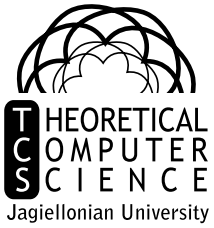
Digital watermarking

Paweł Wanat
Student number: 1052731

Master's thesis
Computer Science—IT Analyst

Supervisor:
dr Grzegorz Matecki



Theoretical Computer Science Department
Faculty of Mathematics and Computer Science
Jagiellonian University
September, 2016

Digital watermarking

Abstract

Digital watermarking is a concept of adding hidden information to digital media as photos or movies so the additional information cannot be spotted with the naked eye. Over time scientists developed various methods that embeds and detects a watermark on a digital content. It is considered difficult to obtain original unwatermarked work from the watermarked without knowing the embedded watermark. In this document, there is presented a method that: having a large set of images which were watermarked with the same watermark using E_BLIND embedder, it is able to deduce the embedded watermark with $99,6\%$ accuracy and so is able to almost recover the original work.

1 Introduction

Hiding information have been a concept since the beginning of human existence. When Adam and Eve ate a fruit from forbidden tree they tried to hide this fact away from the God by just hiding themselves. They did not have better tools for that because it was the moment soon after they gained the knowledge. When time was passing people gained better and better methods for hiding information. We know now about the Caesar cipher which was known at least since Julius Caesar times. Nowadays high-tech cryptography is used to protect Internet users from third-party eavesdrop or to keep their banking save. RSA, MD5, ECC, AES are just some of the methods that are used.

There is another approach to the problem of hiding information, called Steganography. Instead of using cryptography, we hide some information in another one, so the presence of the hidden is unnoticeable. Par example, there are inks that are only visible in UV light or when you apply citric acid on them. So you can send a letter with normal ink and add additional message on the letter using secret-ink. The letter will look like ordinal, so nobody will put attention to check it for some additional information. Another example that is more relevant to digital media is to hide an information in the least significant bits (LSB) of bitmap images. Every pixel contains 3 color: red, green and blue. Usually every of these 3 takes integer value from 0 to 255. When you change the least significant bit of red value then the changed value will differ from original one by at most 1. That small difference is unnoticeable to human eye.

In this document, we will focus on another method of hiding information - Watermarking. Basically, watermarking is a way of hiding information, so we can verify the information only when we are aware of the method how to do so. We can find watermarks on banknotes or government bonds. A way of verifying their existence is to view them by placing a light source on the other side. In a digital world, we usually watermark movies, images or audios. When we watermark images, we exploit the fact that some of information in pictures is redundant. We manipulate with the images, so they look the same, but they are different from original. Methods of watermarking have been developed. Some examples are E_BLIND/D_LC, E_FIXED_LC/D_LC, E_BLK_BLIND/D_BLK_CC, E_MOD/D_LC or E_DCTQ/D_DCTQ described in Digital Watermarking and Steganography [2] book. In this document, we choose E_BLIND/D_LC watermarking system and we show its weakness: assuming that we are provided a set of images with the same embedded watermark, we manage to

successfully remove the watermark from every picture of the provided set. People try to cheat watermark detection systems by cropping, scaling, rotating, transforming, compressing digital contents. However, these operations usually remains large percentage of hidden information. We are much better in this metric because we are getting rid of $99,6\%$ of hidden information.

2 Concept of watermarking

Quoting wikipedia [3]:

> A watermark is an identifying image or pattern in paper that appears as various shades of lightness/darkness when viewed by transmitted light (or when viewed by reflected light, atop a dark background), caused by thickness or density variations in the paper. [1] Watermarks have been used on postage stamps, currency, and other government documents to discourage counterfeiting.

This concept has been introduced to digital world, so authors of intellectual properties (IP) could protect themselves from thieves or people who just forgot to point the source. In last decade when more and more things are being computerized, there is a need to protect the things from being copied and pasted somewhere else in case they should not be moved from origin.

Figure 1: Example of a watermark in the twenty euro banknote. *https://upload.wikimedia.org/wikipedia/commons/8/82/Watermarks_20_Euro.jpg*

Usually when a person thinks about a watermark on images, they imagine partially transparent text that can be seen directly. Techniques have been developed to remove such manually. To protect IPs, better methods had to be created, so that removing a watermark should be considered hard. In this paper, we embeds watermarks imperceptible to human eye. However, some examples will contain visible watermarks to make understanding cleaner.

3 Motivations for watermarking of digital content

Let us consider 3 possible use cases of watermarking digital content.

Direct leak detection

Film production company makes a film. They claim that their new movie is awesome. However, they want to make money, so they want people to pay for watching them in cinema. Life is tough and nobody will pay for a bad movie. Thus the company wants to send the movie to several film critics, so they give a good recommendation and so they could earn money. However, the company is afraid that if they share the movie with critics then they could share the movie onwards. To prevent critics from sharing, the company can watermark every copy, so they will know who made a leak in case of any.

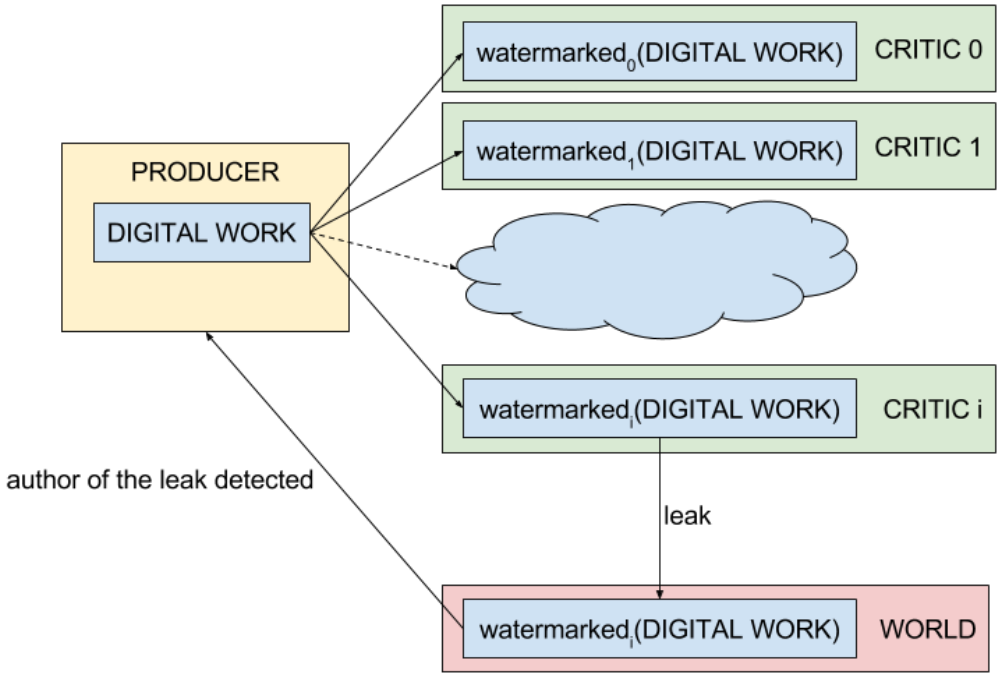


Figure 2: Schema of a detection process when a work was leaked by a direct user.

Figure 2 explains the solution to presented problem. Producer who owns a digital work, watermarks every copy of the digital work with a different watermark and sends the watermarked contents to critics. Whenever a critic

makes a leak, we can identify him by checking if the leaked work contains a watermarked associated with him. Figure 3 shows a potential image that would be leaked by Paweł Wanat.

Figure 3: A picture that would be leaked by Paweł Wanat.

Content tracking

Film production company makes movies. They already have a big portfolio that contains hundreds of digital works. They know that from time to time some malicious people upload their movies to YouTube. They are very sad about that. Verifying that a single movie found on YouTube belongs to them is costly, so before any release they watermark all their movies with single watermark and they just check, if a movie from YouTube contains their watermark.

On Figure 4, we can find that the producer has n movies and there are m movies in the Internet that could potentially belong to the producer. The brute force algorithm would have to compare every producer's movie with every movie from the Internet. That would give us $n \cdot m$ comparisons.

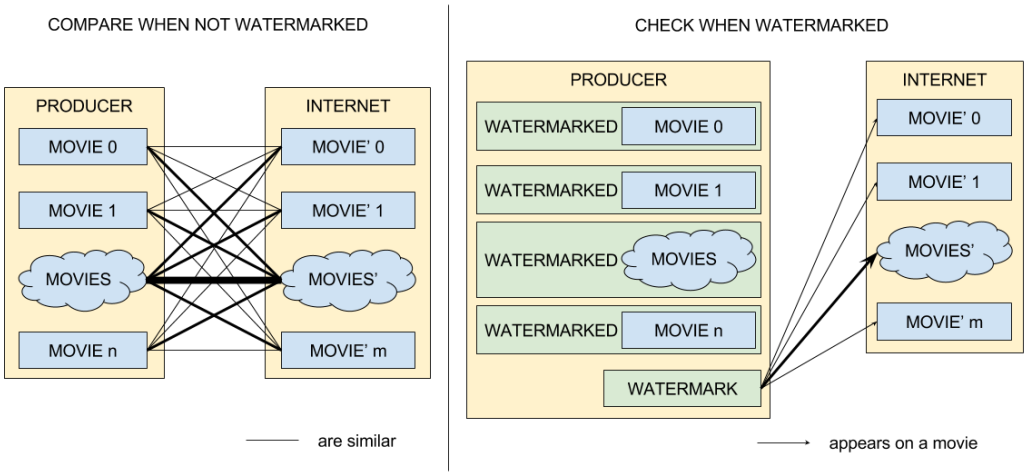


Figure 4: Computation work of comparing, when pictures are watermarked and not.

However, if we watermark movies before, then all we have to do is to check if producer's watermark appears on a movie from the Internet. That requires merely m checks.

Solution to this problem might be used by photographers who post their photos in the Internet. It happens frequently that people are reposting their pictures on their feeds in social media without pointing the source.

End user leak detection

Film production company made a movie. They want to sell their movie to some end users, via cinemas or other film brokers. They are afraid that some end user will find a way to download or record the content and then share it somewhere in the Internet. They want to secure themselves, so they always can identify the end user who made a leak. They decided that they will watermark the movie before sending to any broker and they ask the brokers to watermark it again before sending to any user. When the leak occur, the company will identify the broker and the broker will identify the end user.

