INDIAN STATISTICAL INSTITUTE

Project Report

Image Deconvolution By Richardson Lucy Algorithm

Authors:
Arijit Dutta
Aurindam Dhar
Kaustav Nandy

Supervisor:
Dr. Deepayan Sarkar

November 29, 2010

Abstract

In our project we have tried to restore an image degraded by presence of noise by applying Richardson Lucy Algorithm. Computer implementation of this algorithm is our main aim. Besides we have shown that this algorithm is nothing but an EM Algorithm. In our project we have considered gray images only.

1 Introduction:

Restoration of digital images from their degraded measurement has always been a problem of great interest. A specific solution to the problem of image restoration is generally determined by the nature of degradation phenomena. So it is highly dependent on the nature of the noise present there. Given the noise function, one can use the Richardson-Lucy Algorithm to restore the degraded image. This algorithm was introduced by W.H. Richardson (1972) and L.B. Lucy (1974).

1.1 What is an Image:

An image is nothing but a huge collection of numbers known as pixels. In particular a gray image is an image in which the value of each pixel is a single sample, that is it carries only intensity information. So a pixel in a given image is just the intensity at that particular point. The pixel value is a number between 0 and 1 (both inclusive). 0 denotes the total absence (i.e. black) and 1 denotes the total presence (i.e. white).

1.2 Point Spread Function:

The *Point Spread Function* describes the response of an imaging system to a point source or point object. Following is an example of a PSF:

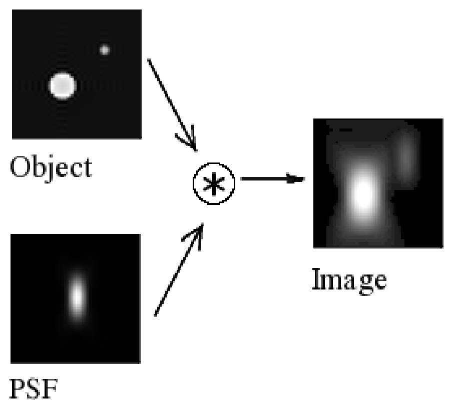


Figure 1: Example of PSF

2 Richardson-Lucy Algorithm:

Suppose,

Y : Degraded Image,

Λ : Original Image,

P : Point Spread Function,

$*$: Operation of Convolution.

Then,

$$Y = \Lambda * P \tag{1}$$

✍ **Comment:** Numerical values of Y, Λ and P can be considered as a measure of frequency at that point.

In the model (1),

$P = \Big(\Big(\, p(i,j) \,\Big)\Big), \quad p(i,j) = \text{P[Photon emitted at } i \text{ is seen at } j]$

$\Lambda = (\lambda_1, \ldots, \lambda_n)' \quad \lambda_i = \text{True pixel value at the } i^{th} \text{ point.}$

$Y = (y_1, \ldots, y_d)' \quad y_j = \text{Observed pixel value at the } j^{th} \text{ point.}$

2.1 Distributions of Observed Pixels:

Notice that y_j is nothing but the count of the photon seen at j. So y_j has a Poisson distribution. In fact,

$$y_j \sim \text{Poisson}(\mu_j)$$

where,

$$\mu_j = \sum_{i=1}^{n} \lambda_i \, p(i,j)$$

2.2 Distribution of Spread Function:

The distribution of spread function may vary from problem to problem. In our problem, we have taken Gaussian spread function which is given by:

$$p(i,j) = \exp\left(- \frac{d(i,j)^2}{\sigma^2} \right)$$

where, $d(i,j) = $ Distance between i and j
In our problem, we have considered standard Eucleadian norm.

2.3 Description of the Algorithm:

Define, the contribution of λ_i on y_j as

$$z(i,j) \sim \text{Poisson}\Big(\lambda_i \, p(i,j)\Big)$$

Then,

$$y_j = \sum_{i=1}^{n} z(i,j)$$

and

$$\frac{\lambda_i \, p(i,j)}{\sum_k \lambda_k \, p(k,j)}$$

is the proportion of y_j emitted by i.
If we know Λ then $z(i,j)$ is estimated by:

$$\hat{z}(i,j) = \frac{y_j \, \lambda_i \, p(i,j)}{\sum_k \lambda_k \, p(k,j)}$$

Given $\hat{z}(i,j)$, λ_i is estimated by:

$$\hat{\lambda_i} = \sum_{j=1}^{d} \hat{z}(i,j)$$

So, ultimately it gives an iterative procedure:

$$\lambda_i^{(t+1)} = \lambda_i^{(t)} \sum_{j=1}^{d} \frac{y_j \, p(i,j)}{\sum_k \lambda_k^{(t)} p(k,j)} \tag{2}$$

