Image Approximation for Sparse Replication

Artemii Yanushevskyi

Université de Toulon, assisted by Thierry Champion

May 25, 2018

Abstract

In this article I intend to research efficient image approximation and its applications in order to produce sparse images resembling original.

What is sparsity?

What is sparsity?

The main idea of **sparsity** or **parsimony** is to describe an object with as few variables as possible.

What about photos?

What about photos?

Dealing with potos, there are various approaches to achieve that, but here I will mainly focus on *optimization methods* that consist of minimization of the sum of two functions:

$$f(x, x_0) + \omega(x, x_0),$$

where x_0 is the original image and x is the **image-variable**.

We will refer to f as a function that *approximates* x to x_0 and ω as a **function-parameter** which depends on what outcome we intend to achieve. We search for x such that the sum is the smallest. Thus achieving *compromise* between approximation of one image to another with regard to the additional condition represented by $\omega(x, x_0)$.

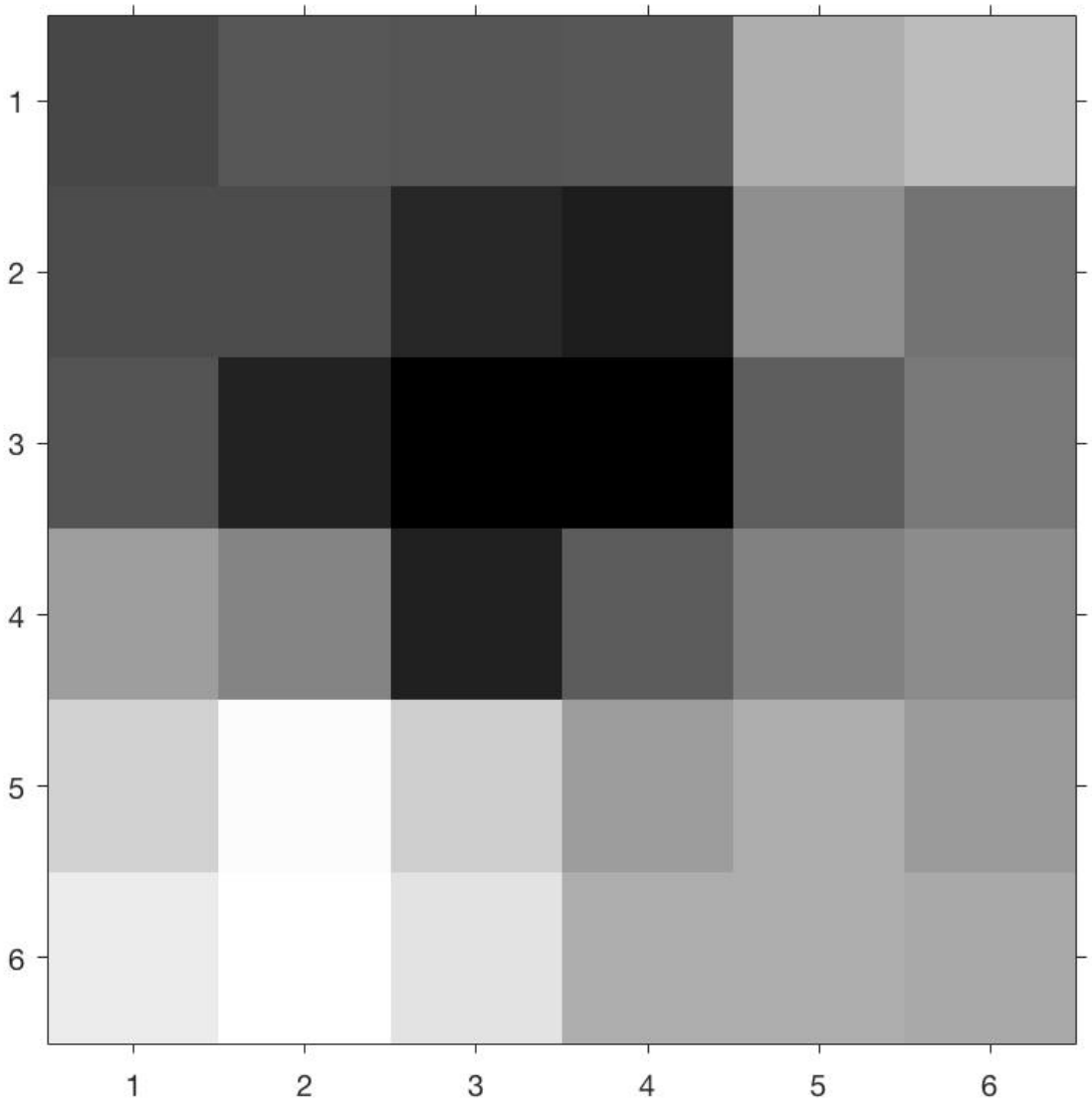
We will refer to f as a function that *approximates* x to x_0 and ω as a **function-parameter** which depends on what outcome we intend to achieve. We search for x such that the sum is the smallest. Thus achieving *compromise* between approximation of one image to another with regard to the additional condition represented by $\omega(x, x_0)$.
The function f is usually some norm and ω is some arbitrary function, preferably differentiable.

Setup

Setup

I will use MATLAB to make numeric computations. Black&white images will be represented by matrices. Coefficients of the matrix ranging from 0 to 1 represent pixels by a quotient of white.

A picture of a few black and white pixels



$$\begin{pmatrix} 0.27843 & 0.34118 & 0.33333 & 0.34118 & 0.68235 & 0.73725 \\ 0.29804 & 0.29804 & 0.15294 & 0.11373 & 0.56078 & 0.45098 \\ 0.32941 & 0.13333 & 0 & 0 & 0.36863 & 0.47451 \\ 0.61961 & 0.51765 & 0.12549 & 0.36078 & 0.50588 & 0.54902 \\ 0.82353 & 0.98824 & 0.81176 & 0.61569 & 0.68235 & 0.60784 \\ 0.92549 & 1 & 0.8902 & 0.68235 & 0.68235 & 0.66275 \end{pmatrix}.$$

Two images, `im` – an initial image, and `white` – a white image, both of a size 6x6.

```
im = imread('im.png');
white = imread('white.png');
```

Convert them to B&W:

```
im = rgb2gray(im);
white = rgb2gray(white);
```

Convert to a matrix

```
im = im2double(im);
white = im2double(white);
```

Matrix representation of an image

$$A_{\texttt{im}} := \texttt{im} = (a_{ij})_{\substack{0 \le i \le N \\ 0 \le j \le M}} . \quad a_{ij} \in [0, 1]$$

`imvar` – a variable-image. The code to assign white image to this variable:

```
imvar = white;
```

We have matrix representation of an image

$$A_{\texttt{imvar}} := \texttt{imvar} = (a_{ij})_{\substack{0\le i\le N\\ 0\le j\le M}}. \quad a_{ij} \in [0,1]$$

The command

```
imshow([im imvar imvar-im im-imvar])
```

concatenates 4 matrices in a row and displays the result.