Figure 5 explains the process of identifying the user in case of an Internet streaming film broker. At first we have a first level of watermarking. We send wm_i(DIGITAL WORK) to broker company i where wm_i(DIGITAL WORK) denotes watermarked version of DIGITAL WORK and index i ensures that for every broker we watermark the work using different watermark. Then there is a second level of watermarking when brokers distribute the work to the end users. When some end user will make a leak then we identify firstly the

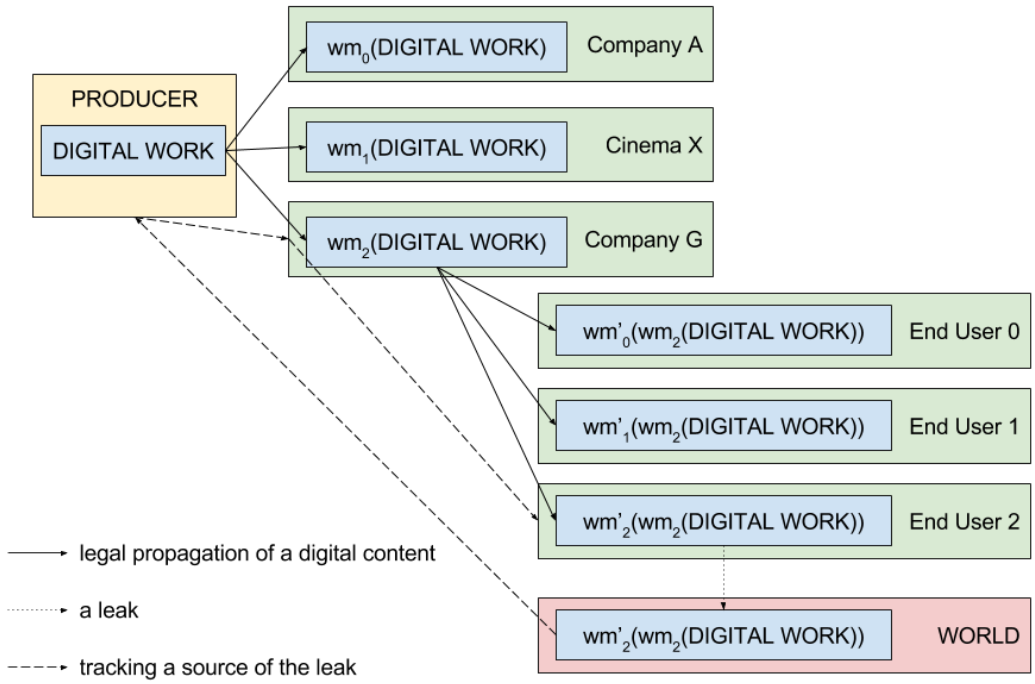


Figure 5: Schema of a detection process when a work was leaked by an end user.

broker and the broker identifies the end user. Figure 6 shows a potential image that would be leaked by Paweł Wanat through Company G.

It is worth to mention how to identify an end user if they recorded hiddenly the movie in a cinema. At first we would have three layers of watermarking: company wide watermark, physical address of subordinate, date and room. Then they would be able to identify the seat of the end user by geometric properties of recorded screen.

4 Background

All the methods shown later will be implemented in source codes for RGB images. However, for theoretical simplicity, we are considering grayscale images only, i.e. every pixel is an integer value from 0 to 255. Another assumption is that all images are of the same size $width \times height$. Later on, c denotes an image, c^k denotes k-th input image, c_i denotes i-th pixel of an image. It is worth to mention that we are using 2D indices, so when writing i-th pixel, the variable i is 2D. Let w be a vector in $\{1, -1\}^{width \times height}$ space, called a watermark and let W be a random vector uniformly distributed over

Figure 6: A picture that would be leaked by Paweł Wanat through Company G.

$\{1,-1\}^{width \times height}$. In the source codes we will be treating a watermark as a 2-color image, where a black pixel i denotes $w_i = 1$ and a white pixel i denotes $w_i = -1$. Below we list these definition and additional ones, so you can return to then quickly, if you forget any.

iff – if and only if
$abs(x) = -x$ if $x < 0$ else x
w – watermark, w_i in $\{-1, 1\}$
c/c^k – content image/k-th content image
c_i/c_i^k – value of pixel i in image c/c^k; $c_i/c_i^k \in \{0, 1, \ldots, 255\}$
$E(X)$ – expected value of random variable X
$Var(X)$ – variance of random variable X
W – random watermark
$A \cdot B = \sum_i A_i B_i$ – dot product
$lc(A, B) = linear\_correlation(A, B) = \frac{A \cdot B}{length(A)}$
Chebyshev's inequality: $Pr(|X - E(X)| >= \epsilon) <= \frac{Var(X)}{\epsilon^2}$

By watermarking system we understand a pair of an embedding algorithm

and a detecting algorithm. Figure 7 describes data flow in a watermarking system. At first embedding algorithm takes a digital content and a watermark. Having them, it produces a watermarked version of the digital content. The detecting algorithm takes a watermark and some digital content. If the digital content contains the watermark, then it returns "Yes", otherwise "No". The solid line presents data flow for embedding algorithms. The dashed and dashed-dotted lines presents data flow for detecting algorithms when the answer is positive. The dotted and dashed-dotted lines presents data flow for detecting algorithms when the answer is negative.

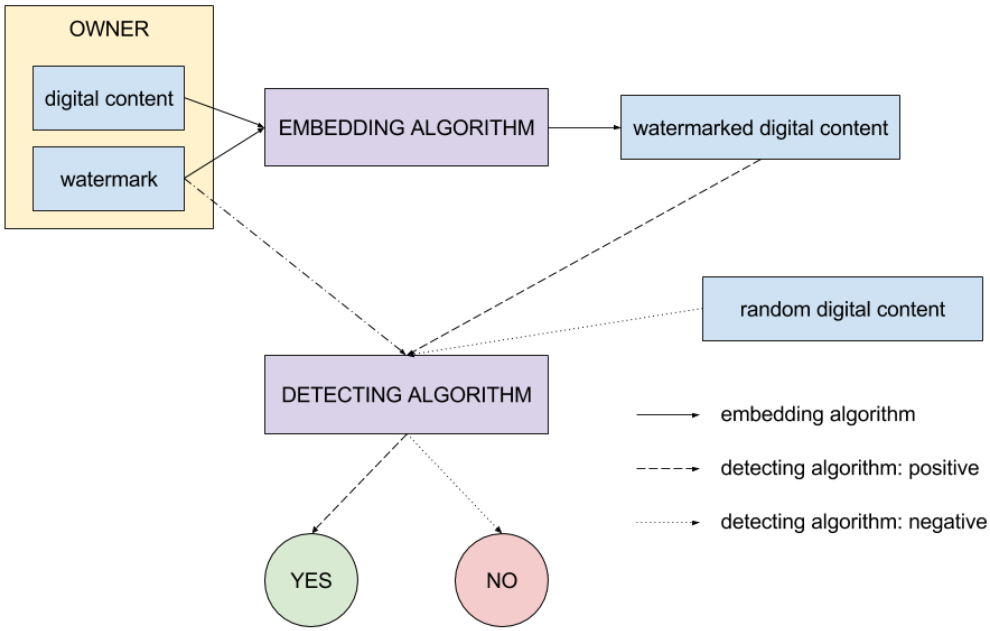


Figure 7: Embedding and detecting algorithms - work flow.

5 E_BLIND/D_LC watermarking system

Blind Embedding (E_BLIND) and Linear Correlation Detection (D_LC) is the simplest watermarking system presented in Digital Watermarking and Steganography [2] book. For this system, a watermark should be randomly generated vector of $\{1, -1\}^{width \times height}$.
The **embedding** algorithm is summation of two matrices. So any pixel in a watermarked content will differ by 1 from the original pixel.

```
input: c - an image,
```

```
        w - a watemark
output: wc - a watermarked image
algorithm:
  for each pixel i do
    wc[i] = c[i] + w[i]
```

The **detecting** algorithm checks value of linear correlation between a digital content and a watermark and in case it reaches some threshold (authors take 0.7 to get negligibly-small false-positive rate) then it reports a positive outcome of detection. In the below code, N denotes number of pixels.

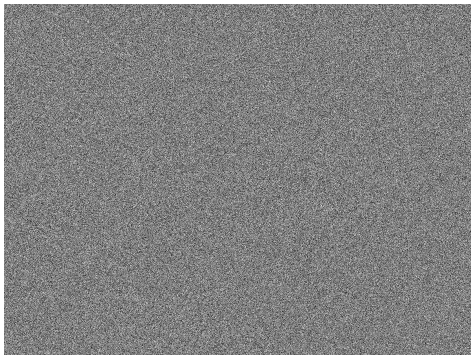
```
input: c - a potentialy watemarked image,
       w - a reference watermark
output: "watermark detected"
          if w appears on c
          else "watermark undetected"
algorithm:
  let lc = sum{c[i]* w[i] : pixel i in watermark} / N
  in
    if lc > 0.7
      then watermark detected
      else watermark undetected
```

To get better understanding why it works, we should consider following points:

1. linear correlation between a watermark and itself is 1
2. having a watermark randomly generated, linear correlation between a watermark and an unwatermarked image is expected to be around 0
3. having a watermark randomly generated, linear correlation between a watermark and a watermarked image is expected to be around 1

Point 1 is true because $lc(w,w) = \frac{\sum_i w_i \cdot w_i}{N} = \frac{\sum_i 1}{N} = 1$.
To show point 2, we take random watermark W and we consider $abs(E(lc(W,c))) = abs(E(\frac{W \cdot c}{|W|}))$. Observe that $W \cdot c$ is random walk around 0, cause c_i is magnitude W_i is direction, all W_i are mutually independent and $Pr(W_i = -1) = Pr(W_i = 1) = 0.5$. Thus $abs(E(W \cdot c))$ is likely to be upper bounded by $const \cdot \sqrt{N}$ and so $abs(E(lc(W,c)))$ is likely to be upper bounded by $\frac{const}{\sqrt{N}}$. So the point is true.
The last point is now simply true:
$lc(c+W,W) = lc(c,W) + lc(W,W) = lc(c,W) + 1 \approx 1$

Figure 8 show the blind embedding in use. On the top, you can find a sample watermark. The left bottom picture is before watermarking. The right bottom one is after watermarking with E_BLIND method.



