Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Кафедра ПМиК

Лабораторная работа 4

по дисциплине «Прикладная стеганография»

Выполнил: ст. гр. ЗМП-41 Лёвкин И. А.

Проверила: Мерзлякова Е.Ю.

Новосибирск 2025

**Описание методов стегоанализа**

**RS-анализ  
Авторы: Jessica Fridrich, Miroslav Goljan, Rui Du.**

RS-анализ – это статистический метод, разработанный для обнаружения стеганографических изменений в изображениях, особенно при использовании LSB-стеганографии (замены наименее значимых битов). Метод основан на анализе корреляционных свойств групп пикселей после применения определённых функций дискретизации.

Основные этапы метода:

1. Изображение разделяется на непересекающиеся блоки пикселей.
2. Для каждого блока вычисляются две характеристики: регулярность (R) и сингулярность (S), которые показывают, насколько изменяются статистические свойства изображения после внедрения скрытых данных.
3. Анализ соотношения R и S позволяет определить факт наличия стеговложения и оценить его объем.

**Преимущества:**

* Эффективен против LSB-стеганографии.
* Позволяет оценить длину скрытого сообщения.

**Недостатки:**

* Чувствителен к шумам и сжатию изображения.
* Менее эффективен против адаптивных методов стеганографии.

**Asymptotically Uniformly Most Powerful (AUMP)  
Авторы: Andrew D. Ker, Patrick Bas, Tomáš Pevný.**

AUMP представляет собой статистический подход, основанный на теории оптимальных решающих правил. Он предназначен для обнаружения стеганографических изменений в условиях малых размеров встраиваемых сообщений и асимптотически стремится к равномерно наиболее мощному критерию.

**Основные принципы:**

* Использует логарифмические отношения правдоподобия (log-likelihood ratio) для проверки гипотез.
* Оптимизирован для работы в условиях, когда размер встроенного сообщения стремится к нулю (асимптотический анализ).
* Позволяет строить детекторы, близкие к оптимальным по критерию Неймана-Пирсона.

**Преимущества:**

* Высокая эффективность при малых объемах скрытых данных.
* Универсальность – может применяться к различным стеганографическим методам.

**Недостатки:**

* Требует точного знания статистики покрытия.
* Вычислительно сложен по сравнению с эвристическими методами.

**Применение и сравнение методов**

Проведём анализ 100 контейнеров, содержащих сообщение с помощью RS-анализа и AUMP.