3 EM Algorithm and Richardson-Lucy Algorithm:'

Here, $z(i,j)$'s are complete data and y_j's are random. So,

$$z(i,j) \mid y_j \sim \text{Bin}\Big(y_j, p_*(i,j)\Big),$$

where,

$$p_*(i,j) = \frac{\lambda_i \, p(i,j)}{\sum_k \lambda_k \, p(k,j)}$$

$p(i,j)$'s are given to us. Our aim is to estimate λ_i's.
By E-M algorithm, first we will calculate

$$\begin{aligned}
&\arg\max_{\lambda} E\Big[\log f_{z(i,j)}(\underset{\sim}{\lambda}) \mid y_j, \underset{\sim}{\lambda}^0\Big] \\
&= \arg\max_{\lambda} E\Bigg[\Bigg\{\log\left(\frac{y_i}{z(i,j)}\right) + z(i,j)\log p_*^0(i,j) \\
&\quad + (y_i - z(i,j))\log\left(1 - p_*^0(i,j)\right)\Bigg\} \Bigg| \, y_j, \underset{\sim}{\lambda}^0\Bigg]
\end{aligned}$$

where $\underset{\sim}{\lambda}^0$ is some initial estimate of $\underset{\sim}{\lambda}$.

4 Computer Implementation:

Suppose, we are given a square image of size $M \times M$. Then there is a total of M^2 pixels. For each of the pixels, we have to apply the algorithm. To compute the denominator of (2), we have to run a loop over all M^2 pixels. This denominator is to be calculated for each of the M^2 terms in the outer most sum of (2). So, for a single iteration step, complexity will be $M^2 \times M^2 \times M^2 = M^6$. Now, even a small image is of 256×256 or 512×512. So, first we have to reduce the complexity.

4.1 Reducing Complexity:

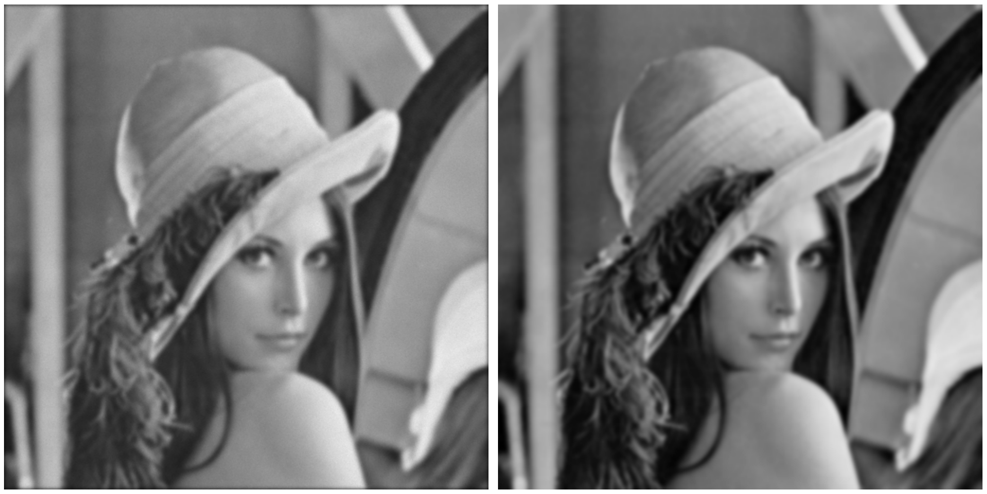
- First, notice that photon emitted from a particular point affects the nearby points most.
- In fact, as the distance between i and j increases, $p(i, j)$ tends to 0.
- So, for a fixed i, we should run the loop over only the range of j for which $p(i, j) > 0$.
- This will reduce the complexity significantly.

✍ **Comment:** We have implemented this algorithm in C. Also we have written a program to build a shared object so that we can directly access it from R.

5 Outputs:

Following are the outputs that we have got after applying this algorithm.

