Registration of two-sided documents suffering from bleed through

Report of CSI4900 project
presented to
Prof. Eric Dubois

by
Yohan Bienvenue
1240071

April 11, 2001

Summary

When documents are written on both sides, the ink might bleed through the paper. This is particularly true with old documents. Reading and studying such documents is not easy when serious bleed through is present. This report proposes a solution to correct the problem and shows how the first part of this solution, the registration process, was implemented. Using image processing techniques, the bleed through can be removed using thresholds but not before the recto and verso sides have been properly registered. First, both sides of the document are digitized. The verso is then flipped and corrected so that the bleed through coordinates of one side correspond exactly with the coordinates of the original writing on the other. This needs to be done because both sides are not scanned precisely the same way and there might be a shift and a rotation difference between the two. To do this we use an affine transformation of six parameters. The parameters are found by optimizing the alignment process. Performance is an issue when applying this algorithm on large size images and techniques are used to create a sample set of pixels, which will be used to estimate the results. It was found that by uniformly choosing only some pixel horizontally and vertically the performance was greatly increased without affecting the alignment results. A technique that chooses only blocks of pixels was also used in combination with the first one to increase performance even more. For certain types of images, a perfect alignment process was not obtained and the algorithm might have to be applied on sections of those images instead of the whole image at once.

Table of contents

Introduction

When writing on both sides of a piece of paper, there is a risk we could create a problem known as *bleed through*. As it can be seen on Figure 1, ink written on the verso side can become visible on the recto side and vice versa. In the extreme, the ink might actually *bleed through* the paper. The problem is quite obvious: trying to read a document suffering from *bleed through* becomes a difficult task, as writings from both sides of the document overlap. This report offers a solution to the problem by describing a method that eliminates the *bleed through* on both sides of the document using image processing techniques. First, the theory of the solution will be explained thoroughly. Then it will be showed how the first part of this solution, the registration, was implemented in the *Matlab* language. Finally, results will be presented and then analyzed to see the efficiency of the implemented solution. I would like to thank Professor Eric Dubois who helped me understand the basics and who offered me great support during the semester.

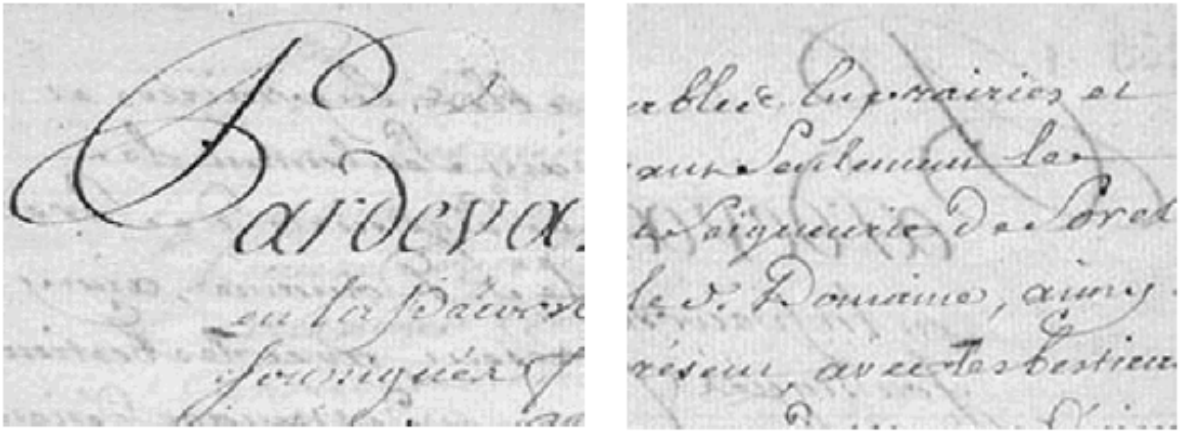


Figure 1: Document with a *bleed through* problem.

Bleed through removal algorithm

Inapplicable solution

First, it should be said that the use of a simple intensity threshold to eliminate the bleed through is not possible. As it can be seen in Figure 1 above, the intensity of the *bleed through* is sometimes very similar to the intensity of the original writing. Too similar to dissociate them with a threshold. What will be used instead are image processing techniques.

Overview

The algorithm is divided in two parts: the registration and the restoration. The registration consists in digitizing both the recto and the document and try to perfectly make the recto and the flipped verso overlap. See Figure 2. By having the *bleed through* perfectly aligned with the original writing (the writing that caused this *bleed through*), it is possible to eliminate the *bleed through* and restore the document using thresholds.

Figure 2: Perfect alignment

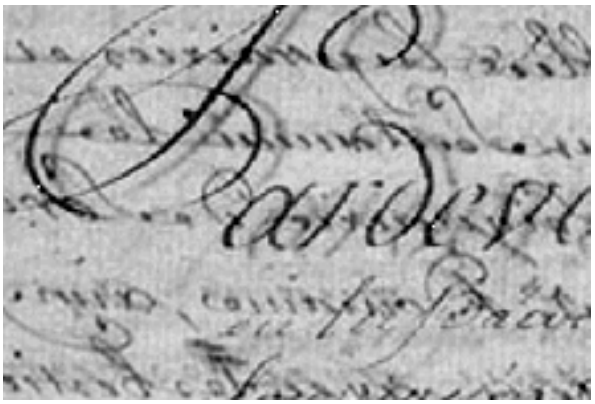


Figure 3: Starting Alignment

Registration process

However, doing this alignment is not that simple. It is improbable that both the recto and verso will be scanned exactly the same way. More likely, some small shifts and rotation will exists between the two resulting images. By trying to do the alignment right away, the situation shown in Figure 3 will probably happen. Therefore, we need some way of correcting the verso so that the alignment becomes perfect, as in Figure 2.

Let us denote the recto of the document as $f_r(x,y)$ and the flipped verso as $f_v(x,y)$. In order to obtain an ideal flipped verso, $f_v^I(x,y)$, we need to apply some transformation to $f_v(x,y)$. This can be done by doing an affine transformation denoted by A_t.

Equation 1: $f_v^I(x,y) = A_t * f_v(x,y)$

This affine transformation involves six parameters: t11, t12, t21, t22 are used to correct the rotation while t13 and t23 correct the horizontal and vertical shifts respectively.

Now, the question is how to find those parameters, that is, the parameters that will correct the verso perfectly. We use what is called the *difference*. The *difference* can be seen as the difference that remains between the recto and the flipped transformed verso when they are aligned. It is a scalar. Let it be denoted by:

Equation 2: $difference = \| f_r - A_t * f_v \|^2$

It is clear that the smaller the difference is, the more perfect the alignment will be. This all depends on our choice of parameters for the transformation. We can find an estimation of those parameters by solving the following optimization problem:

Equation 3: $t_e = \arg\min_t \| f_r - A_t * f_v \|^2$

This equation means that we wish to minimize the difference using the affine transformation parameters. It will return t_e, the estimated *best* parameters.
Using those estimated parameters we obtain our estimated ideal verso:

Equation 4: $f_v^I = At_e * f_v$

Restoration process

At this point, we can proceed to the restoration of the document. Although it is beyond the scope of the project, the restoration process will be briefly explained (for the recto side) to help in the comprehension of the whole *bleed through* removal algorithm process.