**RS-анализ**

**AUMP**

**Вывод по 100 контейнерам RS-анализа**

|  |
| --- |
| encoded\_1.bmp  Average result: 17,10%  Estimated message length (in bytes): 4170.273016291594  encoded\_2.bmp  Average result: 7,86%  Estimated message length (in bytes): 1974.772590345115  encoded\_3.bmp  Average result: 21,20%  Estimated message length (in bytes): 24860.66238791518  encoded\_4.bmp  Average result: 29,55%  Estimated message length (in bytes): 50226.40580636884  encoded\_5.bmp  Average result: 15,80%  Estimated message length (in bytes): 22996.678889774324  encoded\_6.bmp  Average result: 15,02%  Estimated message length (in bytes): 20808.796929071163  encoded\_7.bmp  Average result: 29,53%  Estimated message length (in bytes): 50538.04696213917  encoded\_8.bmp  Average result: 11,09%  Estimated message length (in bytes): 11213.31532871331  encoded\_9.bmp  Average result: 28,73%  Estimated message length (in bytes): 51569.84555729444  encoded\_10.bmp  Average result: 25,17%  Estimated message length (in bytes): 45575.239308337004  encoded\_11.bmp  Average result: 23,28%  Estimated message length (in bytes): 42403.27237603203  encoded\_12.bmp  Average result: 8,10%  Estimated message length (in bytes): 4504.087246972909  encoded\_13.bmp  Average result: 9,38%  Estimated message length (in bytes): 5067.2640087058435  encoded\_14.bmp  Average result: 24,66%  Estimated message length (in bytes): 41005.23056174029  encoded\_15.bmp  Average result: 8,34%  Estimated message length (in bytes): 3902.738193876855  encoded\_16.bmp  Average result: 13,33%  Estimated message length (in bytes): 16700.729014049644  encoded\_17.bmp  Average result: 22,73%  Estimated message length (in bytes): 26363.481055382326  encoded\_18.bmp  Average result: 20,61%  Estimated message length (in bytes): 34905.83621521642  encoded\_19.bmp  Average result: 20,48%  Estimated message length (in bytes): 33248.39171667113  encoded\_20.bmp  Average result: 7,25%  Estimated message length (in bytes): 2561.046186060542  encoded\_21.bmp  Average result: 8,89%  Estimated message length (in bytes): 2347.4503107509636  encoded\_22.bmp  Average result: 8,82%  Estimated message length (in bytes): 4699.350699273489  encoded\_23.bmp  Average result: 14,18%  Estimated message length (in bytes): 15427.346720963706  encoded\_24.bmp  Average result: 14,00%  Estimated message length (in bytes): 16599.2690795609  encoded\_25.bmp  Average result: 27,59%  Estimated message length (in bytes): 45417.97240695092  encoded\_26.bmp  Average result: 13,86%  Estimated message length (in bytes): 16756.34156159868  encoded\_27.bmp  Average result: 15,39%  Estimated message length (in bytes): 24820.53223669811  encoded\_28.bmp  Average result: 13,23%  Estimated message length (in bytes): 9664.354505714604  encoded\_29.bmp  Average result: 13,33%  Estimated message length (in bytes): 16607.358449594656  encoded\_30.bmp  Average result: 10,42%  Estimated message length (in bytes): 9890.427054243315  encoded\_31.bmp  Average result: 29,97%  Estimated message length (in bytes): 52442.73435770984  encoded\_32.bmp  Average result: 12,76%  Estimated message length (in bytes): 16410.486811986142  encoded\_33.bmp  Average result: 33,16%  Estimated message length (in bytes): 61128.88954528181  encoded\_34.bmp  Average result: 29,33%  Estimated message length (in bytes): 51230.745898087815  encoded\_35.bmp  Average result: 23,74%  Estimated message length (in bytes): 25445.70059421772  encoded\_36.bmp  Average result: 15,34%  Estimated message length (in bytes): 17265.275577312947  encoded\_37.bmp  Average result: 9,61%  Estimated message length (in bytes): 6151.444026175092  encoded\_38.bmp  Average result: 20,08%  Estimated message length (in bytes): 31096.619441503677  encoded\_39.bmp  Average result: 10,88%  Estimated message length (in bytes): 13736.117187588308  encoded\_40.bmp  Average result: 21,12%  Estimated message length (in bytes): 31446.843425287385  encoded\_41.bmp  Average result: 8,40%  Estimated message length (in bytes): 3651.0394331898815  encoded\_42.bmp  Average result: 8,91%  Estimated message length (in bytes): 2606.4372293540655  encoded\_43.bmp  Average result: 7,28%  Estimated message length (in bytes): 180.2697890595747  encoded\_44.bmp  Average result: 22,01%  Estimated message length (in bytes): 36888.17650655243  encoded\_45.bmp  Average result: 28,45%  Estimated message length (in bytes): 50825.809341200664  encoded\_46.bmp  Average result: 31,27%  Estimated message length (in bytes): 40179.06577607384  encoded\_47.bmp  Average result: 28,58%  Estimated message length (in bytes): 53118.7353757788  encoded\_48.bmp  Average result: 20,44%  Estimated message length (in bytes): 35041.78183181133  encoded\_49.bmp  Average result: 9,16%  Estimated message length (in bytes): 6281.097031722593  encoded\_50.bmp  Average result: 8,44%  Estimated message length (in bytes): 2755.154862094578  encoded\_51.bmp  Average result: 23,46%  Estimated message length (in bytes): 36387.42443436161  encoded\_52.bmp  Average result: 7,17%  Estimated message length (in bytes): 1078.8943621928788  encoded\_53.bmp  Average result: 10,35%  Estimated message length (in bytes): 9055.327855647838  encoded\_54.bmp  Average result: 22,75%  Estimated message length (in bytes): 26754.337323100932  encoded\_55.bmp  Average result: 33,48%  Estimated message length (in bytes): 48536.34874337083  encoded\_56.bmp  Average result: 7,75%  Estimated message length (in bytes): 2265.689195109833  encoded\_57.bmp  Average result: 22,00%  Estimated message length (in bytes): 42148.00796449851  encoded\_58.bmp  Average result: 13,83%  Estimated message length (in bytes): 18646.329008353816  encoded\_59.bmp  Average result: 14,37%  Estimated message length (in bytes): 15268.892182599748  encoded\_60.bmp  Average result: 15,48%  Estimated message length (in bytes): 21261.12478000147  encoded\_61.bmp  Average result: 13,70%  Estimated message length (in bytes): 5639.339566607324  encoded\_62.bmp  Average result: 24,75%  Estimated message length (in bytes): 39549.14870681628  encoded\_63.bmp  Average result: 6,98%  Estimated message length (in bytes): 64.8088558742161  encoded\_64.bmp  Average result: 18,56%  Estimated message length (in bytes): 14185.098222342811  encoded\_65.bmp  Average result: 13,48%  Estimated message length (in bytes): 14084.598333717156  encoded\_66.bmp  Average result: 9,42%  Estimated message length (in bytes): 5421.939116570213  encoded\_67.bmp  Average result: 15,07%  Estimated message length (in bytes): 22227.299159157505  encoded\_68.bmp  Average result: 16,46%  Estimated message length (in bytes): 17079.081851900835  encoded\_69.bmp  Average result: 27,70%  Estimated message length (in bytes): 42869.18737938494  encoded\_70.bmp  Average result: 18,47%  Estimated message length (in bytes): 23742.24955201026  encoded\_71.bmp  Average result: 22,31%  Estimated message length (in bytes): 39257.756941611704  encoded\_72.bmp  Average result: 7,89%  Estimated message length (in bytes): 2512.8926890685166  encoded\_73.bmp  Average result: 8,93%  Estimated message length (in bytes): 4422.844847929353  encoded\_74.bmp  Average result: 10,33%  Estimated message length (in bytes): 9077.482292954226  encoded\_75.bmp  Average result: 9,76%  Estimated message length (in bytes): 7324.7882572891285  encoded\_76.bmp  Average result: 18,72%  Estimated message length (in bytes): 29105.475162284278  encoded\_77.bmp  Average result: 29,79%  Estimated message length (in bytes): 39021.95311170787  encoded\_78.bmp  Average result: 13,07%  Estimated message length (in bytes): 14042.48832526811  encoded\_79.bmp  Average result: 23,96%  Estimated message length (in bytes): 22237.98180542556  encoded\_80.bmp  Average result: 9,37%  Estimated message length (in bytes): 9059.625739587824  encoded\_81.bmp  Average result: 36,17%  Estimated message length (in bytes): 65661.69909259168  encoded\_82.bmp  Average result: 18,64%  Estimated message length (in bytes): 28031.17137695938  encoded\_83.bmp  Average result: 22,68%  Estimated message length (in bytes): 33701.28540859873  encoded\_84.bmp  Average result: 10,05%  Estimated message length (in bytes): 8654.866592379032  encoded\_85.bmp  Average result: 74,85%  Estimated message length (in bytes): 186775.57500296942  encoded\_86.bmp  Average result: 19,91%  Estimated message length (in bytes): 19583.934420855083  encoded\_87.bmp  Average result: 21,83%  Estimated message length (in bytes): 37022.269866513794  encoded\_88.bmp  Average result: 85,67%  Estimated message length (in bytes): 207687.54217960953  encoded\_89.bmp  Average result: 32,92%  Estimated message length (in bytes): 54940.3993477918  encoded\_90.bmp  Average result: 8,45%  Estimated message length (in bytes): 5392.801772424719  encoded\_91.bmp  Average result: 15,98%  Estimated message length (in bytes): 19400.461739605846  encoded\_92.bmp  Average result: 10,89%  Estimated message length (in bytes): 10961.346873112434  encoded\_93.bmp  Average result: 22,32%  Estimated message length (in bytes): 33149.63639265663  encoded\_94.bmp  Average result: 18,15%  Estimated message length (in bytes): 24992.66685020252  encoded\_95.bmp  Average result: 7,10%  Estimated message length (in bytes): 1397.4036971321532  encoded\_96.bmp  Average result: 27,86%  Estimated message length (in bytes): 35970.46119530605  encoded\_97.bmp  Average result: 15,30%  Estimated message length (in bytes): 20380.19592027833  encoded\_98.bmp  Average result: 6,93%  Estimated message length (in bytes): 72.64080223103042  encoded\_99.bmp  Average result: 8,68%  Estimated message length (in bytes): 4567.524357734648  encoded\_100.bmp  Average result: 23,59%  Estimated message length (in bytes): 35233.13221801197 |