A watermark

Not watermarked picture

Watermarked picture

Figure 8: Comparison between an unwatermarked work and a watermark one.

It is impossible to spot a difference with the naked eye between two bottom images. For the purpose of this document, we are saving files in a bitmap so the compression not to eat the watermark.

It is worth to mention that E_BLIND allows to subtract a watermark from an image, i.e. $wc_i = c_i - w_i$, but then D_LC compares threshold to $abs(lc(c, w))$ instead of $lc(c, w)$. This is a mechanism to pass a hidden message to a watermarked content. Basically, if we watermark with $wc_i = c_i + w_i$ formula then it means message 0, if we use $wc_i = c_i - w_i$ formula then it means message 1. To read the message from the watermarked content, we just check if linear correlation is positive or negative.

6 Breaking E_BLIND

We define here two probabilistic objects: Random Picture Model and Natural Picture Model. **Random Picture Model** is a probability space over the

set of images in which

- the value of every pixel has uniform distribution on $\{0, 1, ..., 255\}$
- the values are mutually independent

Natural Picture Model is a probability space over the set of images that is induced by reality. So we don't really know how it looks like, but we will try to observe some of it's properties and then based on the analysis of Random Picture Model we will try to make some conclusions.
Let us present the trick that will break the E_BLIND method. Basically, in Random Picture Model if we take random picture C and if we pick two adjacent pixels i and j from the picture then

$$Pr(C_i > C_j) = Pr(C_i < C_j). \tag{1}$$

However, if the image C is watermarked with E_BLIND (so $C = O + w$, for some O from RPM) and if $w_i > w_j$ then

$$Pr(C_i > C_j) = Pr(C_i < C_j) + \varepsilon, \tag{2}$$

for some $\varepsilon > 0$ that is common for every i and j.

Let us take a set of input images – a set of cardinality B. For every two adjacent pixels i and j, we focus on average over all images of $sgn(C_i - C_j)$. We prove that $E(avg_k(sgn(C_i^k - C_j^k)))$ is equal to ε when $w_i > w_j$ and is 0 when $w_i = w_j$. What is more, we show that $Var(avg_k(sgn(C_i^k - C_j^k))) = O\left(\frac{1}{B}\right)$ which is very small, if we have B big enough. Based on $avg_k(sgn(C_i^k - C_j^k))$, we would like to predict all the differences of $w_i - w_j$, for all adjacent pixels i and j. That might be hard, so we would be satisfied if we predict 90% of differences correctly. We call our prediction of these differences – *delta*. Having *delta* computed, we try to approximate embedded watermark.
Until now, we had consideration on random variables. To make it understandable from a computer perspective, we set $C^k = c^k$ and we compute *delta*. Then we have heuristic algorithms that can produce a watermark from *delta*.
We prove that the prediction of $w_i - w_j$ differences is correct on the level of 90%. Unfortunately we cannot proof any of above for Natural Picture Model. However, we proceed to treat images the same way and then we make a small adjustment that solves the problem. Finally, We explore how well the breaking algorithm is in reality.

The algorithm have two steps:

1. Computing *delta*.
2. Computing an approximated watermark form *delta*.

6.1 Defining *delta*

We are considering images of certain size $width \times height$. We define $delta(i, j)$ iff pixel i is adjacent to pixel j. We want to claim value $delta(i, j)$ that will denote our prediction about the watermark difference on pixel i and j, i.e. $w_i + delta(i, j) = w_j$ if we predicted correctly.

6.2 Computing *delta*

Let us recall that B is the number of pictures in the input set. For both Random Picture Model and Natural Picture Model we consider $\{C^k\}_k$. These are random variables that describe a set of input images. Then for any adjacent pixels i and j we define:

$$X_{ij}^k = \begin{cases} 1 & \text{, if } C_i^k > C_j^k \\ 0 & \text{, if } C_i^k = C_j^k \\ -1 & \text{, if } C_i^k < C_j^k \end{cases}$$

$Y_{ij} = \frac{\sum_k^B X_{ij}^k}{B}$

We will explore values of $E(Y_{ij})$ and $Var(Y_{ij})$ to get some conclusions about the *delta*.

Let us jump to Random Picture Model and let $p_l = Pr(X_{ij}^k = l)$ then:

$p_0 = \frac{1}{256}$

$p_1 \overset{(1)}{=} p_{-1} = \frac{1-p_0}{2} = \frac{255}{512}$

$E(X_{ij}^k) = 1 \cdot p_1 + 0 \cdot p_0 + (-1) \cdot p_{-1} = 0$

$E(Y_{ij}) = 0$

$Var(Y_{ij}) = \frac{Var(\sum_k^B X_{ij}^k)}{B \cdot B} = \frac{Var(X_{ij}^1)}{B} = \frac{p_1 + p_{-1}}{B} = \frac{255}{256 \cdot B}$

Let us see how Y_{ij} behaves when input is a set of watermarked pictures with the same watermark. We have $C^k = O^k + w$ as input.

In case $w_i > w_j$ then

$$X_{ij}^k = \begin{cases} 1 & \text{, if } O_i^k + w_i > O_j^k + w_j \\ 0 & \text{, if } O_i^k + w_i = O_j^k + w_j \\ -1 & \text{, if } O_i^k + w_i < O_j^k + w_j \end{cases}$$

$$
\begin{aligned}
p_1 &= Pr(X_{ij}^k = 1) = Pr(O_i^k + w_i > O_j^k + w_j) = \\
&= Pr(O_i^k + 2 > O_j^k) = \\
&= \sum_{l=0}^{255} Pr(O_i^k + 2 > O_j^k | O_j^k = l) Pr(O_j^k = l) = \\
&= \left(1 + \sum_{i=1}^{255} \frac{257 - i}{256}\right) \cdot \frac{1}{256} = \frac{33151}{65536} \approx 0.506 \\
p_0 &= Pr(O_i^k + 2 = O_j^k) = \sum_{l=0}^{255} Pr(O_i^k + 2 = O_j^k | O_j^k = l) Pr(O_j^k = l) \\
&= \sum_{l=2}^{255} Pr(O_i^k + 2 = O_j^k | O_j^k = l) Pr(O_j^k = l) \\
&= \frac{254}{65536} \approx 0.004 \\
p_{-1} &= 1 - p_0 - p_1 = \frac{32131}{65536} \approx 0.49
\end{aligned}
$$

Thus the expected value and the variance are equal to:

$$
\begin{aligned}
E(Y_{ij}) &= E(X_{ij}^k) = p_1 - p_{-1} = \frac{255}{256} \cdot \frac{1}{64} \\
Var(Y_{ij}) &= \frac{Var(X_{ij}^k)}{B} \\
&= \frac{p_1(1 - p_1 + p_{-1})^2 + p_0(p_1 - p_{-1})^2 + p_{-1}(1 + p_1 - p_{-1})^2}{B} \\
&= \frac{p_1 + p_{-1} + (p_1 - p_{-1})^2}{B} \\
&\approx \frac{0.996}{B}
\end{aligned}
$$

From equation (2) we know $p_1 - p_{-1} = \varepsilon$ and thus

$$
E(avg_k(sgn(C_i^k - C_j^k))) = E(Y_{ij}) = \varepsilon. \tag{3}
$$

In case $w_i = w_j$ then it is like there were no watermark, so:

$$\begin{aligned} p_0 &= \frac{1}{256} \\ p_1 = p_{-1} &= \frac{255}{512} \\ E(Y_{ij}) &= 0 \\ Var(Y_{ij}) &= \frac{255}{256 \cdot B} \approx \frac{0.996}{B} \end{aligned}$$

In case $w_i < w_j$ then $Y_{ij} = -Y_{ji}$ so:

$$\begin{aligned} E(Y_{ij}) &= -\frac{255}{256} \cdot \frac{1}{64} \\ Var(Y_{ij}) &= \frac{p_1 + p_{-1} + (p_1 - p_{-1})^2}{B} \approx \frac{0.996}{B} \end{aligned}$$

Having the expected values of Y_{ij} computed we obtained a natural predicate that gives *delta*. Basically, we set:

$$delta(i,j) = \begin{cases} 2 & \text{, if } Y_{ij} < -\tau \\ 0 & \text{, if } -\tau <= Y_{ij} <= \tau \text{ , where } \tau = \frac{\varepsilon}{2} \\ -2 & \text{, if } \tau < Y_{ij} \end{cases}$$

Let us apply Chebyshev's inequality to $Pr(|Y_{ij} - E(Y_{ij})| >= \tau)$:

$$Pr(|Y_{ij} - E(Y_{ij})| \geq \tau) \leq \frac{Var(Y_{ij})}{\tau^2} \leq \frac{1}{B\tau^2} = \frac{256 \cdot 256 \cdot 128 \cdot 128}{255 \cdot 255 \cdot B},$$

so if $B > 166000$ then $Pr(|Y_{ij} - E(Y_{ij})| >= \tau) <= 0.1$ and thus 90% of *delta* is correctly predicted. The number 166000 is only an upper bound. Experiments shows that we need much less pictures to get good results (see section 7).

The way to deduce *delta* from the input data is to take $C^k = c^k$ and evaluate *delta* based on that.

