Steganography: hiding information in images

Background

"Steganography is the art and science of writing hidden messages in such a way that no one apart from the intended recipient knows of the existence of the message; this is in contrast to cryptography, where the existence of the message itself is not disguised, but the content is obscured."

In this practical exercise we will examine the use of steganography as an application of digital image processing. By hiding information within digital images we can both disguise the existence of a secret message and to some extent obscure its content.

The principle of steganography in digital images relates to two keys facts: 1) images are generally very large in relation to a message we may send (i.e. image bytes = Row * Column * #Channels), 2) changes can be introduced into the lower order bits of an image without any perceived visual effect to the viewer. We have seen this effect with our examination of human perception of mild changes in image gray levels of colours (IP notes).

Image based steganography, coupled with strong data encryption is now a widely accepted form of secure, disguised digital communication – it is reputedly used by both government intelligence services and terrorists alike. It is also a widely used method of digital image watermarking - prove image ownership and origin.

Here we will examine and code some simple image based steganographic approaches. You are expected to implement the specified tasks as C/C++ programs using the OpenCV library. *Use PNG format images.*

Part A: Simple Steganography (10%)

\+

=

Grayscale Carrier Binary Image Encoded Image

Our first steganographic method involves the simple addition of a binary image to a secondary carrier image in order to conceal the bit-mapped text message contained in the binary image. As the binary image can be added as small 0 or 1 differences to the original, the encoded message is concealed from perception. If the sender and receiver of a message keep the original carrier image secret, no-one can easily recover the encoded message.

Task:

- Implement an encoder program that takes as input an 8-bit grayscale carrier image and a 8-bit binary message image and produces an encoded image with the message added to the lower bits of the carrier. Add binary 0 (black) as +1 and 255 (white) as +0 to the encoded image to minimise changes in the encoded image (i.e. add inverted binary image). Assume that both the carrier and message images are the same dimension. Specify the image file names as command line options. *You will need to handle integer overflow should it occur – the best option is simply to prevent addition occurring where overflow will occur.*
- Implement the reverse decoder program to recover the original text from an encoded image produced with your encoder. Output the encoded message as a 8-bit binary image with values 255 for background (white) and 0 for text (black). *Test images for this part of the practical are available from Blackboard.*
- Use your decoder program to the message in the following examples from the above space:

- Carrier: g*rayscale_carrier.png*
- Message: *message.png*
- Encoded image: *testing.png*

Submit : C/C++ source code for parts the decoder and encoder, a working executable for each, an example encoded/decoded image from the test examples provided and the message from the requested image decoding as a comment in your decoder source file.

Part B: Scrambling the Signal (20%)

The main weakest with the steganographic approach from part A is that anyone who obtains the original carrier image can access your secret message using basic image processing techniques. The message contained in the image is not obscured in any way. In order to overcome this weakness we are going to use a seeded random number generator to generate a random ordering of pixel locations in which to encode the message. By using the same random seed (i.e. random number generator initialiser) we will be able to recover the same random order of pixels when decoding the message – the random seed thus becomes a sort of password to our message.

A suitable random number generator is provided in OpenCV, which is capable of generating random integer from a given seed. Details and examples are available in the OpenCV manual (see class "RNG"). Use this seeded random number generator to generate the random order of pixel locations used for encoding / decoding.

Seeding *the random number generator*

As you will note from the OpenCV manual, the provided random number generator can be seeded with a 64-bit integer. To ensure we get the same unique sequence of pseudo-random numbers to encode and decode our message we will seed the generator with our password. However, user passwords are commonly (and for convenience) generally ASCII strings (i.e. text) rather than 64-bit integers! To allow us to seed the random number generator with an ASCII string password we will make use of a hash function to transform the ASCII string to a 64-bit integer.

The hash function we will use here will be the *djb2* function available from the following URL: *http://www.cse.yorku.ca/~oz/hash.html*. You may use the source code from this webpage directly (with acknowledgement) but beware of ensuring data type consistency.

So, in summary, to seed the random number generator you will need to:

- Prompt the user for a character string password.
- Pass this character string to your *djb2* hash function implementation.
- Use the resulting integer as the random number generator seed.

We are now ready to generate a random order of pixel locations.

Generating *the random order of locations*

As in this part of the practical all the possible locations in the image are being used (to either store a 0 or 1 message image bit) the most efficient way to generate the randomised order of locations is to generate a array structure of all possible locations, (1...R*C), and then iterate over it randomly swapping each location with another indexed by a randomly generated integer (from the random number generator). You may find a 2D point datatype (such as OpenCV's Point2i) useful in this task.

To encode you simple iterate over the message image (starting top, left) and encode the (*i,j)th* bit in the binary message image at the *n*th randomised array location in the carrier image. Incrementing *i,j* and *n* after every message pixel is encoded.