**Вывод по 100 контейнерам AUMP:**

|  |
| --- |
| encoded\_1.bmp  Detection statistic beta = 110.2235  encoded\_2.bmp  Detection statistic beta = -2.6770  encoded\_3.bmp  Detection statistic beta = 26.7905  encoded\_4.bmp  Detection statistic beta = 27.6240  encoded\_5.bmp  Detection statistic beta = 7.5426  encoded\_6.bmp  Detection statistic beta = -3.8916  encoded\_7.bmp  Detection statistic beta = 5.5650  encoded\_8.bmp  Detection statistic beta = -5.3294  encoded\_9.bmp  Detection statistic beta = -1.3443  encoded\_10.bmp  Detection statistic beta = -0.1709  encoded\_11.bmp  Detection statistic beta = -4.7991  encoded\_12.bmp  Detection statistic beta = -17.2238  encoded\_13.bmp  Detection statistic beta = 3.4400  encoded\_14.bmp  Detection statistic beta = 18.6907  encoded\_15.bmp  Detection statistic beta = 1.4507  encoded\_16.bmp  Detection statistic beta = 12.0550  encoded\_17.bmp  Detection statistic beta = 41.3982  encoded\_18.bmp  Detection statistic beta = -7.9041  encoded\_19.bmp  Detection statistic beta = -0.0482  encoded\_20.bmp  Detection statistic beta = 5.2907  encoded\_21.bmp  Detection statistic beta = -10.1980  encoded\_22.bmp  Detection statistic beta = 20.4634  encoded\_23.bmp  Detection statistic beta = 6.1717  encoded\_24.bmp  Detection statistic beta = -7.9993  encoded\_25.bmp  Detection statistic beta = 20.1940  encoded\_26.bmp  Detection statistic beta = 6.4004  encoded\_27.bmp  Detection statistic beta = -0.8708  encoded\_28.bmp  Detection statistic beta = -18.5942  encoded\_29.bmp  Detection statistic beta = -11.5904  encoded\_30.bmp  Detection statistic beta = 2.1358  encoded\_31.bmp  Detection statistic beta = -13.9523  encoded\_32.bmp  Detection statistic beta = 20.3787  encoded\_33.bmp  Detection statistic beta = 6.7397  encoded\_34.bmp  Detection statistic beta = 4.7456  encoded\_35.bmp  Detection statistic beta = 23.8133  encoded\_36.bmp  Detection statistic beta = 53.4237  encoded\_37.bmp  Detection statistic beta = 5.2651  encoded\_38.bmp  Detection statistic beta = 10.4190  encoded\_39.bmp  Detection statistic beta = -3.1706  encoded\_40.bmp  Detection statistic beta = 5.2002  encoded\_41.bmp  Detection statistic beta = -62.6189  encoded\_42.bmp  Detection statistic beta = 10.5314  encoded\_43.bmp  Detection statistic beta = 38.3673  encoded\_44.bmp  Detection statistic beta = 2.7506  encoded\_45.bmp  Detection statistic beta = 7.6166  encoded\_46.bmp  Detection statistic beta = 19.2081  encoded\_47.bmp  Detection statistic beta = 7.0668  encoded\_48.bmp  Detection statistic beta = 4.3595  encoded\_49.bmp  Detection statistic beta = -7.1470  encoded\_50.bmp  Detection statistic beta = 18.2007  encoded\_51.bmp  Detection statistic beta = 4.3320  encoded\_52.bmp  Detection statistic beta = 1.8192  encoded\_53.bmp  Detection statistic beta = -10.8473  encoded\_54.bmp  Detection statistic beta = 12.6650  encoded\_55.bmp  Detection statistic beta = 10.9950  encoded\_56.bmp  Detection statistic beta = 53.2497  encoded\_57.bmp  Detection statistic beta = 5.2756  encoded\_58.bmp  Detection statistic beta = -4.3203  encoded\_59.bmp  Detection statistic beta = -17.2596  encoded\_60.bmp  Detection statistic beta = -15.8968  encoded\_61.bmp  Detection statistic beta = 38.1420  encoded\_62.bmp  Detection statistic beta = -9.5560  encoded\_63.bmp  Detection statistic beta = 15.1761  encoded\_64.bmp  Detection statistic beta = 39.3854  encoded\_65.bmp  Detection statistic beta = 10.1599  encoded\_66.bmp  Detection statistic beta = -14.7212  encoded\_67.bmp  Detection statistic beta = 0.6536  encoded\_68.bmp  Detection statistic beta = 8.1877  encoded\_69.bmp  Detection statistic beta = -9.3077  encoded\_70.bmp  Detection statistic beta = 9.8347  encoded\_71.bmp  Detection statistic beta = 2.1080  encoded\_72.bmp  Detection statistic beta = 5.5581  encoded\_73.bmp  Detection statistic beta = -3.2463  encoded\_74.bmp  Detection statistic beta = 5.9448  encoded\_75.bmp  Detection statistic beta = 1.0917  encoded\_76.bmp  Detection statistic beta = 9.7758  encoded\_77.bmp  Detection statistic beta = 27.1936  encoded\_78.bmp  Detection statistic beta = -0.0346  encoded\_79.bmp  Detection statistic beta = 33.3936  encoded\_80.bmp  Detection statistic beta = -2.3158  encoded\_81.bmp  Detection statistic beta = -8.2851  encoded\_82.bmp  Detection statistic beta = -0.7741  encoded\_83.bmp  Detection statistic beta = 2.2132  encoded\_84.bmp  Detection statistic beta = 6.8512  encoded\_85.bmp  Detection statistic beta = 3.9996  encoded\_86.bmp  Detection statistic beta = 53.4238  encoded\_87.bmp  Detection statistic beta = -15.2261  encoded\_88.bmp  Detection statistic beta = -7.6045  encoded\_89.bmp  Detection statistic beta = -23.3877  encoded\_90.bmp  Detection statistic beta = 1.7931  encoded\_91.bmp  Detection statistic beta = -6.9119  encoded\_92.bmp  Detection statistic beta = 1.9626  encoded\_93.bmp  Detection statistic beta = 13.1152  encoded\_94.bmp  Detection statistic beta = -38.8914  encoded\_95.bmp  Detection statistic beta = 3.1523  encoded\_96.bmp  Detection statistic beta = 39.4272  encoded\_97.bmp  Detection statistic beta = 12.7835  encoded\_98.bmp  Detection statistic beta = 4.3463  encoded\_99.bmp  Detection statistic beta = 9.4742  encoded\_100.bmp  Detection statistic beta = 12.4950 |