6.3 Computing an approximated watermark form *delta*

In this section two algorithms are described. Both of them are heuristic, but we will see that they give good results, i.e. the fraction of correctly predicted pixels of watermark is large enough. The first algorithm is for CPU and the other one is for GPU. Both of them use *update* function, which basically should transform a current solution to a better one. The GPU version execute many updates in parallel, so we need to take care that all

threads are synchronized well.
At first, we consider relaxed problem, i.e. having *delta*, we want to determine a correspondent vector w' in $[-1,1]^{width \times height}$. Then we start by setting $w' = 0$. We perform some updates in order defined later. Having updated vector w'_i computed, we return w as an approximated watermark where:

$$w_i = \begin{cases} 1 & \text{, if } w'_i > 0 \\ -1 & \text{, otherwise} \end{cases}$$

Let us describe the *update* function. It gets adjacent pixels i and j plus a current solution w' as an input and modify provided w' so that:

- the value w'_k remains untouched for k different than i and j
- let $sum = w'_i + w'_j$ and let $delta'$ be the closest number to $delta(i,j)$ such that
  1. $abs(delta') \leq abs(delta(i,j))$
  2. $\frac{sum+delta'}{2} \in [-1,1]$
  3. $\frac{sum-delta'}{2} \in [-1,1]$

  then $w'_i = \frac{sum-delta'}{2}$ and $w'_j = \frac{sum+delta'}{2}$

Such $delta'$ always exists because $delta' = 0$ fulfills these conditions. To formalize the above, we give a pseudocode.

```
def update(w', i, j):
  sum = w'[i] + w'[j]
  delta' = max(min(delta(i, j), 2 - sum, 2 + sum),
               -2 - sum,
               -2 + sum)
  w'[i] = (sum - delta') / 2
  w'[j] = (sum + delta') / 2
```

Note that *update* procedure preserves invariant that sum of w'_i over all pixels i equals 0. It is important for GPU solution to perform updates in a such way that no pixel is updated at the same time by two or more calls.
Let us see an example of several updates on initiated $w' = 0$. We start with:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Let us pick two adjacent pixels to update:

$$\begin{bmatrix} 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Assume that $delta = 2$ for these pixels then the updated solution w' is:

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Let us pick another two adjacent pixels:

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & \mathbf{1} & 0 \\ 0 & \mathbf{0} & 0 \end{bmatrix}$$

Assume that $delta = 0$ for these pixels then:

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0.5 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$$

And so on.

For CPU we randomly pick adjacent pixels i and j to perform *update* on

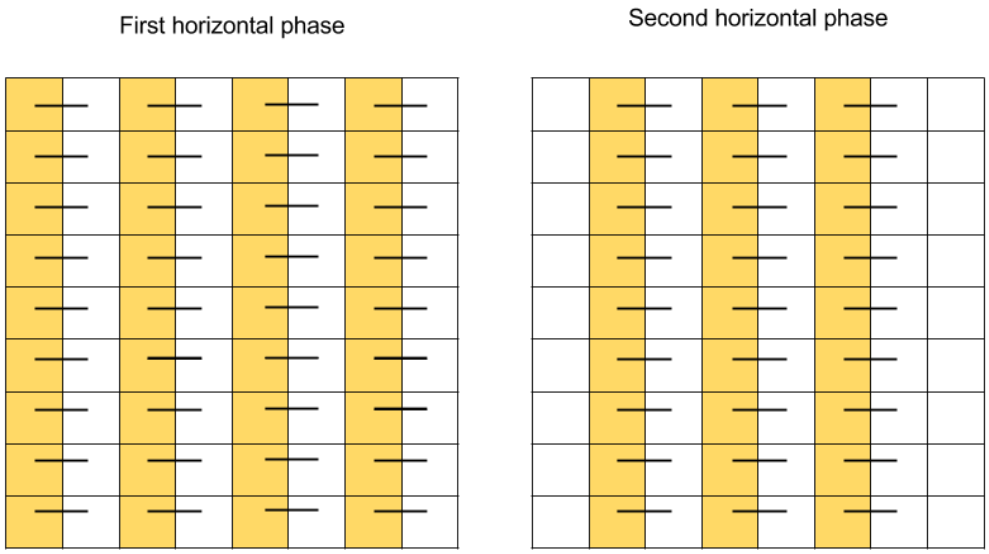


Figure 9: Horizontal phases

them.

```
for every pixel i: set w'[i] = 0
repeat (10*width*height) times:
  i, j = pick_adjacent_pixels_at_random(width, height)
  update(w', i, j)
get w from w'
```

On GPU we have a different approach. We want to make parallel *update* calls and do it in such way that in the same time no two updates would intersect. To make this happen, we call some of pixels active and the rest call inactive. Iteratively, we perform horizontal phases and vertical phases. On Figure 9 and 10 phases are shown. Yellow pixels are the active ones. On their behalf, updates are performed. In a horizontal phase, every active pixel perform an update on itself and their right neighbour. In a vertical phase, every pixel perform an update on itself and their bottom neighbour. In first horizontal phase, the first column consists of active pixels and so every second column. In second horizontal phase active pixels are these, which were inactive in the first phase. If a number of columns is odd then pixels from the last column are inactive in both horizontal phases. For vertical phases, active pixels are grouped similarly but in rows. In the following code *get_id_*∗

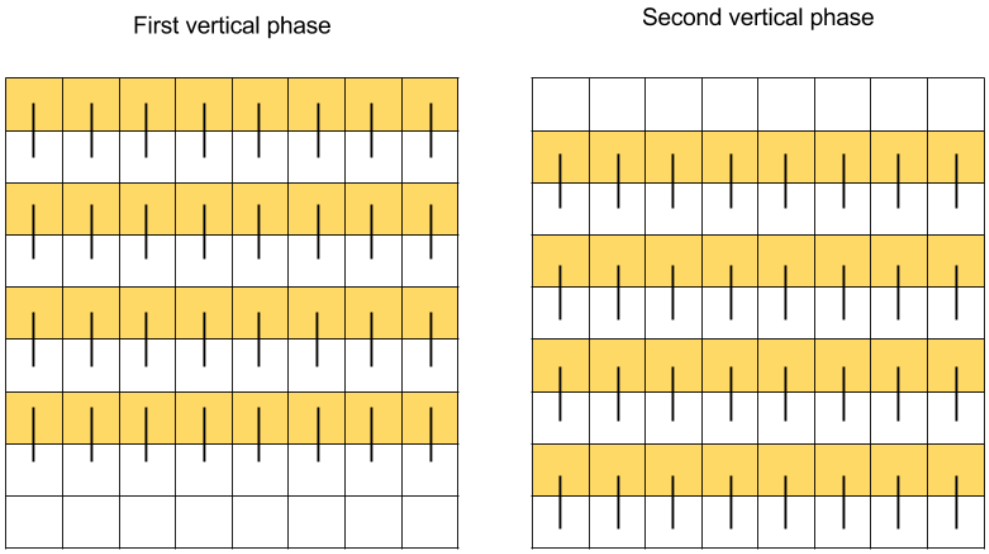


Figure 10: Vertical phases

returns the id of active pixel for a thread. Methods *me_and_*[*right*|*bottom*] take the first pixel and return a pair of pixels: the one provided and right or bottom one, respectively.

```
for every pixel i: set w'[i] = 0
repeat 10 times:
  parallel update(w', me_and_right(get_id_horizontal_1st()))
  parallel update(w', me_and_bottom(get_id_vertical_1st()))
  parallel update(w', me_and_right(get_id_horizontal_2nd()))
  parallel update(w', me_and_bottom(get_id_vertical_2nd()))
get w from w'
```

6.4 Performed test

Input: A set of 636 pictures took mostly by GoPro Hero camera and Samsung Galaxy Nexus phone.
Description: The same watermark was embedded into all photos. The CPU version of breaking algorithm computed an approximation of the underlying watermark.
Result: **77,5%** correctly predicted pixels.
Figure 11 shows how the breaking algorithm managed to predict pixels of the embedded watermark. The green dots denote pixels predicted correctly and the red dots denote the ones predicted wrongly.

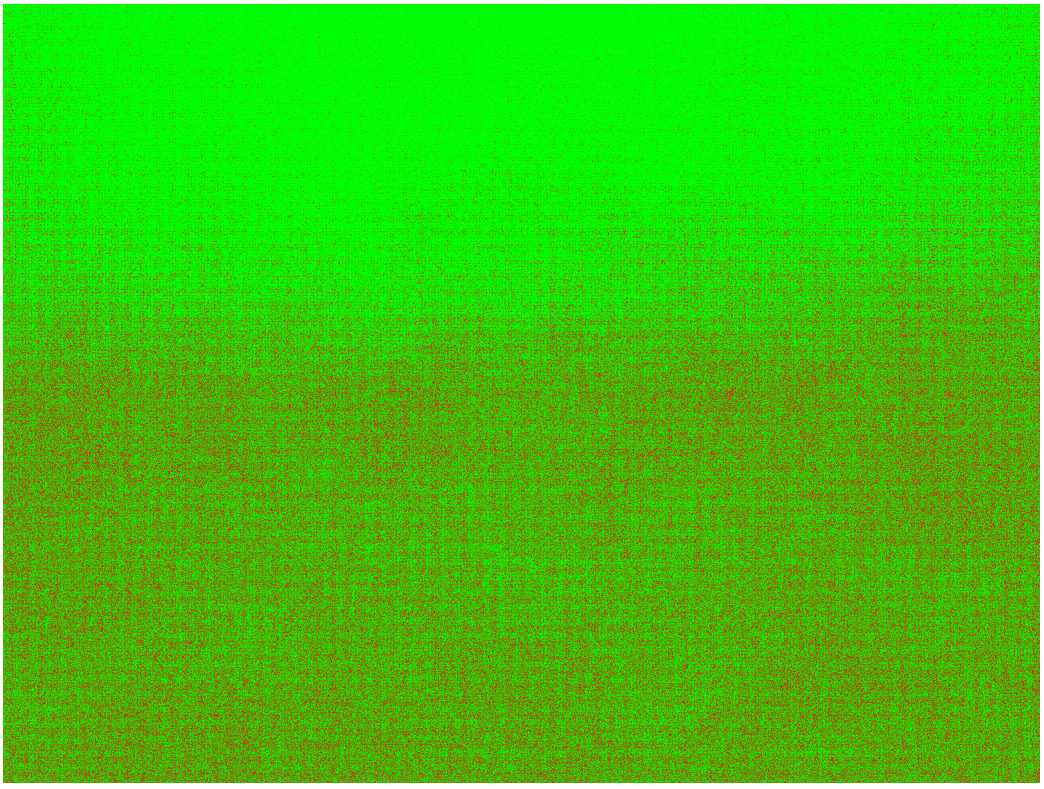


Figure 11: Correctly predicted pixels - green, wrongly predicted pixels - red

It turns out we can do something better. The things, which might have gone wrong are:

- Wrongly predicted watermark from *delta*.
- Too small set of pictures. 636 instead of 166000.