The idea is to replace the pixels of the recto corresponding to *bleed through* with the background. The pixels of the recto might fall in one of those four categories:

1- There is recto writing but no *bleed through*
2- There is *bleed through* but no recto writing
3- There is no recto writing and no *bleed through*
4- There is both recto writing and *bleed through*

Using thresholds, we can identify when a pixel falls in category 2 and we can then replace it with the background value. It is now clear why we need to obtain a perfect alignment in the registration process. If we try to find in which category the pixel falls with a bad alignment, the *bleed through* pixels don't even correspond to the pixels of the recto writing. They have to correspond exactly, like in the original document before the scanning process.

Implementation of the registration process in *Matlab*

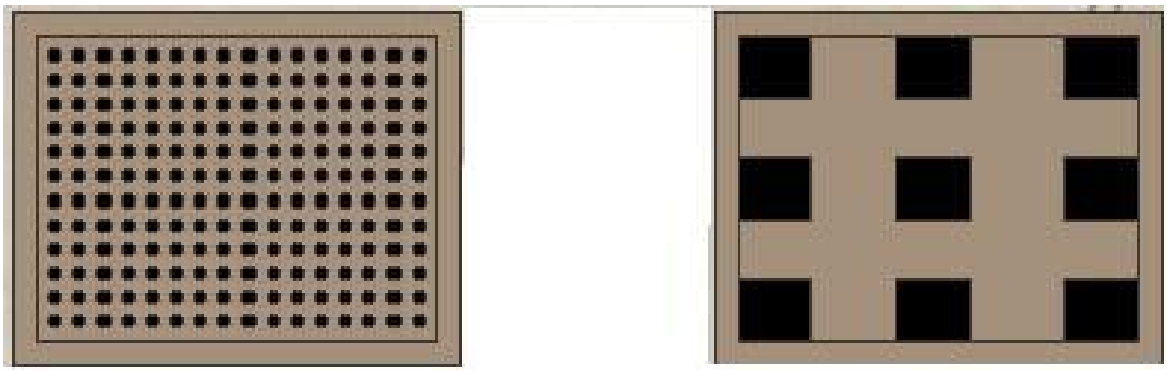
Implementing the registration process in *Matlab* is relatively easy. Built-in functions do a lot to spare us from complex mathematical computations. Please see the code in annex for details not covered in the following section.

Goals

The main goal of this project was performance, that is, to create an implementation giving good results, in the least amount of cycles. The first obvious way to do this is to avoid loops (i.e. FOR loop) as much as possible. There are many ways in Matlab to work-around loops and achieve the same results. One of those is, again, to use Matlab built-in functions. We can manipulate matrices with loops but Matlab functions are optimized for that task and, when used, save a lot of cycles. The program created for this project applied this rule severely. Loops were used only if absolutely necessary.

Another way to save some cycles is to perform all the calculations on a sample of pixels instead of on the entire image. Fortunately, good results are possible using only a fraction of the original pixels. The set of pixels we use is named **S**. At the start of the algorithm, **S** represents all pixels but we reduce **S** by using "**S** reducing techniques". See Figure 4. No matter what technique we use tough, it must select the pixels uniformly throughout the image. The first method consists in choosing every K^{th} pixels horizontally and vertically. The second method used involves using blocks of pixels throughout the image. Both methods can be combined to reduce **S** even more. However, reducing **S** too much will result in a loss of information. Note that an amount of pixels around the perimeter of the image (a border) is not included in **S**. The reason is that the results of a transformation on

these pixels could be out-of-bound. In the end, **S** will be used to estimate the results we would obtain if we used all pixels, using much less cycles.



a) b)

Figure 4: a) Use only every K pixels b) Use blocks of pixels

Overview

Now, for the actual implementation. The algorithm is divided in two halves. The first half registers the recto and verso, that is, it estimates the best transformation parameters. The second half uses those parameters to obtain a visual representation of the alignment.

Defining "**S**"

The first thing the registration algorithm needs to do is define **S**. To do this we start be defining two vectors: SX and SY, representing the vector of x-coordinates and y-coordinates respectively. The border is removed by clipping those vectors directly. Then it is time to apply one or more "**S** reducing technique". The first technique, see Figure4a, is quite straightforward. All we have to do is use a *step* in the range of values to obtain the desired effect.

Ex: To use only every 15 pixels in SX, we do SX = SX(1: 15 : length(SX));

The second technique is quite the opposite. Implementing it without loops is not an easy task. First, we have to make sure that the length of each vector is divisible by the block size. If not, we use padding to make it so. We then reshape the vector in the following manner:

Let SX = [1, 2, 3, 4, 5, 6, 7,…., 12] and BLOCK_SIZE = 4
Then we reshape SX so that it is now in the following form:
SX = [1, 2, 3, 4
5, 6, 7, 8
9, 10, 11, 12]

In other words SX's dimensions are (LENGTH / BLOCK_SIZE) x BLOCK_SIZE. The division is why we wanted the length to be divisible by the block size (we want a integer result). Now, we can delete the lines we want using a simple *step* technique as seen before. The idea is that by deleting those lines, we actually delete entire blocks.
SX = [1, 2, 3, 4, 9, 10, 11, 12] after deleting line#2

We can combine both techniques easily. Simply use the vectors obtained with one technique for the other.

Once this is done we can use the *meshgrid* function to combine SX and SY together and easily obtain **S**.

If SX = [1,2,3] and SY = [1, 4] then S = [SX; SY] = [1, 2, 3, 1, 2, 3
1, 1, 1, 4, 4, 4]

Computation of the *difference*

Before we go on with the next step, we need to explain how the *difference* is calculated. Computing the difference is like a mini algorithm in it's own and, actually, there is even a separate file called *findDiff6par* (find the difference using six parameters) that does this. Although it is not always used when the difference needs to be computed. The six parameters are first used to transform (correct) the image's coordinates. See Figure 5.

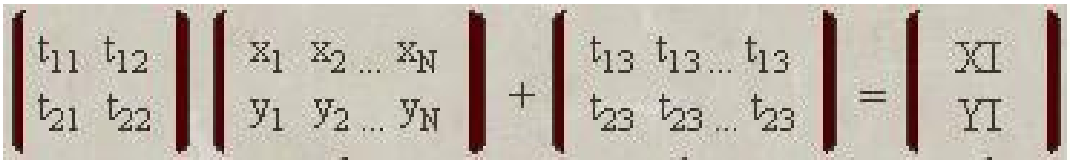


Figure 5: Transformation of image coordinates

We now have the new coordinates (XI and YI in Figure 5) but not the intensity of the pixels at these new coordinates. We can find those by interpolating them using the old intensities and the new coordinates. The *interp2* (2-dimensional interpolation) function is used to do this. It simply returns the new intensity vector. The type chosen is bi-linear since it is the one that gave the best performance. Here is a simple example.

Ex: ZI = interp2(image, Z, XI, YI, '*linear');

Note the *. Since XI and YI elements are equally spaced, we can add the * before the type name to interpolate much faster.

