Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Кафедра ПМиК

Лабораторная работа 3

по дисциплине «Прикладная стеганография»

Выполнил: ст. гр. ЗМП-41 Лёвкин И. А.

Проверила: Мерзлякова Е.Ю.

Новосибирск 2025

**Обзор реверсивных стеганографических методов, основанных на интерполяции**

Реверсивные стеганографические методы, основанные на интерполяции изображений, позволяют скрывать информацию с возможностью полного восстановления исходного изображения после извлечения данных. Эти методы сочетают высокую вместимость (емкость внедрения) и хорошее визуальное качество стегоизображения. Рассмотрим ключевые методы, представленные в предложенных работах:

1. **Jung и Yoo (NMI, Neighbor Mean Interpolation)**
   * **Описание**: метод использует интерполяцию для увеличения изображения, после чего секретные данные внедряются в разницу между интерполированными и исходными пикселями.
   * **Преимущества**: высокая скорость вычислений, хорошее качество изображения (PSNR > 35 дБ).
   * **Недостатки**: ограниченная вместимость по сравнению с более современными методами.
2. **INP (Interpolation by Neighboring Pixel)**
   * **Описание**: улучшенная версия метода Jung, где для интерполяции используются соседние пиксели, что позволяет увеличить объем внедряемых данных.
   * **Преимущества**: большая вместимость, сохранение высокого PSNR.
   * **Недостатки**: сложность вычислений выше, чем у NMI.
3. **NIE (New Interpolation Expansion)**
   * **Описание**: основан на расширении изображения с использованием новых алгоритмов интерполяции, что повышает качество стегоизображения.
   * **Преимущества**: лучшее визуальное качество и высокая безопасность.
   * **Недостатки**: требует больше вычислительных ресурсов.
4. **Метод Нагиевой и Вердиева**
   * **Описание**: использует интерполяцию для создания контейнера, а затем внедряет данные, преобразуя разницу пикселей в двоичную систему.
   * **Преимущества**: очень высокая вместимость и PSNR > 38 дБ.
   * **Недостатки**: Сложность реализации из-за необходимости преобразования данных.
5. **Sabeen Govind (ENMI, Enhanced Neighbor Mean Interpolation)**
   * **Описание**: улучшенная версия NMI с двухэтапной схемой внедрения.
   * **Преимущества**: большая вместимость.
   * **Недостатки**: снижение визуального качества по сравнению с другими методами.

Для реализации был выбран метод INP по следующим причинам**:**

* INP обеспечивает высокую вместимость при сохранении высокого PSNR > 36.
* Метод просто в реализации в отличие от NIE или метода Нагиевой-Вердиева.
* Может быть адаптирован для работы с различными типами изображений.

Алгоритм шифрования:

1. Загрузка изображения-контейнера.
2. Уменьшение изображения.
3. Интерполяция для восстановления размера с использованием соседних пикселей.
4. Преобразование сообщения в бинарный формат.
5. Внедрение битов в разницу между интерполированными и исходными пикселями.

Алгоритм дешифрования:

1. Загрузка изображения-контейнера.
2. Чтение пикселей для восстановления сообщения.
3. Преобразование бинарного формата в сообщение.

**Оценка алгоритма**

Проведём встраивание данных в изображение и оценим ёмкость и величину искажения PSNR на 8-битном изображении с палитрой из оттенков серого. Встраиваемый текст на английском языке и составляет размер 16 КБ. Размер контейнера в свою очередь составляет 257 КБ.

Вывод программы:

|  |
| --- |
| Embed bits: 130576  Capacity (bit/pixel): 0.4981  PSNR: 38.56 dB |

Здесь можно видеть, что ёмкость встраивания составляет 0.4981 бит на пиксель.

Мы встроили текст размером примерно в 50% от максимальной вместимости контейнера. В таком случае PSNR составляет 38.56 ДБ, при таком значении искажения могут быть заметны при внимательном расммотрении изображения.

|  |  |
| --- | --- |
| Рис. 1. Контейнер без сообщения | Рис. 2. Контейнер после встраивания сообщения |

Запустим программу на наборе из 10 контейнеров

|  |  |
| --- | --- |
| 20.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.56 dB  21.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.56 dB  22.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 39.51 dB  23.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.88 dB  24.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.61 dB | 25.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.56 dB  26.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.56 dB  27.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.57 dB  28.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.76 dB  29.bmp  Embed bits: 16322.0  Capacity (bit/pixel): 0.4981  PSNR: 38.66 dB |

**Листинг**

import argparse

import math

from pathlib import Path

import sys

import numpy as np

from PIL import Image

import rdh

ROOT\_DIR = Path(\_\_file\_\_).resolve().parent.parent

if str(ROOT\_DIR) not in sys.path:

    sys.path.append(str(ROOT\_DIR))

import utils.stego as stego

def main():

    parser = argparse.ArgumentParser(description="RDH Stenography for 8-bit BMP images")

    parser.add\_argument("-m", "--message", required=True, help="Message file")

    parser.add\_argument("-i", "--input", required=True, help="Input BMP image")

    parser.add\_argument("-o", "--output", required=True, help="Output stego image")

    args = parser.parse\_args()

    message = open(args.message, "r", encoding="utf-8").read()

    secret\_binary = rdh.text\_to\_bits(message)

    input\_img = Image.open(args.input).convert("L")

    full\_img = np.array(input\_img)

    small\_img = rdh.downscale\_image(full\_img)

    cover\_img = rdh.upscale\_inp(small\_img)

    stego\_img, embedded\_bits = rdh.embed\_secret(cover\_img, secret\_binary)

    rdh.save\_image(stego\_img, args.output)

    recovered\_bits = rdh.extract\_secret(

        stego\_img, cover\_img.astype(np.uint8), embedded\_bits

    )

    output\_text = rdh.bits\_to\_text(recovered\_bits)

    psnr\_val = stego.psnr(cover\_img, stego\_img, input\_img.mode)

    capacity = embedded\_bits / (full\_img.shape[0] \* full\_img.shape[1])

    print("Встраивание завершено.")