- We applied solution for Random Picture Model to Natural Picture Model

Could the watermark be predicted wrongly from *delta*? To verify this another test was performed in which we run the breaking algorithm on a set of images that contained exactly one picture, which was a watermark itself. The solution predicted less that 1% of pixels correctly, which is very good result (In case the algorithm predicts 50% of pixels correctly then it means that if works randomly, 0% and 100% correctness are expected mostly). So first dot is not a case.
Could the set be too small. Yes, it could. However, it is hard to test the solution on a larger set. The CPU solution is executing almost an hour for 636 pictures and current implementation of GPU solution requires more device memory. More pictures - more resources needed to verify. So the second dot might be the case, but we won't verify it.

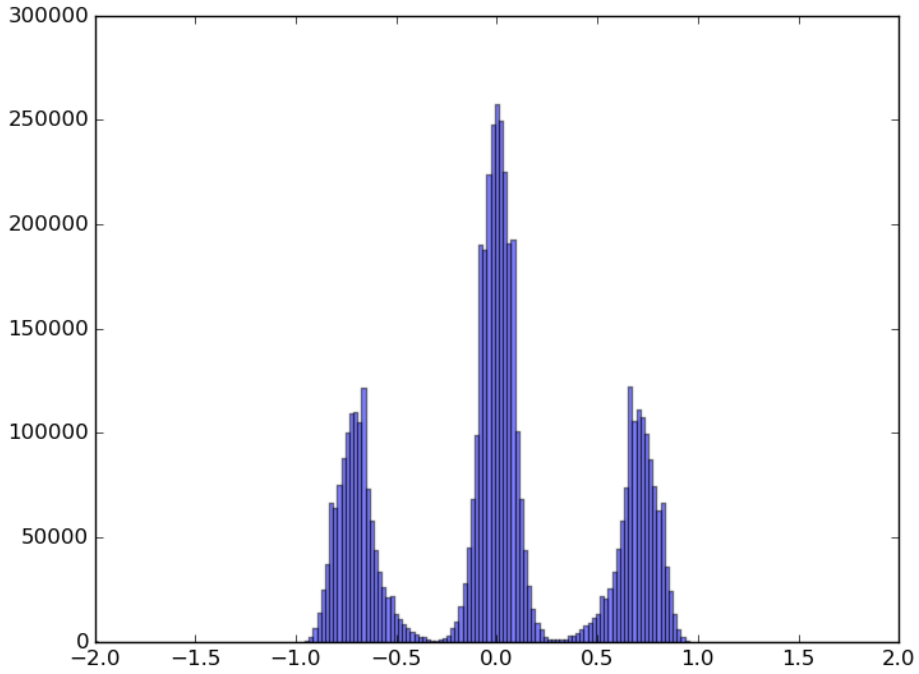


Figure 12: Histogram of horizontal Y_{ij} values

For the third dot the answer is: yes, it is the case. To understand that imagine how a histogram for Y_{ij} would look like. It should have 3 peaks around

$\frac{1}{64}$, 0 and $-\frac{1}{64}$. The first peak comes from pixels i and j where $w_i > w_j$ (from equation (3) we have $E(Y_{ij}) = \varepsilon \approx \frac{1}{64}$), the second is when $w_i = w_j$ and the last when $w_i < w_j$. Let us take a look on a histogram of Y_{ij} horizontal values that was generated on a real data – the set of 636 images. It is presented on Figure 12 We see the peaks around -0.75, 0, 0.75. The peaks are further than in Random Picture Model. Setting τ to 0.3 (used in *delta* definition) we are getting result **99,6%** correctly predicted pixels of an embedded watermark while running on the set of 636 photos. I hope that the location of the peaks could be understood by considering the histogram of $|C_i^k - C_j^k|$ which looks exponentially, while for Random Picture Model, a distribution is a triangle.
Figure 13 presents the histogram of values $(C_i^k - C_j^k)$ aggregated over all possible pixels i and j, and 4 images' indices picked randomly as k.

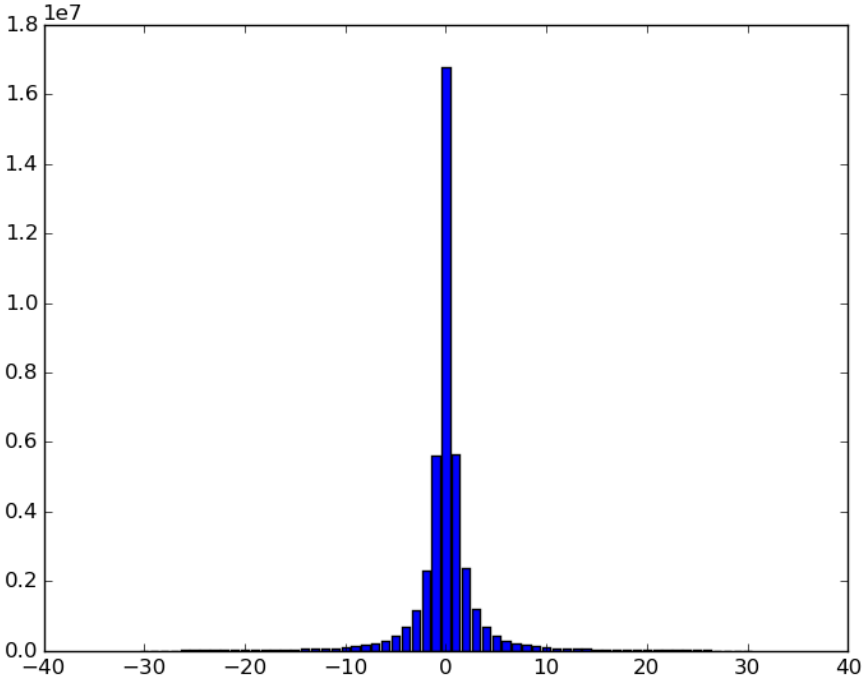


Figure 13: Histogram of $C_i^k - C_j^k$

Note that if we make similar histogram for watermarked bitmaps then it looks quite different what is presented on Figure 14. Thus the shape analysis of this histogram might give a suspicion that inside a set of images there is a watermark embedded with E_BLIND method.

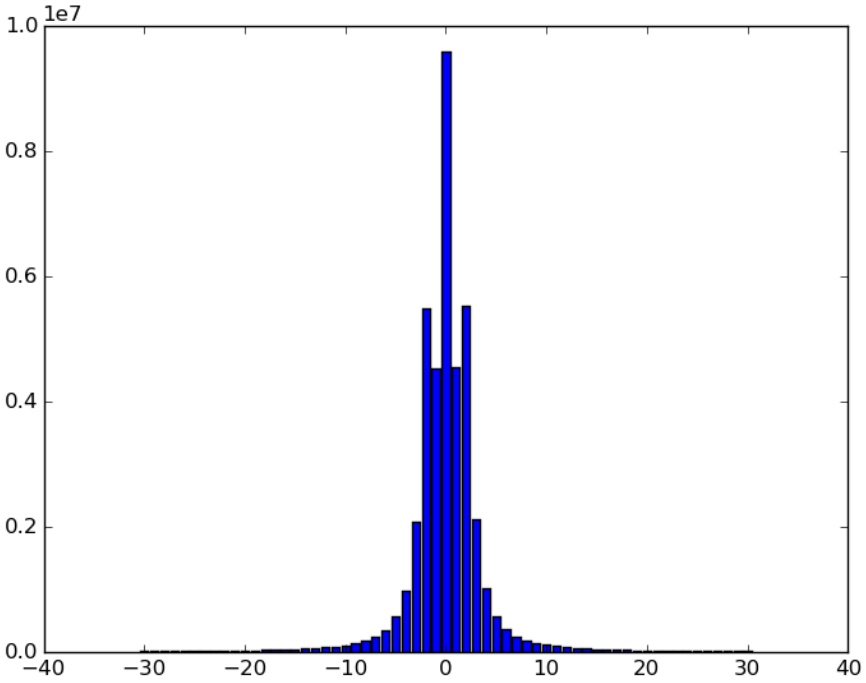


Figure 14: Histogram of $C_i^k + w_i - C_j^k + w_j$

7 Experimental convergence speed of the breaking algorithm

The breaking algorithm was run for every combination of indices *env*, B and *times* on sets of watermarked images. The possible values for indices are described by:

1. Index $env \in \{\text{CPU}, \text{GPU}\}$
2. Index $B \in \{1, 5, 9, 13, 17, 21\}$
3. Index $times \in \{1, 2, 3\}$ in case of $env = \text{CPU}$ or $times \in \{1, 2, 3, 4, 5\}$ in case of $env = \text{GPU}$

Index B denotes that we run the breaking algorithm on a set of cardinality B. Index *times* denotes number of a sample set of cardinality B that is run on *env*. We ran the breaking algorithm and computed linear correlations, which then we averaged over all *times*. So for every *env* and B we got average

linear correlation. The results was displayed on Figure 15 from left to right, sorted by B.

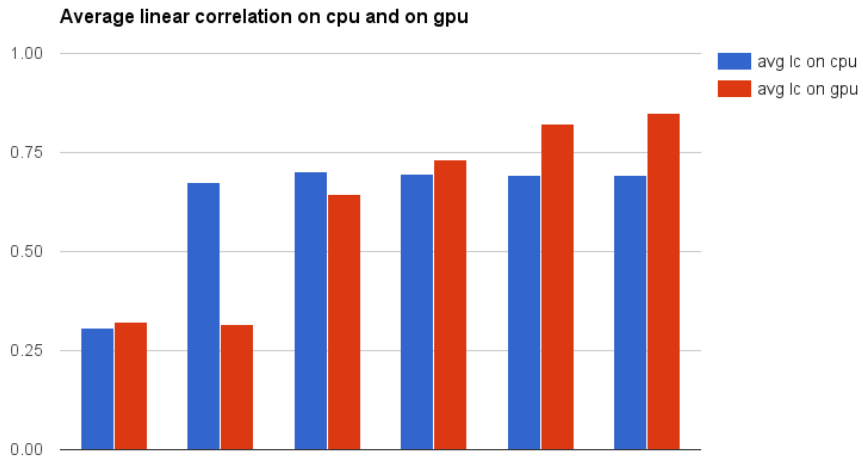


Figure 15: lc-convergence speed on CPU and GPU.