Finally, we can compute the difference using the new intensity vector.

Estimation of shift parameters before optimization

The next logical step is to proceed in finding the best parameters by solving the optimization problem discussed before. However, the optimization function used in Matlab requires a start point, that is, a start value for the parameters. The rotation parameters, t11, t12, t21 and t22, are left alone and represent the identity matrix but the shift parameters, t13 and t23, need to be estimated first. This allows the optimization function to converge on the solution much faster. To do this we simply use a FOR loop on the mini algorithm that computes the difference, where the loop counter is the parameter itself, and keep the parameters that give the smallest difference. It is crude and uses a lot of cycles but it is critical for the optimization function. Failing to do this results in the optimization function to find a local minimum most of the time.

Optimization

Performing the optimization is simple. The function *fminu* is used. It works by minimizing the *findDiff6par* function mentioned earlier. Once it converged to a solution according to some tolerance level, it returns the *best* parameters. Here is an example.

Ex: best_parameters = fminu('findDiff6par', parameters, options);

The *parameters* parameter is a vector containing the six transformation parameters and the *options* parameter is used to choose options like tolerances and step length.
Once this is done, the official work is completed. But more can be done using the *best* parameters, like display of the alignment.

Display

To see the effect of the *best* parameters visually, we can display the alignment and see if it is perfect, near-perfect or way-off. We simply re-do some of the work that was done before but this time we do it using all pixels instead of **S**. We need to do this to obtain the complete image. Needless to say that doing this is very time consuming and stands for 90% of the algorithm's execution time. In short, we simply define **S** as representing all pixels. We then transform the flipped verso like in Figure 5 using the *best* parameters and finally interpolate the final verso intensities by interpolating again. We then add the recto with the transformed verso, display the result and watch the alignment.

Analysis of the Results

Performance

The time required to complete the first part of the algorithm is very reasonable. Although it strongly depends on the size of **S**. Trying to execute the algorithm using all pixels is not a good idea. In fact, even for a very small image it takes several minutes to obtain the parameters. Techniques to reduce **S**, have to be used. The *block* technique (see Figure4b) used alone doesn't really help. **S** is not reduced enough to obtain a reasonable performance. The *pixel* technique (see Figure4a) is much more effective. It increases performance tremendously. When both techniques are used together, the performance is excellent. Some precise results are shown below.

Alignment

The alignment results are mixed, depending on the size of the images and the technique used to reduce **S**. For fair size images (1000x1000) the results are almost perfect regardless of the method chosen to reduce **S**. For bigger images, tough, it seems the technique used to reduce **S** does matter. The *pixel* technique used alone gives good results but when combined with the *block* method, bad alignments start to happen. For extra big images, the alignment is rarely good, no matter what technique was used. Hopefully, this will be solve in future work. Perhaps applying the algorithm on pieces of a big image instead of the whole image and then combine the results might solve this problem. Some visual alignment results are shown below.

Some simulation results (450mhz, 128RAM computer)

1)Test image (See annex for original recto and verso sides of this image)
Dimension: 1088x1186
Resolution: 150dpi

Pixel technique (Size of K = 15):

Time to execute registration process:	25.6 seconds
Time to for registration process + display routine:	52.3 seconds
Size of S:	5700
Difference:	38.286
#function evaluations:	237

*See Figure 6.

Combined technique (Size of K = 15, Size of Block = 10):

Time to execute registration process:	9.8 seconds
Time to for registration process + display routine:	33.2 seconds
Size of S:	1190
Difference:	9.541
#function evaluations:	305

*See Figure 7.