**Листинг**

**RS-анализ**

**RSAnalysis.java**

/\*

\* Digital Invisible Ink Toolkit

\* Copyright (C) 2005 K. Hempstalk

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

\*

\* @author Kathryn Hempstalk

\*/

import java.awt.image.BufferedImage;

import java.io.File;

import java.util.Collections;

import java.util.Enumeration;

import java.util.List;

import java.util.Vector;

import javax.imageio.ImageIO;

/\*\*

\* RS analysis for a stego-image.

\* <P>

\* RS analysis is a system for detecting LSB steganography proposed by Dr.

\* Fridrich at Binghamton University, NY. You can visit her webpage for more

\* information - {@link http://www.ws.binghamton.edu/fridrich/} <BR>

\* Implemented as described in "Reliable detection of LSB steganography in color

\* and grayscale images" by J. Fridrich, M. Goljan and R. Du.

\* <BR>

\* This code was produced with the aid of the authors and has been verified as a

\* correct implementation of RS Analysis. Their assistance has proved

\* invaluable.

\*

\* @author Kathryn Hempstalk

\*/

public class RSAnalysis extends PixelBenchmark {

//CONSTRUCTORS

/\*\*

\* Creates a new RS analysis with a given mask size of m x n.

\*

\* Each alternating bit is set to 1. Eg for a mask of size 2x2 the resulting

\* mask will be {1,0;0,1}. Two masks are used - one is the inverse of the

\* other.

\*

\* @param m The x mask size.

\* @param n The y mask size.

\*/

public RSAnalysis(int m, int n) {

//two masks

mMask = new int[2][m \* n];

//iterate through them and set alternating bits

int k = 0;

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

if (((j % 2) == 0 && (i % 2) == 0)

|| ((j % 2) == 1 && (i % 2) == 1)) {

mMask[0][k] = 1;

mMask[1][k] = 0;

} else {

mMask[0][k] = 0;

mMask[1][k] = 1;

}

k++;

}

}

//set up the mask size.

mM = m;

mN = n;

}

//FUNCTIONS

/\*\*

\* Does an RS analysis of a given image.

\* <P>

\* The analysis data returned is specified by name in the getResultNames()

\* method.

\*

\* @param image The image to analyse.

\* @param colour The colour to analyse.

\* @param overlap Whether the blocks should overlap or not.

\* @return The analysis information.

\*/

public double[] doAnalysis(BufferedImage image, int colour, boolean overlap) {

//get the images sizes

int imgx = image.getWidth(), imgy = image.getHeight();

int startx = 0, starty = 0;

int block[] = new int[mM \* mN];

double numregular = 0, numsingular = 0;

double numnegreg = 0, numnegsing = 0;

double numunusable = 0, numnegunusable = 0;

double variationB, variationP, variationN;

while (startx < imgx && starty < imgy) {

//this is done once for each mask...

for (int m = 0; m < 2; m++) {

//get the block of data

int k = 0;

for (int i = 0; i < mN; i++) {

for (int j = 0; j < mM; j++) {

block[k] = image.getRGB(startx + j, starty + i);

k++;

}

}

//get the variation the block

variationB = getVariation(block, colour);

//now flip according to the mask

block = flipBlock(block, mMask[m]);

variationP = getVariation(block, colour);

//flip it back

block = flipBlock(block, mMask[m]);

//negative mask

mMask[m] = this.invertMask(mMask[m]);

variationN = getNegativeVariation(block, colour, mMask[m]);

mMask[m] = this.invertMask(mMask[m]);

//now we need to work out which group each belongs to

//positive groupings

if (variationP > variationB) {

numregular++;

}

if (variationP < variationB) {

numsingular++;

}

if (variationP == variationB) {

numunusable++;

}

//negative mask groupings

if (variationN > variationB) {

numnegreg++;

}

if (variationN < variationB) {

numnegsing++;

}

if (variationN == variationB) {

numnegunusable++;

}

//now we keep going...