The command

```
imshow([im imvar imvar-im im-imvar])
```

concatenates 4 matrices in a row and displays the result. Note: `imvar-im` is a matrix difference that displays how much two images are different. Later we will compute the norm

To calculate the l-norm to power l of a matrix we use the following formula

$$||A||_l^l = \sum_{i,j=1}^{N \times M} |a_{ij}|^l ,$$

which corresponds to the code below

```
function n = normsum(imvar, p)
    vect = reshape(imvar.',1,[]);
    % straighten matrix into vector
    n = norm(vect, p)^p;
end
```

What does it mean to approximate one image to another?

It basically represents approximation one matrix $A_{\texttt{imvar}}$ to another $A_{\texttt{im}}$ under some norm $\|\cdot\|$.

$$\min\left\{\|A_{\texttt{imvar}} - A_{\texttt{im}}\| \ : \ x \in [0,1]^{N\times M}\right\}.$$

To begin with, I will demonstrate how we approximate a purely white image to a given image using different norms.

Euclidian norm

Substitute $f(x_0, x)$ with $\|x_0 - x\|^2 = \sum_{ij} |a_{ij}|^2$. In previously introduced terms it means we solve the following problem

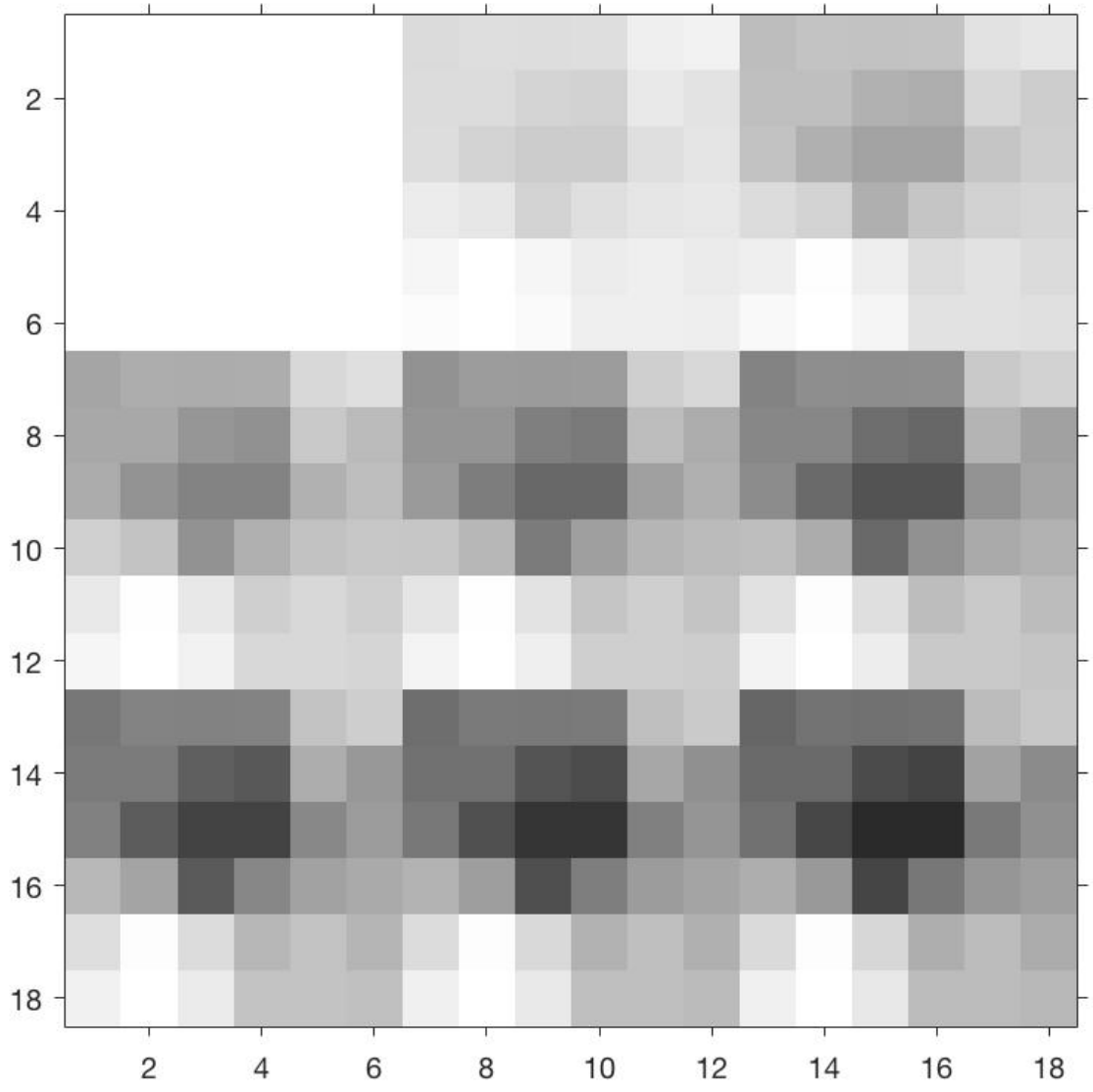
$$\min \left\{ \|A_{\texttt{imvar}} - A_{\texttt{im}}\|^2 \ : \ x \in [0,1]^{N \times M} \right\} .$$

To calculate $\nabla_{\texttt{imvar}} f$ we use this function

```
function gradient = grad_f(im, imvar)
    gradient = 2*(imvar-im);
end
```

Initially, we will consider the method of the steepest decent with fixed step h which we assign to 0.1 to begin with. The code of one iteration is below

```
imvar = imvar - h*gradf(im,imvar);
% to visualize results
figure(1)
imshow([im imvar imvar-im im-imvar])
% find out how close are imvar to im
normsum(imvar, 2)
```