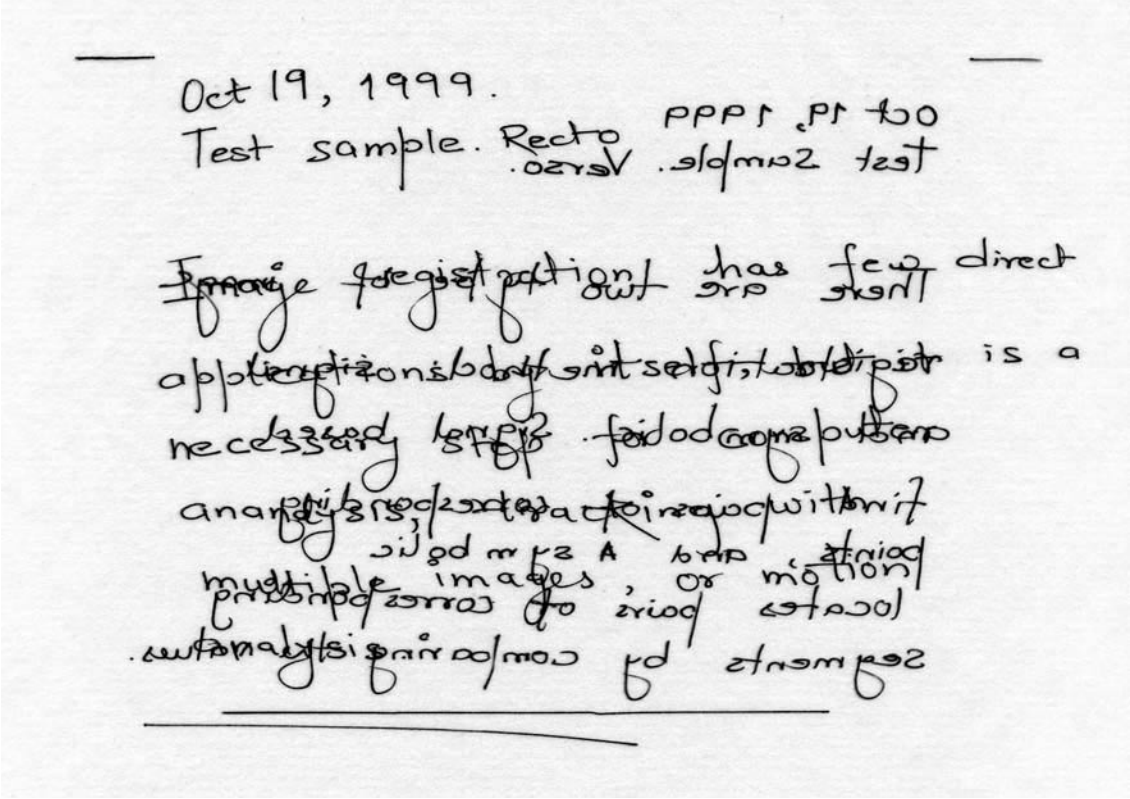


Figure 6: Test image, simulation#1

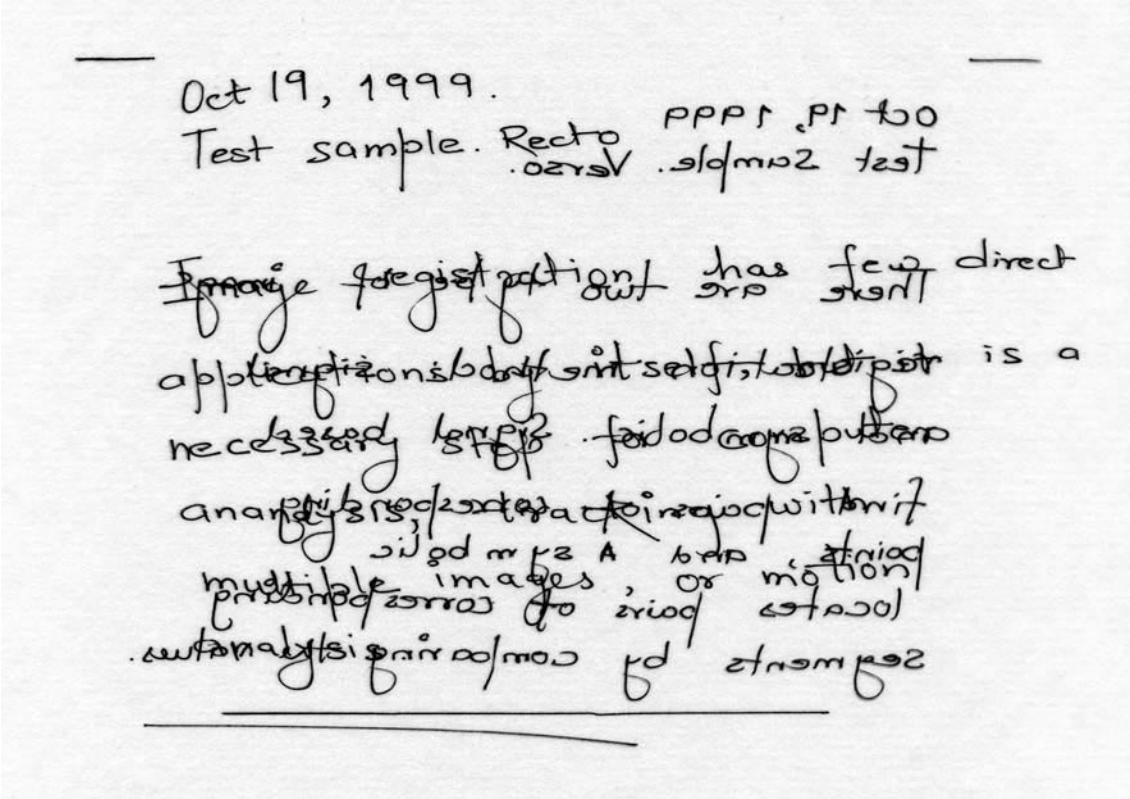


Figure 7: Test image, simulation#2

2)Page2 image (See annex for original recto and verso sides of this image)

Dimension: 1656x1134

Resolution: 96dpi

Pixel technique (Size of K = 15):

Time to execute registration process:	23.6 seconds
Time to for registration process + display routine:	164.1 seconds
Size of S:	5621
Difference:	48.767
#function evaluations:	207

Combined technique (Size of K = 15, Size of Block = 10):

Time to execute registration process:	10.5 seconds
Time to for registration process + display routine:	130.5 seconds
Size of S:	1152
Difference:	10.347
#function evaluations:	299

3)Page1 image (See annex for original recto and verso sides of this image)

Dimension: 1303x2090

Resolution: 96dpi

Pixel technique (Size of K = 27):

Time to execute registration process:	17.14 seconds
Time to for registration process + display routine:	402.4 seconds
Size of S:	5624
Difference:	68.88 (not good)
#function evaluations:	80

Combined technique (Size of K = 27, Size of Block = 10):

Time to execute registration process:	11.8 seconds
Time to for registration process + display routine:	389.6 seconds
Size of S:	1155
Difference:	15.635 (not good)
#function evaluations:	266

Conclusion

A solution to the removal of *bleed through* was presented in this report. It was also shown how the registration process was implemented in *Matlab*. An analysis of results obtained through simulations were also given for this registration process. It should be reiterated that the registration process is an important one for the whole *bleed through* removal algorithm. Without it, the restoration process could not be applied.

The registration process was implemented successfully for many types of images. For bigger images, the alignment is sometimes not perfect tough. The performance of the algorithm was greatly increased by performing an estimation of the results using a sample of pixels only. It was found that by choosing K = ceiling(LENGTH / 80) for both dimensions of the image gave the best results, performance and alignment wise. As for the size of blocks, a single size of 25x25 was chosen as the one giving the best results although it is not obvious still if this size is adequate for all types of images.
In the end, the time required to execute the algorithm on a 200x275 was reduced by a factor of 25 by using both techniques.

Although perfect alignment results were not obtained for all types and sizes of images (big images in particular), they were at least obtained in very reasonable time. Hopefully, future work will tune this algorithm to obtain both good alignment and performance, regardless of the image properties. The addition of the restoration process will then be possible and the complete algorithm should prove extremely useful in correcting documents with *bleed through.*The algorithm should open the door to new ideas as well.