}

//get the next position

if (overlap) {

startx += 1;

} else {

startx += mM;

}

if (startx >= (imgx - 1)) {

startx = 0;

if (overlap) {

starty += 1;

} else {

starty += mN;

}

}

if (starty >= (imgy - 1)) {

break;

}

}

//get all the details needed to derive x...

double totalgroups = numregular + numsingular + numunusable;

double allpixels[] = this.getAllPixelFlips(image, colour, overlap);

double x = getX(numregular, numnegreg, allpixels[0], allpixels[2],

numsingular, numnegsing, allpixels[1], allpixels[3]);

//calculate the estimated percent of flipped pixels and message length

double epf, ml;

if (2 \* (x - 1) == 0) {

epf = 0;

} else {

epf = Math.abs(x / (2 \* (x - 1)));

}

if (x - 0.5 == 0) {

ml = 0;

} else {

ml = Math.abs(x / (x - 0.5));

}

//now we have the number of regular and singular groups...

double results[] = new double[28];

//save them all...

//these results

results[0] = numregular;

results[1] = numsingular;

results[2] = numnegreg;

results[3] = numnegsing;

results[4] = Math.abs(numregular - numnegreg);

results[5] = Math.abs(numsingular - numnegsing);

results[6] = (numregular / totalgroups) \* 100;

results[7] = (numsingular / totalgroups) \* 100;

results[8] = (numnegreg / totalgroups) \* 100;

results[9] = (numnegsing / totalgroups) \* 100;

results[10] = (results[4] / totalgroups) \* 100;

results[11] = (results[5] / totalgroups) \* 100;

//all pixel results

results[12] = allpixels[0];

results[13] = allpixels[1];

results[14] = allpixels[2];

results[15] = allpixels[3];

results[16] = Math.abs(allpixels[0] - allpixels[1]);

results[17] = Math.abs(allpixels[2] - allpixels[3]);

results[18] = (allpixels[0] / totalgroups) \* 100;

results[19] = (allpixels[1] / totalgroups) \* 100;

results[20] = (allpixels[2] / totalgroups) \* 100;

results[21] = (allpixels[3] / totalgroups) \* 100;

results[22] = (results[16] / totalgroups) \* 100;

results[23] = (results[17] / totalgroups) \* 100;

//overall results

results[24] = totalgroups;

results[25] = epf;

results[26] = ml;

results[27] = ((imgx \* imgy \* 3) \* ml) / 8;

return results;

}

/\*\*

\* Gets the x value for the p=x(x/2) RS equation. See the paper for more

\* details.

\*

\* @param r The value of Rm(p/2).

\* @param rm The value of R-m(p/2).

\* @param r1 The value of Rm(1-p/2).

\* @param rm1 The value of R-m(1-p/2).

\* @param s The value of Sm(p/2).

\* @param sm The value of S-m(p/2).

\* @param s1 The value of Sm(1-p/2).

\* @param sm1 The value of S-m(1-p/2).

\* @return The value of x.

\*/

private double getX(double r, double rm, double r1, double rm1,

double s, double sm, double s1, double sm1) {

double x = 0; //the cross point.

double dzero = r - s; // d0 = Rm(p/2) - Sm(p/2)

double dminuszero = rm - sm; // d-0 = R-m(p/2) - S-m(p/2)

double done = r1 - s1; // d1 = Rm(1-p/2) - Sm(1-p/2)

double dminusone = rm1 - sm1; // d-1 = R-m(1-p/2) - S-m(1-p/2)

//get x as the root of the equation

//2(d1 + d0)x^2 + (d-0 - d-1 - d1 - 3d0)x + d0 - d-0 = 0

//x = (-b +or- sqrt(b^2-4ac))/2a

//where ax^2 + bx + c = 0 and this is the form of the equation

//thanks to a good friend in Dunedin, NZ for helping with maths

//and to Miroslav Goljan's fantastic Matlab code

double a = 2 \* (done + dzero);

double b = dminuszero - dminusone - done - (3 \* dzero);

double c = dzero - dminuszero;

if (a == 0) //take it as a straight line

{

x = c / b;

}

//take it as a curve

double discriminant = Math.pow(b, 2) - (4 \* a \* c);

if (discriminant >= 0) {

double rootpos = ((-1 \* b) + Math.sqrt(discriminant)) / (2 \* a);

double rootneg = ((-1 \* b) - Math.sqrt(discriminant)) / (2 \* a);

//return the root with the smallest absolute value (as per paper)

if (Math.abs(rootpos) <= Math.abs(rootneg)) {

x = rootpos;

} else {

x = rootneg;

}

} else {

//maybe it's not the curve we think (straight line)

double cr = (rm - r) / (r1 - r + rm - rm1);

double cs = (sm - s) / (s1 - s + sm - sm1);

x = (cr + cs) / 2;

}

if (x == 0) {

double ar = ((rm1 - r1 + r - rm) + (rm - r) / x) / (x - 1);

double as = ((sm1 - s1 + s - sm) + (sm - s) / x) / (x - 1);

if (as > 0 | ar < 0) {

//let's assume straight lines again...

double cr = (rm - r) / (r1 - r + rm - rm1);

double cs = (sm - s) / (s1 - s + sm - sm1);

x = (cr + cs) / 2;

}

}

return x;

}

/\*\*

\* Gets the RS analysis results for flipping performed on all pixels.