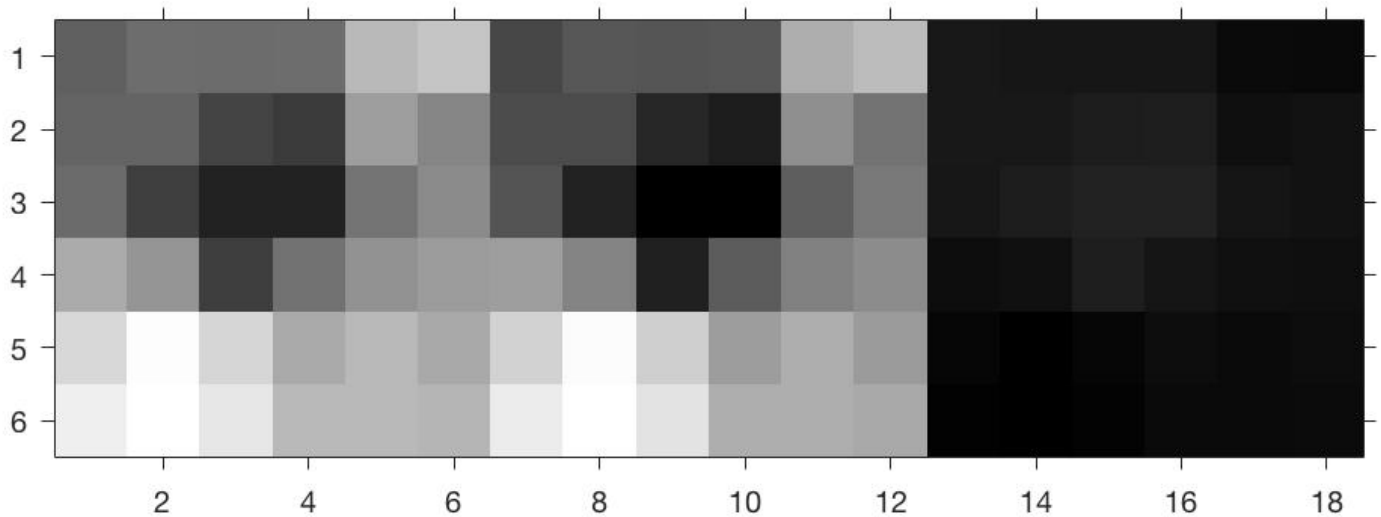


On the each iteration we calculate `normsum(imvar, 2)`

iteraton	$\|A_{\texttt{imvar}} - A_{\texttt{im}}\|^2$
1	11.6620
2	7.4637
3	4.7768
4	3.0571
5	1.9566
6	1.2522
7	0.80141
8	0.5129
9	0.32826

Now we will see how different initial image from our approximating image on a step 9

```
>> imshow([imvar, im, imvar-im])
```



`imvar`, `im` and `imvar-im`

Visually `im` and `imvar` are identical, which proves feasibility of the method. However the image `im-imvar` is not perfectly **black** and $\|A_{\texttt{imvar}} - A_{\texttt{im}}\|^2$ in not close enough to 0.
Let's continue with iterations until a more satisfying result

iteration	$\|A_{\texttt{imvar}} - A_{\texttt{im}}\|^2$
28	0.000068176

It takes 28 steps to approximate one image to another with precision $\varepsilon = 0.0001$.
It is quite time consuming even for that small image, it took 1.53 seconds to perform this approximation. To estimate how much it would take for bigger image we need to calculate how much small images would fit in the bigger image. Say for example we have image 600x600, so there are 10,000 images 6x6 in it. Thus, roughly the time to approximate this image is 10,000 times higher: 15,300 seconds.
Thereby, I need a more efficient algorithm. I discus that in an other section.

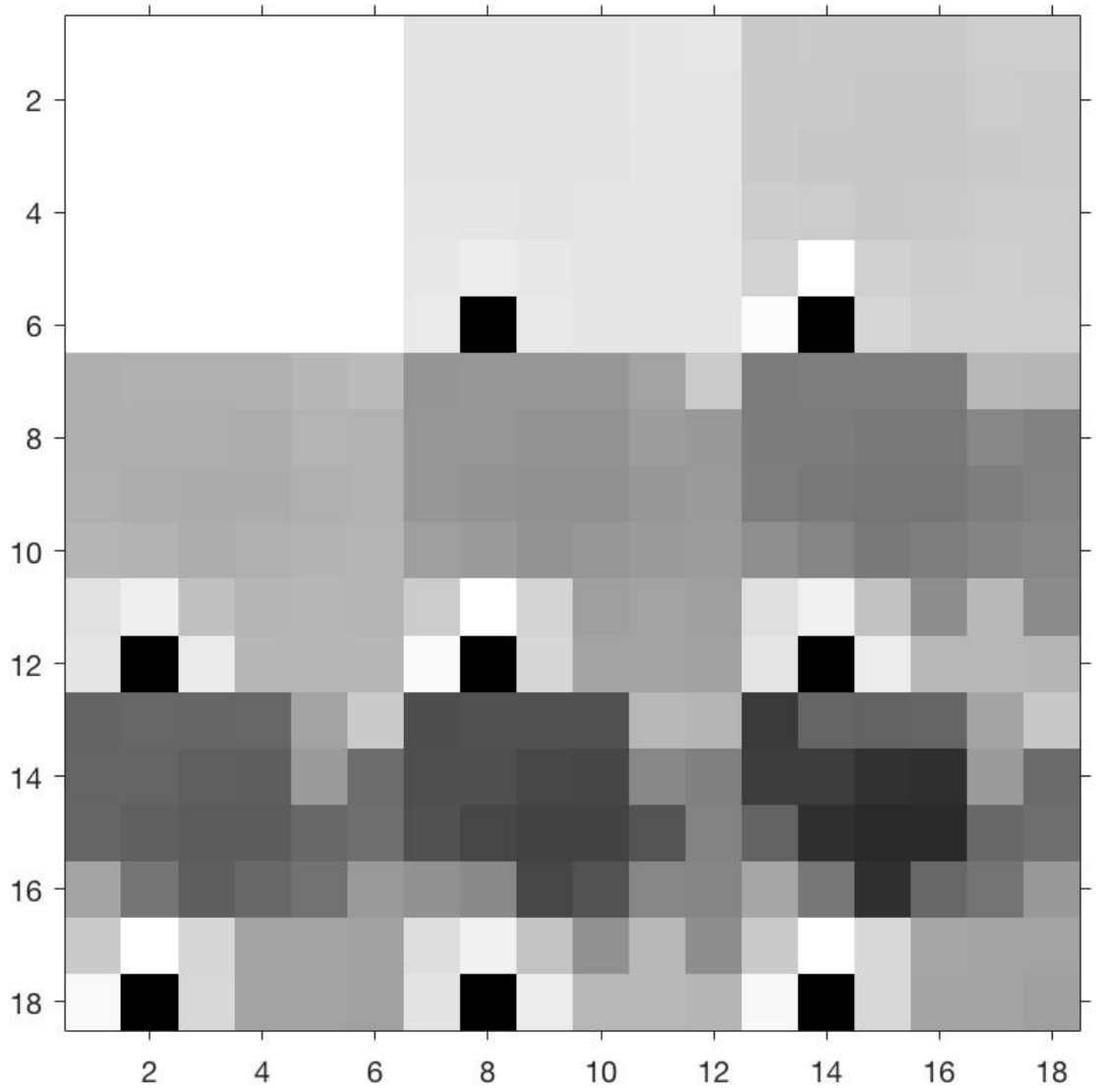
Norm $l = 1.1$

With some modifications in previous code we can now approximate one image to another under the norm $l = 1.1$.
The toughest part is to calculate and code a gradient of the function f.