Annex I : Source Code

%Main Algorihtm

```
format long g
format compact

global S;
global B;
global I;
global N;
global flipped_fv;

%User enters paths for fr & fv (fv is flipped immediately)
%If image is of type truecolor, it is transformed into type grayscale
fr_path = input('Enter path for fr (image recto): ','s');
fr_info = imfinfo(fr_path);
fr = im2double(imread(fr_path));
if (strcmp(fr_info.ColorType, 'truecolor') | strcmp(fr_info.ColorType, 'RGB')) fr = rgb2gray(fr); end

fv_path = input('Enter path for fv (image verso): ','s');
fv_info = imfinfo(fv_path);
flipped_fv = im2double(imread(fv_path));
if (strcmp(fv_info.ColorType, 'truecolor') | strcmp(fr_info.ColorType, 'RGB')) flipped_fv = rgb2gray(flipped_fv); end
flipped_fv = fliplr(flipped_fv);

clear fv_path; clear fr_path;

%CONSTANTS
K = ceil((fv_info.Width)/80);    %use every Kth pixel horizontally
J = ceil((fv_info.Height)/80);   %use every Jth pixel vertically
SIZE = 10;              %size of block in both H&V direction, also, distance between each block
BOR = 25;               %width of border

fprintf('\nK = %d', K);
fprintf('\nJ = %d\n\n', J);

%User chooses method to calculate _P_arameters
method_P = input('Use which method to calculate parameters 1-nil 2-fminu: ');

%User chooses method to reduce size of _S_
method_S = input('Use which method to reduce size of "S" 1-Pixel Samples 2-Pixel Blocks 3-Both: ');

%Initilization of parameters
t13 = 0; t23 = 0;
t11 = 1; t12 = 0;
t21 = 0; t22 = 1;

%start timer (we want to calculate the time the program takes to execute tasks.)
time = cputime;

%Define the set of pixels "S" used to approximate the results.
%------------------------------------------------------------
[m,n] = size(flipped_fv);

%If recto & verso have inequal sizes, they are made equal
delta_n = size(fr, 2) - n;
delta_m = size(fr, 1) - m;
if (delta_n > 0) fr(:, 1:delta_n) = []; end
if (delta_n < 0) flipped_fv(:, 1:abs(delta_n)) = []; end
if (delta_m > 0) fr(1:delta_m, :) = []; end
```

```
if (delta_m < 0) flipped_fv(1:abs(delta_m), :) = []; end

[m,n] = size(flipped_fv);

%initialize S vectors. Note that the pixels outside the border are not used.
SX = [1+BOR:n-BOR];
SY = [1+BOR:m-BOR];

%------------------------------
%Use a method to make S smaller
%------------------------------

%1)PIXEL SAMPLES:Use only every Kth pixel in the image
if ((method_S == 1) | (method_S == 3))
   SX = SX(1:K:length(SX));
            SY = SY(1:J:length(SY));
end

%2)PIXEL BLOCKS METHOD: Cut the image in blocks and use only half of them, that is,
%use blocks of "SIZE" pixels that are "SIZE" pixels appart from each other

if ((method_S == 2) | (method_S == 3))

   %first, add padding so that the vector's size is divisible by SIZE
            if (mod(length(SX),SIZE) ~= 0)
      npad = SIZE - mod(length(SX),SIZE);
      SX = [SX, zeros(1, npad)];
    else
      npad = 0;
    end
            if (mod(length(SY),SIZE) ~= 0)
      mpad = SIZE - mod(length(SY), SIZE);
      SY = [SY, zeros(1, mpad)];
            else
      mpad = 0;
            end
            %Now delete unwanted blocks
            %1- We put all the image blocks in a single matrix, that is, the blocks are transformed into single lines.
            %2- By deleting lines, we actually delete blocks

      %delete a block every two lines (lines that represent blocks)
                        SX = reshape(SX, SIZE, length(SX)/SIZE)';
                        SX(2:2:size(SX,1), :) = [];
                        SY = reshape(SY, SIZE, length(SY)/SIZE)';
                        SY(2:2:size(SY,1), :) = [];
                        %reshape as a 1-dim vector
                        %SX = SX'; SX = SX(:)';
                        %SY = SY'; SY = SY(:)';
                        SX = reshape(SX', 1, size(SX,1)*size(SX,2));
                        SY = reshape(SY', 1, size(SY,1)*size(SY,2));
                        %remove the padding
                        SX((length(SX)-npad):length(SX)) = [];
                        SY((length(SY)-mpad):length(SY)) = [];

                        clear npad;
                        clear mpad;
end

%Create S using SX and SY calculated using method 1,2 or both
[SY,SX] = meshgrid(SY, SX);
SX = SX(:)';
SY = SY(:)';
```

```
S = [SX; SY];
[M,N] = size(S);

%Obtain recto values, members of S
B = interp2(fr, SX, SY, '*linear');

clear SX;
clear SY;

%Create I, used in every interpolation regardeless of method chosen
I = ones(1, N);

%Initialize DIFF, the difference we are trying to minimize
DIFF = inf;

%Initialize RM, the rotation matrix
RM = [t11, t12; t21, t22];

%--------------------------------------------------------------------
%Use method selected by user to calculate parameters and difference
%--------------------------------------------------------------------
%1)NIL METHOD: This method does nothing to correct verso and should only be chosen
%in order to observe visually what happens when parameters are set to default.

%TEST! the two following lines should be deleted after test
%manual input of parameters
t13 = 5; t23 = -1.5; t11 = 1; t12 = 0; t21 = 0; t22 = 1;

if (method_P == 1)
        TM = repmat([t13; t23], 1, N);

        XIYI = RM * S + TM;
        XI = XIYI(1, :);
        YI = XIYI(2, :);
        clear XIYI;

        ZI = interp2(flipped_fv, XI, YI, '*linear');

   DIFF = (B-ZI).*(B-ZI)*I'

   clear ZI; clear XI; clear YI;

end

%2)FMINU METHOD: This method calculates the best parameters by solving an optimization problem
%using the fminu function. An estimate of parameters t13 & t23 must be done first.
if (method_P == 2)

   %The two following loops take samples horizontally and vertically to find an aera near where
   %the global minimum is. The cost is not negligible but failing to do this will result in the
   %optimization function (fminu) to find a LOCAL minimum most of the time.
   for i = -24:0.5:24

      TM = repmat([i; t23], 1, N);

      XIYI = RM * S + TM;
      XI = XIYI(1, :);
                  YI = XIYI(2, :);

      clear XIYI; clear TM;

            ZI = interp2(flipped_fv, XI, YI, '*linear');
```

```
    MINDIFF = (B-ZI).*(B-ZI)*I';

    clear ZI; clear XI; clear YI;

    if (MINDIFF < DIFF)
       DIFF = MINDIFF;
       t13 = i;
    end
  end

  fprintf('\nESTIMATES\n---------\n');
  fprintf('Estimate of t13 = %2.3f\n', t13);

  for i = -24:0.5:24

    TM = repmat([t13; i], 1, N);

    XIYI = RM * S + TM;
    XI = XIYI(1, :);
            YI = XIYI(2, :);

    clear XIYI; clear TM;

            ZI = interp2(flipped_fv, XI, YI, '*linear');

    MINDIFF = (B-ZI).*(B-ZI)*I';

    clear ZI; clear XI; clear YI;

    if (MINDIFF < DIFF)
       DIFF = MINDIFF;
       t23 = i;
    end
  end

  fprintf('Estimate of t23 = %2.3f\n\n', t23);

  clear MINDIFF; clear i;

  %initialize vector of _parameters_ for fminu
  parameters = [t13, t23, t11, t12, t21, t22];

  %Here the vector of _options_ for fminu is initialized.
  my_options = zeros(1, 18);      %fminu options vector
  my_options(1) = 1;              %Display option
  my_options(2) = 0.001;          %Termination tolerance for parameters
  my_options(3) = 0.01;           %Termination tolerance for function value (DIFF)
  %my_options(14) = 1000;         %MaxFunEvals option
  my_options(18) = 0.25;          %Step length
  options = foptions(my_options);

  [minparameters, options] = fminu('findDiff6par', parameters, options);
  t13 = minparameters(1);
  t23 = minparameters(2);
  t11 = minparameters(3);
  t12 = minparameters(4);
  t21 = minparameters(5);
  t22 = minparameters(6);
  DIFF = options(8);

end
```

```
%Output of some results in Matlab window
fprintf('\nPARAMETERS\n----------\n');
fprintf('t13 = %2.5f\nt23 = %2.5f\n', t13, t23);
fprintf('[t11, t12  =  [%2.5f, %2.5f\n t21, t22]     %2.5f, %2.5f]\n', t11, t12, t21, t22);
fprintf('\nDIFFERENCE\n----------\n');
fprintf('DIFF = %4.3f\n', DIFF);
PartialTime = cputime - time;
fprintf('\nTime to execute algorithm: %4.3f\n', PartialTime);

clear S; clear minparameters; clear parameters; clear my_options; clear method_P;
clear B; clear I; clear RM; clear PartialTime;

%-------------------------------------------------------------------------------------------
%DISPLAY
%Here we redo everything using ALL pixels (S = L) to obtain visual results
%Using the previously found parameters, we can interpolate the new _complete_ verso
%-------------------------------------------------------------------------------------------

fprintf('\ncreating S (S = L)');

%here, S = L (minus the borders)
S2Y = [1+BOR:(m-BOR)];
S2X = [1+BOR:(n-BOR)];
m = length(S2Y);
n = length(S2X);
mn = m*n;

%create S
[S2Y,S2X] = meshgrid(S2Y, S2X);
S2X = S2X(:)';
S2Y = S2Y(:)';
S2 = [S2X; S2Y];

clear S2X;
clear S2Y;

fprintf('\ncreating TM2, RM2, XI2 and YI2');

TM2 = repmat([t13; t23], 1, mn);
RM2 = [t11, t12; t21, t22];
XIYI2 = RM2 * S2 + TM2;
XI2 = XIYI2(1, :);
YI2 = XIYI2(2, :);

clear XIYI2; clear S2; clear RM2; clear TM2; clear mn;
clear t11; clear t12; clear t13; clear t21; clear t22; clear t23;

fprintf('\nStarting Interp2');

%obtain new transformed verso by interpolation
ZI2 = interp2(flipped_fv, XI2, YI2, '*linear');
flipped_fv = (reshape(ZI2, n, m))';

clear XI2;
clear YI2;
clear ZI2;

fprintf('\nInterp2 Complete');

%adjust recto's size to match new verso (remove border)
fr(:, 1:BOR) = [];
```