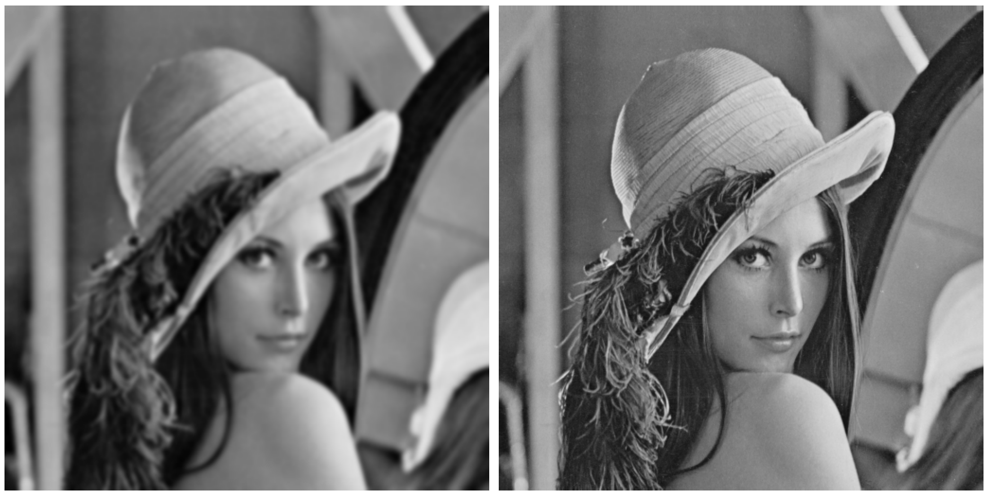
5.1 Output 1:



The picture in the left is blurred using a Gaussian noise with variance 10. It is restored by using the Richardson Lucy Algorithm(Right). **Comments:**

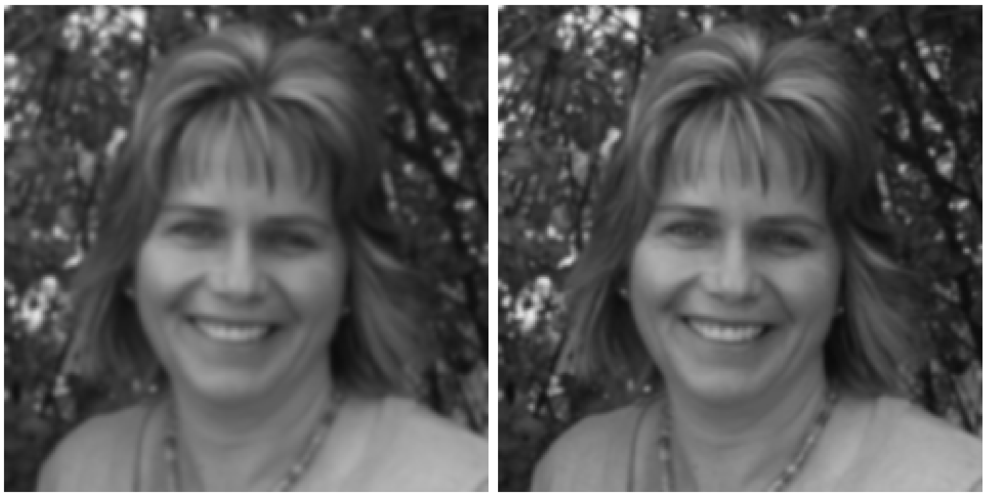
1. This is a 512×512 image.
2. The iteration is done 500 times.
3. It is to be noticed that the PSF vanishes if the distance is more than 11. But even if we run our code considering the points that are in a distance of less than 11, still the complexity is too high. So we have assumed the PSF to be negligible for distance more than 7.
4. Still our program takes more than 2 hours to run.

5.2 Comparison:



The picture in the left is the restored picture, while the picture in the right is the orignal one.
Comment: The restoration is mediocre.

5.3 Output 2:

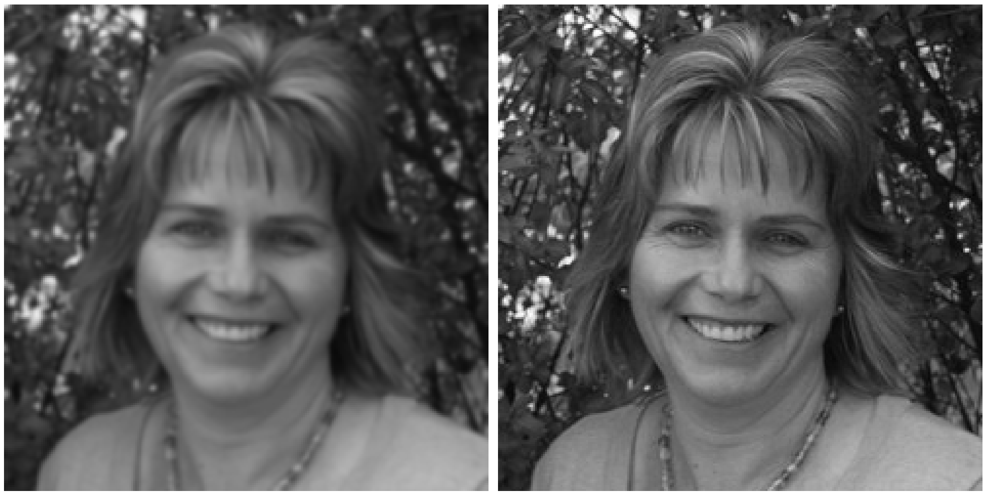


The picture in the left is the blurred one, blurred using Gaussian noise with standard deviation 6. It is restored by RLA(Picture in the right).