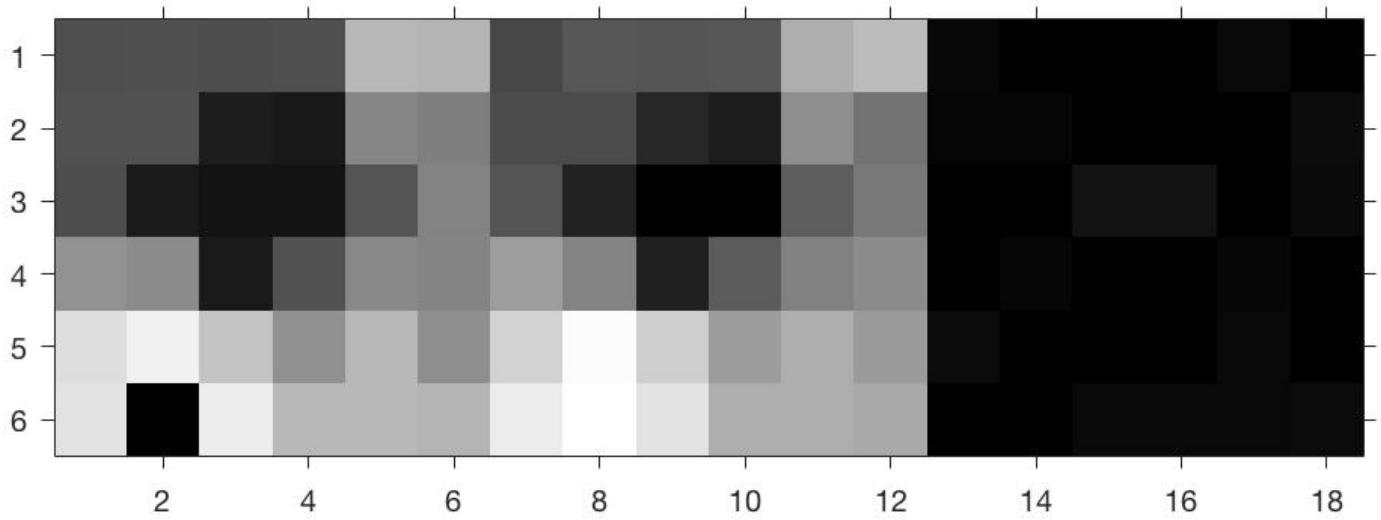
$$f(\mathtt{im}, \mathtt{imvar}) = \|\mathtt{im} - \mathtt{imvar}\|_{l=1.1}^{1.1}$$

$$\nabla_{\mathtt{imvar}} f(\mathtt{im}, \mathtt{imvar}) =$$

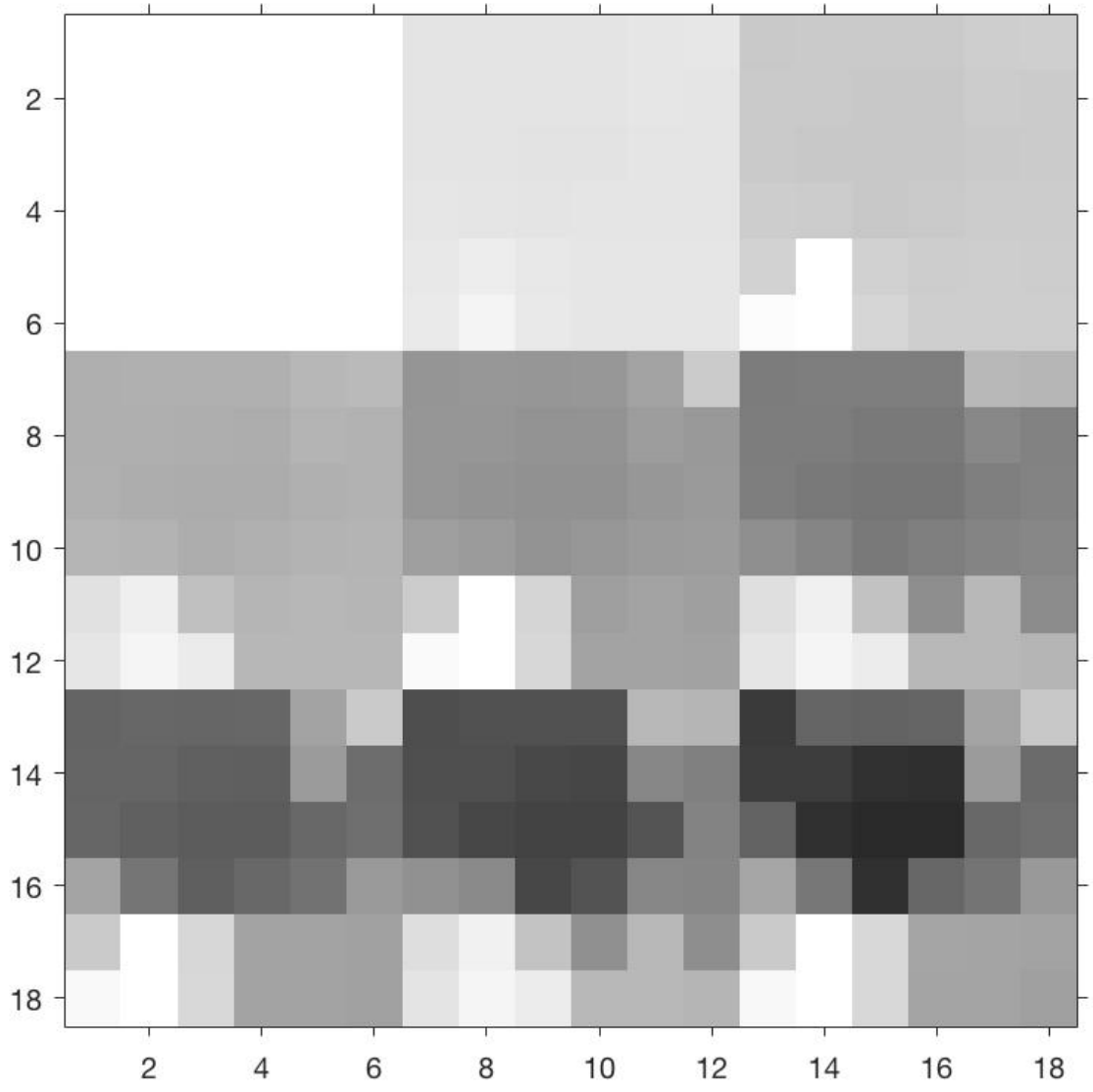
$$\begin{pmatrix} -\frac{1.1(\mathtt{im}_{11}-\mathtt{imvar}_{11})}{|\mathtt{im}_{11}-\mathtt{imvar}_{11}|^{0.9}} & \cdots & -\frac{1.1(\mathtt{im}_{1M}-\mathtt{imvar}_{1M})}{|\mathtt{im}_{1M}-\mathtt{imvar}_{1M}|^{0.9}} \\ \vdots & \ddots & \vdots \\ -\frac{1.1(\mathtt{im}_{N1}-\mathtt{imvar}_{N1})}{|\mathtt{im}_{N1}-\mathtt{imvar}_{N1}|^{0.9}} & \cdots & -\frac{1.1(\mathtt{im}_{NM}-\mathtt{imvar}_{NM})}{|\mathtt{im}_{NM}-\mathtt{imvar}_{NM}|^{0.9}} \end{pmatrix}$$