    print(f"Встроено бит: {embedded\_bits}")

    print(f"Емкость (бит/пиксель): {capacity:.4f}")

    print(f"PSNR: {psnr\_val:.2f} dB")

    print(f"Извлечённый текст: {output\_text}")

    print(f"Совпадает: {output\_text == message}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

import numpy as np

from PIL import Image

def text\_to\_bits(text, encoding="utf-8") -> str:

    return "".join(format(byte, "08b") for byte in text.encode(encoding))

def bits\_to\_text(bits, encoding="utf-8") -> str:

    chars = [bits[i : i + 8] for i in range(0, len(bits), 8)]

    byte\_array = bytearray(int(b, 2) for b in chars if len(b) == 8)

    return byte\_array.decode(encoding, errors="ignore")

def load\_image(path):

    img = Image.open(path).convert("L")

    return np.array(img)

def save\_image(image\_array, path):

    img = Image.fromarray(np.clip(image\_array, 0, 255).astype(np.uint8))

    img.save(path)

def downscale\_image(img):

    return img[::2, ::2]

def upscale\_inp(original):

    h, w = original.shape

    new\_h, new\_w = h \* 2 - 1, w \* 2 - 1

    result = np.zeros((new\_h, new\_w), dtype=np.uint8)

    for i in range(h):

        for j in range(w):

            result[2 \* i, 2 \* j] = original[i, j]

    for i in range(0, new\_h, 2):

        for j in range(1, new\_w, 2):

            left = result[i, j - 1]

            right = result[i, j + 1] if j + 1 < new\_w else left

            result[i, j] = (int(left) + int(right)) // 2

    for i in range(1, new\_h, 2):

        for j in range(0, new\_w, 2):

            top = result[i - 1, j]

            bottom = result[i + 1, j] if i + 1 < new\_h else top

            result[i, j] = (int(top) + int(bottom)) // 2

    for i in range(1, new\_h, 2):

        for j in range(1, new\_w, 2):

            tl = result[i - 1, j - 1]

            tr = result[i - 1, j + 1] if j + 1 < new\_w else tl

            bl = result[i + 1, j - 1] if i + 1 < new\_h else tl

            br = result[i + 1, j + 1] if (i + 1 < new\_h and j + 1 < new\_w) else tl

            result[i, j] = (int(tl) + int(tr) + int(bl) + int(br)) // 4

    return result

def get\_code\_and\_index(d, k):

    M = 2 \*\* (k - 1)

    if d < M:

        return 0, d - M  # center

    else:

        return 1, d - M - (2 \*\* (k - 1))  # shifted for index 1

def get\_symbol\_from\_code(index, code, k):

    M = 2 \*\* (k - 1)

    if index == 0:

        return code + M

    else:

        return code + M + (2 \*\* (k - 1))

def embed\_secret(cover, secret\_bits, k=4):

    h, w = cover.shape

    stego = cover.copy().astype(np.int16)

    i = 0

    embedded\_bits = 0

    for y in range(0, h - 2, 4):

        for x in range(0, w - 2, 4):

            if i + 4 \* k > len(secret\_bits):

                return np.clip(stego, 0, 255).astype(np.uint8), embedded\_bits

            symbols = []

            for j in range(4):

                start = i + j \* k

                end = i + (j + 1) \* k

                if end > len(secret\_bits):

                    bits = secret\_bits[start:] + "0" \* (end - len(secret\_bits))

                    symbols.append(int(bits, 2))

                    embedded\_bits += len(secret\_bits[start:])

                    i = len(secret\_bits)

                    break

                else:

                    bits = secret\_bits[start:end]

                    symbols.append(int(bits, 2))

                    embedded\_bits += k

            else:

                i += 4 \* k

            codes = []

            indexes = []

            for s in symbols:

                index, code = get\_code\_and\_index(s, k)

                codes.append(code)

                indexes.append(index)

            index\_bin = "".join(map(str, indexes))

            I = int(index\_bin, 2)

            M = 2\*\*k

            Pc = int(cover[y + 2, x + 2])

            stego[y + 2, x + 2] = Pc + (I - M)

            positions = [(y, x + 1), (y + 1, x), (y + 1, x + 2), (y + 2, x + 1)]

            for pos, code in zip(positions, codes):

                stego[pos] += code

    return np.clip(stego, 0, 255).astype(np.uint8), embedded\_bits

def extract\_secret(stego, cover, total\_bits, k=4):

    h, w = stego.shape

    recovered\_bits = []

    count = 0

    for y in range(0, h - 2, 4):

        for x in range(0, w - 2, 4):

            if count >= total\_bits:

                break

            positions = [(y, x + 1), (y + 1, x), (y + 1, x + 2), (y + 2, x + 1)]

            codes = [int(stego[pos]) - int(cover[pos]) for pos in positions]

            Pc = int(cover[y + 2, x + 2])

            Pc\_ = int(stego[y + 2, x + 2])

            I = Pc\_ - Pc + 2\*\*k

            I = max(0, min(15, I))

            index\_bin = format(I, f"0{4}b")[-4:]

            indexes = list(map(int, index\_bin))

            for idx, code in zip(indexes, codes):

                if count >= total\_bits:

                    break

                s = get\_symbol\_from\_code(idx, code, k)

                s = max(0, min(2\*\*k - 1, s))

                recovered\_bits.append(format(s, f"0{k}b"))

                count += k

    return "".join(recovered\_bits)[:total\_bits]