Comments:

1. This is a 249×249 image.
2. The iteration is done 500 times.
3. The PSF is considered to be zero for distance more than 6.

5.4 Comparison:



The picture in the left is the restored picture, while the picture in the right is the orignal one.

Comment: The restoration is mediocre.

6 Conclusions:

Even after considering several tricks to reduce complexity, it remains quiet high. The program takes a lot of time to run. Even after that the restoration is not outstanding. We are yet to find a solution of that. Still it is clear that the algorithm gives some sort of improvement in the blurred image.

7 Computer Codes:

7.1 Extracting Pixel Value(R code):

```
library(pixmap)
lena <- read.pnm("oldlennablur.pgm")
write.table(lena@grey,"mylenna", quote=FALSE, row.names = FALSE, col.names= FALSE)
plot(pixmapGrey(as.matrix(read.table("mylenn"))))
```

7.2 Richardon Lucy Algorithm:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define H 6
#define PIX 248
#define NUM 248 * 248
#define SIGMA 25

#ifndef max
  #define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
#endif

#ifndef min
  #define min( a, b ) ( ((a) < (b)) ? (a) : (b) )
#endif

void dataread(float *y);
int distance(int i, int j, int k, int l);
void lambda(float *y, float *lold, float *lnew, float *p);
void renew(float *lold,float *lnew);
void spread(float *p);
void datasave(float *l);

int main(void){

  int i;
  float *y, *lold, *lnew, *p, ttt;
  y = malloc(NUM * sizeof(float));
  lold = malloc(NUM * sizeof(float));
  lnew = malloc(NUM * sizeof(float));
  p = malloc((2 * (H + 1) * (H + 1)) * sizeof(float));

  spread(p);
  dataread(y);
  dataread(lold);

  for(i = 0; i < 40; i++){
    lambda(y, lold, lnew, p);
    renew(lold, lnew);
    printf("%d  \n",i);
  }


  datasave(lnew);

  free(p);
  free(y);
  free(lnew);
  free(lold);

  return 0;
}


/*READING THE DATA*/

void dataread(float *y){
  int i;
  FILE *fp;
  fp = fopen("mylenna","r");
  for(i = 0; i < NUM; i++){
    fscanf(fp,"%f ",&y[i]);
  }
  return;
}

/*DEFINING A NORM*/

int distance(int i, int j, int k, int l){
  int d;
  d =  (i - k) * (i - k) + (j - l) * (j - l);
  return d;
}




/*LAMBDA VECTOR*/

void lambda(float *y, float *lold, float *lnew, float *p){
  int i1, i2, j1, j2, k1, k2;
  float temp, tmp;

    for(i1 = 0; i1 < PIX; i1++){
      for(i2 = 0; i2 < PIX; i2++){

tmp = 0;
for(j1 = max(0, i1 - H + 1); j1 < min(i1 + H - 1, PIX); j1++){
  for(j2 = max(0, i2 - H + 1); j2 < min(i2 + H - 1, PIX); j2++){

    temp = 0;

    for(k1 = max(0, i1 - H + 1); k1 < min(i1 + H - 1, PIX); k1++){
      for(k2 = max(0, i2 - H + 1); k2 < min(i2 + H - 1, PIX); k2++){

temp = temp + lold[k1 * PIX + k2] * p[distance(j1,j2,k1,k2)];
      }
    }

    tmp = tmp + y[j1 * PIX + j2] * (p[distance(j1,j2,i1,i2)])/temp;
  }
}
lnew[i1 * PIX + i2] = lold[i1 * PIX + i2] * tmp;
      }
    }
  return;
}


/*Point Spread Function*/

void spread(float *p){
  int i, j;
  float temp = 0;
  for(i = -H; i <= H; i++){
    for(j = -H; j <= H; j++){
p[i * i + j * j] =  exp(-((float)(i * i + j * j))/SIGMA);
temp += p[i * i + j * j];
    }
  }


  for(i = -H; i <= H; i++)
    for(j = -H; j <= H; j++)
      p[i * i + j * j] = p[i * i + j * j]/temp;
  return;
}

