

Implementation Notes for the MRSAR Features in the MATLAB Environment

Roland Kwitt
Department of Computer Sciences
University of Salzburg
E-Mail: rk Witt@gmx.at
Source: <http://www.wavelab.at/sources>

July 23, 2009

1 Notation

We introduce some notational conventions first: all letters written in typewritten font, e.g. **A**, **b**, etc., denote MATLAB variables. Small boldface letters such as **x** denote vectors, big boldface letters denote matrices, e.g. **Σ** .

2 Computing MRSAR Features

For the implementation of the Multiresolution Simultaneous Autoregressive (MRSAR) features, we strictly follow the original approach presented in [1]. We do not implement the rotation-invariant extension. We require that the image is represented in the pixel domain and we operate on the luminance channel only. Two important issues for the implementation of the MRSAR model are the definition of **pixel neighborhood** and the definition of **multiresolution**. In the original paper, multiresolution is not accomplished by the Gaussian pyramid or some other transform but by changing the neighborhood size. Figure 1 shows the neighborhood definition, where $l = 1$ denotes the commonly known 8×8 neighborhood and $l = 2$ denotes the second multiresolution level. Neighborhoods for $l > 2$ are defined accordingly. Symmetries in the neighborhood are indicated by pixel of the same color. The centering pixel is marked black.

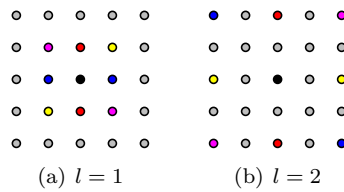


Figure 1: Neighborhood definition according to the multiresolution property. The symmetries are indicated by pixel of the the same color.

In the MRSAR model, the centering pixel with gray value r_{ij} at position i, j is approximated by the following linear combination

$$\begin{aligned} r_{ij} = & c_1(r_{i-l,j} + r_{i+l,j}) \\ & + c_2(r_{i,j-l} + r_{i,j+l}) \\ & + c_3(r_{i-l,j-l} + r_{i+l,j+l}) \\ & + c_4(r_{i-l,j+l} + r_{i+l,j-l}) \end{aligned} \quad (1)$$

where the c_i are the coefficients of the MRSAR model. The task is to estimate these coefficients. Since the solution to Eq. 1 is under-determined by considering only the neighboring pixel alone, the c_i are estimated over a sliding window of size $N \times N$ (originally $N = 21$) in the following way: by taking care of the block borders we first construct a $N - 2l \times 8$ matrix \mathbf{X} by arranging the 8 neighbors of each valid pixel as a row vector. Further, the gray values of each valid pixel are arranged in a $N - 2l \times 1$ vector \mathbf{y} . As an intermediate step, we subtract the mean value of each $N \times N$ block from each pixel, since the original MRSAR model is given as

$$r_s = \mu + \sum_{r \in D_l} c_s r_{s+r} + \epsilon_s, \quad (2)$$

where $s \in \mathcal{Z}^2$ denotes the index tupel of the valid pixel and D denotes the set of neighborhood index tupels for multiresolution level l (e.g. $D_l = \{(-l, 0), (l, 0), \dots, (-l, l), (l, -l)\}$). The construction of \mathbf{X} and \mathbf{y} are accomplished in the MATLAB function `[X,y] = sar(patch,d)`, where `patch` denotes a $N \times N$ block and `d` denotes the multiresolution level l .

```
function [x,y] = sar(patch,d)
    m2 = mean2(patch);
    patch = double(patch) - m2;
    cnt = 1;
    for i=(d+1):size(patch,1)-d
        for j=(d+1):size(patch,2)-d
            x(cnt,:) = [...
                patch(i-d,j), ...
                patch(i+d,j), ...
                patch(i,j-d), ...
                patch(i,j+d), ...
                patch(i-d,j-d), ...
                patch(i+d,j+d), ...
                patch(i-d,j+d), ...
                patch(i+d,j-d)];
            y(cnt) = patch(i,j);
            cnt = cnt + 1;
        end
    end
    y = y';
end
```