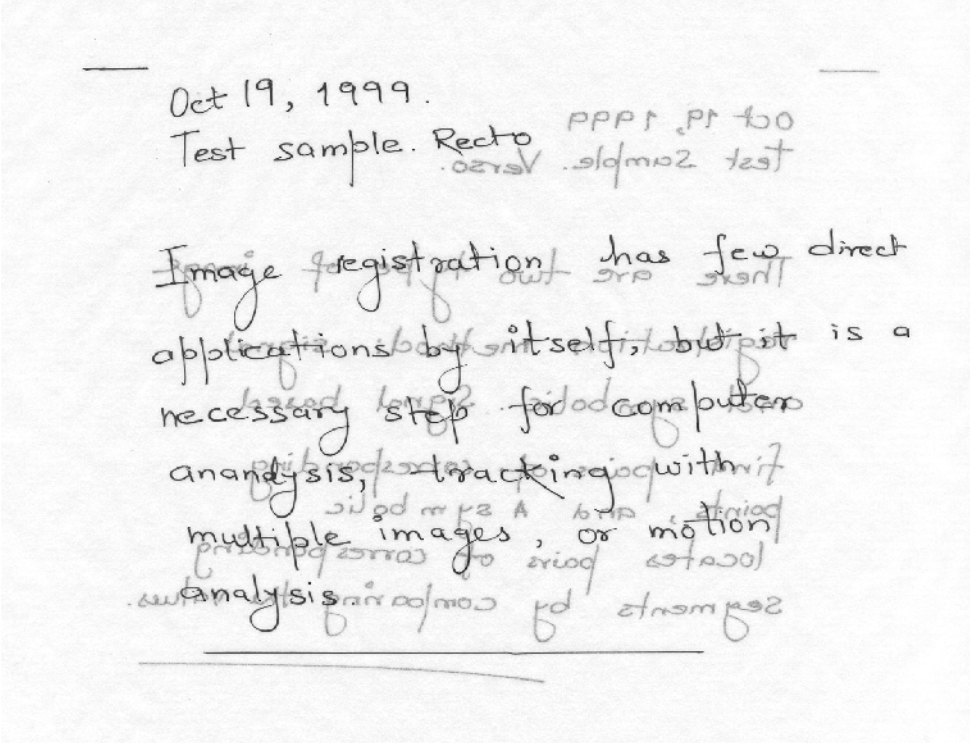
```
fr(:, (size(fr, 2) - BOR + 1):size(fr,2)) = [];
fr(1:BOR, :) = [];
fr((size(fr, 1) - BOR + 1):size(fr,1), :) = [];

fprintf('\nSize of recto adjusted');

clear delta_n; clear delta_m; clear n; clear m; clear BOR;
clear border1; clear border2; clear border3; clear border4;

%Here we transform [0,1] in [1,0]. We then ADD or SUBstract the recto & verso
%--------------------------------------------------------------------------
r_fr = abs(-1.+fr);   %same as imadjust(fr, [0 1], [1 0]) but much much quicker
r_flipped_fv = abs(-1.+flipped_fv);
ADD = r_fr + r_flipped_fv;              %[0,2]
%Two methods to bring the [0,2] values back to [0,1] (USE ONLY ONE)
        %Method 1: Use the average (official method; should always be used for true results)
        %ADD = 0.5 .* ADD;
   %ADD = abs(-1.+ADD);
        %Method 2: Cut out-of-range values (use only to obtain more pronounced visual results)
        ADD = (ADD > 1) + (ADD .* (ADD < 1));
        ADD = abs(-1.+ADD);
%Substract
% SUB = r_fr - r_flipped_fv;
% SUB = abs(-1.+SUB);

clear r_fr;
clear r_flipped_fv;

SizeOfS = N
TotalTime = cputime - time;
fprintf('\n\nTime to execute algorithm + display routine: %4.3f\n', TotalTime);

clear time;
clear N; clear M;

fprintf('\nFinished, saving images....\n\n');

%Show result images directly on screen
%figure, imshow(fr);
%figure, imshow(flipped_fv);
%figure, imshow(ADD);
%figure, imshow(SUB);

%Save result images (recto, verso & ADD) under same format as source images.
ADD_fname = strcat('ADD.', fv_info.Format);
recto_fname = strcat('recto.', fr_info.Format);
verso_fname = strcat('verso.', fv_info.Format);

if (strcmp(fr_info.Format, 'tif') | strcmp(fr_info.Format, 'tiff'))
  imwrite(ADD, ADD_fname, fv_info.Format, 'Resolution', [fr_info.XResolution, fr_info.YResolution]);
  imwrite(fr, recto_fname, fr_info.Format, 'Resolution', [fr_info.XResolution, fr_info.YResolution]);
  imwrite(flipped_fv, verso_fname, fv_info.Format, 'Resolution', [fr_info.XResolution, fr_info.YResolution]);
end
if (strcmp(fr_info.Format, 'jpg') | strcmp(fr_info.Format, 'jpeg'))
  imwrite(ADD, ADD_fname, fv_info.Format, 'Quality', 100);
  imwrite(fr, recto_fname, fr_info.Format, 'Quality', 100);
  imwrite(flipped_fv, verso_fname, fv_info.Format, 'Quality', 100);
else
  %nothing implemented for other file formats yet.
end

clear all;
```