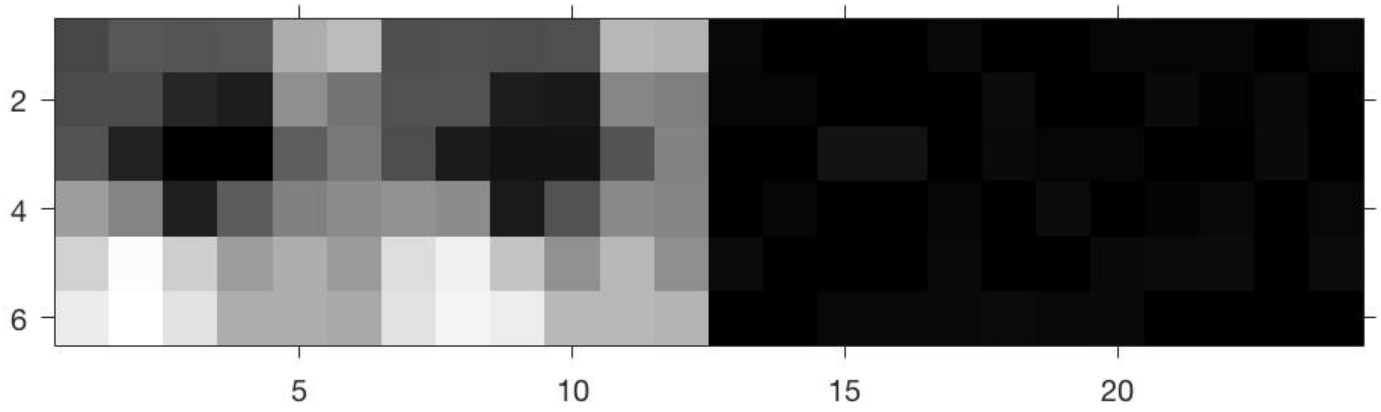


It took about 1.1 seconds to get the result.

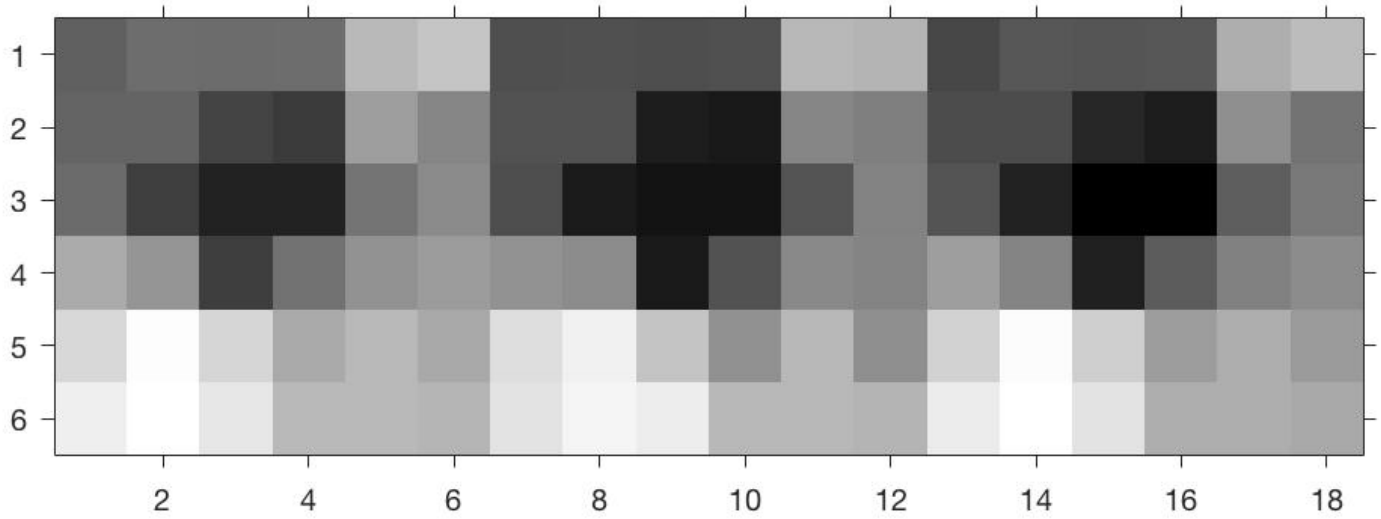


We get a malfunctioning pixel because its initial value is 1. That is one disadvantage of this norm. It is quite insignificant, for practical purposes I replace the value of the pixel with 0.9999.


2
4
6
8
10
12
14
16
18
2
4
6
8
10
12
14
16
18




```
>> imshow([imvar2 imvar1_1 im])
```



imvar $l = 2$, imvar $l = 1.1$, and original im

iteraton	f
1	17.1595
2	13.7489
3	10.6044
4	7.8324
5	5.6035
6	3.9322
7	2.6232
8	1.6886
9	1.4131

After 30th step it stays around 1 and stops to decrease. It is due to the fact that we have to big step $h = 0.1$.
If we choose step $h = 0.01$ squared norm of the difference stops to decrease at 140th iteration with value 0.062305.
Yet again, we would like to change the step dynamically so that it is not as time consuming.

Now I will apply Armijo method to these problems. The central function for Armijo method is:

$$q(t, \texttt{im}, \texttt{imvar}) = f(\texttt{imvar} - t * \boldsymbol{\nabla}_{\texttt{imvar}} f(\texttt{im}, \texttt{imvar}))$$

Now we will discover how efficient Armijo method is when dealing with norm $f = \|\cdot\|_{l=1.01}^{1.01}$. I specifically chose this norm because there is a perception that norm $\|\|_{l=1}$ possesses properties of sparsity. I will develop this in the next sections.
I don't choose l to be equal 1.