\mathbf{X} and \mathbf{y} are then used to estimate the coefficients c_1, c_2, c_3, c_4 in a least-squares sense. This is done in the routine `[S,G,STG,a] = sar_solve(X,y)` which solves estimation problem according to the explanations given by Petrou in [2]. We use the MATLAB function `pinv` to compute the Moore–Penrose pseudoinverse.

```

c = 1;
for i=1:2:size(x,2)-1
S(:,c) = x(:,i) + x(:,i+1);
    c = c + 1;
end
a = pinv(S)*y;

```

The variable **a** then contains the coefficients c_1, \dots, c_4 . Since we are now able to estimate the coefficients, we can slide the $N \times N$ window over the whole image with a pixel increment of d in both directions. For each window position, we run **sar_solve** and estimate the coefficients. Further, we determine the standard deviation of the estimation error ($\epsilon(s)$ in Eq. (2)). The sliding window and calculation of the empirical standard deviation is implemented in the routine **sarslide** which takes as arguments the image **I**, the blocksize N as **blksize**, the pixel increment d as **shift** and an array **levels** of multiresolution levels (e.g. **levels** = [2 3 4]).

```

function data = sarslide(I,blksize,shift,levels)
    szx = size(I,1);
    szy = size(I,2);
    data = [];
    for level=levels
        data_l = [];
        cnt = 1;
        for i=1:shift:szx-blksize+1
            for j=1:shift:szy-blksize+1
                blk = I(i:i+blksize-1,j:j+blksize-1);
                [x,y] = sar(blk,level);
                [S,G,STG,a] = sar_solve(x,y);
                s = std(testfun(a',x) - double(y));
                data_l(cnt,:) = [a' s];
                cnt = cnt + 1;
            end
        end
        data = [data data_l];
    end
end

```

We use the helper function **F = testfun(x,xdata)** to compute the estimates of the center pixel grayscale values and can thus determine the estimation error.

```

function F = testfun(x,xdata)
    tmp = x(1)*xdata(:,1) + x(1)*xdata(:,2) + ...
        x(2)*xdata(:,3) + x(2)*xdata(:,4) + ...
        x(3)*xdata(:,5) + x(3)*xdata(:,6) + ...
        x(4)*xdata(:,7) + x(4)*xdata(:,8);
    F = sum(tmp,2);
end

```

3 A Brief Example

In this example, we explain how to obtain the MRSAR parameters from an image using the **sarslide** function. We compute the MRSAR parameters for the grayscale image **test.jpg**

using the multiresolution levels 1, 2 and 3. The size of the sliding window is 21×21 and the pixel increment in both directions is 4 pixel.

```
>> I = imread('pictures/test.jpg');
>> I = rgb2gray(I);
>> alpha = sarslide(I,21,4,[1 2 3]);
```

Running `size(alpha)` shows that we have obtained a 729×15 matrix with MRSAR parameters and the standard deviations of the approximation error in each subwindow.

```
>> size(alpha)
ans =
    729     15
```

4 Supplementary Functions

The supplementary MATLAB functions we provide enable the computation of MRSAR features for a set of images and allow to compute a distance matrix where each entry in the matrix denotes the distance between two image models. By default we use a three resolution neighborhood and thus obtain a $K \times 15$ matrix for each image. K denotes the number of $N \times N$ sliding window positions obtained with an increment of d pixel. This matrix contains the fitted MRSAR parameters which we subsequently use for similarity measurement. We implement the originally proposed Mahalanobis distance and the Bhattacharya distance and assume that the rows of the data matrix are realizations of a multivariate Gaussian distribution.

4.1 Data Format of the Images

In order to work with the supplementary functions, we have to store the images in a cell array with the following elements: `image`, `dim` and `name`. `image` stores the image as obtained from the MATLAB command `imread`. The image can either be a RGB color image, or a grayscale image. Accordingly, we have to set the `dim` field to 3 (i.e. in case of color images) or to 1 (i.e. in case of grayscale images). The field `name` specifies the image name (for identification purposes). Here is an example of how to construct the required data structure:

```
>> I = imread('/pictures/test.jpg');
>> data{1}.image = I;
>> data{1}.dim = 3;
>> data{1}.name = 'test.jpg'
```

Note: we have mentioned that we only work on grayscale images. However, the extension to color images is straightforward by simple columnwise concatenation of the MRSAR parameters.