\*

\* @param image The image to analyse.

\* @param colour The colour to analyse.

\* @param overlap Whether the blocks should overlap.

\* @return The analysis information for all flipped pixels.

\*/

private double[] getAllPixelFlips(BufferedImage image, int colour, boolean overlap) {

//setup the mask for everything...

int[] allmask = new int[mM \* mN];

for (int i = 0; i < allmask.length; i++) {

allmask[i] = 1;

}

//now do the same as the doAnalysis() method

//get the images sizes

int imgx = image.getWidth(), imgy = image.getHeight();

int startx = 0, starty = 0;

int block[] = new int[mM \* mN];

double numregular = 0, numsingular = 0;

double numnegreg = 0, numnegsing = 0;

double numunusable = 0, numnegunusable = 0;

double variationB, variationP, variationN;

while (startx < imgx && starty < imgy) {

//done once for each mask

for (int m = 0; m < 2; m++) {

//get the block of data

int k = 0;

for (int i = 0; i < mN; i++) {

for (int j = 0; j < mM; j++) {

block[k] = image.getRGB(startx + j, starty + i);

k++;

}

}

//flip all the pixels in the block (NOTE: THIS IS WHAT'S DIFFERENT

//TO THE OTHER doAnalysis() METHOD)

block = flipBlock(block, allmask);

//get the variation the block

variationB = getVariation(block, colour);

//now flip according to the mask

block = flipBlock(block, mMask[m]);

variationP = getVariation(block, colour);

//flip it back

block = flipBlock(block, mMask[m]);

//negative mask

mMask[m] = this.invertMask(mMask[m]);

variationN = getNegativeVariation(block, colour, mMask[m]);

mMask[m] = this.invertMask(mMask[m]);

//now we need to work out which group each belongs to

//positive groupings

if (variationP > variationB) {

numregular++;

}

if (variationP < variationB) {

numsingular++;

}

if (variationP == variationB) {

numunusable++;

}

//negative mask groupings

if (variationN > variationB) {

numnegreg++;

}

if (variationN < variationB) {

numnegsing++;

}

if (variationN == variationB) {

numnegunusable++;

}

//now we keep going...

}

//get the next position

if (overlap) {

startx += 1;

} else {

startx += mM;

}

if (startx >= (imgx - 1)) {

startx = 0;

if (overlap) {

starty += 1;

} else {

starty += mN;

}

}

if (starty >= (imgy - 1)) {

break;

}

}

//save all the results (same order as before)

double results[] = new double[4];

results[0] = numregular;

results[1] = numsingular;

results[2] = numnegreg;

results[3] = numnegsing;

return results;

}

/\*\*

\* Returns an enumeration of all the result names.

\*

\* @return The names of all the results.

\*/

public Enumeration getResultNames() {

Vector names = new Vector(28);

names.add("Number of regular groups (positive)");

names.add("Number of singular groups (positive)");

names.add("Number of regular groups (negative)");

names.add("Number of singular groups (negative)");

names.add("Difference for regular groups");

names.add("Difference for singular groups");

names.add("Percentage of regular groups (positive)");

names.add("Percentage of singular groups (positive)");

names.add("Percentage of regular groups (negative)");

names.add("Percentage of singular groups (negative)");

names.add("Difference for regular groups %");

names.add("Difference for singular groups %");

names.add("Number of regular groups (positive for all flipped)");

names.add("Number of singular groups (positive for all flipped)");

names.add("Number of regular groups (negative for all flipped)");

names.add("Number of singular groups (negative for all flipped)");

names.add("Difference for regular groups (all flipped)");

names.add("Difference for singular groups (all flipped)");

names.add("Percentage of regular groups (positive for all flipped)");

names.add("Percentage of singular groups (positive for all flipped)");

names.add("Percentage of regular groups (negative for all flipped)");

names.add("Percentage of singular groups (negative for all flipped)");

names.add("Difference for regular groups (all flipped) %");

names.add("Difference for singular groups (all flipped) %");

names.add("Total number of groups");

names.add("Estimated percent of flipped pixels");

names.add("Estimated message length (in percent of pixels)(p)");

names.add("Estimated message length (in bytes)");

return names.elements();

}

/\*\*

\* Gets the variation of the blocks of data. Uses the formula f(x) = |x0 -

\* x1| + |x1 + x3| + |x3 - x2| + |x2 - x0|; However, if the block is not in

\* the shape 2x2 or 4x1, this will be applied as many times as the block can

\* be broken up into 4 (without overlaps).

\*

\* @param block The block of data (in 24 bit colour).

\* @param colour The colour to get the variation of.

\* @return The variation in the block.

\*/

private double getVariation(int[] block, int colour) {

double var = 0;

int colour1, colour2;

for (int i = 0; i < block.length; i = i + 4) {

colour1 = getPixelColour(block[0 + i], colour);

colour2 = getPixelColour(block[1 + i], colour);

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[3 + i], colour);

colour2 = getPixelColour(block[2 + i], colour);

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[1 + i], colour);

colour2 = getPixelColour(block[3 + i], colour);

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[2 + i], colour);

colour2 = getPixelColour(block[0 + i], colour);

var += Math.abs(colour1 - colour2);

}

return var;

}

/\*\*

\* Gets the negative variation of the blocks of data. Uses the formula f(x)

\* = |x0 - x1| + |x1 + x3| + |x3 - x2| + |x2 - x0|; However, if the block is

\* not in the shape 2x2 or 4x1, this will be applied as many times as the

\* block can be broken up into 4 (without overlaps).

