vector l, which is an NM column vector, each of whose elements is $f(x,y)-g(x,y)$ for the appropriate point of the image patch. This adds another

$$MN.(\textit{additions} + \textit{some housekeeping})$$

A2.3. Solving the over-constrained linear equations

To do this, Gruen first calculates A^TPA and A^TPl. P is a $NM \times NM$ matrix specifying the weights to be used when summing the squares of the differences of the distorted patches. For the simple case, it is equal to the identity matrix, and can be ignored.

In general, to multiply an $n \times m$ matrix by a $m \times p$ matrix requires nmp multiplications + $\sim nmp$ additions. (There are cunning methods which are slightly faster on large matrices, but they are unlikely to be faster on this size of matrix. See Aho[1] for an introduction to these methods.)

Thus, to calculate A^TPA will require roughly $NMp^2.(\textit{addition} + \textit{multiplication} + \textit{housekeeping})$ operations, since the transpose can be done as part of the matrix multiplication. In fact, we can halve this, since A^TPA will be symmetric, so we need only calculate the upper (or lower) triangle. Again, with careful juggling, integer arithmetic can be used if it is faster.

To calculate A^TPl will require about NMp additions and multiplications — integer arithmetic can again be used.

If we denote the column vector consisting of the (updates to) the parameters as $\underline{\alpha}$ (Gruen uses x in his paper), then we now have that

$$(A^TPA)\underline{\alpha} = (A^Tpl)$$

Since A^TPA is a fairly small, real symmetric positive definite matrix, this can most easily & efficiently be solved using Cholesky decomposition. (See Stoer,[20] chapters 4 and 8, for an explanation of this method and a discussion of the alternatives.) This requires about $p^3/6$ multiplications and additions, + p square roots, to decompose the matrix into a product of triangular matrices. Then, $\sim 2.p^2$ operations (multiplications and additions) are required to solve for $\underline{\alpha}$ given this decomposition.

These matrix operations must be done in floating point (preferably double precision) to avoid overflow and/or significant rounding errors.

A2.4. Termination test

This is very simple — the iterations finish when the changes in the remapping parameters fall below a threshold. This requires 6 absolute value operations, and 6 comparisons — negligible in comparison to the other operations.

A2.5. Summary — operations, time, per iteration

Adding up the figures above, we get

$$(\tfrac{p^2}{2}+p+7).MN\ \textit{mults.} + (\tfrac{p^2}{2}+p+13).MN\ \textit{add} +$$

$$(\tfrac{p^3}{6}+2p^2)\ \textit{floating point operations} + p\ \textit{square roots} + \textit{housekeeping}$$

If we take N, M = 20, and p = 7 (fairly typical values), this gives us about

$$15000\ \textit{mult.} + 18000\ \textit{add} + 155\ \textit{floating pt. ops} + 7\ \textit{square roots} + \textit{housekeeping}$$

per iteration. Clearly, unless floating point operations are very much slower than integer operations, we can neglect the specifically floating point operations in comparison with the other arithmetic. (N.B. Floating point *may* be used for the other operations, but only if it is faster.) If we assume that these operations will take of the order of a μs each (plausible if using floating point on a T800 Transputer), this gives us a total time of about 34ms — so if we allow a bit extra for housekeeping & so forth, we get a round figure of about 50ms.

Alternatively, if we assume that we use integer arithmetic on a Transputer (or a SUN-3), then a multiplication will take about 2μs, and an addition/subtraction ≤ ½μs, giving a total time of about 40ms per iteration for the calculations. Adding in something for housekeeping

etc, we get a rough estimate of about 60ms per iteration.

(Caveat: these times could easily be larger or smaller by a factor of 2 depending on details of the hardware and coding.)

A2.6. Comparison with actual timings

A2.6.1 Our timings

See section 6.2.1. The timings there are compatible with the above analysis, especially since we have not yet fully optimized our program.

A2.6.2 Gruen's timings

Gruen's 1986 paper gives a time of ~2 sec/iteration for a version broadly similar to the above, on a VAX-11/750, which is about 30 times slower than the estimate. Why the discrepancy?

We don't know — but it is very likely that Gruen uses floating point arithmetic throughout (which we expect is notably slower on a VAX-11/750), and he probably hasn't optimized the program as much as the above analysis implies. These two points could well account for the discrepancy.

A2.7. How many iterations?

The number of interations required depends heavily upon the input data (image & initial parameter values), so it is very difficult to get a good theoretical estimate. Section 6.2.1 gives our experimental data.

A3. Refinements of the basic algorithm

A3.1. Adaptability

Gruen's 1985 paper discusses the pro's and con's of having a many-parameter model to fit to the data, and suggests that an "adaptive" approach be used — i.e. an approach which determines which parameters are "nondeterminable", and which excludes them from further analysis. He refers to a possible rejection strategy in Gruen[9] (which we haven't seen), but doesn't appear to be actually using any such strategy in either his 1985 or 1986 papers. We will therefore not consider it further here.