Consistency is important: Ensure consistent use of the random number generator in ordering the co-ordinate array or otherwise your encoder/decoder will not be compatible.

Task:

- Implement encoder *and decoder* programs than operate based on the specification described here in Part B. Use the same 8-bit image inputs/outputs and inverse encoding method as used for Part A. Assume that both the carrier and message images are the same dimension. *Again, you will need to*

handle integer overflow should it occur. Specify the image file names as command line options (e.g. programname carrier message encoded).
Test than your decoder correctly decodes an encoded image that your encoder produces.
Test images for this part of the practical are available from the same URL as for Part A.

- Use your decoder from Part A to visualise the random distribution of message bits in the encoded image (i.e. try and decode an image produced with the Part B encoder with the Part A decoder – you should be able to see no structure of the original message in the decoded result!).

Submit : C/C++ source code for parts the decoder and encoder, a working executable for each, an example encoded/decoded image from the test examples provided together with original carrier, message, encoded image and a copy of the encoded image visualised with your Part A decoder.

Part C: Generating Noise Images (10%)

The problem with our approach for part A/B is still that anyone with the original carrier image can detect the presence of an encoded message even if they cannot read it. This is not very well disguised. Instead of using images of scenes or objects we can generate random noise images that are unique to every communicating pair (sender/receiver).

However, pure noise images are easy to identify from regular image data and are thus easily highlighted as possibly containing hidden information (for instance on the web) – instead we add a specific noise distribution to existing image content (of a scene or similar). This then becomes our original, and unique, carrier image. Any interception of the image, even with knowledge of the original, leaves the interceptor unable to distinguish between the noise added to the original noiseless image and the encoded message information.

Task:

- Using your knowledge of image noise distributions from the course notes & the OpenCV manual section "Random Number Generation – search for class RNG" (dealt with in Part B) write a program that outputs a version of a given specified input image with noise added according to a given distribution. Specify the image file names as command line options *(programname carrier output)*.

  You will need to determine a suitable noise distribution for this task.

- Integrate your password-based random number seeding approach from Part B to allow a sender/receiver pair to generate such common noise distribution on any available image from a common password input.

Submit: C/C++ source code for your noisy image generation program, a working executable and an example images with and without noise added by your program.

Part D: Extending to Colour Images (25%)

Part B only still only works for grayscale carrier images – let's add a little colour!

In order to extend our approach to work with colour carrier images we will distribute the encoded message bits over the three available colour image channels – *this further obscures the hidden content*. As we now have redundancy in our carrier image, as we have (C*R*3) slots to put (C*R) bits from our binary message image we can handle potential integer overflow in a more robust way. We do this by randomly generating each destination slot in the carrier image "on the fly" as we encode the image instead of pre-computing them as a randomly shuffled list. However, we must ensure that we use a slot only once *(or decoding ambiguity will arise)* and that overflow is avoiding by "skipping" slots where integer overflow may occur.

Avoiding overflow / using image slots only once

A suitable method of preventing overflow can be implemented using the following approach:

- For each location *(i,j)* in the message image
  - use the password seeded random number generator to select a random location *(c,r)* in the carrier image (generate *c*, then *r – consistency in this generation is essential*)
  - use the same random number generator to select a random channel *i* as an integer 1-3

- while either this location *(c,r,i)* has already been used for encoding or encoding at this location will result in overflow (i.e. current image value is already 255)
  - regenerate another *(c,r,i)* location until a suitable one is found
- encode message image bit *(i,j)* at location *(c,r,i)*

So that the state of each location *(c,r,i)* – used or unused – can be determined this needs to be recorded during the encoding operation. A suitable structure for doing this is a separate image of dimension *(C*R*3).* A similar structure will be required for decoding so that re-generated locations can be similarly skipped during decoding.

Task:

- Implement encoder *and decoder* programs than operate based on the specification described in Part D. Use the same 8-bit image message input type and inverse encoding method as used for Part A but use an 8-bit RGB colour image as the carrier image (*colour_carrier.png*) and for output. Assume that both the carrier and message images are the same dimension. Specify the image file names as command line options *(programname carrier message encoded)*.
  Test than your decoder correctly decodes an encoded image that your encoder produces.
  Test images for this part of the practical are available from the same URL given in Part A.
- Integrate your password-based noise generator from Part C such that the before a message is encoded a unique random noise distribution is added to the image over all three of the colour channels. The decoder must now also add this random noise distribution to the carrier image before decoding the message as before.
  Test that this integration does not effect the operation of your message encoding/decoding.

Submit: C/C++ source code for the decoder and encoder, a working executable for each and an example encoded image from the test examples provided together with original carrier and message images.