\*

\* @param block The block of data (in 24 bit colour).

\* @param colour The colour to get the variation of.

\* @param mask The negative mask.

\* @return The variation in the block.

\*/

private double getNegativeVariation(int[] block, int colour, int[] mask) {

double var = 0;

int colour1, colour2;

for (int i = 0; i < block.length; i = i + 4) {

colour1 = getPixelColour(block[0 + i], colour);

colour2 = getPixelColour(block[1 + i], colour);

if (mask[0 + i] == -1) {

colour1 = invertLSB(colour1);

}

if (mask[1 + i] == -1) {

colour2 = invertLSB(colour2);

}

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[1 + i], colour);

colour2 = getPixelColour(block[3 + i], colour);

if (mask[1 + i] == -1) {

colour1 = invertLSB(colour1);

}

if (mask[3 + i] == -1) {

colour2 = invertLSB(colour2);

}

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[3 + i], colour);

colour2 = getPixelColour(block[2 + i], colour);

if (mask[3 + i] == -1) {

colour1 = invertLSB(colour1);

}

if (mask[2 + i] == -1) {

colour2 = invertLSB(colour2);

}

var += Math.abs(colour1 - colour2);

colour1 = getPixelColour(block[2 + i], colour);

colour2 = getPixelColour(block[0 + i], colour);

if (mask[2 + i] == -1) {

colour1 = invertLSB(colour1);

}

if (mask[0 + i] == -1) {

colour2 = invertLSB(colour2);

}

var += Math.abs(colour1 - colour2);

}

return var;

}

/\*\*

\* Gets the given colour value for this pixel.

\*

\* @param pixel The pixel to get the colour of.

\* @param colour The colour to get.

\* @return The colour value of the given colour in the given pixel.

\*/

public int getPixelColour(int pixel, int colour) {

if (colour == RSAnalysis.ANALYSIS\_COLOUR\_RED) {

return getRed(pixel);

} else if (colour == RSAnalysis.ANALYSIS\_COLOUR\_GREEN) {

return getGreen(pixel);

} else if (colour == RSAnalysis.ANALYSIS\_COLOUR\_BLUE) {

return getBlue(pixel);

} else {

return 0;

}

}

/\*\*

\* Flips a block of pixels.

\*

\* @param block The block to flip.

\* @param mask The mask to use for flipping.

\* @return The flipped block.

\*/

private int[] flipBlock(int[] block, int[] mask) {

//if the mask is true, negate every LSB

for (int i = 0; i < block.length; i++) {

if ((mask[i] == 1)) {

//get the colour

int red = getRed(block[i]), green = getGreen(block[i]),

blue = getBlue(block[i]);

//negate their LSBs

red = negateLSB(red);

green = negateLSB(green);

blue = negateLSB(blue);

//build a new pixel

int newpixel = (0xff << 24) | ((red & 0xff) << 16)

| ((green & 0xff) << 8) | ((blue & 0xff));

//change the block pixel

block[i] = newpixel;

} else if (mask[i] == -1) {

//get the colour

int red = getRed(block[i]), green = getGreen(block[i]),

blue = getBlue(block[i]);

//negate their LSBs

red = invertLSB(red);

green = invertLSB(green);

blue = invertLSB(blue);

//build a new pixel

int newpixel = (0xff << 24) | ((red & 0xff) << 16)

| ((green & 0xff) << 8) | ((blue & 0xff));

//change the block pixel

block[i] = newpixel;

}

}

return block;

}

/\*\*

\* Negates the LSB of a given byte (stored in an int).

\*

\* @param abyte The byte to negate the LSB of.

\* @return The byte with negated LSB.

\*/

private int negateLSB(int abyte) {

int temp = abyte & 0xfe;

if (temp == abyte) {

return abyte | 0x1;

} else {

return temp;

}

}

/\*\*

\* Inverts the LSB of a given byte (stored in an int).

\*

\* @param abyte The byte to flip.

\* @return The byte with the flipped LSB.

\*/

private int invertLSB(int abyte) {

if (abyte == 255) {

return 256;

}

if (abyte == 256) {

return 255;

}

return (negateLSB(abyte + 1) - 1);

}

/\*\*

\* Inverts a mask.

\*

\* @param mask The mask to invert.

\* @return The flipped mask.

\*/

private int[] invertMask(int[] mask) {

for (int i = 0; i < mask.length; i++) {

mask[i] = mask[i] \* -1;

}

return mask;

}

/\*\*

\* A small main method that will print out the message length in percent of

\* pixels.

\*

\*/

public static void main(String[] args) {

if (args.length != 1) {

System.out.println("Usage: invisibleinktoolkit.benchmark.RSAnalysis <imagefilename>");

System.exit(1);

}

try {

System.out.println("\nRS Analysis results");

System.out.println("-------------------");

RSAnalysis rsa = new RSAnalysis(2, 2);

BufferedImage image = ImageIO.read(new File(args[0]));

double average = 0;

double[] results = rsa.doAnalysis(image, RSAnalysis.ANALYSIS\_COLOUR\_RED, true);

System.out.println("Result from red: " + results[26]);

average += results[26];

results = rsa.doAnalysis(image, RSAnalysis.ANALYSIS\_COLOUR\_GREEN, true);

System.out.println("Result from green: " + results[26]);

average += results[26];

results = rsa.doAnalysis(image, RSAnalysis.ANALYSIS\_COLOUR\_BLUE, true);

System.out.println("Result from blue: " + results[26]);

average += results[26];

average = average / 3;

System.out.println("Average result: " + average);

System.out.println();

List<String> result\_names = Collections.list(rsa.getResultNames());

for (int i = 0; i < results.length; i++) {

System.out.println(result\_names.get(i) + ": " + results[i]);

}

} catch (Exception e) {

System.out.println("ERROR: Cannot process that image type, please try another image.");

e.printStackTrace();

}

}

//VARIABLES

/\*\*

\* Denotes analysis to be done with red.