To implement function q I use the following code

```
function val = q(im, imvar, t)
    val = normsum(imvar - im -
        t*grad_f(im,imvar),1.01);
end
```

where f is $\|\cdot\|_2^2$

```
function func = f(im, imvar)
    func = sumnorm(im - imvar, 2);
end
```

The implementation of the Armijo algorithm

```
function a = armijo(im, imvar)
    % calculate the derivativ of q at the point 0
    q_der_0 = (q(im, imvar, 1/10^8)
        -q(im, imvar, -1/10^8))*10^8/2

    % define parameters
    alpha = 1;
    c = 0.5;
    k = 1;

    % build graphs of q
    X = -10:0.01:10;
    Y1 = [ ];
    for x = X
       y = q(im, imvar, x);
       Y1 = [Y1, y];
    end
```

```
% build graph of q+c*q_der_0*x;
Y2 = [ ];
for x = X
   y = q(im, imvar, 0)+c*q_der_0*x;
   Y2 = [Y2, y];
end

% plot the two graphs
figure(2)
plot(X,Y1,X,Y2)


if q_der_0 < 0
    % start cycle to find the optimal step
```

```
    while k < 1000
        if ( q(im, imvar,alpha/2^k)<
        q(im, imvar,0)+c*q_der_0*alpha/2^k )
            break
        end
        k = k + 1;
    end
else
    a = ”error”
    return
end

k % output how many steps
 % it took to get the optimal step value
a = alpha/2^k % output the optimal step
```

```
    hold on
    plot([a],[q(im, imvar, a)],'b*')
    hold off
end
```

Finally, the one iteration of our approximation

```
    imvar = imvar - armijo(im,imvar)*grad_f(im,imvar);
    figure(1)
    imshow([im imvar imvar-im im-imvar])
    imvarv = normsum(imvar-im, 1.01)
    q(im,imvar,0)
```

To run test I use optimized version of the Armijo function, I avoid abundant logic checks and I do not build graphs. This alone had a significant impact on time efficiency.
Additionally, I preallocated a memory for the array of our pictures on each step.

```
imvars = cell(3,3);
imvarsv = zeros(3,3);
```

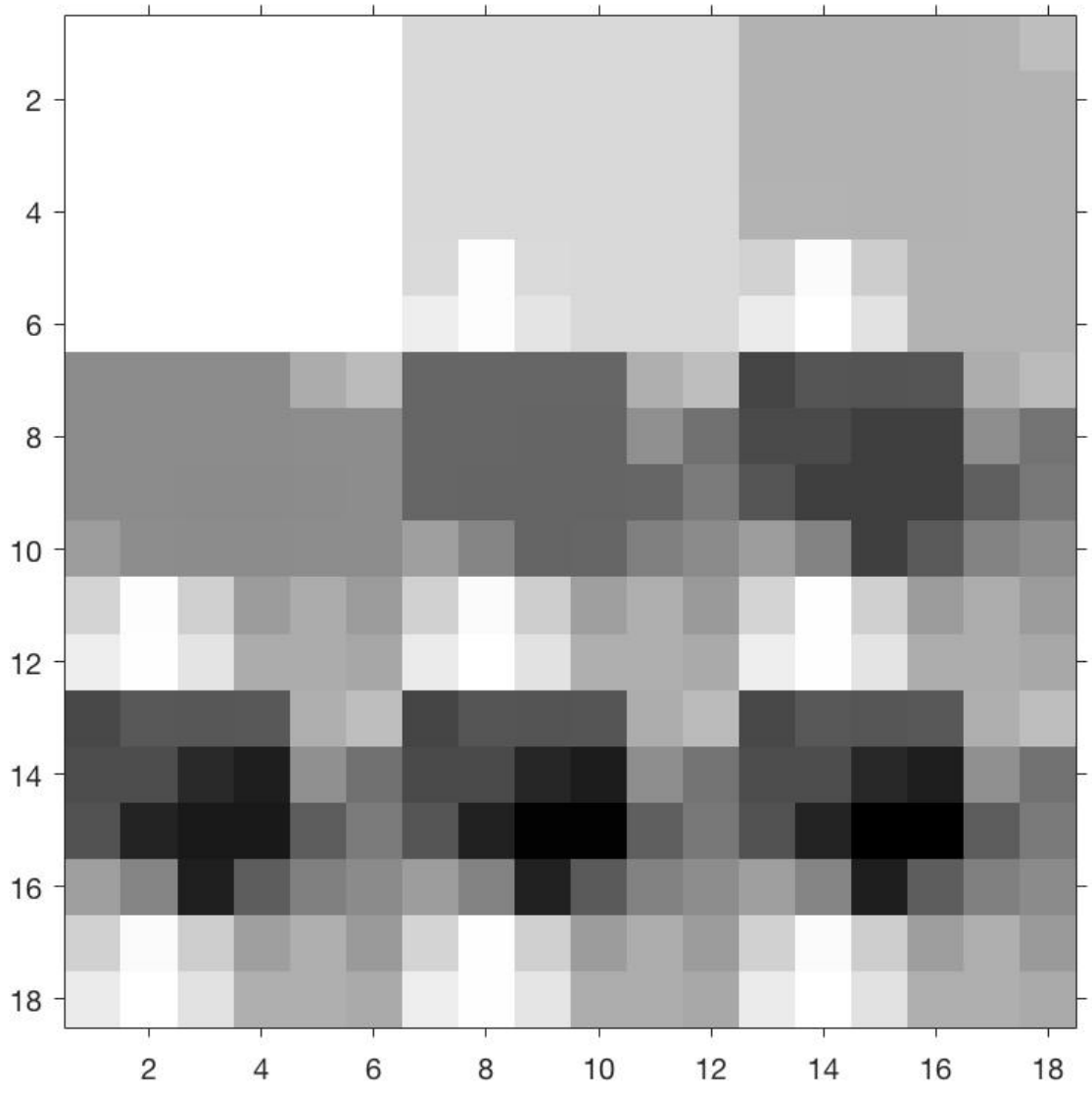
Comparison

Now I will compare efficiency of two methods: first – with constant step and second – Armijo method.