Part E: General Information Hiding (25%)

In Part's C and D we learnt how to encode binary image bits (0 or 1) in an image. As a result we can now hide a 0/1 binary image in another image. However, all digital information (e.g. all computer files!) can be represented as a binary string of 0 and 1 too. Here we'll extend our steganography to handle arbitrary binary information from any given file type.

In order to do this we will read in an arbitrary type message file specified on the command line with the carrier and destination encoded image file names. The contents of the message file can then be treated abstractly – i.e. we don't care what the data is, we just treat it as a sequence of bytes. Each of these bytes can be split into its bit level contents by arithmetically accessing individual bits in a byte. These bits can then be encoded, as per the binary image bits of Part D, using our previous steganography technique.

A basic example of bit level access in the C/C++ language is available from Blackboard:

E_bit.c (you may be able to find an alternative, more elegant method – if so use this)

Limit on message content

When encoding arbitrary files we cannot be sure that the message size (in bits) will fit into the number of available encoding slots in the image (C*R*#channels).

- An 8-bit colour image of size (C*R*#channels) has similarly (C*R*#channels) bit slots. It can thus hold ((C*R*3) / 8) bytes of information. *For example an 8-bit 640x480, 3 channel colour image can hold 115200 bytes (112.5 Kbytes) of encoded information (assuming no overflow problems).*
- A message of length N bytes requires 8N slots to encode the message in a carrier image.
- A message of length K Kbytes requires 1024K*(8N) slots to encode the message in a carrier image.

Task:

- Implement encoder *and decoder* programs than operate using the steganography technique for colour images described in Part D but instead encode an arbitrary type data file as described in the specification of Part E (above). You will need to:

- ensure the user specified carrier image has enough (non-overflow) slots to encode the message.
- make a suitable design decision on how to generate and randomise the sequence of slots used for encoding in the image. You encoder/decoder will need to be consistent here but general consistency with other class members is not required.
- design a method of encoding the length of the file being encoded into the message so that, when decoding, you will know when you have reached the end of the file.

Use the image types as specified for Part D. Specify the file names as command line options *(programname carrier message encoded).*
Test than your decoder correctly decodes an encoded image that your encoder produces.
Test images/files for this part of the practical are available from the same URL given in Part A.

Submit: C/C++ source code for the decoder and encoder, working executables for each, an example encoded image from the test examples provided together with original carrier and message files.

Deliverables / Mark Summary:

You are required to deliver the following components of this practical:

- *Part A (10%)*
  - *Encoder C/C++ source code / Decoder C/C++ source code.*
  - *Encoder / Decoder executables.*
  - *Example encoded and resulting decoded images with original carrier and message images.*
  - *Decoded message from stated URL as comments in decoder source file.*
- *Part B (20%)*
  - *Encoder C/C++ source code / Decoder C/C++ source code.*
  - *Encoder / Decoder executables.*
  - *Example encoded and resulting decoded images with original carrier and message images.*
  - *A copy of the encoded image visualised with your Part A decoder.*
- *Part C (10%)*
  - *C/C++ source code for your program.*
  - *A working executable.*
  - *An example images with and without the noise added by your program.*
- *Part D (25%)*
  - *Encoder C/C++ source code / Decoder C/C++ source code.*
  - *Encoder / Decoder executables.*
  - *Example encoded and resulting decoded images with original carrier and message images.*
- *Part E (25%)*
  - *Encoder C/C++ source code / Decoder C/C++ source code*
  - *Encoder / Decoder executables*
  - *Example encoded and resulting decoded images with original carrier, message file.*

● ***Overall code quality, clarity, documentation, style and design (10%)***

Total = 100%

Submission:

Make it clear in the initial comments of your source code how to run your executable. Your executable must run on one of the lab based PCs – *ensure compatibility before submission.* *All submitted images must be in PNG format.*

Image Processing and Analysis: Assessed Practical Exercise – Steganography (2019-2020)

Plagiarism: You must not plagiarise your work. *You may use program source code from the provided examples, the OpenCV library itself or any other source BUT this usage must be acknowledged in the comments of your submitted file. Automated software tools (e.g. http://theory.stanford.edu/~aiken/moss/) will be used to detect cases of source code plagiarism in this practical exercise which will include automatic comparison against code from previous year groups. Attempts to hide plagiarism by simply changing comments/variable/function names will be detected (and has been in the past).*

You should have been made aware of the Cranfield University policy on plagiarism. Anyone unclear on this must consult the course lecturer prior to submission of this practical.

University Plagiarism Guidance:

http://www.cranfield.ac.uk/library/cranfield/support/page41148.html

To submit your work, create a directory named by your capitalised Firstname and Surname (e.g. *KermitFrog*).

Inside create separate directories for each part of the practical :- PartA ... PartE. Place all required files in the corresponding directory. Do not submit entire Visual Studio project directories – just the required files.

Submission Deadline (ALL): 20th Jan 2020

(Late submissions will be penalised)