8 Running code

Generating random watermark:

```
python generate_watermark.py -o watermark
```

Watermarking pictures with E_BLIND:

```
python watermark_pictures.py\
  --in=photos\
  --out=watermarked\
  --watermark=watermark.bmp\
  --usecuda=true
```

Computing linear correlation of multiple files against a watermark:

```
python compute_linear_correlation.py\
  --in=watermarked\
  --reference=watermark.bmp
```

Finding a watermark embedded in multiple digital works:

```
python break_adj.py\
  --watermarked=watermarked/\
  --deduced=deduced.bmp\
  --size=2592x1944\
  --usecuda=true
```

Generating histogram for differences between adjacent pixels:

```
python diffenerce_histogram.py\
  --in=photos/\
  --rangeradius=30
```

9 Execution speed comparision between CPU and GPU

A test was performed to compare execution speed of CPU and GPU solutions. We used *Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz* for CPU and *GeForce GTX 980* for GPU. There were 100 runs of both versions. The

Figure 16: Comparison of CPU and GPU execution times.

real time of execution is presented on Figure 16. Every run had to process B images, where B was taken from 1 to 100. The times on chart are sorted by B and displayed from left to right. Fluctuations for CPU solutions might

come from the fact that the solution is using multi processing and the processes have to synchronized between them, which works in nondeterministic time.

10 Problems

1. File format: If you take a content and watermark them using E_BLIND and then you save them as JPG then it is more likely that your watermark won't survive a compression and will not be visible anymore. So when I use E_BLIND, I save the watermarked content in BMP.

2. Analysis of breaking algorithm assumes that $C'^k = C^k + w$. In fact, $C'^k = max(0, min(255, C^k + w))$. It should be verified that $Pr(C'^k = max(0, min(255, C^k + w))! = C^k + w)$ in Random Picture Model.

3. Understand why peaks and antipeaks are in -0.75, -0.3, 0, 0.3, 0.75 when considering Natural Picture Model.

4. The tests for breaking E_BLIND accuracy were performed only on one watermark. They should be performed on many such watermarks and we should claim the averaged accuracy.

5. Verify if
$$Pr(C_i > C_j) = Pr(C_i < C_j)$$
is true for Natural Picture Model and if not then approximate the error.

6. Explore and prove the relation between percentage of *delta* predicted correctly and a lower bound on percentage of correctly predicted pixels in an underlying watermark.

11 Epilogue

You can make E_BLIND resistant from the presented attack if you have a set of watermarks and you always pick a watermark at random before watermarking any single picture. In case you are watermarking movie, you should pick a watermark at random for each frame.

References

[1] Christopher J. Biermann. Handbook of pulping and papermaking. *Algorithmica*, page 171, 1996.

[2] Ingemar J. Cox, Matthew L. Miller, Jeffrey A. Bloom, Jessica Fridrich, and Ton Kalker. *Digital Watermarking and Steganography*. Morgan Kaufmann publications, 2008.

[3] Wikipedia Society. Watermark. https://en.wikipedia.org/w/index.php?title=Watermark&oldid=737205277, September 2016. [Online; accessed 1-September-2016].

Supplement - sources

All the codes for this paper can be found on: HTTPS://GITHUB.COM/WANATPJ/H_BLIND The name of the repository H_BLIND means hacking blind embedding - E_BLIND.

File	Description
break_adj.py	Breaking E_BLIND algorithm. Deduces the watermark, given a set of images containing the same watermark embedded with E_BLIND method.
common.py	Utils
compute_linear_correlation.py	Computes linear correlation between a watermark and some image or a set of image. In case of set, it returns average linear correlation.
cudalib.py	Implementations of GPU kernels and processes that run the kernels
difference_histogram.py	Computes the histogram of $C_i^k - C_j^k$
generate_watermark.py	Generates a random watermark
watermark_pictures.py	Watermarks pictures with a certain watermark using E_BLIND embedder.

Initially *delta* was called *edge model*, so whenever you see *edge model* in the codes, please think of *delta*.

break_adj.py

```
import numpy
import os
import random
from matplotlib import pyplot
from multiprocessing import Pool
from optparse import OptionParser
from PIL import Image
from scipy import misc

from common import *

def _prepare_reduction(size):
  wwidth, wheight = size
  return numpy.zeros((wwidth - 1, wheight)), numpy.zeros((wwidth, wheight - 1))

def delta(pixel1, pixel2):
  return 1 if pixel1 < pixel2 else 0 if pixel1 == pixel2 else -1

def extract_delta(f):
  with Image.open(f) as image:
    width, height = image.size
    loaded = numpy.transpose(
        numpy.array(image.convert("L").getdata())
            .reshape((height, width)))
    horizontal = numpy.zeros((width - 1, height))
    vertical = numpy.zeros((width, height - 1))
    for x in range(width - 1):
      for y in range(height):
        horizontal[x][y] = delta(loaded[x][y], loaded[x + 1][y])
    for x in range(width):
      for y in range(height - 1):
        vertical[x][y] = delta(loaded[x][y], loaded[x][y + 1])
    return horizontal, vertical

def matadd(x, y):
  h1, v1 = x
  h2, v2 = y
  return numpy.add(h1, h2), numpy.add(v1, v2)

def get_random_edge(size):
  width, height = size
  if random.randint(0, 1) == 0:
    return {
        'type': 0,
        'x': random.randint(0, width - 2),
        'y': random.randint(0, height - 1) }
  else:
```

```
    return {
        'type': 1,
        'x': random.randint(0, width - 1),
        'y': random.randint(0, height - 2) }

def update(watermark, delta, x1, y1, x2, y2):
  _sum = watermark[x1][y1] + watermark[x2][y2]
  delta = min(delta, 2 - _sum)
  delta = min(delta, 2 + _sum)
  delta = max(delta, -2 - _sum)
  delta = max(delta, -2 + _sum)
  watermark[x1][y1] = (_sum - delta)/2.
  watermark[x2][y2] = (_sum + delta)/2.

def _parse_flags():
  global watermarked, deduced, size, usecuda
  parser = OptionParser()
  parser.add_option("-i",
      "--watermarked",
      dest="watermarked",
      help="location to directory that containes potentialy watermarked "\
          + "images",
      metavar="DIR")
  parser.add_option("-o",
      "--deduced",
      dest="deduced",
      help="a file where to save watermarked images",
      metavar="FILE")
  parser.add_option("-s",
      "--size",
      dest="size",
      help="width x height , par example: 2592x1944",
      metavar="SIZE")
  parser.add_option("-c",
      "--usecuda",
      dest="usecuda",
      help="true iff should use gpu compiting (cuda)",
      metavar="true")
  (options, args) = parser.parse_args()
  watermarked = options.watermarked
  deduced = options.deduced
  size = tuple(map(lambda x: int(x), options.size.split('x')))
  usecuda = True\
      if options.usecuda != None and options.usecuda.lower()[0] == 't'\
      else False

def deduce_edge_model(files, size):
  horizontal, vertical =\
      map_reduce(files, extract_delta, matadd, _prepare_reduction(size))
```

```
  numpy.divide(horizontal, float(len(files)))
  numpy.divide(vertical, float(len(files)))
  guess_edge_fn = numpy.vectorize(
      lambda x: -1 if x < -0.3 else 0 if x <= 0.3 else 1)
  horizontal = guess_edge_fn(horizontal)
  vertical = guess_edge_fn(vertical)
  return horizontal, vertical

def deduce_watermark(size, edge_model):
  width, height = size
  watermark = numpy.zeros(size)
  horizontal, vertical = edge_model
  iterations = 10 * width * height
  while iterations > 0:
    if not (iterations & 0b1111111111111111):
      print iterations
    edge = get_random_edge(size)
    if edge['type'] == 0:
      update(watermark,
          2*horizontal[edge['x']][edge['y']],\
          edge['x'],\
          edge['y'],\
          edge['x'] + 1,\
          edge['y'])
    else:
      update(watermark,
          2*vertical[edge['x']][edge['y']],\
          edge['x'],\
          edge['y'],\
          edge['x'],\
          edge['y'] + 1)
    iterations -= 1
  return numpy\
      .array([watermark[x][y] for y in range(height) for x in range(width)])

def save_hidden_watermark(hidden_watermark):
  global deduced, size
  width, height = size
  hidden_watermark_stream =\
      [chr(255) if val < 0 else chr(0) for val in hidden_watermark]
  Image.frombytes('L', (width, height), "".join(hidden_watermark_stream))\
      .convert('1')\
      .save(deduced)

def main():
  global watermarked, deduced, size, usecuda
  _parse_flags()
  files = filter(
      ImageSizeFilter(size, watermarked),
```

```
        os.listdir(watermarked))
    make_exact_path_fn = numpy.vectorize(lambda x: watermarked + "/" + x)
    files = make_exact_path_fn(files)
    if usecuda:
      import cudalib
      hidden_watermark = cudalib.deduce_watermark(size, files)
    else:
      edge_model = deduce_edge_model(files, size)
      hidden_watermark = deduce_watermark(size, edge_model)
    save_hidden_watermark(hidden_watermark)

main()
```

common.py

```
import numpy
from multiprocessing import cpu_count, Pool
from PIL import Image

class ImageSizeFilter:
  def __init__(self, size, indir):
    self.size = size
    self.indir = indir
  def __call__(self, f):
    with Image.open(self.indir + "/" + f) as image:
      return image.size == self.size

def get_pool():
  return Pool(cpu_count() + 2)

def get_watermark(f):
  watermark_image = Image.open(f)
  return watermark_image.size,\
      numpy.array([1 if y == 0 else -1 for y in watermark_image.getdata()])\
          .astype(numpy.int8)

def get_image_in_grayscale(f):
  with Image.open(f) as image:
    return numpy.array(image.convert("L").getdata()).astype(numpy.uint8)

def linear_correlation(a, b):
  N = a.size
  if N != b.size:
    raise Exception('VECTORS_WITH_DIFFERENT_LENGTH')
  correlation = 0.
  for i in range(N):
    correlation += a[i]*float(b[i])
  return correlation / float(N)

def map_reduce(data, map_fn, reduce_fn, reduced, chunk_size = 8, sync = False):
  while data.size != 0:
    mapped = map(map_fn, data[0 : chunk_size])\
        if sync else get_pool().map(map_fn, data[0 : chunk_size])
    reduced = reduce(\
        reduce_fn,\
        mapped,\
        reduced)
    data = data[chunk_size : ]
    print "Remaining elements for map-reduce: " + str(len(data))
  return reduced
```