\*/

public static final int ANALYSIS\_COLOUR\_RED = 0;

/\*\*

\* Denotes analysis to be done with green.

\*/

public static final int ANALYSIS\_COLOUR\_GREEN = 1;

/\*\*

\* Denotes analysis to be done with blue.

\*/

public static final int ANALYSIS\_COLOUR\_BLUE = 2;

/\*\*

\* The mask to be used for the pixel groups.

\*/

private int[][] mMask;

/\*\*

\* The x length of the mask.

\*/

private int mM;

/\*\*

\* The y length of the mask.

\*/

private int mN;

}

//end of class

**PixelBenchmark.java**

/\*

\* Digital Invisible Ink Toolkit

\* Copyright (C) 2005 K. Hempstalk

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

\*

\*

\* @author Kathryn Hempstalk

\*/

/\*\*

\* A convience class to provide all the base methods

\* for benchmarkers to use.

\*

\* Provides pixel functions commonly used in benchmarking.

\*

\* @author Kathryn Hempstalk.

\*/

public class PixelBenchmark{

/\*\*

\* Gets the red content of a pixel.

\*

\* @param pixel The pixel to get the red content of.

\* @return The red content of the pixel.

\*/

public int getRed(int pixel){

return ((pixel >> 16) & 0xff);

}

/\*\*

\* Gets the green content of a pixel.

\*

\* @param pixel The pixel to get the green content of.

\* @return The green content of the pixel.

\*/

public int getGreen(int pixel){

return ((pixel >> 8) & 0xff);

}

/\*\*

\* Gets the blue content of a pixel.

\*

\* @param pixel The pixel to get the blue content of.

\* @return The blue content of the pixel.

\*/

public int getBlue(int pixel){

return (pixel & 0xff);

}

}

//end of class.

**AUMP**

**aump.m**

function beta = aump(X,m,d)

%

% AUMP LSB detector as described by L. Fillatre, "Adaptive Steganalysis of

% Least Significant Bit Replacement in Grayscale Natural Images", IEEE

% TSP, October 2011.

%

% X = image to be analyzed

% m = pixel block size

% d = q - 1 = polynomial degree for fitting (predictor)

% beta = \hat{\Lambda}^\star(X) detection statistic

%

X = double(X);

[Xpred,~,w] = Pred\_aump(X,m,d); % Polynomial prediction, w = weights

r = X - Xpred; % Residual

Xbar = X + 1 - 2 \* mod(X,2); % Flip all LSBs

beta = sum(sum(w.\*(X-Xbar).\*r)); % Detection statistic

function [Xpred,S,w] = Pred\_aump(X,m,d)

%

% Pixel predictor by fitting local polynomial of degree d = q - 1 to

% m pixels, m must divide the number of pixels in the row.

% OUTPUT: predicted image Xpred, local variances S, weights w.

%

% Implemention follows the description in: L. Fillantre, "Adaptive

% Steganalysis of Least Significant Bit Replacement in Grayscale Images",

% IEEE Trans. on Signal Processing, 2011.

%

sig\_th = 1; % Threshold for sigma for numerical stability

q = d + 1; % q = number of parameters per block

Kn = numel(X)/m; % Number of blocks of m pixels

Y = zeros(m,Kn); % Y will hold block pixel values as columns

S = zeros(size(X)); % Pixel variance

Xpred = zeros(size(X)); % Predicted image

H = zeros(m,q); % H = Vandermonde matrix for the LSQ fit

x1 = (1:m)/m;

for i = 1 : q, H(:,i) = (x1').^(i-1); end

for i = 1 : m % Form Kn blocks of m pixels (row-wise) as

aux = X(:,i:m:end); % columns of Y

Y(i,:) = aux(:);

end

p = H\Y; % Polynomial fit

Ypred = H\*p; % Predicted Y

for i = 1 : m % Predicted X

Xpred(:,i:m:end) = reshape(Ypred(i,:),size(X(:,i:m:end))); % Xpred = l\_k in the paper

end

sig2 = sum((Y - Ypred).^2) / (m-q); % sigma\_k\_hat in the paper (variance in kth block)

sig2 = max(sig\_th^2 \* ones(size(sig2)),sig2); % Assuring numerical stability

% le01 = find(sig2 < sig\_th^2);

% sig2(le01) = (0.1 + sqrt(sig2(le01))).^2; % An alternative way of "scaling" to guarantee num. stability

Sy = ones(m,1) \* sig2; % Variance of all pixels (order as in Y)

for i = 1 : m % Reshaping the variance Sy to size of X

S(:,i:m:end) = reshape(Sy(i,:),size(X(:,i:m:end)));

end

s\_n2 = Kn / sum(1./sig2); % Global variance sigma\_n\_bar\_hat^2 in the paper

w = sqrt( s\_n2 / (Kn \* (m-q)) ) ./ S; % Weights

**main.m**

% Загрузка и подготовка изображения

img\_path = '14.bmp'; % Укажите путь к вашему изображению

X = imread(img\_path);

% Если изображение цветное, преобразуем в градации серого

if size(X, 3) == 3

X = rgb2gray(X);

end

m=16

d=2

% Вызов функции aump с параметрами m=16, d=2

beta = aump(X, m, d);

% Вывод результата

fprintf('Detection statistic beta = %.4f\n', beta);