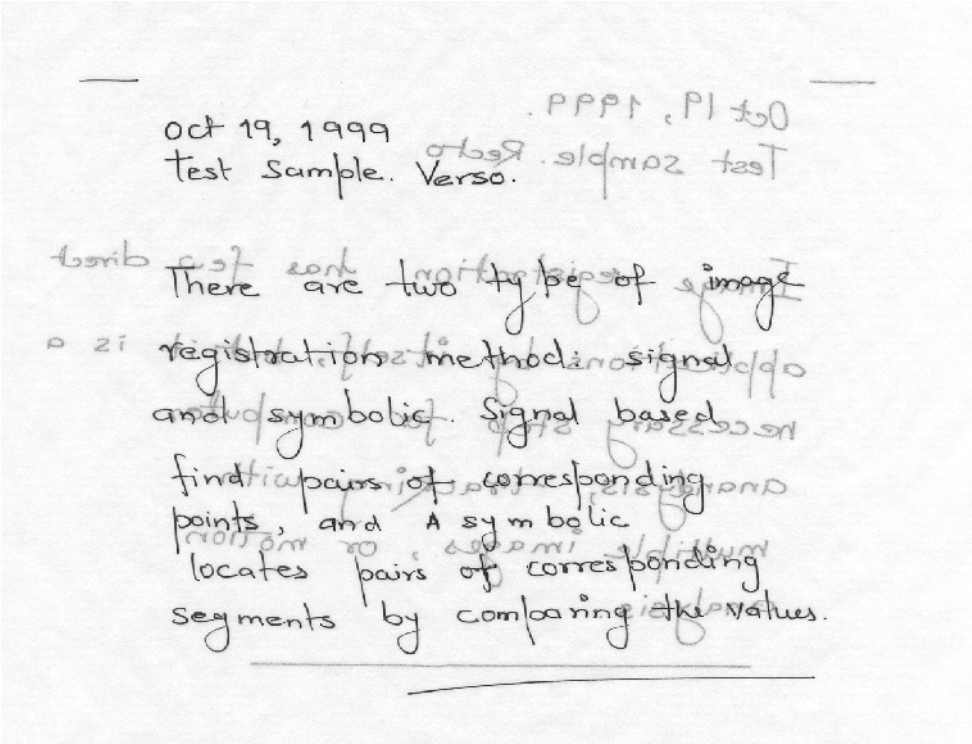
% Difference algorithm

```
function diff = findDiff(parameters)

global S;
global B;
global I;
global N;
global flipped_fv;

t13 = parameters(1);
t23 = parameters(2);
t11 = parameters(3);
t12 = parameters(4);
t21 = parameters(5);
t22 = parameters(6);

RM = [t11, t12; t21, t22];
TM = repmat([t13; t23], 1, N);

XIYI = RM * S + TM;
XI = XIYI(1, :);
YI = XIYI(2, :);

clear XIYI;
clear TM;
clear RM;

ZI = interp2(flipped_fv, XI, YI, '*linear');

diff = (B-ZI).*(B-ZI)*I';

clear ZI;
```

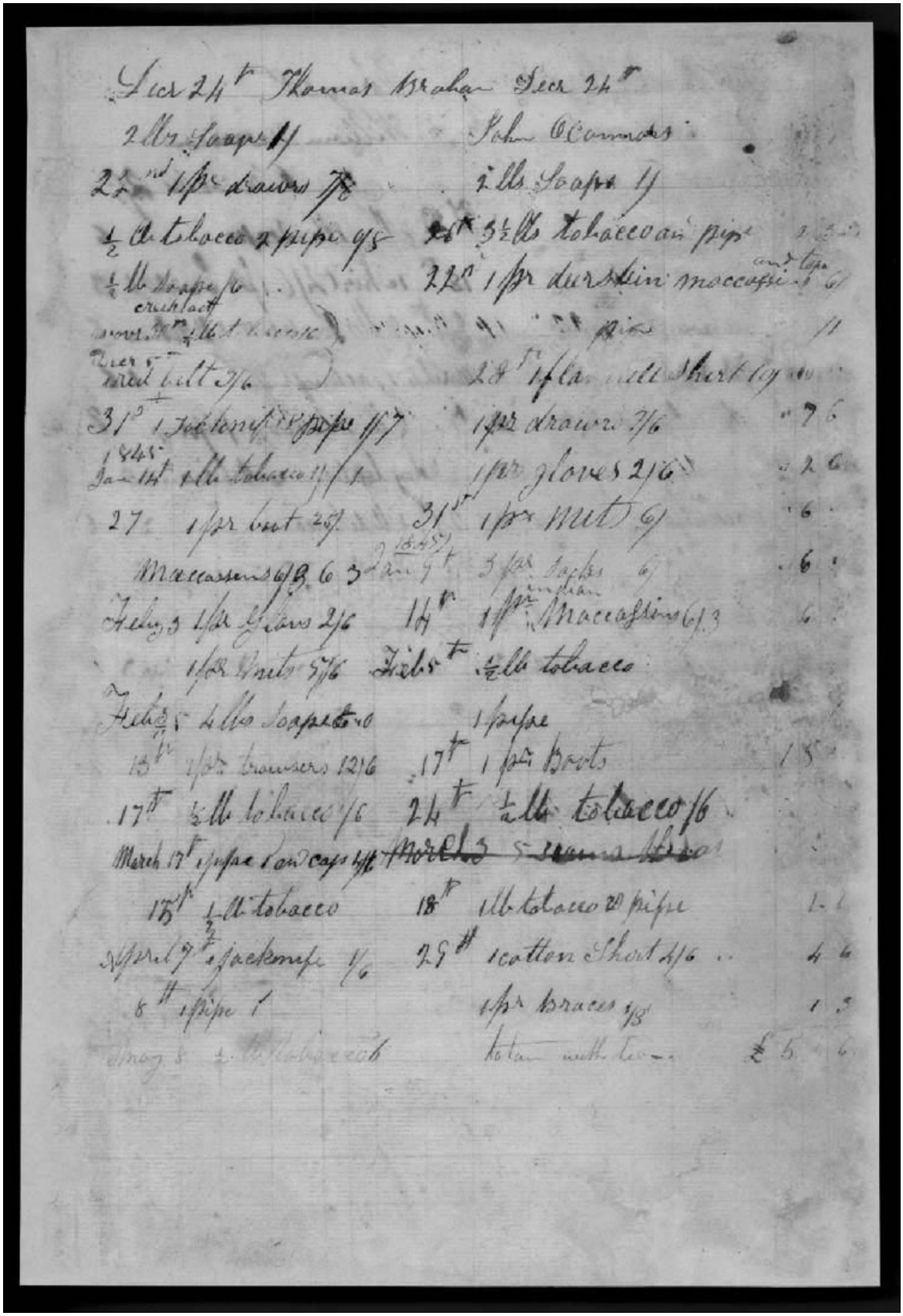
Annex II : Recto and verso of simulations documents