This is the optimized cycle which makes a sample each 15th iteration ($h = 0.01$ is constant)

```
tic
for i = 1:1:3
    for j = 1:1:3
        imvars{i,j} = imvar;
        imvarsv(i,j) = imvarv;
        for k = 1:1:15
            imvar = imvar - h*grad_f(im,imvar);
        end
        figure(1)
        imshow([im imvar imvar-im im-imvar])
        imvarv = normsum(imvar-im, 1.01)
    end
end
imvars = cell2mat(imvars);
matrix2latex(reshape(imvarsv.',1,[])', 'matr.tex')
toc
```

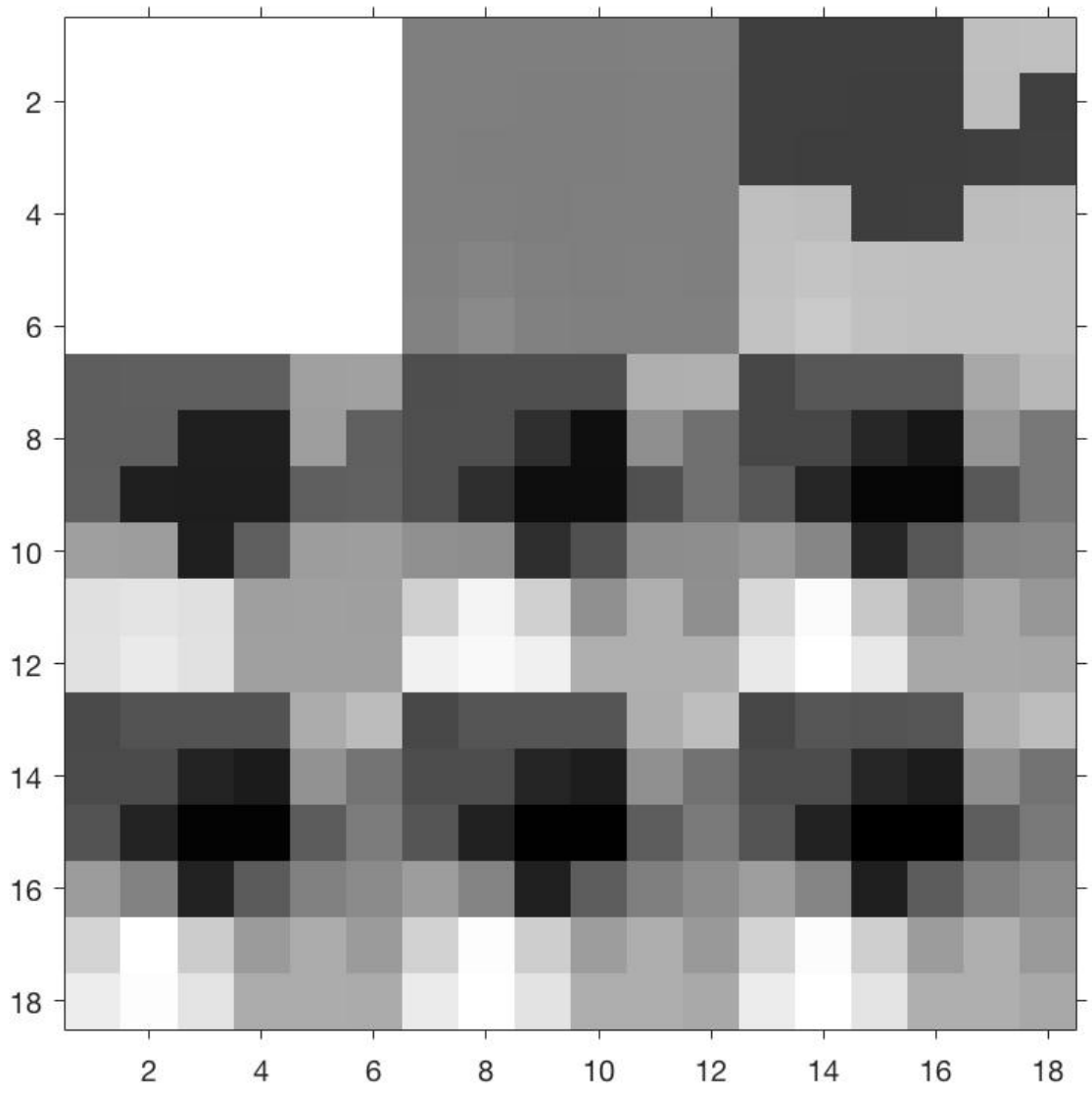
It takes on average 0.75 seconds to make this approximation:



iteraton	f
1	17.9618
2	12.9758
3	8.4819
4	5.0433
5	2.5674
6	1.0905
7	0.34507
8	0.17081
9	0.15819

It is worth noting, that actually the program does 3x3x15 iterations(!). But for the simplicity of demonstration we select every 15th iteration to display the resulting image.
Despite the number of iterations, the precision of approximation is not quite satisfying. Moreover, this method is not reliable since the norm of the differences steadily decrease.
On a top of that, we can not actually set a desired precision of an approximation. This is major disadvantage when accuracy is at stake.

Armijo method – results



With this method we obtain those results in 0.87 seconds.
This method clearly does not have before mentioned downsides.

iteraton	f	Armijo step
1	17.9618	0.5
2	7.9075	0.25
3	4.4445	0.125
4	1.8956	0.0625
5	1.0473	0.03125
6	0.56739	0.015625
7	0.26702	0.0078125
8	0.12232	0.00390625
9	0.057788	0.001953125

The precision of this method is 10 times higher in about the same runtime.
Important to note that on 13th step the norm of the difference is less than 0.01 (0.0071 to be exact). Such computation took 1.4 seconds. And it is expected to continue to steadily decline.

Sparsity

Now I will mainly focus on the notion of sparsity and method to achieve it.
Suppose we have the image we used before and now we would like to amplify clarity, which means that pixels that are dark will get darker and the light pixels even lighter.
Now I will introduce function ω that will encourage sparsity. In other words, function ω "punishes" values of $f + \omega$ with big numbers. Thus values of ω should stay small, while image is approximated so that values of f stay small as well.

0-1 function

$$\omega_{test}(\texttt{imvar}) = \lambda \cdot \sum_{i,j=1}^{N \times M} \texttt{imvar}_{ij} \cdot (1 - \texttt{imvar}_{ij})$$

This function fulfills our expectations. However it is not practical, since it is not differentiable at 0 and 1. It could not be used with methods developed above. (Technically it works, but results are rather unpredictable)
The best function of this type would be this one

$$\omega(\texttt{imvar}) = \lambda \cdot \sum_{i,j=1}^{N \times M} \texttt{imvar}_{ij}^2 \cdot (1 - \texttt{imvar}_{ij})^2.$$