/*RENEW THE VALUE OF LAMBDA VECTOR*/

void renew(float *lold,float *lnew){
  int i;
  for(i = 0; i < NUM; i++)
    lold[i] = lnew[i];
  return;
```

```
}


/*SAVING THE DATA*/

void datasave(float *l){
  int i, j;
  FILE *fp;
  fp = fopen("mylenn","w");

  for(i = 0; i < PIX; i++){
    for(j = 0; j < PIX; j++){
      fprintf(fp,"%f ",l[i * PIX + j]);
    }
    fprintf(fp,"\n");
  }

  return;
}
```

7.3 Shared Object Building

```
#include <R.h>
#include <Rdefines.h>
#include <Rinternals.h>
#include <stdlib.h>

#define H 2
#define PIX 512
#define NUM 512 * 512
#define SIGMA 10

#ifndef max
  #define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
#endif

#ifndef min
  #define min( a, b ) ( ((a) < (b)) ? (a) : (b) )
#endif
```

```
int distance(int i, int j, int k, int l);
void lambda(double *y, double *lold, double *lnew, double *p);
void renew(double *lold,double *lnew);
void spread(double *p);


SEXP lenna(SEXP data)
{
  int i;
  double *y, *lold, *lnew, *p;
  SEXP ans;

  ans = PROTECT(NEW_NUMERIC(NUM));
  double *z = NUMERIC_POINTER(ans);

  y = malloc(NUM * sizeof(double));
  lold = malloc(NUM * sizeof(double));
  lnew = malloc(NUM * sizeof(double));
  p = malloc((2 * (H + 1) * (H + 1)) * sizeof(double));

  for(i = 0; i < NUM; i++){
    y[i] = NUMERIC_POINTER(data)[i];
    lold[i] = y[i];
  }

  spread(p);

  for(i = 0; i < 4; i++){
    lambda(y, lold, lnew, p);
    renew(lold, lnew);
    /* printf("%d  \n",i); */
  }

  /* datasave(lnew); */
  for(i = 0; i < NUM; i++)
    z[i] = lnew[i];
  free(p);
  free(y);
  free(lnew);
  free(lold);

  UNPROTECT(1);
  return ans;
}




/*DEFINING A NORM*/

int distance(int i, int j, int k, int l){
  int d;
  d =  (i - k) * (i - k) + (j - l) * (j - l);
  return d;
}




/*LAMBDA VECTOR*/

void lambda(double *y, double *lold, double *lnew, double *p){
  int i1, i2, j1, j2, k1, k2;
  double temp, tmp;

    for(i1 = 0; i1 < PIX; i1++){
      for(i2 = 0; i2 < PIX; i2++){

tmp = 0;
for(j1 = max(0, i1 - H + 1); j1 < min(i1 + H - 1, PIX); j1++){
  for(j2 = max(0, i2 - H + 1); j2 < min(i2 + H - 1, PIX); j2++){

    temp = 0;
    for(k1 = max(0, i1 - H + 1); k1 < min(i1 + H - 1, PIX); k1++){
      for(k2 = max(0, i2 - H + 1); k2 < min(i2 + H - 1, PIX); k2++){

temp = temp + lold[k1 * PIX + k2] * p[distance(j1,j2,k1,k2)];
      }
    }

    tmp = tmp + y[j1 * PIX + j2] * (p[distance(j1,j2,i1,i2)])/temp;
  }
}
lnew[i1 * PIX + i2] = lold[i1 * PIX + i2] * tmp;
      }
    }
  return;
}


void spread(double *p){
  int i, j;
  double temp = 0;
  for(i = -H; i <= H; i++){
    for(j = -H; j <= H; j++){
p[i * i + j * j] =  exp(-((double)(i * i + j * j))/SIGMA);
temp += p[i * i + j * j];
    }
  }



  for(i = -H; i <= H; i++)
    for(j = -H; j <= H; j++)
      p[i * i + j * j] = p[i * i + j * j]/temp;
  return;
}

/*RENEW THE VALUE OF LAMBDA VECTOR*/

void renew(double *lold,double *lnew){
  int i;
  for(i = 0; i < NUM; i++)
    lold[i] = lnew[i];
  return;
}
```

8 Acknowledgement:

- Dr. Deepayan Sarkar, ISI Delhi,
- Wikipedia
- For this project we have used:
  - . GNU/Linux as the operating system
  - . C as programming language.
  - . R as interpreting language.
  - . LATEXfor preparinging presentation and project report.