A3.2. Estimating the "precision"

One of the reasons for linearizing in the way Gruen does is that the equations are then in the form of a "Gauss-Markov estimation model". Given some (rather dubious) assumptions about the distributions, means and (co)variances of the errors, he can then produce some estimates for the standard deviations of the parameter estimates. These require only a few operations once the Cholesky inversion of A^TPA has been done, and they only need to be done after the iterations have terminated, so they will not significantly increase the matching time.

A3.3. Collinearity constraints

If the camera geometry is known accurately†, then we know the line in the 2nd image along which the match corresponding to a point in the 1st image must lie, so we can eliminate one of the free parameters. (And thus get more accurate answers.) This can either be done by reparameterising the distortions (which leads to a different design matrix & so forth, with one fewer parameters), or by including constraints, commonly known as "collinearity constraints".

Gruen[12] uses a "penalty method" for this (see, for example, Luenberger[15]) in which violation of the constraint is weighted very heavily into the least squares sum being minimised. This allows the same solution method as before to be used, and ensures that the constraints are very nearly satisfied.

This does not significantly increase the run-time, since only a few more numbers (see his paper) need to be calculated each time, and these then get added in (in a suitable fashion) to A^TPA. (Caveat: it *might* affect the number of iterations required.)

† In our application, this might occur if we were "patching up" regions of the image where the original matching performed poorly for some reason — e.g. relatively homogeneous or linear features.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. S. T. Barnard and W. B. Thompson, "Disparity Analysis of Images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-2, no. 4, pp. 333-340, Jul 1980.
3. T. K. W. Chau and G. P. Otto, "Design and Implementation Notes for UCL-CS's version of Gruen's Stereo Matching Algorithm," MMI-137 (UCL-CS) working paper 17, Dept. of Computer Science, University College London, London, Nov., 1987.
4. M. Chevrel, M. Courtis, and G. Weill, "The SPOT satellite remote sensing mission," *Photogrammetric Engineering and Remote Sensing*, vol. 47, no. 8, pp. 1163-1171, 1981.
5. K. A. Collins and J. B. G. Roberts, "Stereo Matching of Satellite Images with Transputers," *Proceedings of IEEE International Conference on Systolic Arrays*, San Diego, Ca., May 1988.
6. T. Day and J-P. Muller, "Quality Assessment of Digital Elevation Models produced by automatic stereo matchers from SPOT image pairs," *Proceedings of XVI Congress of the International Society of Photogrammetry and Remote Sensing*, 1988. (Commission III.)
7. I. J. Dowman, "The Prospects for Topographic Mapping Using SPOT Data," *Proceedings of the SPOT-87 Conference*, 1987.
8. W. Förstner and E. Gülch, *A Fast Operator for Detection and Precise Location of Distinct Points, Corners and Centres of Circular Features*, June 1987. ISPRS meeting in Interlaken
9. A. Gruen, "Processing of Amateur Photography," *Proceedings of XVth Congress of the International Society of Photogrammetry and Remote Sensing*, Rio de Janeiro, 1984. (Commission V.)
10. A. Gruen, *Towards Real-Time Photogrammetry*, 1987. Invited Paper to the 41st Photogrammetric Week, Stuttgart, Sept 14-19, 1987
11. A. W. Gruen, "Adaptive least squares correlation: A powerful image matching technique," *S. Afr. J. of Photogrammetry, Remote Sensing and Cartography*, vol. 14, no. 3, 1985.
12. A. W. Gruen and E. P. Baltsavias, "High Precision Image Matching for Digital Terrain Model Generation," *International Archives of Photogrammetry*, vol. 25, no. 3, p. 254, 1986.
13. D. J. Gugan, "Practical Aspects of Topographic Mapping from SPOT Imagery," *Photogrammetric Record*, vol. 12, no. 69, pp. 349-355, April 1987.
14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
15. D. G. Luenberger, *Linear and Nonlinear Programming*, Addison-Wesley, 1984.
16. H. P. Moravec, "Rover Visual Obstacle Avoidance," *Proc. 7th International Joint Conference on Artificial Intelligence*, vol. 2, pp. 785-790, Vancouver, 1981.
17. J-P. Muller, K. A. Collins, G. P. Otto, and J. B. G. Roberts, "Structure of Stereo Matching using Transputer Arrays," *Proceedings of XVI Congress of the International Society of Photogrammetry and Remote Sensing*, 1988. (Commission III.)
18. G. P. Otto, "Rectification of SPOT Data for Stereo Image Matching," *Proceedings of XVI Congress of the International Society of Photogrammetry and Remote Sensing*, 1988. (Commission III.)
19. D. Pountain, *A Tutorial Introduction to OCCAM Programming*, Inmos, Bristol, 1987. (Including language definition by D. May)
20. J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980.