compute_linear_correlation.py

```
import os
import numpy
from matplotlib import pyplot
from optparse import OptionParser
from PIL import Image

from common import *

def _parse_flags():
  global indir, inreferencefile, referencefile, versuswatermark
  parser = OptionParser()
  parser.add_option("-i",
      "--in",
      dest="indir",
      help="location to directory that containes images to watermark",
      metavar="DIR")
  parser.add_option("-v",
      "--inreference",
      dest="inreference",
      help="monochrome image; black pixel denotes 1, white pixel denotes -1",
      metavar="FILE")
  parser.add_option("-w",
      "--reference",
      dest="reference",
      help="monochrome image; black pixel denotes 1, white pixel denotes -1",
      metavar="FILE")
  (options, args) = parser.parse_args()
  if (options.indir != None and options.inreference != None) \
      or (options.indir == None and options.inreference == None):
    raise Exception("define: indir xor inreference")
  if options.indir != None:
    indir = options.indir
    versuswatermark = False
  elif options.inreference != None:
    inreferencefile = options.inreference
    versuswatermark = True
  referencefile = options.reference

class ComputeLinearCorrelation:
  def __init__(self, reference, indir):
    self.reference = reference
    self.indir = indir
  def __call__(self, f):
    return linear_correlation(
        numpy.array(Image.open(self.indir + "/" + f).convert("L").getdata()),
        self.reference)
```

```
def main():
  global indir, inreferencefile, referencefile, versuswatermark
  _parse_flags()
  (width, height), reference = get_watermark(referencefile)
  if versuswatermark:
    (inwidth, inheight), inreference = get_watermark(inreferencefile)
    lcs = [linear_correlation(reference, inreference)]
  else:
    lcs = get_pool().map(
        ComputeLinearCorrelation(reference, indir),
        filter(ImageSizeFilter((width, height), indir), os.listdir(indir)))
  print "Mean: " + str(numpy.mean(lcs))
  print "Median: " + str(numpy.median(lcs))
  print "Variance: " + str(numpy.var(lcs))
  bins = numpy.linspace(-2, 2, 200)
  pyplot.hist(lcs, bins, alpha=0.5, label='lc')
  pyplot.show()

main()
```

cudalib.py

```
import numpy
import pycuda
import pycuda.autoinit
import pycuda.driver as cuda
from PIL import Image
from pycuda.compiler import SourceModule
from pycuda.elementwise import ElementwiseKernel
from pycuda.reduction import ReductionKernel

from common import *

module = SourceModule("""
  __global__ void watermark_rgb_content_kernel(
      unsigned char* watermarked_content,
      unsigned char* content,
      char* watermark,
      int shift) {
    int i = shift + threadIdx.x + blockDim.x*blockIdx.x;
    watermarked_content[i] =
        (unsigned char) min(255, max(0, content[i] + (int)watermark[i/3]));
  }

  __device__ inline void update(
      float* watermark,
      float delta,
      int id1,
      int id2) {
    float _sum = watermark[id1] + watermark[id2];
    delta = min(delta, 2 - _sum);
    delta = min(delta, 2 + _sum);
    delta = max(delta, -2 - _sum);
    delta = max(delta, -2 + _sum);
    watermark[id1] = (_sum - delta)/2.;
    watermark[id2] = (_sum + delta)/2.;
  }

  __global__ void reinforce_watermark_image_horizontally_kernel(
      int width,
      float* horizontal,
      float* watermark,
      int shift,
      int shift2) {
    int id = threadIdx.x + blockDim.x*blockIdx.x + shift2;
    int perrow = (width - shift) / 2;
    int i = id / perrow;
    int j = ((id - i*perrow) * 2) + shift;
    update(watermark,
```

```
      2*horizontal[i*(width - 1) + j],
      i*width + j,
      i*width + j + 1);
}

__global__ void reinforce_watermark_image_vertically_kernel(
    int width,
    float* vertical,
    float* watermark,
    int shift,
    int shift2) {
  int id = threadIdx.x + blockDim.x*blockIdx.x + shift2;
  int i = id / width;
  int j = id - i*width;
  i = 2*i + shift;
  update(watermark,
      2*vertical[i*width + j],
      i*width + j,
      (i + 1)*width + j);
}

__device__ inline int sgn(int x) {
  if (x < 0) {
    return -1;
  } else if (x == 0) {
    return 0;
  }
  return 1;
}

__global__ void extract_horizontal_delta(
    int width,
    unsigned char* content,
    float* horizontal,
    int shift) {
  int id = threadIdx.x + blockDim.x*blockIdx.x + shift;
  int i = id / (width - 1);
  // int j = id - i*(width - 1);
  // i*width + j == id + i
  horizontal[id] = sgn (content[id + i + 1] - (int)content[id + i]);
}

__global__ void extract_vertical_delta(
    int width,
    unsigned char* content,
    float* vertical,
    int shift) {
  int id = threadIdx.x + blockDim.x*blockIdx.x + shift;
  vertical[id] = sgn (content[id + width] - (int)content[id]);
```

```
  }

  __global__ void sum_reduce_step(
      float* reduced,
      float* partial,
      int shift){
    int id = threadIdx.x + blockDim.x*blockIdx.x + shift;
    reduced[id] += partial[id];
  }

  __global__ void normalize_and_prepare_model(
      float* reduced,
      int imagesCount,
      int shift) {
    int id = threadIdx.x + blockDim.x*blockIdx.x + shift;
    float val = reduced[id] / imagesCount;
    reduced[id] = (val < -0.3) ? -1 : ((val <= 0.3) ? 0 : 1);
  }
  """)

watermark_rgb_content_kernel =\
    module.get_function("watermark_rgb_content_kernel")
reinforce_watermark_image_horizontally_kernel =\
    module.get_function("reinforce_watermark_image_horizontally_kernel")
reinforce_watermark_image_vertically_kernel =\
    module.get_function("reinforce_watermark_image_vertically_kernel")
extract_horizontal_delta_kernel =\
    module.get_function("extract_horizontal_delta")
extract_vertical_delta_kernel =\
    module.get_function("extract_vertical_delta")
sum_reduce_step_kernel = module.get_function("sum_reduce_step")
normalize_and_prepare_model_kernel =\
    module.get_function("normalize_and_prepare_model")

def prepare_gpu_array(array):
  gpu_array = cuda.mem_alloc(array.nbytes)
  cuda.memcpy_htod(gpu_array, array)
  return gpu_array

class ExtractDeltaMapper:
  def __init__(self, size):
    self.width, self.height = size
  def __call__(self, gpu_content):
    horizontal_computations = (self.width - 1)*self.height
    vertical_computations = self.width*(self.height - 1)
    gpu_horizontal_sgn = cuda.mem_alloc(4*horizontal_computations)
    gpu_vertical_sgn = cuda.mem_alloc(4*vertical_computations)
    extract_horizontal_delta_kernel(
        numpy.int32(self.width),
```

```
            gpu_content,
            gpu_horizontal_sgn,
            numpy.int32(0),
            block=(1024, 1, 1),
            grid=(horizontal_computations/1024, 1))
        if horizontal_computations & 1023:
          extract_horizontal_delta_kernel(
              numpy.int32(self.width),
              gpu_content,
              gpu_horizontal_sgn,
              numpy.int32(horizontal_computations/1024*1024),
              block=(horizontal_computations & 1023, 1, 1),
              grid=(1, 1))
        extract_vertical_delta_kernel(
            numpy.int32(self.width),
            gpu_content,
            gpu_vertical_sgn,
            numpy.int32(0),
            block=(1024, 1, 1),
            grid=(vertical_computations/1024, 1))
        if vertical_computations & 1023:
          extract_vertical_delta_kernel(
              numpy.int32(self.width),
              gpu_content,
              gpu_vertical_sgn,
              numpy.int32(vertical_computations/1024*1024),
              block=(vertical_computations & 1023, 1, 1),
              grid=(1, 1))
        return gpu_horizontal_sgn, gpu_vertical_sgn

    class BiMatrixSumReduction:
      def __init__(self, size):
        self.width, self.height = size
      def __call__(self,
          (gpu_horizontal_reduced, gpu_vertical_reduced),
          (gpu_horizontal_sgn, gpu_vertical_sgn)):
        horizontal_computations = (self.width - 1)*self.height
        vertical_computations = self.width*(self.height - 1)
        sum_reduce_step_kernel(
            gpu_horizontal_reduced,
            gpu_horizontal_sgn,
            numpy.int32(0),
            block=(1024, 1, 1),
            grid=(horizontal_computations/1024, 1))
        if horizontal_computations & 1023:
          sum_reduce_step_kernel(
              gpu_horizontal_reduced,
              gpu_horizontal_sgn,
              numpy.int32(horizontal_computations/1024*1024),
```

```
            block=(horizontal_computations & 1023, 1, 1),
            grid=(1, 1))
        sum_reduce_step_kernel(
            gpu_vertical_reduced,
            gpu_vertical_sgn,
            numpy.int32(0),
            block=(1024, 1, 1),
            grid=(vertical_computations/1024, 1))
        if vertical_computations & 1023:
            sum_reduce_step_kernel(
                gpu_vertical_reduced,
                gpu_vertical_sgn,
                numpy.int32(vertical_computations/1024*1024),
                block=(vertical_computations & 1023, 1, 1),
                grid=(1, 1))
        return gpu_horizontal_reduced, gpu_vertical_reduced


def watermark_content(infile, outfile, gpu_watermark):
    rgb_image = Image.open(infile).convert("RGB")
    cpu_rgb_image = numpy.array(rgb_image).flatten()
    if cpu_rgb_image.nbytes < 1024:
        raise Exception("naughty: too small file to make parallel computing")
    cpu_rgb_outimage = numpy.empty_like(cpu_rgb_image)
    gpu_rgb_image = prepare_gpu_array(cpu_rgb_image)
    gpu_rgb_outimage = cuda.mem_alloc(cpu_rgb_image.nbytes)
    cuda.Context.synchronize()
    watermark_rgb_content_kernel(
        gpu_rgb_outimage,
        gpu_rgb_image,
        gpu_watermark,
        numpy.int32(0),
        block=(1024, 1, 1),
        grid=(cpu_rgb_image.nbytes / 1024, 1))
    cuda.Context.synchronize()
    if cpu_rgb_image.nbytes & 1023:
        watermark_rgb_content_kernel(
            gpu_rgb_outimage,
            gpu_rgb_image,
            gpu_watermark,
            numpy.int32(cpu_rgb_image.nbytes/1024*1024),
            block=(cpu_rgb_image.nbytes % 1024, 1, 1),
            grid=(1, 1))
        cuda.Context.synchronize()
    cuda.memcpy_dtoh(cpu_rgb_outimage, gpu_rgb_outimage)
    cuda.Context.synchronize()
    Image.fromarray(cpu_rgb_outimage
        .reshape(tuple(reversed(rgb_image.size)) + (3,))).save(outfile)
    cuda.Context.synchronize()
```

```
def deduce_watermark_from_model(size, gpu_edge_model):
  gpu_horizontal, gpu_vertical = gpu_edge_model
  width, height = size
  cpu_watermark = numpy.zeros(width * height).astype(numpy.float32)
  gpu_watermark = prepare_gpu_array(cpu_watermark)
  for i in range(10):
    for shift in range(2):
      horizontal_computations = ((height - shift) / 2) * width
      vertical_computations = height * ((width - shift) / 2)
      reinforce_watermark_image_horizontally_kernel(
          numpy.int32(width),
          gpu_horizontal,
          gpu_watermark,
          numpy.int32(shift),
          numpy.int32(0),
          block=(1024, 1, 1),
          grid=(horizontal_computations/1024, 1))
      cuda.Context.synchronize()
      if horizontal_computations % 1024 > 0:
        reinforce_watermark_image_horizontally_kernel(
            numpy.int32(width),
            gpu_horizontal,
            gpu_watermark,
            numpy.int32(shift),
            numpy.int32(horizontal_computations/1024*1024),
            block=(horizontal_computations % 1024, 1, 1),
            grid=(1, 1))
      cuda.Context.synchronize()
      reinforce_watermark_image_vertically_kernel(
          numpy.int32(width),
          gpu_vertical,
          gpu_watermark,
          numpy.int32(shift),
          numpy.int32(0),
          block=(1024, 1, 1),
          grid=(vertical_computations/1024, 1))
      cuda.Context.synchronize()
      if vertical_computations % 1024 > 0:
        reinforce_watermark_image_vertically_kernel(
            numpy.int32(width),
            gpu_vertical,
            gpu_watermark,
            numpy.int32(shift),
            numpy.int32(vertical_computations/1024*1024),
            block=(vertical_computations % 1024, 1, 1),
            grid=(1, 1))
        cuda.Context.synchronize()
  cuda.memcpy_dtoh(cpu_watermark, gpu_watermark)
```

```
  cuda.Context.synchronize()
  return cpu_watermark

def deduce_watermark(size, infiles):
  width, height = size
  N = width * height
  B = len(infiles)
  gpu_images = numpy.array(
      map(lambda x: prepare_gpu_array(get_image_in_grayscale(x)), infiles))
  gpu_horizontal =\
      prepare_gpu_array(numpy.zeros((width - 1) * height).astype(numpy.float32))
  gpu_vertical =\
      prepare_gpu_array(numpy.zeros(width * (height - 1)).astype(numpy.float32))
  map_reduce(gpu_images,
      ExtractDeltaMapper(size),
      BiMatrixSumReduction(size),
      (gpu_horizontal, gpu_vertical),
      chunk_size = 1,
      sync = True)
  cuda.Context.synchronize()
  normalize_and_prepare_model_kernel(
      gpu_horizontal,
      numpy.int32(B),
      numpy.int32(0),
      block=(1024, 1, 1),
      grid=(N/1024, 1)
  )
  if N & 1023:
    normalize_and_prepare_model_kernel(
        gpu_horizontal,
        numpy.int32(B),
        numpy.int32(N/1024*1024),
        block=(N % 1024, 1, 1),
        grid=(1, 1)
    )
  normalize_and_prepare_model_kernel(
      gpu_vertical,
      numpy.int32(B),
      numpy.int32(0),
      block=(1024, 1, 1),
      grid=(N/1024, 1)
  )
  if N & 1023:
    normalize_and_prepare_model_kernel(
        gpu_vertical,
        numpy.int32(B),
        numpy.int32(N/1024*1024),
        block=(N % 1024, 1, 1),
        grid=(1, 1)
```

```
    )
cuda.Context.synchronize()
return deduce_watermark_from_model(size, (gpu_horizontal, gpu_vertical))
```