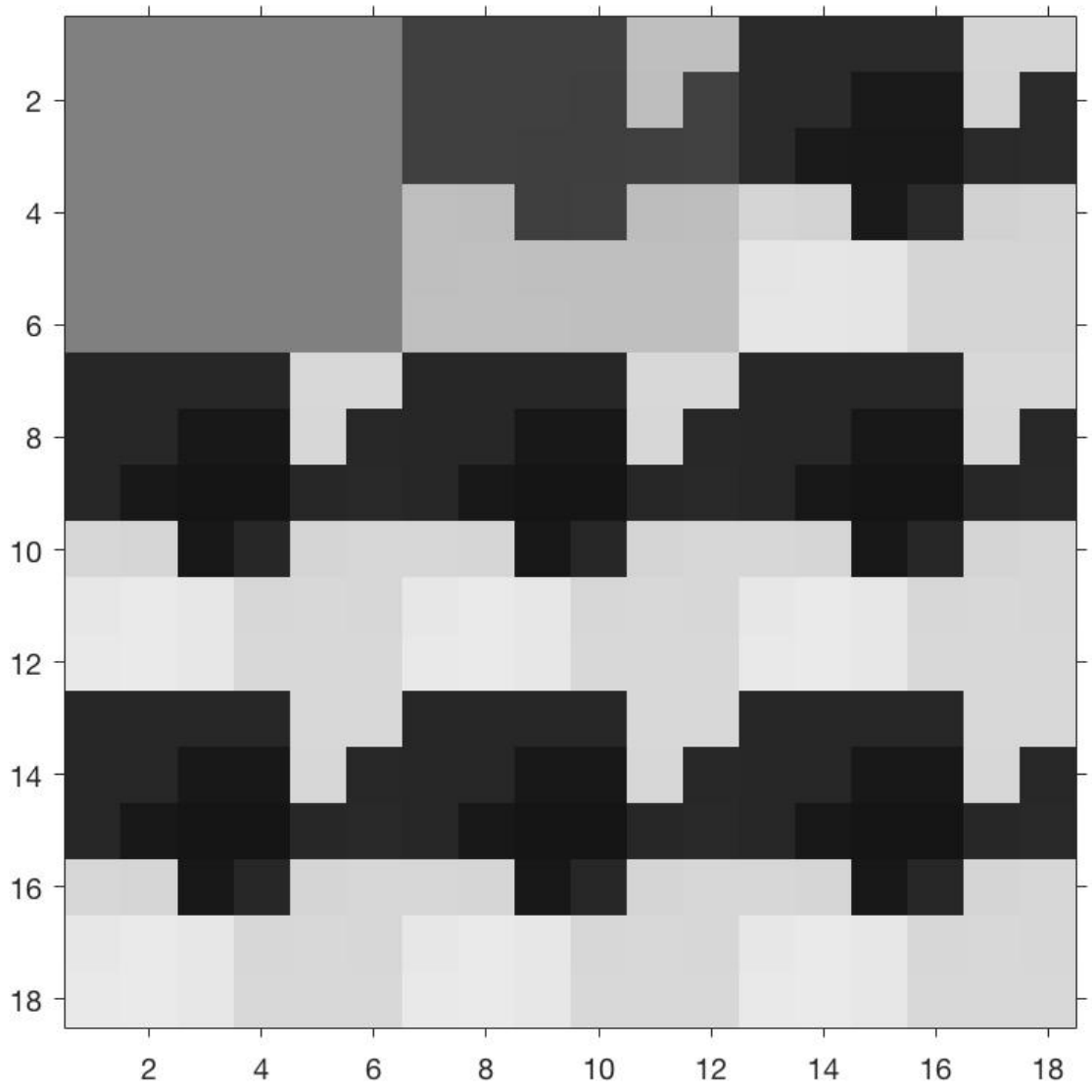
It possesses the properties we need and is differentiable

$$\nabla\omega(\texttt{imvar}) = 2\lambda \cdot \sum_{i,j=1}^{N\times M} \texttt{imvar}_{ij}(\texttt{imvar}-1)(2\texttt{imvar}-1)$$

I am using $f(\cdot) = \|\cdot\|_{l=1.01}^{1.01}$.

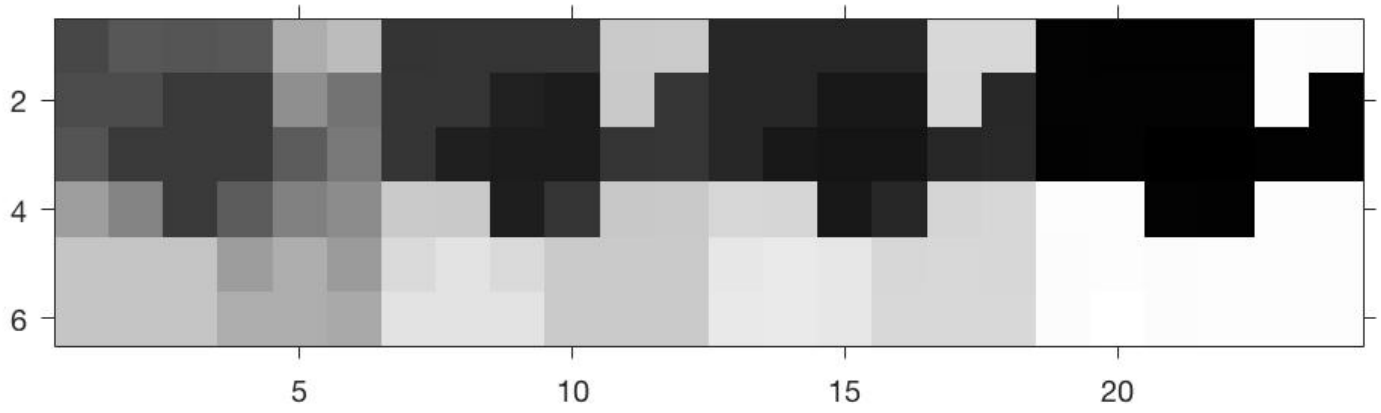
```
function gradient = grad_omega(im, imvar,lambda)
    gradient = lambda*
        ( 2*imvar.*(imvar-1).*(2*imvar-1) );
end
```

Now we will see how this program works with a small image 6x6. Obviously we use Armijo method for efficiency.



iteraton	$f + \omega$
1	15.5427
2	35.8264
3	35.6575
4	35.6286
5	35.6285
6	35.6285
7	35.6285
8	35.6285
9	35.6285

Just in 4 steps we get the desired result. In this case **influence coefficient** λ is equal to 20. The average runtime 1.15 seconds. The average time for all the approximation is slightly more than 1 minute, it decreases with increase of influence coefficient, so does the number of iterations.
In case the value of influence coefficient is greater than 50, the minor details turn invisible while somewhat substantial details become much more pronounced – we will refer to this as **distinct sparsity**.



Influence coefficients: $\lambda = 1, 10, 20, 50$.

Another important parameter is what I call **sensitivity** or **threshold**. In previous example I used threshold 0.5. It characterizes the color of initial image `imvar`.

Threshold coefficients: 0.1, 0.2, ..., 0.9. $\lambda = 10$.

The first row are `imvars` at the beginning, and second are `imvars` after approximation.