4.2 Extracting Features from all Images

We provide a function `sarfe` which returns a cell array of the MRSAR features, or more specifically the mean and the covariance matrix of the MRSAR parameter matrix (i.e. `alpha` in the examples from above). This function is useful since you do not have to care whether your images are grayscale or color versions. In case of color images, the MRSAR parameters for each color band are simply concatenated (columnwise). Subsequently, the mean and covariance matrix are determined. Here is an exemplary call to `sarfe` enabling verbose output

```
>> fe = sarfe(data,'debug',true);
```

The return value `fe` is a cell array with elements `mu` and `cov`. Of course, the size of the cell array `fe` equals the size of the images in `data`. Having obtained all MRSAR parameters for all images, we can go on to measure image similarities.

4.3 Measuring Image Similarity

For similarity measurement we either use the Mahalanobis distance or the Bhattacharya distance (which is also provided as C source code, discussed later). After we have computed all image models using `sarfe` we can run the MATLAB function `ranksar` which accepts the computed models in `fe` as a parameter. The function also has additional parameters which specify the type of distance function to use. An exemplary call to `ranksar` for using the Bhattacharya distance is

```
>> D = ranksar(fe,'debug',true,'method','bhattacharya');
```

This returns a distance matrix `D` which contains the distances between all models. In the function `ranksar`, the Bhattacharya distance is implemented as

```
B(i,j) = 1/8 * (xi.mu - xj.mu)*inv(xi.cov+xj.cov)* ...
          (xi.mu-xj.mu)' + ...
          1/2 * log( det((xi.cov+xj.cov)/2) / ...
          sqrt(det(xi.cov)*det(xj.cov)));
D(i,j) = sqrt(1-exp(-B(i,j)));
```

where `xi` and `xj` contain models i and j (e.g. `xi = fe{i}`). In the same manner you can also specify `'ml'` as a valid argument instead of `'bhattacharya'`. This computes the Mahalanobis distance between the models, implemented by

```
D(i,j) = log(det(xj.cov)) + ...
          trace(inv(xj.cov)*xi.cov) + mahal(xi.mu,xj.features);
```

4.4 Running C Implementation of the Bhattacharya distance

Since it is very time consuming to calculate the Bhattacharya distance in MATLAB (which hinders large-scale tests), we provide a C implementation of the `ranksar` routine. In order to run this code, we first have to dump the MRSAR models to our harddisk. This is implemented in the routine `dump_sardata` which is called as follows

```
>> dump_sardata(fe,'/tmp/sardata',0);
```

This call dumps the fitted MRSAR models stored in `fe` to the folder `/tmp/sardata` (which has to exist). The third argument 0 (controls the offset) is used to control at which model we start to dump. Hence, a value of 10 for example dumps all models starting from model 10 and so on. Upon successful completion of the command, the folder `/tmp/sardata` then contains the mean vectors and the covariance matrices of the models. Listing the files in this directory gives

```
model0.mu model0.cov model1.mu model1.cov
...
```

Next, we can compile the C source file using the provides Makefile `Makefile.mrsar`.

```
$ make -f Makefile.mrsar
```

This gives a binary `Bhatta-MR-SAR`. Typing `Bhatta-MR-SAR --help` lists all possible arguments. To show an exemplary call, we assume that we have dumped the MRSAR models to `/tmp/mrsar` and that we have used color images. Hence our covariance matrix is 45×45 . We copy the `Bhatta-MR-SAR` binary to the directory `/tmp/mrsar` and then run

```
$. /Bhatta-MR-SAR -K 45 -d dist.bin -v
```

which computes the same distance matrix as obtained by the MATLAB function call `D = ranksar(... 'method','bhattacharya')`. Per default the matrix is stored as `dist.bin` which can be modified by the `-d` switch, though. The distance matrix is written using the GSL routine `gsl_matrix_fwrite`, so have to take care of the format in order to do further processing. An easy way to take a look at the distance matrix is to load it into MATLAB again.

```
>> fid = fopen('/tmp/mrsar/dist.bin','rb');  
>> A = fread(fid,'double');  
>> A = reshape(A , [sqrt(length(A)) sqrt(length(A))]);  
>> A = A';
```

References

- [1] J. Mao and A.K. Jain. Texture classification and segmentation using multiresolution simultaneous autoregressive models. *Pattern Recognition*, 25(2):173–188, 1992.
- [2] M. Petrou and P.G. Sevilla. *Image Processing: Dealing with Texture*. Wiley, 2006.