difference_histogram.py

```
import Image
import matplotlib.pyplot as plt
import numpy
import os
from optparse import OptionParser

from common import *

def _parse_flags():
  global indir, rangeradius
  parser = OptionParser()
  parser.add_option("-i",
      "--in",
      dest="indir",
      help="directory that containes images for which the difference histogram" \
          + " will be computed",
      metavar="DIR")
  parser.add_option("-r",
      "--rangeradius",
      dest="rangeradius",
      help="range of the histogram",
      metavar="NUMBER")
  (options, args) = parser.parse_args()
  if not options.indir or not options.rangeradius:
    parser.error('Not all flags specified; run with --help to see the flags;')
  indir = options.indir
  rangeradius = int(options.rangeradius)

def extract_differences(f):
  with Image.open(f) as image:
    width, height = image.size
    result = []
    img = image.convert("L").load()
    for x in range(width - 1):
      for y in range(height):
        result.append(img[x, y] - img[x + 1, y])
    for x in range(width):
      for y in range(height - 1):
        result.append(img[x, y] - img[x, y + 1])
  return result

def histogram_reduce(histogram, values):
  for value in values:
    histogram[value + 255] += 1
  return histogram

def main():
```

```
    global indir, rangeradius
    _parse_flags()
    normalize_file_names_fn = numpy.vectorize(lambda x: indir + "/" + x)
    result = map_reduce(normalize_file_names_fn(os.listdir(indir)),\
        extract_differences,\
        histogram_reduce,
        numpy.zeros(256 + 255, dtype=numpy.uint64))
    plt.bar(numpy.arange(-rangeradius, rangeradius + 1),
        result[255 - rangeradius : 255 + rangeradius + 1],
        align='center')
    plt.show()

main()
```

generate_watermark.py

```
import os
from optparse import OptionParser
from PIL import Image

def _parse_flags():
  global out
  parser = OptionParser()
  parser.add_option("-o",
      "--out",
      dest="out",
      help="a file name without extension where to save the new watermark",
      metavar="FILE")
  (options, args) = parser.parse_args()
  out = options.out

def main():
  global out
  _parse_flags()
  Image.frombytes('1', (2592, 1944), os.urandom(1944 * 2592 / 8))\
      .save(out + ".bmp")

main()
```

watermark_pictures.py

```
import numpy as np
import os
from optparse import OptionParser
from PIL import Image

from common import ImageSizeFilter, get_pool, get_watermark

def _parse_flags():
  global indir, outdir, watermarkfile, alpha, usecuda
  parser = OptionParser()
  parser.add_option("-i",
      "--in",
      dest="indir",
      help="location to directory that containes images to watermark",
      metavar="DIR")
  parser.add_option("-o",
      "--out",
      dest="outdir",
      help="location to directory where to save watermarked images",
      metavar="DIR")
  parser.add_option("-w",
      "--watermark",
      dest="watermark",
      help="watermark file",
      metavar="FILE")
  parser.add_option("-a",
      "--alpha",
      dest="alpha",
      help="magnitude multiplier for watermark, by default = 1",
      metavar="INTEGER")
  parser.add_option("-c",
      "--usecuda",
      dest="usecuda",
      help="true iff should use gpu compiting (cuda)",
      metavar="true")
  (options, args) = parser.parse_args()
  indir = options.indir
  outdir = options.outdir
  watermarkfile = options.watermark
  alpha = 1 if options.alpha == None else int(options.alpha)
  usecuda = True\
      if options.usecuda != None and options.usecuda.lower()[0] == 't'\
      else False

def bounded(val):
  return min(255, max(0, val))
```

```
class WatermarkContentExecutor:
  def __init__(self, width, height, watermark, alpha, indir, outdir):
    self.width = width
    self.height = height
    self.watermark = watermark
    self.alpha = alpha
    self.indir = indir
    self.outdir = outdir
  def __call__(self, f):
    infile = self.indir + "/" + f
    outfile = self.outdir + "/" + f.split(".")[0] + ".bmp"
    image = Image.open(infile)
    img = image.load()
    for x, y in np.ndindex((self.width, self.height)):
      img[x, y] = tuple(map(\
          lambda z: bounded(z + self.alpha * self.watermark[x + y*self.width]),\
          img[x, y]))
    image.save(outfile)

class GpuWatermarkContentExecutor:
  def __init__(self, gpu_watermark, indir, outdir):
    self.gpu_watermark = gpu_watermark
    self.indir = indir
    self.outdir = outdir
  def __call__(self, f):
    global cudalib
    infile = self.indir + "/" + f
    outfile = self.outdir + "/" + f.split(".")[0] + ".bmp"
    cudalib.watermark_content(infile, outfile, self.gpu_watermark)

def main():
  global indir, outdir, watermarkfile, alpha, usecuda
  _parse_flags()
  (width, height), watermark = get_watermark(watermarkfile)
  images_to_map = filter(
      ImageSizeFilter((width, height), indir),
      os.listdir(indir))
  if usecuda:
    global cudalib
    import cudalib
    if alpha != 1:
      # TODO: make it work
      raise Exception("No support for cuda watermarking with alpha != 1")
    map(GpuWatermarkContentExecutor(
        cudalib.prepare_gpu_array(watermark),
        indir,
        outdir), images_to_map)
  else:
    get_pool().map(
```

```
        WatermarkContentExecutor(
            width,
            height,
            watermark,
            alpha,
            indir,
            outdir),
        images_to_map)

main()
```