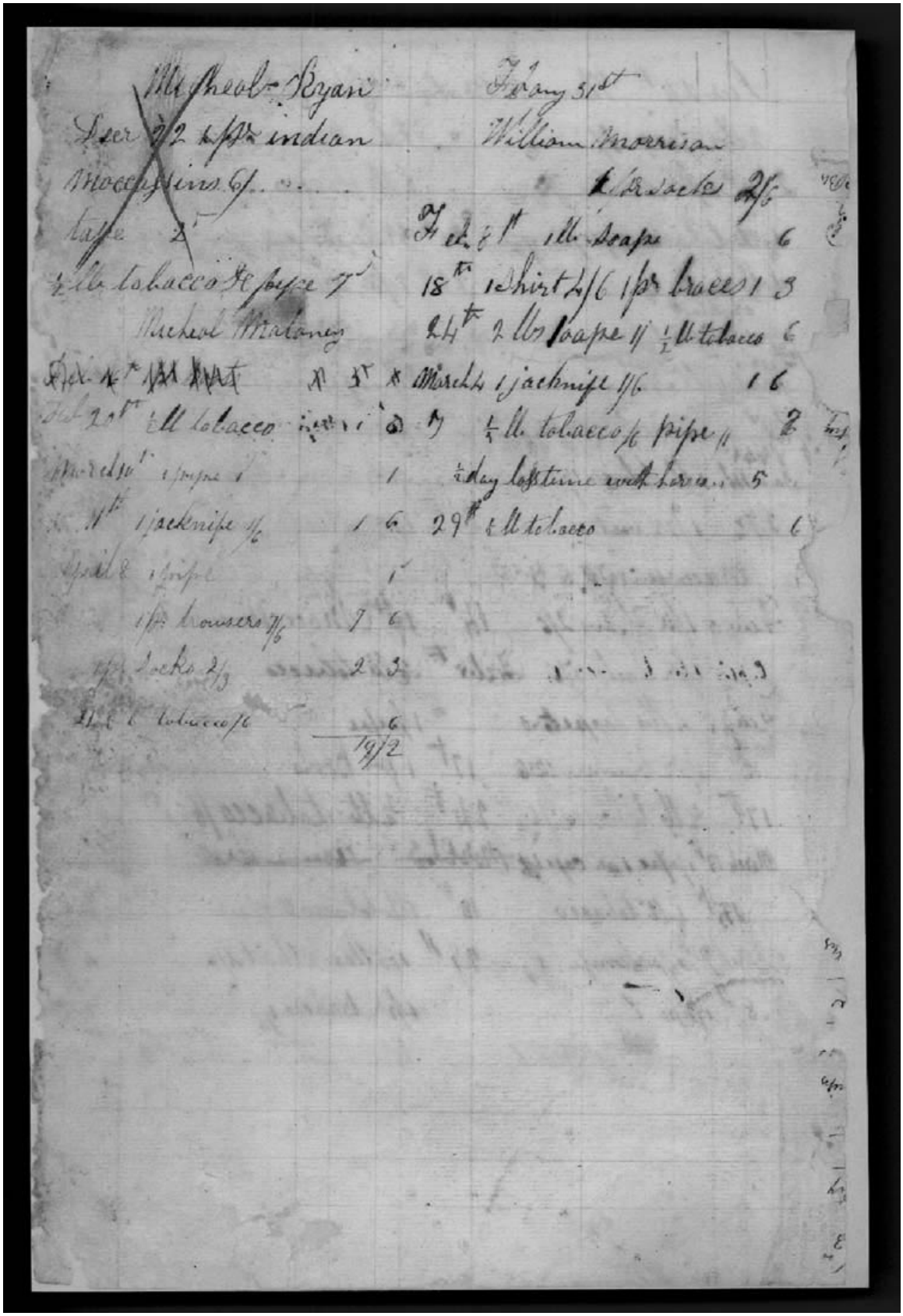
Test image





Page2 image

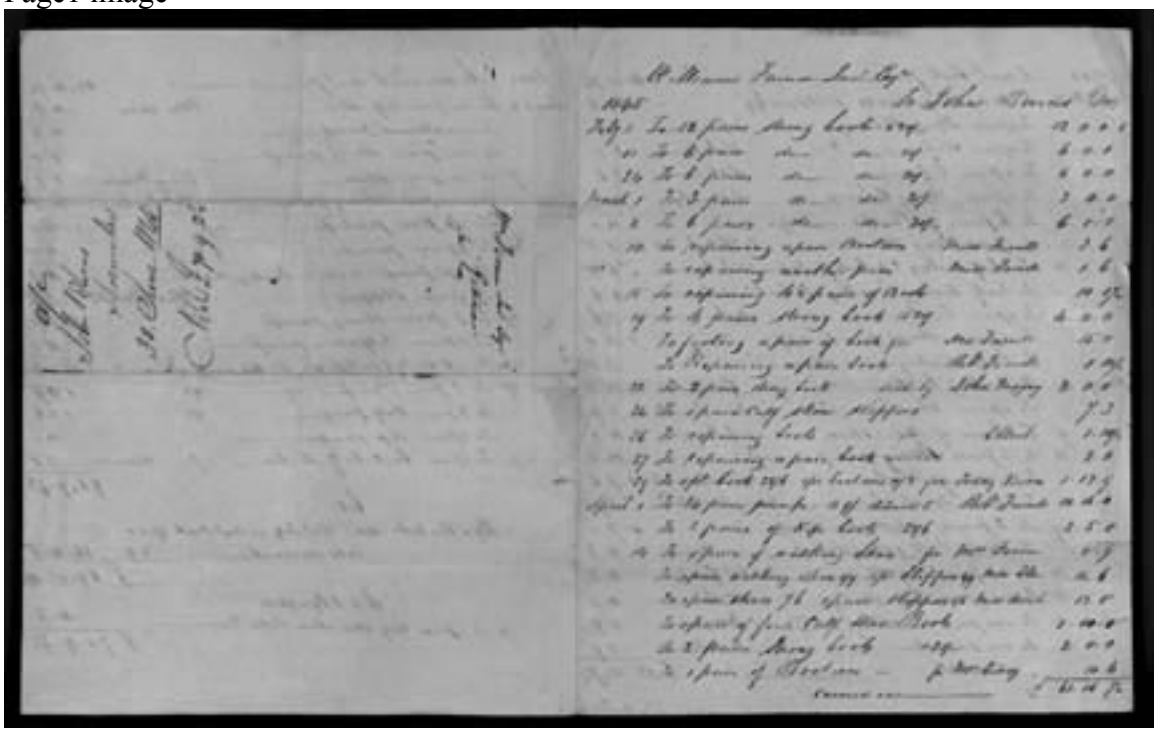




Michael Ryan
Deer 2/2 1/pr indian
Moccasins 6/
Rafe 2
½ lb tobacco ½ pipe 7
Michael Maloney
1 jacknife 1/6
½ lb tobacco
1 pipe 1
1 jacknife 1/6 1 6
Spail 1 pipe 1
1 pr trousers 7/6 7 6
1 pr socks 2/3 2 3
½ lb tobacco 6
19/2

Feby 31st
William Morrison
1 Blanket 2/6
Feb 8th 1 lb soap 6
18th 1 shirt 2/6 1 pr braces 1 3
24th 2 lbs soap 1/ ½ lb tobacco 6
March 1 jacknife 1/6 1 6
½ lb tobacco ½ pipe 1/ 2
½ day lost time work here 5
29th ½ lb tobacco 6

Page1 image



References

DUBOIS, Eric. *"* Reduction of bleed-through in scanned manuscript documents *",* University of Ottawa, Ottawa, 4p.

SHARMA, Gaurav. " Cancellation of show-through in duplex scanning ", in Proc. IEEE Int. Conf. Image Processing, vol. 2, pp. 609-612, Sept. 2000.

ETTER, Dolores. C. KUNCICKY, David. *Introduction to Matlab*, Prentice Hall's , 1999.